WILEY-BLACKWELL

Cladistics 26 (2010) 72–85

Cladistics

10.1111/j.1096-0031.2009.00282.x

POY version 4: phylogenetic analysis using dynamic homologies

Andrés Varón[a,b,*], Le Sy Vinh[a,c] and Ward C. Wheeler[a]

[a]*Division of Invertebrate Zoology, American Museum of Natural History, Central Park West at 79th Street, New York, NY, USA;* [b]*Computer Science Department, The Graduate School and University Center, The City University of New York, 365 Fifth Avenue, New York, NY, USA;* [c]*College of Technology, Vietnam National University, Hanoi, 144 Xuan Thuy, Cau Giay, Hanoi, Vietnam*

Accepted 11 July 2009

Abstract

We present POY version 4, an open source program for the phylogenetic analysis of morphological, prealigned sequence, unaligned sequence, and genomic data. POY allows phylogenetic inference when not only substitutions, but insertions, deletions, and rearrangement events are allowed (computed using the breakpoint or inversion distance). Compared with previous versions, POY 4 provides greater flexibility, a larger number of supported parameter sets, numerous execution time improvements, a vastly improved user interface, greater quality control, and extensive documentation. We introduce POY's basic features, and present a simple example illustrating the performance improvements over previous versions of the application.

© The Willi Hennig Society 2009.

POY is an open source, phylogenetic analysis program for molecular and morphological data. Version 3.0.11 was released in September 2004, and work on version 4.0 began in 2005. After more than a year of public beta testing which started early in 2007, versions 4.0 and 4.1 have now been released.

Version 4 supports maximum parsimony as its optimality criterion[1]. Like most software of this class, POY analyses the standard non-additive, additive, and matrix characters commonly found in other phylogenetic analysis programs (Swofford, 1993; Goloboff, 1999a; Goloboff et al., 2008). Most importantly, POY supports the analysis of dynamic homology (DH) characters, which allow the use of unaligned sequences as characters (Wheeler et al., 2006). With DH characters, POY can infer substitutions, insertions, deletions, inversions, and translocations, at the locus, chromosomal, and genomic level, as the phylogenetic analysis goes on. This makes POY a unique application, providing the broadest range of characters for its users.

The main goals of version 4 were to increase the application's flexibility (e.g. POY 3.0 only supported one set of parameters for all sequences), increase performance, reduce the learning curve for new users, improve quality control, and maximize the maintainability and extensibility of the source code.

Here we describe the basic features of the program. We begin with its most important phylogenetic analysis features (see section on ''Phylogenetic analysis features''), the basic characteristics of the new user interface and command structure (''User interface''), followed by the script execution in sequential and parallel environments (''Script execution''), and a number of other relevant application features as well as limitations (''Other features''). This basic description is followed by performance comparisons (''Performance example''), and a list of available resources for current and new users (''Program resources, availability, distribution, and licence terms'').

This application note is a general overview of POY 4, and is not intended to be a replacement for the user manual. Instead, it is a description of its main features,

*Corresponding author.
E-mail address: avaron@amnh.org

[1]Previous versions of POY supported Maximum Likelihood (ML). See section on ''What the program cannot do'' for further information on this topic.

© The Willi Hennig Society 2009

and some formalisms required to understand the program's use.

Phylogenetic analysis features

As with most phylogenetic analysis software, the features in POY can be divided into three groups: calculating the evolutionary distance between a pair of vectors of states, computing the score of a tree given an assignment of character states to its terminals, and searching for a tree of minimal cost. More complex functions are performed by composing elements of these three groups (e.g. support calculation), while others belong to basic input and output functionality (e.g. printing a consensus tree).

For the most common types of static homology analyses, the first two groups (i.e. distance between vectors of states, and tree score) have well-known algorithms, for which efficient polynomial time solutions exist and have been implemented in POY 4. For dynamic homology characters, however, computing a distance and the cost of a tree can be major computational tasks by themselves.

Following these main groups, we describe the phylogenetic analysis features available in POY in a bottom-up fashion: first the character types that are supported, then the algorithms for the tree cost calculation (informally), and finally the search strategies. We briefly describe the input and output functions in the section on "Other features".

Supported character types

A character is defined with two components: its valid states and the function to compute the evolutionary distance between states. Considering the properties of valid states, two main groups of characters are supported in POY 4: static homology and dynamic homology. To define them, we must first clarify the notion of state.

Character states. We are interested in characters that encompass multiple sources of variation. The following four examples are not exhaustive, but illustrate this diversity.

1. Morphology. A typical character could be the fruit colour of a plant. The character states could be red, green, and yellow. Usually, such a set of valid states corresponds exactly to those observed in the taxa of interest. Consider now two possible encoding schemes: non-additive and additive.

As a non-additive character, the transformation cost between any pair of different states is equal. States that could occur in nature, but were not observed (such as orange), do not have any effect on the score of the phylogenetic hypotheses: if included in the list of acceptable states, it would be ignored throughout the tree cost evaluation.

As an additive character, however, the interpretation is different. Suppose now that the systematist chooses to treat the states as ordered conditions in a continuum, for example by coding red as 1, yellow as 2, and green as 3. If orange were later found occurring in the group of interest, it might be preferable to encode the states of the character with red as 1, orange as 2, yellow as 3, and green as 4, producing an alternative cost regime. If not observed, it would implicitly be included in the character coding scheme.

