Dynamic Homology and Phylogenetic Systematics: A Unified Approach Using POY

Ward Wheeler
Division of Invertebrate Zoology
American Museum of Natural History

Lone Aagesen
Division of Invertebrate Zoology
American Museum of Natural History

Claudia P. Arango
Division of Invertebrate Zoology
American Museum of Natural History

Julián Faivovich
Division of Vertebrate Zoology
American Museum of Natural History

Taran Grant
Division of Vertebrate Zoology
American Museum of Natural History

Cyrille D'Haese
Département Systématique et Evolution
Museum National d'Histoire Naturelle

Daniel Janies
Department of Biomedical Informatics
The Ohio State University

William Leo Smith
Division of Vertebrate Zoology
American Museum of Natural History

Andrés Varón
Department of Computer Science
City University of New York

Gonzalo Giribet
Museum of Comparative Zoology
Department of Organismic & Evolutionary Biology
Harvard University

Published in cooperation with NASA Fundamental Space Biology, the U.S. Army Research Laboratory, and the U.S. Army Research Office

Printed in the United States of America

Library of Congress Cataloging-in-Publication Data

Dynamic homology and phylogenetic systematics : a unified approach
using POY / Ward Wheeler ... [et al.].
p. cm.
"Published in cooperation with NASA Fundamental Space Biology."
Includes bibliographical references.
ISBN 0-913424-58-7 (alk. paper)
1. Homology (Biology) 2. Animals--Classification. 3. Phylogeny. 4. POY. I. Wheeler, Ward.
QH367.5.D96 2006
578.01'2--dc22

2006028134

Contents

An Approach to Phylogenetics

Phylogenetic systematics demands a general and logically consistent analytical framework. To meet that demand, we propose here a system of testing phylogenetic hierarchies with the broadest possible diversity of evidence.

Traditionally, phylogenetic analyses were based entirely on phenotypic evidence derived from such sources as comparative morphology, molecular biology, and ethology, which entailed only a few distinct character types. As genomic data have become more readily available, however, the diversity of character types has increased dramatically to include complex genetic, chromosomal, and even entire genome sequences. Fundamentally, there is no evidentiary distinction to be made among these sources of information. Logical consistency requires that all evidence be treated equivalently. At present, there is no way to test phylogenetic hypotheses with the full diversity of available evidence in a consistent manner, a shortcoming we aim to correct here.

The incorporation of genotypic character types into systematics requires a fundamental rethinking of the way phylogenetic hypotheses are tested and inferences made. A central distinction in this book is made between the static homology approach, where data are encoded (or aligned) in a fixed character matrix prior to phylogenetic analysis, and the dynamic homology approach, where constraints on possible transformations are

reduced and characters (transformation series) are inferred *a posteriori* as a result of phylogenetic analysis.

The ideas in this book have been implemented in the computer program POY, version 3.012, written by Wheeler and colleagues (Wheeler et al. 1996–2003). POY is an open source (`ftp://ftp.amnh.org/pub/molecular/poy`) program written in ML and C. Precompiled binaries are available for Linux x86, MS-Windows, and Mac OS X. The code is fully parallelized (using PVM and MPI) and fault tolerant with multiple program options and algorithms to take advantage of a diversity of multiprocessor platforms, including clusters and shared memory multiprocessor (SMP) machines.

In the first part of this book, we discuss the theoretical core of phylogenetic systematics as it relates to POY. Given a minimal set of evolutionary assumptions, a definable (if large) collection of possible hypotheses is tested with evidence derived from observed biological variation. We are not attempting to be exhaustive in our discussions, but to focus on the issues relevant to POY and its motivation. Readers should go to more general sources (e.g. Schuh 2000) for broader treatments.

The second part presents the problems faced by systematic analysis and discusses their solutions in both theoretical and practical, applied terms. Exact solutions will be few and far between, given the computational complexity of the problem. We are limited, largely, to heuristic solutions that take advantage of distributed computing. We discuss the trade-offs associated with each approach, as well as the optimal search strategies given different phylogenetic problems and computer resources.

The third part is a users' guide for the implementation of these ideas in the program POY, including program installation instructions for multiple platforms, descriptions of over 250 user-specified commands, and instructions for use.

Finally, as may be expected in any field of science confronted with new problems of unanticipated complexity, a diversity of possible solutions exists, and it is only by engaging in open scientific debate that false theories and logically inconsistent methods are eliminated and progress made. Given the infancy of phylogenetic analysis in the genomic era, it should come as no surprise that significant disagreements persist among the authors of this book. Although some amount of compromise is required in collaborative projects, such as this book, and it is impractical and inappropriate to debate our differences of opinion exhaustively, we believe it would be intellectually dishonest to overlook substantive disagreements altogether. The reader should look to the primary literature for more fully developed arguments. Nothing is final and irrefutable in science. We believe our open criticism to be a necessary aspect of the

scientific process that is all too often hidden from the reader in an attempt to present a unified front.

We would like to acknowledge the contributions and support of the institutions and people who made this work possible. Louise Crowley, Philippe Grancolas, Cecilia Kopuchian, Diana Lipscomb, Jyrki Muona, Kurt Pickett, Stefan Richter, Susanne Schulmeister, and George Wilson reviewed drafts of the manuscript in whole or part, offering many helpful criticisms. Elizabeth Werby did much to shepherd the production from its beginning stages and Tanya Das coordinated final production. The work was supported by the NASA Fundamental Space Biology Program under the guidance of Melvin Averner, who provided tremendous support and encouragement. The material was also based upon work supported by the U.S. Army Research Laboratory and the U.S. Research Office under grant number W911NF-05-1-0271.

Finally, Robert MacElroy, also of the NASA Fundamental Space Biology Program, was a great supporter of our research, sharing with us his warmth and intelligence. We lost Bob in 2004, and this work is a remembrance of him.

Part I
Phylogenetics

1 Data

Evidence

The fundamental task of systematics is to explain biological variation by inferring the phylogenetic relationships among organisms and the unique transformation events that link them (Hennig 1966).

Inference of particular functions, adaptations, mechanisms, and constraints, and the many other processes that shaped the evolution of each group, can be informed by the results of phylogenetic analysis. However, evolutionary analysis requires additional assumptions and tests that are external to systematics (for example, Farris 1983; Grandcolas and D'Haese 2003; Grant and Kluge 2003).

Operationally, systematics proceeds by gathering data (observations) from organisms and coding them into evidence to test competing phylogenetic scenarios.

In principle, any observation of a set of creatures has the potential to provide evidence of historical kinship. However, the most objectively critical evidence is derived from those features that are heritable and intrinsic to organisms because they reflect the biological continuity between ancestor and descendant (Hennig 1966). Differences in each of these features can be traced to specific and unique transformations on a cladogram and these transformations allow us to assay the relative

merits of alternative historical explanations. All observations are consistent with all scenarios, but not to an equal extent (see Sober 1983). It is this inequality that drives systematic analysis, forms the basis of our tests, and elevates general historical statements to testable scientific hypotheses.

A distinction is made between data and evidence. Evidence implies an organized set of observations that can be used to test hypotheses. A collection of nucleic acid sequences or statements about an anatomical feature (for example, biramous appendage) have little meaning or value in and of themselves. However, when organized into putatively homologous features (for example, 18S rRNA or abdominal appendages) these observations (data) demand analysis. Coding transforms observation into evidence (characters) and allows hypotheses to be weighed quantitatively.

Value of Evidence

At least initially, we must operate under the principle that all evidence is possessed of equal value in discriminating among phylogenetic scenarios. That is, information from all sources and collected by any means may have value in testing historical hypotheses: the total evidence principle (Kluge 1989). This is not to say that all information is equally discriminating, but this cannot be known prior to systematic analysis.

A corollary of this notion is that phylogenetic characters (that is, coded observations) do not form logical classes on the basis of their ability to differentiate among hypotheses. They can be divided and sorted into all variety of functional, structural, or observational classes, but these have no bearing on the evidentiary content of the characters themselves. This leads to a notion of complete catholicism with respect to sources and types of data—provided that they derive from independently heritable transformations. Anatomical, behavioral, and genomic features are all, on the face of them, informative. There is no reason to segregate character variants into classes or partitions. There are only those features that can objectively distinguish between hypotheses and those that cannot.

Sources of Evidence

Traditionally, the data used in phylogenetic analysis have been categorized as either morphological or molecular. Morphological data have originated principally from studies in comparative morphology and ethology (although the latter are not strictly morphological). Molecular data have originated from studies in molecular biology. According to this categorization, morphological data consist of anatomical features

and can include behavior, while molecular data consists of nucleic acids, proteins, and genomes.

This distinction has given rise to a number of controversies regarding the suitability of each kind of data for phylogenetic analysis. In addition, each type of data is regarded as requiring different analytical treatment. This difference in treatment principally concerns how comparable characters are identified. On the one hand, morphological characters have been treated as a matter of observation, while molecular characters must be inferred. Phylogenetic analysis is better served by distinguishing between phenotypic and genotypic data.

Phenotypic and genotypic data

Phenotypic data result from the structural and functional characteristics that express an organism's genotype along with the organism's response to its environment. Phenotypic data therefore include morphology and behavior as well as many molecular characters, such as amino acid sequences or pheromone profiles. Data collected on phenotypes are derived from structurally and developmentally complex features, which enables testing of each hypothesis of homology in isolation.

However, observed variation may be the result of either heritable transformations or environmental effects. Failure to distinguish between the two causes of variation may confound attempts to infer phylogenetic relationships.

On the other hand, genotypic evidence represents an organism's genetic material and is therefore directly and entirely transmitted from parent to offspring, thus eliminating any concern for the heritability of observed variation. Although this is a clear strength of genotypic data, the peculiarities of these features give rise to novel analytical problems. All instances of each of the four possible nucleotides are physically indistinguishable, regardless of their historical origins: any nucleotide can substitute directly for any other (there are no intermediate states) and any nucleotide can be inserted or deleted. It is therefore impossible to test hypotheses of nucleotide homology in isolation. Only the test of character congruence can be applied to them.

Until the time when whole genomes are available and can be analyzed appropriately and the genetic basis for particular phenotypic variants is known and can be traced to unique transformations, combined analysis of both sources of evidence provides the strongest test of phylogenetic hypotheses.

Homology

At the level of evidence, cladograms imply statements of homology. Alternative cladograms might have alternative optimal homology statements and content. At a basic level, we differentiate among cladograms on their ability to embody the potentially conflicting homology statements of diverse sets of characters.

Features are homologous when their origins can be traced to a unique transformation on the branch of a cladogram leading to their most recent common ancestor. There can be no notion of homology without reference to a cladogram (albeit implicitly) and no choice among cladograms without statements of homology.

This definition of homology makes no reference to "primary" or "secondary" homology (de Pinna 1991). In fact, the perspective here rejects this distinction entirely. De Pinna (1991: 372) based his distinction on the view that homology assessment necessarily requires hypothesis "generation and legitimation" in separate steps, the former rooted in notions of similarity and the latter based on the simultaneous test of character congruence.

All possible hypotheses of homology are defined logically as a function of the number of heritable parts identified for each terminal (just as all possible cladograms are defined logically as a function of the number of terminals; Felsenstein 1978), so no special procedure is required for hypothesis "generation." Likewise, although homology assessment often involves a two-stage procedure of first submitting each hypothesis of homology to a round of separate tests and then submitting the surviving, constrained set of hypotheses to the test of character congruence (that is, "static" homology assessment), this separation is neither a methodological nor epistemological necessity.

POY embodies the concept of dynamic homology (Wheeler 2001a, b) in which the test of character congruence is applied to the entire, unconstrained set of hypotheses of homology, thereby allowing entire transformation series to be discovered on the basis of a single optimality criterion. That is, dynamic homology employs the same procedure to discover both the character (in the traditional sense) and the character-state transformations within the character. Since the same optimality criterion is employed in both cladogram assessment and homology assessment, the globally optimal explanation of the observed variation is achieved by the minimum-cost cladogram-plus-homology-scheme combination.

In the same way that each cladogram has a (potentially) unique set of optimal character origins, each cladogram may have a unique set of optimal correspondences among observed features. Unless these correspon-

dences are unrestricted and allowed to be optimized together with transformations, biased and conditional results may be obtained. Such bias may come from the assumptions of the investigator and his or her notions of the appropriateness of comparison, and conditioned on the hypotheses most in agreement with the preconceived correspondences of "primary" homology.

Dynamic homology is a powerful conceptual approach to the study of highly simplified data types, such as DNA and amino acid sequences or simple morphological structures like annelid segments, where structural or developmental evidence that could allow a defensible choice among competing hypotheses of homology is either nonexistent or unavailable.

Suggested Reading

Farris, J. S. 1983. The logical basis of phylogenetic analysis. *In* N. I. Platnick and V. A. Funk (editors), Advances in Cladistics: 277–302. New York: Columbia University Press.

Hennig, W. 1966. Phylogenetic Systematics. Urbana: University of Illinois Press. 263 pp.

Sober, E. 1983. Parsimony methods in systematics. *In* N. I. Platnick and V. A. Funk (editors), Advances in Cladistics: 37–47. New York: Columbia University Press.

Wheeler, W. C. 2001. Homology and the optimization of DNA sequence data. Cladistics 17: S3–S11.

2 Cladograms

Cladograms, Trees, and Tree-Shaped Objects

Independent of evidence, a cladogram is a branching diagram that depicts the hypothesized phylogenetic (as opposed to ontogenetic or tokogenetic) relationships among terminal taxa. They are Steiner or Wagner trees in much of the systematics literature, meaning that observed taxa (often called operational taxonomic units or OTUs) are confined to the tips (leaves) and are not placed at inner nodes. This is preferred over Prim networks because

- it allows more parsimonious solutions by optimizing novel character combinations to inner nodes as hypothetical taxonomic units (HTUs) (Farris 1970),
- and it avoids the problematic assumption that some observed taxa are directly ancestral to other observed taxa (Platnick 1977).

Cladograms can either be undirected (unrooted) networks or directed (rooted) trees (Farris 1970). Only in the latter case can historical statements be made.

The computer science literature uses "tree" in a slightly different context. A tree is a directed, acyclic graph — in more familiar terms, a rooted branching diagram without reticulation. This literature also

uses "network" to mean an undirected cyclic graph— an unrooted branching diagram with reticulation.

In most cases here, we refer to cladograms. We are, however, drawing on a large wealth of algorithmic and tree manipulation discussion from the computational literature, so we will use these terms more broadly, to the point where they are largely interchangeable. "Network" does not refer to reticulated graphs in this book, although we use the term only rarely and usually describe cladograms as directed or undirected.

Associated with an optimized body of evidence, a directed cladogram becomes a summary statement of phylogeny and homology, representing the historical relationships among both the terminal taxa and the individual characters. Cladograms do not confer information about why or how particular transformations occurred between ancestor–descendant pairs. Also, cladograms need not contain specific information about ancestor–descendant character transformations. In many cases we can determine how costly a cladogram is without ever having to determine the precise ancestral reconstructions required by a tree— for example, by only performing down-pass optimization.

The number of possible cladograms is solely a function of the number of terminals. For n terminals, the number of binary (bifurcated, fully resolved) undirected cladograms is given by (Felsenstein 1978)

$$\frac{(2n-5)!}{2^{n-3}(n-3)!} \qquad \text{(Eq 2.1)}$$

and for directed cladograms by

$$\frac{(2n-3)!}{2^{n-2}(n-2)!} \qquad \text{(Eq 2.2)}$$

The number of possible cladograms therefore increases explosively as taxa are added to an analysis. For a given matrix of static homology statements, finding the optimal cladogram is an NP-complete problem (Garey and Johnson 1977; Garey et al. 1977). Heuristic solutions are therefore required to analyze data sets composed of more than about 20 taxa.

Choice among cladograms is mediated by optimizing the evidence on cladograms and calculating the minimum cost in terms of weighted transformations required to explain the observed variation in light of background knowledge. That is, cladograms differ in their ability to explain conflicting evidence in a single historical scenario. Optimality criteria measure this ability quantitatively and permit the relative evalua-

tion of cladograms. Cladograms are ranked and optimal solutions identified through these values.

POY is mainly concerned with parsimony as an optimality criterion, valuing, as it does, simplicity. The cladogram that minimizes transformations to explain the observed variation is the simplest, maximizes evidential congruence, and has greatest explanatory power. There is also, however, some ability to evaluate cladograms comparatively in terms of their likelihood scores, where the cladograms that maximize the likelihood are preferred.

Cladograms and trees are branching diagrams that depict the historical relationships among taxa as inferred from critical evidence and have associated optimality values. That is, they participate in hypothesis testing through the optimality values and are directly supported by evidence. Although they are most clearly exemplified by most parsimonious and maximum likelihood trees, they include the results of minimum evolution, including neighbor joining (Saitou and Nei 1987) and minimum percent standard deviation (Fitch and Margoliash 1967). In each of these examples, there is a direct and unequivocal relationship between a body of evidence and the tree or cladogram.

These are contrasted by what we refer to as tree-shaped objects, which are branching diagrams that look like and are often interpreted as if they were cladograms or trees, but have altogether different purposes and empirical bases. These primarily include diagrams meant to summarize the results of explicit phylogenetic analysis, most notably the strict consensus, MrBayes trees (Huelsenbeck and Ronquist 2003), parsimony jackknife trees (Farris et al. 1996), and supertrees (for example, Binindа-Emonds et al. 2002). The strict consensus representation is objectively interpretable as a summary of the clades that are unambiguously supported by the evidence, and as such is a valuable tool in systematics. MrBayes and parsimony jackknife trees are majority rule summaries of the frequency of clades in a sample of trees. Supertrees are intended to depict the results of analyses involving overlapping sets of terminals.

None of these different kinds of summaries maximizes an underlying optimality criterion. It is widely recognized that the strict consensus representation is a suboptimal explanation of the evidence—that is, it requires more steps than any one of the fundamental cladograms (for example, Kluge 1989), that supported groups can have a lower resampling frequency than unsupported groups (Goloboff et al. 2003a), and that these depictions should not be interpreted as cladograms or trees. However, it is frequently overlooked that the MrBayes solution suffers from the same analytical problem as the jackknife. That is, the Monte Carlo Markov Chain algorithm generates a sample of cladograms and is meant to estimate the posterior probabilities of clades, not trees. As a majority-rule representation, the MrBayes topology may not be the

maximum posterior probability cladogram, and the groups it shows as being recovered in high frequency can actually be unsupported by the Bayesian optimality criterion.

With supertrees, the gulf between evidence and summary depictions increases even more. Because they are intended to summarize the results of different analyses, it is possible for the source trees to have been inferred under contradictory optimality criteria—indeed, this is actually seen by some authors as a strength of supertrees. Of course, that alone is not necessarily an inappropriate procedure. However, at least some proponents of supertrees encourage their interpretation as actual hypotheses of phylogeny— that is, as cladograms or trees equipped with optimality values, branch lengths, and support metrics adequate to test competing evolutionary scenarios (for example, Bininda-Emonds et al. 2002). Such interpretations are indefensible, as no scientific test is possible in the absence of a clear link to empirical evidence— that is, an evidence-based optimality criterion. The perceived need for supertree methods is that

- the amount of systematic data available outstrips present computational abilities
- the obstacles to combining evidence from different sources cannot be overcome.

A recurring theme in this book is that data set size affects only the exhaustiveness of analysis, not the ability to analyze it in a logically consistent manner. Likewise, all veritable evidence can be combined in simultaneous (that is, supermatrix) analysis.

Terms and Notation

Navigating efficiently through descriptions of cladograms and their manipulations requires terminology. As with the cladogram/tree discussion, there is a rich lexicon of descriptive terms, illustrated in Figure 2.1 on page 17.

OTU (operational taxonomic unit) Taxon selected for phylogenetic study, generally based on observed or inferred characteristics; terminal taxon; leaf.

HTU (hypothetical taxonomic unit) The inferred internal vertices or nodes of a cladogram. These are often interpreted as ancestors, but are, in reality, abstractions whose features are constructed to maximize the optimality criterion.

Node A vertex on a cladogram. All HTUs and OTUs are nodes.

Branch The path connecting two nodes; edge.

Root The basalmost, ur-node of a cladogram.

Binary Describes a fully dichotomous, resolved node or cladogram.

Polytomy An internal node that is not binary.

Cost The value of the optimality criterion of a cladogram. This equals the number of steps (transformations, changes) or length in equally weighted parsimony analysis or weighted sum of transformations more generally. In maximum likelihood analysis, cladogram cost is reported as the absolute value of the log likelihood score.

Synapomorphy A transformation on a branch that leads to an HTU.

Autapomorphy A transformation on a branch that leads to an OTU.

Homoplasy Nonminimal transformations of a character on a cladogram.

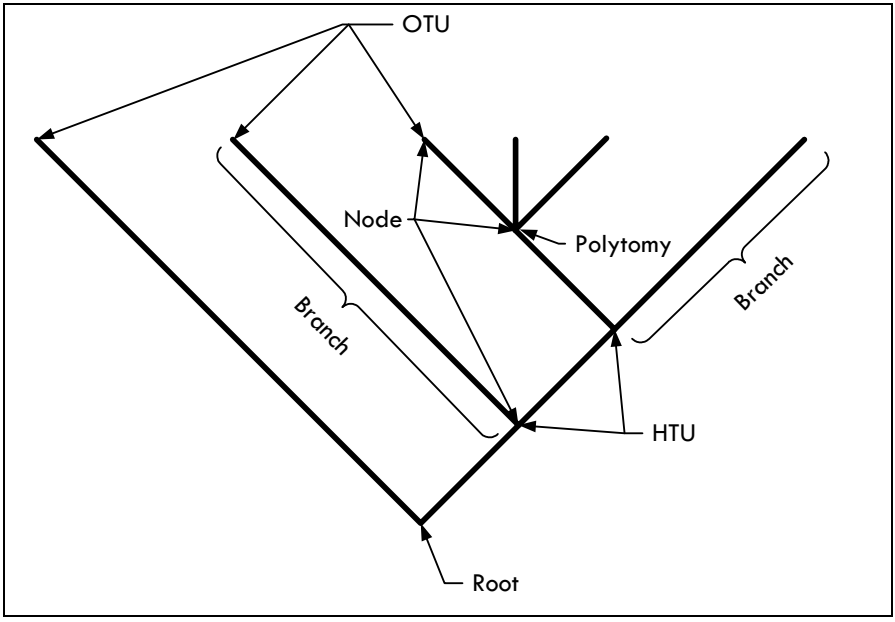

Figure 2.1: Cladogram terms and notation.

Suggested Reading

Farris, J. S., A. G. Kluge, and M. J. Eckhardt. 1970. A numerical approach to phylogenetic systematics. Systematic Zoology 19: 172–189.

Felsenstein, J. 1978. The number of evolutionary trees. Systematic Zoology 27: 27–33.

Goloboff, P. A. 2005. Minority rule supertrees? MRP, compatibility, and minimum flip may display the least frequent groups. Cladistics 21: 282–294.

Platnick, N. I. 1977. Cladograms, phylogenetic trees, and hypothesis testing. Systematic Zoology 26: 438–442.

Prendini, L. 2001. Species or supraspecific taxa as terminals in cladistic analysis? Groundplans versus exemplars revisited. Systematic Biology 50: 290–300.

3 Analysis

Cladograms and Data

Cladograms explain and summarize character state transformations. Coded data are fitted to a scenario in order to optimize a specified criterion. Desirable or "best" cladograms are chosen by this same criterion through relative evaluation of alternative scenarios. In POY, this criterion is usually simplicity or parsimony. Other criteria exist, such as likelihood, and most of the issues discussed here apply equally well to these other optimality criteria.

Systematic analysis is dominated by two problems. First, how costly is a specific cladogram? And second, which cladogram best optimizes a given cost function? The first topic is referred to most frequently as character optimization, while the second is referred to as cladogram or tree search.

In systematics, optimization refers to the fitting of characters to cladograms on the basis of an objective criterion. Cladistic parsimony optimization involves minimizing the number of *ad hoc* hypotheses on a cladogram (Farris 1983). Character optimization forms the foundation for much of evolutionary biological studies, and the cladograms upon which they are optimized is obviously central.

These operations are the intellectual and operational basis of all areas of historical biology.

Observations are coded into evidence in a myriad of ways. Each character coding scheme relies on distinct assumptions that result in different quantitative optimizations of a given character based on the optimality criterion and the cladogram. These choices may affect not only the type and number of transformations on a specific cladogram, but also which cladogram is deemed best or optimal. POY implements a variety of character type or coding optimizations, striving for maximum interpretive flexibility.

Implications of Coding Methods

As discussed above, observations themselves cannot be used to test hypotheses. An intermediate, coding step is required. Since the choice of a coding method imposes rules and behaviors on the data, they have a huge impact on analysis.

POY implements a variety of coding methods and techniques of analysis. Currently, there are five general data codings:

- Additive (Farris 1970)
- Nonadditive (Fitch 1971)
- Matrix or Sankoff (Sankoff and Rousseau 1975)
- Sequence (dynamic homology; Wheeler 2001a, b)
- Chromosomal (Wheeler, submitted).

The last two require heuristic solutions, a number of which are implemented in POY. The specifics of their analysis are described in Chapter 5 Character Optimization.

Additive

First described for binary characters, Farris (1970; see also Kluge and Farris 1969) defined the procedure for optimizing qualitative characters (0, 1, 2, et seq.) where the cost of transformation between states, expressed as steps, is the additive distance between them. Hence, a transformation between states 1 and 4 would have a cost of 3. One implication of this form of coding is that each state subsumes the previous. The states form a linear transformation series where each state logically contains all the previous states and homologies. For example,

Code	Description
0	Absence of antennae
1	Presence of simple antennae
2	Presence of antennae with bristles

State 2 (antennae with bristles) implies that there must be antennae in the first place in order to have these bristles. Due to the nested aspect of the states in this character type, states (other than the most derived, which is always evidence for monophyly) can imply paraphyletic groups.

This form of character is most frequently used for qualitative anatomical and other nonsequence data.

Nonadditive

Fitch (1971) described a procedure to optimize nucleotide sequence characters that made no assumptions about the ordering of states ("unordered"). This procedure applies equal costs to all possible transformations, irrespective of their distance or difference. All states may transform into each other and no nested homology relationships among states are implied. As such, each shared state is evidence of a monophyletic group.

Although this form of character was originally described for nucleotide and protein sequence data, it is used for a broad variety of qualitative data.

Matrix or Sankoff

Sankoff and Rousseau (1975) used dynamic programming to describe the general case of multistate qualitative characters. This technique enables the investigator to assign arbitrary transformation (that is, edit) costs between states. The method is a more general approach and can express additive and nonadditive characters as special cases. For the case above,

Additive

State	0	1	2
0	-	1	2
1	1	-	1
2	2	1	-

Non-additive

State	0	1	2
0	-	1	1
1	1	-	1
2	1	1	-

Arbitrary

State	0	1	2
0	-	1	4
1	1	-	3
2	4	3	-

These matrix characters are most often used in analyses of prealigned molecular sequence data. Transition–transversion cost ratios and indel or gap costs can be specified as a 5×5 matrix of transformation costs. The following matrix has a transition cost of 1, a transversion cost of 2, and an indel cost of 4.

	A	C	G	T	-
A	-	2	1	2	4
C	2	-	2	1	4
G	1	2	-	2	4
T	2	1	2	-	4
-	4	4	4	4	-

Although, in principle, these cost matrices are not limited by metricity (triangle inequality, including symmetry), significant issues are created by nonmetric transformation matrices (Wheeler 1993). These are encountered most commonly in situations where investigators treat indels (that is, gaps) as missing data, since indel transformations are given zero cost.

Sequence

These characters consist of unaligned molecular sequence data, usually nucleic acids. Given the lack of an *a priori* homology scheme, not only do the transformations between nucleotide states vary with the topology on which they are optimized, but the correspondences (homologies) do as well. The task of optimizing these characters, assigning dynamic homology is doubly difficult.

The correspondences among homologues must be determined and evaluated simultaneously with transformations and usually requires heuristic solutions. Traditionally, this type of information has been subjected to multiple alignment prior to coding and analysis by nonadditive or matrix optimization that avoid computational complexities in optimization. Optimization heuristics were proposed by Wheeler (1996), but discussion of these features in the multiple alignment context go back to Sankoff (1975).

Like cladogram search, the solution to the general problem of determining HTU sequences on a given cladogram is NP-complete (refer to "P, NP, and NPC" on page 24). POY offers four heuristic procedures to solve this problem:

- Direct optimization (Wheeler 1996)
- Iterative-pass optimization (Wheeler 2003b)
- Fixed-states optimization (Wheeler 1999a)
- Search-based optimization (Wheeler 2003c).

Although search-based optimization can, in principle, guarantee an exact solution (if through brute force), this is unlikely to be practical for real data.

The four methods can be divided into two general approaches:

- Direct optimization (DO) and iterative-pass optimization (IP) strive to construct HTU sequences such that the overall cladogram is of minimal length. This is done through modified two- and three-dimensional string matching, respectively.
- Fixed-states optimization (FSO) and search-based optimization (SBO) draw optimal HTU sequences from a pool of predetermined sequences. This can be a small (FSO) or large (SBO) collection of candidate sequences. Dynamic programming (as in matrix characters above) is used to identify the best HTU sequences and determine cladogram cost.

Chromosomal

The chromosomal character type optimizes nucleotide and locus-level variation simultaneously. This is accomplished without genomic annotation (basically, locus-level alignment homology statements) and in a topology-specific manner.

Chromosomes can be thought of as linear or circular strings of sequence characters. In addition to the transformation events optimized under sequence character coding (nucleotide insertion, deletion, and substitution), a chromosome can suffer sequence character origin, loss, and rearrangement (movement from one place to another). The chromosomal character type extends the dynamic homology framework to "loci" or the homologies among the sequence characters themselves. As such, it adds yet another level of complexity and heurism to analysis. POY currently implements this optimization procedure using breakpoint analysis, chromosomal direct optimization, and a fixed-states approach.

Currently, this character type is most used for analysis of complete mitochondrial, bacterial, and viral genomes.

Computational Issues

The two fundamental problems mentioned above, cladogram cost and cladogram search, each present complexities. This situation is made more difficult by the fact that they occur as a nested pair. Determining the cost of a given cladogram is repeated over all cladograms during the search process to yield the optimal or best scenario.

Three problems addressed by POY are members of a set of classically difficult computational challenges. The calculation of the cost of a given cladogram for unaligned sequence data, the calculation of transformation costs between chromosomal character states, and the search for the optimal cladogram are all NP-complete.

P, NP, and NPC

Computational problems are classified according to the time complexity of their algorithmic solutions. Put simply, problems are "easy" or "hard."

Easy problems are those for which there exists a polynomial time algorithmic solution (P). By this, we mean that there exists an algorithm that can solve the problem in time $O(n^k)$ for some constant k.

Hard problems (NP or nondeterministic polynomial time) require super-polynomial time to solve, but if given a solution, the solution can be verified in polynomial time. NP-complete problems (NPC) exist in a nether world where no known polynomial time solutions exist, but there is no proof of their nonexistence either (Figure 3.1). These problems are frequently combinatorially explosive, with solution spaces increasing at a factorial pace. Furthermore, there are a large number of NPC problems (for example, traveling salesman, circuit design, scheduling), and if a polynomial time solution were found for any of them, it would apply to all (Cormen et al. 2001).

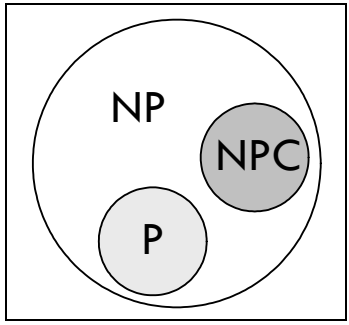

Figure 3.1: A vision of the relationship among P, NP, and NPC held by most computer scientists.

NPC problems are hard in the computational sense. Unfortunately, we are saddled with them. This has two practical implications for systematic research. The first is that we will almost never be dealing with exact solutions to any of our problems. Heuristics will drive our analysis. As heuristic procedures improve, so will our analyses. The second is the employment of parallel computation to analyze large and complex data sets.

Techniques

Exact methods

Although the general problems we discussed here have no prospect of an exact solution, certain, likely small cases can be tractable. For these situations, it may well be possible to explicitly enumerate all possible solutions. For eight terminal taxa, there are only 10,395 possible unrooted cladograms. Searching through such a space for an optimal solution would not be terribly taxing. Analogously, a linear chromosome with eight gene regions would allow 40,320 possible orderings of loci. Certainly a large number, but not impossibly so.

Such brute force methods rapidly become unworkable as the size of the problem increases. Two techniques in general use, however, enable the analysis of somewhat larger data sets. Dynamic programming (used in character optimization) and branch-and-bound techniques (used in cladogram search) implicitly enumerate and examine all possible solutions without actually visiting each scenario. Both methods evaluate partial, subsolutions that allow the algorithms to exclude large areas of unprofitable search space, making exact solutions tractable for larger problems. These eventually reach their limit and maintain utility predominantly in small analytical studies as opposed to those with the hundreds to thousands of taxa commonly studied in real data sets.

Heuristic methods

Approximate or heuristic solutions provide systematics with its usable results. Many of the heuristic approaches taken in cladogram search are well known: branch swapping, random addition of taxa, simulated annealing in the form of the ratchet and tree drifting, and genetical algorithms as in tree fusing. Each of these techniques aids in the searching of cladogram space and attempts to move solutions from locally to more globally optimal solutions.

Cladogram cost heuristics are of two types:

- First, there are techniques like direct optimization and iterative-pass optimization that break the problem down into a series of two (direct optimization) or three (iterative-pass optimization) node comparisons, calculating locally optimal solutions, which generate total cladogram costs.
- A second approach is embodied by fixed-states and search-based optimization that reduce the world of possible HTU sequence solutions to tractable, but informative sets. These sequence solutions are then applied using dynamic programming to calculate the overall cladogram cost.

Parallel Computation

Given the use of randomization, and the huge size of cladogram space, the cladogram search problem is often labeled "embarrassingly parallel." By this is meant that the problem is ripe for analysis by distributed, concurrent computation. Integrating heuristic solutions efficiently in a parallel environment is often not quite so trivial. Significant effort is employed to make use of multiprocessor systems for systematic analysis. Parallel strategies implemented in POY are discussed in Chapter 8 Parallel Processing.

Integration of Heuristic Analysis

Systematics has become a highly technical field in data collection, analysis, and interpretation. Many decisions are required to code data and appropriately apply this evidence to evaluate cladograms. Although fundamentally a series of transitive pairwise hypothesis tests, the evaluation process integrates to complex algorithmic search procedures. Optimal cladograms are found through exploration of the linked dynamics of search, transformation, and homology.

Suggested Reading

Cormen, T. H., C. E. Leiserson, and R. L. Rivest. 2001. Introduction to algorithms, second edition. Cambridge, MA: MIT Press. 1180 pp.

Farris, J. S. 1983. The logical basis of phylogenetic analysis. *In* N. I. Platnick and V. A. Funk (editors), Advances in Cladistics: 277–302. New York: Columbia University Press.

Sankoff, D. and P. Rousseau. 1975. Locating the vertices of a Steiner tree in arbitrary space. Mathematical Programming 9: 240–246.

Wheeler, W. C. 1996. Optimization alignment: the end of multiple sequence alignment in phylogenetics? Cladistics 12: 1–9.

Wheeler, W. C. 1999. Fixed character states and the optimization of molecular sequence data. Cladistics 15: 379–385.

Wheeler, W. C. 2003. Iterative pass optimization of sequence data. Cladistics 19: 254–260.

Wheeler, W. C. 2003. Search-based optimization. Cladistics 19: 348–355.

Part II
POY

4 Overcoming Limitations in Phylogenetics

Systematics achieves objective knowledge through hypothesis testing and, as in any other scientific discipline, makes two elementary demands on any investigation.

Repeatability Minimally, analytical procedures must be transparent, clear, and specific in how they treat data, as this is necessary to allow other investigators to repeat the analysis and test claimed results.

Epistemological coherence For scientific inferences to be valid, they must be methodologically, theoretically, and philosophically consistent. Empirical investigations must be firmly rooted in notions of evidence and inference, and they must describe and defend what is done, what is assumed, and why.

These two requirements, although crucial in science, are compromised by procedures such as manual ("by-eye") alignment of sequences, recoding of gaps, and nonobjective exclusion of data. These actions are common, but inject unnecessary irreproducibility through subjective and inconsistent methodologies.

Alignment

Conventional analysis of DNA sequence data is based on a preliminary step of positional homology identification commonly referred to as multiple sequence alignment. The alignment of DNA or amino acid sequences, in the form of a matrix of static "primary" homologies, or fixed alignment, conventionally constitutes the first of a two-step process in phylogenetic analysis. The second step is cladogram searching. Cladogram searching never tests the hypothesis of homology made by the initial alignment. As a consequence, the fixed alignment is treated as prior, or background, knowledge. However, nucleotide homologies are cladogram specific and cannot be treated objectively as knowledge obtained independently of phylogenetic analysis.

Two-step analysis is by far the most commonly used methodology, but nucleotide homology is necessarily topology dependent, regardless of the alignment or optimization algorithm. Consequently, impaired optimality or logical inconsistency may be introduced when sequence homology assessment is broken into two independent steps. This separation of alignment and cladogram searching is a less efficient means of discovering the globally optimal solution and often leads to incompatibility of the optimality criteria applied at different stages of analysis. Commonly, indel information is treated differently in the homology (alignment) step and subsequent cladogram searching, as when gaps are given some cost (as a fifth state) during the alignment but treated as missing data during the cladogram search.

While highly sophisticated cladogram searching algorithms have been developed and widely adopted, there has been less attention paid to the optimization of sequence data. By integrating homology identification and cladogram searching into the single phylogenetic framework advocated here and implemented in POY, inconsistencies are removed and the search for optimal cladograms is performed more efficiently.

Manual alignment

The most egregious violation of the criterion of repeatability is manual alignment. Any alignment procedure that depends on the investigator to visually align the hundreds of thousands or even millions of nucleotides commonly collected for current studies will be irreproducible and highly prone to bias, except in the most trivial of cases. Simple comparative cases also demonstrate that mechanical editing errors, such as deleting or inserting bases, among other problems, might well be common (Giribet et al. 2002).

Manual alignments generally lack any explicit discussion of how they are generated or the reasoning behind the chosen hypotheses of homology. Most workers who perform manual alignments seem to rely on vague notions of subjective pattern recognition, a method that lacks any logical connection to hypothesis testing. Alternatively, it is common for manual alignments to be informed by consideration of secondary structure (e.g., Kjer 1995). Superficially, ribosomal secondary structure may seem to provide an explicit, repeatable, and objectively defensible basis for performing manual alignments. However, several points should be considered.

First, secondary structure does not actually solve the problem of nucleotide homology. At best, it places constraints by establishing putative limits between loops and stems, but the nucleotides within each of those units must still be homologized (see Giribet 2002).

Second, determination of secondary structure is not nearly as simple and unambiguous as many studies suggest. The actual pattern of bonding is probabilistic and depends on the minimization of free energy and the thermodynamic stability of the resulting structure (see Durbin et al. 1998 and Mathews et al. 1999 for reviews). Programs explicitly designed to model secondary structure might find multiple, equally probable models, but such inherent ambiguities are overlooked when alignments are performed manually. Indeed, in phylogenetic studies, secondary structure is typically inferred by aligning with a sequence of "known" secondary structure, although the basis of that knowledge remains uncertain in many cases.

Third, although it might be reasonable to expect selective pressures to apply to secondary structure interactions (that is, requirements of compensatory changes), it is unclear just how relevant those interactions are compared to selective pressures applied at other structural levels.

Fourth, and most fundamentally, there is no necessary connection between functional considerations, including secondary structure, and the concept of homology, which refers strictly to the historical identity of objects related through shared transformation events. As discussed in the preceding sections, there are a number of biological problems that can be approached best by producing similarity-based alignments (or thermodynamic models) that aim to detect certain kinds of patterns in sequences, but those procedures are irrelevant to homology assessment and phylogenetic inference. In light of the many epistemological and theoretical shortcomings of the approach, manual alignment should be seen as a nonscientific procedure that is best avoided.

Automated multiple sequence alignment

Automated multiple sequence alignment, using the heuristic algorithms of programs such as Clustal W (Thompson et al. 1994), Malign (Wheeler and Gladstein 1994), or others, is a vast improvement over manual alignment. Automated multiple alignment does not solve the problem of finding globally optimal cladograms, but it is at least repeatable by other investigators. Alignment parameters, such as indel cost, are consistently applied over the sequences and are reported as part of the analysis. Unfortunately, much of the value of automated, explicit alignment procedures is squandered in two ways: manual "refinement" of alignments and application of inconsistent analytical assumptions in subsequent steps. Many authors might feel the necessity of "improving" automated alignments by eye without checking whether the correction actually improves the cost of the alignment. Hence, the manual alignment does not maximize the same criterion.

Exclusion of Data

The expurgation of data sets is another common procedure in phylogenetic analysis. This is most often justified on the basis of "difficulty" in alignment and "noise" in the data.

Alignment problems

Genotypic data that are prealigned, whether manually or using automatic alignment software such as Clustal W and Malign, can display regions claimed to be ambiguously aligned. Once such an ambiguously aligned region has been identified, it is usually excluded from further analysis. Two obvious concerns arise immediately: the first is the basis for identifying the ambiguity of the data, and the second is the general question of exclusion of data.

Typically, these regions are identified subjectively as disturbingly "ambiguous" on the basis of a single alignment. The criteria are invariably subjective. These criteria nearly universally come down to length variation. Investigators are uncomfortable with indel-rich regions or, worse, primary homologies that do not conform to preconceived notions of relationship (also a common problem during manual alignments). Rarely, if ever, are numerical criteria expounded to define or remove these regions. When they have been (Gatesy et al. 1993; Wheeler et al. 1995), the fraction of truly "alignment unambiguous" positions is absurdly small. Homologies in sequence data are established as part of a phylogenetic analysis, derived from a specific cladogram. There is no justification to discard data in systematics, where the data are the historical variation. Problems of interpre-

tation lie in the methodology used to analyze the data, not in the data themselves.

Noise and saturation

The exclusion of presumably "noisy" positions or even loci has become commonplace in molecular sequence-based systematics. Often, "noisy" positions or loci are determined *a priori*, using pairwise comparisons to create "saturation" plots (Adkins and Honeycutt 1994; Brown et al. 1979; Mindell and Thacker 1996). As argued by Allard et al. (1999), homoplasy is not uncovered from pairwise divergences, branch lengths, or even numbers of changes at a site, but by the actual state changes of single characters optimized on a cladogram.

The perceived problem with "noisy" versus "good" data stems from a misunderstanding of homoplasy and its informativeness (Källersjö et al. 1999). Furthermore, the exclusion of "noisy" data assumes that homoplasy necessarily confounds the inference of phylogeny. If nucleotide evolution were to occur at random, homoplasy might be misinformative, uninformative, or even informative. Wenzel and Siddall (1999) showed via simulation that half the characters in an analysis must be random for there to be a greater than even chance of overwhelming even a single unique and unreversed synapomorphy. This should not be surprising, considering that, given even a moderate number of randomized sites, there are so many ways to arrange the four possible nucleotide states among taxa that the chance of forming a pattern such that historically relevant data are contravened is extremely low (Wenzel and Siddall 1999).

Exclusion of data is not supported by conceptual or empirical analyses. In some cases, exclusion of whole DNA fragments has been justified in regions where there are no corresponding regions in other taxa (in essence an issue of missing data and autapomorphy), such as in novel loops in ribosomal genes or when the loops show enormous length variation. This procedure, when used, has been tested empirically via congruence measures (for example, sequence fragments that differ in length by orders of magnitude; Giribet et al. 2000).

Taxon exclusion

Analogous to the exclusion of sequence regions, taxa are often excluded. In the worst cases, available taxa are completely ignored. For example, Regier and Shultz (2001) presented a new gene to study arthropod phylogeny, elongation factor-2, but avoided inclusion of *Drosophila melanogaster* because the position of this organism did not satisfy their phylogenetic expectations.

In some cases, exclusion of taxa might be justified by using explicit procedures such as the relative-rate test (for example, Aguinaldo et al. 1997). In yet other cases, taxa excluded after a first set of analyses show "unacceptable" cladogram topologies (for example, Nardi et al. 2003). In these cases, an *ad hoc* explanation is supplied (such as biased nucleotide composition, long branch attraction, etc.) to explain the unorthodox placement of the taxon in question.

The main problem with excluding taxa from analyses is that it does not allow discovery of the phylogenetic position of those taxa. Instead of taxon exclusion, it has been repeatedly shown that increased taxon sampling is the only method of dealing with unorthodox placement of taxa.

POY

POY implements the concept of dynamic homology to overcome the limitations in phylogenetics discussed above. It seeks to combine the two historically disconnected processes of multiple alignment and cladogram searching into one step. Under this concept, intermediate alignment steps are avoided by directly assessing the number of evolutionary events—that is, DNA sequence transformations. This is accomplished through the generalization of existing character optimization procedures to insertion and deletion events (indels) and base substitutions. The crux of the approach is the treatment of indels as processes as opposed to the patterns implied by multiple sequence alignment (that is, gaps in static or fixed alignment). This method generates more efficient explanations of sequence variation (shorter, optimal trees) than do multiple alignments and produces multiple optimal results if more than a single optimal cladogram/alignment exists (Wheeler 1996, 2002). Within the framework of dynamic homology (for example, direct optimization), for any optimality criterion implemented in POY, insertion/deletion events are inferred historical events and used as information along with substitutions when hypothesizing common ancestry.

Optimality criteria and epistemological background

Since POY treats phylogenetic reconstruction as a single-step process, it solves the problem of inconsistency generated by conventional procedures that might use contradictory optimality criteria for alignment and phylogenetic analysis. Since there are several ways to establish homologies, POY offers different rationales to infer homologies under different optimality criteria: parsimony and maximum likelihood.

In both cladogram searching and character optimization, POY provides the user with complete control over the search strategies and computer resources used, implementing most of the recently developed algorithms for cladogram searching (such as random addition sequences [RAS], subtree pruning and regrafting [SPR], tree bisection and regrafting [TBR], ratchet, drift, and tree fusing) and a number of unique sequence character optimization algorithms (direct, iterative pass, fixed states, and search-based optimization; see Wheeler 1996, 1999a, 2003b, 2003c).

Here is a brief description of each of the four main sequence character optimization methods in POY. The specific algorithms are presented in Chapter 5 Character Optimization.

Direct optimization Direct optimization simultaneously evaluates nucleic acid sequence homologies and cladograms. Homologies are revised as POY performs the search, finding the homologies (nucleotide-to-nucleotide) that minimize the cladogram cost. It creates optimal cladograms without using multiple alignments and treats indels as phylogenetically informative transformation events.

Iterative pass optimization Iterative pass overcomes one limitation of direct optimization. With direct optimization, hypothetical ancestral sequences are optimized based only on descendant sequences. Iterative pass optimizes hypothetical ancestral sequences based on both descendant and ancestral sequences. This method is also founded on a nucleotide-to-nucleotide dynamic homology framework.

Fixed states optimization Fixed state optimization implements a strategy different from direct optimization and iterative pass optimization. Instead of constructing hypothetical ancestral sequences, fixed state optimization treats each sequence as a single, multistate character (static fragment-to-fragment homology). The edit cost of transforming each sequence/character into every other sequence/character is calculated as the weighted sum of the independent nucleotide transformations. From this cost matrix, the minimum cost cladogram is calculated.

Search-based optimization Search-based optimization overcomes a limitation of the fixed states optimization method associated with heuristics that are implemented to speed up searches. Search-based optimization employs enlarged sets of ancestral sequences. Frequently, these constitute less costly reconstructions than those of direct optimization, iterative pass optimization, and fixed state optimization. In the sense of finding all most-parsimonious trees, search-

based optimization is the most exhaustive algorithm currently implemented in POY, and potentially the most time-consuming.

Suggested Reading

Giribet, G. and W. C. Wheeler. 1999. On gaps. Molecular Phylogenetics and Evolution 13: 132–143.

Giribet, G., W. C. Wheeler, and J. Muona. 2002. DNA multiple sequence alignments. *In* R. DeSalle, G. Giribet, and W. C. Wheeler (editors), Molecular Systematics and Evolution: Theory and Practice: 107–114. Basel: Birkhäuser Verlag.

Phillips, A., D. Janies, and W. C. Wheeler. 2000. Multiple sequence alignment in phylogenetic analysis. Molecular Phylogenetics and Evolution 16: 317–330.

Wheeler, W. C. 1994. Sources of ambiguity in nucleic acid sequence alignment. *In* B. Schierwater, B. Streit, G. P. Wagner, and R. DeSalle (editors), Molecular Ecology and Evolution: Approaches and Applications: 323–352. Basel: Birkhäuser Verlag.

5 Character Optimization

This and Chapter 6 Cladogram Searching, present the specification of the algorithms currently used in POY. Chapter 5 contains character optimization procedures and Chapter 6 contains cladogram search procedures. Often, details are omitted to improve readability and clarity of explanations. The algorithms can be implemented in different ways, programming languages, and a diversity of speedups, but such matters are out of scope here. Our intention is that the general steps are presented so that the users of POY will understand their function and be able to more effectively exploit program options and strategies.

The chapter begins with notation and definitions ("Notation and Definitions" on page 38), followed by the algorithms for qualitative character optimization ("Qualitative Character Optimization Algorithms" on page 40), those specifically used in the sequence character optimization algorithms ("Sequence Optimization Algorithms" on page 45), and finally, a brief discussion of the likelihood methods available in POY ("Likelihood" on page 57).

Notation and Definitions

Notation

Each algorithm is written using the following atomic elements and notation:

variables are text elements that carry a number. The number could change during an algorithm execution, and is subject to any valid function over its domain. Variables will be written in lowercase italics. For example, *a*, *b*, *number_of_taxa*.

logical operators are premises that control the flow of the algorithm. These will be written in **bold**. For example, **if ... then ... else**.

literals are constants and predefined values. These are written in sans serif type. For example, A, C, G, T, Gap, 5.

assignment sets the value of a variable and is represented with the arrow symbol $\leftarrow$. It could also assign the output of a function. For example, $a \leftarrow 4$, $taxa \leftarrow 45$.

functions are algorithms that perform a predefined operation on a set of inputs $I = \langle i_1, i_2, \ldots, i_k \rangle$ and return a result. Every function is named and its result assigned to a variable. Functions will be written in italics followed by the set I in parentheses. For example, *the_maximum* $\leftarrow$ *max*(a, b, c) stores the maximum value of the set $\langle a, b, c \rangle$ in the variable *the_maximum*.

Common mathematical operations such as addition, multiplication, division, and subtraction as well as set operations like union and intersection are represented with the standard mathematical symbols ($+, \times, /, -, \cap, \cup$).

Definitions

The following algorithms deal mainly with structures commonly called trees or cladograms. In more technical terms, *a binary rooted tree* is a *connected graph* $T = (V, E)$ where V is the set of vertices and E the set of edges, such that

1. there are no cycles in T;
2. there is only one node with degree 2, and it is called *root*;
3. all the nonroot nodes have degree 3 or 1 if they are *internal* or *terminal* (also called *leaves*) respectively.

Vertex and node will be used interchangeably in the text, and will always be referring to the operational taxonomic units (OTUs) or hypothetical

taxonomic units (HTUs), if the node is a terminal (leaf) or internal respectively. Each vertex is described with the following information items, represented using dot notation:

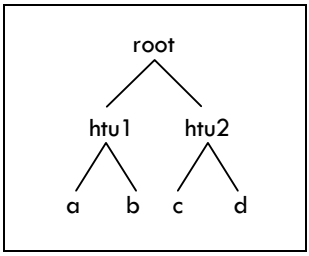

Figure 5.1: A tree example.

left is the left descendant of *n* (*htu1.left* = a, *htu2.left* = c in Figure 5.1).

$$n.left = \begin{cases} \text{empty} & \text{if } n \text{ is an OTU} \\ \text{node} & \text{otherwise} \end{cases}$$

right is the right descendant of *n* (*htu1.right* = b, *htu2.right* = d in Figure 5.1).

$$n.right = \begin{cases} \text{empty} & \text{if } n \text{ is an OTU} \\ \text{node} & \text{otherwise} \end{cases}$$

parent is the parental node of (*htu1.parent* = root, *root.parent* = empty in Figure 5.1).

$$n.parent = \begin{cases} \text{empty} & \text{if } n \text{ is the } \textit{root} \\ \text{node} & \text{otherwise} \end{cases}$$

chars is the set of a character of node *n*.

$$n.chars = \begin{cases} \text{observations} & \text{if } n \text{ is an OTU} \\ best\_htu(n.left,\ n.right & \text{otherwise} \end{cases}$$

where the best_htu depend on the algorithm being used. See the following sections for further details.

chars contents depend on the type of data being analyzed. For example in additive characters, *chars* maintains intervals, that is, ranges of character states that the taxonomic unit could have, but in direct optimization, is a complete nucleotide sequence. At the beginning of each section is included a precise description of any special conditions for this item.

cost is the weighted number of evolutionary steps required in the subtree formed by all the descendants of *n*.

$$n.cost = \begin{cases} 0 & \text{if } n \text{ is an OTU} \\ n.left.cost + n.right.cost + & \text{otherwise} \\ n.chars \rightarrow n.left.chars + & \\ n.chars \rightarrow n.right.chars & \end{cases}$$

where a $\rightarrow$ b is the number of evolutionary steps required to transform the character a to character b. Note that no node exists outside of a given cladogram; therefore, in this context, a node mention implies a cladogram definition, even though the topology is not described explicitly.

Qualitative Character Optimization Algorithms

Given a cladogram C, an optimization algorithm assigns an appropriate set of characters to its hypothetical taxonomic units (HTUs) such that the total evolutionary cost of the tree is minimized (most parsimonious character assignment). Initially, only the OTU characters are defined in the tree. For this reason, every optimization algorithm performs two steps, a *down pass* and an *up pass* (Algorithms 5.1 and 5.2).

Down pass

In a down pass, the smallest set of possible character states is assigned to each node in the cladogram starting in the leaves *down* to the root (and so the term down pass), resulting in the calculation of the cost of the most parsimonious assignment of states in for each node. In computer science this way to *traverse* a tree is called *postorder*, which can be attained by the use of *recursive* algorithms.

Recursive Functions

A function (algorithm) $f(x)$ is recursive if has the form

$$f(0) = k$$
$$f(t+1) = g(t, f(t))$$

that is, $f(t+1)$ depends on the value of $f(t)$ and another function $g(x,y)$.

A recursive algorithm simplifies a postorder traversal of a tree. Given a the cladogram $C=(V, E)$, applying a function $h(n)$ on each node $v \in V$ in postorder, is accomplished by $post(n)$ where

$$post(n) = \begin{cases} h(n) & \text{if } n \text{ is a leaf} \\ post(n.left), post(n.right), h(n) & \text{otherwise} \end{cases} \quad \text{(Eq 5.1)}$$

that is, in order to apply $h(n)$ in a node n, check whether n is a leaf or not. If it is, then apply the function $h(n)$. Otherwise, first apply the function to the left and right children of n, and only then calculate $h(n)$.

Up pass

The set of states on each node during the down pass may leave different levels of inconsistencies, depending on each algorithm and character types; a second pass, now called up pass, is required to properly assign HTU states.

The up pass consists of applying a function in the nodes of the tree in *preorder*, that is, starting at the root and moving up toward the leaves. This is easily accomplished by the use of recursion as well (see previous section). Given node n, initially the root of a cladogram, to apply $h(n)$ use

$$pre(n) = \begin{cases} h(n) & \text{if } n \text{ is a leaf} \\ h(n), pre(n.left), pre(n.right) & \text{otherwise} \end{cases} \quad \text{(Eq 5.2)}$$

that is, *after* applying $h(n)$, perform the same operation in the left and right children. If n is a terminal node, then just apply $h(n)$.

This pair of general algorithms will be used not only in the qualitative character section but also to the sequence and chromosomal characters.

Additive characters

The Additive Characters Optimization was the first formal definition of an optimization algorithm, specifically applied for *additive characters*. It was described by Farris (1970), based on a previous definition of Wagner trees (Wagner 1961). It finds one optimization, but not *all*, as pointed out by the author (Farris 1983; Goloboff 1993a). Different approaches to find all the possible character optimizations for additive characters have also been proposed (Goloboff 1993a; Gladstein 1997), but their description is out of scope on this book.

A character is additive if and only if its transformation costs can be represented by distances on a line. For example, a transformation from state 0 to 6 is counted as adding six steps to the tree, while a transformation from state 1 to 2 is counted as one step.

Formally, given an ordered set of states $c_1, c_2, \ldots, c_n$ of character c the evolutionary cost of transforming c_i to c_j is defined as $|i - j|$.

As the character states are ordered, every set of possible states including c_i and c_j must include all the intermediate states in between them, and so, any operation on such sets must consider this special case (for example, $\cap$, $\cup$).

As with any optimization algorithm, two steps are performed, a down pass and an up pass (Algorithms 5.1 and 5.2).

Down pass

This procedure is used to estimate the chars interval on each node that leads to the most parsimonious character optimization in a cladogram. As the character is additive, the median of every pair of nodes can be expressed as an interval between two values, and therefore each chars is the smallest interval that is a median between the left and right children of a node (Algorithm 5.1).

Algorithm 5.1 Downpass for the optimization of additive and nonadditive characters – *Additive_Downpass(n)*

Require: *n* is a node of a cladogram, initially its root
Require: *chars* is an interval of characters states
  if *n* is not a terminal **then**
    Additive_Downpass(n.left)
    Additive_Downpass(n.right)
    n.chars $\leftarrow$ *n.left.chars* $\cap$ *n.right.chars*

```
    if n.chars = ∅ then
      n.chars ← n.left.chars ∪ n.right.chars
    end if
  end if
```

Up pass

This procedure assigns the final characters to each internal node that produces the most parsimonious arrangement.

Algorithm 5.2 Uppass for additive characters optimization – *Additive_Uppass(n)*

```
Require: n is a node of a cladogram, initially its root
Require: n.chars is an interval assigned in Additive_Downpass
  if n is not a leaf and is not the root of C terminal then
    if n.chars is of the form ⟨a,..., b⟩ then
      n.chars ← n.chars ∩ n.parent.chars
    end else begin
    n.chars ← n.parent.chars
    end if
    Additive_Uppass(n.left)
    Additive_Uppass(n.right)
  end if
```

Nonadditive characters

Fitch (1971) proposed this algorithm to be used on gene sequences, to search for parsimonious cladograms when all nucleotide replacements have the same cost. As Fitch himself pointed out, this method could be used on any set of characters that complies with these requirements, so it is applicable on any unweighted, multistate, nonadditive character. Formally, the transformation cost between any pair of states c_i and c_j, $0 \leq j < n$ of character c with n states is a constant K

$$\sigma(c_i, c_j) = K.$$

In nonadditive characters, as opposed to additive ("Additive characters" on page 42), all character transformations have the same evolutionary cost. For example, a transformation from state 0 to 6 is an event that adds K steps to the cladogram.

The algorithm is an extension of the Additive Characters Optimization (Algorithms 5.1 and 5.2), with the new definition of the chars set, opposed to an interval set. The up pass makes a final assignment of all possible character states based on the information collected in the down pass (Algorithm 5.1).

Sankoff algorithm

The previous algorithms solve two special cases of a more general character evolution model proposed by Sankoff and Rousseau (1975). Additive and nonadditive characters assume the same cost of transformations (in one or no dimensions, respectively). Sankoff and Rousseau (1975) proposed a different representation, where transformation costs between n states in character c correspond to an n-by-n matrix S, where $S_{ij}, 0 \leq i, j < n$ represents the transformation cost from state c_i to c_j

$$\sigma(c_i, c_j) = S_{ij}.$$

This model is computationally more intensive, as there is no limit on the distribution of the costs, and there is no symmetry ($S_{ij} \neq S_{ji}$) constraint. For this reason, the algorithm most commonly used in this model belongs to a computational method called *dynamic programming*.

Dynamic programming is similar to the divide and conquer family of techniques, but is better when the subproblems are not independent (two subproblems may share a sub-subproblem); it is commonly used for optimization problems like phylogenetic trees and multiple sequence alignment.

Algorithm 5.3 Nonadditive Characters Up pass – *Non_Additive_Uppass(v)*

Require: *n* is a node of a cladogram, initially its root
Require: *n.chars* is assigned in *Additive_Downpass* using a set instead of an interval.
  if *n* is not an OTU nor the root of the cladogram **then**
    if *n.parent.chars* $\subseteq$ *n.chars* **then**
      n.chars $\leftarrow$ *n.parent.chars*
    else if *n.left.chars* $\cap$ *n.right.chars* $\neq \varnothing$ **then**
      n.chars $\leftarrow$ *n.parent.chars* $\cup$ *n.chars*
    else
      n.chars $\leftarrow$ *n.chars* $\cup$ (*n.parent.chars* $\cap$ *n.left.chars*) $\cup$ (*n.parent.chars* $\cap$ *n.right.chars*)
    end if
    Additive_Uppass(n.left)
    Additive_Uppass(n.right)
  end if

Again, analyzing Sankoff characters includes a down pass and an up pass but the algorithms are quite different.