2. Sequence of loci. Suppose now that we are analysing sequences of loci from the mitochondrial chromosome. For the sake of argument, we assume that all species in the analysis have exactly the same set of loci. The character is the chromosome itself, and the states are represented by the order of loci; it is not the elements included in each state, but their particular order, which is phylogenetically informative. We can also assume that the locus permutations in our sample do not constitute all the potential states, but a fraction of a much larger set, including all possible permutations (super-exponentially many, i.e. $n!$ for n loci). Unlike the morphology example, the mechanisms that could explain such permutations do not include substitutions per se. Instead, the distance between a pair of permutations could be computed using very different mechanisms (e.g. inversions, tandem duplication–random loss). For such a character, the homologies between loci are not tested, but rather the order in which they occur.

3. Nucleic acid sequence. In this example, a particular locus is the character (e.g. 18S rRNA). The states observed are RNA sequences, i.e. words in the {A, C, G, U} alphabet. Although we observe only a small fraction of the words, the states that could have occurred in nature include, in principle, all the possible words of this alphabet: an infinite number of states.

4. Complete chromosome. Suppose now that we are interested in the analysis of a complete chromosome from a group of plants. Assume that we have one complete chromosome for each terminal that is believed to be homologous across the group. Moreover, we have annotated those chromosomes such that the limits of functional units are well established. We will further assume in the analysis that rearrangements, gain, and loss of functional units are possible, but restricted to our predefined limits (i.e. we consider the rearrangement of the two halves of a functional unit to be impossible). However, the correspondences between functional units are uncertain, and we would like to generate them for each phylogenetic analysis.

Unlike the previous two examples, a chromosome state is not defined by a small but an infinitely large

alphabet. Each functional unit could be, potentially, any DNA sequence. This character is the composition of the previous two examples, where DNA sequences are the elements comprising each character state. We are interested in the insertions, deletions, and substitutions occurring between corresponding functional units, and also in the higher level events that modify the order in which these units occur. Clearly, a huge number of possible states is not being observed, yet must be considered in the character coding scheme if we want to produce a meaningful analysis.

Two characteristics should be highlighted from the previous examples.

1. Not all the states need to be observed to be relevant on the analysis. Depending on conditions, states that have not been observed may have no (e.g. as in non-additive characters) or a fundamental effect (e.g. additive, DNA genes as described above).

2. A character could have infinitely many states, describing complex entities, such as the order of the elements composing it. Moreover, there could also be infinitely many possible elements.

We say that *a character C is a set of states, where each state is an ordered set of elements from a predefined alphabet* Σ. In our morphological example, Σ = {red, yellow, green}, and the valid states are ordered sets with only one element, i.e. $C = \Sigma^1$ (⟨red⟩, ⟨yellow⟩, ⟨green⟩; a terminal could have multiple states). In the locus sequence example, the alphabet is the set of mitochondrial genes, i.e. Σ = {CO1, CO2, CO3, ATP6,...}, while C includes all the permutations of the elements in E. In this case, every valid state must include all the genes (i.e. an exponential, but finite number of states). In the sequence character example, the alphabet is Σ = {A, C, G, U}, while the valid states are all the sequences that could be created with it, i.e. $C = \Sigma^*$ (i.e. infinitely many states). In the chromosomal character example, the alphabet itself is Σ = {A, C, G, T}* (i.e. all the words that can be created with {A, C, G, T}), and the valid states are $C = \Sigma^*$. In this case, the alphabet itself has an infinite number of elements.

We are ready to define static homology and dynamic homology characters.

Static homology characters. Let A and B be two states of a character. A *correspondence* between the elements in A and B is a relation between them. We define *static homology characters* as those in which for every element in A there is at most one corresponding element in B, and the correspondence relations are transitive (i.e. let $a \in A$, $b \in B$, and $c \in C$ be elements of different states, where a corresponds to b, and b corresponds to c; then a and c must also correspond to each other). Corresponding elements with the same value match the notion of primary homology (de Pinna, 1991).

Dynamic homology characters. We define as dynamic homology characters (Wheeler, 2001) the complement of their static homology counterparts: for some pair of states A and B, there exists an element $a \in A$ that has more than one corresponding element in B, or the correspondences are not transitive. Dynamic homology characters typically have states that may have different cardinalities, and no putative homology statements among the state elements. These characters formalize the multiple possibilities in the assignment of correspondences (primary homologies) between the elements in a pair of states, which can only be inferred from a transformation series linking the states, and the distance function of choice. A subset of correspondences from dynamic homology sequence character that matches the conditions of static homology characters (i.e. at most one corresponding element, and transitivity) is what De Laet (2004) has called comparable bases. (See the definition of sequence characters below.)

In the first two examples, the correspondences are hypothesized a priori, and tested in the phylogeny. To illustrate this, in the morphology example, the element red in the state ⟨red⟩ corresponds only to the element yellow in the state ⟨yellow⟩; in the sequence of loci example, the occurrence of the subsequence ⟨CO1,ATP6,CO2⟩ in a state can only correspond to a subsequence containing exactly those three elements in another state (e.g. ⟨CO2,CO1,ATP6⟩).

In the later two examples, a hypothesis of correspondence between the elements of a state is based on a particular sequence of intermediate states spanning them. In a phylogenetic context, such intermediate conditions are only sound if defined as hypothetical ancestral states of a tree. To illustrate this case, consider the nucleic acid sequence example. Assume that the following pair of sequences are homologous: AGAGA GAG and GA. To simplify the example, suppose that only insertions, and deletions, could have occurred in the transformation from one sequence into the other. It would be difficult then to define with certainty a set of correspondences between these two sequences prior to a phylogenetic analysis: there are 14 possible correspondence relations between the elements of this pair of states. In static homologies, only one set of correspondences can be selected for the analysis, while under dynamic homologies, multiple correspondences are considered.

Static homology characters. POY 4 recognizes five types of static homology characters: Sankoff, additive, non-additive, breakpoint, and inversion.

Sankoff characters have n valid states, and an $n \times n$ metric distance matrix m such that $m_{i,j}$ holds the distance between state i and state j. The maximum number of states accepted is limited only by the memory constraints of the computer executing POY. Sankoff

characters can be loaded from dpread files (Wheeler et al., 2006), prealigned molecular files, or generated from an implied alignment (see section on "Transformations between character types"). The distance computation between a pair of vectors of states has time complexity $O(n^2)$.

The following two static homology characters (additive and non-additive) are common special cases of Sankoff characters, for which the distance between two vectors of states can be computed in constant time ($O(1)$).

Additive characters allow each state $i \in N$, $0 \leq i \leq 255$, with distance matrix $m_{i,j} = |j - i|$. Additive characters can be loaded from Nona/TNT matrices, or NEXUS files.

Non-additive characters are also known as unordered characters (Fitch, 1971). POY supports up to 30 states in 32-bit architectures, and 62 states in 64-bit architectures. The distance matrix is the Hamming distance (1950):

$$m_{i,j} = \begin{cases} 1 & if\ i \neq j \\ 0 & othewise. \end{cases}$$

Non-additive characters can be loaded from Nona/TNT, NEXUS files, prealigned molecular files, or automatically generated from the implied alignment of dynamic homology characters when the cost of all substitutions is some constant a, and that of all indels is some constant b (see section on "Supported character types").

Breakpoint characters consist of sequences in any user-defined alphabet (known in the POY 4 user interface as custom alphabets). Typically, each element in the alphabet corresponds to a homologous locus. The evolutionary distance between these sequences is computed as the breakpoint distance (Blanchette et al., 1997). Formally, given two permutations $A = \langle a_1 ... \ a_n \rangle$ and $B = \langle b_1 ... \ b_n \rangle$ of elements in some alphabet Σ, we say that every a_i and a_{i+1} are adjacent elements in A (a_1 and a_n are also considered adjacent in circular chromosomes). A pair $x, y \in \Sigma$ is a breakpoint if x and y are adjacent in A but not in B. Given a breakpoint cost c, the breakpoint distance between two sequences A and B *is* $c\beta(A, B)$, where $\beta(A, B)$ is the number of breakpoints in A (and symmetrically in B). Breakpoint characters can be loaded from *custom alphabet* files (Varón et al., 2008). The time complexity to compute the distance between a pair of states is $O(n)$.

Inversion characters consist of sequences in any user-defined alphabet extended with the tilde sign ($\sim$) to represent "inverted" characters, i.e. their reverse complement. Typically, each element is a locus, where loci with the same name are homologous. In this notation, $\sim A$ *is* the inversion of A (i.e. the reverse complement of A) and vice versa. The evolutionary distance between these sequences is the inversion distance (Caprara, 1997). Formally, let $A = \langle a_1 ... \ a_n \rangle$ and $B = \langle b_1 ... \ b_n \rangle$ be a pair of permutations of the same set of elements. An inversion of a subsequence $a_i, a_{i+1}, ..., a_j$ *is* $\sim a_j, ..., \sim a_{i+1}, \sim a_i$, such that $\sim\sim x = x$. Given an inversion cost c, the inversion distance between the permutations A and B *is* $c\iota(A, B)$, where $\iota(A, B)$ is the minimum number of inversions required to transform A into B. Inversion distances in POY are computed using the high-performance functions of GRAPPA (Moret et al., 2002). Inversion characters can be loaded from custom alphabet files (Varón et al., 2008).

Dynamic homology characters. Dynamic homology characters are generically referred to as "molecular" in the POY 4 user interface. Such naming is due to their more common usage with molecular sequences, but the input data need not represent molecular characters. The following dynamic homology character types are supported.