Down pass

The down pass performs in a far more expensive optimization procedure. As the cost of each evolutionary step cannot be assumed, all the possible paths are (in principle) potential solutions. It is necessary then to keep track of each state on each node and its associated cost for the subtree it roots. Algorithm 5.4 does the following in postorder:

1. For each character state *i*, calculate the cost of transforming *i* to each possible character state in the child nodes.
2. Choose the minimum as the cost of the current node.
3. Store the information of the character combinations that lead to such best cost.

This requires $O(n^2)$ nested for loops compared with the linear $O(n)$ times of the previous character types (Algorithm 5.1).

Up pass

The up pass is still as simple as previously described. Traversing the cladogram *C* in preorder do the following on each node *n:*

1. If *n* is the root or a leaf, no further changes are needed in the character optimization of the node.
2. Otherwise, if *n* is a left node, the subset of *from_left* that is *true* is the set of possible characters of *n*.
3. Otherwise, if *n* is a right node, the subset of *from_right* that is *true* is the set of possible characters of *n*.

The method is more general and expensive, but allows arbitrary cost scenarios.

Sequence Optimization Algorithms

POY introduces a new set of algorithms for nucleotide sequence data. Their main advantage is that the sequences do not require a previous alignment and are treated as a character state of the homologous sequences; an alignment may be generated as a by-product of the most parsimonious cladogram.

The main drawback is the computational costs of these methods. If searching for the most parsimonious cladogram is expensive itself, performing a topology-specific alignment is even more so, nesting two NP-Hard problems. This drawback has been dealt with by POY in several ways, including heuristics and parallelization, the two most important computational tools for this kind of task.

The algorithms that will be introduced are direct optimization (page 47), fixed states and search-based optimization (page 55), and iterative pass

optimization (page 56). Each performs the down pass and up pass operations as previously explained ("Qualitative Character Optimization Algorithms" on page 40).

Algorithm 5.4 Down pass of the Sankoff-Rousseau Algorithm – *Downpass_Sankoff(n)*

Require: *n* is a node of a cladogram, initially its root
Require: the character *n.chars* has *k* states
Require: *n.cost* is an array of size *k* such that $n.cost_i$, $0 \leq i < k$, is the total cost of the tree rooted by *n* if *n* is assigned the state c_i of c.
Require: If *n* is a leaf then $n.cost_i \leftarrow 0$ if the observed state of *n* is *i*, otherwise $n.cost_i \leftarrow \infty$
  if *n* is not an OTU **then**
    Downpass_Sankoff(n.left)
    Downpass_Sankoff(n.right)
    for $i = 0$ to $k - 1$ **do**
      $n.cost_i \leftarrow \infty$
      $from\_left_i \leftarrow$ false
      $from\_right_i \leftarrow$ false
    end for
    $min\_left \leftarrow \infty$
    $min\_right \leftarrow \infty$
    for $i = 0$ to $k - 1$ **do**
      for $j = 0$ to $k - 1$ **do**
        if $min\_left > n.left.cost_i + S_{ij}$ **then**
          $min\_left \leftarrow n.left.cost_i + S_{ij}$
        end if
        if $min\_right > n.right.cost_i + S_{ij}$ **then**
          $min\_right \leftarrow n.right.cost_i + S_{ij}$
        end if
      end for
      $n.cost_i \leftarrow$ min_left + min_right
      for $j = 0$ to $k - 1$ **do**
        if $n.cost_i + S_{ij} =$ min_right **then**
          $n.from\_left_j \leftarrow$ true
        end if
        if $n.cost_i + S_{ij} =$ min_right **then**
          $n.from\_right_j \leftarrow$ true
        end if
      end for
    end for
  end if

Direct optimization

This is probably the most important technique of the set currently included in POY. The following algorithms are somehow based on the rational and methods proposed for direct optimization and serve as speedups or restrictive cases. They also inherit its computational expensiveness and difficulties.

The algorithm for direct optimization is from Wheeler (1996, 2002), and finds a set of parsimonious homologies for a set of sequences of different length by treating the full homologous nucleotide string as a character and each sequence as a state.

It is important to make clear that multiple sequence alignment is not performed, but it is *implied* by the cladogram. The researcher must still define the 5×5 transformation matrix S, that specifies the transformation cost between the four bases and gaps. The algorithm is implemented for nucleic acid sequences only, but is also applicable to protein sequences.

Down pass

The down pass is extremely different from the previously described down passes. There is a new problem to be solved, as mentioned previously: the minimum edit pairwise sequence alignment. Given a cladogram C, on each node n do the following in postorder:

1. Align the *chars* of the children of n.
2. Find a best ancestor for the aligned sequences on each nucleotide and assign it as n's *chars*.

The second step is conceptually the same as in the Sankoff algorithm, applied for the whole nucleotide sequence. The pairwise alignment and a best ancestor are found in POY, as shown in Algorithm 5.5.

Algorithm 5.5 Down pass in the direct optimization procedure – *Do_Downpass(n)*

Require: *n* is a node of a cladogram, initially its root
Require: *n.left.chars* is the left child sequence with length *m*
Require: *n.right.chars* is the right child sequence with length *n*
  if *n* is not a leaf **then**
    Do_Downpass(n.left)
    Do_Downpass(n.right)
    $M \leftarrow$ *score(n.left.chars, n.right.chars)* {Algorithm 5.6}
    alignment $\leftarrow$ *backtrack(M, m, n)* {Algorithm 5.7}
    $n.left.chars \leftarrow alignment_1$
    $n.right.chars \leftarrow alignment_2$

```
    n.chars ← best_htu(n.left.chars, n.right.chars) {Algorithm 5.8}
    n.cost ← n.left.cost + n.right.cost + M[m][n]
  end if
```

Pairwise alignment: calculating the best score The pairwise alignment algorithm using dynamic programming is widely known as the Needleman–Wunsch algorithm (Needleman and Wunsch 1970).

The issue to solve is basically to find an optimal edit of two sequences B and C of length m and n to the sequences B' and C' of length o, such that the alignment cost

$$A = \sum_{i=1}^{o} \sigma(B'_i, C'_i)$$

is minimized; B_i is the i^{th} nucleotide in the sequence B, and $\sigma(x, y)$ is the cost of transforming nucleotide x into y. Note that B and C and the alignment A include all the possible IUPAC codes, *excepting* a gap, which is no data at all.

Let's first define the *empty string* λ as $|\lambda| = 0$, that is, it contains no elements of the alphabet on it. A *prefix* of a string S of length n is a string U such that $S_1 = U_1$, $S_2 = U_2$, ..., $S_k = U_k$, $0 < k < n$ or $U = \lambda$. By definition, λ is the prefix of every sequence. Dynamic programming works by aligning prefixes of B and C progressively in the most efficient way, until the two full sequences have been aligned.

Algorithm 5.6 Score Matrix calculation in the Needleman–Wunsch algorithm –*Score(B,C)*

Require: *B* and C are the sequences of length *m* and *n* to be aligned
Require: *M* is the (*m* + 1) x (*n* + 1) scores matrix for the *B* and C edit
  M[0][0] = 0
  for *i* = 1 to *m* **do** {Cost of aligning $B_1 ... B_i$ with λ}
    M[*i*][0] = M[*i* – 1][0] + σ (B_i, *gap*)
  end for
  for *i* = 1 to *n* **do** {Cost of aligning $C_1 ... C_i$ with λ}
    M[0][i] = M[0][i – 1] + σ (C_i, *gap*)
  end for
  for *i* = 1 to *m* **do** {Fill progressively M}
    for *j* = 1 to *n* **do**
      cost_align = M[*i* – 1][*j* – 1] + σ (B_i, C_j)
      cost_gap_in_C = M[*i* – 1][*j*] + σ (C_i, *gap*)
      cost_gap_in_ B = M[*i*][*j* – 1] + σ (*gap*, B_j)
      if *cost_align* ≤ *cost_gap_in_ B* AND *cost_align* ≤ *cost_gap_in_ A*
    then
        M[*i*][*j*] = *cost_align*
      else if *cost_gap_in_C* ≤ *cost_gap_in_B* **then**
        M[*i*][*j*] = *cost_gap_in_C*
      else
        M[*i*][*j*] = *cost_gap_in_B*
      end if
    end for
  end for
  return M

Let $Score(i,j)$ be the score of aligning prefixes $B_1, ..., B_i$ and $C_1, ..., C_j$, calculated as follows:

$$Score(i,j) = \begin{cases} 0 & \text{if } B = C = \lambda \\ Score(i-1,0) + \sigma(B_i, \text{gap}) & \text{if } j = \lambda \text{ and } i \neq \lambda \\ Score(0,j-1) + \sigma(C_i, \text{gap}) & \text{if } j \neq \lambda \text{ and } i = \lambda \\ min(i,j) & \text{otherwise} \end{cases}$$

where $min(i,j)$ is

$$min(i,j) = \begin{cases} Score(i-1, j-1) + \sigma(B_i, C_j) \text{ Align } B_i \text{ and } C_i \\ Score(i-1, j) + \sigma(B_i, \text{gap}) \text{ Align } B_i \text{ with a gap} \\ Score(i, j-1) + \sigma(C_j, \text{gap}) \text{ Align } C_i \text{ with a gap} \end{cases} \quad \text{(Eq 5.3)}$$

that is, three possible alignments are evaluated on each time:

1. $B_1, \ldots, B_{i-1}$ and $C_1, \ldots, C_{j-1}$
2. $B_1, \ldots, B_{i-1}$ and $C_1, \ldots, C_j$
3. $B_1, \ldots, B_i$ and $C_1, \ldots, C_{j-1}$

The dynamic programming technique builds an $(n + 1) \times (m + 1)$ matrix M, such that the matrix element $M_{i,j}$ is the score of aligning prefixes B_i and C_j. In order to calculate $Score(i,j)$, only the matrix elements $M_{i-1, j-1}$, $M_{i-1,j}$ and $M_{i,j-1}$ are necessary (Equation 5.3). The best score for aligning sequences A and B will then be $Score(i,j)$ and is calculated by Algorithm 5.6.

Pairwise alignment: calculating the edits The necessary edits of the sequences B and C can be found by backtracking the matrix M previously built. This consists basically of following from $M_{m,n}$ which one is the best option between $M_{i-1,j-1}$, $M_{i-1,j}$ and $M_{i,j-1}$ (Equation 5.3). Depending on the best option, the resulting edit may align B_m and C_n, insert a gap in C to align B_i with it, or insert a gap in B to align C_i with it. The operation with the rest of the prefix is repeated, as shown in Algorithm 5.7.

Algorithm 5.7 Backtrack of the Needleman–Wunsch pairwise alignment – *Backtrack(M, x, y)*

Require: Let *x* and *y* be two strings of length *m* and *n* to be edited for an optimum alignment, *x* and *y* are the indices in *B* and C

Require: Let M be a $(m + 1) \times (n + 1)$ matrix from *Score(x, y)* (Algorithm 5.6)

Require: *sequences* is an array of strings, where $sequences_1$ is one string and $sequences_2$ is the other string

if $M[x-1, y-1] \leq M[x-1, y]$ AND $M[x-1, y-1] \leq M[x, y-1]$ **then**
{Align B_x and C_y}
sequences ← *Backtrack*(M, $x-1$, $y-1$)
append to $sequences_1$ the nucleotide B_x

```
    append to sequences_2 the nucleotide C_y
  else if M[x − 1, y] ≤ M[x, y − 1] then
    {Align B_x with a gap}
    sequences ← Backtrack(M, x − 1, y)
    append to sequences_1 the nucleotide B_x
    append to sequences_2 a gap
  else {Align C_y with a gap}
    sequences ← Backtrack(M, x, y − 1)
    append to sequences_1 a gap
    append to sequences_2 the nucleotide C_x
  end if
  return sequences
```

Estimation of the ancestor This is the final step in the direct optimization down pass. For each HTU is assigned a nucleotide sequence H, using the IUPAC ambiguous nucleotides codes, choosing on each position H_i the set that minimizes $\sigma(A_i, H_i) + \sigma(B_i, H_i)$, as presented in Algorithm 5.8 and no gaps on it.

Algorithm 5.8 HTUs *v* sequence estimation based on the alignment of its children

```
Require: The descendant nodes n.left and n.right are two already aligned
    and edited sequences of length m
  for i = 1 to m do
    best_cost ← ∝
    for all code x in IUPAC do
      if σ(x, n.left_i) + σ(x, n.right_i.chars) < best_cost then
        best_cost ← σ(x, n.left_i.chars) + σ(x, n.right_i.chars)
        n.chars_i ← x
      end if
    end for
  end for
```

Up pass

The up pass (Algorithm 5.9) consists basically of traversing in preorder the tree and performing the following on each node:

1. Align each node with its ancestor, as each HTU may have a different size.
2. Assign on each hypothetical nucleotide the IUPAC code that gives the best cost for the total sum of weighted evolutionary steps between the ancestor and descendants, but eliminating positions that optimize to gaps, which are not observed in real data.

The procedure is similar to any other up pass previously explained, assigning the best IUPAC code on each position, to produce the most parsimonious nucleotide sequence.

Algorithm 5.9 Up pass in direct optimization – *Uppass(n)*

Require: *n* is a node with a sequence character of length *p*
Require: *n.parent*, *n.left*, and *n.right* already had a down pass of direct optimization, with length *q*, *r*, and *s*, respectively
  if *n* is not a leaf **then**
    if *n* is not the root **then**
      M ← *Score*(*n.chars*,*n.parent.char*)
      alignment ← *Backtrack*(*M*, *p*, *q*)
      n.chars ← *Final_HTU*(*alignment*) {See Algorithm 5.10}
    end if
    Uppass(*n.left*)
    Uppass(*n.right*)
  end if

Extending direct optimization: chromosome data

Complete chromosomal and genomic sequence data sets are a form of comparative data with modes of variation not easily accommodated into existing character types. Generalization of optimization techniques is required to take advantage of this new source of information. The techniques of chromosomal optimization presently implemented in POY are directly applicable to the analysis of complete genomes of single-chromosome organisms. Many systematic problems we face today involve such taxa or data including viral, bacterial, and mitochondrial systems.

Algorithm 5.10 Finding the best sequence of nucleotides for an HTU – *Final_HTU*(*alignment*,*v*)

Require: alignment is an array with two strings of length *m*, containing the alignment of *n* and *n.parent*
  $i \leftarrow 1$
  $j \leftarrow 1$
  $k \leftarrow 1$
  while $i \leq m$ **do**
    if $n.left_i$ AND $n.right_i$ are a gap **then**
      $i \leftarrow i + 1$
    end if
    if $alignment_{1\,k}$ is a gap **then**
      $k \leftarrow k + 1$

```
        best_cost ← ∞
        for all code x in IUPAC do
          if σ(x, n.left_i) + σ(x, n.right_i ≤ best_cost) then
            best_cost ← σ(x, n.left_i) + σ(x, n.right_i)
            htu_i ← x
          end if
        end for
        k ← k + 1
      end if
      i ← i + 1
      j ← j + 1
    end while
```

Chromosomal analysis is treated by POY as a character type. As such, it can be simultaneously optimized with other forms of data such as anatomy, other sequence information, or even other chromosomes in a complex genome. The general principles of optimization and combined analysis apply here as they do with all sources of character information. The procedures described here are from Wheeler (submitted).

In addition to the previously mentioned differences typical of nucleotide sequences, chromosomes vary in the loci complement and organization. Some loci might originate or be lost (origin–loss, analogous to nucleotide insertion–deletion) and might be rearranged (their relative position in the chromosome). These variation patterns for homologous chromosomes should then be arranged in the most parsimonious tree.

By using the DO approach, no locus identity annotation is necessary and the homologies are topology specific. The analysis is analogous to a multiple alignment of sequence data and, as there, homology constraints are unnecessary.

A chromosome is defined here as a series of adjacent nucleotide strings of known boundaries. POY represents a chromosome as a string of nucleotides with pipes ('|') marking the *locus* boundaries. Formally, a chromosome is a string from the alphabet {*A, C, G, T,* |} such that any substring found between two "|" is not empty. In DO, the whole chromosome is analyzed as a character; the distance between OTUs is given by the transformation cost between them, using the following components:

Indels Indels within a locus are treated like in any other DNA sequence, by DO, that is, the cost of transformation between two nucleotide sequences of the same locus is the cost of the pairwise alignment between the sequences (Algorithms 5.6, 5.7). That means that the transformation matrix is still necessary.

Origin or loss Loss of origin of a locus is treated in a similar way as any indel. The pairwise alignments of the complete locus set of the taxa included in an analysis of one chromosome are calculated and used as a transformation costs matrix for a full chromosome alignment, that is, instead of using an alphabet with the bases, as in any nucleotide sequence, assign one letter to each *locus* and create the alignment. For every pair, the transformation cost is the edit cost of the nucleotide sequence pairwise alignment of the *loci*.

Locus rearrangement This is currently measured using the *breakpoint metric* (Sankoff and Blanchette 1998). The breakpoint distance between two chromosomes is, simply stated, the number of locus adjacencies present in one chromosome and not the other.

Formally, the best alignment A of chromosomes B and C of nucleotide length m and n, and o and p *loci*, minimize the edit cost

$$\Theta(B,C) = \sum_{i=0}^{M} \left(\theta(B_{li}, C_{li}) + \sum_{j_i=0}^{L_i} \sigma(B_{li}^{j}, C_{li}^{j}) + \varepsilon(\{B_{li}, B_{li+1}\}, \{C_{li}, C_{li+1}\}) \right)$$

where M is the number of homologous *loci*, L_i is the number of nucleotides on *locus* L, and $\theta(x,y)$ is the origin-loss cost defined as

$$\theta(B_{li}, C_{li}) = \begin{cases} 0 & B_{li} \neq \text{gap and } C_{li} \neq \text{gap} \\ locus\_gap + (L_i \times locus\_size\_gap) & \text{otherwise} \end{cases}$$

given the *locus_gap* cost of adding or losing a *locus* and the cost *locus_size_gap* per each added or lost nucleotide in the locus. The breakpoint cost is defined as

$$\varepsilon(\{B_{li}, B_{li+1}\}, \{C_{li}, C_{li+1}\}) = \begin{cases} 0 & \text{if } \{B_{li}, B_{li+1}\} = \{C_{li}, C_{li+1}\} \\ 1 & \text{otherwise.} \end{cases}$$

Calculating the σ and θ functions is not difficult, as the regular pairwise alignment methods previously exposed are used on each substring and the total set of loci using the pairwise edit matrix for their alignment. The main issue remaining is finding the best reordering of the locus that minimizes the cost of the function.

POY performs this operation in a way analogous to the Wagner trees building, by adding sequentially the loci of one sequence and calculating on every available position the one that minimizes the total cost in addition to the breakpoint metric. Then an alignment of the *loci* is per-

formed, using the pairwise edit costs of the *loci* nucleotides sequences to produce a chromosome alignment (Algorithm 5.11). After it is built, some *locus* can be exchanged for cost improvements, in a similar way as SPR does on trees, to find an efficient alignment (not shown in the algorithm for simplicity).

Algorithm 5.11 Downpass for Direct Optimization of Chromosome Data

Require: *n* is a node to be optimized in a cladogram, initially its root
Require: Define *algn*
Require: *n.left* and *n.right* are chromosome sequences of p, q, $p \leq q$ *loci* length
Require: M is a q x p matrix such that M_{ij}, $i \leq q$, $j \leq p$ is the cost of the pairwise alignment of *loci* $n.right_i$ and $n.left_j$
  if *n* is not a leaf **then**
    for $i = 0$ to q **do** {Adds sequentially the *loci* in the best position}
      $cost \leftarrow \infty$
      for each *locus* $l \in n.right$ but *not* $\in algn$ **do**
        if $\Theta(n.left_i, l) < cost$ **then**
          $algn_j \leftarrow n.left_i$
          $cost \leftarrow \Theta(n.left_i, algn)$
        end if
      end for
    end for
    S $\leftarrow$ *Score*(*n.right, algn*) {using M as transformation cost matrix in Algorithm 5.8}
    $sequences_1$ and $sequences_2 \leftarrow$ backtrack(*n.right,algn,S*) {Algorithm 5.7}
    $n.right.chars \leftarrow sequences_1$
    $n.left.chars \leftarrow sequences_2$
    n.chars $\leftarrow$ best_htu (*n.left, n.right*) {Using M as transformation cost matrix (Algorithm 5.8)}
  end if

The up pass works in exactly the same way as explained for regular nucleotide sequences under DO, considering the new edit cost.

Fixed states and search-based optimization

Fixed states (Wheeler 1999a) and search-based optimization (Wheeler 2003c) constitute the most rigid and restrictive methods of sequence optimization algorithms. Both consist of treating each sequence as a different character state of the homologous sequences, to build a Sankoff matrix with the cost of each pairwise alignment (Algorithm 5.12). Then the Sankoff Optimization is used to build the most parsimonious tree, using the observed sequences as the only possible character states (fixed

states) or adding some other sequences built by some heuristic (search based). Note that if, instead of using some set of sequences, *all* possible sequences are included in the search-based optimization, to build a huge transformation matrix, an exact solution is computable (if you have infinite time and computational power).

Algorithm 5.12 Buildup of the Sankoff matrix of costs for the Fixed States Optimization

Require: An array T with n nucleotide sequences to build the Sankoff Matrix, and length T_{Li} of each sequence i
Require: A matrix C of $n \times n$ elements
  for $i = 1$ to n **do**
    $C_{i,i} \leftarrow 0$
    for $j = i + 1$ to n **do**
      $M \leftarrow Score(T_i, T_j)$
      $C_{i,j} \leftarrow M_{TLi}, M_{TLj}$
    end for
  end for

In the same way, for chromosome sequences, the down pass function is replaced with the modified down pass for chromosome data, and the rest of the procedure remains the same.

Iterative pass optimization

In iterative pass optimization (Wheeler 2003b), the sequences are aligned for the topology, as in the other direct optimization techniques. The node sequences are initialized using the direct optimization down pass (Algorithm 5.5). Then the up pass is run iteratively, reevaluating each HTU with its two descendants and its parental node, until no more changes occur in the cladogram. The Needleman–Wunsch algorithm in a three-dimensional matrix is used to perform this HTU reevaluation, either for simple nucleotide sequences or full chromosomes, with their respective edit cost functions (Algorithm 5.13).

Algorithm 5.13 Iterative pass optimization method

Require: Node n is the root of a cladogram to be optimized *Downpass(r)*
  repeat
    Uppass_IT {Algorithm 5.14}
  until No more changes occur in the cladogram

The method is obviously more time consuming than the simple direct optimization, and there is no time bound for the time the tree will take to stabilize. The up pass is shown in Algorithm 5.14.

Algorithm 5.14 Up pass iteration steps – Uppass_IT(v)

Require: A complete down pass on node *n*.
Require: aligned is a vector with the edited sequences that were realigned
  if *n* is not a leaf **then**
    if *n* is not the root **then**
      aligned ← *Align*(*n.parent, n.left, n.right*)
      n.chars ← *Best_HTU*(*aligned*[1], *aligned*[2], *aligned*[3]) {See Algorithm 5.15}
    end if
    Uppass_IT(n.left)
    Uppass_IT(n.right)
  end if

Algorithm 5.15 Finding the best HTU between three aligned sequences – Best_HTU(a, b, c)

Require: a, b, and c are aligned nucleotide sequences with length *l*
Require: htu is the resulting HTU sequence of the algorithm
  for *i* = 0 to *l* **do**
    best_cost ← ∝
    for all code *x* in IUPAC **do**
      if $\sigma(x, a1_i) + \sigma(x, b_i) + \sigma(x, c_i) \leq$ *best_cost* **then**
        best_cost ← $\sigma(x, v.left_i) + \sigma(x, v.right_i)$
        htu_i ← x
      end if
    end for
  end for
  return htu

Likelihood

Likelihood as an optimality criterion

The use of likelihood in POY is an alternate optimality criterion. The models created and implemented are conceived as interpretive tools, without any necessary relationship to the actual process of change in nature. This is somewhat different from the philosophy of other likelihood approaches embodied in software such as PAUP* (Swofford 2002), which treat the statistical models used as depictions of the actual process of evolution—in essence, natural law. The procedures described here are those of Wheeler et al.(2006).

The optimality approach is both liberating and constraining, in that the numerical values POY attaches to cladograms may not be directly com-

parable to those in other implementations. Part of this is due to the dynamic homology approach to sequence characters, but a second (and perhaps more important) factor is that the likelihood analogues to the sequence character heuristics are doing slightly different things. Each is a different sort of approximation approach, and whereas the parsimony cladogram lengths are exactly comparable, the likelihood values may not be.

Comparison with other likelihood methods

The main difference between likelihood in POY and other implementations is the foundation of dynamic homology. No preexisting multiple alignment is required or used in the analysis (although users can input prealigned sequences), and nucleotide homology is determined as an optimization problem on the cladogram. In this case, the homology scheme that maximizes overall cladogram likelihood (since the absolute value of the natural logarithm of likelihoods is used this is minimizing cost as with parsimony analysis) is chosen. In reality, it is not a single homology scheme but all possible schemes given that each homology scenario contributes to overall likelihood, albeit many in a very minor way (Felsenstein 2004). When cladograms are diagnosed or homologies displayed, the state assignments are the dominant (highest value) likelihoods, or in the terminology of Barry and Hartigan (1987), most parsimonious likelihood state assignments. During cladogram search, this form of likelihood can be used (direct optimization, iterative pass) or the more frequently used maximum average likelihood (Barry and Hartigan 1987) with appropriate program options (fixed states, search-based with `totallikelihood` on page 326). This alternate flavoring of likelihood, coupled with the multiple homology schemes of dynamic homology makes the numerical likelihood values not strictly comparable.

Pairwise comparison

The most basic step of likelihood dynamic homology is the likelihood version of the string matching in direct optimization. POY does not presently use the Thorne et al. (1991; TKF) procedure, but modifies Wheeler (1996) in a vein similar to McGuire et al. (2001), simply using "gap" as a fifth state in a Markov process. This approach is far simpler computationally than the Thorne et al. (1991) method, but does not have the indel-length birth–death model that would make affine indel likelihoods jibe with the actual model of transformations as opposed to the kludge implemented in POY. On the other hand, the TKF model has at least one property that is, on the face of it, undesirable. The birth–death model for indel length can create asymmetries that appear illogical (Figure 5.2).

```
-TGT-C-     -TG-TC-
G-C-ACA     G-CA-CA
```

Figure 5.2: Redrawn from Thorne et al. (1991). These two pairwise alignments have different likelihoods.

This sort of asymmetry would probably express itself as attaching different likelihoods to alternate roots of the optimized cladogram.

Models

POY uses a five-state, 10-parameter, general time reversible model (GTR) to optimize cladograms. This may be constrained to require all indel transition probabilities to be identical or that classes such as transitions, transversions, all nucleotide substitutions, or even all transformations have equal probability. In the same way, each of the five state frequencies (A, C, G, T, gap) may vary (Figure 5.3).

	A	C	G	T	gap
A	-				
C	a	-			
G	b	c	-		
T	d	e	f	-	
gap	g	h	i	j	-

Figure 5.3: Default Q matrix. Each of a–j is independently estimated.

Substitution models are created by numerically determining Eigen values and iterating branch lengths in terms of time. In general, the transition (Q) values are estimated by pairwise parsimony-based alignments and counting up the transformation of each type (including indels) along the sequence. Q matrices can also be input by the user.

The approach with these models is to be as general as possible and allow users to specify simpler models as special cases. This no doubt compromises efficiency, but allows flexibility.

Models are, by default, reestimated for every branch of each cladogram examined, but they can be fixed throughout the analysis. Fixed state and search-based optimization transformation models are constructed before cladogram search, but for each sequence state pair independently for all sequence characters.

Dynamic homology

When dealing with a pairwise comparison, the central difference between likelihood and parsimony alignment is in the summing of the likelihoods of all alignments between two sequences, as opposed to finding the one (or several) with lowest cost. This expresses itself in the string matching component (Needleman and Wunsch 1970), where the cost of a cell in the dynamic programming matrix is determined by the minimum path to that cell for parsimony minimization and the sum of the three paths to that cell for likelihood. The final likelihood will be the sum of all likelihoods of aligning the two sequences, but when a single pairwise alignment is presented—usually the dominant or highest likelihood component—this single alignment may contain only a small fraction of that total likelihood (Hein et al. 2000; this can be explored in POY using the default behavior-dominant `likelihood` on page 274, or total with `totallikelihood` on page 326 and `trullytotallikelihood` on page 328).

The direct optimization procedure for likelihood proceeds as defined in Algorithms 5.6, 5.7, and 5.8 with the exception that likelihood pairwise optimizations are performed as opposed to minimum cost parsimony comparisons. The likelihood score reported under direct optimization for likelihood is the sum of the likelihoods of each of the HTUs created on a single down pass of the cladogram. The branch length iterations are not revisited after this single pass. The dominant HTU sequence is used for optimization of its ancestral node.

Iterative pass optimization under likelihood proceeds in ways more similar to more familiar implementations. As with parsimony-based iterative pass, HTU sequences are constructed using a three-dimensional (3-D) direct optimization. In this case the 3-D optimization is likelihood based, with simultaneous iteration of the lengths of the three adjacent branches. Iterative pass also repeatedly reestimated HTU sequences, revisiting each node and its connecting branches until the HTU sequences (hence overall likelihood) are stable.

When fixed states and search-based optimization are employed, likelihoods can again be dominant (maximum parsimony likelihood) or total (maximum average likelihood using `totallikelihood`). To begin, a pairwise likelihood edit cost matrix is calculated to create the maximum likelihood value (or minimum – log L) between two sequences on a branch. The branch length is iterated until the maximum value is attained for each pair individually. Each likelihood transformation between sequence states may imply a different branch length. This matrix is the square of the number of sequence states (that is, observed unique sequences for fixed states; user specified size for search-based).

Cladogram optimization proceeds as for standard fixed states and search-based using the dynamic programming of matrix or Sankoff characters. The default behavior is to calculate likelihoods based on the maximum likelihood state, which yields a maximum parsimony likelihood. The likelihoods of all states reconstructions is summed using `totallikelihood`, yielding a maximum average likelihood value. As mentioned before, the branch lengths, upon which the likelihoods of transformations between sequence character states depend, are determined prior to cladogram optimization, hence branch lengths are not iterated during this stage.

Implied alignment

Likelihood implied alignment can be generated based on a cladogram as with parsimony analysis. The alignment will be based on the dominant likelihood HTUs optimized to internal nodes. All the caveats that apply to implied alignments under parsimony apply for the likelihood version (Wheeler 2003a). Their use as multiple alignments for likelihood cladogram searching is as fraught with difficulty and suspect as before.

Combined analysis

Combined (also referred to as simultaneous, concatenated, or total evidence) analysis is a driving force behind POY. Likelihood analysis is no exception. Qualitative characters can be optimized under likelihood using the general, elegant, and simple model of Tuffley and Steel (1997). Nonadditive (unordered) character transformations are treated directly as Tuffley and Steel (1997) derive, and additive characters are binary recoded and then treated as nonadditive characters.

In combination with sequence characters, likelihood combined analysis can be performed. At present, neither matrix (= Sankoff) nor chromosomal characters can be optimized under likelihood.

```
poy -likelihood chel.morph chel.seq
```

Figure 5.4: Example of a combined analysis run using likelihood as an optimality criterion, where "chel.morph" represents a morphological matrix and "chel.seq" represents a sequence data matrix.

Support measures

The two main measures of support provided by POY, Bremer and jackknife values, can be calculated using likelihood optimality.

The Bremer values POY reports, the difference in log L between the maximum likelihood cladogram with and without a group, are the likelihood ratios of groups.

Jackknife values under likelihood would seem to be close kin to the sort of clade-specific posterior probabilities produced by MrBayes (Huelsenbeck and Ronquist2003). Although they may be very close in value, the jackknife output in POY is a measure of group support to character resampling (albeit using likelihood) and not an attempt to generate posterior probabilities.

Suggested Reading

Farris, J. S. 1970. Methods for computing Wagner trees. Systematic Zoology 19: 83–92.

Fitch, W. M. 1971. Toward defining the course of evolution: minimum change for a specific tree topology. Systematic Zoology 20: 406–416.

Gusfield, D. 1997. Algorithms on Strings, Trees, and Sequences: Computer science and Computational Biology. Cambridge, MA: Cambridge University Press. 534 pp.

Sankoff, D. and P. Rousseau. 1975. Locating the vertices of a Steiner tree in arbitrary space. Mathematical Programming 9: 240–246.

Wheeler, W. C. 1996. Optimization alignment: the end of multiple sequence alignment in phylogenetics? Cladistics 12: 1–9.

Wheeler, W. C. 1999. Fixed character states and the optimization of molecular sequence data. Cladistics 15: 379–385.

6 Cladogram Searching

Optimization procedures always assume a given topology or cladogram as the framework for the optimization. The second component of systematic analysis is cladogram search, which is discussed here.[1]

The most obvious algorithm to ensure an exact solution is to enumerate all possible cladograms, optimize the characters on each one of them, and determine the optimal solution. This is pratically impossible for all but the most trivial data sets (fewer than 12 or 13 taxa), due to the size of the search space. Analyses of real data sets rely almost exclusively on heuristic solutions.

In addition to the notation of Chapter 2, cladogram structures will be represented as in Figure 6.1.

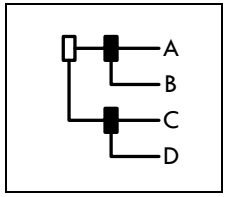

Figure 6.1: Representation of a cladogram. The root is the empty box, any HTU is represented as a filled box, and the OTUs (leaves) are letters.

1. This chapter is based in part on work by Jan De Laet.

Definitions

Given a cladogram $C = (V,E)$ with root r and some vertex $v, r, v \in V$, and a collection of characters:

Descendants of v, D_v of some vertex v are the set of nodes that include v in their path to r.

Direct descendants of v are the only two neighbors of v that include it in their path to r.

C.cost of a tree is the sum of the cost at r ($r.cost$) of all the characters using the algorithms of Chapter 5.

c.weight is the weight of a character **c**.

pop(X) is a function that will return the last element of the stack **X**.

push(X, a) is a function that will append at the end of the stack **X** the element **a**.

Branch and Bound

Branch and bound is a generally applicable exact solution procedure for cladogram search widely used in several problem solving techniques (Nilsson 1998). Branch and bound is not currently implemented in POY. It is presented here to demonstrate how an exact algorithm works and how it relates to heuristic solutions. The entire cladogram space must be (at least implicitly) examined. Branch and bound relies on growing (that is, before all taxa are added) cladogram costs. Once the cost of a partial cladogram on a search path is higher than the current lowest cost cladogram (Algorithm 6.1), that search path is halted. The procedure was first applied in a phylogenetic cladogram search by Hendy and Penny (1982).

Algorithm 6.1 Branch and Bound Algorithm

```
Require: C is a cladogram
Require: L is the set of optimal trees, initially empty
Require: best is initially infinity
Require: t is initially 1
  if t > n then
    for each edge (a, b) in C do
      make a copy NC = (NV, NE) of C
      remove (w, v) from NE
      append Tt and a new vertex x to NV
      append (w x), (x, v) and (x, Tt) to NE
      if the NC.cost ← best then
        best ← BB (NC, t + 1, best)
      end if
    end for
else
    if C.cost < best then
      empty L
      append C to L
      return C.cost
    else if C.cost = best then
      append C to L
      return best
    end if
end if
```

Branch and bound is significantly faster than enumerating all cladograms. However, it is unable to calculate the cost frequency because not all the cladograms are evaluated. This problem can be solved, and the number of visited cladograms increased, by setting up the cutoff factor greater than the current best cost.

Wagner Trees

A Wagner tree is built by adding sequentially the set of terminals in the edge that yields the least expensive cladogram. When more than one location produces the same minimum cost, each cladogram is used as the starting point for the next terminal, and both are evaluated separately (Algorithm 6.2).

Algorithm 6.2 Wagner tree building

Require: t is an initial tree containing only two OTU, T_n and T_{n-1}

```
Require: Let T be the set of the OTUS not contained in τ, initially T_1, ..., T_{n-2}
Require: WAG is the resulting stack of Wagner trees, initially containing τ
Require: TMP and TMP_2 are temporary stacks of trees that will produce a Wagner tree
while TMP is not empty repeat
    C ← pop(TMP)
    if C contains all the OTUS then
    push(WAG, C)
  else
  for each OTU t ∉ C do
    best ← infinity
    for each edge (w, v) in C do
        {test t in every edge of C, and store the optimal positions in TMP_2}
        make a copy NC = (NV, NE) of C
        remove (w, v) from NE
        append t and a vertex x to NV
        append (w, x), (x, v) and (x, t) to NE
        if NC.cost < best then
          empty TMP_2
          push (TMP_2, NC)
          best ← NC.cost
        else if NC.cost = best then
          push (TMP_2, NC)
        end if
    end for
  end for
  while TMP_2 is not empty do {move all trees in TMP_2 to TMP}
    push (TMP, pop (TMP_2))
  end while
end while
return WAG
```

Wagner trees were formalized first by Wagner (1961), with the original algorithm from Kluge and Farris (1969) and Farris (1970). The resulting cladogram is commonly used as a starting point for several other heuristics.

Randomization of taxon addition order is often introduced, which is referred to as Random Addition Sequence (RAS).

Branch Swapping

To traverse the cladogram search space to find an optimal or near optimal solution, several methods are used. The most common of these are the Nearest Neighbor Interchange (NNI), Subtree Pruning and Regrafting (SPR), and Tree Bisection and Reconnection (TBR). SPR and TBR are widely used and more general; therefore NNI will not be considered here, and the set of trees generated in the procedure is a proper subset of those produced in SPR and TBR.

Subtree pruning and regrafting

As a general rearrangement procedure of trees (this and the following description follow Goloboff 1996a), SPR begins with a predefined topology to produce a new one; Algorithm 6.3 presents the procedure, with a pictoral version in Figure 6.2. It consists of the following steps:

1. Prune a clade, creating two subtrees.
2. Insert a new inner node along a branch of either subtree.
3. Attach the remaining subtree as the second sibling of the new inner node.

Formally, given a tree $C = (V,E)$ with root r, choose a node $v \in V$ with incident edges (w, v), (v, x), and (v, y) and an edge (z_1, z_2) such that z_1, z_2 are not descendants of v, the function

$$\chi(C, v, y, z_1, z_2) = C_{spr} = (V_{spr}, E_{spr}) \qquad \text{(Eq 6.1)}$$

such that

$$V_{spr} = V - v + z_{spr}$$

and

$$E_{spr} = E - (v, y) - (w, v) - (v, x) - (z_1, z_2) + (w, x) + (z_1, z_{spr}) + (z_{spr}, z_2) + (z_{spr}, y)$$

creates a new tree C_{spr} with the clade rooted by y pruned and inserted in the edge (z_1, z_2).

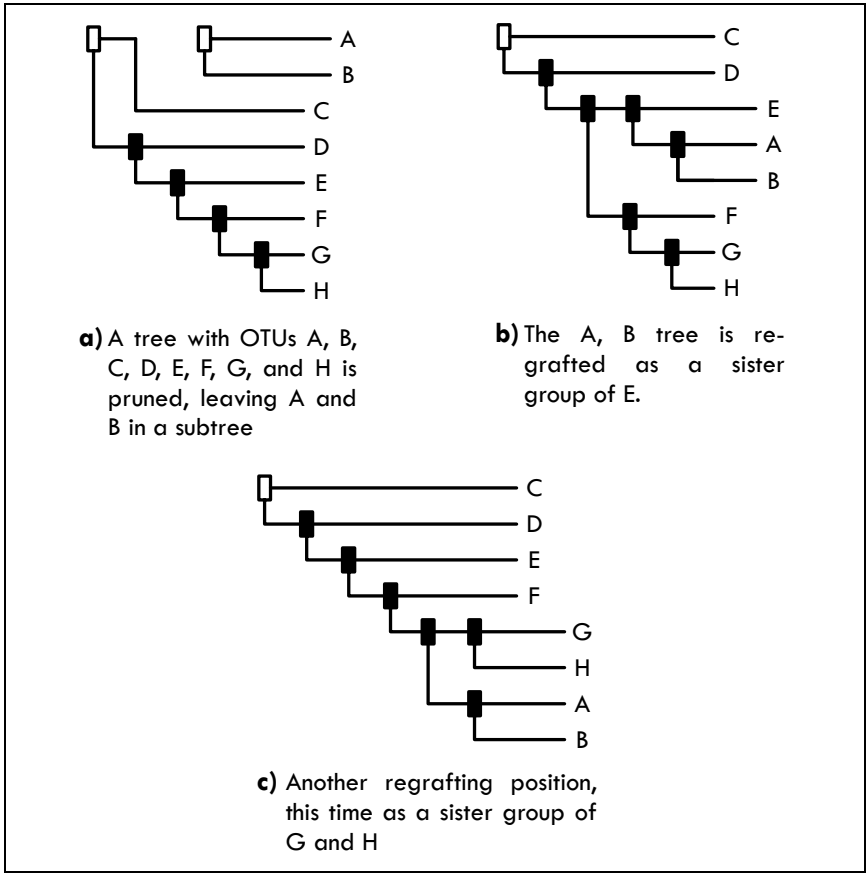

Figure 6.2: Subtree pruning and regrafting procedure example.

Algorithm 6.3 would in theory test all the set of trees t' produced from each tree t in an array T. In reality, a heuristic is used to reduce the search space. This heuristic is defined as the predicates

$$\beta(t', glob\_best\_cost, loc\_best\_cost)$$

and

$$\alpha(t, t', best\_cost)$$

Algorithm 6.3 Subtree Pruning and Regrafting – *SPR(T)*

Require: *BEST_TREES* is a stack of the best trees
Require: *T* is the stack of trees to be rearranged during SPR.
Require: $t = (V, E)$ is a tree with root *r*
  $t \leftarrow$ *pop(T)*
  global_best_cost $\leftarrow$ *t*.cost
  push(T, t)
  repeat
    $t \leftarrow$ *pop(T)*
    for all vertex $v \in V$ with degree 3 **do** {Not the leaves, nor the root}
      for all edge $(a, b) \in E$ not descendant of *v* **do**
        for all vertex *d* direct descendant of *v* **do**
          $t_{spr} \leftarrow \chi(t, v, d, a, b)$ {Equation 6.1 on page 67}
          if $\beta(t_{spr}, global\_best\_cost, local\_best\_cost)$ **then**
            Clear *BEST_TREES*
            local_best_cost $\leftarrow$ t_{spr}.*cost*
            push(BEST_TREES, t_{spr})
            if *local_best_cost* < *global_best_cost* **then**
              global_best_cost $\leftarrow$ local_best_cost
            end if
          end if
          if $\alpha(t_{spr}, global\_best\_cost, local\_best\_cost)$ **then**
            push(T, t_{spr})
          end if
        end for
      end for
    end for
  until *T* is empty
  return *BEST_TREES*

which define whether a tree t' should be included in the list of best cladograms *T* when:

1. t'.*cost* is equal to or better than the cost of the best rearrangement of *t*, so the functions are defined as

$$\beta(t,\ t,\ glob\_best\_cost,\ loc\_best\_cost) = \begin{cases} \text{true} & \text{if } (t.cost < loc\_best\_cost) \\ \text{false} & \text{otherwise} \end{cases}$$

$$\alpha(t,\ t,\ glob\_best\_cost,\ loc\_best\_cost) = \begin{cases} \text{true} & \text{if } (t.cost = loc\_best\_cost) \\ \text{false} & \text{otherwise} \end{cases}$$

2. *t.cost* is equal to or better than any other rearrangement created so far, so the functions are defined as

$$\beta(t,\ t,\ glob\_best\_cost,\ loc\_best\_cost) = \begin{cases} \text{true} & \text{if } (t.cost < glob\_best\_cost) \\ \text{false} & \text{otherwise} \end{cases}$$

$$\alpha(t,\ t,\ glob\_best\_cost,\ loc\_best\_cost) = \begin{cases} \text{true} & \text{if } (t.cost = glob\_best\_cost) \\ \text{false} & \text{otherwise} \end{cases}$$

Note that these functions can also include some user-defined threshold (`slop`) to include a broader range of cladograms in the search space and increase the probability of escaping from local minima.

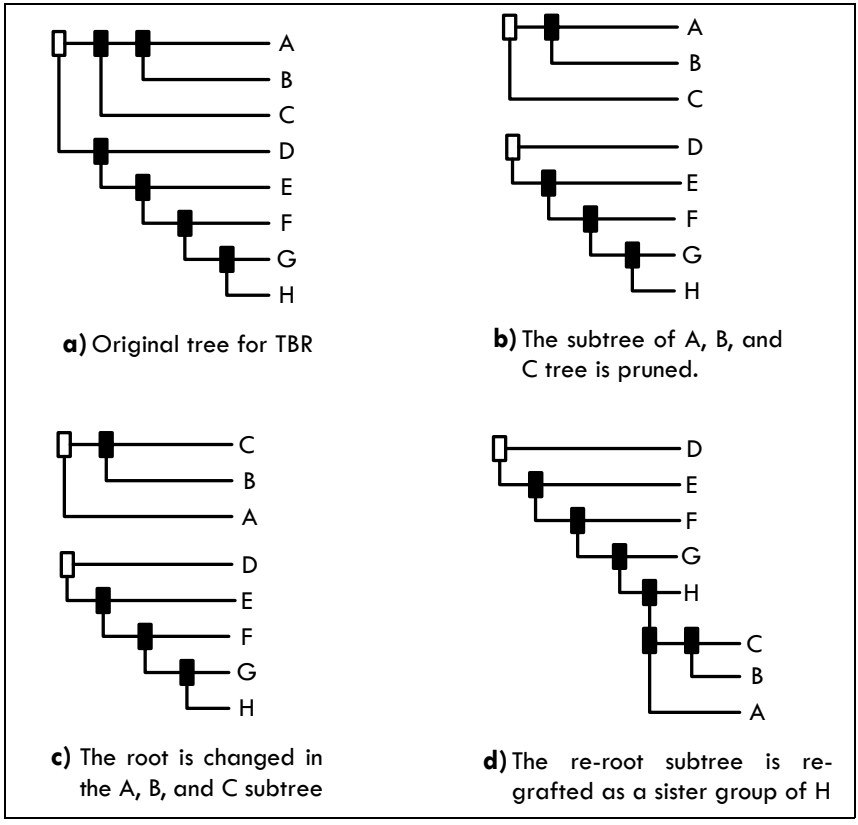

Figure 6.3: Tree bisection and reconnection example.

Tree bisection and reconnection

This procedure is similar to SPR (page 67), but the reconnection of the pruned clade cycles through change in the chosen root.

As a consequence, the trees produced by SPR are a proper subset of TBR; the same result is accomplished in TBR if the root selection gives back the same root defined in the SPR procedure. The algorithm for TBR is a simple modification of the SPR strategy, as shown in Algorithm 6.4 and exemplified in Figure 6.3 on page 70.

Algorithm 6.4 Tree Bisection and Reconnection – *TBR(T)*

Require: *BEST_TREES* is a stack of the best trees
Require: *T* is the stack of trees to be rearranged using TBR
Require: Let $t = (V,E)$ be a tree with root r
  $t \leftarrow$ *pop(T)*
  global_best_cost $\leftarrow$ *t*.cost
  push(T, t)
  repeat
    $t \leftarrow$ *pop(T)*
    for all vertex $v \in V$ with degree 3 **do** {Not the leaves, nor the root}
      for all edge $(a, b) \in E$ not descendant of v **do**
        for all vertex d direct descendant of v **do**
          for all possible root ρ on an edge descendant of d **do**
            $t_{tbr} \leftarrow \chi(t, v, \rho, a, b)$ {Equation 6.1 on page 67}
            if β (t_{tbr}, *global_best_cost*, *local_best_cost*) **then**
              Clear *BEST_TREES*
              local_best_cost $\leftarrow$ t_{tbr}.cost
              push(BEST_TREES, t_{tbr})
              if *local_best_cost* < *global_best_cost* **then**
                global_best_cost $\leftarrow$ local_best_cost
              end if
            end if
            if α (t_{tbr}, *global_best_cost*, *local_best_cost*) **then**
              push(T, t_{tbr})
            end if
          end for
        end for
      end for
    end for
  until *T* is empty
  return *BEST_TREES*

Ratcheting

Ratcheting (Algorithm 6.5) (Nixon 1999) enhances branch swapping. It is an iterative approach that randomly reweighs a set of characters during an SPR or TBR procedure. By iterating through the original and the new random weights, the analysis enlarges the searched space, making it less likely that a result is wallowing in a local optimum.

Algorithm 6.5 Ratcheting during Branch Swapping

Require: C is the stack of initial trees
Require: Let *n* be the number of characters to be reweighed
Require: Let *w* be the factor by which each weighted character will be multiplied
Require: Let *it* be the number of iterations to be performed
Require: Every character c has a weight c.*weight*
Require: A tree rearrangement function *rearrange(C)* (for example, *SPR(C)* or *TBR(C)*) that returns the set of best rearrangements
  for *i* = 1 to *it* **do** {Perturbation of weights}
    C ← *rearrange(C)*
    R ← random set of *n* characters
    for all character c ∈ *R* **do**
      c.*weight* ← c.*weight* × *w*
    end for
    C ← *rearrange(C)*
    for all character c ∈ *R* **do** {Reset the original weights}
      c.weight ← c.*weight* ÷ *w*
    end for
  end for
  return C

Tree Drifting

Like ratcheting, tree drifting (Goloboff 1999, 2002) is a search strategy to avoid local minima by increasing the reachable search space during branch swapping. Instead of reweighting a set of characters, tree drifting swaps suboptimal cladograms according to a predefined optimality threshold (Algorithm 6.6).

Tree drifting is an application of simulated annealing, a general optimization heuristic that temporarily accepts suboptimal solutions as a way to escape from local optima.

Goloboff (1999) described an acceptance probability that is based on the length difference and on the RFD or Relative Fit Difference

(Goloboff and Farris 2001) between a candidate cladogram and the current best cladogram. POY uses a simpler acceptance probability that is based on the cost difference, according to the following rules:

1. Candidate trees that are better than the current best cladogram(s) are always accepted.
2. Candidate trees that are as good as the current best cladogram(s) are accepted with a user-defined probability.
3. Candidate trees that are worse than the current best cladogram(s) are accepted with a probability $\frac{1}{n+c-b}$, where b is the current best cost, c is the cost of the candidate tree, and n is a user-defined factor.
4. The tree drifting algorithm is a slight modification of either SPR (Algorithm 6.3) or TBR (Algorithm 6.4). Basically, the β and α functions are modified to consider the acceptance thresholds as defined above. Let θ, τ be random real numbers ($0 \leq \theta, \tau \leq 1$) in the new function

$$\beta(t,\ glob\_best\_cost,\ loc\_best\_cost) = \begin{cases} \text{true} & \text{if } (t.cost < glob\_best\_cost) \\ \text{false} & \text{otherwise} \end{cases}$$

$$\alpha(t,\ glob\_best\_cost,\ loc\_best\_cost) = \begin{cases} \text{true} & \text{if } (t.cost = glob\_best\_cost) \\ \text{true} & \text{if } (t.cost = glob\_best\_cost) \text{ and } (\theta < driftequalaccept) \\ \text{true} & \text{if } \tau < 1 \div (driftequalaccept + t.cost - glob\_best\_cost) \\ \text{false} & \text{otherwise} \end{cases}$$

Note that only a limited number of suboptimal trees are accepted on each round. The modified algorithm (either the SPR or the TBR according to the user selection) is alternated through multiple iterations (Algorithm 6.6).

Algorithm 6.6 Iterations for the Tree Drifting Algorithm

Require: Let C be a set of trees to be used as a starting point.
Require: Let *it* be the number of ratcheting iterations
Require: Let *drift(C)* be the modified version of the *SPR(C)* or *TBR(C)*
  for $i = 1$ to *it* **do**
    C ← *drift(C)*
    C ← *SPR(C)* or *TBR(C)* depending on the rearrangement method

being used in *drift(C)*
end for

Tree Fusing

This search technique is based on the idea that local optima might be avoided by exchanging subgroups of identical composition but not resolution between different trees (Goloboff 1999, 2002). Tree fusing is an application of a genetical algorithm (Moilanen 1999, 2001) where compatible groups are exchanged among cladograms.

The algorithm implemented in POY is slightly different from Goloboff's original procedure; Algorithm 6.7 shows the method.

Algorithm 6.7 Tree Fusing

Require: Let C be the stack of starting trees for tree fusing
Require: Let *BEST_TREES* be the stack of the best trees found by tree fusing, initially empty
for all tree $t_1 \in$ C **do**
for all tree $t_2 \in T$, $t_1 \neq t_2$ **do**
for all subtrees $\tau_2 \in t_2$ **do**
for all subtrees $\tau_1 \in t_1$ **do**
if τ_1 and τ_2 have the same OTUs but different resolution **then**
swap τ_2 and τ_1 in t_1 and t_2
if do branch rearrangement with tree fusing **then**
TMP $\leftarrow$ *SPR(C)* or *TBR(C)* depending on the rearrangement method selected
end if
Append TMP to BEST_TREES
end if
end for
end for
end for
end for
return BEST_TREES

Static Approximation

Static approximation (Wheeler 2003a) is a cladogram search heuristic designed specifically for use with non-prealigned nucleotide sequences. First, an implied alignment for nucleotides is calculated using a specified optimization algorithm. Second, branch swapping is performed using this implied alignment. If a more parsimonious topology is found, the

nucleotide implied alignment is reevaluated using the specified optimization algorithm and a new implied alignment is calculated.

This algorithm concentrates calculation time on the cladograms that are good candidates for useful solutions: Cladogram searching with a static alignment is an approximation of the best cladogram cost. On the basis of this approximation, most rearrangements need not be analyzed.

Algorithm 6.8 Static Approximation

```
Require: Let t = (V,E) be a starting tree
Require: Let DO(t) be the direct optimization algorithm that returns an
    optimized full implied alignment
Require: Let align be a set of implied alignments currently in use as
    character for each OTU
  quick_cost ← t.cost{Calculate the cost of t using align as character}
  if quick_cost < global_best_cost then
    t ← DO(t) {Calculate a new set of implied alignments}
    if t.cost < global_best_cost then
      Clear BEST_TREES
      align ← The implied alignment of t
    end if
    if t.cost <= global_best_cost then
      push(BEST_TREES, t)
    end if
  end if
  return BEST_TREES
```

The algorithm can be easily implemented by modifying any of the branch rearrangement algorithms in the β and α predicates and the contents of their if statements as shown in Algorithm 6.8.

Parallel Cladogram Searching

POY can take advantage of parallel processing to reduce execution time. At present, parallelism in POY occurs at the cladogram search level. As such, POY divides up the search space and search activities among processors, and centrally manages their coordination. An alternate approach, data parallelism, would divide the data up among processors and exploit parallel architectures. Either or both can be effective. For now, POY parallel algorithms and options are search specific. How to divide search problems can be complex. This is due to the interaction of algorithm, hardware, and data, making general rules for efficient parallel searches difficult to specify. POY strives to be flexible, allowing the user to control the degree of parallelism applied to various stages of a search. On the one hand, a single problem can be divided among all

available processors. Alternatively, search operations can be restricted to single processors, parallelism coming from having many of these problems to do, such as with random replicates. Furthermore, not all algorithms scale perfectly to large systems so that the subdivision of resources might be beneficial. Below we discuss the general approach to parallel execution in POY. Specifics of their use and the decision making involved follow.

The basics

The fundamental scenario for parallel execution in POY consists of an investigator with a series of tasks to perform, each of which is time consuming, for example, a series of random replicate builds to create an initial cladogram. Three strategies can be taken:

- Distribute calculation of each task
- Distribute task calculations in chunks
- Tune parallel processes.

Distribute calculation of each task

The first strategy is to distribute the calculations of each task (that is, each random replicate) among parallel resources, then perform each replicate in turn.

Algorithm 6.9 Master algorithm for parallel replicates

```
for i = 1 to number of replicates do
  for j = 1 to number of taxa to add do
    for k = 1 to number of addition points on tree do
      send addition point and current tree to SLAVE
    end for
    for k = 1 to number of addition points on tree do
      receive cladogram length from SLAVE
      compare and retain lowest cost placement
    end for
    add taxon in best position
  end for
end for
return lowest cost cladogram
```

This form of distributed cladogram building divides the tasks into relatively small portions calculated on the slave processors. The master and slave processes communicate frequently, exchanging information, tasks, and results. This strategy is the default parallel algorithm for many of the cladogram search operations. It applies to random replicate cladogram building, SPR and TBR branch swapping, tree fusing, ratchet-

ing, and drifting. In each case, the task is a cladogram or collection of cladograms to swap, or ratchet to be performed in parallel over all available processors. There are two properties of this scenario that may lead to inefficiencies. First, this strategy imposes a communication burden: if there is little to do in the calculations performed by the slave tasks, most computational effort will be devoted to the mechanics of processing messages.

Algorithm 6.10 Slave algorithm for parallel replicates

```
receive addition point and current tree from MASTER
calculate tree length
tree length to MASTER
```

A second potential limitation is in the amount of work. In order to build an initial cladogram, there are only $2t-5$ places to add the t^{th} taxon to a nascent cladogram. A machine with 100 processors will be largely idle building a cladogram of 50 taxa replicated 100 times. The same machine might be quite well occupied for a data set with 1000 terminals. Similar limits from dependencies on the number of taxa apply to branch swapping and other refinement procedures.

Distribute task calculations in chunks

The second strategy is to distribute tasks in large chunks for calculation as slave processes. In essence, each slave performs an independent, sequential task. Parallelism would then come from distributing the number of tasks to perform.

Algorithm 6.11

```
for i = 1 to number of replicates desired do
   send random replicate to SLAVE
end for
for i = 1 to number of replicates desired do
   receive replicate trees and cost from SLAVE
   retain lowest cost result
end for
return lowest cost result
```

This second approach is specified in POY by using one of several multi commands:

- Wagner builds: `multibuild` (page 283)
- Tree drifting: `multidrift` (page 284)
- Ratcheting: `multiratchet` (page 285)
- Random replicates: `multirandom` (page 285)

Tune parallel processes

The third is an intermediate strategy. Typically, investigators perform a series of searches, each of which would consist of a build step, swapping, tree fusing, ratcheting, and perhaps a final round of branch swapping. A large number of these sequences might be performed, each with a randomly chosen starting point. Using the first strategy, each random replicate could be performed over all available processors. If `multi` commands are specified, they are honored within that part of the search. For example, if 100 build replicates are specified before refinement with `multibuild`, those cladogram builds take place on slaves independently.

Algorithm 6.12

```
receive random replicate from MASTER
for j = 1 to number of taxa to add do
   for k = 1 to number of addition points on tree do
      calculate tree length
      compare and retain lowest cost placement
   end for
   add taxon in best position
end for
send lowest cost tree to MASTER
```

Alternatively, using the second strategy, each complete replicate search could be performed independently and sequentially by slave processes. The 100 builds per replicate would be performed in turn by each slave process as part of its search. The two strategies can be combined and tuned through the use of controller processes using the `controllers` (page 234) command, which assigns slaves to subclusters. Each subcluster controller is seen by the master as a slave, and each slave sees its subcluster controller as its master.

Controllers work with `multirandom` to distribute complete search replicates to each of the controllers' processes as opposed to the individual slaves. Each controller executes its replicate search by distributing work as broadly as possible to its own set of slave processes. POY divides the number of available slave processes evenly among the controllers as subclusters of processes. The degree of parallelism can be adjusted to maximum efficiency by varying the number of controller processes.

Algorithm 6.13

```
for all random replicates desired do
   send random replicate to CONTROLLER
end for
```

```
for i = 1 to number of replicates desired do
   receive replicate cladograms and cost from CONTROLLER
   retain lowest cost result
end for
return lowest cost trees
```

Parallelizing replicates reduces communication with the master process and helps to avoid taxon-number limits of some parallel algorithms. This strategy is often useful in improving the efficiency of cladogram search strategies on large clusters or in situations where large numbers of replicates are desired. All search strategies currently implemented in POY operate in parallel when the `parallel` (page 293) command is specified. This includes initial cladogram construction, SPR and TBR swapping, tree fusing, ratcheting, and drifting. Evaluation techniques such as jackknifing and Bremer support calculations also operate in parallel through cladogram search and swapping algorithms, respectively. Some character-specific edit cost matrices are also determined in parallel (for example, fixed states and search-based optimization).