Sequence characters support as valid states any word in Σ^*, from a predefined alphabet Σ (typically $\Sigma = \{A, C, G, T\}$). Sequence characters allow the occurrence of insertion, deletion, and substitution events to calculate the evolutionary distance and correspondences of elements implied by each tree. A deletion of position i in the sequence $s = \langle s_1, ..., s_i, ..., s_n \rangle$ yields the sequence $\langle s_1, ..., s_{i-1}, s_{i+1}, ..., s_n \rangle$. An insertion is symmetric to the deletion. A substitution with element e in position i generates the sequence $\langle s_i, ..., s_{i-1}, e, s_{i+1}, ..., s_n \rangle$. To define the distance function we must first define the set of *edited* sequences. Let $\Sigma_{in} = \Sigma \cup \{indel\}$ be an extended alphabet that includes the placeholder *indel* which does not occur in Σ. The set of edited sequences $ed(A) \subset \Sigma_{in}^*$, $A \in \Sigma^*$, contains all the sequences that can be produced by inserting *indel* elements in A. A transformation cost matrix (*tcm*) is $|\Sigma_{in}| \times |\Sigma_{in}|$ matrix holding the distance between every pair of elements in Σ_{in}. An indel block is a subsequence containing only *indel* elements. Given some constant c and a transformation cost matrix *tcm* such that $tcm(x, y) \in \mathbb{N}$, $x, y \in \Sigma_{in}$ is the cost of transforming x into y, the alignment (or edition) cost between two sequences A and B, $A, B \in \Sigma^*$ of length n containing k maximal indel blocks is $algn(A, B) = ck + \Sigma_{o \leq i < n} tcm(A_i, B_i)$, where A_i and B_i are ith elements in the sequence A and B, respectively. The distance between two sequences C and D is defined as $d(C, D) = \min_{|C'| = |D'|} algn(C', D')$, where $C' \in ed(C)$ and $D' \in ed(D)$ (Fig. 1a, b).

An important difference between POY version 3 and version 4 is the way a non-metric *tcm* is handled. A non-metric *tcm* was not supported in POY version 3, and would produce incorrect results and tree lengths. POY 4 supports non-metricity, provided it is caused by a low (but greater than zero) indel cost. The application issues a warning when non-metric *tcm*'s are being used. This feature, however, does not imply that POY 4 somehow avoids trivial alignments when the indel cost is too low

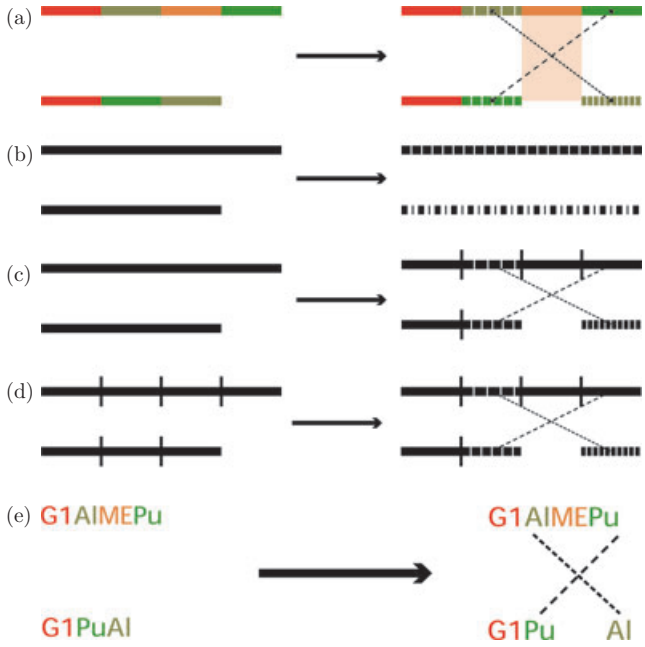

Fig. 1. Homologies potentially inferred by the different classes of dynamic homology characters (excepting genomes), compared with a reference set of transformations. (a) Input sequences on the left and expected homology statements on the right. The sequences present four (upper sequence) and three loci (lower sequence), with indels occurring in the green loci, as well as a locus rearrangement. The orange locus shows an indel event between the two sequences. (b) As sequence characters. Insertions, deletions, and substitutions are inferred. For sufficiently complex sequences, the alignment will expand, trespassing the locus "limits". (c) As raw chromosome characters. With no user-provided limits, POY 4 attempts to infer rearrangements, and locus indels, in addition to sequence insertions, deletions, and substitutions. The program attempts to establish locus limits based on conserved segments. (d) As chromosome characters. With user-provided limits between loci, POY 4 attempts to infer rearrangements and locus indels, as well as sequence insertions, deletions, and substitutions. The program will not attempt to modify the user-provided locus limits. (e) As annotated chromosome characters, employing the user-provided alphabet to represent homologous loci. Only rearrangements, locus indels, and locus substitutions can be inferred directly by the application.

(e.g. AAA— and —TAA). Its main usage is to define a very low indel cost in conjunction with a gap opening parameter (i.e. affine gap costs).

POY 4 also accepts any alphabet: nucleotide (using the complete IUPAC codes, see Liebecq, 1992), amino acid (a subset of the IUPAC codes, see Liebecq, 1992), and user-defined custom alphabets (Varón et al., 2008). Sequence characters can be loaded from FASTA files, NEXUS files with the unaligned block, custom alphabet files, and most file formats produced by GenBank. The time complexity to compute the distance between a pair of states of cardinality m and n is $O(mn)$.

Chromosomal characters have as valid states any word in Σ^*, where $\Sigma = \{A, C, G, T\}$. Each element of a state represents a chromosomal fragment, and each fragment a nucleotide sequence character itself. Chromosomal characters can detect fragment inversions, fragment rearrangements, and fragment indels, along with the familiar sequence-level insertions, deletions, and substitutions within the segment. The distance computation is done in two steps: a pairwise alignment at the fragment level, under the user-provided parameters, followed by a rearrangement distance computation using the functions provided by GRAPPA (Moret et al., 2002). The selection of homologous segments is heuristic and is described elsewhere (Vinh et al., 2006).

Segment limits can be specified or inferred in three different ways, yielding three different character types.

Automatic segment detection uses complete unaligned *nucleotide* sequences. During the tree cost computation, the sequences are divided into distinctly conserved regions (blocks), according to the user-provided parameters. The blocks can then be subjected to rearrangement events, which are heuristically detected (Fig. 1c) (Vinh et al., 2006). The distance computation consists of the following steps: detection of potentially homologous regions, computation of their pairwise distance using pairwise alignments, removal of inserted segments (segments that have no homologues), and rearrangement computation using breakpoint or inversion distance through GRAPPA (Moret et al., 2002). This type of character can be loaded from the same file types supported for sequence characters.

Partitioned chromosomes where the user divides *nucleotide* sequences using the pipe symbol (|) in the input sequences. The program does not automatically detect blocks in this case, but employs those defined by the pipes. Rearrangements, inversions, and segment indels are detected (Fig. 1d) (Vinh et al., 2006). The distance computation consists of a pairwise alignment of the user-provided segments, detection of homologous segments according to the user-provided parameters, removal of inserted segments, and rearrangement distance calculation using breakpoint or inversion distance through GRAPPA (Moret et al., 2002). Partitioned chromosomes can be loaded from FASTA files, where each fragment is delimited with a pipe sign (|).

Annotated chromosomes where the user assigns *a name* to each individual locus. Loci with shared names are considered homologues. Employing this user-defined alphabet, locus indels and rearrangements can be detected (Fig. 1e). The distance calculation continues as in partitioned chromosomes, with the difference that elements with the same name are assumed to be homologous and no homology detection is required. Annotated chromosomes can be loaded from custom alphabet files.

Rearrangement distances can be computed using the breakpoint distance (Blanchette et al., 1997), or the inversion distance (Caprara, 1997), computed by

GRAPPA (Moret et al., 2002). POY 4 supports both linear and circular chromosomes, but not mixtures.

Genome characters are defined as sets of chromosomes. For this type of character, there is no implied order for the chromosomes, and therefore the user input order is irrelevant. POY automatically detects homologous chromosomes, and considers chromosomal insertions and deletions, along with those events occurring within a chromosomal character as described in the previous section. Genome characters can be loaded from FASTA files, where each chromosome is delimited with the @ sign.

Tree cost calculation

Well-known algorithms are used for the three most commonly used static homology characters: the cost of trees with non-additive (Fitch, 1971) and additive (Farris et al., 1970) characters is computed in $O(nm)$ time complexity, where n *is* the number of nodes in the tree and m *is* the number of characters. The cost calculation for trees with Sankoff characters (Sankoff and Rousseau, 1975) has time complexity $O(nms^2)$, where s is the maximum number of character states. These algorithms yield exact tree costs and an optimal assignment to the interior nodes. For breakpoint, and inversion characters, the tree cost calculation is heuristically approximated, with an overall time complexity of $O(nm)$, where n is the number of nodes in the tree and m *is* the cardinality of the breakpoint or inversion states.

The tree cost calculation for dynamic homology characters, i.e. sequence, chromosome, and genome characters, is at least NP-Hard (e.g. Wang and Jiang, 1994). POY 4 implements a number of heuristic algorithms to bound the tree cost. These algorithms can be divided into two classes: initial assignment to the interior nodes of the tree, and iterative improvement to refine the total cost calculated for that tree.

Initial assignment. The initial assignment is similar in spirit to the *down-pass* in static homology algorithms (e.g. Fitch, 1971). During the diagnosis of an input tree with n terminals, POY 4 computes $2n$—3 implied alignments, one for each possible root (i.e. the alignments inferred for each possible rooted tree from the initial unrooted tree). From these, the best alignment (i.e. the one yielding the lowest tree cost) is assigned to the tree. Each tree can only have one alignment assigned.

Sequence characters. There are three basic algorithms for an initial tree cost calculation in POY 4: fixed states (Wheeler, 1999) (similar but stronger than the Lifted Assignment of Wang et al., 1996), direct optimization (Wheeler, 1996), and affine direct optimization (Varón et al., 2009). The first is a two-approximation method of time complexity $O(n^3)$. As currently implemented, fixed states yields tighter results (i.e. better tree costs) for molecular characters with amino-acid or large user-defined alphabets (more than six elements), and therefore is the recommended heuristic for those character types.

Direct optimization and affine direct optimization have time complexity $O(nms^2)$, where s is the maximum state cardinality. These algorithms yield tighter results for nucleotide alphabets or small user-defined alphabets (fewer than seven elements). Direct optimization is used when the gap opening parameter is 0, otherwise affine direct optimization is employed.

Chromosomal and genome characters. Within the chromosomal types, a set of k medians is heuristically selected and maintained at each node, where k is a user-provided parameter. With larger k, more medians are maintained. Each median is created using a randomized greedy algorithm, and improved using a local search, rearranging each median to produce a new one of lower cost, until no better can be found (Vinh et al., 2006).

Iterative improvement. Once an initial character assignment is performed, POY can iteratively improve the overall tree cost by adjusting the characters of each interior node, based on the corresponding characters assigned to its three neighbours. The adjustment of the characters on each node can occur with two possible methods: using the same techniques of the initial assignment, or an exact three-dimensional alignment.