Algorithm 6.14

```
receive random replicate from MASTER
for j = 1 to number of taxa to add do
   for k = 1 to number of addition points on tree do
      send addition point and current tree
   end for
   for k = 1 to number of addition points on tree do
      receive cladogram length from SLAVE
      compare and retain lowest cost placement
   end for
   add taxon in best position
end for
send lowest cost trees to MASTER
```

Algorithm 6.15

```
receive addition point and current tree from CONTROLLER
calculate tree length
send tree length to CONTROLLER
```


Suggested Reading

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

Gusfield, D. 1997. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge, MA: Cambridge University Press. 534 pp.

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

Moilanen, A. 2001. Simulated evolutionary optimization and local search: introduction and application to cladogram search. Cladistics 17: S12–S25.

Nixon, K. C. 1999. The Parsimony Ratchet, a new method for rapid parsimony analysis. Cladistics 15: 407–414.

7 Evaluation and *a Posteriori* Analysis

In this chapter, different methods implemented in POY to further analyze the results of phylogenetic analyses are examined. In addition to methods implemented directly in POY, methods implemented in other programs are discussed that can be used in concert with POY. Most of the methods discussed are used commonly (for example, Bremer support and strict consensus trees). However, some are new, modified, or have not been applied broadly.

Currently, there is a debate in the literature concerning the scientific value of extensive data exploration of the variety described herein (for example, see Giribet 2003; Grant and Kluge 2003 and references cited within), which we will not address here. Our approach in this chapter is catholic: we present the diversity of *a posteriori* analyses that can be performed in POY and leave the reader to determine the utility of each approach.

Examination of Inferred Homologies

In POY, cladogram preference is governed by the economical optimization of all character transformations (for example, transitions, transversions, nucleotide insertions and deletions, phenotypic changes, gene duplications and losses, and gene rearrangement). Thus, POY not only provides a phylogenetic hypothesis for the included taxa, but it also pro-

vides a database of nucleotide, genomic, and phenotypic evolution. In fact, POY actually goes one step further: it provides a predictive framework for genomic organization. For example, the recently implemented methods for analyzing genomic rearrangements, insertions, deletions, duplications, and rearrangement clearly provide the first step toward automating the process of genome annotation using a historical, phylogenetic approach instead of the most commonly applied method: similarity.

Cladogram diagnosis: transformation series and hypothetical ancestors

The command `diagnose` (page 238) outputs information about inferred character transformations at the level of nucleotides, loci, behavior, etc. over the cladogram. The order of the characters presented in the output will follow the order that the user lists the input data files in the command line. Thus, it is often convenient to put phenotypic characters first in the POY command line because these characters will then have the same numbering system as in the Hennig86/Nona data file format and their character number will remain unchanged if multiple, equally parsimonious alignments result from the analysis.

The command `diagnose` (page 238) can be included in the same POY command line used to perform the search. Alternatively, a topology found in another analysis can be input and diagnosed separately. To ensure consistency of calculated costs and optimizations, it is important to employ the same optimization algorithm and transformation costs for diagnosis as were used in the cladogram searching (unless the goal is to determine the effect those choices have on cladogram cost and inferred nucleotide homologies).

The `diagnose` output includes four sections:

- Topology and cost of tree to diagnose
- Minimum and maximum branch costs
- A list of character changes for each branch, giving both the ancestral and derived states and specifying which transformations are definite (that is, unambiguous) and the class of transformation (for example, indel, transversion, etc.)
- Hypothetical ancestral reconstructions organized as ordered sequences with ambiguously optimized nucleotide states reported as IUPAC symbols plus additional codes to accommodate indels—note that these hypothetical ancestral sequences are not the same as those generated by the command `printhypanc` (page 303), which prints hypothetical ancestral sequences with arbitrarily resolved ambiguous positions that can be used in search-based optimization (page 55).

If more than one cladogram is diagnosed, these four sections will be generated for each cladogram consecutively.

The cost of the diagnosed cladogram can differ from the cost of the cladogram that results from the search when the input topology contains polytomies introduced by consensus procedures or collapsing of zero-length branches. POY arbitrarily resolves input polytomies for diagnosis. The resulting cost and character changes differ from the original cladogram when the new binary version differs from the original binary cladogram. If the cladogram to be diagnosed is derived from a previous POY analysis, it is important that the binary version of the cladogram (reported in the POY output file) be input for diagnosis. In addition, the same root terminal used in the search must be specified. Due to the heuristics of the direct optimization algorithm, alternative rootings of the same topology might imply different lengths and character transformations. The effect of rooting is generally diminished when the more thorough iterative pass algorithm is used, but even then this is a potential source of disagreement.

The character change list of the diagnosis output is presented in the following order:

- Ancestral taxon
- Descendant taxon
- Character (phenotypic, DNA fragment, or locus)
- Position (place within a fragment of DNA numbered from 0 in the descendant fragment)
- Ancestral character state
- Descendant character state
- Type of change (optimization dependent or definite).

The columns of the table are formatted in tab-delimited columns that can be imported into a spreadsheet program for easy visualization and manipulation, as has been done in the table below.

Ancestor	Descendant	Character	Position	Ancestral state	Descendant state	Type of change	Definite
HTU12	DBA2J	[0]	[2]	C	T	Ti	*
			[4]	G	T	Tv	*
			[6]	G	A	Ti	*
			[8]	A	G	Ti	*
			[10]	T	C	Ti	*
			[32]	A	N	ABC	

Alternatively, you can use `grep`, `sort`, and `uniq` programs on Unix machines or searching tools in text editors to find information of interest.

Except in the case of prealigned sequence data or matrices of phenotypic data, the position number in the character change list is not based on (and may differ from) the column number in an implied alignment. Cladogram diagnosis is performed using the unaligned sequence data, with the implied alignment calculated as a result of optimization.

Implied Alignments: Visualization of Nucleotide Homologies

Although POY can analyze prealigned sequence data, where nucleotide homology has been determined prior to the phylogenetic analysis, it is specifically designed to avoid aligning sequences as a separate step in homology assessment. In fact, direct optimization, as originally presented, did not provide a means of visualizing nucleotide homology as aligned characters. However, recognizing that alignments may be useful as visual representations of nucleotide homology and as standard taxon-by-character matrices that can be input into programs such as Winclada (Nixon 2002) and TNT (Goloboff et al. 2003b), the command `impliedalignment` (page 263) generates an alignment by optimizing the data on a cladogram (Wheeler 2003b). This command takes the dynamic homologies established through direct optimization and traces them back through the cladogram, linking the unaligned sequence positions through the respective transformation series.

Although they resemble standard alignments visually, implied alignments are derived from a radically different procedure and may return a different result than one would obtain by inserting the same topology as a guide tree and generating an alignment under the same parameters in, for example, Clustal W. Whereas Clustal W aligns independent, homoplastic insertions of the same nucleotide in a single column, POY assigns them to different columns because they are not homologous and are, therefore, not parts of the same transformation series. In addition, as discussed and illustrated by Wheeler (2003b), the implied alignment for a given cladogram is not necessarily unique. The implied alignment for each fragment is reported separately in the order in which they appear in the POY script, with each assigned a character number (initialized at 0). Figure 7.1 and Figure 7.2 illustrate implied alignment.

The command `phastwincladfile` (page 293) also generates an implied alignment, but saves it as a taxon-by-character matrix in Hennig86/Nona format with the optimal topologies appended below the matrix. If multiple equally parsimonious alignments result from the analysis, `phastwincladfile` only gives the implied alignment of the first cladogram. To obtain a Hennig86/Nona matrix for each cladogram, either convert each implied alignment manually or input each topology in a separate POY script with `phastwincladfile`.

These matrix files can then be opened directly in standard phylogenetic software (such as Nona, Winclada, or PAUP*) for additional visualization (for example, visualization of character transformation) or searching.

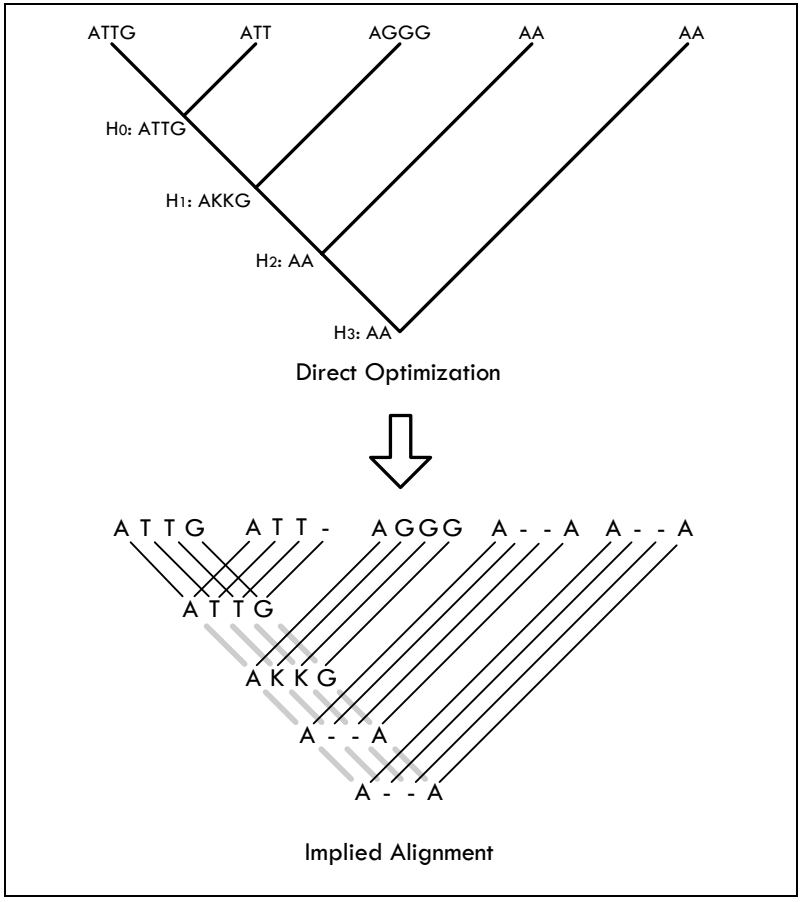

Figure 7.1 Illustration of implied alignment. Direct optimization determines hypothetical ancestors (H_i). The minimum path for nucleotides is calculated from hypothetical ancestors observed sequence by inserting gaps.

A	T	T	G
A	T	T	-
A	G	G	G
A	-	-	A
A	-	-	A

Figure 7.2 Multiple sequence alignment implied by direct optimization.

It may be tempting to use an implied alignment (or Hennig86/Nona matrix) as the basis for subsequent phylogenetic searching using different optimality criteria or parameters, or to use the implied alignment of a ribosomal gene in conjunction with other prealigned data. However, this practice is inconsistent because the optimal homologies depicted by the implied alignment are specific to a given cladogram and are almost certain to be suboptimal for any other topology. Bear in mind that an implied alignment is really just a means of visualizing nucleotide transformation series, and the optimal set of nucleotide homologies for a given data set is topology and parameter specific.

Implied Alignment Algorithm

The algorithm for implied alignment is that of Wheeler (2003a). The method is not a traditional multiple sequence alignment procedure, but yields an organized synapomorphy scheme that is based on a specific cladogram.

The procedure requires an optimized cladogram with internal nodal sequences determined. These HTU sequences can be determined by direct optimization, iterative pass optimization, fixed-state optimization, or search-based optimization.

The algorithm is linear in number of taxa and quadratic in sequence length.

Algorithm 7.1 Get the number of inferred homologies in the nucleotide sequence, and assign to every nucleotide in a sequence a homology number *Numerate* (*ancestor, child, ancestor_aligned, child_aligned, counter*)

Require: ancestor and child are two connected nodes *a* and *b* where *a* is the direct ancestor of *b*
Require: *ancestor_aligned* and *child_aligned* are the aligned nucleotide sequences of length *l ancestor* and *child*

$j \leftarrow 0$
$k \leftarrow 0$
for $i = 0$ to $l - 1$ do
  if $ancestor\_aligned_i = gap$

```
        child_homologies_i ← counter
        j ← j + 1
        counter ← counter + 1
      end else
        if child_aligned_i <> gap
          child_homologies_j ← ancestor_homologies_k
          j ← j + 1
        end
      k ← k + 1
    end
done
return counter
```

Algorithm 7.2 Assign the homologies numeration to every HTU in preorder *Numerate_preorder* (*node,counter*)

Require: if node is the root, the array homologies of length *j* is initialized
numbering homologies$_k$ *k*, $1 \leq k \leq j$
ancestor,child ← *Align_sequences (node.chars node.left.chars)*{Algorithms 5.7, 5.8, and 5.9 Chapter 5}
counter ← *Numerate* (*node, node.left, ancestor, child, counter*)
ancestor,child ← *Align_sequences (node.chars node.right.chars)*
counter ← *Numerate* (*node, node.right, ancestor, child, counter*)
counter ← *Numerate_preorder (node.left, counter)*
counter ← *Numerate_preorder (node.right, counter)*
return *counter*

Algorithm 7.3 *Align_sequences()*

Require: a cladogram C with root *r* and *n* leaves
counter = *Numerate_preorder* (*r*,0)
(me, thee) ← Direct_pairwise (*ancestor.sequence, descendant.sequence, E*)
M is a matrix of size *n* x *counter*
for each leaf *t* in C **do**
  j ← number of nucleotides in *t.chars*
  for *i* = 0 to *j* – 1 **do**
    position ← *t.homologies*$_i$
    $M_{t.position}$ ← *t.chars*$_i$
  done
done
return *M*

Evaluation of Multiple Optimal Cladograms

Although researchers may hope that the available evidence unambiguously supports a single, optimal cladogram, empirical data are often not so generous and multiple optimal hypotheses are found. Those clades that are unambiguously supported by the available evidence are those that are common to all optimal cladograms: they can be efficiently summarized by the strict consensus representation. Although the majority rule consensus is primarily used for depicting group frequencies from jackknife resampling, it can be generated for any set of cladograms. For example:

- `poystrictconsensustreefile` (page 301) writes the strict consensus tree in POY parenthetical notation to a specified file.
- `printtree` (page 304) used with `plotfile` (page 294) writes the best cladograms and their strict consensus at the end of a POY run to file `poy.tree`. The output file is specified by `plotfile`.
- `plotmajority` (page 296) determines whether and how `printtree` plots the majority rule consensus tree. The command has three modes: the majority rule consensus tree is not plotted (this is the default setting), the majority rule consensus tree is plotted without clade identification numbers, and the majority rule consensus tree is plotted with clade identification numbers (each branch is labeled with the identification number and the clade frequency).

No other consensus calculations (such as Adams or combinable components) are currently supported directly in POY, although these can be calculated by generating a phastwincladfile that is opened in another phylogenetic program.

Finally, some authors (for example, Carpenter 1988) have recommended *a posteriori* weighting as a method of selecting a subsample of one or more cladograms from a larger number of equally most-parsimonious cladograms. Successive approximation weighting (Farris 1969) has been used for this. It was not, however, originally intended for this purpose. POY includes a modified version of Goloboff's (1993b) implied weighting (`goloboff` on page 258) that maximizes total fitness during cladogram search, by altering fragment costs according to their relative fit. This differs from Goloboff's method, as implemented in Pee-Wee (Goloboff 1996b), which alters the costs of each individual character.

Partitioned Analysis and Evaluation of Partition Incongruence

One of the basic premises underlying the development of POY is that combined analysis (also referred to as simultaneous analysis and total evidence) is optimal. Nevertheless, many advocates of total evidence

analysis are also interested in exploring the behavior of particular data partitions. This is most commonly done by analyzing each partition of interest in isolation and inspecting the results visually or quantifying the partition incongruence with one or more of a variety of methods.

The most convenient means of carrying out these analyses is to use a single script file that contains all the necessary POY scripts used to analyze each data partition, with the scripts set to run consecutively. For separate, partitioned analyses, each partition of interest must be in a separate data file, that is, partitions formed by inserting pound signs (#) in the sequences of a single file cannot be analyzed separately unless sections are deactivated using an astersik (*).

Common measures of partition congruence include the Mickevich–Farris (1981) Extra Steps Index, given by Equation 7.1.

(Eq 7.1)

$$ILD = \frac{Length_{\text{combined data set}} - \sum Length_{\text{individual data sets}}}{Length_{\text{combined data set}}}$$

Mickevich–Farris Extra Steps Index is a character-based measure of partition congruence. This index can be rescaled by using a denominator that reflects the difference between the maximum tree length from the combined data (bush) and the minimum (sum of the individual lengths), a measure that has been termed the RILD (Rescaled ILD) (Wheeler and Hayashi 1998).

(Eq. 7.2)

$$RILD = \frac{Length_{\text{combined data set}} - \sum Length_{\text{individual data sets}}}{\sum \text{Max Length}_{\text{individual data set}} - \sum Length_{\text{individual data sets}}}$$

Both the Mickevich–Farris Index and the RILD require calculating tree lengths for individual and combined partitions (Figure 7.3). In the case of the RILD, an extra cladogram search is required to calculate the Max $\text{Length}_{\text{Combined}}$ using the command `weight` (page 329), which assigns a weight to the partition under regular analytical searches (`-weight -1`).

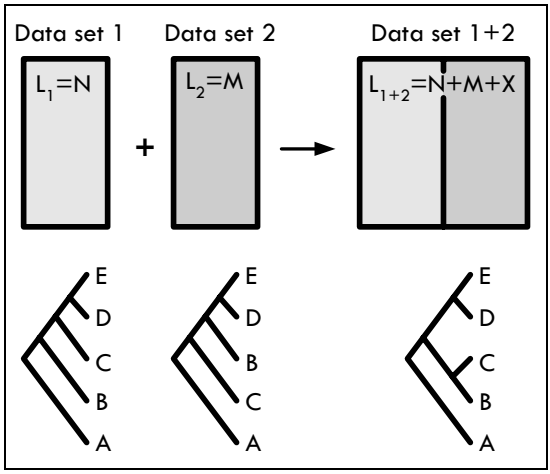

Figure 7.3 Homoplasy (X) component of combining data partitions.

Common topological measures of partition incongruence include percent shared groups and percent compatible groups (Wheeler 1995), and the Topological ILD (TILD; Wheeler 1999b). The latter is determined by generating a group inclusion matrix for optimal topology from the combined data set and each partition, then calculating the Mickevich–Farris Index that results from the analysis of those matrices.

The TILD_N is based on the TILD but measures the ratio of how much topological homoplasy (incongruence) there is versus how much there could be in the worst case (complete bush). That is, it corrects for the effect of unresolved input topologies by dividing by the maximum possible length of the combined data sets minus the sum of lengths of the individual data sets.

(Eq. 7.3)

$$TILD_N = \frac{Length_{\text{combined data set}} - \sum Length_{\text{individual data sets}}}{\sum \text{Max Length}_{\text{individual data set}} - \sum Length_{\text{individual data sets}}}$$

Few methods are available that examine the behavior of partitions directly through *a posteriori* analysis of total evidence results (Grant and Kluge 2003). However, one simple approach is to optimize each partition separately on the optimal total evidence topology to determine cost, distribution of transformations, and tree statistics for that partition. This can be done in POY by inputting the optimal topology and diagnosing it with each partition of interest or by opening the

Hennig86/Nona files in Winclada and activating/deactivating the characters of each partition.

Jackknife Resampling

A common method for evaluating nodal support is to use a resampling technique such as the parsimony jackknife (Farris et al. 1996; see also Goloboff et al. 2003a). POY can also implement jackknifing using maximum likelihood as the optimality criterion. Jackknifing is invoked by the command `jackboot` (page 266). It generates the N submatrices where N is set using the command `replicates` (page 311) by randomly resampling the original data with a specified deletion probability for any site of e^{-1} (approximately 0.37) by default, following Farris et al. (1996) and performing a full analysis of each submatrix, giving N pseudo-replicate analyses.

A strict consensus is calculated automatically for each pseudo-replicate and the majority-rule consensus is calculated for the N pseudo-replicates, all of which are included in the default output file. The group frequencies on the cladogram are the jackknife percentages. The following is a typical jackknifing script using parsimony under direct optimization for 1000 pseudo-replicates:

```
poy datafiles -jackboot -replicates 1000
-jackfrequencies all -jacktree
```

The default output can be modified with the commands `jackoutgroup` (page 270), `jackpseudoconsensustrees` (page 270), `jackpseudotrees` (page 271), `jacktree` (page 271), `jackftrees` (page 269), and `jackwincladefile` (page 272). Cladogram search strategies that differ and are less aggressive than the original search for the most parsimonious trees often lead to jackknife support values that are incorrect and unpredictable (that is, the values may be lower or higher than expected).

Jackknifing Algorithm

Parsimony jackknifing was described by Farris et al. (1996; see also Farris 1997) as a measure of character support for clades. The original description was for nonadditive characters. In POY, jackknifing is performed over all characters and types present in an analysis. Standard jackknifing applies a delete probability (usually e^{-1}) to each character. Given the dynamic homology regime for several character types, such deletion is impossible, since homology reconstruction would be affected as well as cladogram cost. As a result, POY reweighs the deleted fraction of characters to zero.

Jackknifing then proceeds through specified search options for each replicate. The POY specific function is the calculation of jackknife

cladogram costs for direct optimization characters. As mentioned above, the nucleotides are not deleted, but HTU costs are altered according to the jackknife values.

Algorithm 7.4 Get Jackknife delete array

```
delete_array ← allocate array of array_size Booleans
for i ← 1 to array_size do
  if Random() < (rand_max × fraction) then delete_array_i ← false
  else delete_array_i ← true
done
return delete_array
```

Algorithm 7.5 Modify HTU cost

```
new_cost ← old_cost
for i ← 1 to Length(left_descendant) do
  if (not delete_array_(i mod (Length(delete_array)))) then
    new_cost ← new_cost − σ_left_descendanti , right_descendanti
done
return new_cost
```

Bremer Support

Bremer support (branch support, decay index; Bremer 1988, 1994) assesses the decisiveness of corroboration of a given clade by comparing the length of the optimal cladograms with those of suboptimal solutions to determine how much worse (that is, more costly) a cladogram must be for that clade to be absent.

Bremer values in POY can be calculated for all groups simultaneously by TBR swapping under a set of constraints designated by a group inclusion matrix, as specified by the commands `bremer` (page 220) and `constrain` (page 233). A group inclusion matrix (one character for each member for a clade; Farris 1973) can be output using `poystrict-consensuscharfile` (page 300). These simultaneous Bremer support calculations might lead to overestimates of group support. This is because POY calculates Bremer values based on a single round of TBR branch swapping. Relevant cladograms more than a single TBR swap away will not contribute to the Bremer values calculated in this fashion.

Partitioned Bremer support values (Baker and DeSalle 1997) can be calculated in POY, as outlined by Hormiga et al. (2003).

Bremer Support Algorithm

There are two ways to determine Bremer support values in POY. The first is to perform a series of searches, where each group supported on the examined cladogram is constrained not to occur in the result (refer to `constrain` on page 233 and `disagree` on page 238). The Bremer support value for that clade is then the difference between the unconstrained result and the constrained result. For a fully resolved cladogram of n taxa, $n - 2$ searches would be required. A benefit of this approach is that these searches can be thorough, hence the values accurate. The drawback is that this can be a very time-consuming approach.

A second, less conservative approach simultaneously calculates Bremer values for all nodes on the cladogram (refer to `bremer` on page 220, `constrain` on page 233, and `topology` on page 323). This method is fast and approximate and described below. Clades with marginal support would profitably be tested with the more exhaustive constrained search procedure.

The `bremer` command in POY performs an SPR (`bremerspr` on page 221) or TBR (`bremer` on page 220) swap on an input cladogram to be tested. Shortcuts in cladogram cost calculation (for example, direct optimization and fixed-states characters) are not used, since such Bremer values would not be comparable to other cladogram costs.

As an alternative to the `bremer` command, Bremer support values can be estimated for each node separately. This approach is much more time-consuming, but it is a more rigorous implementation and often increases the accuracy (that is, lower values) of Bremer support values (employed in Frost et al. 2001b). To take this approach, the group inclusion matrix must be split manually into individual matrices of one character each (that is, one file per group). Using the commands `disagree` (page 238) and `constrain` (page 233), use each matrix as a constraint file in separate POY runs for each group to search for the optimal cladogram that does not include the specified group. These separate POY runs can be executed consecutively in a single batch file. Bremer values are then calculated for each group as the difference between the cost of the particular constraint cladogram and the original, optimal cladogram.

Algorithm 7.6 Get Bremer array

```
cladogram_cost ← Optimize (T, Obs)
for (all local_cladograms in TBR or SPR of T swap round) do
    local_cost ← optimize (local_cladogram, Obs)
    for (all clades in Grps) do
        if (clade not in local_cladogram) then
```

```
        bremer_array_clade ← (bremer_clade_clade, local_cost –
          cladogram_cost)
    done
done
return bremer_array
```

Fit Indices

Since POY deals primarily with dynamic homology assessment, the only fit index that is clearly interpretable is the report on the optimality criterion—that is, the cladogram cost or length or the likelihood value. The ensemble consistency index (CI; Kluge and Farris 1969) and retention index (RI; Farris 1989) have little meaning in the context of dynamic homology. To calculate conceptually equivalent fit indices, the chosen optimization algorithm must be applied to find the lowest cost optimization for the worst possible topology and the highest (that is, an unresolved bush). The minimum cost for a fragment can be estimated by a search on that character alone, the maximum by applying `-weight -1` (refer to `weight` on page 329) and, in essence, searching for long trees.

Coarse, nonsearch estimates of modified CI and RI values are reported in the output when the command `indices` (page 263) is specified.

Parameter Sensitivity Analysis

The results of any analysis depend on assumptions. Among the many analytical assumptions an investigator must make are those relating to the relative costs assigned to transformations. In molecular systematics this is most often assessed in terms of indel, transversion, and transition costs. As Morrison and Ellis (1997) showed, phylogenetic results can be even more dependent on these values than they are on the method of analysis. Wheeler (1995) proposed a method of sensitivity analysis to explore the effect of varying these and other parameters, such as differential gap opening and gap extension costs (although Wheeler only examined transition, transversion, and indel costs).

The result of this kind of sensitivity analysis can be presented in several ways:

- As a value reflecting the percentage of parameters under which a given node is recovered (Giribet et al. 2001);
- More visually, as a so-called "Navajo rug" (Wheeler 1995; Janies 2001; Giribet 2003), a graphic plot where the parameter space is represented as a grid with parameters represented in two (or more) axes (typically indel-to-change ratio on one axis and transversion-to-transition cost ratio in the second axis) and color coding to designate

monophyly or nonmonophyly of a given group (see Figure 7.4 on page 95);

- A consensus (strict or majority rule) of the clades recovered under the different parameter sets (for example, Edgecombe et al. 2002; Schulmeister et al. 2002);
- As a congruence surface (Wheeler 1995).

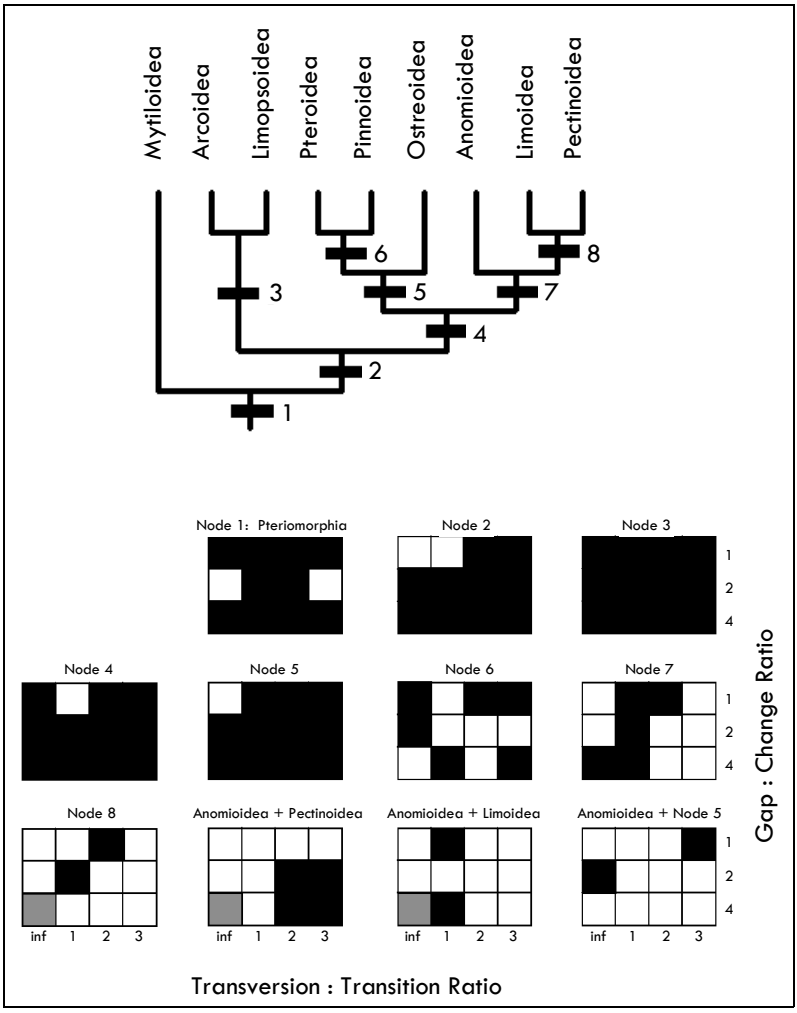

Figure 7.4 Illustration of a "Navajo rug" representing a sensitivity analysis.

In addition to using sensitivity analysis simply as a data exploration tool to compare results, it has also been used to select a parameter set and corresponding cladogram as optimal. This is accomplished by calculat-

ing the partition incongruence under each of the parameter sets of interest, using either topology- or character-based measures. In this framework, the parameter set that minimizes incongruence among partitions (however incongruence is measured) is considered optimal.

The values of the congruence analysis can also be plotted in different ways, such as congruence surfaces (see Wheeler 1995) or color gradient surfaces (see Janies 2001; Frost et al. 2001a).

All these congruence measures depend on the specified partitions. In order to have a truly partition-free metric, this dependence must be removed. The Meta-Retention Index (MRI), Farris's Retention Index calculated over all characters, has been proposed as a measure for choosing trees (Wheeler et al. 2006).

$$MRI = \frac{\sum Length_{fragments} - Length_{\text{combined data set}}}{\sum Length_{fragments} - \text{Minimum Length}_{fragments}} \qquad \text{(Eq. 7.4)}$$

where

fragment = character for fixed homology statements (for example, morphology)

In maximum likelihood analysis, the absolute likelihood value can be used to choose the best topology even when analyzed under different models. However, likelihood practitioners often prefer to rely on a hierarchical table of models from more simple to more complex and use a likelihood ratio test (or other criteria such as the Akaike Information Criterion) to choose a model without making it too complex (Huelsenbeck and Crandall 1997; Posada and Crandall 1998). A hierarchical table of models such as that implemented in MODELTEST (Posada and Crandall 1998) has not been implemented in POY, but topology-based partition congruence can be used to select models.

Suggested Reading

Bremer, K. 1988. The limits of amino acid sequence data in angiosperm phylogenetic reconstruction. Evolution 42: 795–803.

Farris, J. S. 1989. The retention index and the homoplasy excess. Systematic Zoology 38: 406–407.

Giribet, G. 2003. Stability in phylogenetic formulations and its relationship to nodal support. Systematic Biology 52: 554–564.

Goloboff, P. A., J. S. Farris, M. Källersjö, B. Oxelman, M. Ramírez, and C. A. Szumik. 2003. Improvements to resampling measures of group support. Cladistics 19: 324–332.

Grant, T. and A. G. Kluge. 2003. Data exploration in phylogenetic inference: scientific, heuristic, or neither. Cladistics 19: 379–418.

Wheeler, W. C. 1995. Sequence alignment, parameter sensitivity, and the phylogenetic analysis of molecular data. Systematic Biology 44: 321–331.

8 Parallel Processing

Clusters and Parallel Processing

Phylogenetics is computationally demanding. Much attention has been paid to heuristics of cladogram search when the user has a single processor. However, comparative genomic data sets are growing at a staggering pace. Fortunately there is ample opportunity to exploit the power of parallelism in phylogenetics by tuning cladogram search heuristics on a computer with multiple CPUs or a distributed network of processors.

In this chapter, the allocation of processing resources in POY to achieve parallel efficiency is discussed. With appropriate algorithms, a parallel computer performs multiple operations concurrently on separate CPUs. In contrast, a single CPU machine proceeds sequentially, performing each operation in turn. The decrease in calculation time in a parallel computer depends on

- the efficiency in which data and computational work can be divided into smaller problems and assigned to multiple processors.
- the extent to which algorithms depend on intermediate solutions in order to move on to further operations.

POY is run on supercomputers by groups at university and government laboratories around the world. POY was originally designed to take advantage of Beowulf-class computing clusters with relatively low band-

width and high latency interconnects. As a result, there are many options the user can specify to tune parallel algorithms to take advantage of and minimize the shortcomings of a particular machine.

Granularity, Scaling, Overhead, and Load Balancing

Granularity indicates the degree to which complex problems are broken down into smaller pieces for parallel computation. Fine-grained approaches assign small amounts of work to each processor and require a high degree of communication among processors to coordinate the calculations. Alternatively, coarse-grained approaches assign larger chunks of work and require commensurately less interprocessor communication. Since there is more communication in a fine-grained approach, loads can be balanced more evenly among processors than in more coarsely grained algorithms. The efficiency of parallel calculations depends largely on the communications and synchrony requirements of the tasks that are assigned.

Overhead consists of bookkeeping by the master processor to manage the message passing, the actual time of interprocessor communication, and any additional work that might need to be done to break the problem into smaller tasks. Because phylogenetic searches have unpredictable running times, balancing the processing load among slave processors is a difficult but not insurmountable problem. Individual hardware and user environments will determine the best approach to load balancing.

Scaling

Ideally, calculations would scale in direct proportion to the number of processors devoted to a problem. This is rarely the case. With perfect linear scaling, one expects with parallel execution over 100 slave processors a 100-fold speedup as compared to sequential execution time. However, there are elements of the search procedure that are not parallelized as well as overhead, which prevent perfectly linear speedup.

Overhead

A search procedure might require 100 separate cladogram searches on a single processor. The command `replicates` (page 311) performs random cladogram building followed by swapping in sequence a specified number of times, collects the best results for final refinement, and outputs the best cladogram. If the mean execution time of a single search is 150 seconds, it will take approximately 15,000 seconds to get through 100 searches on a single processor. However, on a cluster of 100 processors dedicated to a single user using the commands `parallel`

(page 293) and `replicates` (page 311) this analysis would be executed in only one third to one twentieth the time, instead of the expected reduction of close to 100.

Using these commands, the 100 processors work collectively on each search in a sequence. Building each cladogram can be partitioned out to many processors, but this is a fine-grained operation. For anatomical characters, the cost of computation for adding a single taxon to a subtree is small and scales linearly with the number of characters. The cost of computation for adding a single taxon is simply a cladogram length calculation resulting from comparing the character states of the taxon to be added to the union of character states for ancestor and descendants on the branch to be added (Goloboff 1996a). When building a cladogram in parallel, the cladogram length for each taxon addition must be communicated from each slave back to the master for each addition point for every taxon in a data set. As a result, the ratio of communication relative to computation is very high and the parallel computer spends much time shuttling instructions and intermediate results back and forth, lowering overall efficiency. Luckily more coarsely grained approaches are available.

Load balancing

In phylogenetic analysis, an obvious coarse-grained parallel approach is to evaluate random searches concurrently and independently on different processors. An alternative strategy for the example problem of 100 searches on a parallel computer is to use the commands `parallel`, `replicates`, and `multirandom` (page 285) (refer to "Parallel Environments" on page 114). These commands execute 100 searches that run concurrently because `multirandom` specifies that the work be allocated as one replicate per processor. The 100 searches complete cladogram building and swapping within each replicate on its own processor. The slave jobs are independent of any need for communication save for the master processor sending out the work and collecting the final results. This run might complete with a 75-fold rather than the 100-fold speedup expected from a 100 processors. It is common for the random replicates to have a wide distribution of execution times. This is due to the path the heuristic searches take, the number of cladograms found, and other factors. The way this search was designed, the master must wait for all searches to be returned before final refinement can begin. Many slave processors that have finished their replicates must sit idle until the slowest replicate completes.

There are many ways to address asynchrony of execution times in POY. First, when conducting a `multirandom` search. building and swapping are conducted on each processor and time spent swapping is probably contributing most significantly to the variance in execution time. The

independent initial random addition sequence cladogram builds can be performed on single processors, while swapping is accomplished over several. Such a run on 100 processors is specified using the commands `parallel`, `replicates`, `multibuild` (page 283), and `buildsperreplicate` (page 223). The results might improve the speedup to 88-fold, but not the full linear 100-fold speedup.

Strategies that employ the commands `multibuild` and `multiratchet` (page 285) can exhibit nearly linear speedup. However, the efficiency of swapping depends on the size of the cluster and the hardware used (Janies and Wheeler 2001). In this case, the previous strategy can still be described as coarse-grained because builds are done on a one-replicate-per-processor basis. However, `multibuild` specifies builds without terminal swapping. On their own, builds tend to have a low variance of execution times (on a homogeneous cluster), and thus most of the results would come back within about 10% of each other in a homogeneous cluster. Once all builds were collected, the swapping proceeded on the best cladogram (or cladograms if there are many at minimal length) in parallel.

While performing branch swapping using SPR, each taxon and each internal node are removed from a candidate cladogram (but not derooted) and placed back at all other possible internodes. In parallelization of SPR, various pairs of subtrees resulting from prunings are passed to individual slave processors where the optimality value for each regraft is calculated. Once a set of regraftings is assessed, the best topologies and cladogram lengths are reported back to the master. SPR involves moderate communication-to-computation ratios. For SPR in POY on a cladogram with t taxa, there are $2t - 4$ branches to prune off the cladogram. A load-balancing problem arises in placing the subtrees back on the cladogram. The larger the subtree (number of descendents) the smaller the set of internodes that remain as possible regrafting points. Thus granularity of swapping is variable. The number of candidate replacements varies from 3 to $2t - 3$, thus producing a large spread of execution times. In TBR, the number of subtrees pruned is smaller than SPR because prunings occur only along internal branches. However, the number of regrafts in TBR is greater than in SPR because subtrees are derooted and all possible rerootings are tried. The number of candidate replacements in TBR is the product of the number of branches (minus one replacement point that represents the original tree). Thus a load balancing problem occurs in TBR, but it is limited by the number of prunings rather than regrafts as in SPR.

In both modes of branch swapping the shortest cladograms are kept and further rounds of rearrangements are tried on the shortest cladograms until no shorter candidate cladograms are found. It is not possible to predict how many swapping rounds will be needed, as this is

tied to heuristics, data set structure, and length of initial cladogram builds. The speedup of swapping has been empirically determined to be limited to 32 processors on a variety of data sets for a large cluster with 100-Mbps Ethernet interconnects (Janies and Wheeler 2001). This should be on specific cluster/multiprocessor hardware because SPR and TBR branch swapping are used directly or underlying many refinement techniques (for example, ratcheting, drifting, tree fusing). Thus, although the strategies to balance load described below are made with swapping examples, they are broadly applicable.

Efficient Parallel Searches

There are many ways to efficiently allocate resources on a large cluster in single and multiuser environments. One of the most popular is to use scheduling software that acts as a front-end filter to permit jobs into the cluster based on characteristics of the user and the work requested. This type of resource control will be unique to each cluster and the practice of its administrators.

POY users have the ability to dynamically create and modify subclusters of slave processors within the large cluster to take advantage of the differences in optimal behavior of cladogram building and swapping procedures. Given a cluster of 256 processors connected by 100-Mbps Ethernet, multiple cladogram building is efficient to several hundred processors, so that is not the issue. However, swapping is most efficient at 32 processors. Thus, the processor pool can be divided using the command `controllers` (page 234). In this example 256 / 32 = 8 so you specify `controllers 8` to create 8 subclusters of 32 processors each. In a command line with `controllers 8`, `parallel`, `multirandom`, `replicates 8`, `multibuild`, and `buildsperreplicate 32`, eight simultaneous full searches with 32 random addition sequences each are executed in parallel by each subcluster. The best cladogram(s) of the 32 builds within each cluster are submitted to branch swapping in parallel using the processors within each subcluster. Command lines are provided below based on experiments with a 256-processor cluster. The best choices for any given cluster should be determined empirically by benchmarking as in Janies and Wheeler (2001).

Several full command lines follow that illustrate the techniques described above to achieve parallel efficiency.

```
poy -parallel -controllers 8 -multirandom
-replicates 8 -multibuild -buildsperreplicate 32
datafile > stdout 2> stderr
```

This command line performs eight full searches (Wagner tree through refinement) in parallel. Each search would have 32 Wagner builds (for a total of 256 builds), the best of which are submitted to swapping.

Alternatively:

```
poy -parallel -controllers 8 -multirandom
-replicates 64 multibuild -buildsperreplicate 32
datafile > stdout 2> stderr
```

The above command line performs 64 full searches. One full search (representing 32 Wagner builds plus swapping) will be spawned in each of the 8 subclusters. As each subcluster completes a full search, that subcluster will receive another search replicate until all 64 replicates are complete. A command line such as the one above can be modified to take advantage of stopping rule commands. For example:

```
poy -parallel -controllers 8 -multirandom
-replicates 64 multibuild -buildsperreplicate 32
-minstop 15 -stopat 3 datafile > stdout 2> stderr
```

The above command line performs up to 64 full searches (build through refinement) starting by sending one to each subcluster. However, instead of going through 64 searches, this command will stop if a putative minimum cladogram length is found three times after having completed at least 15 cladogram searches. Otherwise the 64 full searches are performed.

Efficiency in multiuser environments

Even an efficient search strategy can be upset by a complex multiuser environment. Individual users can overload with large memory demands, and many clusters are made of heterogeneous processors with various capabilities. In addition, the environment can change during the course of long runs. Designing a command line to deal with these sources of dynamic variation is difficult. One method built into POY to balance load on the fly as the search progresses is dynamic process migration using the command `dpm` (page 241). Dynamic process migration is intended to optimize overall cluster performance by moving work from processors with high CPU load to more capable or more idle processors. Periodically (and self-adjusting based on success) POY processes that use `dpm` commands poll the cluster looking to move tasks from busy, slow nodes onto relatively idle, fast processors. The `dpm` algorithm works on a market model in which processors bid for tasks based on their load. The user defines the rules of the market in order to tune the bidding to the heterogeneity of the cluster and user load. The key command `dpmacceptratio` (page 242) defines a threshold in the ratio of instantaneous processor performance to trigger migration of a POY job to another processor. Instantaneous processor performance response is a combined measure of current load and intrinsic CPU speed. In practice, bidding (that is, calculations of instantaneous processor performance) should not occur on every possible occasion but with sensible intervals defined by `dpmperiod` (page 245). It is possible to

automatically modulate bidding with respect to success using `dpmautoadjustperiod` (page 242). With this command, if requests lead to process migration, the bid period is reduced; however, if bids are unsuccessful, the period is increased such that the overhead of polling is a small percent of the total runtime of a search. On a cluster available at the AMNH with a diverse set of processors and users, the command string `-dpm -dpmautoadjustperiod -dpmacceptratio 1.2` has worked well. These commands switch jobs when a novel processor appears 20% faster than the one currently used.

Suggested Reading

Foster, I. 1995. Designing and Building Parallel Programs. Addison Wesley, Reading, MA. Available online at http://www-unix.mcs.anl.gov/dbpp/

Janies, D. and W. C. Wheeler. 2001. Efficiency of parallel direct optimization. Cladistics 17: S71–S82.

9 Guidelines for Research

Performance

Success in phylogenetic analysis has become increasingly dependent on computational speed. In describing POY optimization and cladogram search algorithms earlier, we noted that the use of large data sets increases computational demands exponentially.

Given finite analytical resources and ever-enlarging taxonomic and character samples, there are trade-offs between the amount of data and the exhaustiveness of analysis. No one set of analytical guidelines will be appropriate for all problems. POY offers a broad variety of options to tailor search strategies to the needs of the user, the demands of the problem, and the availability of computational resources.

This chapter offers some guidelines on how to use POY for specific data sets and analyses. Our focus is on the analysis of data implementing the dynamic homology concept used by the character optimization methods unique to POY. These are methods that are computationally demanding because they consist of nested NP-complete problems: homologies are determined as the most parsimonious topology is found.

The critical factor for the reader to effectively use POY is in how to design search strategies. Although POY brings both innovative heuris-

tics and the power of parallel processing to phylogenetic analysis, there are no set rules for its use. Each data set has its own idiosyncrasies, as does each computing environment on which POY is used. As a consequence, we describe three strategic areas:

- Partitioning of sequence data to create smaller data sets and reduce the computational load
- Overcoming the problem of composite optima inherent to large data sets and dynamic character optimization
  - Search algorithms and their combination
  - The use of sensitivity analysis
- Taking full advantage of POY in a parallel computing environment.

There are, as yet, no global rules on search strategies in POY. The suggestions here were developed in the context of our individual research projects.

Partitioning Sequence Data

Direct optimization examines all potential nucleotide homologies between two (or three in the case of iterative pass optimization) sequences. As a result, the time consumed is proportional to the product of the lengths of the sequences compared. Although correct, this procedure can be time-consuming for long pieces of DNA. In cases where sequences exhibit alternating stretches of conserved and variable regions (such as within and between amplification primers), economies can be realized by partitioning the sequences into pieces corresponding to these areas.

Since nucleotide homologies are not examined over the separate partitions, sequence optimization proceeds more rapidly. A sequence divided into *n* equal-length fragments will see an acceleration factor of approximately *n*. Giribet et al. (2000) and Edgecombe et al. (2002) first used this strategy with POY.

The caveat to following this strategy is that care must be taken in how DNA sequences are partitioned. While partitioning increases computational efficiency, it introduces *ad hoc* hypotheses into the analysis. That is, a partition assumes that the fragments are mutually exclusive. For example, if a partition creates three fragments, the analysis assumes that there are no homologues between the first and second fragments. If this were the case, the analysis could not establish the homology and would therefore not find the most parsimonious cladogram.

Use of flanking primers is the most conservative procedure for partitioning genotypic data. This strategy reduces the *ad hoc* nature of the

assumptions required to partition the data because it uses gene sequences as they were identified in the laboratory.

Figure 9.1 illustrates this strategy. A sequence is determined by sequencing three overlapping fragments using three sets of flanking primers. The resulting contiguous sequence is then partitioned using the primers as markers. The result is five fragments, each one being homologous in all the species for which the gene was amplified. In principle, this should have no effect on the analysis since the split separated nonhomologous nucleotides.

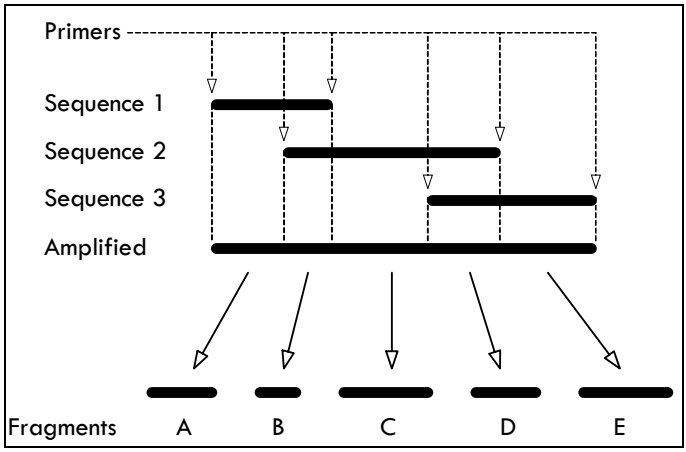

Figure 9.1: Three sequence fragments isolated by flanking primers, amplified to form a contiguous sequence, then partitioned into five fragments used in analysis.

The results of an analysis using this strategy can be validated by performing an analysis of the least cost cladogram from the partitioned data using the unpartitioned, single data set. If the partition's homology assumptions are valid, the cladogram cost from this second analysis using a single data set should be the same as a partitioned data set. If the single data set cladogram cost is less than the partitioned data set cost, then one should examine the assumptions upon which the partitions are based (see several examples in Giribet 2001).

In the same way that flanking primers can be used to concatenate multiple fragments for a given locus, multiple gene fragments or complete loci can be used simply by specifying the individual files for each gene. In addition, comparative data on secondary structure for ribosomal genes can be used to specify stems and loops or any other type of information that allows the investigator to partition the data unambiguously. Again, the more fragmentation the data suffers, the more possibilities for entering biases.

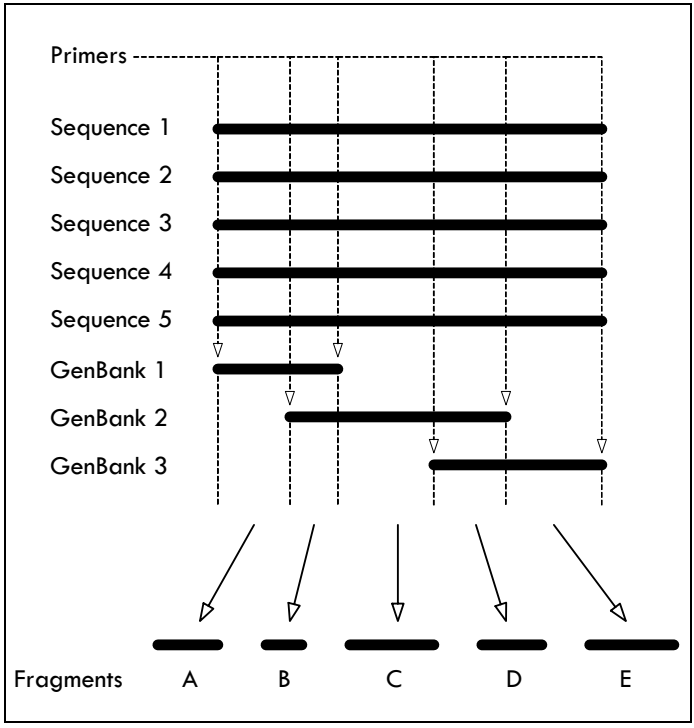

Figure 9.2: Separation of eight sequences by flanking primers into segments to ameliorate the effects of missing sequence data.

From a purely operational point of view, fragments can also be specified to ameliorate the effects of poor sequences, missing data, or lack of the overlap of sequences downloaded from a data base. Figure 9.2 illustrates this process. Imagine that an investigator sequenced 1000 bp for a given locus for five taxa, and three more partial sequences were downloaded from GenBank. The overlapping of the eight sequences may not be complete, and therefore dividing the sequences into fragments could assist with the handling of missing data.

Composite Optima

With large data sets, it is possible for an analysis to become stuck in a local optimum that is globally suboptimal—what Maddison (1991) calls an island of cladograms. This is a classic example of a complex solution "landscape" with peaks and valleys of local optima (2, 4, 5, and 6 in Figure 9.3 on page 111) and a global optimum (7 in Figure 9.3). The most common solution to this problem is the use of a cladogram search strategy that initializes on multiple, randomly selected starting points (a

RAS or Wagner tree as described page 65), then performs the cladogram search using TBR (as described page 71).

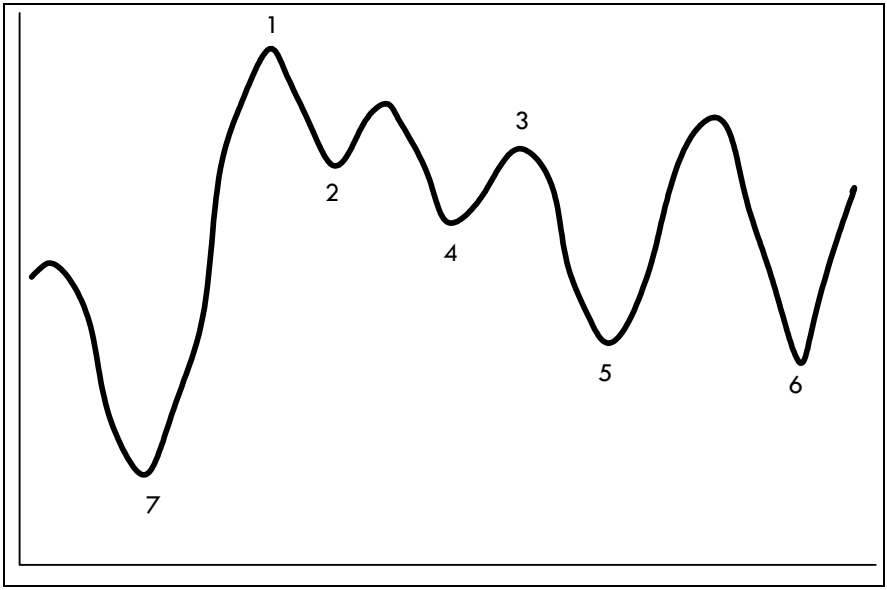

Figure 9.3: Local optima depiction for cladogram optimality space. The vertical is cladogram length. RAS + TBR procedure is likely to hit points 1 and 3 and get down to 2, 4, or 5, but will be stuck there. To reach shorter cladograms such as in 6 or the global optimum in 7, qualitatively different techniques are required.

However, the RAS + TBR strategy can be inefficient for large data sets (Goloboff 1999, 2002). Since the problem of optimizing length-variable DNA sequences on even a single cladogram is also NP-complete, heuristic strategies are employed (such as direct optimization) at the nucleotide homology stage as well. With this added level of complexity, search strategies must be even more aggressive than those for data sets based on anatomical or prealigned data.

In large data sets, individual sections of the cladogram can act as separate problems, each with its own local search space. A given RAS + TBR round might yield an optimal or nearly optimal configuration for one clade, but the remainder is markedly suboptimal and so on for multiple large clades. This is the notion of composite optima (Goloboff 1999). Such data sets are unlikely ever to achieve a satisfactory result simultaneously for all cladogram segments through the use of RAS + TBR replicates alone.

To reach a satisfactory result for such data sets, qualitatively different approaches are required to deal with the problem of composite optima:

- The use of algorithms such as ratcheting (Nixon 1999), tree fusing (Goloboff 1999), sectorial searches (Goloboff 1999), and tree drifting (Goloboff 1999);
- The use of strategies combining several cladogram search algorithms;
- The use of results from a sensitivity analysis as input cladograms for a refinement search.

Search algorithms

There are two main approaches to ameliorating the problems of local and composite optima in phylogenetic cladogram searching:

- Simulated annealing-type methods (for example, ratcheting and tree drifting), which strive to break out of local optima;
- Genetical algorithm-type methods (for example, tree fusing), which seek to exchange and combine results from multiple local optima.

These secondary refinement strategies are usually based on preliminary RAS + TBR searches.

Random addition sequences generate a diversity of starting points for refinement (that is, swapping) and, through their random component, allow exploration of multiple local optima (islands). Branch swapping might well be satisfactory for smaller or highly structured data sets, but even if not, will certainly provide fodder for further refinement. The number of cladograms that are saved per random replicate is an important factor in execution time since comparing cladograms is time-consuming. As stated by Goloboff (1999, page 416), "Saving too many trees per replication is a waste of time, because the trees found by swapping differ too little from the original tree(s) and are unlikely to lead to new optima."

Farris et al. (1996) showed that from a practical point of view, it is an inefficient use of resources to attempt to find all most-parsimonious cladograms for a given data set. Given that the object is usually a strict consensus of these optimal cladograms, the objective is to find the minimum number of cladograms with maximum disparity. It is likely that the strict consensus of a few, very different cladograms will be quite similar to that generated from thousands of slightly different cladograms. Hence, it is usually wise to keep the number of cladograms saved per random replicate to a minimum.

Ratcheting and tree drifting both accept suboptimal cladograms to generate potentially useful search paths to new, hopefully global optima. In principle, the ratchet/drift is an extension of RAS, in that

each successive replicate involves the exploration of, and potential departure from, a different local optimum. Therefore, solutions can be explored through the generation of multiple RAS departure points, followed by swapping refinement, or through the generation of fewer starting points but with the application of several ratchet/drift replicates on each original cladogram. Considering the time constraints involved with the building of initial cladograms in dynamic homology procedures (at least in POY), emphasis on swapping routines over initial building is logical.

Tree fusing exchanges information between cladograms and, as such, can yield an optimal cladogram from two suboptimal candidates with optimal subtrees. For example, Goloboff (1999) cited two advantages in using tree fusing:

- First, it uses suboptimal cladograms so long as some branches have an optimal configuration. As a result, replicates that might lead to an optimum are not rejected.
- Second, tree fusing provides a way to validate the global optimum: similar or identical least-cost cladograms that are independently based on multiple replicates support the global optimum.

Combining strategies

Goloboff (1999) described the combined use of cladogram search methods and how analyses can be improved. He notes the disparity between the time it takes to build cladograms from scratch for starting points and the time it takes to find a least cost cladogram based on such cladograms. We discuss these combined strategies further in "POY Process Flow" on page 114.

Most search procedures used in POY are based on RAS + TBR replicates, often with ratchet and tree fusing rounds within each replicate, followed by tree fusing among the replicate results and final TBR refinement.

Sensitivity analysis output

The results of a sensitivity analysis can be used effectively as a starting point for a more aggressive search under one or more parameter sets. For example, these cladograms can be submitted to tree fusing. The idea is in some way similar to ratcheting the data, because the cladograms obtained from the sensitivity analysis have been generated under different analytical conditions or weighting schemes. This strategy has been proven to give quick results for large data sets where simply building through the replicates would take a long time. The use of tree fusing is recommended if this strategy is chosen.

Parallel Environments

Strategies for using POY parallel processing capabilities are described in Chapter 8 Parallel Processing.

POY Process Flow

The high level structure of POY is presented in Figure 9.4 on page 115. A distinction is made between building cladograms and refining cladograms.

Refinement of cladograms consists of heuristic procedures to find better cladograms on the basis of preexisting cladograms. Available procedures in POY are branch swapping using SPR and TBR, tree drifting, ratcheting, and tree fusing.

Depending on the origin of the cladograms that are presented as input to these algorithms, a distinction is made between input cladogram refinement, replicate cladogram refinement, and final cladogram refinement:

- Input cladogram refinement refines input cladograms from the `topology` (page 323) and/or `topofile` (page 322) commands that are specified on the command line. The resulting cladograms are then set aside for final refinement later on.
- Replicate cladogram refinement during any replicate is based on the cladograms that are first constructed from scratch during the build step of that replicate. For each replicate, the cladograms that result from the replicate refinement are set aside for final refinement after all replicates are done.
- Final refinement starts on the basis of the cladograms that came out of input cladogram refinement and all replicate refinements.

Refinement of input cladograms Refine (optimize) cladograms that are input from another application or from a previous POY cladogram search.

Replicate build step Build an initial least cost cladogram from character data.

Replicate cladogram refinement Refine (optimize) cladogram replicates from initial least cost cladograms.

Final cladogram refinement Refine (optimize) the final cladogram from all least cost replicates and least cost input cladograms.

The methods selected to perform cladogram searches and optimize cladogram costs at each stage will determine the computational intensity of the analysis. It will also determine what amounts to a large data set. By selecting the appropriate algorithms for each stage, you will be able to efficiently utilize calculation time.

The implementation of dynamic homology (as described) is computationally demanding. When using algorithms such as direct optimization at any stage, data sets of modest size will have the computational demands of a large data set when compared to static homology searches.

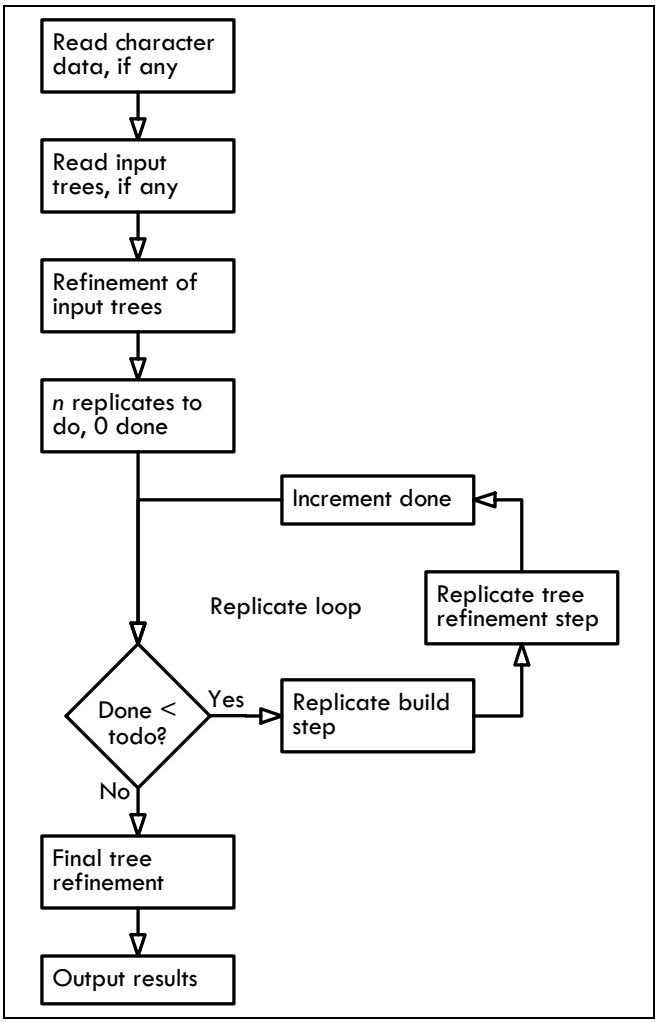

Figure 9.4: POY process flow.

Examples of Strategies

The examples of strategies shown here are only suggestions and starting points to elaborate your own search strategies, according to your data set, optimality criterion, and goals (exhaustiveness, data exploration, etc.).

The following examples use the default optimality criterion (parsimony) and search algorithm (direct optimization) unless specified. It has to be said that using a different optimality criterion such as maximum likelihood or a different search algorithm such as iterative pass optimization or search-base optimization is even more computationally demanding. Search strategies should be designed accordingly.

Additional detailed examples and tutorials are presented in Chapter 14 Tutorials.

A "quick and dirty" strategy

This is a strategy to get a simple Wagner tree quickly using shortcut heuristics such as `quick` (page 305) and `staticapprox` (page 317) and skipping the TBR branch swapping round. By default, direct optimization is used.

```
-norandomizeoutgroup -quick -staticapprox -notbr
```

The `quick` (default) command causes POY to not swap on non-minimal cost cladograms during branch swapping but only on minimum cost cladograms. Static approximation (`staticapprox`) uses a fixed implied alignment as a branch swapping cost heuristic. Every time a shorter cladogram is found, a new implied alignment is calculated (based on that shorter cladogram) and this new implied alignment used for further swapping.

The RAS + TBR strategy

The strategy of RAS + TBR implies several replicates of random addition sequence ("Wagner Trees" on page 65) completed by a round of tree bisection and reconnection branch swapping (page 71).

```
-replicates n
```

In that strategy, several cladograms can be built at each replicate:

```
-replicates n -buildsperreplicate m
```

By default, TBR branch swapping is performed. If POY is run in a parallel environment, `multirandom` (page 285) and `multibuild` (page 283) may be added depending on problem size and computational resources.

The more replicates, the more starting points are explored. However, each or most of the starting points might yield the same minimum length cladograms. To limit the number of replicates in the case of multiple hits of the minimal length, `stopat` (page 318) and `minstop` (page 282) can be used. For example:

```
-multirandom -replicates x -multibuild
-buildsperreplicate y -stopat z -minstop t
```

This command line is designed to run in a parallel environment to perform *x* replicates with *y* cladogram Wagner builds per replicate; if the same minimum cladogram cost is hit *z* times, the analysis will stop, but only after having completed at least *t* complete search replicates.

RAS + TBR is efficient for very small data sets (< 50 taxa). For larger data sets, however, it may be less effective. In these cases, RAS + TBR should be used as a starting point for more complex search strategies.

A basic strategy: RAS + TBR + Ratchet + TF

For this basic strategy, we assume that only character data are used, so that input cladograms are not used in the search. The strategy described here is a good starting point for a data set of 100 taxa or fewer (see D'Haese 2002, for example), but is already an intense strategy.

```
-replicates w -buildreplicate x -slop y -ratchettbr z
-treefuse
```

A diversity of starting points is generated using *w* rounds of RAS + TBR, saving preferably few cladograms per iteration (command `maxtrees` on page 282). Cladograms within a small percentage ($\frac{y}{10}$ %) of the least cost cladogram will be evaluated completely after heuristic cladogram cost calculation.

Ratcheting ("Ratcheting" on page 72) is used to refine replicate cladograms obtained by RAS + TBR. The severity of the ratchet and number of replicates can be adjusted.

The final cladograms will be refined with tree fusing ("Tree Fusing" on page 74) to combine the best sectors of each cladogram. In our experience, adding tree fusing after replicate rounds enhances the results only when dealing with data sets with more than 50 taxa. The values of *w*, *x*, *y*, and *z* can be modified according to how aggressive the user wants the search to be.

Drifting ("Tree Drifting" on page 72) could be added to or even replace ratcheting in the procedure:

RAS + TBR + Ratchet + DFT + TF or RAS + TBR + DFT + TF:

```
-replicates w -buildreplicate x -slop y -drifttbr
-numdrifttbr n -treefuse
```

In the final cladogram refinement step, a round of TBR branch swapping accepting all cladograms within *n* tenths of a percent of the current minimum value can be added:

```
-checkslop n
```

Sensitivity analysis + tree fusing

For a large data set (>100 taxa), where a good diversity of cladograms is already available, a simple tree fusing approach and a last round of swapping (hence, skipping the replicate loop) is a good strategy to find optimal cladograms.

The idea is to present tree fusing with a diversity of cladograms from different local optima.

The first step is to generate a diversity of cladograms using sensitivity analysis (Wheeler 1995). Several approaches are possible:

- Analysis of a static alignment (could be obtained by the command `impliedalignment`)
- "Quick and dirty" analysis
- RAS + TBR analysis
- More complex analyses such as the RAS + TBR + Ratchet + TF described above (if possible).

However the source cladograms are generated, this process is repeated for different parameter sets (for example, gap-transversion-transition = 1-1-1, 2-1-1, 2-2-1, 4-2-1; see Figure 9.5).

Gap:Transversion Ratio						
	8	8-1-2	8-1-1	16-2-1	32-4-1	8-1-0
	4	4-1-2	4-1-1	8-2-1	16-4-1	4-1-0
	2	2-1-2	2-1-1	4-2-1	8-4-1	2-1-0
	1	1-1-2	1-1-1	2-2-1	4-4-1	1-1-0
	0.5		1-2-2	1-2-1	2-4-1	1-2-0
		0.5	1	2	4	∝
		Transversion:Transition Ratio				

Figure 9.5: Different values of gap, transversion, and transition for a two-dimensional table using gap:transversion (G:Tv) and transversion:transition (Tv:Ts) ratios.

The second step is to collect all the cladograms obtained from the different parameter set analyses into the same file. Then the file is submitted as a topofile to tree fusing and other final refinement.

Tree fusing is controlled through several options (`poy -llist fuse`).

There are other strategies to generate a relevant diversity of cladograms such as jackknifing.

This strategy can also be used to verify the cladogram costs in a sensitivity analysis.

Strategies using different optimization heuristics or optimality

Different optimization heuristics at different stages can be implemented through the use of a series of scripts. Initial scripts might specify optimization heuristics that are less computationally intense and a subsequent script that processes the topologies (command `topofile` on page 322) through a more intense optimization heuristic or different optimality criterion altogether.

Example 1 Input the best topologies from a static alignment analyzed under parsimony with sensitivity analysis; then the best cladograms from each parameter set are fed to tree fusing under direct optimization.

Example 2 Search and swapping are done under direct optimization, then a final round is done under iterative pass. This strategy saves the time required at the build stage. Note that in general strategies like this will work better the closer the suboptimal topologies differ in structure from the optimal cladograms. One would expect to have fewer differences when dealing with sequences that do not vary wildly in length.

Example 3 Search and swapping are done under direct optimization for several parameter sets; then the resulting cladograms are submitted to tree fusing under maximum likelihood.

A general point is that these strategies will all work more effectively with a final round of swapping, holding additional cladograms. This explores the existence of additional equally optimal cladograms within a local minimum. Also, it is wise to save only a few cladograms at each step to avoid endless swapping on cladograms that are in the same local optimum. At the same time, it is best to pass cladograms with maximum disparity to subsequent steps. The `fitchtrees` (page 253) command is designed for this, ensuring that cladograms kept (set by `maxtrees` on page 282) are a random subset of those that would have been kept if the cladogram buffer would have been larger. This is based on an algorithm suggested by W. Fitch (Janies and Wheeler 2002).

Part III
Using POY

10 Requirements

Operating System and Binaries

POY strives to be platform independent. In general, POY is compiled for the latest stable operating system version and message passing software that are being distributed for each major architecture. If a shift is made in operating systems that causes some incompatibility, then POY binaries are posted for the two most recent versions (3.012 is current as of publication of this book) to `ftp://ftp.amnh.org/pub/molecular/poy`.

Users should use the same binary throughout a project to maintain full comparability among results. Users can always compile from sources, and instructions are provided in the chapter on installation.

Hardware

POY runs on computers from laptops and desktops to Beowulf clusters of various sizes to symmetric multiprocessing hardware and distributed systems. There are no particular requirements for disk space. Processor speed, memory, and communications bandwidth and latency are important, but faster systems cost more. Since limited financial resources are the norm, POY is designed to work well in low-bandwidth, high-latency Beowulf clusters as well as more tightly integrated machines. It is impor-

tant, however, to tune process granularity for your problem and hardware (for example, one-processor-per-replicate strategies as discussed in Chapter 8 Parallel Processing and Chapter 9 Guidelines for Research).

The critical factor is being able to analyze data using a number of replicates that enables you to find minimum cost cladograms multiple times in a reasonable amount of time. Choosing hardware depends not only on understanding parallelism and its associated heuristics, but also on understanding the time and storage complexity of the many optimization styles and refinement heuristics included in POY.

Time Complexity of Cladogram Building

The cost of computation is combinatorially expansive for an exact search. The number of possible rooted topologies increases as a power series, where i is the starting point and t is the number of taxa (Cavalli-Sforza and Edwards 1967). The problem is NP-complete, and heuristics are required for biologically interesting data sets. For example, for 30 taxa representing exemplars of most metazoan phyla, an exact search would require examining over 5×10^{38} cladograms. The heuristics commonly implemented in phylogenetic algorithms are sensitive to the order in which taxa are accreted into an alignment or cladogram optimization procedure. Thus random replication of the Wagner build process is important to adequate searching. When considering heuristics in POY, the cost of computation of each Wagner build on a single slave processor increases proportional to t^2, where t is the number of taxa in the data set (Janies and Wheeler 2001).

Time Complexity of Cladogram Refinement

Branch swapping underlies many refinement heuristics in cladogram searching such as ratcheting, tree fusion, and tree drifting (Goloboff 1999, Nixon 1999). In practice two modes of branch swapping—subtree pruning and regrafting (SPR) and tree bisection and reconnection (TBR)—are employed by POY users. The default behavior of POY is to use rounds of branch swapping at many points during the overall search, including cladogram building, to make refinements of intermediate results. An understanding of the time complexity of these operations is central to understanding how to assign reasonable amounts of work to the machinery you have at hand.

When considering heuristics in POY, the cost of computation for branch swapping is proportional to t^2 for SPR and $t^{2.5}$ for TBR, where t is the number of taxa.

When an analysis has a moderate to large number of taxa and only a small number of processors are available, branch swapping should be

minimized and advanced heuristics avoided. For example, a quick and dirty strategy (shown below) for various groups of >50 animals for 16S rRNA can produce cladograms marginally longer (within 0.25% or less of minimal cost) than an intense strategy (also shown below). However, the intense strategy takes more than 20 times longer than the quick and dirty strategy.

Quick and dirty strategy

```
-nodiscrepancies -norandomizeoutgroup -sprmaxtrees 2
-tbrmaxtrees 2 -fitchtrees -holdmaxtrees 50 -quick
-staticapprox -replicates 16
```

Intense strategy

```
-nodiscrepancies -norandomizeoutgroup -sprmaxtrees 2
-tbrmaxtrees 2 -fitchtrees -holdmaxtrees 50 -buildtbr
-buildmaxtrees 2 -replicates 16 -ratchettbr 25
-ratchetpercent 10 -ratchetseverity 3 -ratchettrees 2
-treefuse -fuselimit 10 -fusemingroup 5
-fusemaxtrees 100 -slop 10 -checkslop 10
```

A moderate amount of tree fusing can be applied to the cladograms produced by various parameters specified by `molecularmatrix` (page 283) in conjunction with a quick and dirty strategy. This often reduces the cladogram cost to the same lower bound as found by the intense strategy to within 0.15% of the minimal cost achieved using the intense strategy. Strategies such as quick and dirty searches with parameter variation followed by tree fusing are efficient for cladogram searches with large numbers of taxa.

Storage Requirements

Wheeler (2003c) introduced search-based optimization (see page 55). Search-based optimization is a variation on fixed-states optimization that employs a user-supplied diversity of potential state solutions for hypothetical ancestral sequences rather than by inferring these states *de novo*. Search-based optimization can result in near-minimal length cladograms in a fraction of the time required for direct optimization (see page 47).

Studies of simple data sets on 32-bit systems indicate that search-based optimization can offer a three-fold speedup over direct optimization, or for very large searches, exhibit much longer execution times. However, the efficacy of search-based optimization is limited by the size of the state set to be searched. Thus search-based optimization is sensitive to the memory limitations of 32-bit systems. These algorithms are cur-

rently being investigated in 64-bit systems, which afford much greater memory space and bandwidth.

To address these inefficiencies, a method of reconstructing ancestral sequences termed iterative pass optimization (see page 56) was implemented (Wheeler 2003b). Using iterative pass, each of the internal vertices of a cladogram is preliminarily estimated in the down pass. Upon completion of the down pass, rather than performing a single up pass, iterative revisiting of the internal vertices is used to refine the ancestral sequence reconstructions. Iterations are repeated (avoiding cycles) until stability is achieved, thus reducing the total cladogram cost. Iterative pass optimization often produces cladograms that are lower cost than those that result for direct optimization but with a large increase in computational time.

The space requirements of these methods are demanding. Whereas direct optimization typically requires l^2 elements of storage (where l is the DNA sequence length), l^3 elements of storage are required for iterative pass. The storage requirements for search-based optimization are proportional to $l^2 S^2$ (where S is the number of states provided). Some memory economy has been implemented with the command `iterativelowmem` (page 265) by using the optimized diagonal transition algorithms described by Ukkonen (1985).

11 Installation

Installation

Source code, documentation, test data sets, stable binaries, and test binaries are available via anonymous ftp for Linux, Microsoft Windows, and Mac OS X at `ftp://ftp.amnh.org/pub/molecular/poy`.

- Make sure to use binary mode for the transfer.
- You will then need to decompress the binary.
- You can change the name of the binary after downloading.
- Make sure to use `chmod u + x` to make the binary user executable on Unix-like systems.

Download

POY is available on the American Museum of Natural History's ftp site. To obtain POY executable and support files, take the following steps:

1. In Windows-based systems, create a folder named `POY`.
   - This is the folder to which you will download the POY source files and from which POY is executed.
   - Alternative configurations can be used with appropriate file management.

- For parallel installations (for example, on Unix-like systems), see "Parallel Installation" on page 131.

2. Connect to the POY ftp site using `anonymous` as the user ID and your email address as a password.

   `ftp://ftp.amnh.org/pub/molecular/poy`.

3. Download the following files to the POY folder (step #1). Use binary transfer for the documentation PDFs and utilities of interest. Use ASCI transfer for text files of interest. Note that some are architecture specific.

Table 11.1: POY reference and utilities to download

File name	Description
poy3.pdf	POY users' manual in PDF (Acrobat) format (*POY Users' Manual*)
README	Notes on the current release of POY
README.MAC	Notes on the current release of POY for Macintosh users
jack2hen.exe	An executable that converts POY output to a Hennig86/Nona format matrix used for constraint files (refer to "Constraint file" on page 157)
jack2hen.c	The source code for .exe

4. Download the following folders to the POY folder (step #1). Use ASCI transfer for sample data, output, and script files of interest. Note that some are architecture specific.

Table 11.2: POY sample data folders to download

Folder	Description
sampledatados	Examples of input data files and an example of a command script for use in a Windows implementation of POY (refer to "Input Data" on page 147)
sampledatalinux	Examples of input data files and an example of a command script for use in a Linux implementation of POY (refer to "Input Data" on page 147)
samplematrices	Examples of Sankoff matrices of transition, transversion, and gap costs (refer to "Multiple loci" on page 154)
sampleoutput	Examples of output files in various formats: POY, TREEVIEW, WINCLADA, and PAUP/MACLADE

5. On the ftp site, change to the folder named `version3-current` or equivalent, which contains the stable versions of the POY executable.

6. Download the file archive appropriate to your architecture to the POY folder (step #1).

Table 11.3: POY archives to download from the `version3-current` folder

File name	Description
poy.MacOSX.gz	POY executable for Macintosh OS X
poy.linux.gz	POY executable for Linux
poypc.exe.gz	POY executable for Windows
poygui.tar.gz	POY graphical user interface
poy.source.tar.gz	POY source code and make files

Also download the release notes (`release.notes`) in this folder.

The following is a transcript of the steps described above for a Mac OS X system.

```
[yourcomputer:~] ftp ftp.amnh.org
Connected to philosophy.amnh.org.
220 FTP Server ready
Name (ftp.amnh.org:you): anonymous
331 Anonymous login ok, send your complete email address as
your password.
Password:
230-

************************************************************
 * Welcome to the American Museum of Natural History FTP
archive *
************************************************************

   All transactions, anonymous or otherwise are logged.

   Any questions or problems, please contact
ftpadmin@amnh.org.


 -------
 ~
230 Anonymous access granted, restrictions apply.
Remote system type is Unix.
Using binary mode to transfer files.
ftp> bin
200 Type set to I.
ftp> cd pub/molecular/poy
250 CWD command successful.
ftp> cd version3-current
250 CWD command successful.
ftp> get poy.3.0.12.MacOSX.tgz
```

```
local: poy.3.0.12.MacOSX.tgz remote: poy.3.0.12.MacOSX.tgz
500 EPSV not understood.
227 Entering Passive Mode (209,2,162,204,164,138).
150 Opening BINARY mode data connection for
poy.3.0.12.MacOSX.tgz (1272176 bytes).
100% |*****************************************************|
1242 KB  325.52 KB/s    00:00 ETA
226 Transfer complete.
1272176 bytes received in 00:03 (325.29 KB/s)
ftp> quit
221 Goodbye.
```



```
gzip -d poy.3.0.12.MacOSX.tgz
tar -xvf poy.3.0.12.MacOSX.tar

[yourcomputer:~] you% ./poy.MacOSX
executing ./poy.MacOSX

This is POY
version 3.0.12 (May 6 2003)

 poy.MacOSX -cat_helptopics       show commands for specific
help topics
 poy.MacOSX -cat_commandbrowsing  show commands to browse
command documentation
```

Install

To install POY, take the following steps:

1. Decompress (unzip) the archives downloaded in step #6 above.
   - For Windows, use a current unzip utility, such as that offered by `http://www.aladdinsys.com/win/index html`.
   - For Unix-like systems, take the following steps:

     ```
     gzip -d poy.3.0.12.MacOSX.tgz
     tar -xvf poy.3.0.12.MacOSX.tar
     chmod a+x poy.3.0.12.MacOSX
     ```

2. You can now rename the POY executable filename if you wish. For example, rename `poypc.exe` to `poy.exe`.

POY is now ready to run on this machine (refer to Chapter 12 The POY Interface).

You can test POY with the example scripts and data. In Unix-like systems, you can also do the following:

```
./poy.3.0.12.MacOSX
```

If POY is installed successfully, you will see the following:

```
This is POY
version 3.0.12 (May 6 2003)
poy.MacOSX -cat helptopic show commands for
specific help topics
poy.MacOSX -cat commandbrowsing show commands to
browse command documentation
```

If you intend to run POY in a parallel environment, continue with the following section.

Parallel Installation

In addition to sequential implementation, POY has been parallelized using Parallel Virtual Machine (PVM) and the Message Passing Interface Standard version 1.2 (MPI 1.2) for use on clusters and traditional supercomputers. Linear speedup is possible for several commonly used tree-search algorithms over hundreds of processors (Janies and Wheeler 2001).

The essential software elements of a parallel cluster for POY are PVM or MPI and a remote shell program (such as rsh or ssh). PVM is an open source project freely available on the internet (http://www.csm.ornl.gov/pvm/pvm_home.html). This software package permits a heterogeneous collection of Unix and Windows computers (as well as other platforms) hooked together by a network to be used as a single large parallel computer. Rather than a library, MPI is a *standard* that has been adopted by several hardware vendors to ensure portability of their parallel software. As a standard, there exist several MPI implementations, some of them vendor specific. The most widely used free implementations are MPICH (http://www-unix.mcs.anl.gov/mpi/mpich) and LAM/MPI (http://www.lam-mpi.org). POY requires version 3.4.3 or later of PVM or an MPI 1.2 standard compliant implementation (such as MPICH 1.2.5).

Many Linux distributions come with PVM installed in `/usr/share/pvm3`. These versions of PVM may be the most up-to-date, and permissions may be properly set on that directory. Make sure to have a 3.4 PVM revision that has debug features and supports the PVM_RSH variable via `ftp://netlib2.cs.utk.edu/pvm3`.

To be comprehensive, we describe the installation of POY for both message passing libraries and the configuration of a cluster from scratch. Most of what we present is for Linux administrators with root access. Examples are provided for installation in a user's home directory and other architectures so that you can modify these instructions.

If you use SSH, it must be password-free across all nodes. For example, type

```
ssh -keygen -d
```

Follow prompts using `no Passovers` by pressing Return then enter

```
cp ~/.ssh/id_dsa.pub authorized_keys
chmod 700 ~/.ssh
chmod 600 ~/.ssh/authorized_keys
```

Then use `scp` to allow access to other nodes.

```
scp .ssh/id_dsa.pub yourhost1:.ssh/authorized_keys
scp .ssh/id_dsa.pub yourhost2:.ssh/authorized_keys
scp .ssh/id_dsa.pub yourhost3:.ssh/authorized_keys
scp .ssh/id_dsa.pub yourhost4:.ssh/authorized_keys
scp .ssh/id_dsa.pub yourhost5:.ssh/authorized_keys
```


A root user can use `/usr/share/pvm3` to install PVM on all nodes.

A root user can install POY for all nodes for all users. To do this, copy the POY binaries to `/usr/local/bin` and `/usr/share/pvm3/bin/Linux` on all nodes (or `/usr/share/pvm3/bin/YOURARCHITECTURE-HERE`). If Network File System (NFS) is configured, you do not have to put binaries on each node.

A root user can use an `/etc/profile.d/pvm.csh` file (or shell of choice) to set up environmental variables for all users. The `pvm.csh` file should contain

```
PVM_ROOT=/usr/share/pvm3
PVM_RSH=/usr/bin/ssh
```

A regular user does not normally have write access to the directories mentioned above. Nevertheless, it is not difficult to install POY and PVM in your home directory.

Users must set up environment variables in their shell's dot file. There is a `.cshrc` stub file made by the creators of PVM that is very useful. The following is an example of a `.cshrc` stub.

```
# append this file to your .cshrc to set path
    according to machine type.
# you may wish to use this for your own programs
    (edit the last part to
# point to a different directory e.g. ~/bin/
    _$PVM_ARCH.
#
if (! $?PVM_ROOT) then
  if (-d ~/pvm3) then
    setenv PVM_ROOT ~/pvm3
  else
    echo PVM_ROOT not defined
```

```
    echo To use PVM, define PVM_ROOT and rerun
your
      .cshrc
  endif
endif

if ($?PVM_ROOT) then
  setenv PVM_ARCH `$PVM_ROOT/lib/pvmgetarch`
#
# uncomment the following line if you want the PVM
    executable directory
# to be added to your shell path.
#
#       set path=($path $PVM_ROOT/bin/$PVM_ARCH)
endif
```


Users who install in their home directory, for example using c shell and SSH for internode communication, should use the `.cshrc` stub file and have the following lines in `.cshrc`.

```
set path=(/usr/local/bin:/bin:/usr/bin:/usr/X11R6/
bin:username/bin)
setenv PVM_RSH /usr/bin/ssh
setenv PVM_ROOT /home/username/bin/pvm3
setenv TERM vt100
```

Then make the directory `/username/bin`. Compile PVM in this directory according to the instructions found with the source at `ftp://netlib2.cs.utk.edu/pvm3`. Normally, this requires decompressing the source, making sure your environmental variables are active (restart `.csh` if necessary), and typing `make` and when the compilation is complete, typing `make install`. See modifications for Mac OS X Installation below.

Starting PVM

PVM tutorials are provided by its authors at

`www.netlib.org/pvm3/book/pvm-book.html`.

The basics of PVM are quite simple. To start, type `PVM` followed by a list of slaves within a `phosts.file`.

If running from `/usr/local/bin/ usr/share,` create a `phosts.file` that contains

```
yourhost1.yourdomain ep=/usr/home/bin/ dx=/usr/share/pvm3/lib/pvmd
yourhost2.yourdomain ep=/usr/home/bin/ dx=/usr/share/pvm3/lib/pvmd
yourhost3.yourdomain ep=/usr/home/bin/ dx=/usr/share/pvm3/lib/pvmd
yourhost4.yourdomain ep=/usr/home/bin/ dx=/usr/share/pvm3/lib/pvmd
```

```
yourhost5.yourdomain ep=/usr/home/bin/ dx=/usr/share/pvm3/lib/pvmd
```

If running from `/usr/local/bin/ usr/share`

```
yourhost1.yourdomain ep=/home/username/bin/ dx=/home/username/bin/pvm3/lib/pvmd
yourhost2.yourdomain ep=/home/username/bin/ dx=/home/username/bin/pvm3/lib/pvmd
yourhost3.yourdomain ep=/home/username/bin/ dx=/home/username/bin/pvm3/lib/pvmd
yourhost4.yourdomain ep=/home/username/bin/ dx=/home/username/bin/pvm3/lib/pvmd
yourhost5.yourdomain ep=/home/username/bin/ dx=/home/username/bin/pvm3/lib/pvmd
```

Then start PVM by typing

```
master /home/username 22 # pvm phosts.file
```

PVM responds

```
pvmd already running.
pvm>
```

The `pvm>` prompt signifies that you are in the PVM console where you can check slave hosts by typing

```
conf
```

PVM responds

```
5 hosts, 1 data format
                HOST     DTID     ARCH   SPEED       DSIG
yourhost1.yourdomain    40000    Linux    1000 0x00408841
yourhost2.yourdomain    80000    Linux    1000 0x00408841
yourhost3.yourdomain    c0000    Linux    1000 0x00408841
yourhost4.yourdomain   100000    Linux    1000 0x00408841
yourhost5.yourdomain   140000    Linux    1000 0x00408841
pvm>
```

If some of the slaves are missing from the `conf`, slaves can be added manually by typing

```
add yourhost.yourdomain
```

If unsuccessful at this point, PVM will perform tests and provide debugging information.

```
ps -al
        HOST            TID   PTID   PID FLAG 0x COMMAND
yourhost1.yourdomain  40003      -  5994  4/c -
yourhost2.yourdomain  80002  40003  6441  6/c,f poy
yourhost4.yourdomain 100002  40003  1018  6/c,f poy
yourhost5.yourdomain 140002  40003  8389  6/c,f poy
yourhost3.yourdomain  c0002  40003  4178  6/c,f poy
```

To exit the console but leave PVM running type

```
pvm> quit
```

Back in your shell you can execute POY in parallel via the command line or scripts.

Compiling POY from source code

The following must be installed before attempting to compile POY: GCC, PVM, and OCaml. PVM/MPI is optional for sequential versions.

- GCC is available at `ftp://ftp.gnu.org/gnu/gcc`
- OCaml is available at `http://caml.inria.fr/`

and one of the following:

- PVM is available at `ftp://netlib2.cs.utk.edu/pvm3`
- MPI from your hardware vendor or from a free source such as MPICH

Although POY is used on a wide variety of platforms, it is specifically designed to take advantage of Beowulf clusters that any research group can assemble from evermore inexpensive PC components and switching equipment. As a result, POY source and make files are distributed with the latest GNU/Linux systems in mind. However, PVM, MPI, and POY makefiles may have to be tailored to a specific system. Instructions on PVM or MPI compilation and installation are available on their websites. The source code distribution includes a Makefile to be executed with version 1.0 or later of the GNU make utility. Up- to- date instructions are available in the INSTALL file; instructions for the current version are explained below.

Modify the Makefile in the lines containing the parallel library (MPI or PVM) and the OCaml distribution. You will find something like:

Run the Makefile using the follow sets of commands:

For PVM:

```
make wpvm
make depend
make poy
```

This will produce an executable file called *poy* that will perform parallel operations using an installed PVM implementation.

For MPI:

```
make wmpi
make depend
make mpipoy
```

This will produce an executable file called *mpipoy* that will perform parallel operations using an installed MPI implementation.

Install the files by following your PVM or MPI instructions; for further installation details (like having the program available to all the processors) consult your system administrator.

12 The POY Interface

Command Line

Default behavior

The minimum POY command line consists of the name of the executable and a data file. For example,

```
poy data
```

This command line performs a single Wagner cladogram build.

- The first taxon in the file `data` is used as the outgroup and the cladogram is built based on the order in which the taxa are listed in `data`.
- Direct optimization is used throughout for calculating edit costs.
- If `data` consists of nucleotide data, gaps are assigned a weight of 2 while transitions and transversion are assigned a weight of 1.
- A single round of TBR branch swapping is performed to refine the best cladogram found during the building step.
- The best cladograms found after swapping and their binary representations with polytomies arbitrarily resolved are output to the screen, with cladograms represented in nested parenthetical representations. In addition, a small amount of data on the work completed, such as

the number of cladograms examined in various stages of the search and the number of operations performed, are output to the screen.

The set of operations illustrated above is unlikely to represent an adequate search. Typically, POY is executed from the command line either directly or via a shell script containing several interacting commands that set up a more aggressive search.

Syntax

On the command line, POY treats any text that is not preceded by a hyphen as an input file unless a command uses the subsequent text as an input or output filename or an argument. For example, data can be provided in multiple input files, and any number of commands can be applied to guide the optimization. There are five basic components of POY's command line syntax, shown in Figure 12.1.

Component	Description
-command	The command (refer to Chapter 15 Command References). Some commands expect input files, output files, arguments, or modes.
input	The name of the input files (refer to Chapter 13 POY Input and Output) provided to POY by the user (for example, nucleotide data, chromosomal data, morphological data, topologies in nested parenthetical format, constraint files in Hennig format, ancestral nucleotide states, and molecular matrix files).
output	The name of the output files to which POY writes (for example, phastwinclada matrices, ancestral states, and ASCII art cladograms).
stdout	The name of the standard output file created by the shell to which POY results are written (for example, nested parenthetical cladograms, diagnoses, and implied alignments).
stderr	The name of the standard progress file created by the shell to which POY writes progress and error messages (for example, reporting on builds, intermediate cladogram lengths, tallies of costs hit for each replicate, time of runs, number of alignments performed and cladograms searched, fault tolerant events, and process migration reports).

Figure 12.1: Components of a POY command line.

The basic syntax of the command line is

```
poy input -command argument -command > stdout 2>
stderr
```

If a line with this form is entered or executed from a script file, the shell of the operating system shell will direct the results of the POY search to the file named immediately after >. Similarly, the progress and errors reported by POY will be directed to the file named immediately after 2>. This is standard output and error reporting syntax typical of Unix-like operating systems. It also works under DOS for Windows NT and XP and under certain shells for Mac OS X.

The command line must be a continuous line of commands, with the following features:

- Command names must be preceded by a hyphen.
- Commands that expect input must be followed immediately by appropriate arguments or file names.
- Strings without a hyphen will be treated as an input file, *not* a command.
- Commands and their arguments or input files must be separated by at least one space.
- File names for *input*, *output*, *stdout*, and *stderr* are user-specified strings. As such, these strings can be anything that is legal with the operating system and shell. Intelligent file names help users keep track of various results (for example, `.seq`, `.mat`, `.out`, `.err`, `.aln`, `.tree`, and `.chrom`). The tutorials in Chapter 14 demonstrate this kind of naming convention.

Examples

The following command line performs direct optimization on the nucleotides included in the file named `data`. Nucleotide transitions, transversions, and insertion–deletion costs specified are in `matrix`. The analysis results are written to `stdout` (the standard output file). Program progress and error messages are written to `stderr` (the standard program progress file). The contents of and formats for these files are treated in Chapter 13 POY Input and Output.

```
poy data -molecularmatrix matrix > stdout 2> stderr
```

The following command line cooptimizes the nucleotide data in `data` with the morphological information in `morph.hng`. Nucleotide transitions, transversions, and insertion–deletion costs are specified in `matrix`. However, weights for morphological character change are specified by the command `-weight 2`. The analysis results are written to `stdout` (the standard output file). Program progress and error messages are written to `stderr` (the standard program progress file).

```
poy data -molecularmatrix matrix -weight 2 morph.hng
> stdout 2> stderr
```

The following command line optimizes genome composition and order data in a file `genomes`. However, weights for various events are provided by arguments in the commands `locusgap` (page 279), `locussizegap` (page 279), `breakpoint` (page 220), and `molecularmatrix` (page 283). The analysis results are written to `stdout` (the standard output file). Program progress and error messages are written to `stderr` (the standard program progress file).

```
poy -chromosome -rearrange genomes -locusgap 2
-locussizegap 2 -distcost 2 -molecularmatrix matrix >
stdout 2> stderr
```

Scripts

Script files are plain text files that consist of one or more command lines and often variables that are interpreted by the operating system's shell. To execute an existing script, type the script name at the shell's prompt. It is often useful to use the `&` command after the script name to run it in the background. For example

```
[Your-Computer:~] yourcomp% sh script.sh &
```

Basic knowledge of Unix shell scripting or DOS batch programing can be valuable for guiding a set of related POY runs. Below is an example of a Unix shell script that is useful in this context

- to illustrate the use of a shell variable as indicated by the for statement and the `$`
- to provide a practical script users can adopt to check all input files to see whether they are POY legal.

The simple shell script `test1.sh` runs a normal POY search. The script `test1.sh` does not run POY all the way through. Instead, it tests the files one by one as the script variable indicated in the command line by `$FILE` is passed to the POY command line. Below are the contents of `test1.sh`.

```
for FILE in data1 data2 data3
do
poy -topodiagnoseonly $FILE > $FILE.out 2> $FILE.err
done
```

If each file contains POY legal text, a positive response is written to `$FILE.err`.

Once each data and input file is POY legal, it is often useful to check data files together for equal complements of taxa. The following script does not run a full POY search, but instead tests the data from multiple loci as a set.

```
poy data1 data2 data3 -topodiagnoseonly $FILE
```

Refer to `terminalsfile` (page 320) for help in making complementary data files. Once all the data are acceptable to POY as indicated, analyses can be performed.

Similarly, for large sets of loci, shell variables—exemplified here by `FILES_ALL=`, `FILES_NUCLEAR=`, and `FILES_MITO=`—are useful to pass a string representing all or subsets of the data—exemplified here by `'data1 data2 data3'`, `'data1 data2'`, and `'data3'`.

```
FILES_ALL='data1 data2 data3'
poy $FILES_ALL > $FILES_ALL.out 2> $FILES_ALL.err
FILES_NUCLEAR='data1 data2'
poy $FILES_ALL > $FILES_NUCLEAR.out 2> $FILES_NUCLEAR.err
FILES_MITO='data3'
poy $FILES_MITO > $FILES_MITO.out 2> $FILES_MITO.err
```

The following is an example of a Wagner script that performs a search with 10 random builds, examining cladograms within 1% of the minimum cost, and performing a final TBR refinement (the default). The script consists of the following command line:

```
poy data1 data2 data3 -replicates 10 -slop 10
-molecularmatrix matrix > stdout 2> stderr
```


To create or edit a script

You can create or edit a script or other input file directly using a text editor or a Unix editor like vi, EMacs, or pico. When using a text editor, make sure script files are saved as plain text appropriate for the operating system on which you want to execute them. Typically, the script file must contain only characters, spaces, and line breaks legal for the operating system on which it will be executed. For example, if scripts or other input files are created or modified with text editors or word processors, it is crucial to make sure that the resulting files are saved as text only (or DOS text only).

Some programs, especially word processors, regardless of platform, may insert illegal, misplaced, and sometimes invisible characters that do not conform to formats required by POY. If such characters are present, POY will most likely exit with an error message. These problems can be avoided by using POY legal editors (but see also the command `enabletmpfiles` on page 249). The following are editors commonly used by POY users:

- BBEDIT (`ftp://ftp.barebones.com`) is a good Mac editor suited to Unix.
- DOS users find that EMacs (`http://www.gnu.org/software/emacs/emacs.html`) works well.

- Word processors that show all characters (for example, "show paragraph" command in MSWord or "show codes" command in WordPerfect) might be useful to root out formatting errors. Wordpad from MS Windows works well.

Monitoring Output

In Unix systems, the commands `more`, `less`, and `cat` are useful for viewing text files such as the `stdout` and `stderr` of POY. The Unix program `tail` is useful to see the end of a file. Because the `stderr` file can be very large, the command `tail` and its options are useful to visualize the end of the `stderr` file and to see data such as numbers of minimum cost cladograms hit and other statistics of the run. For example, typing `tail -50 filename` will show the last 50 lines of the file `filename`.

During long runs, the `stderr` file should be monitored for discrepancies between heuristic and checkpoint cladogram cost calculations. If discrepancies are large, the user can address this by adjusting slop values for various cladogram building and cladogram refinement procedures (for example, `buildslop` on page 222, `checkslop` on page 228, and `ratchetslop` on page 308; alternatively, all slop values can be set using `slop` on page 314 if they are not individually tuned) . Slop values can be used to examine suboptimal cladograms near the length of the putative optimal cladogram at the expense of greater execution times.

Cladogram Buffers

Indecisive or missing data can lead to many equally parsimonious resolutions that occupy cladogram buffers and slow searches considerably due to swapping on many cladograms from a particular build or random replicate (searching in a single region of cladogram space). Often cladograms generated in a single replicate differ only in the placement of a few taxa. As a result, the full diversity of topologies that would be generated in swapping on many cladograms per build can be found by swapping on a random subsample of these cladograms (refer to `fitchtrees` on page 253). CPU time is better spent for a more global search of cladogram space by the use of many replicates and aggressive refinement of the best cladograms from replicates based on which hold only a few cladograms. Thus one should generate many replicates, swap on a few randomly sampled cladograms within each replicate, collect the best cladograms, and submit them to further refinement.

A random subsample of cladograms generated in a build can be kept in POY using `fitchtrees` and one of the `maxtrees` commands (refer to `maxtrees` on page 282). The command `buildmaxtrees` (page 222)

sets buffers to hold *n* cladograms within build replicates. The command `holdmaxtrees` (page 260) sets the overall hold buffer for all replicates to enable the accumulation of cladograms from all builds. If `holdmaxtrees` or `buildmaxtrees` or other cladogram buffers for other operations are not specified, `maxtrees` sets the overall buffer limit.

Aggressive Searches and Operations Order

As described above in simple scripts, POY will do one or many random build replicates with the first taxon as the outgroup followed by TBR swapping. A more aggressive search suited to large data sets should include a mix of the commands below. Regardless of the order in which they are listed in the command line, they are summarized here in the order in which POY executes them within a replicate.

1. Perform initial build (stepwise addition of taxa), consulting options (for example, `buildslop` on page 222 and `buildmaxtrees` on page 222). If running in parallel, spawn jobs to slave nodes, consulting options (for example, `parallel` on page 293, `multirandom` on page 285, `replicates` on page 311, `buildsperreplicate` on page 223).
2. Submit best topologies held in buffers from building to SPR and TBR swapping consulting options (for example, `cutswap` on page 236, `sprmaxtrees` on page 317, `intermediate` on page 263).
3. Submit best topologies held in buffers from swapping to tree drifting plus drifting options (for example, `driftspr` on page 248, `numdriftspr` on page 287, `numdriftchanges` on page 287).
4. Submit best topologies held in buffers from tree drifting to ratcheting plus ratcheting options (for example, `ratchetspr` on page 309, `ratchettbr` on page 309, `multiratchet` on page 285).
5. Submit best topologies held in buffers from ratcheting to tree fusing plus fusing options (for example, `fusemingroup` on page 255, `fuselimit` on page 254).
6. If `replicates` (page 311) is set to $n > 1$, submit best topologies held in buffers to a final round of TBR swapping consulting options (for example, `checkslop` on page 228).
7. Output best topologies if `repintermediate` (page 311) is invoked.
8. Proceed to additional replicates.
9. When all build replicates are complete, perform a final round of TBR swapping consulting options (for example, `checkslop` on page 228).
10. Output results as specified.

Types of Output

The simplest form of POY output is the nested parenthetical cladograms followed by their costs in the standard output file. POY does not put commas between clades, so the user might want to do a find and replace of spaces for commas on POY nested parenthetical cladogram files to make them readable by many visualization programs such as MESQUITE, MACCLADE, and TREEVIEW.

Keep in mind that POY output cladograms are unrooted. It is up to the user to define a root. This can be easily done in some visualization programs especially if multiple taxa are used as an outgroup. Alternatively, the POY command `topooutgroup` (page 324) writes rerooted cladograms to the standard output file. The following is a simple script that illustrates this important post-processing step:

```
poy -enabletmpfiles -topooutgroup outgroupname
-topofile topos > stdout 2> stderr
```

where `topos` is a file containing topologies and `outgroupname` is the name of the outgroup taxon.

Hypothetical ancestral sequences with resolved ambiguities can be output by POY to a user-specified file for use in search-based optimization by including the commands `printhypanc` (page 303) and `hypancfile` (page 260).

The commands `phastwincladfile` (page 293) and `impliedalignment` (page 263) output WINCLADA/PHAST-readable alignments based on the best cladograms that results from a POY search (Wheeler 2003b).

The command `topofile` (page 322) specifies a starting topology that begins a search from specific cladograms. This strategy is commonly used with `treefuse` (page 327) for "quick and dirty" searches as illustrated in the requirements section. When using `topofile`, cladograms must be in parenthetical notation just as in POY output, using full names of terminals and with each cladogram followed by [n] (an optional integer representing the cost). The entire list of cladograms must be terminated with a semicolon. As an alternative to topofile, one can enclose the entire nested parenthetical string representing a single cladogram between double quotes within a command line. Multiple cladograms can be separated with * as treated with examples in "Input Cladograms" on page 156.

Although we do not recommend using a starting topology as a search strategy unless significant cladogram building and refinement replicates have already been conducted, starting cladograms can be subjected to

ratcheting with the commands `ratchettbr` (page 309) and `topofile` (page 322).

A starting topology file can also be used to determine jackknife support. For example, the script below performs 1000 pseudo-replicates (sensu Farris et al. 1996) that serve as support measures for nodes:

```
poy -replicates 1000 -jackboot -jackfrequencies all
-jacktree -topofile filename > stdout 2> stderr
```

The program jack2hen (source `jack2hen.c` or binaries available via `ftp://ftp.amnh.org`) converts POY output (one or more cladograms in nested parentheses format) into a matrix in Hennig86/Nona format of group inclusion characters and computes consensus cladograms of various strictness levels, for example, the script

```
jack2hen n < poy_output_file > jack2hen_output_file
```

which converts POY output into a consensus cladogram based on an integer argument ("n") representing a percentage—100 for strict consensus, lower numbers for less strict structures. The strict consensus cladograms files can also be generated in POY with the command `poystrictconsensustreefile` (page 301). Majority rule consensus cladograms can also be obtained directly from POY in ASCII art format with the command `printtree` (page 304).

The constraint portion of the jack2hen output can be used to calculate Bremer values in POY (refer to `bremer` on page 220). In POY under bremer, these calculations are performed for all groups at once by TBR swapping based on group inclusion characters in the constraints file by using the command string `-bremer -constrain jack2hen_output_file`. These files can also be generated using `poystrictconsensustreefile` (page 301).

Global Bremer support calculations might lead to overestimates of group support because POY calculates these values based on a single TBR branch swap sample. Alternative strategies (introduced in Frost et al. 2001a) to calculate Bremer values on a group-by-group basis using `disagree` (page 238) are discussed in Chapter 7.

Strict rule consensus cladograms can also be obtained directly from POY in ASCII art format with the command `printtree` (page 304).

13 POY Input and Output

This chapter describes the data input and results output supported in POY. POY analysis results and other output are described starting on page 161.

General Format Requirements

POY input files must be formatted and output files are written in plain text—that is, devoid of control and hidden characters. Use a text editor appropriate to your operating environment to create input files. POY will halt processing and describe the problem via an error message if it is unable to read an input file.

Input Data

This section describes the four data input formats supported in POY and their usage.

HENNIG86/NONA Format

POY uses standard HENNIG86/NONA input data files for phenotypic data. HENNIG86 and NONA are applications used for the analysis of phenotypic or prealigned sequence data. This file format is treated in detail at `http://www.gwu.edu/~clade/faculty/lipscomb/web.pdf`.

Similarly, many applications such as TNT, PHAST, and WINCLADA have adopted modified versions of this format for phenotypic and prealigned sequence data. Bear in mind that the purpose of this format in the context of POY is to accommodate phenotypic data in combined analysis with unaligned nucleotide data. Other programs are better optimized to run phenotypic and prealigned data if that suits your purposes. The following is an example and list of guidelines for HENNIG86/NONA format suited to POY.


```
xread
'sample file'
5 4
Anagallis 00000
Rudbeckia 10000
Quercus   11000
Cichorium 1111[12]
;
proc/ ;
```

1. Start a HENNIG86/NONA format file with an `xread` segment. This segment includes the character matrix and its description.
2. Following the word `xread`, use two nonoptional single quotes (`' '`) in which to enclose optional comments.
3. Following the quotes, insert the number of characters and the number of taxa separated by a space (for example, `4 5`).
4. Next insert the list of the taxa and their character data in several lines using the following elements for each line.

- A continuous string for the taxon name. The name of a taxon may include letters and numerals but the first character must be a letter (for example, `Anagallis`).
- At least one space or tab; but many of these can be used to make visually appealing columns of data.
- Characters must be numerals ranging from `0` to `9`. An unknown or missing state may be represented by either `?` or `-` (the latter often used for inapplicable). Polymorphisms must be enclosed between square brackets—for example, `[12]`. See *Cichorium* data in the previous example.
- After the last taxon, an `xread` segment is closed by a semicolon (`;`). At this point, the user can end the entire file with `proc/;` and have a working file such as the example on page 148.

However, an optional `tread` segment that specifies topologies to be diagnosed can be included after the `xread`. The format of topologies are nested parentheses with taxa presented in numerals based on their order in the `xread` segment starting from 0, separated without commas. Topologies must end with a semicolon.

Weighting of characters and specifications about additivity can be accomplished with a `ccode` segment. This segment summarizes character data. `ccode` statements are scanned from left to right, and whatever code specifiers are currently in effect are applied to the characters in the scopes as they are read. Besides `[,]`, `+`, `-`, and `/`, `ccode` also accepts `*`, which discards all previous specifiers in a single `ccode` statement. Left and right parentheses—`(` and `)`—used in SPA (Goloboff 1996d) and PHAST (Goloboff 1996c; see also Goloboff 1998) to toggle Sankoff characters, are read but ignored otherwise. The default settings are `[-/1`. So `ccode /2 0.2 ] 3 4 * + 3;` will turn character 4 inactive and assign weight 2 to characters 0, 1, 2, 4, and 5. Character 3 is made active but otherwise keeps its default setting `[/1`. The following is an example of HENNIG86/NONA format that incorporates `ccode` and `tread` options, as well as polymorphisms.

```
xread
''
5 4
Anagallis     000000
Rudbeckia     [01]0000
Quercus       11000
Cichorium     11111
;
ccode /2 0.2 ] 3 4 *+ 3;
tread
(0 (1 (2 3)));
proc/ ;
```

`printccode` *output*

The output from `printccode` (page 302) is written in HENNIG86/NONA ccode format. The output is a summary of character settings in each HENNIG86/NONA format input data file. The output is written to the standard output file.

Structure

The `ccode` output has the following structure:

1. Character settings are written horizontally in the order in which they occur in the input data file.
2. The following table describes the meaning of values in each position.

Position	Description	Values
1	Character number	An integer with the first character = 0
2	Activity	[= active] = not active
3	Additivity	+ = additive – = nonadditive
4	Weight	/n where n = character weight
5	Polymorphism	p = polymorphic characters otherwise blank

Example

For the following input data

```
xread
''
4 5
Cichorium 0[01] 110
Rudbeckia 01    101
Quercus   1?    ?29
Anagallis 02    032
;
tread
'input cladograms'
(3 (0 1 2))
;
ccode +.-1;
ccode ] 2 /2 3.;
```

```
tread
'more input cladograms'
((0 1) (2 3))*
(0(1(2 3)));
p/;
```

`printccode` writes the following to the standard output file

```
0[+/1  1[-/1p  2]+/1  3]+/2  4]+/2
```

described in the following table:

Character	Description
0	active, additive, weight = 1
1	active, non-additive, weight = 1
2	not active, additive, weight = 1
3	not active, additive, weight = 2
4	not active, additive, weight = 2

Sankoff character data

Character data can be entered in a modified HENNIG86/NONA format that follows much of the SPA format of Goloboff (1996d). This allows the user to do two things not possible with HENNIG86/NONA format: (1) use more than nine character states (the number is only limited by the integer size of the platform), and (2) specify arbitrary state transformation cost matrices.

There are four modifications that are required to the Hennig96/NONA format. First, `xread` is replaced by `dpread` at the top of the file. Second, character states must be separated by spaces (to allow more than one digit in character states). Third, the states in a polymorphic character must be separated by ".". Fourth, the state transformation cost matrix is placed at the bottom of the file in the ccode block, preceded by a line containing the word "costs," a "["or "]" to define whether that a character is included, the characters to which the cost matrix should be applied, and a dollar sign "$" followed by the number of states in the character. This is followed by a formatting line with the state numbers. Additive and nonadditive characters can be intermingled with the matrix characters but are still subject to the nine-state limit.

Example

```
dpread
'test Sankoff data'
2 4
one   0 [0.1] 2
two   1 1     0
three 2 2     1
four  3 3     2
;
costs [ 0.1 $4
0 1 2 3
1 1 2 4
1 1 2 4
2 2 1 4
4 4 4 1
;
proc /;
```

In the above example, characters 0 and 1 would have the cost matrix applied while character 2 would be additive. This type of character can be used to code additive and nonadditive characters with large numbers of states.

Nucleotide data

Nucleotide sequence data in POY is represented using IUPAC (International Union of Pure and Applied Chemistry) ambiguity codes included in the table below:

Symbol	Meaning	Origin of designation
G	G	Guanine
A	A	Adenine
T	T	Thymine
C	C	Cytosine
R	G or A	puRine
Y	T or C	pYrimidine
M	A or C	aMino
K	G or T	Keto
S	G or C	Strong interaction (3 H bonds)

W	A or T	Weak interaction (2 H bonds)
H	A or C or T	not-G, H follows G in the alphabet
B	G or T or C	not-A, B follows A
V	G or C or A	not-T (not-U), V follows U
D	G or A or T	not-C, D follows C
N	G or A or T or C	aNy
X	N or Gap	

The data can be entered as either POY's simple block format or simplified FASTA format, which is often available from GenBank.

POY Block Format

Each entry in POY block format consists of

- a taxon name
- followed by a return
- followed by an unbroken string of bases conforming to IUPAC codes
- followed by a carriage return.

Do not include any gaps or dashes in sequences. Prealigned data can be used if all sequences do not include gaps and are exactly the same length. If the sequence data for a taxon are missing, the sequence name must be followed by two carriage returns. For example,

```
Anagallis¶
ACGAATAGC¶
¶
Rudbeckia¶
AGGTTTAGAG¶
¶
Cichorium¶
¶
Quercus¶
TUTUTUTUU¶
¶
```

In summary, the following are the rules for POY block format:

1. The file(s) must be formatted in plain text with returns suited to your operating system.
2. Each taxon and data consists of three lines:
   - a string representing the taxon name followed by a return
   - a string representing sequence characters followed by a return

- a return separating taxa (the last taxon does not require this separator).

3. Do not place spaces or underscores in taxon names.
4. A sequence must consist of an unbroken string of IUPAC codes.
5. Do not use gap characters in sequences.
6. If a taxon has no sequence data, enter a return followed by a blank line.

Multiple loci

Very long sequences can be optimized in POY. However, CPU time and memory requirements will increase quadratically with the length of sequences (discussed in Phillips et al. 2000). Thus cutting the loci into fragments can save dramatically on computation time and space (Giribet 2001, 2002). Although this is not a necessity, standard practice for POY users is to delineate regions as they are flanked by primer pairs or by secondary structural features or separate loci, treating each as a character in POY. This procedure is treated in detail in Chapter 9, Guidelines for Research. In addition to speeding up the calculations, this practice allows for more specific user-controlled hypotheses of putative homology between regions to be expressed. In the event that the user chooses the block format, each sequence character is contained in an individual file. In order to prepare these files, data can be imported from many formats to a visual sequence editor. Useful editors can be found for various platforms.

Platform	Visual Sequence Editor
Windows	http://jwbrown.mbio.ncsu.edu/BioEdit/bioedit.html
Mac OS X and classic	http://evolve.zoo.ox.ac.uk/software/Se-Al/main.htm
Solaris and Linux	http://bimas.dcrt.nih.gov/gde_sw.html
Mac OS X	http://www.msu.edu/~lintone/macgde/main.html

In addition to the issues addressed above, multiple block format data files can be used to accommodate multiple loci. This practice facilitates analysis of loci in various combinations and with phenotypic data. When using more than one input data file, make sure they conform to the following requirements:

- Taxon name strings must match exactly including capitalization.
- Each file must have the same taxa (unless using a terminals file). To have POY resolve inconsistencies among files, use `terminalsfile` (page 320).

- The order of taxa in each file does not have to match but the first taxon in the first file is the default outgroup.
- The user is advised to input phenotypic data first on command lines. Doing so will make the numbering system in which characters were labeled in the data file correspond to the numbering system used in the character diagnosis output file.

FASTA Format

FASTA is a program used to compare protein and nucleotide sequence data and is a common format for GenBank nucleotide records. POY accepts concatenated GenBank records as data if they are FASTA formatted and the taxon names are POY legal. In addition, POY implements enhancements to the FASTA format that enable the user to take the following actions:

- Include comments with taxon names (the comment feature can be useful to encapsulate verbose accession information from GenBank).
- Specify whether gaps are ignored.
- Specify sequence fragments.
- Specify which sequence fragments are excluded in an analysis.

For example, the command `fastashortname` (page 252) directs POY to take only the first contiguous string following the marker > as the taxon name. For example, `>taxona Comments on Taxon A` causes POY to read the taxon name as `taxona`. On the other hand, in the absence of the command, the taxon name is read as `taxona_Comments_on_Taxon_A`.

The command `deletegapsfrominput` (page 238) causes POY to ignore gaps coded in sequences. For example, the sequence `acct-ggt-tc` is read as `acctggttc`.

Multiple loci

In contrast to block format, FASTA format can accommodate multiple loci or fragments in a single file. In FASTA format, sequence fragments can be delineated by placing the symbol `#` at the fragment boundary. Note that the beginning and end of the sequence do not require demarcation. For example, the string `aact#gcggg` parses into two fragments: `aact` and `gcggg`. On the other hand, the string `#aact#gcggg#` parses into four fragments: a missing fragment, `aact`, `gcggg`, and another missing fragment. The number of fragments must be the same for all taxa.

Sequence fragments or loci can be excluded from the analysis by placing the symbol `*` in any position within the fragment to be deleted. For example, the string `agag*t#tcccag` parses into two fragments:

agagt and tcccag. The first fragment is ignored in the analysis. To take effect, the * symbol need only be placed in one fragment—corresponding fragments in other sequences do not require it, but they all are deleted. For example, the following FASTA formatted file

```
>taxona
acc#aacgt-t#tttttt#a*t
>taxonb
*#aacgtcc#ttt*tt#
```

will be read as

	Fragment			
Taxon	**1**	**2**	**3**	**4**
taxona	acc	aacgtt	tttttt	at
taxonb	missing	aacgtcc	ttttt	missing

In the analysis, fragments 1, 3, and 4 are ignored.

Input and output are also discussed in Chapter 12, The POY Interface. Some issues regarding specific formats are described below.

Input Cladograms

POY standard output is a specification of the optimal cladograms in parenthetical format. POY topologies do not use commas as separators. Minimally, POY output consists of the topology followed by the cladogram cost in brackets, terminated by a semicolon. For example,

```
(Anagallis (Cichorium (Rudbeckia Quercus))) [120];
```

If the user specifies POY's minimal output (the most parsimonious cladograms in parenthetical format) or uses poytreefile (page 301) these files can often be reused in long scripts that employ cladogram fusing or other operations on intermediate results. Moreover, although POY topologies do not use commas as separators, the user can easily replace spaces with commas required by some visualization programs.

Input cladograms read by the command topofile (page 322) can be specified either in the format written by POY to the standard output file or in HENNIG86/NONA format using tread. When entered on the command line using topology (page 323), enclose POY parenthetical cladogram format in quotation marks. For example,

```
poy data -diagnose -topology "(Anagallis (Cichorium
Rudbeckia Quercus));" > stdout 2> stderr
```

When entered on the command line using `topology`, more than one cladogram can be specified. Separate cladograms using either `*` or `[n]` (where `n` is an integer). The cladograms must be terminated with a semicolon. For example,

```
poy data -diagnose -topology "(Anagallis (Cichorium
Rudbeckia Quercus)) * ((Cichorium Rudbeckia) (Quercus
Anagallis)) * (Cichorium (Rudbeckia (Quercus
Anagallis)));" > stdout 2> stderr
```

Other Input Files

This section describes other input data formats accepted by POY.

Constraint file

Constraint files specify clades that must be recovered or avoided by an analysis. A constraint file must be in plain text, written in HENNIG86/NONA format using the `xread` and `proc` segments. However, instead of character state values, the data are binary characters that represent a taxon's inclusion in various clades. Constraint files are used with the commands `agree` (page 217), `constrain` (page 233), and `disagree` (page 238). For example, the command line and constraint file

```
poy data -agree -constrain constraintfile > stdout 2>
stderr
```

where `constraintfile` is

```
xread
'example of constraint file'
2 4
Cichorium 01
Rudbeckia 00
Quercus   11
Anagallis 11
cc- 0.1;
proc /;
```

recovers the lowest cost cladogram that includes the clades (Quercus Anagallis) and (Cichorium Quercus Anagallis).

Conversely, the command line

```
poy data -disagree -constrain constraintfile > stdout
2> stderr
```

recovers the lowest cost cladogram that does not include the clades (Quercus Anagallis) and (Cichorium Quercus Anagallis).

Constraint files are also used to estimate Bremer support values for clades with `bremer` (page 220) or `disagree` (page 238). Refer to "Bremer Support" on page 92.

Convert POY output cladograms to constraints

It is often the case that you will want to include constraints that are the output of a POY analysis. To convert POY output to a constraint file in HENNIG86/NONA format, use the utility Jack2Hen. The Jack2Hen output consists of a consensus cladogram based on an argument `n` representing a percent, as in 100 for strict consensus and lower numbers for less strict structures. The syntax of Jack2Hen is discussed in Chapter 12.

Terminals file

The command `terminalsfile` (page 320) can be used in conjunction with data or input cladograms.

Usage with data

The command `terminalsfile` creates an authoritative list of the names of taxa used in an analysis. Taxa that are included in the terminals file but not in some input data files are treated as having missing data. Taxa that are not in the terminals file but are included in an input data file are not included in the analysis.

Terminals files must be in plain text with white space after each taxon in accordance with your operating system. When using `terminalsfile`, an overview of taxa included and excluded is printed to stderr for each input data file. As such, `terminalsfile` is useful as a debugging tool. This feature reduces the need to edit data files. Without `terminalsfile`, all input data and cladograms need to have exactly the same complement of taxa. Thus taxa for which data is missing have to be indicated in each data file.

To prevent running an analysis with too much missing information, the following check is performed for each data file. If the number of taxa to be added from the terminals file is more than 20% of the number of taxa that are in the data file (after discounting the taxa that are not in the terminals file), the program exits. Use `minterminals` (page 282) to change this threshold to another percentage (`-minterminals 0` skips the test).

Usage with input cladograms

If the user invokes the command `terminalsfile` with input cladograms (see `topology` (page 323) and `topofile` (page 322)), taxa not in the terminals file will be pruned from the cladogram. Conversely, whenever an input cladogram lacks a terminal that is in the terminals file, POY adds that terminal as a basal branch to the cladogram. Without `terminalsfile`, all input topologies must conform exactly to the set of taxa that are in use.

Frequencies file

The command `basefreq` (page 219) reads base and indel frequencies from a file for use in likelihood analysis (see `likelihood` (page 274)). The file has the following format:

```
pctA  pctC  pctG  pctT  pctI ;
```

where

`pctA` = frequency of Adenine expressed as a decimal

`pctC` = frequency of Cytosine expressed as a decimal

`pctG` = frequency of Guanine expressed as a decimal

`pctT` = frequency of Thymine/Uracil expressed as a decimal

`pctI` = frequency of indel/gap expressed as a decimal

For example,

```
0.2  0.1  0.3  0.35  0.05;
```

The command `printqmat` (page 304) used under `likelihood`, prints transition, base frequency, and other likelihood parameter information to stdout.