Approximate uses the initial assignment algorithm of each character to pick a better median for each interior node. On every iteration, POY produces three potential medians, corresponding to the three possible directions to compute the initial assignment algorithm (Varón et al., 2009) (Fig. 2). This method is supported in all the dynamic homology characters.

Exact performs a complete three-dimensional alignment of the three neighbour sequences of an interior node, and creates an optimal median which is the new sequence of the node (Sankoff et al., 1976; Wheeler, 2003). This method is supported only in nucleic acid sequence characters.

The two methods can be applied until one of the following two conditions occurs: no further tree cost

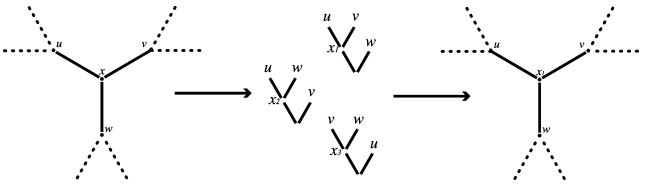

Fig. 2. An iteration of the approximated iterative improvement. To improve x, DO or affine-DO is used to produce x_1, x_2, and x_3, in the three possible rooted trees with terminals u, v, and w. If the best assignment x_1 yields a better score than the original x, then it is replaced, otherwise no change is made.

improvement can be made, or a user-specified maximum number of iterations is reached. The selected method is applied to all the dynamic homology nucleotide sequences.

Phylogenetic tree search

POY 4 provides numerous algorithms for heuristic searches of the most parsimonious tree. To simplify the exposition, in the time complexity description of the following algorithms we will assume that the computation of the character distances and interior nodes of the trees takes constant time. Due to implementation details, most of the algorithms mentioned below have a $O(\log n)$ overhead factor, where n is the number of terminals. In a modern analysis, this factor is typically small compared with the number of characters and sequence lengths. Nevertheless, it will be eliminated in a future version of POY.

Initial tree building. Every heuristic search algorithm requires a method to generate the initial set of trees. POY 4 includes three main methods: branch and bound (Hendy and Penny, 1982), Wagner tree building (Farris et al., 1970), and minimum spanning tree guided.

Branch and bound. This method of tree building provides, in principle, an exact solution to the phylogeny problem (Hendy and Penny, 1982). Unfortunately, this is only true if the calculation of the tree cost is exact, something that cannot be guaranteed for some character types. Therefore, if a user builds a tree using branch and bound, the solution is exact up to the goodness of the tree cost algorithm. The overall time complexity of branch and bound remains exponential in the number of terminals, and therefore it is only recommended for data sets with a very small number of terminals.

Wagner tree. The Wagner algorithm (Farris et al., 1970) uses a greedy strategy to create an initial tree, by iteratively connecting a terminal to the tree in the best position. Due to its greedy nature, the algorithm is sensitive to the order in which the terminals are added. This order-dependency is used as a heuristic to visit a larger portion of the tree space, limited to ''sound'' trees. By default, when using this algorithm, POY randomizes the terminal addition sequence. The overall time complexity of the implementation of this algorithm is $O(n^2)$.

Minimum spanning tree guided. A third strategy available in the application is the use of a minimum spanning tree (MST) (Cormen et al., 2001). An MST generates a sequence of terminals that can produce better results compared with a single, randomized, Wagner tree algorithm. Unfortunately, this method has limited use in real data sets, where the distance between terminals is usually not metric due to polymorphisms and sample errors, and randomization is used with a larger number of repetitions to improve the overall search results. The overall time complexity of the algorithm is $O(n^2)$.

Additionally, POY 4 provides methods to build trees with positive constraints, i.e. build trees where certain clades are required to exist. These methods can be applied together with any of the Wagner tree or the minimum spanning tree building strategies previously described. Negative constraints will be supported in a future version.

Local search strategies. The local search consists of the iterative modification of a current tree, in an attempt to find a similar tree of better score. POY supports a number of algorithms, classified in the various components that they involve for a local search: neighborhood, trajectory, branch break order, and join method. Additionally, the trees visited during the search can be *sampled* (e.g. to collect trees for Bremer, 1994, support).

Neighborhood. The neighborhood describes those trees that can be evaluated, given the current best tree. These are known as the neighbours of the current best, hence the name. POY supports nearest neighbour interchange (NNI), sub-tree pruning and regrafting (SPR), and tree bisection and reconnection (TBR) (see Felsenstein, 2004, for a survey of these algorithms). These sets can be limited further using a positive constraint (an unresolved tree that shows clades that must be present in a neighbour). Every neighbourhood in POY 4 consists of successive branch breaks, joins, reroots (in TBR), and the trajectory of the search (i.e. the tree that is selected for the next iteration). Each can be fine tuned, as follows.

1. Branch break order. POY includes algorithms to break the branches in decreasing length order (`distance`), fully randomized breaking order (`randomized`), to break only once, and never again, even if the local optimum has changed (`once`). By default, the distance method is employed.

2. Join specifies those branches that can be joined and in what order. The options available include `constraint` to specify either a sectorial search or a tree that constrains possible solutions to the problem, `all` to turn off all the heuristics used by the program to reduce the number of trees evaluated during a local search, and `sectorial` to specify sectorial searches constrained by the subtree size.