Command file

Command files contain segments of command lines that can be inserted in a command line using `commandfile` (page 232). Command files have the following format:

$\texttt{command}_1\ \texttt{argument}_1\ \ldots\ \texttt{command}_n\ \texttt{argument}_n$

where

$\texttt{command}_i$ = the i^{th} command

$\texttt{argument}_i$ = the argument to the i^{th} command

Example

If the file `printtrees` has the following content

```
-printtree -plotstrict on -plotfile treeplot.txt
```

then the command line

```
poy chel.seq -commandfile printtrees
```

expands to

```
poy chel.seq -printtree -plotstrict on -plotfile
treeplot.txt
```

Molecular matrix file

This file is used by `molecularmatrix` (page 283) to specify the nucleotide transformation costs. The file consists of five rows and columns. Values occur in the following order: adenine, cytosine, guanine, thymine/uracil, and gap. The costs must be symmetrical (that is, C to T = T to C) and metric (that is, the cost of C to A is less than C to T + T to A). For example

```
0 2 1 2 4
2 0 2 1 4
1 2 0 2 4
2 1 2 0 4
4 4 4 4 0
```


is a cost matrix that has the following structure:

From \ To	A	C	G	T	gap
A	0	2	1	2	4
C	2	0	2	1	4
G	1	2	0	2	4
T	2	1	2	0	4
gap	4	4	4	4	0

Likelihood cost matrix file

This file is used by `qmatrix` (page 305) to specify the nucleotide transformation probabilities and indel probabilities for likelihood analysis. The file consists of five rows and columns. Values occur in the following order: adenine, cytosine, guanine, thymine/uracil, and gap. For example,

```
0.9  0.02 0.05 0.02 0.01
0.02 0.9  0.02 0.05 0.01
0.05 0.02 0.9  0.02 0.01
0.02 0.05 0.02 0.9  0.01
0.01 0.01 0.01 0.01 0.96
```

is a cost matrix that has the following structure:

		To				
		A	**C**	**G**	**T**	**gap**
From	**A**	0.9	0.02	0.05	0.02	0.01
	C	0.02	0.9	0.02	0.05	0.01
	G	0.05	0.02	0.9	0.02	0.01
	T	0.02	0.05	0.02	0.9	0.01
	gap	0.01	0.01	0.01	0.01	0.96

Results and Other Output

This section describes how POY reports the results of an analysis and other output.

POY topology

POY standard output is a specification of the optimal cladograms in nested parenthetical format. The output consists of the topology followed by the cladogram cost in brackets, terminated by a semicolon. For example,

```
(Anagallis (Cichorium (Rudbeckia Quercus))) [120];
```

Use `repintermediate` (page 311) to print cladograms resulting from each replicate as they finish. This is especially useful to get information when performing long searches.

Outputting cladograms

The command line

```
poy data -poytreefile filename > stdout 2> stderr
```

outputs cladograms and their binary representations and any other information, such as implied alignments and diagnoses (if specified) to the stdout. In addition, it will also output only parenthetical cladograms (not binaries) to the file named `filename`.

Alternatively, if the user wants only binary reprehensions, use `poybin-treefile` (page 300). Having a file that contains only cladograms can be useful for reuse of cladograms from intermediate searches, such as in the context of a tree-fusing analysis.

Consensus techniques

POY has various commands that can be used for calculating majority rule and strict consensus cladograms. For example,

```
poy data -poystrictconsensustreefile filename >
stdout 2> stderr
```

writes the strict consensus cladogram of all the lowest cost cladograms resulting from the search in POY parenthetical notation to file `file-name`.

```
poy data -printtree > stdout 2> stderr
```

writes the lowest cost cladograms and their strict consensus at the end of a POY run to default file `poy.tree` in ASCII art. The output of `printtree` (page 304) can be modified with commands such as `plotmajority` (page 296), which can work in the following modes:

`plotmajority off` the majority rule consensus cladogram is not plotted

`plotmajority short` the majority rule consensus cladogram is plotted without clade identification numbers

`plotmajority long` the majority rule consensus cladogram is plotted with clade identification numbers (each branch is labeled with the identification number and the percentage-wise clade frequency).

When conducting a jackknife search in POY (described in greater detail in "Jackknife Resampling" on page 91), a strict consensus is calculated automatically for each pseudo-replicate. In addition, a 50% majority rule consensus of all pseudo-replicate cladograms can be output via the use of the command `jackftrees` (page 269). For example,

```
poy data -replicates 1000 -jackboot
-jackfrequencies all -jackftree filename > stdout 2>
stderr
```

In the context of `jackboot` (page 266), default output can be changed with the following commands:

- `jackoutgroup` (page 270) sets the single taxon to be used as outgroup for jackknifing

- `jackpseudoconsensustrees` (page 270) used with `jackstart` (page 271) writes the strict consensus cladograms for the individual pseudo-replicates output to stdout.
- `jackfpseudoconsensustrees` (page 268) used with `jackstart` (page 271) creates the file `filename` with a `tread` statement that describes the strict consensus cladograms of the individual pseudoreplicates.

Hypothetical ancestral states

POY writes hypothetical ancestral sequence reconstructions with resolved ambiguities of the best cladogram to the file named `poy.hypanc` or a file name set by `hypancfile` (page 260). For example,

```
GTGACGATAAATAACGATCCGGAACTCTAATGAGTTTCCGTAATCGGAATGAGTACAATTTAAATCCGTTAACGAGGAGC
GTGACGAAAAATAACGATACGGAACTCAATCGAGTCTCCGTAATCGGAATGAGTACACTTTAAATCCTTTAACGAGGATC
GTGACGAAAAATAACGATACGGAACTCAATCGAGTCTCCGTAATCGGAATGAGTACACTTTAAATCCTTTAACGAGGATC
GTGACGAAAAATAACAATACAGGACTCAATCCGAGGCCCTGTAATTGGAATGAGTACACTTTAAATCCTTTAACAAGGAA
GTGACGAAAAATAACGATACGGAACTCTAATGAGTCTCCGTAATCGGAATGAGTACAATTTAAATCCTTTAACGAGGATC
GTGACGAAAAATAACGATACGGAACTCTAATGAGTCTCCGTAATCGGAATGAGTACAATTTAAATCCTTTAACGAGGATC
GTGACGAAAAATAACAATACGGGACTCTATCCGAGGCCCCGTAATTGGAATGAGTACACTTTAAATCCTTTAACGAGGAT
GTGACGAAAAATAACAATACAGGACTCATATCCGAGGCCCTGTAATTGGAATGAGTACACTTTAAATCCTTTAACAAGGA
GTGACGATAAATAACGATCCGGAACTCTAATGAGTTTCCGTAATCGGAATGAGTACAATTTAAATCCTTTAACGAGGATC
GTGACGATAAATAACGATCCGGAACTCTAATGAGTTTCCGTAATCGGAATGAGTACAATTTAAATCCTTTAACGAGGAGC
GTGACGATAAATAACGATCCGGAACTCTAATGAGTTTCCGTAATCGGAATGAGTACAATTTAAATCCTTTAACGAGGAGC
;
```

This can be useful for generating sets of sequences for search-based optimization.

Phastwincladfile

When used with `impliedalignment` (page 263), the output from `phastwincladfile` (page 293) is written in WINCLADA format based on the best cladogram. Bear in mind that POY output cladograms are considered to be unrooted, so the basal node is never resolved. It is up to the user to define a root in a visualization program.

The output includes

- any phenotypic data and the implied alignment for the first binary cladogram in the buffer
- the topology for all cladograms
- nucleotide sequence data and phenotypic data separated by a blank
- nucleotide sequence data recoded to numeric values, to accommodate the HENNIG86/NONA format

Nucleotide	Numeric code
adenine	0
cytosine	1
guanine	2
thymine	3
gap	4

- polymorphisms enclosed in brackets—for example, N is coded as [0123]).

An example of a `phastwincladfile` is given below:

```
xread
'POY run based on 6 original characters but expanded
to 43 with nucleotides and other extensive data
zero-length branches are not collapsed in the cladograms
below'
43 4
Anagallis 00000232012030003001203112200131300320233 3
Rudbeckia 10000232012000003001203012200131003120231 3
Quercus   11000232012000003001203012200131003120231 3
Cichorium 11111232012000003001203012200131300320231 3

;
cc +0 +1 +2 +3 +4 (5.42 ) ;
cost=1 0/1 1 0/2 1 0/3 1 0/
```

When used without `phastwincladfile`, `impliedalignment` provides aligned bases in columns for each character as specified in the input files. For example:

```
Implied alignment from character 0
Homology lines 24
Cichorium  XGAGTGATGATGGTGTTGTGTGTT
Quercus    XGAGTGATGATGGGGTTGTGTGTT
Rudbeckia  XTTCTGATGATGCCGTTGTGTGCT
Anagallis  TG-CTGACGACGGTGTTGTTTGTT


Implied alignment from character 1
Homology lines 39
Cichorium  GCTCACAGCATACAATGCTAGTTAA-ACAAGGA-GATGA
Quercus    GCTCACAGCATACAATGCTAGTTAA-ACAAGGA-GATGA
Rudbeckia  GCTCACAGCATACAATGCTAGTTAA-ACAAGGA-GATGA
Anagallis  GTTCACAACACACAATGCTTG-TAA-AGATGGATGGGGA
```

Bear in mind that an implied alignment will be provided based on each most parsimonious cladogram; thus this command can generate voluminous output.

14 Tutorials

Before You Start

This chapter takes you through the steps for running POY. It shows you how to

- run POY from the command line ("Tutorial 1: Command Line Basics" on page 169)
- create and use scripts ("Tutorial 2: Scripts" on page 173)
- calculate Bremer supports ("Tutorial 3: Bremer Support" on page 175)
- weight nucleotide transformations and analyze multiple loci ("Tutorial 4: Step Matrices and Multiple Data Sets" on page 176)
- perform combined analysis of phenotypic and genotypic data ("Tutorial 5: Combined Analysis" on page 178)
- input starting cladograms ("Tutorial 6: Starting Cladograms" on page 180)
- perform fixed states and search-based optimization ("Tutorial 7: Fixed States and Search-Based Optimization" on page 182)
- analyze chromosomal characters ("Tutorial 8: Chromosomal Optimization" on page 183)

- perform likelihood analysis ("Tutorial 9: Maximum Likelihood Optimization" on page 187)
- run multiple scripts for sensitivity analysis ("Tutorial 10: Multiple Scripts" on page 192)
- run POY in a parallel environment ("Tutorial 11: Parallel Environments" on page 194)
- and run POY using scheduling software ("Tutorial 12: Scheduling Software" on page 196).

These tutorials are intended for new POY users as well as for users who have previously used POY only on a single CPU and want to use POY in a parallel environment. While the tutorials discuss strategies for analysis, their main purpose is to provide you with a greater familiarity using POY. We strongly encourage you to explore POY's additional options and develop your own search strategies. The most effective search strategy is different for each data set—there are some rules of thumb, but no guaranteed optimal strategies. For a fuller discussion of analysis strategies, refer to Chapter 9 Guidelines for Research.

In addition to the POY executable, the only other application software you need for these tutorials is a text processor, such as Textpad, BBEDIT, EMACS, Word, WordPad, or NotePad, and the utilities `jack2jen` and `filterstates.exe` (for Windows).

Preliminary tasks

1. Make sure you have downloaded all of the POY files as described in Chapter 11 Installation on page 127.
2. Create a folder named `Tutorial`.
   - Although you can place this folder anywhere, we suggest you make it a subfolder to the `POY` folder you created during installation.
   - In addition, nothing depends operationally on this new folder being named `Tutorial`.
3. Copy the POY executable from the `POY` folder to the new Tutorial folder.
4. Copy the `jack2jen.exe` program from the `POY` folder to the new Tutorial folder.
5. Copy all of the files from the `POY`>Tutorial folder to the new Tutorial folder.
   - The subfolders are environment-specific: `Tutorial-dos` for Windows, `Tutorial-linux` for Linux, and `Tutorial-mac` for Unix operations on Macintosh OS X.
   - These files must be in the same folder as the POY executable you use in the tutorials.

6. Review Chapter 12 The POY Interface on how to launch POY and how to create and edit scripts in your operating system.
7. Review Chapter 13 POY Input and Output on POY file formats.

Tutorial 1: Command Line Basics

This tutorial illustrates the basics of running POY from the command line. Although typically you will want to use scripts, the basics described in this tutorial familiarize you with how POY works.

A single input file with default settings

1. Navigate to the Tutorial folder.
2. At the system prompt, type the following command line and press Enter:

   ```
   poy mol1.txt
   ```

   Note that the input data file `mol1.txt` is unaligned genotypic data in FASTA format, so that only analyses on genotypic data are performed.

   To run, POY only requires the executable and one data file.

   The most relevant default options for this run are

   - a single, order-specified build of the initial tree (that is, the initial Wagner tree is built using the order in which the taxa are read from the input file)
   - direct optimization is used for calculating tree cost
   - indels are assigned a weight of 2 while transitions and transversion are assigned a weight of 1
   - a single round of TBR branch swapping is performed to improve upon the original Wagner tree (that is, find shorter trees)
   - to stop the analysis at any time, press and hold `CTRL` and type `c`.

3. When the analysis is complete (approximately 77 seconds on our 850-MHz Pentium III machine), POY displays the program progress on-screen (Figure 14.1).

```
121 trees examined in initial construction.
0 trees examined in SPR.
1426 trees examined in TBR.
0 trees examined in Drift SPR.
0 trees examined in Drift TBR.
0 trees examined in RATCHET SPR.
0 trees examined in RATCHET TBR.
0 trees examined in Check Slop TBR.
0 trees examined in Bremer TBR.
0 trees examined in Tree Fusing.
1547 trees examined in all.
35 trees packed
1547 joins calculated
1. average join span
6037 alignments performed
Process took 77 seconds
```

Figure 14.1: Program progress displayed on-screen.

Output files

POY's standard output file contains the results of the analysis in POY parenthetical format. If only the output file is specified, the program progress still displays on-screen.

4. At the system prompt in the Tutorial folder, type the following command line and press Enter:

   `poy mol1.txt > basic1.out`

   - The expression `> basic1.out` sends the POY output to the file `basic1.out`.

5. When the analysis is complete, POY both displays the results on-screen (Figure 14.1) and writes the results of the analysis to `basic1.out` (Figure 14.2).

```
(Hagfish (Cow (Human ((Horn_Shark Saw_Shark)
((Salmon Goldfish) (Tree_Frog Newt))))))[3002]
;
Binary representations of best cladograms:
(Hagfish (Cow (Human ((Horn_Shark Saw_Shark)
((Salmon Goldfish) (Tree_Frog Newt))))))[3002]
```

Figure 14.2: Output file `basic1.out`.

Note that if POY finds zero length branches, it reports both

- the collapsed representations shown first (equivalent to `ambiguous-` in Nona and PAUP*)
- and binary representations of the optimal solution.

The binary representation is important for length verification. Implied alignments are calculated from the binary representation. Spurious length discrepancies can result if an implied alignment is optimized on a different binary version of the same collapsed tree

.

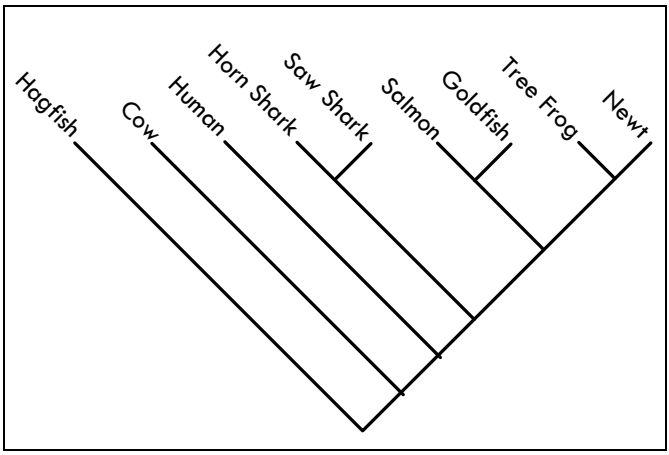

Figure 14.3: Cladogram of `basic1.out`.

Program progress output

If you are working in Linux, Windows NT, Windows XP, or Mac OS X you can specify a file to which POY writes program progress, including error messages.

Important If you use another version of Windows,

- skip this section
- remove `2> basic1.err` and equivalent commands (that is, `2> [filename].[ext]`) from the subsequent command lines in this tutorial
- also remove those commands from the scripts in the tutorials that follow.

6. At the system prompt in the Tutorial folder, type the following command line and press Enter:

```
poy mol1.txt > basic1.out 2> basic1.err
```

- The expression `> basic1.out` sends the POY output to the file `basic1.out`.
- The expression `2> basic1.err` sends program progress (that is, the log file or on-screen information) to the file `basic1.err`.

7. When the analysis is complete, POY writes both the results of the analysis to `basic1.out` (Figure 14.2) and program progress to `basic1.err`.
   - The content of `basic.err` is the same as Figure 14.1.
   - This information can only be displayed on-screen when running Windows 95, Windows 98, Windows 98 SE, Windows ME, or Windows 2000.

Change default behavior

All POY analyses are created by adding commands and input files to the basic command line you have been using. The following command line overrides POY's default behavior by causing the analysis to build the initial Wagner tree by randomizing the order of taxon addition. The default is to add taxa in the order in which they occur in the data input file.

8. At the system prompt in the Tutorial folder, type the following command line and press Enter:

```
poy mol1.txt -nooneasis > basic2.out 2> basic2.err
```

   - Note that all POY commands are preceded by a dash.
   - `nooneasis` is the countercommand to the default command `oneasis` (see Chapter 15 Command References for more on countercommands and `oneasis/nooneasis`).

9. Compare `basic2.out` and `basic2.err` to `basic1.out` and `basic1.err`.
10. At the system prompt, type the following command line and press Enter:

```
poy mol1.txt -nooneasis -replicates 5 > basic2.out
  2> basic2.err
```

    - The command `replicates 5` causes POY to perform the command line in step 1 above on five random addition sequences.
    - In most cases, the order in which commands occur does not affect POY's behavior.

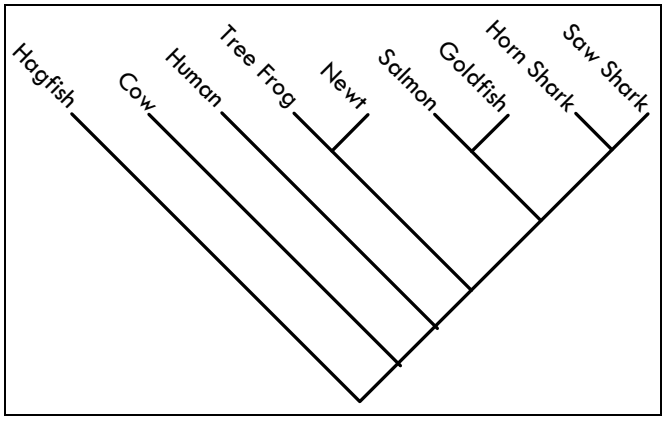

Figure 14.4: Cladogram of `basic2.out`.

Tutorial 2: Scripts

This tutorial introduces the use of scripts in batch files (also known as shell scripts)—the most common and useful method for running POY. A batch file can contain one or multiple scripts. In the latter case, consecutive POY jobs are run sequentially. We describe multiscript batch files in Tutorial 10 on page 192.

When using your text editor of choice in creating and editing batch files, make sure to save the file as plain text.

1. In your text editor, type the following command line:

```
poy mol1.txt -nooneasis -approxbuild -replicates 2
  -nonrandomizeoutgroup -slop 1 -checkslop 2 -exact
  > single1.out 2> single1.err
```

   - You can type this script as either one continuous line or you can break the line at appropriate places.
   - However, use only a single line break to start a new line of the script—POY reads two consecutive line breaks as the end of the command line.
   - This script develops the command line from Tutorial 1 by performing two random addition sequence replicates (`replicates 2`). By default, it uses direct optimization.
   - The command `approxbuild` (page 218) reduces calculation time during the building of a Wagner tree by using an approximation method for the initial builds. For some data sets, it is better to devote calculation time to branch swapping and other heuristics (e.g., ratcheting) than to initial builds.

- By default, TBR is performed. `slop 1` causes POY to perform TBR on all trees that are within 0.1% of the least cost tree instead of the least cost tree only.
- The command `norandomizeoutgroup` enables you to specify the root. This is important when applying the direct optimization algorithm because the placement of the root affects the optimization of the hypothetical ancestral sequences and, therefore the cost calculation. By default, if this command is invoked, the first taxon in the input file is designated as the root.
- After all replicates have completed, TBR branch swapping is performed on all trees found that are within 0.2% of the least cost tree (`checkslop 2`) instead of the shortest trees only.

2. Save the batch file in plain text format as `single1.bat`.
   - Make sure to save `single1.bat` to the Tutorial folder.
   - In order for Windows to recognize a file as a batch file, the file must have the extension `.bat`. This is not required when using other operating systems.
3. From the system prompt in the Tutorial folder, run the batch file.
   - In Windows, type `single1.bat` then press Enter, or double-click the file icon.
   - In Linux, type `sh single1.bat &` then press Enter. In Linux, the symbol `&` [ampersand] causes POY to run in the background.
4. When the script completes, compare your results to ours. They might differ due to the random order of taxon addition and few replicates.
   - In our analysis, this script results in a single least cost tree of 2,962 weighted steps ("weighted" because indels have twice the weight as transitions and transversions). Figure 14.5 shows our output file `single1.out`.
   - In our analysis, the tree is 40 steps shorter than the simple analysis in Tutorial 1. However, it takes significantly longer (198 seconds versus 77 seconds on our 850-MHz Pentium III).


```
(Hagfish (Cow (Human ((Tree_Frog Newt) ((Salmon
Goldfish) (Horn_Shark Saw_Shark))))))[2962]
;
Binary representations of best cladograms:
(Hagfish (Cow (Human ((Tree_Frog Newt) ((Salmon
Goldfish) (Horn_Shark Saw_Shark))))))[2962]
```

Figure 14.5: Output file `single1.out`.

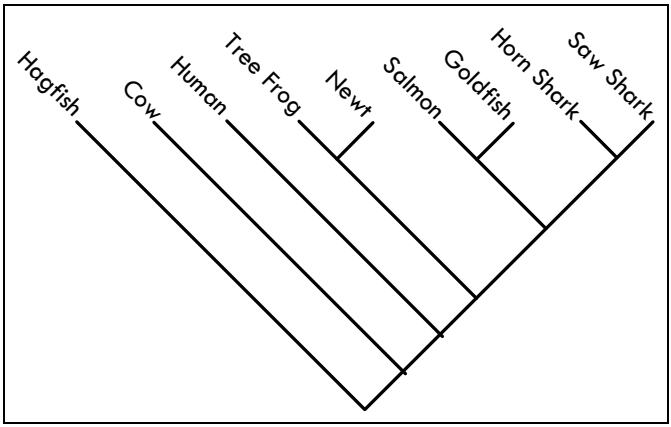

Figure 14.6: Cladogram of `single1.out`.

Tutorial 3: Bremer Support

This tutorial introduces the estimation of Bremer support values in POY. It will require the use of the `jack2hen.exe` program to generate a constraint matrix for the analysis.

Create constraint file

1. In the Tutorial folder, save a copy of `single1.out` (created in Tutorial 2) as `tree.txt`.
2. In your text editor, edit `tree.txt` to create a tree input file that is the binary representation of the best cladogram(s) from your analysis as shown in Figure 14.7 (compare to Figure 14.5).

```
(Hagfish (Cow (Human ((Tree_Frog Newt) ((Salmon
Goldfish) (Horn_Shark Saw_Shark))))))[2962]
```

Figure 14.7: Tree input file `tree.txt`.

3. Save the tree input file `tree.txt` as plain text format in the Tutorial folder.
4. In your text editor, type the following command line:

   ```
   jack2hen 100 < tree.txt > tree.con
   ```

   This script converts the binary representation of the best cladograms in `tree.txt` to the constraint file `tree.con` used for the evaluation of Bremer support. For more on constraint files, refer to "Constraint file" on page 157.
5. Save this file in the Tutorial folder and name it `j2h.bat` as plain text format.

6. Run the batch file `j2h.bat`.
7. When the conversion is complete, open `tree.con` in your text editor and delete all data after the matrix of taxa.
   - Keep the semicolon after the last taxon name.
   - Delete all `cc` lines, `tread` lines, tree in parenthetical format, and the final `p/;` line.
8. Save `tree.con` as plain text format in the Tutorial folder.

Estimate Bremer support

9. In your text editor, type the following command line:

```
poy mol1.txt -topology "(Hagfish (Cow (Human
  ((Tree_Frog Newt) ((Salmon Goldfish) (Horn_Shark
  Saw_Shark))))))[2962]" -bremer -constrain tree.con
  > bremer1.out
```

   - Note that the string in quotes consists of the topology and tree length output from Tutorial 2. If you are evaluating the topology from another analysis, change this string accordingly.
   - If POY finds a shorter tree during the Bremer calculation, it will report that shorter tree in the output file.
10. Save this file in the Tutorial folder and name it `bremer1.bat` as plain text format.
11. Run the batch file `bremer1.bat`.
    - When the POY run completes, `bremer1.out` will contain the Bremer support values for each of the nodes in your cladogram.

Tutorial 4: Step Matrices and Multiple Data Sets

This tutorial demonstrates how to use a step matrix to specify nucleotide transformation costs. It also introduces the analysis of multiple input data files.

The step matrix we use in this tutorial (`g1ts1tv1.txt`) has the structure shown in Figure 14.8.

	A	C	G	T	indel
A	0	1	1	1	1
C	1	0	1	1	1
G	1	1	0	1	1
T	1	1	1	0	1
indel	1	1	1	1	0

Figure 14.8: The structure of the step matrix `g1ts1tv1.txt`.

This step matrix changes the weighting of nucleotide transformations used in Tutorial 2: indels now have the same cost as transitions and transversions. An alternative is to set indel cost using the command `gap N`, where N is any positive integer; `gap 1` overrides the default setting of indels weighted twice the cost of transitions and transversions by setting the indel cost to 1.

However, use of a step matrix enables changing the weights for all three transformation costs. For example, another step matrix in the set we provide (`g4ts2tv1.txt`) has the structure shown in Figure 14.9. This step matrix weights indels at four, transversions at two, and transitions at one (refer to "Molecular matrix file" on page 160).

	A	C	G	T	indel
A	0	2	1	2	4
C	1	0	2	1	4
G	1	2	0	2	4
T	2	1	2	0	4
indel	4	4	4	4	0

Figure 14.9: The structure of the step matrix `g4ts2tv1.txt`.

1. In the Tutorial folder, save a copy of `single1.bat` (created in Tutorial 2) as `multiple1.bat`.
2. In your text editor, edit `multiple1.bat` to create the following script:

```
poy mol1.txt mol2.txt -molecularmatrix g1ts1tv1.txt
  -nooneasis -approxbuild -replicates 5
  -nonrandomizeoutgroup -slop 2 -checkslop 5
  -ratchettbr 5 -treefuse -exact > multiple1.out 2>
  multiple1.err
```

   This requires that you make the following changes to the original script:

   - Add the input data file name `mol2.txt`.
   - Add the commands `molecularmatrix g1ts1tv1.txt` (the file name `g1ts1tv1.txt` must immediately follow the `molecular-matrix` command), `ratchettbr 5`, and `treefuse`.
   - Change the output and program progress file names to `multiple1.out` and `multiple1.err`, respectively.
   - Change `replicates 2` to `replicates 5`, `slop 1` to `slop 2`, and `checkslop 2` to `checkslop 5`.

   This script causes POY to perform

   - five random addition replicates

- TBR branch swapping on trees within 0.2% of the least cost tree for each replicate, followed by five rounds of TBR ratcheting per replicate
- tree fusing on the best cladograms from each of the five replicates
- a final round of TBR branch swapping on trees within 0.5% of the least cost.

3. Save batch file `multiple1.bat` as plain text format in the Tutorial folder.
4. Run the `multiple1.bat`.

   In our analysis, this script results in a single least cost tree of 7,013 equally weighted steps. The analysis took approximately seven hours to complete on our 850-MHz Pentium III.

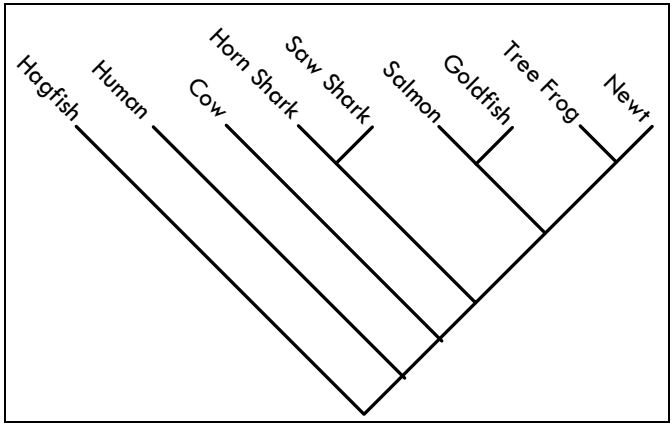

Figure 14.10: Cladogram of `multiple1.out`.

Tutorial 5: Combined Analysis

This tutorial demonstrates combined analysis in POY—the simultaneous analysis of phenotypic and genotypic data. It also demonstrates how to economize on calculation time by partitioning DNA sequence data into fragments.

For this tutorial, we use two new nucleotide sequence input data files: `mola1.txt` and `mola2.txt`. These contain the same data as the three used in Tutorial 3. However, these new input data files partition the sequences into fragments. In these files, the character # separates putatively homologous fragments based on primer sequences or other highly conserved markers (refer to "FASTA Format" on page 155).

Breaking contiguous sequences into fragments constrains nucleotide comparisons to within each fragment. This might result in a globally suboptimal solution. However, use of fragments increases calculation speed and reduces memory requirements significantly. The latter is of special concern when using iterative pass optimization, as we do in this tutorial. Moreover, the effect, if any, on cladogram cost calculations using the fragmented sequences can be evaluated by diagnosing the putatively optimal cladogram with the original, unfragmented sequences.

This tutorial also demonstrates combined analysis by incorporating phenotypic data in the file morph.txt, a file in Hennig86/Nona format (refer to "HENNIG86/NONA Format" on page 148).

For this tutorial, we build on the script you created in Tutorial 3.

1. In the Tutorial folder, save a copy of `multiple1.bat` (created in Tutorial 4) as `totalev1.bat`.
2. In your text editor, edit `totalev1.bat` to create the following script:

```
poy -weight 1 morph.txt mola1.txt mola2.txt -
  nooneasis -gap 1 -replicates 2 -checkslop 2 -
  iterativepass -phastwincladfile totalev1.ss -
  outgroup hagfish -exact > totalev1.out 2>
  totalev1.err
```

   This requires that you make the following changes to the original script:

   - Add the commands `weight 1`, `gap 1`, `iterativepass`, `phastwincladfile totalev1.ss`, and `outgroup hagfish`.
   - Add the input data file name `morph.txt`. Note that this new input file name must follow immediately after the new command `weight`.
   - Change the input data file names to `mola1.txt` and `mola2.txt`.
   - Change the output and program progress file names to `totalev1.out` and `totalev1.err`, respectively.
   - Delete the command `molecularmatrix g1ts1tv1.txt`.

   This script causes POY to

   - perform simultaneous analysis of all data partitions, including the phenotypic data in `morph.txt`;
   - perform equal weighting of nucleotide and morphological transformations using the command `gap 1` instead of the previously used command for step matrices `molecularmatrix`;
   - perform iterative pass optimization instead of the default direct optimization;

- use `outgroup hagfish` to override the default behavior of taking the first taxa in the input file as the outgroup—useful for evaluating different outgroups and when input files differ order taxa differently;
- generate a Nona file named `totalev1.ss` that contains the concatenated implied sequence alignments and phenotypic data set with the least cost trees appended—useful for optimizing phenotypic characters and verifying tree costs.

3. Save batch file `totalev1.bat` as plain text format in the Tutorial folder.
4. Run `totalev1.bat`.

   In our analysis, this script results in a single least cost tree of 7,345 steps. The analysis took two days on our 850-MHz Pentium III.

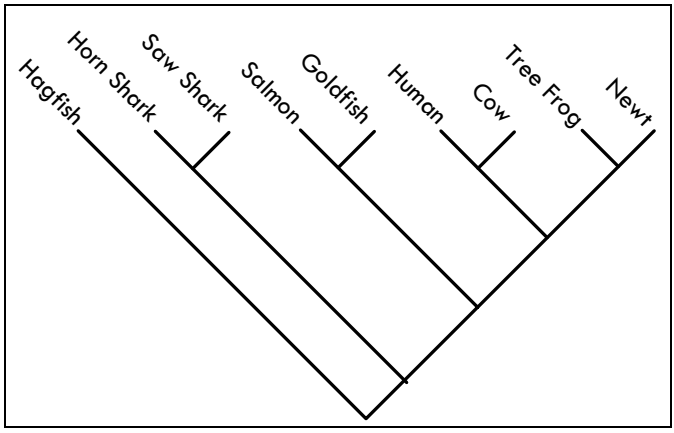

Figure 14.11: Cladogram of `totalev1.out`.

Tutorial 6: Starting Cladograms

This tutorial demonstrates how to use topologies from previous analyses as starting cladograms for POY to improve upon. Cladograms must be specified as described in "Input Cladograms" on page 156.

For this tutorial, we build on the script you created in Tutorial 5.

1. In the Tutorial folder, save a copy of `totalev1.bat` (created in Tutorial 5) as `totalev2.bat`.
2. In your text editor, edit `totalev2.bat` to create the following script:

```
poy weight 1 morph.txt mol1a.txt mol2a.txt -gap 1 -
  nooneasis -approxbuild -replicates 2 -
  norandomizeoutgroup -outgroup hagfish -slop 1 -
```

```
checkslop 2 -exact -iterativepass -
phastwincladfile totalev2.ss -topofile tree.txt -
printtree -plotfile totalev2.tree > totalev2.out
2> totalev2.err
```

This requires that you make the following changes to the original script:

- Add the commands `topofile trees.txt`, `printtree`, and `plotfile totalev2.tree`.
- Change the file name `totalev1.ss` to `totalev2.ss`.
- Change the output and program progress file names to `totalev2.out` and `totalev2.err`, respectively.

This script causes POY to

- perform simultaneous analysis of all data partitions
- perform SPR and TBR branch swapping on both the topologies contained in the file `trees.txt` and those obtained from two random addition replicates
- write a graphical representation of the minimum cost cladogram(s) to the file `totalev2.tree`.

3. Save `totalev2.bat` as plain text format in the Tutorial folder.
4. Run `totalev2.bat`.

   This analysis took 1.5 days on our 850-MHz Pentium III.
5. In your text processor, open `totalev2.tree` to view the graphical representation of the best cladograms.

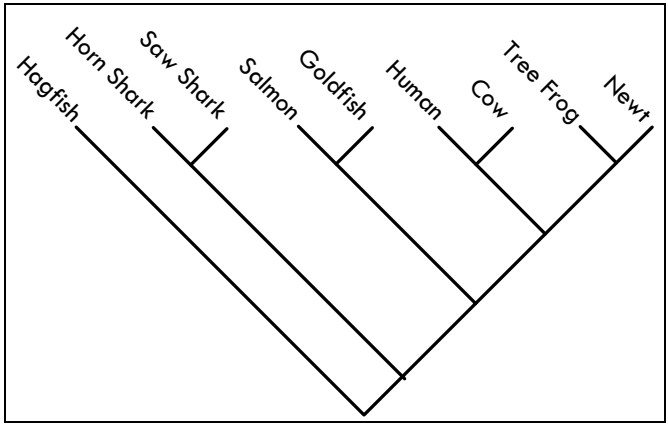

Figure 14.12: Cladogram of `totalev2.out`.

Tutorial 7: Fixed States and Search-Based Optimization

This tutorial demonstrates how to perform fixed states and search-based optimization. In "Tutorial 5: Combined Analysis" on page 178, the optimization algorithm was changed from the default option (direct optimization) to iterative pass optimization, which only required that the command `iterativepass` be specified. Switching to fixed states is accomplished in the same way, and search-based optimization is accomplished as a modification of standard fixed states optimization.

1. In your text editor, type the following command line:

   ```
   poy -fixedstates mol1.txt mol2.txt -gap 1 -
     nooneasis -replicates 20 > fixed.out 2> fixed.err
   ```

   This script causes POY to perform fixed states optimization on the data in `mol1.txt` and `mol2.txt`. Note that the command `fixed states` precedes the file names it is applied to. One can treat different data files in different ways by breaking up the filenames (for example, `poy mol1.txt -fixed states mol2.txt...` would perform direct optimization on `mol1.txt` and fixed states on `mol2.txt`).
2. Save the file as `fixed.bat` in the Tutorial folder.
3. Run `fixed.bat`.

   In our run, we obtained a single tree of cost 1,348.

Search-based optimization performs fixed states calculations on an expanded state set (that is, the set of potential hypothetical ancestral sequences). Therefore, running a search-based optimization analysis involves two steps:

- Defining (generating) the state set
- Calculating cladogram costs based on the set of hypothetical ancestral sequences.

The state set is generated by optimizing sequences on different cladograms and saving the resulting ancestral sequences to a file. The cladograms (and resulting state sets) may be derived by diagnosing one or a few input cladograms, by searching with a more or less exhaustive strategy, and by using either the direct optimization or iterative pass algorithms (do not use the fixed states option, as this will not expand the state set beyond the observed sequences). The state set must be calculated for each data set separately.

4. In your text editor, type the following scripts, each separated by a hard return:

   ```
   poy mol1.txt -gap 1 -nooneasis -replicates 20
     -printlotshypanc -hypancfile mol1.hyp¶
   ```

```
poy mol2.txt -gap 1 -nooneasis -replicates 20
  -printlotshypanc -hypancfile mol2.hyp¶
```

This will perform 20 random addition searches for each data file (using default settings for searches) and save all the hypothetical ancestral sequences from both final and intermediate cladograms to `mol1.hyp` and `mol2.hyp`, respectively.

5. Save this batch file as `hypanc.bat`.
6. Run `hypanc.bat`.
7. In your text editor, open `mol1.hyp` and `mol2.hyp`.

   In each file there is a set of sequences of variable lengths with all ambiguities arbitrarily resolved. At the end of each file insert a semicolon and save the results. The semicolon is required for POY to process the sequences in the next steps.

   It is not uncommon for sequences to be repeated. Including repeated states in the state set will slow down fixed states calculations, so it is recommended that they be filtered out. This may be done with an ancillary program called `filterstates.exe`, which should be in your Tutorial folder.
8. In your text editor type the following scripts, each separated by a hard return.

   ```
   filterstates < mol1.hyp > mol1a.hyp¶
   ```

   ```
   filterstates < mol2.hyp > mol2a.hyp¶
   ```

   Running these scripts will cause the program filterstates to examine each file, report the number of redundant sequences relative to the total number, purge them from the file, and save the results as `mol1.hyp` and `mol2.hyp`, respectively.
9. Save this file as `filter.bat`
10. Run `filter.bat`.

    We can now calculate the cladogram costs using the expanded state set.
11. In your text editor, type the following script:

    ```
    poy -fixedstates mol1.txt mol2.txt -newstates
      mol1a.hyp -newstates mol2a.hyp -gap 1 -nooneasis -
      replicates 50 > sbo.out 2> sbo.err
    ```

    In our run, this returned a single tree of cost 1,113.

Tutorial 8: Chromosomal Optimization

This tutorial illustrates chromosomal optimization, which allows the homology of both individual nucleotides and entire loci (for example, the genes of a chromosome) to be tested by optimizing both character

types simultaneously. As implemented in POY, the boundaries between loci must be known (assumed) *a priori*, with the occurrence of rearrangements, insertions, and deletions of those regions being the events inferred though phylogenetic analysis. Running a chromosome optimization analysis is straightforward, but the output differs from that of analysis of standard nucleotide data.

1. In your text editor type the following script:

```
poy -molecularmatrix 421 -chromosome data.chrom
  -linear -rearrange -diagnose -impliedalignment >
  chromo.out 2> chromo.err
```

   This simple script employs the following specified and default commands:

   - Calculates cost of nucleotide transformations using the Sankoff matrix 421, which specifies a cost of 4 for indels, 2 for transversions, and 1 for transitions
   - Builds initial Wagner tree in the order of terminals in `data.chrom`
   - Inputs the data file data.chromo and treats it as a linear chromosome
   - Allows rearrangements as well as insertions and deletions of fragments
   - Assigns default breakpoint cost of 10
   - Assigns default locus indel cost of 10, with no additional cost for locus length
   - Prints the cladogram diagnosis and implied alignment to the outfile.

2. Save the file as `chromo.bat`.
3. Execute `chromo.bat`.
4. In your text editor open `chromo.out`.

   This analysis resulted in two equally optimal cladograms of cost 112. Locus homologies for each of the optimal solutions are reported in three ways. First, beneath the standard diagnostic information POY reports the Annotation Table showing the relative adjacencies of each locus in each taxon (including HTUs):

```
-------- ANNOTATION TABLE --------
  Hagfish:     0|1|2|3|
  Tree_Frog:   0|1|2|3|4|
  Newt:        0|1|2|3|4|
  Cow:         1|2|3|4|
  Human:       1|2|3|4|
  Horn_shark:  1|2|3|4|
  Salmon:      0|1|2|5|3|4|
```

```
Goldfish:     0|1|2|5|3|4|
Sawshark:     1|2|3|4|
HTU0:         0|1|2|3|
HTU1:         0|1|2|3|4|
HTU2:         0|1|2|5|3|4|
HTU3:         0|1|2|3|4|
HTU4:         1|2|3|4|
HTU5:         1|2|3|4|
HTU6:         1|2|3|4|
HTU7:         0|1|2|3|4|
```

Loci are numbered in the order in which they are read from the data file, beginning with 0.

Second, the annotation table is followed by the presence/absence table that shows presence (+) and absence (-) of each locus in each taxon (including HTUs):

```
-------- PRESENCE/ABSENCE TABLE --------
                0  1  2  3  4  5
  Hagfish:      +  +  +  +  -  -
  Tree_Frog:    +  +  +  +  +  -
  Newt:         +  +  +  +  +  -
  Cow:          -  +  +  +  +  -
  Human:        -  +  +  +  +  -
  Horn_shark:   -  +  +  +  +  -
  Salmon:       +  +  +  +  +  +
  Goldfish:     +  +  +  +  +  +
  Sawshark:     -  +  +  +  +  -
  HTU0:         +  +  +  +  -  -
  HTU1:         +  +  +  +  +  -
  HTU2:         +  +  +  +  +  +
  HTU3:         +  +  +  +  +  -
  HTU4:         -  +  +  +  +  -
  HTU5:         -  +  +  +  +  -
  HTU6:         -  +  +  +  +  -
  HTU7:         +  +  +  +  +  -
```

Finally, the complete chromosomal implied alignment is reported:

```
--- Complete mitochondrial implied alignment from character 0 ---
    Locus 0
    Hagfish             AAAAA
    Tree_Frog           AGAAA
    Newt                CCCCA
    Cow                 XXXXX
    Human               XXXXX
    Horn_shark          XXXXX
```

```
Salmon              -AAAT
Goldfish            -AAAT
Sawshark            XXXXX

Locus 1
Hagfish             CCCCC
Tree_Frog           CTCCC
Newt                GTGGG
Cow                 GTGGG
Human               GTGGA
Horn_shark          ---AA
Salmon              C-AGT
Goldfish            C-AGT
Sawshark            --AAA

Locus 2
Hagfish             GGGGG
Tree_Frog           GAGGG
Newt                AATAA
Cow                 AATAA
Human               AAT-A
Horn_shark          TTCAT
Salmon              GAGGT
Goldfish            GACGC
Sawshark            TTAAT

Locus 3
Hagfish             TTTTT
Tree_Frog           TCTTT
Newt                -TCTT
Cow                 CCCCA
Human               TAGGG
Horn_shark          TTCAT
Salmon              GCTAA
Goldfish            GCT-A
Sawshark            TTCAT

Locus 4
Hagfish             XXXXXXX
Tree_Frog           -GA-GGG
Newt                -TA-GGG
Cow                 TT-ATTT
Human               -T--TTA
Horn_shark          -G--GGG
Salmon              -AA-ATT
Goldfish            -AT-ATT
```

```
Sawshark            -G--GGG

Locus 5
Hagfish             XXXXXX
Tree_Frog           XXXXXX
Newt                XXXXXX
Cow                 XXXXXX
Human               XXXXXX
Horn_shark          XXXXXX
Salmon              CCAGGT
Goldfish            CCATGT
Sawshark            XXXXXX
```

That is, loci inferred to be homologous are grouped together and the nucleotides within them related through transformation series, reported in the format of an alignment. Loci that are absent for a given taxa are reported as a string of Xs. Note that concatenating the implied alignments for the different loci and optimizing them on the inferred cladogram will not recover the same cost, as the cost of rearrangements and locus indels is not accounted for.

As in standard diagnosis and implied alignment printing, the information is provided subsequently for each optimal solution.

Tutorial 9: Maximum Likelihood Optimization

This tutorial demonstrates how to perform analyses under the maximum likelihood optimality criterion. That is, rather than evaluating cost as a weighted number of steps with relative weights determined in a Sankoff matrix, this evaluates cost as a likelihood score with relative change penalties determined in a transition probability matrix. POY calculates likelihood scores (as negative natural log likelihoods) as a function of a five-state (each nucleotide plus indel) Markov model; the model parameters may be specified *a priori* or estimated during the analysis. Other model parameters, such as the proportion of invariant sites (θ, theta), the shape parameter (α) of the gamma (Γ) distribution, and the rate classes, may also be specified or estimated. Because the alignment is not predetermined, there are several ways to estimate the likelihood score of a cladogram—that is, by calculating only the likelihood of the dominant optimization alignment (default `likelihood`), by summing all major optimization alignments (`totallikelihood`), or by summing all likelihoods (`trullytotallikelihood`). Likewise, maximum likelihood can be implemented using fixed-states optimization, search-based optimization, or iterative pass optimization, and all of the tree searching algorithms can be applied as well. All of these options are not demonstrated here, but a procedure is shown for estimating the

transition matrix and running a maximum likelihood analysis with estimated and specified parameters.

1. In your text editor type the following script:

   ```
   poy -likelihood mol1.txt mol2.txt > ml1.out 2>
     ml1.err
   ```

   This is the simplest script to run a maximum likelihood analysis using the default settings. Important default settings include the following:

   - Calculate the likelihood of only the dominant optimization alignment (which may only be a very small fraction of the total likelihood).
   - Perform the initial build in the order in which taxa occur in the first data file.
   - Estimate θ and α parameters to model among-site rate variation from all pairwise alignments of input sequences and fix them for the duration of the search.
   - Estimate transition probabilities from all pairwise comparisons of input sequences and fix them for the entire search.
   - Estimate base and indel frequencies from all pairwise comparisons of input sequences and fix them for the entire search.
   - Set the likelihood rounding multiplier to 100 (=0.01 log likelihood units).
   - Set the threshold for two likelihood scores to be considered equal at 1/`likelihoodroundingmultiplier`—that is, 1/100, or 0.01 log likelihood units.
   - Estimate the transition matrix by recounting all the types of event for each node-to-node comparison throughout the search.
   - Perform a maximum of 100 branch length iterations.
   - Set the size of iteration steps during branch length optimization to 5 units.
   - Use `s6g` submodel, equivalent to GTR + indels with all indels treated equally.
2. Save the file as `ml1.bat` in the tutorial folder.
3. Execute `ml1.bat` and examine the results.
4. In your text editor open `ml1.err`.

   Note that after reporting on the initial maximum likelihood options, the base frequencies calculated from pairwise comparisons are reported:

   ```
   "Estimating likelihood parameters (using 9 pairwise
   comparisons)...
   ```

```
Estimated base frequencies : 0.262510 0.277968
0.197670 0.259127 0.002725
On to theta and gamma
Done"
```

It then reports on the build and refinement stages of the analysis and summarizes the run data (for example, this run took 179 seconds on a 1.7-GHz Pentium 4).

5. In your text editor open `ml1.out`.

   This file reports the maximum likelihood tree and its log likelihood score.

```
(Hagfish ((((Tree_Frog Newt) (Horn_shark Sawshark))
(Cow Human)) (Salmon Goldfish)))[5651.57]
;
Binary representations of best trees:
(Hagfish ((Goldfish Salmon) ((Human Cow)
((Tree_Frog Newt) (Sawshark
Horn_shark)))))[5651.57]
;
```

   Note that there is a large difference between the likelihood reported with the tree (in square brackets) and the likelihoods reported during the search. For internal calculations tree costs are determined as the likelihood of obtaining each descendant from each ancestor, multiplying each time, as this is the portion of the likelihood score that varies among trees. However, to calculate the correct likelihood, the likelihood of having the root ancestor in the first place must also be included; because this is constant over all topologies it is simply added in the final step. The correct value is the one reported with the tree topology (5,651.57 in this case).

6. Save a copy of `ml1.bat` as `ml2.bat` in the tutorial folder.
7. In your text editor open `ml2.bat` and modify the script to read as follows:

```
poy -likelihood mol1.txt mol2.txt -totallikelihood
  -diagnose > ml1.out 2> ml2.err
```

   This involves making the following change to the previous script:

   - Add the command `totallikelihood`, which determines the likelihood score of an optimization alignment by summing all major alignments. This is a better approximation of the actual likelihood score, but it is much slower (this run took 643 seconds on the same computer).
   - Add the command `diagnose`, which causes diagnostics to be printed to the outfile.

8. Save the changes you made to `ml2.bat`.

9. Execute `ml2.bat`.
10. In your text editor open `ml2.out`.

    The following is reported:

```
(Hagfish ((((Tree_Frog Newt) (Horn_shark Sawshark))
(Cow Human)) (Salmon Goldfish)))[5632.85]
;
Binary representations of best trees:
(Hagfish (((Cow Human) ((Tree_Frog Newt) (Sawshark
Horn_shark))) (Goldfish Salmon)))[5632.85]
;
```

    Note the difference in the likelihood score obtained from this and the previous script.

    Immediately beneath the trees is the following output, which reports the relative maximum and minimum branch lengths in likelihood units:

```
Hypothetical Ancestral Nodes:
Node          Branch Length   Description
------------------------------------------------------
Hagfish      [   0 -   0 ]
Tree_Frog    [10765 -13222 ]
Newt         [10454 -12915 ]
Cow          [10087 -13374 ]
Human        [9112 -12406 ]
Horn_shark   [12694 -17153 ]
Salmon       [6333 -11933 ]
Goldfish     [7571 -13164 ]
Sawshark     [8150 -10187 ]
HTU0         [ Root Node ] = (Hagfish (((Cow Human)
((Tree_Frog Newt) (Sawshark Horn_shark))) (Goldfish Salmon)))
HTU1         [2039 -5918 ] = (Cow Human)
HTU2         [2174 -9867 ] = ((Cow Human) ((Tree_Frog Newt)
(Sawshark Horn_shark)))
HTU3         [3345 -6472 ] = (Tree_Frog Newt)
HTU4         [1355 -6231 ] = ((Tree_Frog Newt) (Sawshark
Horn_shark))
HTU5         [3191 -6204 ] = (Sawshark Horn_shark)
HTU6        [16278 -24068 ] = (((Cow Human) ((Tree_Frog Newt)
(Sawshark Horn_shark))) (Goldfish Salmon))
HTU7         [3215 -7236 ] = (Goldfish Salmon)
```

11. Save a copy of `ml1.bat` as `ml3.bat`.
12. In your text editor modify the script to read as follows:

```
poy -likelihood mol1.txt mol2.txt -noestimatep
  -noestimateq -noestimateparamsfirst -
  likelihoodesttranseachtime > ml3.out 2> ml3.err
```

This involves making the following changes to the existing script:

- Add `noestimateparamsfirst` to prevent fixing parameters for the entire search from initial pairwise estimates of input sequences, but to reestimate parameters throughout the search, including θ and α parameters to model among-site rate variation.
- Add `noestimatep` to prevent fixing base and indel frequency estimates for the entire search from initial pairwise alignments of input sequences, but to estimate tansition probabilities for and from every pair of sequences (including HTU sequences) as they are encountered.
- Add `noestimateq` to prevent to fixing transition probabilities for the entire search from initial pairwise alignments of input sequences, but to estimate base and indel frequencies for and from every pair of sequences (including HTU sequences) as they are encountered.
- Add `likelihoodesttranseachtime` to recount all the types of transition for each pairwise comparison for reestimating the transition matrix (q).

13. Save the changes to `ml3.bat`.
14. Execute `ml3.bat` and examine the results in `ml3.out` and `ml3.err`.
15. Save a copy of `ml1.bat` as `ml4.bat`.
16. In your text editor open `ml4.bat` and convert it into the following script:

    ```
    poy -likelihood mol1.txt mol2.txt -gammaclasses 4 -
      gammaalpha 2 -theta 0.10 -invariantsitesadjust >
      ml3.out 2> ml3.err
    ```

    This involves making the following modifications to the previous script:

    - Add the command `gammaclasses 4` to specify four rate classes for Γ (gamma) rate heterogeneity parameter.
    - Add the command `gammaalpha 2` to specify α (alpha), the shape parameter of the Γ distribution, at 2.
    - Add `theta 10` to specify the proportion of invariant sites (θ) at 10%.
    - Add `invariantsitesadjust` to adjust the likelihoods for the invariant sites.

17. Save the changes you made to `ml4.bat`.

18. Execute ml4.bat and examine the results in `ml4.out` and `ml4.err`.

Tutorial 10: Multiple Scripts

This tutorial demonstrates how to include multiple scripts in a single batch file, so that POY runs a sequence of analyses consecutively. Running multiple scripts consecutively enables you to assess the effect of different optimality criteria, trees search strategies, input files and input file structures, and indel and nucleotide transformation costs.

For this tutorial, we build on the script you created in Tutorial 4. The multiscript batch file you create will apply three alternative nucleotide transformation costs.

1. In the Tutorial folder, save a copy of `totalev1.bat` (created in Tutorial 5) as `sensitivity1.bat`.
2. In your text editor, edit `sensitivity1.bat` to create the following script:

```
poy -weight 1 morh.txt mola1.txt mola2.txt -
  molecularmatrix g1ts1tv1.txt -nooneasis -
  approxbuild -replicates 5 -nonrandomizeoutgroup -
  slop 2 -checkslop 5 -noleading -outgroup hagfish -
  exact > 111.out 2> 111.err
```

   This requires that you make the following changes to the original script:
   - Add the command `molecularmatrix g1ts1tv1.txt`.
   - Change the output and program progress file names to `111.out` and `111.err`, respectively.
   - Delete the commands `gap 1`, `iterativepass`, `phastawincladfile totalev1.ss`.
3. Copy and paste the entire script twice, separating the scripts with two line breaks.
4. In the second script,
   - change the command `weight 1` to `weight 2`
   - change the output and program progress file names to `211.out` and `211.err`, respectively
   - change the file name `g1ts1tv1.txt` to `g2ts1tv1.txt`.
5. In the third script,
   - change the command `weight 1` to `weight 4`
   - change the output and program progress file names to `421.out` and `421.err`, respectively
   - change the file name `g1ts1tv1.txt` to `g4ts2tv1.txt`.

When complete, the batch file should contain three scripts, each separated by two line breaks as in Figure 14.13.

```
poy -weight 1 morph.txt mol1a.txt mol2a.txt -
molecularmatrix g1tv1ts1.txt -nooneasis -
approxbuild -replicates 2 -checkslop 2 -exact -
phastwincladfile 111.ss > 111.out 2> 111.err
¶
poy -weight 2 morph.txt mol1a.txt mol2a.txt -
molecularmatrix g2tv1ts1.txt -nooneasis -
approxbuild -replicates 2 -checkslop 2 -exact -
phastwincladfile 211.ss > 211.out 2> 211.err ¶
¶
poy -weight 4 morph.txt mol1a.txt mol2a.txt -
molecularmatrix g4tv2ts1.txt -nooneasis -
approxbuild -replicates 2 -checkslop 2 -exact -
phastwincladfile 421.ss > 421.out 2> 421.err
```

Figure 14.13: Multiple script file `sensitivity1.bat`.

6. Save `sensitivity1.bat` as plain text format in the Tutorial folder.
7. Run `sensitivity1.bat`.

   This batch file runs the three scripts in succession. The scripts differ in the weights they apply to different nucleotide transformations and to phenotypic data.

   - The first script weighs all nucleotides and phenotypic characters equally. The step matrix file `g1ts1tv1.txt` assigns a value of one to all transformations (see Figure 14.8 on page 176). The command `weight 1` sets the weight of phenotypic character transformations to one.
   - In the second script, the step matrix file `g2ts1tv1.txt` assigns a value of two to indels and one to transversions and transitions, and `weight 2` sets the weight of phenotypic character transformations to two (equal to indels).
   - In the third script, the step matrix file `g4ts2tv1.txt` sets indels to four, transitions to two, and transversions to one. With `weight 4` it sets phenotypic characters to four (equal to indels).

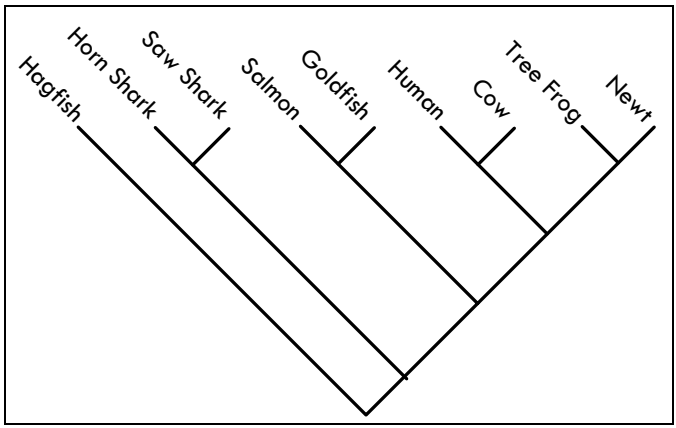

Figure 14.14: Cladogram of `111.out`, `211.out`, and `421.out`.

Tutorial 11: Parallel Environments

This tutorial demonstrates how to use POY in a parallel environment (and assuming that your cluster includes 30 processors, although the tutorial will work with more or less). In particular, it demonstrates the use of controllers to allocate calculations to subclusters. There are two different ways to perform a parallel execution of POY depending on which message passing library you are using: PVM or MPI. The following tutorial applies for PVM users. Small modifications are required when using MPI, and these can be found at the end of the section. For a more complete discussion of how to use POY in a parallel environment, refer to "Parallel Environments" on page 114. This tutorial does not illustrate how to begin a parallel session; you will need to consult your System Administrator for that information (see also Tutorial 12).

By default, the parallel command causes POY to treat the entire cluster (that is, all processors) as a single virtual machine. That is, it will divide each replicate across the entire cluster, assigning very little work to each processor and requiring a great deal of interprocessor communication. In this strategy, replicates are done consecutively, not simultaneously. This is an extremely fine-grained strategy, and it tends to be a highly inefficient approach to parallel computation.

This tutorial demonstrates how to modify the default parallel behavior of POY to alter the granularity and fine-tune your analysis to decrease computation time.

1. In your text editor, type the following script:

```
poy mola1.txt mola2.txt -weight 1 morph.txt -
  nooneasis -slop 5 -checkslop 10 -exact -parallel -
```

```
multirandom -replicates 30 > para1.out 2>
para1.err
```

The command `multirandom` modifies the parallel behavior of POY by causing it to run each replicate simultaneously on a separate slave processor. This is a coarse-grained approach, because it assigns a large amount of work (an entire replicate, including building and refinement) to each processor and requires only minimal communication among processors.

2. Copy and paste this script, separating it from the first with two line breaks.
3. In the second script, make the following changes:
   - Add the command `controllers 3`. This command divides the available nodes into three subclusters. In conjunction with the command `-multirandom`, replicates are distributed among these three subclusters rather than being assigned to individual nodes. This approach is intermediate in granularity, and it can allow the user to fine-tune analyses to particular parallel environments, algorithms, and data sets.
   - Change the output and program progress file names to `para2.out` and `para2.err`, respectively.

   When complete, the batch file should contain two scripts, separated by two line breaks as in Figure 14.15.

```
poy mola1.txt mola2.txt -weight 1 morph.txt
-nooneasis -slop 5 -checkslop 10 -exact -parallel
-multirandom -replicates 30 > para1.out 2>
para1.err ¶
¶
poy mola1.txt mola2.txt -weight 1 morph.txt
-nooneasis -slop 5 -checkslop 10 -exact -parallel
-multirandom -replicates 30
-controllers 3 > para2.out 2> para2.err
```

Figure 14.15: Multiple script file `sensitivity1.bat`.

4. Save `parallel.bat` as plain text format in the Tutorial folder.
5. Run `parallel.bat`.

   This batch file runs the two scripts in succession. Compare the runtime with and without using `controllers`.

Modifications for MPI

1. The processes are started using the MPI program provided by your distribution. The most common is *mpirun*; check you MPI distribution documentation (version 2 of the standard recommends

mpirun, but your distribution may be slightly different). The text below assumes mpirun.