3. Rerooting specifies the roots that can be used during TBR. By default, the order in which roots are visited follows a breadth-first search algorithm on the branches (Cormen et al., 2001), starting at the nodes incident in the broken branch. The number of trees evaluated at this step can be limited with the `bfs` argument, specifying the maximum distance allowed for each new root from the initial root. The distance is defined as the number of branches in the path connecting the new with the original root.

4. *Trajectory* specifies how the program selects the next neighbouring tree to be evaluated. The default algorithm is a greedy *first best*, which selects the first tree found that has better score than the current best, `around` to evaluate completely the neighbourhood before selecting the next local optimum, simulated annealing (`annealing`) (Kirkpatrick et al., 1983), which uses a probabilistic function to choose a tree, and tree drifting (`drift`) [a modified version from that described by Goloboff (1999b; Varón et al., 2008)].

Samplers. As the local search is executed, POY 4 provides various *sampler* methods, to allow users to collect information, either for error recovery, support calculations, or analytical purposes. For instance, all trees that have been visited during a search can be printed out with the visited argument.

Escaping local optima. Local searches are often not sufficient to generate satisfactory solutions. A number of algorithms exist to escape locally optimum solutions; POY 4 supports two main classes: tree fusing and search space perturbation.

Tree fusing is described by Goloboff (1999b) to find better trees in complex data sets. The basic algorithm consists of selecting pairs of trees uniformly at random; the first is considered the source and the second the target. These trees are compared, and for all pairs of compatible subtrees, the subtree in the source replaces the corresponding subtree in the target. (A pair of subtrees is compatible if both contain the same set of terminals, but their topologies differ.) If the best tree resulting from this exchange has a lower score than the target, then this new tree replaces the target. This procedure is repeated for a user-determined number of iterations. The algorithm can be tuned, by selecting a local search strategy to follow the new subtree selection, as well as the number, and algorithm to select trees that are maintained between iterations.

Perturbation is a basic strategy that allows the user to perform a local search (or a series of local searches) on a modified set of characters. The tree space (i.e. the space representing the cost of each tree) is therefore "perturbed", and depending on the perturbation method, could help the search by escaping locally optimum trees and finding better solutions. The most notable form of perturbation is the parsimony ratchet (Nixon, 1999). The basic ratchet algorithm consists of perturbing the tree space by reweighting a random set of characters, according to user-provided parameters, followed by a local search, and the resulting tree is used in a new iteration. When the user-selected number of iterations is completed, the search space is restored, and a new local search proceeds. The original tree is replaced with the final only if better. Along with the parsimony ratchet, all the transformations (including those described in the section on "Transformation between character types") are supported as perturbation methods.

Search command. POY 4 introduces a new command: `search`. It is intended as a default search strategy for most users. This strategy includes tree building using the Wagner algorithm (Farris et al., 1970), swapping using TBR, swapping using exhaustive direct optimization (Varón et al., 2008), Nixon's parsimony ratchet (1999), and tree fusing (Goloboff, 1999b). The command supports arguments to specify the maximum or minimum execution time, minimum number of hits before stopping, and the maximum number of trees to be held (measured in memory). The function takes care of removing duplicated trees and reducing repeated effort. Upon completion, it reports the number of trees built, the number of rounds of tree fuse, the best tree cost found, and the number of times that cost was found (hits).

Search is *a recommended way to execute an analysis*. It does not eliminate the user responsibility to ensure that a reasonable tree search is performed for the input data set. It is important to verify that several searches converge to the minimum cost (i.e. maximize the "hits"), and a reasonable number (of the order of hundreds) of replications are performed (each tree fuse can be considered a separate replicate).

User interface

Previous versions of POY consisted solely of a command line application, with very limited flexibility in the kinds of analysis and parameters that could be chosen by the user. Version 4 has several user interfaces that can be selected according to the user preferences (e.g. the requirements are different when executing a complete analysis on a computer cluster, or learning how to use the application on a personal computer).

POY 4 is an interactive application. Users can issue commands and obtain an immediate response. This behaviour eases the learning curve for new users, provides a friendly environment to test input data and analysis conditions before executing a major analysis, and reduces the likelihood of errors in the input data, by allowing users to "explore" before executing a complete analysis.

Along this line, a simpler set of commands has been defined, allowing users to perform complex analyses and heuristic searches, with fewer commands. The complete grammar is described in the user manual (Varón et al., 2008). For example, Fig. 3(a) shows a script to read an input file, build ten trees, perform a local search, fuse them, and report the results. If the fuse step should use SPR instead of TBR (the default) for a local search, then the script can be modified easily to achieve this effect (Fig. 3b).

(a)
```
read ("file.fas")
build ()
swap ()
fuse ()
report (trees)
```

(b)
```
read ("file.fas")
build ()
swap ()
fuse (swap (spr))
report (trees)
```

Fig. 3. Two scripts that read an input file, build ten trees, swap to find the optimum, fuse, and report the results in parenthetical notation. (a) Using the default parameters. (b) Using SPR to improve fuse.

Notice that the new structure increases readability, using a simple pattern of a verb (the command) followed by arguments for the command in parentheses.

A complete description of the various user interfaces as well as practical examples are available in the program manual (Varón et al., 2008).