2. The MPI 1 standard does allow a program to control the number of processes started, as PVM does. Therefore the total number of processes is defined by mpirun, and the master process will have rank 1.
3. The total number of processes starts should be less than or equal to the number of controllers plus the number of slaves plus 1. If the total number of processes is less than this sum, the rest of the processes will be used for fault tolerance features (process replacement in case of failure).
4. There is no control on the processes; the number that begin the calculations will be the same number that will finish (excepting failures).
5. Note that neither the PVM and MPI versions provide a high level of fault tolerance in avoiding losing computation if a process fails.
6. -jobspernode, -maxprocessors, -onan, -onannum, and -randomizeslaves are PVM-dependent functions. These are not available in the MPI version. For controlling these variables, check your MPI documentation.
7. In the MPI version, -solospawn defines the number of slaves to be used, *not the number of slave jobs spawned on a stand-alone multiprocessor machine.*

Tutorial 12: Scheduling Software

Most clusters operate through front-end scheduling software (also known as batch loaders or queues). This tutorial demonstrates how to run POY in PVM through the Sun Grid Engine (SGE) scheduling software. Note that different scheduling software require slightly different scripts, and systems administrators have different rules for cluster use, so the scripts in this tutorial might need to be modified for your system.

1. In your text editor, type the following script:

```
#!/bin/csh -f

#$ -N tutorial
#$ -pe pvm 30
#$ -V
#$ -S /bin/csh
#$ -cwd
#$ -M grant@amnh.org
#$ -m bea

set slots=`expr $NSLOTS - 1`
```

```
poy -weight 1 morph.txt mola1.txt mola2.txt -
  parallel -numslaveprocesses $NSLOTS -onan -
  nooneasis -slop 5 -checkslop 10 -exact -parallel -
  multirandom -replicates 30 -controllers 3 -
  phastwincladfile tutorial.ss
```

2. Save the file as `tutorial.sh` as plain text format in the Tutorial folder.

   This script contains the commands for SGE (each preceded by `S#`) followed by the commands for POY (the same as in Tutorial 8). Important parameters to be aware of are the following:

   - `-N`: designates the name of the job to be submitted to SGE (called `tutorial` here)
   - `-pe pvm`: designates the number of hosts to be requested for the run (`30` here) and generates a hosts file to start PVM
   - `-M`: causes an email to be sent to the designated address
   - `-m`: stipulates when an email is to be sent; `b` causes an email to be sent when the run begins, `e` when it ends, and `a` if it aborts.

   POY scripts differ very little from other parallel scripts. All POY commands will work through a scheduler. The main difference is that no.out or.err progress report files are specified. This is because the scheduling software generates these files automatically. These files are given an extension followed by the job ID number assigned automatically by the scheduler. For example, the following files were generated in our analysis:

   ```
   tutorial.po127
   tutorial.pe127
   tutorial.o127
   tutorial.e127
   ```

   where `tutorial` is the name of the job, `127` is the job ID number, `po` is the SGE out file, `pe` is the SGE standard error file, `o` is the POY out file, and `e` is the POY standard error file. Otherwise, inputting (for example, tree or consensus files) and generating (for example, tree graphics and Hennig86/NONA matrices) files is the same with or without a scheduler.

   Another difference is the inclusion of the command `onan`, which causes POY to spawn jobs on the master processor. In a normal parallel environment, the master runs all the jobs for all users and also houses the input and output files from analyses. As such, traffic is usually quite heavy on the master, so it is unwise to further burden this processor with tree searching. However, scheduling software designates a different master for each run. Since most of the time the master is waiting to receive information from slaves, it is not heavily used and jobs can safely be spawned to it.

3. To submit the job to SGE, at the prompt type
   `qsub tutorial.sh`
   An email will be generated to notify you when your job has begun.
4. To check the status of your job in the queue, at the prompt type
   `qstat`
5. To delete a job from the queue, at the prompt type
   `qdel jobIDnumber`
   For example, `qdel 127` would remove job number 127.

15 Command References

This chapter describes the commands used to perform analysis using POY. The first section lists commands by function. Each reference includes a brief description of the command's usage. The second section lists commands alphabetically. Each reference includes syntax rules, specification of parameters, and an example of and comments on the command's usage.[1]

Commands by Function

This section lists POY commands by function. Detailed command references begin on page 216.

1. This chapter is based on the command line reference in POY (version 3) designed and implemented by Jan DeLaet.

Input Data

The following commands affect POY input data input. Commands that affect input data for likelihood analysis are listed separately, beginning on page 207.

Command	Page	Description
agree	217	Performs optimization subject to the constraint that specified clades are recovered.
change	227	Sets the cost for nucleotide change.
constrain	233	Reads cladogram search constraints from one or more specified files.
defaultweight	237	Sets the weight of all character data that follow.
disagree	238	Performs a cladogram search such that clades specified by a constraint file are not recovered.
extensiongap	251	Sets the cost of gaps that follow the first in a series of gaps.
fixedstates	253	Performs fixed state optimization on a specified file.
gap	257	Sets the cost of gaps.
goloboff	258	Performs implied weights optimization.
kfactor	273	Sets the k value for implied character weighting.
leading	273	Counts leading and trailing gaps in tree cost.
molecularmatrix	283	Reads nucleic acid transformation costs from a specified file.
multiplier	284	Sets the multiplier for implied weighting.
prealigned	302	Reads sequence data as prealigned from a specified file.
terminalsfile	320	Sets terminal taxon names.
topofile	322	Reads topologies from a specified file.
topology	323	Reads an input cladogram from the command line.
trailinggap	326	Sets the cost of leading and trailing gaps in a sequence.
weight	329	Sets the weight of the data in the preceding input data file.

Output

The following commands control POY output.

Command	Page	Description
bremer	220	Calculates Bremer support values using TBR and writes results to standard output.
catchslaveoutput	227	When using `parallel` (page 293), writes all standard output of slave tasks to the standard output of the master task.
characterweights	227	Writes search statistics for each character to standard output.
diagnose	238	Writes branch costs and apomorphy lists to standard output.
discrepancies	239	Writes the cost difference between trees found using shortcuts and trees found on a complete down pass.
hypancname	261	Writes the names of hypothetical ancestral nucleotide sequences to standard output.
impliedalignment	263	Writes a topology-specific multiple alignment based on the synapomorphy scheme to standard output.
indices	263	Writes fit statistics (consistency index and retention index) for a topology to standard output.
intermediate	263	Writes intermediate search results to standard output.
phastwincladfile	293	Writes implied alignments to a specified file.
plotencoding	293	Sets the characters used to plot trees.
plotechocommandline	294	Writes the command line as the first line of the tree output file.
plotfile	294	Sets the name of the tree output file.
plotfrequencies	295	Sets how clade frequencies are tabulated in the least cost trees.
plotmajority	296	Sets how to plot majority rule consensus trees.
plotoutgroup	296	Sets the outgroup used for tree plotting.
plotstrict	297	Sets how to plot strict consensus trees.
plottrees	297	Sets how to plot optimal trees.
plotwidth	298	Sets the number of columns used for plotting trees.
poybintreefile	300	Writes the optimal trees in POY topology format to a specified file.
poystrictconsensuscharfile	300	Writes the strict consensus tree in Hennig86/Nona format to a specified file.

Command	Page	Description
poystrictconsensus-treefile	301	Writes the strict consensus tree in POY topology format to a specified file.
poytreefile	301	Writes the optimal cladograms in POY topology format to a specified file.
printccode	302	Writes a ccode-like summary of the character settings in each Hennig86/Nona formatted input data file to standard output.
printhypanc	303	Writes hypothetical ancestral sequences for optimal cladograms to a specified file.
printlotshypanc	303	Writes hypothetical ancestral sequences for intermediate and optimal trees to a specified file.
printqmat	304	Writes likelihood parameters to standard output.
printtree	304	Writes a graphic of the optimal trees and their strict consensus trees to a standard output file.
quote	306	Writes a specified string to standard output.
repintermediate	311	Writes the results of individual random replicates to standard output.
showiterative	314	Writes iterative pass progress information to standard output.
spewbinary	316	Writes binary trees (that is, trees without polytomies) used internally by POY to standard output. Default value.
stats	318	Writes cladogram search statistics to standard output.
time	321	Displays execution time in seconds.
verbose	328	Writes cladogram search progress information to the standard error file.

Optimization

This section describes commands that control how POY performs optimization.

Constraints

The following commands affect how optimization is constrained.

Command	Page	Description
agree	217	Performs optimization subject to the constraint that specified clades are recovered.
constrain	233	Reads cladogram search constraints from one or more specified files.

Command	Page	Description
disagree	238	Performs a cladogram search such that clades specified by a constraint file are not recovered.
dropconstraints	248	Performs cladogram building using constraints, then performs cladogram refinement without constraints.

Additive characters

The following commands describe how POY performs additive character optimization.

Command	Page	Description
recode	311	Accelerates additive and nonadditive characters optimization. Default value.

Nonadditive characters

The following commands describe how POY performs nonadditive character optimization.

Command	Page	Description
recode	311	Accelerates additive and nonadditive characters optimization. Default value.

Implied weights

The following commands describe how POY performs implied weights optimization.

Command	Page	Description
goloboff	258	Performs implied weights optimization.
kfactor	273	Sets the k value for implied character weighting.
multiplier	284	Sets the multiplier for implied weighting.

Sankoff characters

The following input format allows Sankoff character optimization.

Command	Page	Description
dpread	151	Input file for Sankoff characters

Direct optimization

Direct optimization is the default optimization algorithm.

Fixed state and search-based optimization

The following commands describe how POY performs fixed state optimization.

Command	Page	Description
compressstates	233	Tracks states during the build process of search-based optimization, then removes unused states during the cladogram search.
fixedstates	253	Performs fixed state optimization on a specified file.
newstates	286	Reads the state set for search-based optimization from a specified file.

Iterative pass

The following commands describe how POY performs iterative pass optimization.

Command	Page	Description
iterativeinitsingle	264	Sets initial HTU states by resolving ambiguities in preliminary phase (that is, the down pass) from direct optimization.
iterativekeepbetter	264	Retains the hypothetical ancestral states and cladogram cost that is the lesser of standard up/down pass and iterative pass.
iterativelowmem	265	Minimizes memory allocation during iterative pass optimization.
iterativepass	265	Performs iterative pass optimization.
iterativepassfinal	266	Uses iterative pass to reestimate final states.
iterativerandom	266	Resolves ambiguities in hypothetical ancestral states using random preference.
maxiterations	281	Sets the maximum number of iterative passes to perform.
showiterative	314	Writes iterative pass progress information to standard output.

Chromosomal characters

The following commands describe how POY performs chromosomal character optimization.

Command	Page	Description
allchroms	217	Optimizes HTU chromosomes by examining all combinations of loci in the three adjacent HTU/OTU chromosomes.
breakpoint	220	Sets the breakpoint cost.
buildchrom	221	Optimizes chromosomal characters by building HTU chromosomes one locus at a time in the manner of cladogram building. Default value.
chromosome	229	Causes POY to use the chromosome character type.
chromfilter	230	Ignores all chromosome fragments whose length is greater than 0 and less than a specified length.
circular	230	Treats chromosomal characters as circular, as opposed to linear. Default value.
fixedstates	253	Performs fixed state optimization on a specified file.
impliedalignment	263	Writes a topology-specific multiple alignment based on the synapomorphy scheme to standard output.
linear	278	Treats chromosomal characters as linear, as opposed to circular.
locusgap	279	Sets the constant fraction of the cost of locus origin-loss events.
locussizegap	279	Sets the variable fraction of the cost of locus origin-loss events.
locusswap	280	Refines chromosome HTU construction.
n2reorder	285	Selects reorderings of loci on a chromosome using a method akin to cladogram building.
onechroms	289	Optimizes HTU chromosomes by examining a single resolution of loci that have been ambiguously assigned to the candidate HTU. Default value.
rearrange	310	Enables locus rearrangement to occur when optimizing chromosomal characters.
reversible	312	Enables the loci to be treated as unsigned—that is, either the given locus or its reverse complement can be used in optimization.
showchromsearch	313	Display of chromosomal HTU optimization progress information.
somechroms	315	Optimizes HTU chromosomes by examining all combinations of loci that have been ambiguously assigned to the candidate HTU.

Static approximation

The following commands describe how POY performs a static approximation cladogram search.

Command	Page	Description
staticapprox	317	Performs static approximation for sequence optimization during cladogram refinements.
staticapproxbuild	317	Performs static approximation for sequence optimization during cladogram building.

Likelihood

The following commands affect input data when performing likelihood analysis.

Command	Page	Description
basefreq	219	Reads nucleotide frequencies from a file.
estimatep	250	Uses nucleic acid and gap frequencies estimated from input data and pairwise alignments, which are then fixed for the duration of the search.
estimateparamsfirst	250	Uses likelihood parameters estimated from input data and pairwise alignments, which are then fixed for the duration of the search.
estimateq	251	Uses probabilities for transitions estimated from pairwise alignments, which are then fixed for the duration of the search.
freqmodel	254	Sets how nucleic acid and gap frequencies are estimated.
gammaalpha	256	Sets the alpha parameter for the gamma distribution.
gammaclasses	256	Sets the number of rate classes for a discrete gamma distribution.
invariantsitesadjust	264	Adjusts likelihoods for theta fraction invariant sites.
likelihood	274	Performs likelihood analysis.
likelihoodconvergencevalue	274	Sets the maximum likelihood difference for which two likelihood scores are considered equal.
likelihoodestimationsize	275	Sets the number of pairwise sequence comparisons used to estimate likelihood parameters.
likelihoodesttranseachtime	275	Estimates the transition matrix for each comparison of two sequences.
likelihoodextensiongap	275	Sets the likelihood of gaps that follow the first in a series of gaps.
likelihoodmaxnumiterations	276	Sets the maximum number of branch cost iterations performed.

Command	Page	Description
likelihoodrounding-multiplier	276	Sets the rounding value used in likelihood calculations.
likelihoodstep	277	Sets the size of iteration steps during branch cost optimization.
likelihoodtrailinggap	277	Sets the likelihood of leading and trailing gaps.
qmatrix	305	Reads a transition cost ratio matrix for likelihood analysis from a specified file.
submodel	319	Enforces symmetries in transition probabilities during likelihood analysis.
theta	321	Sets the theta value for likelihood analysis.
totallikelihood	326	Calculates the likelihood of an optimization alignment by summing only the major optimization alignments.
trullytotallikelihood	328	Calculates the likelihood of an optimization by summing all optimizations.

Cladogram Search

This section describes commands that control how POY performs cladogram searches.

Constrained searches

The following commands affect how cladogram searching is constrained.

Command	Page	Description
agree	217	Performs optimization subject to the constraint that specified clades are recovered.
constrain	233	Reads cladogram search constraints from one or more specified files.
disagree	238	Performs a cladogram search such that clades specified by a constraint file are not recovered.
dropconstraints	248	Performs cladogram building using constraints, then performs cladogram refinement without constraints.

SPR and TBR

The following commands describe how POY performs subtree pruning and regrafting (SPR) and tree bisection and reconnection (TBR) cladogram searches.

Command	Page	Description
approxquickspr	218	Calculates tree lengths for SPR using an approximation shortcut.
approxquicktbr	219	Calculates tree lengths for TBR using an approximation shortcut.
cutswap	236	Performs a shortcut during branch swapping based on approximate tree costs.
quick	305	Performs branch swapping only on minimum cost trees.
spr	316	Performs an SPR cladogram search.
sprmaxtrees	317	Sets the maximum number of trees held in buffers during an SPR cladogram search.
tbr	319	Performs a TBR cladogram search.
tbrmaxtrees	320	Sets the maximum number of trees held in buffers during a TBR cladogram search.

Ratcheting

The following commands describe how POY performs a ratcheted cladogram search.

Command	Page	Description
checkfrequency	228	Sets the frequency of message checking when performing parallel ratcheting.
multiratchet	285	Spawns individual ratchet jobs on slave nodes.
ratchetoverpercent	307	Spawns a specified number of extra ratchet jobs to accommodate unequal execution times during parallel execution.
ratchetpercent	308	Sets the percentage of characters reweighted during a ratchet search.
ratchetseverity	308	Sets the weight multiplier for reweighting characters during a ratchet search.
ratchetslop	308	Sets the percent limit of suboptimal trees evaluated during a ratchet search.
ratchetspr	309	Sets the number of ratchet iterations using SPR.
ratchettbr	309	Sets the number of ratchet iterations using TBR.
ratchettrees	310	Sets the number of trees saved during ratchet iterations.

Tree fusing

The following commands describe how POY performs tree fusing.

Command	Page	Description
fuseafterreplicates	254	Performs tree fusing on the results of random replicates as they are calculated. This is the default value when tree fusing (`treefuse` on page 327) is on.
fuselimit	254	Sets the maximum number of tree fusing pairs held in memory.
fusemaxtrees	255	Sets the maximum number of trees held in memory during tree fusing.
fusemingroup	255	Sets the minimum number of taxa in a subtree that can be exchanged.
fusingrounds	256	Performs tree fusing a specified number of times.
treefuse	327	Performs cladogram search using tree fusing.
treefusespr	327	Performs a SPR cladogram search on new cladograms found during a tree fusing search.
treefusetbr	328	Performs a TBR cladogram search on new cladograms found during a tree fusing search.

Tree drifting

The following commands describe how POY performs tree drifting.

Command	Page	Description
approxdrift	218	Calculates tree lengths for tree drifting using an approximation shortcut.
driftequallaccept	247	Sets the percentage of instances to accept equally parsimonious trees during tree drifting.
driftlengthbase	247	Sets a parameter used to specify the probability of accepting a suboptimal tree.
driftspr	248	Performs tree drifting using SPR.
drifttbr	248	Performs tree drifting using TBR.
multidrift	284	Spawns individual tree drifting jobs to slave nodes.
numdriftchanges	287	Sets the number of topological changes permitted per tree drifting round.
numdriftspr	287	Sets the number of SPR tree drift rounds.
numdrifttbr	287	Sets the number of TBR tree drift rounds.

Evaluation

This section describes commands that control how POY evaluates analysis results.

Reconstruction of hypothetical ancestral states

The following commands control how POY reports hypothetical ancestral states.

Command	Page	Description
hypancfile	260	Sets the output file name for hypothetical ancestral nucleotide sequences.
hypancname	261	Writes the names of hypothetical ancestral nucleotide sequences to standard output.
printhypanc	303	Writes hypothetical ancestral sequences for optimal cladograms to a specified file.
printlotshypanc	303	Writes hypothetical ancestral sequences for intermediate and optimal trees to a specified file.

Consensus techniques

The following commands control how POY reports consensus trees.

Command	Page	Description
jackpseudoconsensustrees	270	Writes the strict consensus trees for the individual pseudo-replicates to standard output.
jacktree	271	Writes the majority rule consensus tree from the strict consensus trees of all jackknife pseudo-replicates to standard output.
jackwincladefile	272	Writes the strict consensus trees from the individual pseudo-replicates to a specified file in Winclada readable format.
plotmajority	296	Sets how to plot majority rule consensus trees.
plotstrict	297	Sets how to plot strict consensus trees.
plottrees	297	Sets how to plot optimal trees.
poystrictconsensuscharfile	300	Writes the strict consensus tree in Hennig86/Nona format to a specified file.
poystrictconsensustreefile	301	Writes the strict consensus tree in POY topology format to a specified file.
printtree	304	Writes a graphic of the optimal trees and their strict consensus trees to a standard output file.

Implied alignment

The following commands control how POY reports alignment.

Command	Page	Description
impliedalignment	263	Writes a topology-specific multiple alignment based on the synapomorphy scheme to standard output.
phastwincladfile	293	Writes implied alignments to a specified file.

Support

The following commands control how POY reports support metrics.

Command	Page	Description
		Bremer
bremer	220	Calculates Bremer support values using TBR and writes results to standard output.
bremerspr	221	Calculates Bremer support values using SPR and writes to standard output.
		Jackknife
jackboot	266	Performs Farris et al. (1996) jackknifing.
jackfrequencies	269	Writes the frequency of clades in a jackknife analysis to standard output.
jackoutgroup	270	Sets the outgroup for jackknifing.
jackpseudoconsensustrees	270	Writes the strict consensus trees for the individual pseudo-replicates to standard output.
jackpseudotrees	271	Writes the least cost trees for the individual pseudo-replicates to standard output.
jackstart	271	Builds a set of initial trees using jackknifing, then performs the cladogram search as specified.
jacktree	271	Writes the majority rule consensus tree from the strict consensus trees of all jackknife pseudo-replicates to standard output.
jackwincladefile	272	Writes the strict consensus trees from the individual pseudo-replicates to a specified file in Winclada readable format.

Fit statistics

The following commands control how POY reports fit statistics.

Command	Page	Description
stats	318	Writes cladogram search statistics to standard output.

Parallel Processing

This section describes commands that control how POY executes in a parallel processing environment.

Dynamic process migration

The following commands control how POY dynamic process migration functions in a computer cluster environment.

Command	Page	Description
dpm	241	Performs load balancing by using dynamic process migration.
dpmacceptratio	242	Sets the threshold parameter for dynamic process migration.
dpmautoadjustperiod	242	Adjusts the frequency with which performance is evaluated for dynamic process migration.
dpmjobspernode	243	Sets the number of reserve jobs spawned for dynamic process migration.
dpmmaxperiod	243	Sets the maximum time period used in adjusting the frequency of performance evaluation by `dpmautoadjustperiod`.
dpmmaxprocessors	244	For parallel processing using dynamic process migration, sets the number of reserve jobs to spawn.
dpmminperiod	244	Sets the minimum time period used in adjusting the frequency of performance evaluation by `dpmautoadjustperiod`.
dpmperiod	245	For parallel processing using dynamic process migration, sets the initial time period used by `dpmautoadjustperiod`.
dpmproblemsize	246	Sets the size of the calculation used to evaluate performance for dynamic process migration.

Parallel processing and load balancing

The following commands control how POY performs parallel processing and load balancing.

Command	Page	Description
catchslaveoutput	227	When using `parallel` (page 293), writes all standard output of slave tasks to the standard output of the master task.
controllers	234	Sets the number of subclusters used in a parallel cladogram search.

Command	Page	Description
jobspernode	272	Sets the number of processes spawned on each node.
maxprocessors	281	Sets the number of spawned jobs.
minstop	282	Sets the minimum number of random replicates that must be completed before processing ceases.
multidrift	284	Spawns individual tree drifting jobs to slave nodes.
multirandom	285	Spawns individual random replicates to slave nodes.
multiratchet	285	Spawns individual ratchet jobs on slave nodes.
onan	288	Spawns jobs on the master as well as on slave nodes.
onannum	288	Sets the number of jobs spawned to the master node.
parallel	293	Executes POY as a parallel process using PVM.
randomizeslaves	307	Sets PVM task identification numbers (TIDs) randomly.
ratchetoverpercent	307	Spawns a specified number of extra ratchet jobs to accommodate unequal execution times during parallel execution.
solospawn	315	Sets the number of slave jobs spawned on a stand-alone multiprocessor machine.
stopat	318	Sets the minimum number of minimum cost replicates that must be found before the search stops.

System Commands

This section describes commands that perform various POY maintenance functions.

Command	Page	Description
clist	231	Displays a list of all commands cross-referenced to commands that contain a specified string in their usage information.
cllist	231	Displays a list with descriptions of all commands cross-referenced to commands that contain a specified string in their usage information.
dlist	239	Displays a list of commands that contain a specified string in their usage information.
dllist	240	Displays a list with descriptions of commands that contain a specified string in their usage information.
help	259	Displays all commands with their descriptions.
list	278	Displays a list of all commands that contain a specified string in their usage information.
llist	278	Displays a list with descriptions of all commands that contain a specified string in their usage information.

Command	Page	Description
nopoyini	286	Causes POY to not recognize the file `poy.ini` in the current directory as a command file.
time	321	Displays execution time in seconds.
verbose	328	Writes cladogram search progress information to the standard error file.

Execution Time

This section describes commands that decrease POY execution time.

Command	Page	Description
approxbuild	218	Builds the initial tree using an approximation shortcut.
approxquickspr	218	Calculates tree lengths for SPR using an approximation shortcut.
approxquicktbr	219	Calculates tree lengths for TBR using an approximation shortcut.
buildmaxtrees	222	Sets the maximum number of trees held during initial cladogram builds.
buildslop	222	Sets the percent limit of suboptimal trees evaluated during the cladogram building process.
checkslop	228	Sets the percent limit of suboptimal trees evaluated during a final TBR refinement.
maxtrees	282	Sets the maximum number of trees held in buffers during processing. See `buildmaxtrees` (page 222) and `sprmaxtrees` (page 317).
quick	305	Performs branch swapping only on minimum cost trees.
slop	314	Sets the percent limit of suboptimal trees evaluated during a search.
staticapprox	317	Performs static approximation for sequence optimization during cladogram refinements.
staticapproxbuild	317	Performs static approximation for sequence optimization during cladogram building.

Exhaustive Searches

This section describes commands that increase how exhaustively POY performs cladogram searches.

Command	Page	Description
buildmaxtrees	222	Sets the maximum number of trees held during initial cladogram builds.
buildslop	222	Sets the percent limit of suboptimal trees evaluated during the cladogram building process.
buildsperreplicate	223	Performs a specified number of addition sequences in the build phase of a single replicate.
checkslop	228	Sets the percent limit of suboptimal trees evaluated during a final TBR refinement.
driftspr	248	Performs tree drifting using SPR.
drifttbr	248	Performs tree drifting using TBR.
exact	251	Calculates tree costs more accurately for direct optimization on the down pass by creating an implied alignment and performing a complete Sankoff optimization.
iterativepass	265	Performs iterative pass optimization.
maxtrees	282	Sets the maximum number of trees held in buffers during processing. See `buildmaxtrees` (page 222) and `sprmaxtrees` (page 317).
ratchetspr	309	Sets the number of ratchet iterations using SPR.
ratchettbr	309	Sets the number of ratchet iterations using TBR.
replicates	311	Sets the number of random addition sequence searches or the number of jackknife pseudo-replicates.
slop	314	Sets the percent limit of suboptimal trees evaluated during a search.
treefuse	327	Performs cladogram search using tree fusing.

Commands

This section contains a description of all POY commands, listed in alphabetical order.

about Prints general information about POY to standard output.

Syntax

```
-about
```

Arguments

none

Example

```
poy -about
```

agree Performs optimization subject to the constraint that specified clades are recovered.

Syntax

```
-agree confile
```

Arguments

Argument	Description
confile	A string. The name of the constraint file. One or more constraint files can be specified. Default = none

Example

```
poy chel.seq -constrain -agree pycno.con
```

Comments

This command modifies `constrain` (page 233) and must be used in conjunction with it. For a description of constraint file format requirements, refer to "Constraint file" on page 157.

allchroms Optimizes HTU chromosomes by examining all combinations of loci in the three adjacent HTU/OTU chromosomes.

noallchroms

Default value.

Syntax

```
-allchroms
-noallchroms
```

Arguments

None

Example

```
poy -chromosome -circular -rearrange chel.chrom
  -iterativepass -allchroms
```

Comments

1. Only operative when `iterativepass` (page 265) is used.
2. Not relevant if `fixedstates` (page 253) is used for chromosomal optimization.
3. Can be very time-consuming.

approxbuild Builds the initial tree using an approximation shortcut.

noapproxbuild

Default value. Builds the initial tree with corrections for heuristic tree calculation errors.

Syntax

```
-approxbuild
-noapproxbuild
```

Arguments

None

Example

```
poy chel.seq -approxbuild
```

Comments

1. The shortcut bypasses the correction of heuristic tree calculation errors and thereby reduces processing time. `noapproxbuild` (the default value) is slower, but yields a better initial tree.
2. `approxbuild` is useful when using `multirandom` (page 285).

approxdrift Calculates tree lengths for tree drifting using an approximation shortcut.

noapproxdrift

Default value. Calculates tree lengths without using shortcuts.

Syntax

```
-approxdrift
-noapproxdrift
```

Arguments

None

Example

```
poy chel.seq -drifttbr -numdrifttbr 5 -approxdrift
```

Comments

The shortcut bypasses the correction of heuristic tree calculation errors and thereby reduces processing time. `noapproxbuild` (the default value) is slower, but yields a better initial tree.

approxquickspr Calculates tree lengths for SPR using an approximation shortcut.

noapproxquickspr

Default value. Calculates tree lengths without shortcuts.

Syntax

```
-approxquickspr
-noapproxquickspr
```

Arguments

None

Example

```
poy chel.seq -spr -approxquickspr
```

Comments

The shortcut bypasses the correction of heuristic tree calculation errors and thereby reduces processing time. `noapproxquick-spr` (the default value) is slower, but yields a better initial tree.

approxquicktbr Calculates tree lengths for TBR using an approximation shortcut.

noapproxquicktbr

Default value. Calculates tree lengths without using a shortcut.

Syntax

```
-approxquicktbr
-noapproxquicktbr
```

Arguments

None

Example

```
poy chel.seq -approxquicktbr
```

Comments

The shortcut bypasses the correction of heuristic tree calculation errors and thereby reduces processing time. `noapproxquick-tbr` (the default value) is slower, but yields a better initial tree.

basefreq Reads nucleotide frequencies from a file.

Syntax

`-basefreq` *`freqfile`*

Arguments

Argument	Description
freqfile	A string. The name of the file containing the nucleotide frequencies. Default = none

Example

```
poy chel.seq -likelihood -basefreq base.frq
```

Comments

1. This command is used only under likelihood analysis.
2. Refer to "Frequencies file" on page 159.
3. When a file is not specified, POY estimates frequencies as specified by `estimateq` (page 251).

breakpoint

Sets the breakpoint cost.

Syntax

```
-breakpoint
```

Arguments

Argument	Description
cost	An integer. The cost of a breakpoint event in chromosomal optimization. Default = 10

Example

```
poy -chromosome -circular -rearrange -reversible
  chel.chrom -breakpoint 100
```


bremer

Calculates Bremer support values using TBR and writes results to standard output.

nobremer

Default value. Bremer support values are not calculated.

Syntax

```
-bremer
-nobremer
```

Arguments

None

Example

```
poy chel.seq -topofile chel.topo -replicates 0
  -constrain chel.con -bremer
```

Comments

1. A constraint file is required—refer to `constrain` (page 233).
2. Because TBR is used instead of collapsing ever more catholic searches, reported values are likely to overestimate group support.
3. If POY reports negative Bremer values, less costly trees were found and are reported to standard error. Increase the value of `checkslop` (page 228) to expand the cladogram search.

bremerspr Calculates Bremer support values using SPR and writes to standard output.

nobremerspr

Default value. Bremer support values are not reported.

Syntax

```
-bremerspr
-nobremerspr
```

Arguments

None

Example

```
poy chel.seq -topofile chel.topo -replicates 0
  -constrain chel.con -bremerspr
```

Comments

1. A constraint file is required—refer to `constrain` (page 233).
2. Because SPR is used instead of collapsing ever more catholic searches, reported values are likely to overestimate group support.
3. If POY reports negative Bremer values, less costly trees were found and are reported to standard error. Increase the value of `checkslop` (page 228) to expand the cladogram search.

buildchrom Optimizes chromosomal characters by building HTU chromosomes one locus at a time in the manner of cladogram building. Default value.

nobuildchrom

Optimization is not one locus at a time.

Syntax

```
-buildchrom
-nobuildchrom
```

Arguments

None

Example

```
poy -chromosome -circular -rearrange chel.chrom
  -iterativepass -buildchrom
```

Comments

1. Only operative when `iterativepass` (page 265) is used.
2. Not relevant if `fixedstates` (page 253) is used for chromosomal optimization.
3. Can be very time- consuming.

buildmaxtrees Sets the maximum number of trees held during initial cladogram builds.

Syntax

```
-buildmaxtrees maxtrees
```

Arguments

Argument	Description
maxtrees	An integer. The maximum number of trees held in buffers during a single build process. Default = the value specified by `maxtrees` (page 282)

Example

```
poy chel.seq -buildmaxtrees 1
```

Comments

See `fitchtrees` (page 253), `holdmaxtrees` (page 260), and `maxtrees` (page 282).

buildslop Sets the percent limit of suboptimal trees evaluated during the cladogram building process.

Syntax

```
-buildslop limit
```

Arguments

Argument	Description
limit	An integer. The limit of suboptimal trees evaluated, expressed in tenths of a percent (that is, a value of 5 represents a 0.5% limit). Default = 0 or the value set by `slop` (page 314)

Example

```
poy chel.seq -buildslop 2 -slop 5 -checkslop 10
```

Comments

1. While this command slows the build process, it compensates for shortcuts taken in calculating tree costs.
2 This command causes POY to evaluate the least cost trees and those costlier trees within the specified percentage.
3. This command does not affect the cladogram search process. See `checkslop` (page 228) and `slop` (page 314).

buildsperreplicate Performs a specified number of addition sequences in the build phase of a single replicate.

Syntax

```
-buildsperreplicate addseq
```

Arguments

Argument	Description
addseq	An integer. The number of addition sequences used in the build phase of a replicate. Default = `oneasis` (page 289) is on so that the first addition sequence of the first replicate is from the first data file specified on the command line; subsequent addition sequences are pseudorandom

Example

```
poy chel.seq -replicates 10 -buildsperreplicate 3
```

Comments

If `oneasis` (page 289) is specified, all addition sequences are pseudorandom.

buildspr Performs SPR branch swapping after the initial cladogram build holding only a single tree.

nobuildspr

Default value. POY does not perform an SPR cladogram search after the initial build.

Syntax

```
-buildspr
-nobuildspr
```

Arguments

None

Example

```
poy chel.seq -replicates 10 -buildsperreplicate 3
  -buildspr
```

Comments

`buildspr` attaches an SPR refinement to each build before other refinements or build replicates take place. This is in opposition to SPR, which would operate after all build replicates are completed.

buildtbr Performs TBR branch swapping after the initial cladogram build holding only a single tree.

nobuildtbr

Default value. POY does not perform a TBR cladogram search after the initial build.

Syntax

```
-buildtbr
-nobuildtbr
```

Arguments

None

Example

```
poy chel.seq -replicates 10 -buildsperreplicate 3
  -buildtbr
```

Comments

`buildtbr` attaches a TBR refinement to each build before other refinements or build replicates take place. This is in opposition to TBR, which would operate after all build replicates are completed.

cat_cmdgroups Writes a list of commands grouped by function to the standard error file.

Syntax

```
-cat_cmdgroups
```

Arguments

None

Example

```
poy -cat_cmdgroups
```

cat_commandbrowsing Writes a list of commands grouped to browse command documentation to the standard error file.

Syntax

```
-cat_commandbrowsing
```

Arguments

None

Example

```
poy -cat_commandbrowsing
```

cat_helptopics Writes a list of commands grouped to browse command documentation to the standard error file.

Syntax

```
-cat_helptopics
```

Arguments

None

Example

```
poy -cat_helptopics
```

cat_infiles Writes information about valid input file formats to the standard error file.

Syntax

```
-cat_infiles
```

Arguments

None

Example

```
poy -cat_infiles
```

cat_introduction Writes information about analysis using POY to the standard error file.

Syntax

```
-cat_introduction
```

Arguments

None

Example

```
poy -cat_introduction
```

cat_outfiles Writes information about valid output file formats to the standard error file.

Syntax

```
-cat_outfiles
```

Arguments

None

Example

```
poy -cat_outfiles
```

cat_programflow Writes information about POY's structure and execution to the standard error file.

Syntax

```
-cat_programflow
```

Arguments

None

Example

```
poy -cat_programflow
```

Comments

When `-helpfile EREF` (page 259) is specified, a flowchart in .PNG format is output to `EREF.png`.

cat_references Writes a list of the literature referred to in command line help to the standard error file.

Syntax

```
-cat_references
```

Arguments

None

Example

```
poy -cat_references
```

cat_setupparallel Writes information about how to set up a job for parallel processing to the standard error file.

Syntax

```
-cat_setupparallel
```

Arguments

None

Example

```
poy -cat_setupparallel
```

catchslaveoutput When using `parallel` (page 293), writes all standard output of slave tasks to the standard output of the master task.

nocatchslaveoutput

Default value.

Syntax

```
-catchslaveoutput
-nocatchslaveoutput
```

Arguments

None

Example

```
poy chel.seq -parallel -catchslaveoutput
```

Comments

This command is useful in monitoring and debugging parallel operations by spawned jobs.

change Sets the cost for nucleotide change.

Syntax

```
-change chngcost
```

Arguments

Argument	Description
chngcost	An integer. The nucleotide change cost. Default = 1

Example

```
poy chel.seq -change 2
```

characterweights Writes search statistics for each character to standard output.

nocharacterweights

Default value. POY does not write search statistics for each character to standard output.

Syntax

```
-characterweights
-nocharacterweights
```

Arguments

None

Example

```
poy chel.seq -characterweights
```

Comments

See also `stats` (page 318).

checkfrequency Sets the frequency of message checking when performing parallel ratcheting.

Syntax

`-checkfrequency` *`msgfreq`*

Arguments

Argument	Description
`msgfreq`	An integer. Default = 0

Example

```
poy chel.seq chel.morph -parallel -ratchettbr 100
  -multiratchet -checkfrequency 1
```

Comments

Used only with `multiratchet` (page 285).

checkslop Sets the percent limit of suboptimal trees evaluated during a final TBR refinement.

Syntax

`-checkslop` *`limit`*

Arguments

Argument	Description
limit	An integer. The limit of suboptimal trees evaluated, expressed in tenths of a percent (that is, a value of 5 represents a 0.5% limit). Default = 0 or the value set by `slop` (page 314)

Example

```
poy chel.seq -slop 1 -checkslop 10
```

Comments

1. While this command is time-consuming, additional TBR refinement compensates for shortcuts taken in calculating tree costs.
2. This command causes POY to evaluate the least cost trees and those costlier trees within the specified percentage.
3. Evaluation is performed after TBR branch swapping.
4. High values specified for `checkslop` in conjunction with low values for `slop` (page 314) are often an efficient search strategy.

chromosome Causes POY to use the chromosome character type.

nochromosome

Turns off POY's use of chromosome character type.

Syntax

```
-chromosome
-nochromosome
```

Arguments

None

Example

```
poy -chromosome -linear chel.chrom -nochromosome
  chel.morph
```

Comments

1. The chromosome character type is based on DNA sequences of multiple loci. The input file looks like the input file for the molecular character type, except a pipe (`|`) separates the DNA strands of different loci. POY checks all loci of all OTUs to determine the unique fragments, and then it uses the Needleman–Wunsch algorithm to determine the cost of aligning each

of these fragments with each other fragment. In the optimization phase, chromosomes are "aligned" by treating entire loci as the units to align. The cost of a "chromosome alignment" is the sum of the costs of aligned fragments in the two taxa (for which the cost has already been determined using NW) plus the cost of gaps that appear against entire loci. See `locusgap` (page 279) and `locussizegap` (page 279) for information on how the gap cost is calculated.

2. If the command `rearrange` (page 310) is used, then different orderings of the loci are also tried and breakpoint costs are also included in the cost of the chromosome alignment.
3. The algorithm used for the chromosome alignment is like the NW algorithm, and it is guaranteed to find the best placing of gaps for a given ordering of loci. However, if you allow for reordering, it is not guaranteed to find the best overall cost because it cannot try out all possible orders. See `rearrange` for more information.
4. When the chromosomes are aligned, the loci that are aligned with each other can be considered to be homologous loci.
5. Optimization can be accelerated greatly by the using `fixedstates` (page 253).

chromfilter Ignores all chromosome fragments whose length is greater than 0 and less than a specified length.

Syntax

```
-chromfilter length
```

Arguments

Argument	Description
length	An integer. The size of fragments filtered. Default = 0

Example

```
poy -chromosome -circular -rearrange chel.chrom
  -chromfilter 10
```

Comments

Use with `chromosome` (page 229) to ignore small fragments of DNA between loci.

circular Treats chromosomal characters as circular, as opposed to linear. Default value.

nocircular

Chromosomal characters are treated as linear. Has the effect of `linear` (page 278).

Syntax

```
-circular
-nocircular
```

Arguments

None

Example

```
poy -chromosome -circular -rearrange -reversible
  chel.chrom
```

Comments

In a circular chromosome, the first and last loci are adjacent.

clist Displays a list of all commands cross-referenced to commands that contain a specified string in their usage information.

Syntax

```
-clist xref
```

Arguments

Argument	Description
xref	A string. Command usage information is searched for this string, to which commands are cross-referenced. Default = none

Example

```
poy -clist change
```

cllist Displays a list with descriptions of all commands cross-referenced to commands that contain a specified string in their usage information.

Syntax

```
-cllist xref
```

Arguments

Argument	Description
xref	A string. Command usage information is searched for this string, to which commands are cross-referenced. Default = none

Example

```
poy -cllist change
```

commandfile Reads the content of a specified file and writes it into the command line.

Syntax

```
-commandfile commfile
```

Arguments

Argument	Description
commfile	A string. The name of a command file. Default = `poy.ini`

Example

```
poy chel.seq -commandfile quick.command
```

Comments

1. Refer to "Command file" on page 159.
2. A command line can contain one or more instances of `commandfile`.
3. A command file can contain one or more instances of `commandfile` (that is, nested command files). However, care should be taken to ensure that there are no circular references.
4. Abbreviation: use `--` as an abbreviation for `commandfile` (for example, `-- commfil1.txt` is the same as `commandfile commfil1.txt`.

commandfiledir Sets the directory in which POY command files are located.

Syntax

```
-commandfiledir dirname
```

Arguments

Argument	Description
dirname	A string. The name of the directory in which command files are located. Default = the current directory

Example

```
poy -commandfiledir /usr/share/commands chel.seq
  -commandfile quick.command
```

expands to

```
poy chel.seq -commandfile /usr/share/commands/
  quick.command
```

Comments

A command line can contain one or more instances of `commandfiledir`.

compressstates Tracks states during the build process of search-based optimization, then removes unused states during the cladogram search.

nocompressstates

Default value. POY does not track states during search-based optimization.

Syntax

```
-compressstates
-nocompressstates
```

Arguments

None

Example

```
poy -fixedstates chel.seq -newstates chel.states
  -compressstates
```

Comments

1. This command can reduce processing time, but can also result in suboptimal solutions.
2. If `checkslop` (page 228) is specified, the state set is returned to its original size for the final round of cladogram search.
3. The initial state set is specified by `newstates` (page 286).

constrain Reads cladogram search constraints from one or more specified files.

noconstrain

Turns off reading of files as constraint files.

Syntax

```
-constrain confile -noconstrain
```

Arguments

Argument	Description
confile	A string. The name of the constraint file. One or more constraint files can be specified. Default = none

Example

```
poy chel.seq -constrain pycno.con -disagree
  spiders.con -nocontrain chel.morph
```

Comments

1. Refer to "Constraint file" on page 157.
2. This command is intended for use with static data. It specifies the character states that must be recovered without homoplasy.
3. `constrain` remains on until noconstrain is specified.
4. Multiple constraint files can be specified.

controllers Sets the number of subclusters used in a parallel cladogram search.

Syntax

```
-controllers clustnum
```

Arguments

Argument	Description
clustnum	The number of subclusters used in a cladogram search. Default = 1

Example

```
poy chel.seq -parallel -controllers 5 -replicates 50
  -multirandom
```

Comments

1. This command shortens processing time by balancing two issues in parallel processing: minimizing overhead versus maximizing node utilization. It is particularly useful in performing randomized cladogram search strategies, where the number of

replicates is matched to the size of subclusters (subclusters perform replicates independently).

2. This command divides the slave nodes into the specified number of groups and spawns controller tasks for each subcluster. For example, `-controllers 3` on a cluster of 90 slave nodes creates three clusters of 30 nodes each; on a cluster of 92 slave nodes, one cluster of 30 nodes is created and two clusters of 31 nodes are created.
3. By creating subclusters of 16 to 32 nodes, POY is able to maintain good parallel performance while ensuring that all nodes are efficiently utilized. For example, specifying `-controllers 3 -replicates 30` (page 311) on a 90 slave node cluster causes POY to perform replicates on one subcluster while it performs other tasks on the remaining nodes. Otherwise, POY would utilize 30 of the 90 nodes for the random search while the other 60 nodes remained idle.

crashcontroller When using `parallel` (page 293) and `controllers` (page 234), causes the controller to crash.

Syntax

`-crashcontroller` *`crash`*

Arguments

Argument	Description
`crash`	An integer. The lower the value, the more frequently the controllers will crash.

Example

```
poy chel.seq -parallel -controllers 5 -replicates 50
  -multirandom -crashcontroller 50
```

Comments

Use this command for testing and verifying fault-tolerant features in parallel.

crashreserve When using `parallel` (page 293) and `dpm` (page 241), causes reserve slave jobs to interrupt calculations and exit.

Syntax

`-crashreserve` *`crash`*

Arguments

Argument	Description
crash	An integer. The lower the value, the more frequently the process will exit.

Example

```
poy chel.seq -parallel -controllers 5 -replicates 50
  -multirandom -crashreserve 50
```

Comments

Use this command for testing and verifying fault-tolerant features in parallel.

crashslave When using `parallel` (page 293), causes slave jobs to interrupt calculations and exit.

Syntax

```
-crashslave crash
```

Arguments

Argument	Description
crash	An integer. The lower the value, the more frequently the process will exit.

Example

```
poy chel.seq -parallel -controllers 5 -replicates 50
  -multirandom -crashslave 50
```

Comments

Use this command for testing and verifying fault-tolerant features in parallel.

cutswap Performs a shortcut during branch swapping based on approximate tree costs.

nocutswap

Default value. POY uses swapping shortcuts.

Syntax

```
-cutswap
-nocutswap
```

Arguments

None

Example

```
poy chel.seq -cutswap
```

Comments

This command can reduce processing time, but can also result in suboptimal solutions.

datadir Sets the directory location for input data files.

Syntax

```
-datadir dirname
```

Arguments

Argument	Description
dirname	The name of the directory in which input data files are located. This command works with both character data files and command-invoked input files such as `molecularmatrix` (page 283) Default = the current directory

Example

```
poy -datadir /gentins/Lisianthius its1a its1b
  its1c -datadir /Costs -molecularmatrix cost1
```

expands to

```
poy /gentins/Lisianthius/its1a /gentins/Lisianthius/
  its1b /gentins/Lisianthius/its1c -molecularmatrix
  /Costs/cost1
```

defaultweight Sets the weight of all character data that follow.

Syntax

```
-defaultweight weight
```

Arguments

Argument	Description
weight	An integer. The weight assigned to character data in files following the command. Default = 1

Example

```
poy -defaultweight 2 -weight 3 chel.morph chel.seq
```

`chel.morph` receives weight 3 while `chel.seq` receives weight 2.

Comments

`weight` (page 329) used in association with an individual file overrides `defaultweight`.

deletegapsfrominput Deletes gaps from sequences entered in FASTA format.

Syntax

```
-deletegapsfrominput
```

Arguments

None

Example

```
poy chel.fasta -deletegapsfrominput
```

Comments

1. Refer to "FASTA Format" on page 155.
2. Gaps coded in sequences are deleted, in contrast to `fastashortname` (page 252), which enables annotations and retains gaps.

diagnose Writes branch costs and apomorphy lists to standard output.

nodiagnose

Default value. POY writes only the tree cost.

Syntax

```
-diagnose
-nodiagnose
```

Arguments

None

Example

```
poy chel.seq -diagnose
```

Comments

This command works with either the results of an analysis or directly from a topology (for example, `topology` on page 323).

disagree Performs a cladogram search such that clades specified by a constraint file are not recovered.

Syntax

```
-disagree -constrain confile
```

Arguments

Argument	Description
confile	A string. The name of the constraint file. Default = none

Example

```
poy chel.seq -constrain -disagree pycno.con
```

Comments

1. Refer to "Constraint file" on page 157.
2. This command can be used to report Bremer values for clades, in contrast to `bremer` (page 220), which reports values for all clades by comparing constrained and unconstrained search results.
3. Use this command to report least cost trees not specified by `constrain`.

discrepancies Writes the cost difference between trees found using shortcuts and trees found on a complete down pass.

nodiscrepancies

Default value. POY writes only the tree cost from the analysis using shortcuts.

Syntax

```
-discrepancies
-nodiscrepancies
```

Arguments

None

Example

```
poy chel.seq -nodiscrepancies
```

Comments

Results are written to standard output.

dlist Displays a list of commands that contain a specified string in their usage information.

Syntax

```
-dlist xref
```

Arguments

Argument	Description
xref	A string. Command usage information is searched for this string. Default = none

Example

```
poy -dlist gap
```

dllist Displays a list with descriptions of commands that contain a specified string in their usage information.

Syntax

```
-dllist xref
```

Arguments

Argument	Description
xref	A string. Command usage information is searched for this string. Default = none

Example

```
poy -dllist gap
```

dogaptie During optimization, specifies which sequence in which a gap is placed when a tie occurs in the dynamic programming table.

Syntax

```
-dogaptie select
```

Arguments

Argument	Description
select	A string. A predefined code that specifies the sequence in the dynamic programming table into which a gap is to be inserted. `dogaptie` accepts the following values. **Value** — **Description** `shorter` — Insert the gap in the shorter sequence `longer` — Insert the gap in the longer sequence `nopreference` — POY selects the sequence arbitrarily based on the order of descendant sequences Default = `nopreference`

Example

```
poy chel.seq -dogaptie longer
```

Comments

1. When inserting gaps while building a dynamic programming table during direct optimization (`topology` on page 323), the value of placing the gap in either constituent sequence might be the same. Intermediate sequences might be different with different gap placements. This command directs POY as to how to resolve these ties.
2. In previous versions of POY, ties were resolved de facto using the `nopreference` option. The `shorter` and `longer` options have been introduced in order to make cost calculations independent of the order of siblings in a tree.
3. When diagnosing trees from a previous run (refer to `topofile` on page 322 and `topology` on page 323), make sure to use the same setting for this command.
4. This command also affects iterative pass optimization (`iterativepass` on page 265).

dpm

Performs load balancing by using dynamic process migration.

nodpm

Default value. POY does not perform Sankoff optimization.

Syntax

```
-dpm
-nodpm
```

Arguments

None

Example

```
poy chel.seq -parallel -dpm
```

Comments

Dynamic process migration moves processes from overutilized to underutilized nodes. Dpm causes additional slave processes to be spawned. Performance can be significantly improved when

- Simultaneously running more than one POY script
- Nodes have significantly different processing speeds.

dpmacceptratio Sets the threshold parameter for dynamic process migration.

Syntax

```
-dpmacceptratio dpmparam
```

Arguments

Argument	Description
dpmparam	A decimal. The ratio of processor performance that causes POY to move a task to an underutilized node. Default = 1.5

Example

```
poy -chel.seq -parallel -dpm -dpmacceptratio 1.2
```

Comments

The performance ratio is a combined measure of processor load and processor speed.

dpmautoadjustperiod Adjusts the frequency with which performance is evaluated for dynamic process migration.

nodpmautoadjustperiod

Default value. POY does not adjust the dynamic process migration performance ratio.

Syntax

```
-dpmautoadjustperiod
-nodpmautoadjustperiod
```

Arguments

None

Example

```
poy chel.seq -parallel -dpm -dpmautoadjustperiod
```

Comments

When a performance evaluation results in a task migration, the time period to the next evaluation is reduced. When no migration occurs, the time period is increased.

dpmjobspernode Sets the number of reserve jobs spawned for dynamic process migration.

Syntax

```
-dpmjobspernode jobsnum
```

Arguments

Argument	Description
jobsnum	An integer. The number of reserve jobs spawned for dynamic process migration. Default = 1

Example

```
poy chel.seq -parallel -jobspernode 2 -dpm
  -dpmjobspernode 1
```

dpmmaxperiod Sets the maximum time period used in adjusting the frequency of performance evaluation by `dpmautoadjustperiod`.

Syntax

```
-dpmmaxperiod maxtime
```

Arguments

Argument	Description
maxtime	An integer. The maximum time period allowed for `dpmautoadjustperiod` in adjusting the frequency with which performance is evaluated for dynamic process migration. Default = 100

Example

```
poy chel.seq -parallel -dpm -dpmautoadjustperiod
  -dpmmaxperiod 25
```

dpmmaxprocessors For parallel processing using dynamic process migration, sets the number of reserve jobs to spawn.

Syntax

```
-dpmmaxprocessors maxjobs
```

Arguments

Argument	Description
maxjobs	An integer. The maximum number of reserve jobs that can be spawned for dynamic process migration. Default = the number of nodes in the cluster

Example

```
poy chel.seq -parallel -dpm -dpmmaxprocessors 50
```

dpmminperiod Sets the minimum time period used in adjusting the frequency of performance evaluation by `dpmautoadjustperiod`.

Syntax

```
-dpmminperiod mintime
```

Arguments

Argument	Description
mintime	An integer. The minimum time period allowed for `dpmautoadjustperiod` in adjusting the frequency with which performance is evaluated for dynamic process migration. Default = 1

Example

```
poy chel.seq -parallel -dpm -dpmautoadjustperiod
  -dpmminperiod 5
```

dpmnumslaveprocesses Sets the maximum number of reserve jobs spawned on PVM nodes, excluding the master node.

Syntax

```
-dpmnumslaveprocesses procs
```

Arguments

Argument	Description
procs	An integer. The maximum number of reserve jobs spawned on PVM nodes. Default = number of entries in PVM configuration - 1 × the value *jobsnum* set by `jobspernode` (page 272)

Example

```
poy chel.seq -parallel -dpm -dpmnumslaveprocesses 50
```

dpmonannum Sets the number of reserve jobs spawned on the master node.

Syntax

`-dpmonannum` *jobs*

Arguments

Argument	Description
jobs	An integer. The number of reserve jobs spawned on the master node. Default = 0

Example

```
poy chel.seq -parallel -dpm -dpmmonannum 10
```

Comments

`onan` (page 288) must be in the command line. This command is likely to be most useful on multiprocessor machines.

dpmperiod For parallel processing using dynamic process migration, sets the initial time period used by `dpmautoadjustperiod`.

Syntax

`-dpmperiod` *inittime*

Arguments

Argument	Description
inittime	An integer. The initial time period assigned for use by `dpmautoadjustperiod` in adjusting the frequency with which performance is evaluated for dynamic process migration. Default = 10

Example

```
poy chel.seq -parallel -dpm -dpmperiod 5
```

dpmproblemsize Sets the size of the calculation used to evaluate performance for dynamic process migration.

Syntax

```
-dpmproblemsize dpmsize
```

Arguments

Argument	Description
dpmsize	An integer. The size of the performance evaluation calculation used for dynamic process migration. Default = 5000

Example

```
poy chel.seq -parallel -dpm -dpmproblemsize 500
```

Comments

To improve performance evaluation, increase the calculation size, but this will add overhead to the dpm process.

dpmsolospawn Sets the number of reserved tasks spawned in PVM when there is only a single node.

Syntax

```
-dpmsolospawn tasks
```

Arguments

Argument	Description
tasks	An integer. The number of reserve tasks spawned. Default = 1

Example

```
poy chel.seq -parallel -dpm -dpmsolospawn 50
```

Comments

This command is most useful on multiprocessor machines.

driftequallaccept Sets the percentage of instances to accept equally parsimonious trees during tree drifting.

Syntax

```
-driftequallaccept pcttime
```

Arguments

Argument	Description
pcttime	An integer. The percent of time in which to accept equally parsimonious trees. Default = 50

Example

```
poy chel.seq -driftbr -numdriftbr 10
  -driftequallaccept 10
```

driftlengthbase Sets a parameter used to specify the probability of accepting a suboptimal tree.

Syntax

```
-driftlengthbase base
```

Arguments

Argument	Description
base	An integer. The cost used as a base in setting the probability of accepting a suboptimal tree. Default = 2

Example

```
poy chel.seq -driftbr -numdriftbr 10
  -driftlengthbase 3
```

driftspr Performs tree drifting using SPR.

nodriftspr

Turns off SPR tree drifting.

Syntax

```
-driftspr
-nodriftspr
```

Arguments

None

Example

```
poy chel.seq -driftspr -numdriftspr 10
```

Comments

The number of drifting rounds is specified by `numdriftspr` (page 287).

drifttbr Performs tree drifting using TBR.

nodrifttbr

Turns off TBR tree drifting.

Syntax

```
-drifttbr
-nodrifttbr
```

Arguments

None

Example

```
poy chel.seq -drifttbr -numdrifttbr 10
```

Comments

The number of drifting rounds is specified by `numdrifttbr` (page 287).

dropconstraints Performs cladogram building using constraints, then performs cladogram refinement without constraints.

nodropconstraints

Default value. Performs cladogram building and cladogram search using constraints.

Syntax

```
-dropconstraints
-nodropconstraints
```

Arguments

None

Example

```
poy chel.seq -constrain pycno.con -dropconstraints
```

editnames Prevents a program abort when reserved characters are encountered in a terminals file (`terminalsfile` on page 320) or character data file ("Input Data" on page 147).

Syntax

```
-editnames chars
```

Arguments

Argument	Description
chars	A string. Six characters that replace the reserved characters `()[]*;`. Replacement characters must be in the order shown in the preceding sentence.

Example

```
poy chel.seq -editnames "himom"
```

Comments

The substitution will not occur in tree topologies specified by `topofile` (page 322) and `topology` (page 323).

enabletmpfiles Uses temporary files for preprocessing tasks.

noenabletmpfiles

Default value. Temporary files are not used.

Syntax

```
-enabletmpfiles
-noenabletmpfiles
```

Arguments

None

Example

```
poy -topofile chel.topo -printtree -plotfile
  chel.tree -enabletmpfiles
```

Comments

1. Three temporary files are used: `poy1.tmp`, `poy2.tmp`, and `poy3.tmp`.
2. The temporary files are located in the current directory.

Caution Running more than one POY job in the same directory simultaneously with `enabletmpfiles` enabled can cause all jobs to crash.

3. Existing temporary files are overwritten, regardless of whether `overwriteprotection` (page 292) is enabled.
4. Temporary files are used for two purposes:
   - Convert Mac OS X and DOS text files to Unix text format.
   - When no character data are specified, temporary files are used for plotting trees or calculating consensus trees.

estimatep Uses nucleic acid and gap frequencies estimated from input data and pairwise alignments, which are then fixed for the duration of the search.

noestimatep

Default value. Reestimates frequencies as each sequence pair is encountered.

Syntax

```
-estimatep
-noestimatep
```

Arguments

None

Example

```
poy chel.seq -likelihood -estimatep
```

estimateparamsfirst Uses likelihood parameters estimated from input data and pairwise alignments, which are then fixed for the duration of the search.

noestimateparamsfirst

Default value. Reestimates parameters for each sequence pair as they are encountered.

Syntax

```
-estimateparamsfirst
-noestimateparamsfirst
```

Arguments

None

Example

```
poy chel.seq -likelihood -estimateparamsfirst
```

estimateq Uses probabilities for transitions estimated from pairwise alignments, which are then fixed for the duration of the search.

noestimateq

Default value. Reestimates transition probabilities for each sequence pair as they are encountered.

Syntax

```
-estimateq
-noestimateq
```

Arguments

None

Example

```
poy chel.seq -likelihood -estimateq
```

exact Calculates tree costs more accurately for direct optimization on the down pass by creating an implied alignment and performing a complete Sankoff optimization.

noexact

Default value. Calculates tree costs less accurately.

Syntax

```
-exact
-noexact
```

Arguments

None

Example

```
poy chel.seq -exact
```

Comments

1. This command improves tree cost calculations when `slop` (page 314) is used.
2. This command increases calculation time.

extensiongap Sets the cost of gaps that follow the first in a series of gaps.

Syntax

```
-extensiongap gapcost
```

Arguments

Argument	Description
gapcost	An integer. The cost of gaps in a series that follows the first. Default = gap cost set by `gap` (page 257) or `molecularmatrix` (page 283)

Example

```
poy chel.seq -gap 2 -extensiongap 1
```

Comments

1. When this command is not used, the cost of a string of gaps is a multiple of the single gap cost set using `gap` (page 257) or `molecularmatrix` (page 283).
2. When this command is used, the cost of a series of contiguous gaps of length *n* is

   $$first\_gap\_cost + gapcost \times (n - 1) \qquad \text{(Eq 15.1)}$$

   where *first_gap_cost* is the gap cost set by either `gap` or `molecularmatrix`.

fastashortname Enables annotations in Fasta format sequence files.

nofastashortname

Default value. POY reads the entire line as the taxon name.

Syntax

```
-fastashortname
-nofastashortname
```

Arguments

None

Example

```
poy chel.fasta -fastashortname
```

Comments

1. This command causes POY to read the first contiguous string after the marker `>` as a taxon name. The rest of the string is ignored. For example, in `> taxona comments on taxona` the taxon name is read as `taxona`. On the other hand, in the absence of the command the taxon name is read as `taxona_comments_on_taxona`.
2. Refer to "FASTA Format" on page 155.

3. Gaps coded in sequences are retained, in contrast to `delete-gapsfrominput` (page 238), which enables annotations and deletes gaps.

fitchtrees Ensures the randomness of the subset of trees stored in buffers by adding additional trees to full buffers through random replacement of stored trees.

nofitchtrees

Default value. Trees are stored in buffers until full. Additional trees are discarded.

Syntax

```
-fitchtrees
-nofitchtrees
```

Arguments

None

Example

```
poy -maxtrees 3 -fitchtrees
```

Comments

This command is based on the algorithm of W. Fitch. This command is useful in keeping tree diversity even when buffers are made small to speed up searches.

fixedstates Performs fixed state optimization on a specified file.

nofixedstates

Turns off fixed state optimization.

Syntax

```
-fixedstates
-nofixedstates
```

Arguments

None

Example

```
poy -fixedstates chel.seq
```

Comments

1. This command must be placed before the data input file name in the command line. For example, for the command line

   ```
   poy file1 -fixedstates file2 -seed -1
     -nofixedstates file3
   ```

   only `file2` data are optimized using fixed state optimization.

2. Applies to chromosomal characters.

freqmodel Sets how nucleic acid and gap frequencies are estimated.

Syntax

```
-freqmodel freq
```

Arguments

Argument	Description
freq	A string. A predefined code that specifies the number of nucleotide frequencies to estimate. `freqmodel` accepts the following values: **Value** — **Description** `f5` — All nucleotides and gaps `f2` — Two classes: nucleotides and gaps `f1` — All frequencies set to 0.2 Default = `f5`

Example

```
poy chel.seq -likelihood freqmodel f2
```

fuseafterreplicates Performs tree fusing on the results of random replicates as they are calculated. This is the default value when tree fusing (`treefuse` on page 327) is on.

nofuseafterreplicates

Turns off tree fusing on the results of random replicates as they are calculated.

Syntax

```
-fuseafterreplicates
-nofuseafterreplicates
```

Arguments

None

Example

```
poy chel.seq -replicates 10 -treefuse
  -fuseafterreplicates
```

fuselimit Sets the maximum number of tree fusing pairs held in memory.

Syntax

```
-fuselimit pairlim
```

Arguments

Argument	Description
pairlim	An integer. The maximum number of tree fusing pairs held in memory. Default = the maximum available in memory

Example

```
poy chel.seq -treefuse -fuselimit 10
```

fusemaxtrees Sets the maximum number of trees held in memory during tree fusing.

Syntax

```
-fusemaxtrees maxnum
```

Arguments

Argument	Description
maxnum	An integer. The maximum number of trees held in memory during tree fusing. Default = the maximum available in memory

Example

```
poy chel.seq -treefuse -fusemaxtrees 50
```

fusemingroup Sets the minimum number of taxa in a subtree that can be exchanged.

Syntax

```
-fusemingroup minnum
```

Arguments

Argument	Description
minnum	An integer. The minimum number of taxa in a subtree that can be exchanged during tree fusing. Default = 5

Example

```
poy chel.seq -treefuse -fusemingroup 3
```

fusingrounds Performs tree fusing a specified number of times.

Syntax

```
-fusingrounds fusenum
```

Arguments

Argument	Description
fusenum	An integer. The number of tree fusings to perform after the initial tree fusing. Default = 2

Example

```
poy chel.seq -treefuse -fusingrounds 1
```

gammaalpha Sets the alpha parameter for the gamma distribution.

Syntax

```
-gammaalpha alpha
```

Arguments

Argument	Description
alpha	An integer. The alpha parameter for the gamma distribution. Default = none

Example

```
poy chel.seq -likelihood -gammaclasses 3 -gamma 0.2
```

Comments

Use this command when gamma classes are set greater than zero using `gammaclasses` (below).

gammaclasses Sets the number of rate classes for a discrete gamma distribution.

Syntax

```
-gammaclasses gamma
```

Arguments

Argument	Description
gamma	An integer. The number of rate classes for a discrete gamma distribution. Default = 0

Example

```
poy chel.seq -likelihood -gammaclasses 3
```

gc Writes Ocaml garbage collection statistics at the end of run to the standard error file.

nogc

Default value. Ocaml garbage collection statistics are not written to the standard error file.

Syntax

```
-gc
-nogc
```

Arguments

None

Example

```
poy chel.seq -gc
```

Comments

Use this command for debugging and memory usage information.

gap Sets the cost of gaps.

Syntax

```
-gap gapcost
```

Arguments

Argument	Description
gapcost	An integer. The cost assigned to gaps. Default = 2

Example

```
poy chel.seq -gap 1
```

Comments

1. If `extensiongap` (page 251) is used, this command sets the cost of the first gap in a series of gaps, while `extensiongap` sets the cost of subsequent gaps. For example, the cost of a series of contiguous gaps of length *n* is

   $$gapcost + extension\_gap\_cost \times (n - 1) \qquad \text{(Eq 15.2)}$$

   where *extension_gap_cost* is the gap cost set by `extensiongap`.
2. Gap costs set using `molecularmatrix` (page 283) take precedent over costs set by `gap`.

goloboff Performs implied weights optimization.

Syntax

`-goloboff` *weights*

Arguments

Argument	Description
weights	A string. A predefined code that specifies the weight parameters to use in the optimization. `goloboff` accepts the following values: **Value** — **Description** `ck` — Performs implied weighting as described in Goloboff (1993b) `cm` — Uses a modified form of Goloboff's implied weighting formula `ri` — Weights using the retention index Default = none

Example

```
poy chel.seq -goloboff ck
```

Comments

1 See also `kfactor` (page 273) and `multiplier` (page 284).

2 `ck` uses the following weighting formula:

$$\frac{(k+1)}{(k+1+s+m)} \qquad \text{(Eq 15.3)}$$

where

k = Goloboff's concavity factor

s = cost of character on the tree

m = minimum cost of the character

help Displays all commands with their descriptions.

Syntax

```
-help
```

Arguments

None

Example

```
poy -help
```

helpdumpfile Writes all available help to an .HTML and .PNG file using a specified file name.

Syntax

```
-helpdumpfile outname
```

Arguments

Argument	Description
outname	A string. The name of the file to which all available help writes output. Two files are created: *outname*.html and *outname*.png.

Example

```
poy -helpdumpfile poydocs
```

helpfile Writes the requested command line help request to a specified file.

Syntax

```
-helpfile outname
```

Arguments

Argument	Description
outname	A string. The name of the file to which a requested help command writes output.

Example

```
poy -helpfile docs
```

helpwidth Sets the column width for help output.

Syntax

```
-helpwidth column
```

Arguments

Argument	Description
column	An integer. The column width in characters for help output. Default = 80

Example

```
poy -helpfile docs -helpwidth 120
```

Comments

1. A positive value of *column* that is less than 60 is read as 60.
2. A negative value of *column* is read as positive infinity (that is, output is one line).

holdmaxtrees Sets the maximum number of trees held in memory for all random replicates.

Syntax

```
-holdmaxtrees maxnum
```

Arguments

Argument	Description
maxnum	An integer. The maximum number of trees held in memory for all random replicates. Default = the maximum available in memory

Example

```
poy chel.seq -replicates 10 -maxtrees 1 -holdmaxtrees
  50
```

Comments

maxtrees (page 282) will limit tree buffer size within replicates while holdmaxtrees operates when replicate results are collected for final refinement. Low values for maxtrees can decrease execution time, while allowing larger tree buffers for completeness using holdmaxtrees.

hypancfile Sets the output file name for hypothetical ancestral nucleotide sequences.

Syntax

```
-hypancfile hypfile
```

Arguments

Argument	Description
hypfile	A string. The name of the file to which hypothetical ancestral nucleotide sequences are written. Default = `poy.hypanc`

Example

```
poy chel.seq -printhypanc -hypancfile chel.hypanc
```

Comments

1. Used in conjunction with `printhypanc` (page 303).
2. Unlike `diagnose` (page 238), output from this command resolves ambiguously optimized ancestral states.
3. Use this command to create source files for search-based optimization (refer to "Hypothetical ancestral states" on page 163).

hypancname Writes the names of hypothetical ancestral nucleotide sequences to standard output.

nohypancname

Default value. Hypothetical ancestors are not output.

Syntax

```
-hypancname
-nohypancname
```

Arguments

None

Example

```
poy chel.seq -printhypanc -hypancfile chel.hypanc
  -hypancname
```

Comments

Use this command in conjunction with `newstates` (page 286).

iafiles Writes implied alignments for specified trees to a series of files.

noiafiles

Turns off printing of implied alignments.

Syntax

```
-iafiles
-noiafiles
```

Arguments

None

Example

```
poy chel.seq -impliedalignment -iafiles
```

Comments

1. All output files are POY readable.
2. This commands writes implied alignments either from cladogram searches or from searches and input topologies. For example,

   ```
   poy seq1 seq2 -iafiles -topofile trees -nobuild
   ```

   uses the sequence data files `seq1` and `seq2` with the topology file `trees`.
3. The number of output files is equal to the number of trees that result from a sequence data file. If `trees` contains three trees, then there will be six output files—three for each of the two sequence data files.
4. The naming convention for output files from topology files is

   `ia.`*`topology_file_name`*

   where *`topology_file_name`* is the name of the topology data input file. For the example above, the output file name is `ia.trees`.
5. The naming convention for output files from sequence data is

   `sequence_file_name``.ia.`*`topology_file_i`*

   where *`sequence_file_name`* is the name of the sequence data input file and *`topology_file_i`* designates the i^{th} tree output to `ia.trees`. The command line above generates six output files.

   ```
   seq1.ia.tree0
   seq1.ia.tree1
   seq1.ia.tree2
   seq2.ia.tree0
   seq2.ia.tree1
   seq2.ia.tree2
   ```
6. If the sequence data is partitioned into fragments, output file will contain separate implied alignments for each fragment. Refer to "FASTA Format" on page 155.
7. While sequence fragments that have been excluded from the analysis do not contribute to tree costs, they are included in the output file.

impliedalignment Writes a topology-specific multiple alignment based on the synapomorphy scheme to standard output.

noimpliedalignment

Turns off writing alignments from the synapomorphy scheme.

Syntax

```
-impliedalignment
-noimpliedalignment
```

Arguments

None

Example

```
poy chel.seq -impliedalignment
```

Comments

Applies to chromosomal characters as well. Produces locus annotation information.

indices Writes fit statistics (consistency index and retention index) for a topology to standard output.

noindices

Turns off writing fit statistics.

Syntax

```
-indices
-noindices
```

Arguments

None

Example

```
poy chel.seq -indices
```

intermediate Writes intermediate search results to standard output.

nointermediate

Turns off writing fit statistics.

Syntax

```
-intermediate
-nointermediate
```

Arguments

None

Example

```
poy chel.seq -intermediate
```

Comments

1. This command can be helpful with protracted searches by recording intermediate results.
2. Use `catchslaveoutput` (page 227) to collect these from slave nodes in parallel runs.

invariantsitesadjust Adjusts likelihoods for theta fraction invariant sites.

noinvariantsitesadjust

Turns off likelihood adjustments for theta fraction invariant sites.

Syntax

```
-invariantsitesadjust
-noinvariantsitesadjust
```

Arguments

None

Example

```
poy chel.seq -likelihood -invariantsitesadjust
```

iterativeinitsingle Sets initial HTU states by resolving ambiguities in preliminary phase (that is, the down pass) from direct optimization.

noiterativeinitsingle

Default value. Initialization with preliminary phase HTU states.

Syntax

```
-iterativeinitsingle
-noiterativeinitsingle
```

Arguments

None

Example

```
poy chel.seq -iterativepass -iterativeinitsingle
```

iterativekeepbetter Retains the hypothetical ancestral states and cladogram cost that is the lesser of standard up/down pass and iterative pass.

noiterativekeepbetter

Default value. Uses the hypothetical ancestral state derived from the iterative pass.

Syntax

```
-iterativekeepbetter
-noiterativekeepbetter
```

Arguments

None

Example

```
poy chel.seq -iterativepass -iterativekeepbetter
```

Comments

Ensures that the lower cost reconstructed HTU states are maintained.

iterativelowmem Minimizes memory allocation during iterative pass optimization.

noiterativelowmem

Default value. No process is instantiated for minimizing memory allocation during iterative pass optimization.

Syntax

```
-iterativelowmem
-noiterativelowmem
```

Arguments

None

Example

```
poy chel.seq -iterativepass -iterativelowmem
```

Comments

This command can significantly reduce memory requirements under iterative pass, but there is a small execution time cost.

iterativepass Performs iterative pass optimization.

noiterativepass

Turn off iterative pass.

Syntax

```
-iterativepass
-noiterativepass
```

Arguments

None

Example

```
poy chel.seq -iterativepass
```

Comments

1. This command performs a three-dimensional optimization alignment (that is, it uses three vertices at each internal node).
2. This command is storage and time consumptive.

iterativepassfinal Uses iterative pass to reestimate final states.

nointerativepassfinal

Turns off use of iterative pass for reestimation of final states.

Syntax

```
-interativepassfinal
-nointerativepassfinal
```

Arguments

None

Example

```
poy chel.seq -iterativepass -iterativepassfinal
```

iterativerandom Resolves ambiguities in hypothetical ancestral states using random preference.

noiterativerandom

Turns off use of iterative pass for reestimation of final states.

Syntax

```
-iterativerandom
-noiterativerandom
```

Arguments

None

Example

```
poy chel.seq -iterativepass -iterativerandom
```

jackboot Performs Farris et al. (1996) jackknifing.

nojackboot

Turns off jackknifing.

Syntax

```
-jackboot
```

```
-nojackboot
```

Arguments

None

Example

```
poy chel.seq -replicates 100 -notbr -jackboot
  -maxtrees 2
```

Comments

1. Specify the number of pseudo-replicates using `replicates` (page 311).
2. Set the cladogram search strategy within each pseudo-replicate by using commands that affect `replicates`.
3. The default output consists of the strict consensus trees for the pseudo-replicates and a majority rule consensus tree of these. The clade frequencies in the majority rule consensus tree are the jackknife percentages.
4. To change the default output, use `jackoutgroup` (page 270), `jackpseudoconsensustrees` (page 270), `jackpseudotrees` (page 271), `jacktree` (page 271), and `jackwincladefile` (page 272).
5. Shallow cladogram search strategies can underestimate jackknife support values.
6. This form of jackknifing does not delete nucleotides, but reweights a fraction to zero, thereby preserving homology determination in length-variable sequences.

jackcharfile Writes the majority rule consensus tree derived from the strict consensus trees of all jackknife pseudo-replicates to a specified file in xread format ("Input Cladograms" on page 156).

Syntax

```
-jackcharfile consfile
```

Arguments

Argument	Description
consfile	A string. The name of the file to which the majority rule consensus tree is written in xread format.

Example

```
poy chel.seq -replicates 100 -notbr -jackboot
  -jackcharfile chel.jack
```

Comments

1. `jackboot` (page 266) or `jackstart` (page 271) must be specified.
2. The resulting file can be used as a constraint file (`jackstart`).

jackfpseudoconsensustrees Writes the strict consensus trees for the individual pseudo-replicates to a specified file.

Syntax

```
-jackfpseudoconsensustrees fname
```

Arguments

Argument	Description
fname	A string. The name of the file to which the strict consensus trees are written.

Example

```
poy chel.seq -replicates 100 -notbr -jackboot
  -jackfpseudoconsensustrees chel.jack_replicates
```

Comments

1. `jackboot` (page 266) or `jackstart` (page 271) must be specified.
2. Output is in parenthetical notation. For each consensus tree, the number of best cladograms on which it is based is indicated between square brackets.
3. See `jackpseudoconsensustrees` (page 270).

jackfpseudotrees Writes the least cost trees for all pseudo-replicates to a specified file.

Syntax

```
-jackfpseudotrees bestrees
```

Arguments

Argument	Description
bestrees	A string. The name of the file to which the individual least cost trees are written.

Example

```
poy chel.seq -replicates 100 -notbr -jackboot
  -jackfpseudotrees chel.jack-trees
```

Comments

1. `jackboot` (page 266) or `jackstart` (page 271) must be specified.

2. Output is in parenthetical format. Individual trees are enclosed in brackets.

jackfrequencies Writes the frequency of clades in a jackknife analysis to standard output.

Syntax

```
-jackfrequencies cladfreq
```

Arguments

Argument	Description
cladfreq	A string. A predefined code that specifies which clade frequencies are reported. `jackfrequencies` accepts the following values: **Value** — **Description** `all` — Report frequencies for all clades in the set of pseudo-replicate strict consensus trees `majority` — Report frequencies for all clades' jackboot majority rule consensus tree `off` — Do not report frequencies Default = `majority`

Example

```
poy chel.seq -replicates 100 -notbr -jackboot -
  jackfrequencies 80
```

jackftrees Writes the majority rule consensus tree derived from the strict consensus trees of all pseudo-replicates to a specified file.

Syntax

```
-jackftrees constree
```

Arguments

Argument	Description
constree	A string. The name of the file to which the majority rule consensus tree is written.

Example

```
poy chel.seq -replicates 100 -notbr -jackboot
  -jackftrees chel.jack
```

Comments

1. `jackboot` (page 266) or `jackstart` (page 271) must be specified.
2. Output is in parenthetical format followed by a text-based graphic.
3. The graphic is included as a comment so that the file can be used as an input file for `topofile` (page 322).
4. The clade frequencies in this tree are the jackknife support values.

jackoutgroup

Sets the outgroup for jackknifing.

Syntax

`-jackoutgroup` *`outname`*

Arguments

Argument	Description
`outname`	A string. The taxon name of the outgroup used in jackboot analysis. Default = First taxon in the first data file

Example

```
poy chel.seq -replicates 100 -notbr -jackboot -
  jackoutgroup Artemia
```

jackpseudoconsensustrees

Writes the strict consensus trees for the individual pseudo-replicates to standard output.

nojackpseudoconsensustrees

Turns off reporting of strict consensus trees.

Syntax

```
-jackpseudoconsensustrees
-nojackpseudoconsensustrees
```

Arguments

None

Example

```
poy chel.seq -replicates 100 -notbr -jackboot
  -jackpseudoconsensustrees
```

Comments

1. To take effect, the command line must include both `jackboot` (page 266) and `jackstart` (page 271).

2. In the output file, the number of trees is enclosed in brackets.

jackpseudotrees Writes the least cost trees for the individual pseudo-replicates to standard output.

nojackpseudotrees

Turns off reporting of pseudo-replicate least cost trees.

Syntax

```
-jackpseudotrees
-nojackpseudotrees
```

Arguments

None

Example

```
poy chel.seq -replicates 100 -notbr -jackboot
  -jackpseudotrees
```

Comments

In the output file, the cost of individual trees is enclosed in brackets.

jackstart Builds a set of initial trees using jackknifing, then performs the cladogram search as specified.

nojackstart

Default value. Performs jackknifing only.

Syntax

```
-jackstart
-nojackstart
```

Arguments

None

Example

```
poy chel.seq -buildsperreplicate 10 -notbr -jackboot
  -jackstart
```

Comments

To take effect, the command line must include `jackboot` (page 266).

jacktree Writes the majority rule consensus tree from the strict consensus trees of all jackknife pseudo-replicates to standard output.

nojacktree

Turns off jackknifing.

Syntax

```
-jacktree
-nojacktree
```

Arguments

None

Example

```
poy chel.seq -buildsperreplicate 10 -notbr -jackboot
  -jacktree
```

Comments

1. To take effect, the command line must include both `jackboot` (page 266) and `jackstart` (page 271).
2. In the output file, the clade frequencies are the jackknife support values.

jackwincladefile Writes the strict consensus trees from the individual pseudo-replicates to a specified file in Winclada readable format.

Syntax

```
-jackwincladefile wcfile
```

Arguments

Argument	Description
wcfile	A string. The name of the file to which the strict consensus trees are written in Winclada readable format. Default = none

Example

```
poy chel.seq -buildsperreplicate 10 -notbr -jackboot
  -jackwincladefile chel.jack-pha
```

Comments

To take effect, the command line must include both `jackboot` (page 266) and `jackstart` (page 271).

jobspernode Sets the number of processes spawned on each node.

Syntax

```
-jobspernode procs
```

Arguments

Argument	Description
procs	An integer. The number of processes spawned per node. Default = 1

Example

```
poy chel.seq -parallel -jobspernode 4
```

Comments

1. Typically, the value of *procs* is the number of processors per node.
2. Use `solospawn` (page 315) to set the number of jobs spawned on stand-alone multiprocessor machines.

kfactor

Sets the k value for implied character weighting.

Syntax

```
-kfactor k
```

Arguments

Argument	Description
k	An integer. The value of the k factor used for implied weighting. Default = 0

Example

```
poy chel.seq -goloboff ck -kfactor 10
```

Comments

Refer to `goloboff` (page 258).

leading

Counts leading and trailing gaps in tree cost.

noleading

Leading and trailing gaps are ignored in calculating tree cost.

Syntax

```
-leading
-noleading
```

Arguments

None

Example

```
poy chel.seq -noleading
```

Comments

1. When `noleading` is used, leading and trailing gaps are accounted for in initial HTU optimization to prevent trivial nonoverlapping alignments, but thereafter are ignored in calculating tree cost.
2. This command is often used with `trailinggap` (page 326).
3. Use of this command can cause metricity problems.

likelihood

Performs likelihood analysis.

nolikelihood

Turns off likelihood analysis.

Syntax

```
-likelihood
-nolikelihood
```

Arguments

None

Example

```
poy chel.seq -likelihood
```

Comments

1. Additive and nonadditive characters are treated as in Tuffley and Steele (1997).
2. Likelihoods are reported as the absolute value of their natural log.

likelihoodconvergencevalue

Sets the maximum likelihood difference for which two likelihood scores are considered equal.

Syntax

```
-likelihoodconvergencevalue diff
```

Arguments

Argument	Description
diff	An integer. The inverse of the similarity threshold. Default = 100

Example

```
poy chel.seq -likelihood -likelihoodconvergencevalue
  10
```

Comments

1. The similarity threshold is equal to

$$\frac{1}{diff} \qquad \text{(Eq 15.4)}$$

so that with *diff* = 100, likelihood score differences less than 0.01 are treated as equal.

2. As *diff* increases, calculation time increases.

likelihoodestimationsize Sets the number of pairwise sequence comparisons used to estimate likelihood parameters.

Syntax

```
-likelihoodestimationsize compare
```

Arguments

Argument	Description
compare	An integer. The number of pairwise comparisons. Default = parallel processing, all pairwise comparisons = sequential mode, the number of taxa

Example

```
poy chel.seq -likelihood -likelihoodestimationsize
  5000
```

likelihoodesttranseachtime Estimates the transition matrix for each comparison of two sequences.

nolikelihoodesttranseachtime

Default value. The transition matrix is not estimated for each comparison.

Syntax

```
-likelihoodesttranseachtime
-nolikelihoodesttranseachtime
```

Arguments

None

Example

```
poy chel.seq -likelihood -likelihoodesttranseachtime
```

likelihoodextensiongap Sets the likelihood of gaps that follow the first in a series of gaps.

Syntax

```
-likelihoodextensiongap gaps
```

Arguments

Argument	Description
gaps	An integer. The relative increase in likelihood of gaps that follow the first in a series of gaps. Default = none

Example

```
poy chel.seq -likelihood -likelihoodextensiongap 2
```

likelihoodmaxnumiterations Sets the maximum number of branch cost iterations performed.

Syntax

```
-likelihoodmaxnumiterations iterate
```

Arguments

Argument	Description
iterate	An integer. The maximum number of branch cost iterations performed. Default = 100

Example

```
poy chel.seq -likelihood -likelihoodmaxnumiterations
  25
```

likelihoodroundingmultiplier Sets the rounding value used in likelihood calculations.

Syntax

```
-likelihoodroundingmultiplier round
```

Arguments

Argument	Description
round	An integer. The inverse of the rounding value. Default = 100

Example

```
poy chel.seq -likelihood -
  likelihoodroundingmultiplier 50
```

Comments

1. In likelihood analysis, internal optimization alignments are performed using double precision floating point arithmetic. However, values returned as integers. *round* sets the rounding value used for this conversion using the formula

$$\frac{1}{round}. \qquad \text{(Eq 15.5)}$$

2. To increase precision, increase the value of *round*.

likelihoodstep Sets the size of iteration steps during branch cost optimization.

Syntax

```
-likelihoodstep steps
```

Arguments

Argument	Description
steps	An integer. The size of iteration steps. Default = 5

Example

```
poy chel.seq -likelihood -likelihoodstep 10
```

Comments

To decrease step size, increase the value of *steps*.

likelihoodtrailinggap Sets the likelihood of leading and trailing gaps.

Syntax

```
-likelihoodtrailinggap gaps
```

Arguments

Argument	Description
gaps	An integer. The relative increase in likelihood of leading and trailing gaps. Default = none

Example

```
poy chel.seq -likelihood -likelihoodtrailinggap 2
```

linear Treats chromosomal characters as linear, as opposed to circular.

nolinear

Default value. Chromosomal characters are treated as circular. Has the effect of `circular` (page 230).

Syntax

```
-linear
-nolinear
```

Arguments

None

Example

```
poy -chromosome -linear -rearrange -reversible
  chel.chrom
```

Comments

See `circular` (page 230).

list Displays a list of all commands that contain a specified string in their usage information.

Syntax

```
-list xref
```

Arguments

Argument	Description
xref	A string. Command usage information is searched for this string. Default = none

Example

```
poy -list gap
```

llist Displays a list with descriptions of all commands that contain a specified string in their usage information.

Syntax

```
-llist xref
```

Arguments

Argument	Description
xref	A string. Command usage information is searched for this string. Default = none

Example

```
poy -llist gap
```

locusgap Sets the constant fraction of the cost of locus origin-loss events.

Syntax

```
-locusgap cost
```

Arguments

Argument	Description
cost	An integer. The cost of the origin or loss of a locus irrespective of its length. Default = 10

Example

```
poy -chromosome -circular -rearrange -reversible
  chel.chrom -breakpoint 100 -locusgap 200
```

Comments

1. The value is constant for all origin-losses, unless `-locussizegap` is also used, in which case the cost of the origin-losses depends on the size of the locus it is placed against.
2. The total origin-loss cost is (locusgap + (locussizegap × locus length)).

locussizegap Sets the variable fraction of the cost of locus origin-loss events.

Syntax

```
-locussizegap cost
```

Arguments

Argument	Description
cost	An integer. The variable fraction of the cost of locus origin-loss events. Default = 0

Example

```
poy -chromosome -circular -rearrange -reversible
  chel.chrom -breakpoint 100 -locusgap 200
  -locussizegap 1
```

Comments

1. The value is a multiplier of the length of the locus that is lost or gained.
2. Can be used in concert with `locusgap` (page 279).
3 The total origin-loss cost is (locusgap + (locussizegap × locus length)).

locusswap

Refines chromosome HTU construction.

Syntax

```
-locusswap size
```

Arguments

Argument	Description
size	An integer. The size (number of adjacent loci) to remove and reinsert to improve the quality of HTU chromosome optimization. Default = 0

Example

```
poy -chromosome -circular -rearrange chel.chrom
  -n2reorder -locusswap 3
```

Comments

1. In the chromosome reordering step, perform a swapping step that is analogous to branch swapping. For every chunk size from 1 to less than or equal to *size*, take each consecutive chunk of loci and place it in each possible position. As soon as a better chromosome is found, move to the next chunk. Keep going through the loop of all chunk sizes until no change is made for any chunk of any size.

2. The default value is 0, which means this swapping step does not execute.
3. If *size* is greater than the length of the chromosome to swap minus 2 (since there are $n-1$ possible chunk sizes, and the $n-1^{st}$ will already have been tried with $n = 1$), the chunk size used for that chromosome is the length minus 2.
4. Use with `chromosome` (page 229), `rearrange` (page 310), and `n2reorder` (page 285).

maxiterations Sets the maximum number of iterative passes to perform.

Syntax

```
-maxiterations iterate
```

Arguments

Argument	Description
iterate	An integer. Command usage information is searched for this string. Default = unlimited

Example

```
poy -chel.seq -iterativepass -maxiterations 10
```

Comments

This command acts as a stopping rule when a stable solution is not found.

maxprocessors Sets the number of spawned jobs.

Syntax

```
-maxprocessors procs
```

Arguments

Argument	Description
procs	An integer. The number of spawned jobs. Default = number of nodes

Example

```
poy chel.seq -parallel -maxprocessors 20
```

Comments

The minimum value for *procs* is 2 (one master, one slave).

maxtrees Sets the maximum number of trees held in buffers during processing. See `buildmaxtrees` (page 222) and `sprmaxtrees` (page 317).

Syntax

`-maxtrees` *`trees`*

Arguments

Argument	Description
`trees`	An integer. The maximum number of trees held in buffers. Default = number available in memory

Example

```
poy chel.seq -maxtrees 100
```

Comments

The value of *`trees`* is used by all other `maxtrees` commands as their default value: `buildmaxtrees`, `sprmaxtrees`, `holdmaxtrees` (page 260), and `fitchtrees` (page 253).

minstop Sets the minimum number of random replicates that must be completed before processing ceases.

Syntax

`-minstop` *`minreps`*

Arguments

Argument	Description
`minreps`	An integer. The minimum number of replicates. Default = 0

Example

```
poy chel.seq -replicates 1000 -minstop 5 -stopat 3
```

Comments

In conjunction with `stopat` (page 318), this command performs the specified iterations regardless of the value set by `stopat`. Without this command, processing stops once the number of minimum cost trees specified by `stopat` are found.

minterminals Sets the limit for missing data when terminal taxa are specified.

Syntax

`-minterminals` *`pctmiss`*

Arguments

Argument	Description
pctmiss	An integer. The limit for missing data expressed as a percentage. Default = 80

Example

```
poy -terminalsfile chel.tax -minterminals 50 chel.seq
  chel.morph
```

Comments

Use this command to change the default value used by `terminalsfile` (page 320) to limit the amount of missing data in the analysis.

molecularmatrix Reads nucleic acid transformation costs from a specified file.

Syntax

```
-molecularmatrix matfile
```

Arguments

Argument	Description
matfile	A string. The name of the file containing the transformation cost matrix data. Default = none

Example

```
poy -molecularmatrix titv.mat chel.seq
```

Comments

1. Refer to "Multiple loci" on page 154.
2. Costs specified using this command take precedence over costs specified by `gap` (page 257) and `change` (page 227).
3. When `extensiongap` (page 251) is used, the gap row and column of the matrix are used for the cost of the first in a series of gaps. Subsequent gap costs are as specified by `extensiongap`.

multibuild Performs a specified number of random addition sequence builds on slave nodes.

Syntax

```
-multibuild builds
```

Arguments

Argument	Description
builds	An integer. The number of random addition sequence builds performed on slave nodes. Default = 0

multidrift Spawns individual tree drifting jobs to slave nodes.

nomultidrift

Default value. Individual tree drifting jobs are not spawned to slave nodes.

Syntax

```
-multidrift
-nomultidrift
```

Arguments

None

Example

```
poy chel.seq -parallel -drifttbr -numdrifttbr 10
  -multidrift
```

Comments

This command modifies the behavior of `driftspr` (page 248) and `drifttbr` (page 248).

multiplier Sets the multiplier for implied weighting.

Syntax

```
-multiplier weight
```

Arguments

Argument	Description
weight	An integer. The multiplier used in implied weighting. Default = 100

Example

```
poy chel.seq chel.morph -goloboff ck -multiplier 100
```

Comments

Used with `goloboff` (page 258).

multirandom Spawns individual random replicates to slave nodes.

nomultirandom

Default value. Individual random replicates are not spawned to slave nodes.

Syntax

```
-multirandom
-nomultirandom
```

Arguments

None

Example

```
poy chel.seq -replicates 25 -multirandom -parallel
```

Comments

1. This command utilizes the parallel environment for full searches by sending random replicates to each node or sub-cluster.
2. This command is affected by `controllers` (page 234).

multiratchet Spawns individual ratchet jobs on slave nodes.

nomultiratchet

Default value. Individual ratchet jobs are not spawned to slave nodes.

Syntax

```
-multiratchet
-nomultiratchet
```

Arguments

None

Example

```
poy chel.seq chel.morph -parallel -ratchettbr 100
  -multiratchet
```

n2reorder Selects reorderings of loci on a chromosome using a method akin to cladogram building.

non2reorder

Default value. Reordering selection method not used.

Syntax

```
-n2reorder
-non2reorder
```

Arguments

None

Example

```
poy -chromosome -circular -rearrange -reversible
  chel.chrom -n2reorder
```

Comments

The HTU chromosome is built one locus at a time, each time choosing the locus to add that gives the best total optimization cost (that is, alignment cost + breakpoint cost) at that point.

newstates Reads the state set for search-based optimization from a specified file.

Syntax

```
-newstates newfile
```

Arguments

Argument	Description
newfile	A string. The name of the file containing hypothetical ancestral states. Default = none

Example

```
poy -fixedstates chel.seq -newstates chel.states
```

Comments

1. The format for *newfile* is described in "Hypothetical ancestral states" on page 163.
2. Each sequence input file (modified by `fixedstates` on page 253) must have a newstates command associated with it.

nopoyini Causes POY to not recognize the file `poy.ini` in the current directory as a command file.

Syntax

```
-nopoyini
```

Arguments

None

Example

```
poy chel.seq -nopoyini -commandfile poy.commands
```

Comments

1. If this command is in `poy.ini` or a command file reached from `poy.ini`, the contradiction is arbitrarily resolved by POY taking `nonpoyini` as the single command in `poy.ini`.
2. This command is overruled by the command `-commandfile poy.ini`.

numdriftchanges Sets the number of topological changes permitted per tree drifting round.

Syntax

```
-numdriftchanges changes
```

Arguments

Argument	Description
changes	An integer. The number of changes permitted per tree drift round. Default = 20

Example

```
poy chel.seq -drifttbr -numdrifttbr 10
  -numdriftchanges 30
```

numdriftspr Sets the number of SPR tree drift rounds.

Syntax

```
-numdriftspr rounds
```

Arguments

Argument	Description
rounds	An integer. The number of SPR tree drift rounds. Default = 1

Example

```
poy chel.seq -driftspr -numdriftspr 10
```

numdrifttbr Sets the number of TBR tree drift rounds.

Syntax

```
-numdrifttbr rounds
```

Arguments

Argument	Description
rounds	An integer. The number of TBR tree drift rounds. Default = 1

Example

```
poy chel.seq -drifttbr -numdrifttbr 10
```

numslaveprocesses Sets the maximum number of regular slave jobs that can be spawned on PVM nodes, excluding the master node.

Syntax

```
-numslaveprocesses procs
```

Arguments

Argument	Description
procs	An integer. The maximum number of reserve jobs spawned on PVM nodes. Default = number of entries in PVM configuration - 1 × the value *jobsnum* set by `jobspernode` (page 272)

Example

```
poy chel.seq -parallel -numslaveprocesses 100
```

onan Spawns jobs on the master as well as on slave nodes.

noonan

Default value. Spawns jobs on slave nodes only.

Syntax

```
-onan
-noonan
```

Arguments

None

Example

```
poy chel.seq -parallel -onan
```

onannum Sets the number of jobs spawned to the master node.

Syntax

```
-onannum jobs
```

Arguments

Argument	Description
jobs	An integer. The number of jobs spawned on the master node. Default = 0

Example

```
poy chel.seq -parallel -onan -onannum 2
```

oneasis Reads the random addition sequence order of taxa for the first replicate from the order of taxa in the first data file.

nooneasis

Default value. All random additions randomized.

Syntax

```
-oneasis
-nooneasis
```

Arguments

None

Example

```
poy chel.seq -replicates 100 -oneasis
```

onechroms Optimizes HTU chromosomes by examining a single resolution of loci that have been ambiguously assigned to the candidate HTU. Default value.

noonechroms

Examines more than a single resolution.

Syntax

```
-onechroms
-noonechroms
```

Arguments

None

Example

```
poy -chromosome -circular -rearrange chel.chrom
  -iterativepass -onechroms
```

Comments

1. Only operative when `iterativepass` (page 265) is used.
2. Not relevant if `fixedstates` (page 253) is used for chromosomal optimization.

onversionconflict Sets the action POY takes when it detects a difference version numbers or dates between the master and slave tasks.

Syntax

```
-onversionconflict action
```

Arguments

Argument	Description
action	A string. A predefined code that specifies POY detects task version difference between the master and slave. `onversionconflict` accepts the following values: **Value** — **Description** `deletehost` — Removes the host of the slave task from the pool of hosts used `exit` — Immediately exits the program `warn` — A warning is written to the standard error file Default = `deletehost`

Example

```
poy chel.seq -onversionconflict exit
```

Comments

1. POY reads versions from the standard error file, where version numbers and version dates are reported when tasks start.
2. In a PVM environment with a single host, `deletehost` will have the same effect as `exit`.
3. When `exit` is specified, adding nodes to the PVM configuration while POY is running might cause POY to abort immediately.

outgroup Sets the outgroup.

Syntax

```
-outgroup outtaxon
```

Arguments

Argument	Description
outtaxon	A string. The name of the taxon used as the outgroup. Default = none

Example

```
poy chel.seq -outgroup Artemia
```

Comments

1. Used in conjunction with `nooneasis oneasis` (page 289), `randomizeoutgroup` (page 306), and `rerootafterbuild` (page 312).
2. `plotoutgroup` (page 296) and `jackoutgroup` (page 270) will use *outtaxon* as their default value.
3. When `outgroup` is not used, the outgroup taxon is set using one of the following methods.
   - If `terminalsfile` (page 320) is used, the outgroup is the first taxon in the specified terminals file.
   - If `terminalsfile` is not used, the outgroup is the first taxon in the first input data file encountered on the command line.

overwritedataprotection

Prevents POY from overwriting input data files.

Syntax

```
-overwritedataprotection protect
```

Arguments

Argument	Description
protect	A string. A predefined code that specifies whether input data files can be overwritten: `overwritedataprotection` accepts the following values: **Value** — **Description** `on` — Do not overwrite input data files `off` — Overwrite input data files Default =on

Example

```
poy chel.seq -overwritedataprotection off -topofile
  chel.tree -phastwincladfile chel.seq
```

Comments

1. When protection is enabled, POY performs the file name check before any calculations are performed. If conflicts are found, POY writes a notification to the standard error file and exits.
2. The input file is read before it is overwritten.
3. In conjunction with `iafiles` (page 261), only the trees file `ia.trees` is protected.

overwriteprotection Prevents POY from overwriting input data files with output files.

Syntax

```
-overwriteprotection protect
```

Arguments

Argument	Description
protect	A string. A predefined code that specifies whether input data files can be overwritten by output files. `overwriteprotection` accepts the following values: **Value** — **Description** `on` — Do not overwrite input data files `off` — Overwrite input data files Default = `off`

Example

```
poy chel.seq -overwriteprotection off -topofile
  chel.tree > chel.seq
```

Comments

When protection is enabled, POY performs the file name check before any calculations are performed. If conflicts are found, POY writes a notification to the standard error file and exits.

pairmatrix Generates a pairwise distance matrix between taxa.

nopairmatrix

Default value. Pairwise matrix not generated.

Syntax

```
-pairmatrix
-nopairmatrix
```

Arguments

None

Example

```
poy chel.seq -pairmatrix
```

parallel

Executes POY as a parallel process using PVM.

noparallel

Default value. POY executes sequentially.

Syntax

```
-parallel
-noparallel
```

Arguments

None

Example

```
poy chel.seq -parallel
```

Comments

The POY code for parallel processing is fault tolerant: the master task detects slave tasks that die or become unreachable, then makes appropriate adjustments. Partial results from slave node failures are recalculated on other slaves.

phastwincladfile

Writes implied alignments to a specified file.

Syntax

```
-phastwincladfile phastfil
```

Arguments

Argument	Description
phastfil	A string. The name of the file to which output is written. Default = none

Example

```
poy chel.seq -phastwincladfile chel.pha
```

Comments

The output is in Hennig86/Nona format and can be read by Phast, Winclada, and POY. Refer to "Phastwincladfile" on page 163.

plotencoding

Sets the characters used to plot trees.

Syntax

```
-plotencoding charactr
```

Arguments

Argument	Description
charactr	A string. A predefined code that specifies the characters used by `printtree` (page 304) to plot trees. `plotencoding` accepts the following values: **Value** — **Description** `ASCII` — Use ASCII character set `UTF-8` — Use UTF-8 encoded unicode character set Default = `ASCII`

Example

```
poy chel.seq -printtree -plotfile chel.tree
  -plotencoding UTF-8
```

Comments

UTF-8 produces aesthetically better plots, but requires that you have unicode fonts and a viewer that understands UTF-8 character encoding. This can be done, for example, using Microsoft Word by importing the file as UTF-8 encoded text or using vim with `:set encoding=utf-8` in a supporting terminal.

plotechocommandline Writes the command line as the first line of the tree output file.

Syntax

```
-plotechocommandline
```

Arguments

None

Example

```
poy chel.seq -plotechocommandline
```

plotfile Sets the name of the tree output file.

Syntax

```
-plotfile plotname
```

Arguments

Argument	Description
plotname	A string. The name of the file to which output is written. With a value of `stdout`, output is directed to standard output. Default = `poy.tree`

Example

```
poy chel.seq -printtree -plotfile chel.tree
```

Comments

If the file already exists, output is appended to it.

plotfrequencies Sets how clade frequencies are tabulated in the least cost trees.

Syntax

```
-plotfrequencies cladfreq
```

Arguments

Argument	Description
cladfreq	A string. A predefined code that specifies how to report clade frequencies for `printtree` (page 304). `plotfrequencies` accepts the following values: **Value** — **Description** `all` — Tabulate frequencies for all clades (absolute and percentage-wise number of occurrences) `majority` — Tabulate majority clade frequencies (absolute and percentage-wise number of occurrences) `off` — Do not tabulate frequencies Default = `off`

Example

```
poy chel.seq -jackboot -replicates 100
  -plotfrequencies all
```

Comments

Use in conjunction with `printtree` (page 304).

plotmajority

Sets how to plot majority rule consensus trees.

Syntax

```
-plotmajority plot
```

Arguments

Argument	Description
plot	A string. A predefined code that specifies how to plot the majority rule consensus tree for `printtree` (page 304). `plotmajority` accepts the following values: **Value** — **Description** `long` — Plot the majority rule consensus tree with clade identification numbers. The output includes the identification number and percentage-wise clade frequency. `majority` — Plot the majority rule consensus tree without clade identification numbers `off` — Do not plot the majority rule consensus tree Default = `off`

Example

```
poy chel.seq -printtree -plotfile chel.tree
  -plotmajority long
```

plotoutgroup

Sets the outgroup used for tree plotting.

Syntax

```
-plotoutgroup outtaxon
```

Arguments

Argument	Description
outtaxon	A string. The name of the taxon used as the outgroup by `printtree` (page 304). Default = see Comments below

Example

```
poy chel.seq -printtree -plotfile chel.tree
  -plotoutgroup Artemia
```

Comments

1. Used in conjunction with `printtree` (page 304).
2. The default value for *`outtaxon`* is the outgroup value set by `outgroup` (page 290).
3. When `outgroup` is not used, the default value for *`outtaxon`* is either
   - the first taxon in the first input data file encountered on the command line
   - or, when there is not input data file, the first taxon in the first tree input data set (see `topofile` on page 322 and `topology` on page 323).

plotstrict

Sets how to plot strict consensus trees.

Syntax

```
-plotstrict plot
```

Arguments

Argument	Description
`plot`	A string. A predefined code that specifies how to plot strict consensus trees for `printtree` (page 304). `plotstrict` accepts the following values: **Value** — **Description** `long` — Plot the strict consensus trees with clade identification numbers. The output includes the identification number and percentage-wise clade frequency. `majority` — Plot the strict consensus trees without clade identification numbers `off` — Do not plot the strict consensus trees Default = `long`

Example

```
poy chel.seq -printtree -plotfile chel.tree
  -plotstrict long
```

Comments

Used in conjunction with `printtree` (page 304).

plottrees

Sets how to plot optimal trees.

Syntax

```
-plottrees plot
```

Arguments

Argument	Description
plot	A string. A predefined code that specifies how to plot majority rule consensus trees for `printtree` (page 304). `plottrees` accepts the following values: **Value** — **Description** `long` — Plot the optimal trees with clade identification numbers. The output includes the identification number and percentage-wise clade frequency. `majority` — Plot the optimal trees without clade identification numbers `off` — Do not plot the optimal trees Default = `long`

Example

```
poy chel.seq -printtree -plotfile chel.tree
  -plottrees long
```

Comments

Used in conjunction with `printtree` (page 304).

plotwidth

Sets the number of columns used for plotting trees.

Syntax

```
-plotwidth columns
```

Arguments

Argument	Description
columns	An integer. The number of columns used for `printtree` (page 304) output. Default = 80 Minimum = 25

Example

```
poy chel.seq -printtree -plotfile chel.tree
  -plotwidth 120
```

Comments

1. Used in conjunction with `printtree` (page 304).
2. If a tree plot requires more columns than are specified, the plot is broken into subtrees which fit the width as specified.
3. Long terminal names (greater than *columns* - 5 characters) are truncated.

polyaddconverttorange Converts polymorphisms in additive characters optimization to the smallest range of values that span the original values.

nopolyaddconverttorange

The polymorphic character is treated as nonadditive.

Syntax

```
-polyaddconverttorange
-nopolyaddconverttorange
```

Arguments

None

Example

```
poy chel.seq chel.morph -nopolyaddconverttorange
```

Comments

1. Use this command when polymorphisms in additive characters optimization are not contiguous.
2. The polymorphism is converted into the smallest range that includes the original polymorphism. For example, [3578] converts to [345678].
3. If `phastwincladfile` (page 293) is used in conjunction with `nopolyaddconverttorange`, the output file includes the conversion of polymorphisms to nonadditive characters.

poybintreefile Writes the optimal trees in POY topology format to a specified file.

Syntax

```
-poybintreefile treefile
```

Arguments

Argument	Description
treefile	A string. The name of the file to which output is written. Default = none

Example

```
poy chel.seq -poybintreefile chel.bin
```

Comments

1. Refer to "POY topology" on page 161.
2. Polytomies are resolved and siblings are ordered.
3. In the output file, siblings are ordered left to right.
4. When no input character data are used, all input cladogram data are used.
5. Use this command to create output used for diagnostics.
6. To create a file for use in a tree drawing program, use `poytreefile` (page 301).

poystrictconsensuscharfile Writes the strict consensus tree in Hennig86/Nona format to a specified file.

Syntax

```
-poystrictconsensuscharfile treefile
```

Arguments

Argument	Description
treefile	A string. The name of the file to which output is written. Default = none

Example

```
poy chel.seq -poystrictconsensuscharfile chel.con
```

Comments

1. The file can be used as a constraint file (see "Constraint file" on page 157).

2. When no input character data are used, all input cladogram data are used.

poystrictconsensustreefile Writes the strict consensus tree in POY topology format to a specified file.

Syntax

```
-poystrictconsensustreefile treefile
```

Arguments

Argument	Description
treefile	A string. The name of the file to which output is written. Default = none

Example

```
poy chel.seq -poystrictconsensustreefile chel.strict
```

Comments

1. Refer to "POY topology" on page 161.
2. The file can be used as a constraint file (see "Constraint file" on page 157).
3. When no input character data are used, all input cladogram data are used.

poytreefile Writes the optimal cladograms in POY topology format to a specified file.

Syntax

```
-poytreefile treefile
```

Arguments

Argument	Description
treefile	A string. The name of the file to which output is written. Default = none

Example

```
poy chel.seq -poytreefile chel.poy
```

Comments

1. Refer to "POY topology" on page 161.
2. Use this command to create an input file for cladogram drawing applications. This file will display as read (that is, the

cladogram drawing application does not change the resolution of the cladograms based on an optimality criterion).

3. When no input character data are used, all input cladogram data are used.

prealigned Reads sequence data as prealigned from a specified file.

noprealigned

Reads sequence data as unaligned.

Syntax

```
-prealigned seqfile
-noprealigned seqfile
```

Arguments

Argument	Description
seqfile	A string. The name of the file from which data are read. Default = none

Example

```
poy chel.seq -prealigned chel.pro
```

Comments

1. For `prealigned`, sequences cannot contain gaps.
2. For `prealigned`, sequences must be of equal length.
3. Use `noprealigned` for sequences of unequal length.

printccode Writes a ccode-like summary of the character settings in each Hennig86/Nona formatted input data file to standard output.

noprintccode

Turns off writing of ccode-like summary.

Syntax

```
-printccode
-noprintccode
```

Arguments

None

Example

```
poy chel.morph -printccode
```

Comments

1. Use this command when polymorphisms in additive characters optimization are not contiguous.
2. The polymorphism is converted into the smallest range that includes the original polymorphism. For example, [3578] converts to [345678].
3. If `phastwincladfile` (page 293) is used in conjunction with `nopolyaddconverttorange`, the output file includes the conversion of polymorphisms to nonadditive characters.

printhypanc Writes hypothetical ancestral sequences for optimal cladograms to a specified file.

noprinthypanc

Turns off writing hypothetical ancestral sequences.

Syntax

```
-printhypanc
-noprinthypanc
```

Arguments

None

Example

```
poy chel.seq -printhypanc -hypancfile chel.hypanc
```

Comments

1 Refer to "Hypothetical ancestral states" on page 163.

2 `hypancfile` (page 260) specifies the file to which output is written.

3 The output consists of the hypothetical ancestral sequences of the optimal tree with ambiguities resolved.

4 The output is used to prepare the `newstates` (page 286) file for search-based optimization.

printlotshypanc Writes hypothetical ancestral sequences for intermediate and optimal trees to a specified file.

noprintlotshypanc

Turns off writing hypothetical ancestral sequences.

Syntax

```
-printlotshypanc
-noprintlotshypanc
```

Arguments

None

Example

```
poy chel.seq -printhypanc -hypancfile chel.hypanc
  -printlotshypanc
```

Comments

1. Modifies and must be used with `printhypanc` (page 303).
2. Refer to "Hypothetical ancestral states" on page 163.
3. `hypancfile` (page 260) specifies the file to which output is written.
4. The output consists of the hypothetical ancestral sequences of the optimal tree with ambiguities resolved.
5. The output is used to prepare the `newstates` (page 286) file for search-based optimization.

printqmat

Writes likelihood parameters to standard output.

noprintqmat

Turns off writing likelihood parameters.

Syntax

```
-printqmat
-noprintqmat
```

Arguments

None

Example

```
poy chel.seq -likelihood -printqmat
```


printtree

Writes a graphic of the optimal trees and their strict consensus trees to a standard output file.

noprinttree

Turns off writing a graphic of the optimal trees and their strict consensus trees.

Syntax

```
-printtree
-noprinttree
```

Arguments

None

Example

```
poy chel.seq -printtree -plotfile chel.tree
```

Comments

1. Output is written to `poy.tree`.
2. If `poy.tree` exists, output is appended to the end of the file.
3. Graphics can be written from topology data (see `topofile` on page 322 and `topology` on page 323).
   - If the command line includes character input data, the topology data are diagnosed using the character data. Costless branches are collapsed. However, note that input topologies should not have polytomies because POY resolves these arbitrarily, which can affect diagnosis and the occurrence of costless branches.
   - If the command line does not include character input data, the output has the same resolution as the topology data.
4. Cladograms are treated as unrooted, so that the base node is never resolved.

qmatrix Reads a transition cost ratio matrix for likelihood analysis from a specified file.

Syntax

```
-qmatrix qmatfile
```

Arguments

Argument	Description
qmatfile	A string. The name of the file from which data are read. Default = none

Example

```
poy chel.seq -likelihood -qmatrix chel.mat
```

Comments

Refer to "Likelihood cost matrix file" on page 160.

quick Performs branch swapping only on minimum cost trees.

noquick

Default value. Performs branch swapping on all trees found during the search.

Syntax

```
-quick
-noquick
```

Arguments

None

Example

```
poy chel.seq -noquick
```

Comments

This command accelerates the cladogram search process.

quote Writes a specified string to standard output.

Syntax

```
-quote string
```

Arguments

Argument	Description
string	A string. Any character string. Default = none

Example

```
poy chel.seq -quote "please help me"
```

Comments

Use this command to include a comment line in the standard output file.

randomizeoutgroup Randomizes the addition of all taxa, including the outgroup.

norandomizeoutgroup

Default value. Sets the first taxon of the first input data set as the starting point for adding terminals.

Syntax

```
-randomizeoutgroup
-norandomizeoutgroup
```

Arguments

None

Example

```
poy -replicates 10 -randomizeoutgroup
```

Comments

1. When using constraints, use `norandomizeoutgroup`.

2. When using `randomizeoutgroup` with `jackboot` (page 266) and `jackstart` (page 271), also use `jackoutgroup` (page 270).
3. When using `randomizeoutgroup` with `printtree` (page 304) to output consensus cladograms, also use `plotoutgroup` (page 296).

randomizeslaves Sets PVM task identification numbers (TIDs) randomly.

norandomizeslaves

Default value. TIDs are not set randomly.

Syntax

```
-randomizeslaves
-norandomizeslaves
```

Arguments

None

Example

```
poy chel.seq -parallel -norandomizeslaves
```

Comments

This command causes jobs to be distributed at random among slave nodes.

ratchetoverpercent Spawns a specified number of extra ratchet jobs to accommodate unequal execution times during parallel execution.

Syntax

```
-ratchetoverpercent extra
```

Arguments

Argument	Description
extra	An integer. The number of extra ratchet jobs to spawn. Default = none

Example

```
poy chel.seq chel.morph -parallel -ratchettbr 100
  -ratchetoverpercent 50
```

Comments

Once the full complement of ratchet jobs is complete, unfinished jobs are terminated and the search continues as specified.

ratchetpercent Sets the percentage of characters reweighted during a ratchet search.

Syntax

```
-ratchetpercent reweight
```

Arguments

Argument	Description
reweight	An integer. The percentage of characters to reweigh. Default = 15

Example

```
poy chel.seq chel.morph -ratchettbr 100
  -ratchetpercent 20
```

ratchetseverity Sets the weight multiplier for reweighting characters during a ratchet search.

Syntax

```
-ratchetseverity multiply
```

Arguments

Argument	Description
multiply	An integer. The multiplier used to reweight characters during a ratchet search. Default = 2

Example

```
poy chel.seq chel.morph -ratchettbr 100
  -ratchetseverity 3
```

ratchetslop Sets the percent limit of suboptimal trees evaluated during a ratchet search.

Syntax

```
-ratchetslop limit
```

Arguments

Argument	Description
limit	An integer. The limit of suboptimal trees evaluated expressed in tenths of a percent (that is, a value of 5 represents a 0.5% limit). Default = 0 or the value set by `slop` (page 314)

Example

```
poy chel.seq chel.morph -ratchettbr 100 -slop 5
  -ratchetslop 0
```

Comments

While this command slows processing, it abates the effects of shortcuts used in calculating tree costs.

ratchetspr Sets the number of ratchet iterations using SPR.

Syntax

```
-ratchetspr iterate
```

Arguments

Argument	Description
iterate	An integer. The number of iterative rounds of ratcheting performed using SPR. Default = 0

Example

```
poy chel.seq chel.morph -ratchetspr 100
```

ratchettbr Sets the number of ratchet iterations using TBR.

Syntax

```
-ratchettbr iterate
```

Arguments

Argument	Description
iterate	An integer. The number of iterative rounds of ratcheting performed using TBR. Default = 0

Example

```
poy chel.seq chel.morph -ratchettbr 100
```

ratchettrees Sets the number of trees saved during ratchet iterations.

Syntax

```
-ratchettrees treenum
```

Arguments

Argument	Description
treenum	An integer. The number of trees saved. Default = 2

Example

```
poy chel.seq chel.morph -ratchettbr 100 -ratchettrees 1
```

rearrange Enables locus rearrangement to occur when optimizing chromosomal characters.

norearrange

Turns off locus rearrangement.

Syntax

```
-rearrange
-norearrange
```

Arguments

None

Example

```
poy -chromosome -linear -rearrange chel.chrom
  -nochromosome chel.morph
```

Comments

1. In "chromosome alignment," try different orderings of the loci to optimize the reordering of loci on the same chromosome. Change the order of the loci and then align them, adding the breakpoint cost to the cost of aligning the loci (the induced set of breakpoints; Sankoff and Blanchette 1998).
2. To calculate the breakpoint cost, only loci that appear in both of the chromosomes are considered. The breakpoint cost is the number of adjacent pairs of these loci that appear in the reduced version of one of the chromosomes but not in the reduced version of the other chromosome.
3. The loci are considered to be the homologous if they are aligned against each other. If just `-rearrange` is used, a simplistic algorithm for choosing reorderings is used. If

`n2reorder` (page 285) is also used, then an algorithm akin to cladogram building is used to try to find better reorderings.

recode Accelerates additive and nonadditive characters optimization. Default value.

norecode

Turns off acceleration of additive and nonadditive character optimization.

Syntax

```
-recode
-norecode
```

Arguments

None

Example

```
poy chel.morph -norecode
```

Comments

Recode improves the speed of additive and nonadditive character optimization.

repintermediate Writes the results of individual random replicates to standard output.

norepintermediate

Default value. Results of individual random replicates are not written to standard output.

Syntax

```
-repintermediate
-norepintermediate
```

Arguments

None

Example

```
poy chel.seq -replicates 10 -repintermediate
```

Comments

Use this command in conjunction with `replicates`.

replicates Sets the number of random addition sequence searches or the number of jackknife pseudo-replicates.

Syntax

`-replicates` *`repnum`*

Arguments

Argument	Description
repnum	An integer. The number of random addition sequence searches or pseudo-replicates to perform. Default = none

Example

```
poy chel.seq -replicates 100
```

Comments

1. In conjunction with `buildsperreplicate` (page 223), sets the number of random addition sequence searches.
2. In conjunction with `jackboot` (page 266), sets the number of pseudo-replicates.

rerootafterbuild Performs stepwise addition by rerooting trees to a specified outgroup before performing cladogram search procedures.

norerootafterbuild

Default value. Trees are not rerooted to a specified outgroup before cladogram search.

Syntax

```
-rerootafterbuild
-norerootafterbuild
```

Arguments

None

Example

```
poy chel.seq -rerootafterbuild -outgroup Artemia
  -buildsperreplicate 10 -randomizeoutgroup
```

Comments

1. Used in conjunction with `randomizeoutgroup` (page 306).
2. The outgroup to which the candidate tree is rerooted is set by `outgroup` (page 290).
3. Using this command can change the cladogram cost reported at the conclusion of the stepwise addition because the reconstructed ancestral states depend on the chosen outgroup.

reversible Enables the loci to be treated as unsigned—that is, either the given locus or its reverse complement can be used in optimization.

noreversible

Default value. Loci are treated as signed.

Syntax

```
-reversible
-noreversible
```

Arguments

None

Example

```
poy -chromosome -linear -rearrange -reversible
  chel.chrom
```

Comments

Both orderings are tried when computing the cost matrix for loci. The orientation is recorded.

seed Sets the seed value for the generation of random numbers.

Syntax

`-seed` *num*

Arguments

Argument	Description
num	An integer. The seed value for pseudo-random number generation. Default = – 1 The default value – 1 uses the system time in seconds.

Example

```
poy chel.seq -replicates 100 -seed 105
```

Comments

1. Values of 0 or greater guarantee exact reproducibility under `noparallel` (page 293).
2. Values of 0 or greater do not guarantee exact reproducibility under `parallel` (page 293) because the system cannot know in advance when requests to slave nodes will be answered (a result of slave node processor speed and load). Disparate results are not a defect in POY.

showchromsearch Display of chromosomal HTU optimization progress information.

noshowchromsearch

Default value. Chromosomal HTU optimization progress information not displayed.

Syntax

```
-showchromsearch
-noshowchromsearch
```

Arguments

None

Example

```
poy -chromosome -circular -rearrange chel.chrom
  -showchromsearch
```

Comments

The output can be prolix.

showiterative Writes iterative pass progress information to standard output.

noshowiterative

Default value. Iterative pass progress information is not output.

Syntax

```
-showiterative
-noshowiterative
```

Arguments

None

Example

```
poy chel.seq -iterativepass -showiterative
```


slop Sets the percent limit of suboptimal trees evaluated during a search.

Syntax

```
-slop limit
```

Arguments

Argument	Description
limit	An integer. The limit of suboptimal trees evaluated expresses in tenths of a percent (that is, a value of 5 represents a 0.5% limit). Default = 0

Example

```
poy chel.seq -slop 5
```

Comments

1. While this command slows processing, it abates the effects of shortcuts used in calculating tree costs.
2. This command sets the default for all other slop commands (`buildslop` on page 222, `checkslop` on page 228, and `ratchetslop` on page 308).

solospawn Sets the number of slave jobs spawned on a standalone multiprocessor machine.

Syntax

```
-solospawn jobs
```

Arguments

Argument	Description
jobs	An integer. The number of jobs spawned. Default = 4

Example

```
poy chel.seq -parallel -solospawn 20
```

Comments

Use `jobspernode` (page 272) to set the number of jobs spawned on slave nodes in parallel clusters.

somechroms Optimizes HTU chromosomes by examining all combinations of loci that have been ambiguously assigned to the candidate HTU.

nosomechroms

Default value.

Syntax

```
-somechroms
```

```
-nosomechroms
```

Arguments

None

Example

```
poy -chromosome -circular -rearrange chel.chrom
  -iterativepass -somechroms
```

Comments

1. Only operative when `iterativepass` (page 265) is used.
2. Not relevant if `fixedstates` (page 253) is used for chromosomal optimization.
3. Can be very time-consuming.

spewbinary Writes binary trees (that is, trees without polytomies) used internally by POY to standard output. Default value.

nospewbinary

Turns off output of binary trees used internally by POY.

Syntax

```
-spewbinary
-nospewbinary
```

Arguments

None

Example

```
poy chel.seq -nospewbinary
```

Comments

Use the output to compare POY results with the output of other applications or to use as topology input to POY.

spr Performs an SPR cladogram search.

nospr

Turns off SPR cladogram search.

Syntax

```
-spr
-nospr
```

Arguments

None

Example

```
poy chel.seq -notbr -spr
```

sprmaxtrees Sets the maximum number of trees held in buffers during an SPR cladogram search.

Syntax

```
-sprmaxtrees max
```

Arguments

Argument	Description
max	An integer. The maximum number of trees held in buffers during SPR cladogram search. Default = the value set by `maxtrees` (page 282)

Example

```
poy.seq chel.seq -spr -sprmaxtrees 1
```

staticapprox Performs static approximation for sequence optimization during cladogram refinements.

nostaticapprox

Turns off static approximation.

Syntax

```
-staticapprox
-nostaticapprox
```

Arguments

None

Example

```
poy chel.seq -staticapprox
```

Comments

Static approximation uses fixed implied alignment during branch swapping. Each time a less costly tree is found, a new implied alignment is calculated. This new implied alignment is then used for subsequent swapping.

staticapproxbuild Performs static approximation for sequence optimization during cladogram building.

nostaticapproxbuild

Turns off static approximation.

Syntax

```
-staticapproxbuild
-nostaticapproxbuild
```

Arguments

None

Example

```
poy chel.seq -staticapproxbuild
```

Comments

Use static approximation (Wheeler 2003a) for sequence optimization while building cladograms. Static approximation consists of using a fixed implied alignment while adding taxa; each time a shorter cladogram is found, a new implied alignment is calculated (based on that shorter tree) and this new alignment is used for further additions.

stats Writes cladogram search statistics to standard output.

nostats

Turns off writing cladogram search statistics.

Syntax

```
-spr
-nospr
```

Arguments

None

Example

```
poy chel.seq -stats
```

stopat Sets the minimum number of minimum cost replicates that must be found before the search stops.

Syntax

```
-stopat minnum
```

Arguments

Argument	Description
minnum	An integer. The minimum number of minimum cost replicates that must be found before the search ends. Default = the value set by `replicates` (page 311) or `multirandom` (page 285)

Example

```
poy chel.seq -replicates 100 -stopat 5
```

Comments

Use in conjunction with `minstop` (page 282).

submodel Enforces symmetries in transition probabilities during likelihood analysis.

Syntax

```
-submodel syms
```

Arguments

Argument	Description
syms	A string. A predefined code that specifies how symmetries are determined. `submodel` accepts the following values: **Value** — **Description** `s10` — Each of the 10 reversible transitions are allowed to vary (a super-GTR) `s6g` — GTR+gaps with all indels treated equally `s3g` — Purine transitions, pyrimidine transitions, transversions, and indels have different probabilities `s2g` — Transitions, transversions, and indels have different probabilities `s1g` — All base substitutions have equal probability, indels are different `s1` — All transitions have equal probability Default = `s6g`

Example

```
poy chel.seq -likelihood -submodel s10
```

tbr Performs a TBR cladogram search.

notbr

Turns off TBR cladogram search.

Syntax

```
-tbr
-notbr
```

Arguments

None

Example

```
poy chel.seq -spr -notbr
```

tbrmaxtrees Sets the maximum number of trees held in buffers during a TBR cladogram search.

Syntax

```
-tbrmaxtrees max
```

Arguments

Argument	Description
max	An integer. The maximum number of trees held in buffers during TBR cladogram search. Default = the value set by `maxtrees` (page 282)

Example

```
poy chel.seq -tbrmaxtrees 1 -checkslop 10
```

terminalsfile Sets terminal taxon names.

Syntax

```
-terminalsfile termfile
```

Arguments

Argument	Description
termfile	A string. The name of the file containing terminal taxon names. One or more terminal files can be specified. Default = none

Example

```
poy -terminalsfile chel.tax chel.seq chel.morph
  -minterminals 50
```

Comments

1. Refer to "Terminals file" on page 158.
2. If this command is not used, inconsistencies in taxon names among input files will cause POY to terminate.
3. When an input data file includes a name that is not in the terminals file, POY skips the input data for that terminal.
4. When an input data file does not have data corresponding to a taxon in the terminals file, POY includes the taxon in the analysis and treats it as having missing data.
5. Inconsistencies between an input data files and the terminals file are written to the standard error file.
6. This command makes it possible to

- include a taxon that has missing data in an analysis without including it in a data file
- exclude a taxon from an analysis without editing data files.

7. POY compares the terminals file and the data input files to avoid the inclusion of too much missing data.
   - If the number of taxa added from the terminal file is greater than 20% of the taxa in the input data files, then POY will terminate.
   - To change this percentage, use `minterminals` (page 282).
8. When an input cladogram specified by `topofile` (page 322) or `topology` (page 323) includes a name that is not in the terminals file, that terminal is pruned from the tree.
9. When an input cladogram specified by `topofile` (page 322) or `topology` (page 323) does not have data corresponding to a taxon in the terminals file, POY adds the terminal as a basal branch.
10. Inconsistencies between an input cladogram and the terminals file are written to the standard error file.
11. POY does not limit missing data for input cladograms as it does for input data files.

theta

Sets the theta value for likelihood analysis.

Syntax

```
-theta thetanum
```

Arguments

Argument	Description
thetanum	An integer. The theta value (the proportion of invariant sites) expressed as a percentage. Default = none

Example

```
poy chel.seq -likelihood -theta 0.1
```

time

Displays execution time in seconds.

notime

Turns off display of execution time.

Syntax

```
-time
```

```
-notime
```

Arguments

None

Example

```
poy chel.seq -notime
```

topodiagnoseonly Skips all cladogram building and refinement when an input topology is specified.

notopodiagnoseonly

Default value. Performs cladogram building and refinement when an input topology is specified.

Syntax

```
-topodiagnoseonly
-notopodiagnoseonly
```

Arguments

None

Example

```
poy chel.seq -topofile chel.topo -impliedalignment
  -topodiagnoseonly
```

Comments

Use this command to evaluate input topologies or to calculate implied alignments for input cladograms.

topofile Reads topologies from a specified file.

Syntax

```
-topofile intree
```

Arguments

Argument	Description
intree	A string. The name of the input cladogram file. Default = none

Example

```
poy chel.seq -topofile chel.topo
```

Comments

1. Refer to "Input Cladograms" on page 156.

2. If no character input data are included in the command line, trees are read as is. However, the basal node is not resolved—that is, (a(b(c d))) is treated as (a b(c d)).
3. If character input data are included in the command line, trees are diagnosed. Zero cost branches are collapsed.
4. If an outgroup is specified using `topooutgroup` (page 324), the trees are rerooted to it.
5. If `toposkipidentical` (page 325) is used, identical input cladograms (after rerooting when an outgroup is specified) are filtered out.

topolist Writes topologies with costs to standard output after reading and diagnosing input cladograms.

notopolist

Turns off writing topologies after reading and diagnosing input cladograms.

Syntax

```
-topolist
-notopolist
```

Arguments

None

Example

```
poy chel.seq -topofile chel.topo -notopolist
```

Comments

Use this command to check topologies before branch swapping.

topology Reads an input cladogram from the command line.

Syntax

```
-topology "intree;"
```

Arguments

Argument	Description
"intree;"	A string. The input cladogram. One or more trees can be specified. The trees must be terminated by a semicolon and enclosed in double quotes. Default = none

Example

```
poy chel.seq -topology "(Americhernus
  (((((((Chanbria Limulus) (Centruoides (Hadrurus
```

```
Pauroctonus))) Mastigoproctus) (Amblypygid
((Rhiphicephalus Vonones) ((Aportus Alentus)
Thermobius)))) Hypochilus) Thelichoris) Gea)
Artemia));"
```

Comments

1. Refer to "Input Cladograms" on page 156 in Chapter 13.
2. If no character input data are included in the command line, cladograms are read as is. However, the basal node is not resolved—that is, (a(b(c d))) is treated as (a b(c d)).
3. If character input data are included in the command line, cladograms are diagnosed. Zero cost branches are collapsed.
4. If an outgroup is specified using `topooutgroup` (page 324), the cladograms are rerooted to it.
5. If `toposkipidentical` (page 325) is used, identical input cladograms (after rerooting when an outgroup is specified) are filtered out.

topooutgroup Sets the outgroup used to which input cladograms are rerooted.

Syntax

`-topooutgroup` *`taxon`*

Arguments

Argument	Description
`taxon`	A string. The taxon name of the outgroup. Default = none

Example

```
poy chel.seq -topofile chel.topo -topooutgroup
  Artemia
```

Comments

1. Input cladograms are rerooted to the specified taxon immediately after they are read.
2. Sibling clades in the rerooted cladogram are ordered in one of two ways:
   - from small to large
   - or alphabetically.
3. The reordering of siblings can lead to a change in diagnosed costs or a change in zero-cost branch assessments. If the input cladograms were obtained from a previous analysis, the original cost might not result, even with the same outgroup.

topopickrandom Sets the number of input cladograms pseudorandomly (`seed` on page 313) from all input cladograms specified following `toposkipidentical` (page 325) or `notoposkipidentical`.

Syntax

`-topopickrandom` *`trees`*

Arguments

Argument	Description
`trees`	An integer. The number of input cladograms selected pseudorandomly from all input cladograms. Default = the total number of trees specified following `toposkipidentical` or `notoposkipidentical` on the command line.

Example

```
poy chel.seq -topofile chel.seq -topopickrandom 3
```

Comments

1. If the value of *`trees`* is greater than the number of input cladograms, it is set to the number of input cladograms.
2. `topopickrandom` takes precedence over `topopwpickrandom`.

topopwpickrandom Sets the percent of input cladograms pseudorandomly (`seed` on page 313) from all input cladograms specified following `toposkipidentical` (page 325) or `notoposkipidentical`.

Syntax

`-topopwpickrandom` *`percent`*

Arguments

Argument	Description
`percent`	An integer. The percent of input cladograms selected pseudorandomly from all input cladograms. Default = 100

Example

```
poy chel.seq -topofile chel.seq -topopwpickrandom 25
```

Comments

`topopickrandom` (page 325) takes precedence over `topopwpickrandom`.

toposkipidentical Filters out duplicate input cladograms.

notoposkipidentical

Default value. Does not filter out duplicate input cladograms.

Syntax

```
-toposkipidentical
-notoposkipidentical
```

Arguments

None

Example

```
poy chel.seq -topofile chel.seq -topopwpickrandom 33
  -notoposkipidentical
```

Comments

1. Filtering occurs
   - after any rerooting (if `topooutgroup` (page 324) is used)
   - after diagnosis and collapsing of zero-cost branches (if character input data are present).
2. Cladograms are evaluated as unrooted, so that (a(b c)) is identical to ((b c)a).

totallikelihood Calculates the likelihood of an optimization alignment by summing only the major optimization alignments.

nototallikelihood

Default value. Calculates the likelihood based on the dominant optimization alignment or state in fixed-states or search-based analysis.

Syntax

```
-totallikelihood
-nototallikelihood
```

Arguments

None

Example

```
poy chel.seq -likelihood -totallikelihood
```

Comments

In contrast to this command, `trullytotallikelihood` (page 328) uses all optimization alignments to calculate the likelihood. `trullytotallikelihood` results in higher likelihoods, but at a premium in execution time.

trailinggap Sets the cost of leading and trailing gaps in a sequence.

Syntax

```
-trailinggap gapcost
```

Arguments

Argument	Description
gapcost	An integer. The cost of leading and trailing gaps. Default = none

Example

```
poy chel.seq -trailinggap 1 -noleading
```

Comments

If this command is not used, leading and trailing gaps are assigned the value set by `gap` (page 257) or `molecularmatrix` (page 283).

treefuse Performs cladogram search using tree fusing.

notreefuse

Turns off tree fusing.

Syntax

```
-treefuse
-notreefuse
```

Arguments

None

Example

```
poy chel.seq -replicates 10 -treefuse
```

treefusespr Performs a SPR cladogram search on new cladograms found during a tree fusing search.

notreefusespr

Turns off SPR cladogram search on new cladograms found during tree fusing.

Syntax

```
-treefusespr
-notreefusespr
```

Arguments

None

Example

```
poy chel.seq -replicates 10 -treefuse -treefusespr
```

treefusetbr Performs a TBR cladogram search on new cladograms found during a tree fusing search.

notreefusetbr

Turns off TBR cladogram search on new trees found during tree fusing.

Syntax

```
-treefusetbr
-notreefusetbr
```

Arguments

None

Example

```
poy chel.seq -replicates 10 -treefuse -treefusetbr
```

trullytotallikelihood Calculates the likelihood of an optimization by summing all optimizations.

notrullytotallikelihood

Default value. Calculates the likelihood based on the dominant optimization alignment.

Syntax

```
-trullytotallikelihood
-notrullytotallikelihood
```

Arguments

None

Example

```
poy chel.seq -likelihood -trullytotallikelihood
```

Comments

In contrast to this command, `totallikelihood` (page 326) uses a subset of potential optimizations near the major optimizations to calculate the likelihood. `trullytotallikelihood` results in higher likelihoods, but at a premium in execution time.

verbose Writes cladogram search progress information to the standard error file.

noverbose

Default value. Cladogram search progress information is not output.

Syntax

```
-verbose
```

```
-noverbose
```

Arguments

None

Example

```
poy chel.seq -noverbose
```

weight

Sets the weight of the data in the preceding input data file.

Syntax

```
-weight weight
```

Arguments

Argument	Description
weight	An integer. The weight applied to data in the preceding file. Default = 1

Example

```
poy -defaultweight 2 -weight 3 chel.morph chel.seq
```

Comments

This command affects only the single data file that precedes it on the command line.

Glossary

additive character A character type for which the transformation costs are determined by the number of steps between characters. Compare nonadditive characters and Sankoff characters.

ad hoc An improvised and often impromptu examination. In cladistics, used in reference to *a priori* invocations of homoplasy to explain similarity as nonhomologous.

analogy A character state in two or more taxa that does not have a single origin.

ancestor An internal node on a cladogram. A taxon from which another taxon is descended.

apomorphy The state of a character derived from an ancestral character state (plesiomorphy). See also autapomorphy, synapomorphy, and polarity.

a posteriori Knowledge derived from experience. In the context of phylogenetic analysis, information derived from an analysis, particularly the evaluation of a cladogram. Compare *a priori*.

a priori Knowledge derived prior to experience. In the context of phylogenetic analysis, assumptions and principles upon which an analysis is based, such as parsimony. Compare *a posteriori*.

autapomorphy An apomorphy unique to one taxon. For example, the hairlessness of cetaceans is autapomorphic among mammals.

automatic alignment Alignment of molecular characters based on an algorithmic procedure, generally computer-based.

bandwidth A network's or device's communication capacity—how much data it can pass per unit time

Bayes The statistical methodology concerned with posterior probabilities. Reverend Thomas Bayes (born in London, 1702, died in Tunbridge Wells, 1761), a Nonconformist minister in England, set out his theory of probability in *Essay towards solving a problem in the doctrine of chances* published in the *Philosophical Transactions of the Royal Society of London* in 1764.

Beowulf The legendary sixth-century hero who freed the Danes of Heorot by destroying the oppressive monster Grendel. As a metaphor, "Beowulf" has been applied to a new strategy in high performance computing that exploits mass-market technologies to overcome the oppressive costs in time and money of supercomputing. A supercomputer built from commodity, off-the-shelf components, in the form of a network (or cluster) dedicated to parallel processing.

bifurcation See dichotomy. Compare polytomy.

binary character Character with two states (for example, chitin exoskeleton: (0) absent, (1) present).

branch The path or edge connecting two nodes.

branch and bound An exact solution algorithm that guarantees finding all optimal solutions. In phylogenetics, this procedure is used to find all optimal cladograms for a data set.

branch swapping A stepwise method of searching for optimal phylogenetic trees in which monophyletic groups on a cladogram are rearranged and evaluated for improvements in optimality.

Bremer support index A measure of a clade's optimality. The index is the difference between the minimum cost cladogram with and without the clade.

cache A portion of a computer's memory intended to improve performance by providing a temporary storage area for instructions and data; the cache can be part of the computer's main memory or a bank of specialized, high speed memory

character A historically independent transformation series. See transformation, homology, and nested character.

character congruence The degree of agreement among the characters in supporting a hypothesized cladogram. The degree of synapomorphy and lack of homoplasy. Indicated by summary metrics such as

the consistency index. Compare topological congruence. See also congruence.

character state transformations The ordered array of states that a character can take. The hypothesized descent of a character's states.

character state The possible form that a character takes. For binary characters, the two states are often *structure present* and *structure absent*. For multistate characters, the state consists of one of multiple forms (for example, if the character is an RNA position, a possible state is one of the four nucleotides: adenine, guanine, cytosine, or uracil). For any set of taxa, character states are represented by a corresponding set of discrete values.

chromosome The cellular structure consisting of DNA and protein that consists of a linear sequence of genes.

clade A monophyletic group of taxa, that is, an ancestor taxon and all its descendants. On a cladogram, a clade is represented by all taxa arising from a single node.

cladistics The classification of taxa using character synapomorphies, from which identification of monophyletic groups is inferred.

cladogram A dendrogram (a diagram with a branching structure) that represents hypothetical clades (monophyletic groups of taxa) and that identifies hypothesized character state transformations. The outcome of a phylogenetic analysis. Compare tree.

cluster A network of computers created to perform a specialized process, such as a Beowulf created for parallel processing.

combined analysis The inclusion of diverse sources of available evidence. The term is generally applied when molecular and morphological data are analyzed simultaneously.

composite optima Term introduced by Goloboff (1999) that refers to the presence of optima for different subclades such that all combined create the global optimum.

computational complexity theory Part of the theory of computation dealing with the resources required during computation to solve a given problem. The most common resources are time (how many steps does it take to solve a problem) and space (how much memory does it take to solve a problem).

congruence Agreement of data. Compare incongruence. See also character congruence and topological congruence.

consensus A method for depicting incompatibilities among cladograms. Monophyletic groups are formed based on logical consistency. Strict consensus consists of only those monophyletic groups that are the same in all fundamental cladograms.

consistency index A measure of consistency, bound between 0 (maximal inconsistency) and 1 (perfect consistency). Those values are computed as the ratio of minimum cladogram cost divided by the actual cladogram length.

convergence The evolution of similar or identical characters in two or more taxa that do not share a common ancestor. See also analogy. Compare to homology.

cost The weighted sum of character state transformations required to map a topology from a data set of character states. Compare length.

criterion See optimality criterion.

data The set of characters studied or observed.

data partition See partition.

decay analysis See Bremer support index

dichotomy The branching pattern of a clade's resolved evolutionary history: two (and only two) descendant branches arise from a single ancestral branch. Compare polytomy.

direct optimization Algorithm for analyzing systematic data (usually sequence data) under dynamic homology via single-step phylogenetic analysis that avoids multiple alignment.

down pass Optimization of character states on a cladogram from terminal taxa to root. Compare up pass.

duplication A nucleotide sequence copied from one location on the genome to another. Compare paralogy.

dynamic homology Type of homology that is not fixed prior to the phylogenetic analysis and that therefore is cladogram-dependent. Compare static homology.

dynamic programming A computational method for finding an optimal solution to a problem by selecting the optimal next step at each stage in the computation. See also heuristic solution.

edge See branch.

edit cost The cost, in weighted events, to transform one structure into another. In a pairwise alignment following the Needleman and Wunsch algorithm, the cost implied by the given alignment.

evidence Observations that have been coded into characters and can be used to differentiate among phylogenetic hypotheses (that is, cladograms).

exact solution A problem-solving method in which all possible solutions are, at least implicitly, evaluated. Compare heuristic solution.

fixed state optimization A type of character optimization in which hypothetical ancestral sequences are not inferred during cladogram construction. Rather, sets of hypothetical ancestral sequences are drawn from observed sequences and then searched for efficient fits as nodal sequences via dynamic programming to calculate cladogram cost.

fundamental cladogram The optimal individual topologies output by a phylogenetic analysis. Consensus techniques combine fundamental cladograms to arrive at a compromise representation.

gap The absence of a corresponding nucleotide in a nucleic acid sequence when compared to another. See also indel.

gene rearrangement A change in the relative gene order of a string of genes (that is, a chromosome).

genetical algorithm A type of heuristic refinement to (in this case) phylogenetic hypotheses that involves recombination and selection to increase the optimality of a solution. See tree fusing.

genotype The genetic array of an organism.

genotypic data Character states derived from DNA, RNA, and gene sequence data; in contrast to the concept of molecular data, genotypic data represent directly an organism's genetic signal. Compare phenotypic data.

granularity The degree to which a process can be divided into subprocesses that can be allocated to individual CPUs in a parallel processing environment.

heritable A trait that is passed with or without change from the ancestor to its descendants. A nonheritable trait is a characteristic of the individual that results from environmental modifications.

heuristic solution A partial solution to a problem, presumably usefully close to the optimal one, or even the optimal, that results from the application of algorithms that tentatively reduce the size of the problem to a manageable scale. There is no way to know if it is actually the optimal solution; the optimal solution might be different or impossible to determine.

hierarchy A graded or ranked series of nested objects. A cladogram is a hierarchy of clades.

homology When a character is shared by two or more taxa that is descended from an ancestor of the taxa. The null hypothesis in constructing monophyletic groups during tree construction. Compare homoplasy. See primary homology and secondary homology.

homoplasy When a character state is shared by two or more taxa that is not the result of common ancestry, but instead is the result of parallelism or convergence. Compare homology.

HTU See hypothetical taxonomic unit.

hypothesis An empirically testable scientific statement.

hypothesis testing The relative evaluation of competing hypotheses with critical evidence.

hypothetical taxonomic unit The collection of character states ascribed to a taxon inferred by a phylogenetic analysis, in contrast to an observed taxon. Compare operational taxonomic unit.

implicit enumeration The effective evaluation of all the topologies that are possible for a given number of taxa without examining each possibility.

implied alignment A visual representation of the synapomorphies involving sequence substitutions and insertion/deletions on a particular cladogram.

implied weighting A method that searches for trees that maximize the function $F = K/(K + S)$, where K is a concavity constant (set by the user) and S are the extra steps. For details see Goloboff (1993b). See also successive approximation weighting.

incongruence also known as incompatibility. This refers to different characters or trees suggesting different groups of relationships. Incongruence is due to the presence of homoplasy. Compare congruence.

indel An insertion/deletion event at a location in nucleic acid or protein sequence—for example, the insertion of a nucleotide in DNA sequence. See also gap.

ingroup The set of taxa hypothesized as monophyletic. See outgroup.

integer arithmetic An algorithm that uses only integers in its calculations; integer arithmetic is less demanding and computationally exact as opposed to floating point arithmetic.

intron A noncoding segment of a gene. A nucleotide sequence that occurs between coding regions of a gene. Typical of eukaryotes.

inversion Rotation of a chromosomal segment through 180° so that the based order is reversed.

iterative pass optimization An algorithm that combines three-sequence direct optimization with iterative improvement.

jackknife A method used to evaluate in a phylogenetic tree. A jackknife consists of reconstructing the data matrix by sampling a subset of the original without replacement.

job A unit of work, which can be a single program or a group of programs working in concert.

kernel The section of a computer program that provides the program's essential services and, in the case of the operating system, remains in memory at all times.

latency The time difference between a data request and when the data are delivered.

length The unweighted sum of character state transformations required to map a topology from a data set of character states. Compare cost.

likelihood See maximum likelihood.

load balancing The relative load or business of jobs spawned on slave processors.

locus A fragment of the genome under analysis. Free of functional considerations (a locus could simply be a PCR-fragment of unknown function).

majority rule consensus A representation of those nodes that are present in 50% or more of the equally parsimonious trees resulting from an analysis. Compare strict consensus.

manual alignment The manual adjustment of sequences to establish homology hypotheses.

master node A node that assigns work and manages communication on a dedicated network, such as a Beowulf.

matrix character See Sankoff character.

maximum likelihood An optimality criterion used to infer phylogenetic relationships from character data. Given a data set, a set of parameters (branch lengths, substitution rates, etc.) and a topology, ML seeks to maximize the probability of the data given the cladogram and the parameters. (See Felsenstein 1981 for an entree to ML applications in phylogeny reconstruction.)

maximum parsimony See parsimony.

Mbps Megabits per second; a common unit of measure for transmission rates over networks.

metricity See triangle inequality.

middleware A computer program that manages communication between the application layer used by users and an operating layer.

missing data Data entries that are missing for a taxon or set of taxa. Usually represented by a question mark (?) in the matrix.

molecular data Character states derived from DNA, RNA, protein sequence, and chromosomal data. Compare morphological data, phenotypic data, and genotypic data.

molecular systematics Phylogenetic analysis using genome, nucleic acid, or protein data.

monophyly When a group of taxa are classified as descendants of a common ancestor. Compare polyphyly.

morphological data Character states derived from anatomical features and behavior. Compare molecular data and phenotypic data.

MPI Message passing interface; middleware that handles communication on a parallel processing cluster, such as a Beowulf.

multifurcation See polytomy. Compare dichotomy.

multistate character A character with more than two states.

mutational bias The tendency of a region of DNA to become biased in its nucleotide composition.

nearest neighbor interchange A search algorithm for the optimal topology that juxtaposes neighboring clades, then evaluates the result. If the new topology is more parsimonious than the original, it becomes the starting point for a new NNI iteration. A less complete algorithm than subtree pruning and reconnection or tree bisection and reconnection.

network (1) An unrooted tree. (2) A grid of hardware.

network backbone The cable or other transmission mechanism that connects all elements of a network.

network interface card The device installed on individual computers that enables communication over a network.

Newick format The representation of a topology using nested parentheses. For example, (A,(B,C)) specifies taxa B and C as sister clades.

NIC See network interface card.

NNI See nearest neighbor interchange.

node (1) A vertex on a cladogram with connections to one (terminal taxon), two (root of cladogram), or three (HTU) other nodes. (2) An individual computer in a cluster such as a Beowulf. Typically, nodes are counted by the number of motherboards, which may have more than one CPU and network interface.

nonadditive character A character type for which transformation costs are the same regardless of the number of steps between states. Compare additive character and Sankoff character.

NP Nondeterministic polynomial time. A set of computational decision problems that may have more than one result but are solvable by a number of steps that is a polynomial function with respect to the input.

The concept of nondeterminism invokes a method of generating potential solutions using trial and error. Finding a solution may take exponential time as long as a potential solution can be verified in polynomial time. Compare P and NP-complete.

NP-complete Nondeterministic polynomial time complete. A set of computational decision problems that is a subset of NP problems with the additional property that it is also NP-hard. Formally defined by Cook (1971) as follows. A decision problem C is NP-complete if it is in NP and if every other problem in NP is reducible to it. "Reducible" here means that for every NP problem L, there is a polynomial-time algorithm that transforms instances of L into instances of C, such that the two instances have the same truth values. As a consequence, if we had a polynomial time algorithm for C, we could solve all NP problems in polynomial time. Compare to P and NP. Important in the context of finding parsimonious trees (Foulds and Graham 1982).

nucleic acid sequences A chain of ribonucleic or deoxyribonucleic acids, typically representing a naturally occurring region of genome or transcriptome. In a computational context, nucleic acid sequences are typically represented by ASCII (American Standard Code for Information Interchange) strings conforming to conventions defined by IUPAC (The International Union of Pure and Applied Chemistry).

observation A feature of a natural object, organism, or phenomenon, determined by human senses or perceived via instruments. For example, a region of DNA amplified and sequenced with machinery and PCR reagents can be considered an observation.

ontogenetic Having to do with development of an organism.

operational taxonomic unit An observed individual, strain, species, or any classified group of organisms used in phylogenetic analysis. Compare hypothetical taxonomic unit.

optimality criterion The standard such as parsimony or likelihood that determines how well a cladogram explains a set of observed taxa.

optimization Based on an optimality criterion, the procedures used to numerically evaluate observed taxa and determine the cladogram that best explains their characteristics.

origin-loss The insertion or deletion of a locus between ancestral and descendant chromosomes.

OTU See operational taxonomic unit.

outgroup The taxon or taxa used in a phylogenetic analysis to establish character synapomorphies among the taxa of interest. The latter are hypothesized to form a monophyletic group with respect to the outgroup.

overhead Resources such as processing time or storage space consumed for purposes that are incidental to, but necessary to, the main computation. In practice, overhead includes initializing the POY run and communications throughout the run.

P In computational complexity theory, the set P for "polynomial" consists of all those decision problems that can be solved on a deterministic sequential machine in an amount of time that is polynomial with respect to the size of the input. Compare with NP and NP-complete.

parallel processing A computing environment that uses multiple processors to perform work on each CPU concurrently.

paraphyletic A group of taxa descended from a common ancestor that does not include all descendants of that ancestor. Compare monophyly.

paralogy The correspondence relationship between two genetic loci that is due to a gene duplication event, as opposed to an orthology that is due to lineage splitting.

parsimony An optimality criterion based on simplicity, used to infer phylogenetic relationships from character data. The lowest cost cladogram is the most favored.

partition A user defined segment of related data such as a locus within a genome.

phenotype Any supergenetic feature of organisms. Any characteristic that is not a component of the genome.

phenotypic data Character states derived from anatomical features, behavior, and protein sequences—that is, characteristics derived by inheritance, in contrast to the genetic signal itself. Compare genotypic data and morphological data.

platform independent A computing paradigm and software that can be used on any computing architecture via the universality of the source code.

plesiomorphy The ancestral state of a character from which one or more character states are derived (apomorphy). See synapomorphy and polarity.

point of failure The process or device on a network that halts processing.

polarity The identification of character states as ancestral (plesiomorphic) or derived (apomorphic).

polynomial time A polynomial time algorithm finds a solution in a finite number of steps that can be computed as a polynomial function of the problem's size. See also P, NP, and NP-complete.

polyphyly When a group of taxa are classified together, but they are not descendants of the same ancestor. Phylogenetics does not permit polyphyly, whereas phenetics does. Compare monophyly.

polytomy The branching pattern of a clade having an unresolved evolutionary history: three or more descendant branches arise from a single ancestral branch. Compare dichotomy.

portal A pathway into and out of a computer.

positional homology When amino acids or nucleotides are in the same relative position along a protein or nucleic acid sequence, respectively, and are hypothesized as being homologous. Each amino acid or nucleotide is a character state in an alignment.

prealigned Sequence data (amino acids or nucleotides) of equal length that do not require alignment or unequal length that have been subjected to a process such as multiple sequence alignment to set up static columns of provisionally homologous residues.

primary homology Coined by de Pinna (1991) to refer to initial, similarity-based propositions of homology. De Pinna asserted that homology discovery necessarily requires the proposition of "primary" homology based on a similarity relation for hypothesis "generation," followed by proposition of "secondary" homology based on the test of character congruence for hypothesis "legitimation." Only the latter was considered phylogenetic by de Pinna. These views are rejected in this book as conceptually misguided and operationally unnecessary. See also dynamic homology, static homology, and positional homology.

purine A base found in nucleic acids that consists of two connected rings of carbon and nitrogen; exemplified in DNA and RNA by adenine or guanine.

PVM Parallel virtual machine; middleware that handles communication on a parallel processing cluster, such as a Beowulf. See also MPI.

pyrimidine A base found in nucleic acids that consists of a single ring of carbon and nitrogen; exemplified in DNA as cytosine and thymine and in RNA as cytosine and uracil.

random addition sequence A Monte Carlo procedure in which many random replicates are used to vary the addition order of the taxa, creating weighted Wagner trees, a primary step common to all phylogenetic search algorithms. In POY intermediate swapping steps are often included during initial cladogram building. See commands: `buildspr` and `buildtbr`, which are used in conjunction with replicates.

RAS See random addition sequence.

ratchet A simulated annealing and Monte Carlo procedure in which various swapping replicates contain some fraction of characters that

have been reweighted or groups constrained. Ratcheting is an iterative procedure and all characters are set back to original weights before final cladogram length for each replicate is calculated. Because this procedure temporarily changes the optimality landscape for the data, local optima can be escaped. Such escapes are not possible for typical branch swapping algorithms, which, like hill climbing algorithms, may be restricted to local optima. The original procedure is from Nixon (1999). Due to its heavy reliance on branch swapping, ratcheting is CPU intensive but is useful for searches with large numbers of taxa and scales well across hundreds of processors in POY.

recursion A computational process that spawns itself. Recursion in used in the algorithms in POY that generate and processe cladograms.

refinement Any of the procedures used after (and in the case of POY, during) initial cladogram construction. These include branch swapping, tree drifting, ratcheting, tree fusion, and (not yet in POY) sectorial searches. See Goloboff (1999) for a discussion of the role of refinement in searches on large data sets. See Janies and Wheeler (2001) for scaling of refinement procedures in a Beowulf cluster.

repeatability In a scientific experiment or analysis other scientists must be able to performed the same experiment in the same way and achieve the same result. For example, manual alignment or hand editing of aligned sequences is an unscientific procedure because it is not repeatable. Software provides a phylogenetic investigator a scientific means to analyze molecular sequences due to explicit codification and parameterization of procedures that can be repeated by other investigators.

reticulation The confluence (as opposed to splitting) of lineages in a cladogram or tree. Several types of genetic exchange may occur between taxa (for example, lateral gene transfer in microorganism, genes of mitochondrial origin in animals). As of now, reticulate evolution is largely treated as a class of homoplasy, but procedures to distinguish reticulation from other types of homoplasy are the subject of research.

reversal (1) In phylogenetic analysis, a character state transformation that reverses the character synapomorphy. That is, the character transforms from a derived to an ancestral state. (2) In reference to genomes, a synonym for inversion.

root The basalmost node, connected to two other nodes, that determines the polarity of character transformation in the whole tree. See also outgroup.

rooted Refers to a cladogram in which the direction of character transformation (polarity) is determined. Compare unrooted. See also root and outgroup.

run To execute a program; a program scheduled for execution.

Sankoff character A character type represented by a matrix that assigns transformation costs specific to character state changes. It is a general case of multistate qualitative characters for which additive and nonadditive characters are special cases.

scenario A phylogeny depicts not only a hypothesis of evolutionary relationships but also the relative order, frequency, and often independent evolution of features of organisms.

script A small program often interpreted directly by the operating systems shell or a high level language, such as PERL, without compilation. Scripts are often used to direct other programs, such as POY, through several related runs or use variables (environmental or user defined) to configure a programs source code for compilation into a binary.

search-based optimization An extension of fixed state optimization in which hypothetical ancestral sequences are not inferred during cladogram construction. Rather, sets of hypothetical ancestral sequences are precomputed and then searched for efficient fits as nodal sequences via dynamic programming to calculate costs of various cladograms. Although search-based optimization can, in principle, guarantee an exact solution (through a search of all possible ancestral sequences), this is unlikely to be practical. Thus search-based optimization derives its strength as an procedure via the examination of heuristic subsets of possible ancestral sequences akin to direct optimization's (or any random taxon addition procedure's) use of heuristic subsets of possible topologies.

secondary homology Term coined by de Pinna (1991) to refer to "legitimized" homology statements that result from the test of character congruence. Compare primary homology.

secondary structure The two-dimensional pattern of molecule folding of nucleotide and amino acid sequences caused by base- (nucleotide) and amino acid-pairing interactions. Secondary structure is modeled on the basis of thermodynamic (and ultimately quantum) interactions and minimization of free energy to maximize molecular stability.

sensitivity analysis The exploration of the effect of variation in parameter values and assumptions on analytical results.

sequence character A contiguous string of nucleotides or amino acids, whereby transformation involves the independent substitution, insertion, or deletion of those component units.

shared-ancestral character state See symplesiomorphy.

shared-derived character state See synapomorphy.

shared memory multiprocessor See SMP.

simulated annealing A general class of heuristic algorithms derived from the Metropolis algorithm to solve NP-complete optimization problems based on the analogy of the behavior of systems in thermal equilibrium at a finite temperature (see Kirkpatrick et al. 1983). Examples in systematics include tree drifting (Goloboff 1999) and the ratchet (Nixon 1999).

simultaneous analysis See combined analysis.

sister taxa Monophyletic taxa that arise from a common node.

slave node A node that performs work on a dedicated network, such as a Beowulf.

SMP shared memory multiprocessor. A computer having more than one CPU, all of which share memory.

spawn To launch a process from a currently running program.

SPR See subtree pruning and regrafting.

static approximation The heuristic search strategy of generating implied alignments, analyzing them as static (fixed) matrices (either in POY or an external program like PAUP* or TNT) to generate more or improved cladograms, and using the resulting cladograms for additional searching under dynamic homology.

static homology The relation among parts within transformation series constrained prior to phylogenetic analysis and coded as columns in a static (or fixed) matrix. Compare dynamic homology.

Steiner tree A minimum cost tree with a set of terminal and internal vertices. The nonterminal vertices are called Steiner vertices or Steiner points.

stderr Standard error output from a computer program. Redirected to file with the command `2> filename`.

stdin Standard input, usually the keyboard. Can be redirected from a file via <, as in the command `< filename`.

stdout Standard output from a computer program. Redirected to a file with the command `> filename`.

strict consensus A consensus tree formed from two or more separate fundamental trees that has only those clades shared by all fundamental trees. Compare majority rule consensus.

string-matching The minimum edit cost determination frequently used in sequence analysis. The most commonly used algorithm is that of Needleman and Wunsch (1970).

subtree pruning and regrafting (SPR) A cladogram search algorithm for the optimal topology that moves a monophyletic group from one branch to another, then evaluates the result. If the new topology is

more parsimonious than the original, it becomes the starting point for a new SPR iteration. Compare tree bisection and reconnection.

successive approximation weighting A method for reweighting characters based on some measure of their relative values.

supertree A tree-shaped object derived from the combination of two or more fundamental cladograms of partially overlapping sets of taxa. See also tree-shaped object.

support General epistemological concept relating evidence to hypotheses. As such, both verificationist (for example, statistico-probabilistic) and refutationist concepts have been implemented in systematics, though rarely with explicit definitions. Defined by Grant and Kluge (2003: 383) as "the degree to which critical evidence refutes competing hypotheses." A hypothesis is unsupported if it is either (1) decisively refuted by the critical evidence or (2) contradicted by other, equally optimal hypotheses (that is, evidence is ambiguous, such as when multiple most-parsimonious cladograms obtain); otherwise, it is supported.

support index See Bremer support index.

switch A device that manages communication traffic among nodes on a network.

symplesiomorphy A primitive character state shared by two or more taxa that do not constitute a monophyletic group.

synapomorphy A derived character state shared by two or more taxa that constitute a monophyletic group, which is taken as evidence of their common ancestry.

taxon (plural taxa) also known as an operational taxonomic unit (OTU). An observed individual, a strain, a species, or any classified group of organisms used in phylogenetic analysis. Compare hypothetical taxonomic unit.

TBR See tree bisection and reconnection.

tokogenetic Having to do with relationships among organisms within a species.

topology In mathematics generally, the description of geometries that are not affected by changes in size and shape, and which maintain continuity when joined. Specifically in phylogenetics, the branching geometry that describes the evolutionary relationship among taxa. For example, if A, B, and C are taxa, then (A,(B,C)) describes a possible topology or evolutionary relationship among them.

topological congruence The agreement among the branching patterns of cladograms. Often employed to measure partition congruence. Compare character congruence. See also congruence. Compare character congruence.

total evidence See combined analysis.

transformation Conceptually, a change from a primitive to a derived character state, a → a'. Operationally, an edit cost.

transition The substitution of nucleotides like-to-like. That is, a purine for a purine or a pyrimidine for a pyrimidine.

transposition The movement of a DNA segment from one location to another. The new location can be on the same or a different chromosome.

transversion The substitution of nucleotides that is not like-to-like. That is, a purine for a pyrimidine or a pyrimidine for a purine.

tree In systematics, a directed (rooted) branching diagram representing the phylogeny of a group of terminals. In computer science literature, a directed, acyclic graph. See also network, cladogram, Wagner tree, and Steiner tree.

tree bisection and reconnection A cladogram search algorithm that consists of bisecting a topology, then reconnecting the two subtopologies at all possible branches. Compare subtree pruning and regrafting.

tree drifting A simulated annealing-class cladogram searching algorithm that accepts suboptimal solutions during branch swapping with a specified probability as a way of escaping local optima. Proposed by Goloboff (1999). See also simulated annealing and ratchet.

tree fusing A cladogram searching algorithm that proceeds by exchanging subgroups between different cladograms. Proposed by Goloboff (1999). See also genetical algorithm.

tree length The total number of character state transformations required to map a topology from a data set of character states. Compare cost.

tree-shaped object A branching diagram with terminal taxa at the leaves that looks like a cladogram, but has no associated optimality value. As a result, it cannot participate in hypothesis testing (for example, "supertrees").

triangle inequality The statement of a metric space for any three points A, B, and C: $d(AB) \leftarrow d(AC) + d(CB)$. Metricity also requires $d(AB) = d(BA)$ for all AB. The triangle inequality is most frequently invoked in character transformation cost matrices among multistate characters. Nonmetric matrices imply unobserved states. In essence, the triangle inequality rules out shortcuts.

uniquely-derived See autapomorphy.

unrooted Refers to a cladogram in which the direction of character transformation (polarity) is not determined. Compare rooted. See also outgroup.

up pass Optimization of character states on a cladogram from root to terminal taxa. Compare down pass.

vertex A node on a cladogram.

Wagner tree Term coined by Farris (1970) for a cladogram in which observed taxa are confined to terminal nodes (tips or leaves) and characters are optimized independently to inner nodes, creating hypothetical taxonomic units.

Bibliography

Adkins, R. M., and R. L. Honeycutt. 1994. Evolution of the primate cytochrome *c* oxidase subunit II gene. Journal of Molecular Evolution 38: 215–231.

Aguinaldo, A. M. A., J. M. Turbeville, L. S. Lindford, M. C. Rivera, J. R. Garey, R. A. Raff, and J. A. Lake. 1997. Evidence for a clade of nematodes, arthropods and other moulting animals. Nature 387: 489–493.

Allard, M. W., J. S. Farris, and J. M. Carpenter. 1999. Congruence among mammalian mitochondrial genes. Cladistics 15: 75–84.

Baker, R. H., and R. DeSalle. 1997. Multiple sources of character information and the phylogeny of Hawaiian drosophilids. Systematic Biology 46: 654–673.

Barry, D., and J. Hartigan. 1987. Statistical analysis of hominoid molecular evolution. Statistical Science 2: 191–207.

Bininda-Emonds, O. R., J. L. Gittleman and M. A. Steel. 2002. The (super) Tree of Life: Procedures, problems, and prospects. Annual Review of Ecology and Systematics 33: 265–289.

Bremer, K. 1988. The limits of amino acid sequence data in angiosperm phylogenetic reconstruction. Evolution 42: 795–803.

Bremer, K. 1994. Branch support and tree stability. Cladistics 10: 295–304.

Brown, W. M., M. George, Jr., and A. C. Wilson. 1979. Rapid evolution of animal mitochondrial DNA. Proceedings of the National Academy of Sciences of the USA 76: 1967–1971.

Carpenter, J. M. 1988. Choosing among multiple equally parsimonious cladograms. Cladistics 4: 291–296.

Cavalli-Sforza, L. L., and A. W. F. Edwards. 1967. Phylogenetic analysis: Models and estimation procedures. American Journal of Human Genetics 19: 233–257.

Cook, S. A. 1971. The complexity of theorem-proving procedures. *In* Proceedings of the 3rd ACM Symposium on Theory of Computing, Shaker Heights, OH: 151–158. New York: ACM Press.

Cormen, T. H., C. E. Leiserson, and R. L. Rivest. 2001. Introduction to algorithms, 2nd ed. Cambridge, MA: MIT Press. 1180 pp.

De Pinna, M. C. C. 1991. Concepts and tests of homology in the cladistic paradigm. Cladistics 7: 367–394.

D'Haese, C. A. 2002. Were the first springtails semi-aquatic? A phylogenetic approach by means of 28S rDNA and optimization alignment. Proceedings of the Royal Society of London, Series B, Biological Sciences 269: 1143–1151.

Durbin, R., S. R. Eddy, A. Krogh, and G. Mitchison. 1998. Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. Cambridge: Cambridge University Press. 368 pp.

Edgecombe, G. D., G. Giribet, and W. C. Wheeler. 2002. Phylogeny of Henicopidae (Chilopoda: Lithobiomorpha): a combined analysis of morphology and five molecular loci. Systematic Entomology 27: 31–64.

Farris, J. S. 1969. A successive approximations approach to character weighting. Systematic Zoology 18: 374–385.

Farris, J. S. 1970. Methods for computing Wagner trees. Systematic Zoology 19: 83–92.

Farris, J. S. 1973. On comparing the shapes of taxonomic trees. Systematic Zoology 22: 50–54.

Farris, J. S. 1983. The logical basis of phylogenetic analysis. *In* N. I. Platnick and V. A. Funk (editors), Advances in Cladistics: 277–302. New York: Columbia University Press.

Farris, J. S. 1989. The retention index and the homoplasy excess. Systematic Zoology 38: 406–407.

Farris, J. S. 1997. The future of phylogeny reconstruction. Zoologica Scripta 26: 303–311.

Farris, J. S., V. A. Albert, M. Källersjö, D. Lipscomb, and A. G. Kluge. 1996. Parsimony jackknifing outperforms neighbor-joining. Cladistics 12: 99–124.

Felsenstein, J. 1978. The number of evolutionary trees. Systematic Zoology 27: 27–33.

Felsenstein, J. 1981. Evolutionary trees from DNA sequences: a maximum likelihood approach. Journal of Molecular Evolution 17: 368–376.

Felsenstein, J. 2004. Inferring Phylogenies. Sunderland, MA: Sinauer Associates, 664 pp.

Fitch, W. M. 1971. Toward defining the course of evolution: minimum change for a specified tree topology. Systematic Zoology 20: 406–416.

Fitch, W. M. and E. Margoliash. 1967. Construction of phylogenetic trees. A method based on mutation distances as estimated from cytochrome *c* sequences is of general applicability. Science 155: 279–284.

Foulds, L. R. and R. L. Graham. 1982. The Steiner problem in phylogeny is NP-complete. Advances in Applied Mathematics 3: 43–49.

Frost, D. R., R. Etheridge, D. Janies, and T. A. Titus. 2001a. Total evidence, sequence alignment, evolution of polychrotid lizards, and a reclassification of the Iguania (Squamata: Iguania). American Museum Novitates 3343: 1–38.

Frost, D. R., M. T. Rodrigues, T. Grant, and T. A. Titus. 2001b. Phylogenetics of the lizard genus *Tropidurus* (Squamata: Tropiduridae: Tropidurinae): direct optimization, descriptive efficiency, and sensitivity analysis of congruence between molecular data and morphology. Molecular Phylogenetics and Evolution 21: 352–371.

Garey, M. R., R. L. Graham, and D. S. Johnson. 1977. The complexity of computing Steiner minimal trees. SIAM Journal on Applied Mathematics 23: 835–859.

Garey, M. R. and D. S. Johnson. 1977. The rectilinear Steiner tree problem is NP-complete. SIAM Journal on Applied Mathematics 23: 826–834.

Gatesy, J., R. DeSalle, and W. C. Wheeler. 1993. Alignment–ambiguous nucleotide sites and the exclusion of systematic data. Molecular Phylogenetics and Evolution 2: 152–157.

Giribet, G. 2001. Exploring the behavior of POY, a program for direct optimization of molecular data. Cladistics 17: S60–S70.

Giribet, G. 2002. Relationships among metazoan phyla as inferred from 18S rRNA sequence data: a methodological approach. *In* R. DeSalle, G. Giribet and W. C. Wheeler (editors), Molecular Systematics and Evolution: Theory and Practice: 85–101. Basel: Birkhäuser Verlag.

Giribet, G. 2003. Stability in phylogenetic formulations and its relationship to nodal support. Systematic Biology 52: 554–564.

Giribet, G., D. L. Distel, M. Polz, W. Sterrer, and W. C. Wheeler. 2000. Triploblastic relationships with emphasis on the acoelomates and the position of Gnathostomulida, Cycliophora, Plathelminthes, and Chaetognatha: A combined approach of 18S rDNA sequences and morphology. Systematic Biology 49: 539–562.

Giribet, G., G. D. Edgecombe and W. C. Wheeler. 2001. Arthropod phylogeny based on eight molecular loci and morphology. Nature 413: 157–161.

Giribet, G., W. C. Wheeler, and J. Muona. 2002. DNA multiple sequence alignments. *In* R. DeSalle, G. Giribet, and W. C. Wheeler (editors), Molecular Systematics and Evolution: Theory and Practice: 107–114. Basel: Birkhäuser Verlag.

Gladstein, D. 1997. Efficient incremental character optimization. Cladistics 13: 21–26.

Goloboff, P. A. 1993a. Character optimization and calculation of tree lengths. Cladistics 9: 433–436.

Goloboff, P. A. 1993b. Estimating character weights during tree search. Cladistics 9: 83–91.

Goloboff, P. A. 1996a. Methods for faster parsimony analysis. Cladistics 12: 199–220.

Goloboff, P. A. 1996b. Pee–Wee. Ver. 2.5.1. American Museum of Natural History.

Goloboff, P. A. 1996c. PHAST. Ver. 1.1. American Museum of Natural History.

Goloboff, P. A. 1996d. SPA (Sankoff Parsimony Analysis). Ver. 1.1. American Museum of Natural History.

Goloboff, P. A. 1998. Nona, v. 2.0. Ver. 2.0. Program and documentation available at www.cladistics.com.

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

Goloboff, P. A. 2002. Techniques for analyzing large data sets. *In* R. DeSalle, G. Giribet, and W. Wheeler (editors), Techniques in Molecular Systematics and Evolution: 70–79. Basel: Birkhäuser Verlag.

Goloboff, P. A., and J. S. Farris. 2001. Methods for quick consensus estimation. Cladistics 17: S26–S34.

Goloboff, P. A., J. S. Farris, M. Källersjö, B. Oxelman, M. Ramírez, and C. A. Szumik. 2003a. Improvements to resampling measures of group support. Cladistics 19: 324–332.

Goloboff, P. A., J. S. Farris, and K. Nixon. 2003b. TNT: Tree analysis using New Technology. Version 1.0. Ver. Beta test v. 0.2. Program and documentation available at http://www.zmuc.dk/public/phylogeny/TNT/ .

Grandcolas, P., and C. D'Haese. 2003. Testing adaptation with phylogeny: how to account for phylogenetic pattern and selective value together. Zoologica Scripta 32: 483–490.

Grant, T., and A. G. Kluge. 2003. Data exploration in phylogenetic inference: scientific, heuristic, or neither. Cladistics 19: 379–418.

Hein, J., C. Wiuf, B. Knudsen, M. B. Moller, and G. Wibling. 2000. Statistical alignment: computational properties, homology testing and goodness-of-fit. Journal of Molecular Evolution 302: 265–279.

Hendy, M. D., and D. Penny. 1982. Branch and bound algorithms to determine minimal evolutionary trees. Systematic Zoology 59: 277–290.

Hennig, W. 1966. Phylogenetic Systematics. Urbana: University of Illinois Press, 263 pp.

Hormiga, G., M. Arnedo, and R. G. Gillespie. 2003. Speciation on a conveyor belt: sequential colonization of the Hawaiian Islands by *Orsonwelles* spiders (Araneae, Linyphiidae). Systematic Biology 52: 70–88.

Huelsenbeck, J. P., and K. A. Crandall. 1997. Phylogeny estimation and hypothesis testing using maximum likelihood. Annual Review of Ecology and Systematics 28: 437–466.

Huelsenbeck, J. P., and F. Ronquist. 2003. MrBayes: Bayesian Inference of Phylogeny, v. 3.0. Program and documentation available at: http://mrbayes.csit.fsu.edu/index.php.

Janies, D. 2001. Phylogenetic relationships of extant echinoderm classes. Canadian Journal of Zoology 79: 1232–1250.

Janies, D., and W. C. Wheeler. 2001. Efficiency of parallel direct optimization. Cladistics 17: S71–S82.

Janies, D. A., and W. C. Wheeler. 2002. Theory and practice of parallel direct optimization. *In* R. DeSalle, G. Giribet, and W. C. Wheeler (editors), Molecular Systematics and Evolution: Theory and Practice: 115–123. Basel: Birkhäuser Verlag.

Källersjö, M., V. A. Albert, and J. S. Farris. 1999. Homoplasy increases phylogenetic structure. Cladistics 15: 91–93.

Kirkpatrick, S., C. Gellat, and M. Vecchi. 1983. Optimization by simulated annealing. Science 220: 671–680.

Kjer, K. M. 1995. Use of rRNA secondary structure in phylogenetic studies to identify homologous positions: An example of alignment and data presentation from the frogs. Molecular Phylogenetics and Evolution 4: 314–330.

Kluge, A. G. 1989. A concern for evidence and a phylogenetic hypothesis of relationships among *Epicrates* (Boidae, Serpentes). Systematic Zoology 38: 7–25.

Kluge, A. G., and J. S. Farris. 1969. Quantitative phyletics and the evolution of anurans. Systematic Zoology 18: 1–32.

Maddison, D. R. 1991. The discovery and importance of multiple islands of most-parsimonious trees. Systematic Zoology 40: 315–328.

Mathews, D. H., J. Sabina, M. Zuker, and D. H. Turner. 1999. Expanded sequence dependence of thermodynamic parameters improves prediction of RNA secondary structure. Journal of Molecular Evolution 288: 911–940.

McGuire, G., M. C. Denham, and D. J. Balding. 2001. Models of sequence evolution for DNA sequences containing gaps. Molecular Biology and Evolution 18: 481–490.

Mickevich, M. F., and J. S. Farris. 1981. The implications of congruence in *Menidia*. Systematic Zoology 27: 143–158.

Mindell, D. P., and C. E. Thacker. 1996. Rates of molecular evolution: phylogenetic issues and applications. Annual Review of Ecology and Systematics 27: 279–303.

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

Moilanen, A. 2001. Simulated evolutionary optimization and local search: Introduction and application to tree search. Cladistics 17: S12–S25.

Morrison, D. A., and J. T. Ellis. 1997. Effects of nucleotide sequence alignment on phylogeny estimation: a case study of 18S rDNAs of Apicomplexa. Molecular Biology and Evolution 14: 428–441.

Nardi, F., G. Spinsanti, J. L. Boore, A. Carapelli, R. Dallai, and F. Frati. 2003. Hexapod origins: Monophyletic or paraphyletic? Science 299: 1887–1889.

Needleman, S. B., and C. D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology 48: 443–453.

Nilsson, N. J. 1998. Artificial Intelligence; a new synthesis. San Francisco: Morgan Kaufmann Publishers, 513 pp.

Nixon, K. C. 1999. The Parsimony Ratchet, a new method for rapid parsimony analysis. Cladistics 15: 407–414.

Nixon, K. C. 2002. Winclada, v. 1.00.08. Program and documentation available at www.cladistics.com.

Phillips, A., D. Janies, and W. C. Wheeler. 2000. Multiple sequence alignment in phylogenetic analysis. Molecular Phylogenetics and Evolution 16: 317–330.

Platnick, N. I. 1977. Cladograms, phylogenetic trees, and hypothesis testing. Systematic Zoology 26: 438–442.

Posada, D., and K. A. Crandall. 1998. MODELTEST: testing the model of DNA substitution. Bioinformatics 14: 817–818.

Regier, J. C., and J. W. Shultz. 2001. Elongation factor-2: a useful gene for arthropod phylogenetics. Molecular Phylogenetics and Evolution 20: 136–148.

Saitou, N., and M. Nei. 1987. The neighbor-joining method: a new method for reconstructing phylogenetic trees. Molecular Biology and Evolution 4: 406–425.

Sankoff, D. 1975. Minimal mutation trees of sequences. SIAM Journal on Applied Mathematics 28: 35–42.

Sankoff, D., and M. Blanchette. 1998. Multiple genome rearrangement and breakpoint phylogeny. Journal of Computational Biology 5: 555–570.

Sankoff, D., and P. Rousseau. 1975. Locating the vertices of a Steiner tree in arbitrary space. Mathematical Programming 9: 240–246.

Schuh, R. T. 2000. Biological systematics. Principles and applications. Ithaca, NY: Cornell University Press, 236 pp.

Schulmeister, S., W. C. Wheeler, and J. M. Carpenter. 2002. Simultaneous analysis of the basal lineages of Hymenoptera (Insecta) using sensitivity analysis. Cladistics 18: 261–268.

Sober, E. 1983. Parsimony methods in systematics. *In* N. I. Platnick and V. A. Funk (editors), Advances in Cladistics: 37–47. New York: Columbia University Press.

Swofford, D. L. 2002. PAUP* 4.0: Phylogenetic Analysis Using Parsimony (*and Other Methods). Ver. 4. Suderland, MA: Sinauer Associates.

Thompson, J. D., D. G. Higgins, and T. J. Gibson. 1994. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice. Nucleic Acids Research 22: 4673–4680.

Thorne, J. L., H. Kishino, and J. Felsenstein. 1991. An evolutionary model for maximum likelihood alignment of DNA sequences. Journal of Molecular Evolution 33: 114–124.

Tuffley, C., and M. Steel. 1997. Links between maximum likelihood and maximum parsimony under a simple model of site substitution. Bulletin of Mathematical Biology 59: 581–607.

Ukkonen, E. 1985. Finding approximate patterns in strings. Journal of Algorithms 6: 132–137.

Wagner, W. H. 1961. Problems in the classification of ferns. Recent Advances in Botany 1: 841–844.

Wenzel, J. W., and M. E. Siddall. 1999. Noise. Cladistics 15: 51–64.

Wheeler, W. C. 1993. The triangle inequality and character analysis. Molecular Biology and Evolution 10: 707–712.

Wheeler, W. C. 1995. Sequence alignment, parameter sensitivity, and the phylogenetic analysis of molecular data. Systematic Biology 44: 321–331.

Wheeler, W. C. 1996. Optimization alignment: the end of multiple sequence alignment in phylogenetics? Cladistics 12: 1–9.

Wheeler, W. C. 1999a. Fixed character states and the optimization of molecular sequence data. Cladistics 15: 379–385.

Wheeler, W. C. 1999b. Measuring topological congruence by extending character techniques. Cladistics 15: 131–135.

Wheeler, W. C. 2001a. Homology and DNA sequence data. *In* G. P. Wagner (editor), The Character Concept in Evolutionary Biology: 303–317. San Diego: Academic Press.

Wheeler, W. C. 2001b. Homology and the optimization of DNA sequence data. Cladistics 17: S3–S11.

Wheeler, W. C. 2002. Optimization Alignment: down, up, error, and improvements. *In* R. DeSalle, G. Giribet, and W. C. Wheeler (editors), Techniques in molecular systematics and evolution: 55–69. Basel: Birkhäuser.

Wheeler, W. C. 2003a. Implied alignment: a synapomorphy-based multiple-sequence alignment method and its use in cladogram search. Cladistics 19: 261–268.

Wheeler, W. C. 2003b. Iterative pass optimization of sequence data. Cladistics 19: 254–260.

Wheeler, W. C. 2003c. Search-based optimization. Cladistics 19: 348–355.

Wheeler, W. C., J. Gatesy, and R. DeSalle. 1995. Elision: A method for accommodating multiple molecular sequence alignments with alignment-ambiguous sites. Molecular Phylogenetics and Evolution 4: 1–9.

Wheeler, W. C., and D. S. Gladstein. 1994. MALIGN: a multiple sequence alignment program. Journal of Heredity 85: 417–418.

Wheeler, W. C.,and C. Y. Hayashi. 1998. The phylogeny of extant chelicerate orders. Cladistics 14: 173–192.

Wheeler, W. C., D. S. Gladstein, and Jan De Laet. 1996–2003. POY. Version 3.0. ftp.amnh.org /pub/molecular/poy. Documentation by Daniel Janies and Ward Wheeler. Command-line documentation by J. De Laet and W. C. Wheeler.

Wheeler, W.C., M.J. Ramírez, L. Aagesn, and S. Schulmeister. 2006. Partition-free congruence analysis: implications for sensitivity analysis. Cladistics 22: 256–263.

Index

B

C

U

W