Script execution

POY 4 accepts files containing scripts for non-interactive execution. A script is a sequence of valid POY 4 commands. The execution of scripts in POY 4 does not necessarily follow exactly the input order specified by the user. Instead, a script is analysed and modified to achieve the same analytical effort (measured in number of trees evaluated, randomized procedures executed, etc.), while reducing memory consumption, and limiting the amount of information exchanged between processes when executing in parallel.

To understand the script execution better, we must first describe the parallelization strategy used in POY 4, followed by the description of the script analysis and optimization methods employed in the application.

Parallel model

POY 4 supports parallel execution using any implementation of the Message Passing Interface (MPI) version 1.0. MPI has become the most important standard for parallel execution using Message Passing. By using MPI, POY 4 can be executed in parallel under virtually any architecture, from laptops with multiple cores, to computer clusters running Linux, Windows, or Mac OS X.

The parallelization model used in POY 3 consisted of a master–slave model of computation, where one process (the master) directed other processes (the slaves) to perform certain calculations upon request. For instance, if ten trees were to be built using the Wagner algorithm, and 11 processes were available, then the master would order each of the ten slaves to perform one of the builds. During most of the computation, however, the master would remain idle, waiting for requests from the slave processes.

The parallel model of POY 3 posed significant scalability difficulties. Even for fast networks, if sufficient processes attempted to communicate concurrently, the master process was a bottleneck, producing sub-linear scalability and even reduced performance under a number of circumstances (Janies and Wheeler, 2001; Wheeler et al., 2003). To solve this problem, POY 3 included ''controller'' processes, which could serve as intermediate relays, responsible for managing a smaller number of slaves (Janies and Wheeler, 2001). Although the scalability limitations could be reduced in this way, the problem remained at a larger scale, while increasing the number of idle processes overall.

POY 4 is fundamentally different in that there is no process directing the computation of any other process. Instead, upon receiving the input script, each process independently decides what tasks it should perform. There exists a master process, which performs the same operations that other processes would, but also centralizes access to input files when other processes cannot directly (as in some computer clusters), and generates the desired program output (e.g. printing the trees in a file).

The fundamental advantage of this parallelization model is the increased scalability and the reduced volume of communications. Moreover, resources are better exploited, by eliminating an idle process (the master), which can instead spend resources on the analysis itself. It follows that POY 4 can scale even in computers with two cores, as both processes are responsible for part of the complete analysis.

There are two fundamental limitations in POY 4's model: fault tolerance has been eliminated, as have the parallelization of the operations within a tree (e.g. parallel building of a single tree). The former has a lower priority, but the latter will be included in future releases of the application.

Script analysis

A script analysis consists of three steps: dependency analysis, memory optimization, and parallel execution division.

Dependency analysis. In the first step, POY 4 analyses the data dependencies between different components of a script. For example, the calculation of the jackknife support value information is independent of the search for the most parsimonious tree (but not assigning the support values to the shortest tree found). POY 4 evaluates mutual dependencies in input files, output files, trees, jackknife frequencies, bootstrap frequencies, and BREMER supports, to produce a dependency graph that describes how commands relate to each other.

Memory optimization. Once the dependency analysis is completed, POY 4 classifies each command in the script into one of four classes that allow the application to optimize their execution:

Parallelizable is a command that can be executed in parallel. Examples of commands of this class are `build` and `swap`.

Composable is a command that can be applied composed over intermediate results, yielding exactly the same output as if it was applied once over all the results directly. For example, selecting the shortest tree among ten trees has the same effect as selecting the best tree among the first two, then selecting the best between the result of the previous selection and the third tree, and so on until all the trees are evaluated. An example from this class is `select` (`best`).

Linearizable is a command that can be applied independently with subsets of results, yielding the same effect as applying it to all the results (Fig. 4).

Non-composable are commands that cannot be parallelized, and set hard limits in the way a script is executed. An example of this class of commands is `report` (`treestats`).

Script execution is modified in the following manner: parallelizable, composable, and linearizable commands can be modified to improve performance, conforming to *pipelines*, while non-composable commands break the pipelines. To understand how these pipelines are formed, we will illustrate them using an example.

Figure 5 shows a script that can be described as follows: read an input file, build 1000 trees, swap each until its local optimum is found, redraw the screen, select the best trees and filter out duplications, report the remaining trees to the screen in graphical format, and quit the application. If executed in this way, at peak memory consumption, POY 4 would require enough memory to hold 1000 trees.

```
swap ()
redraw ()
exit ()
```

Fig. 4. The redraw command to refresh the screen contents. It would have the same effect as executing it once after all the trees have been swapped, or each time a tree is swapped. This type of command yields a greater execution order flexibility.

```
read ("file")          (* Non-Composable *)
build (1000)           (* Parallelizable *)
swap ()                (* Parallelizable *)
redraw ()              (* Linnearizable  *)
select ()              (* Composable     *)
report (graphtrees)    (* Non-Composable *)
quit ()                (* Non-Composable *)
```

Fig. 5. A POY 4 script, with comments showing the type of each command.

If we look at the same script considering the class each command belongs to, a different picture emerges. The core of the script is parallelizable, linearizable, and composable. It follows that this script could be executed more efficiently in the following way: read the input file, and repeat 1000 times the following three steps: build one tree, swap, redraw the screen, and select the best trees in memory. Upon concluding the 1000 repetitions, report the remaining trees in memory on screen in graphical format, and quit the application. Overall, POY 4 will only use as much memory as the maximum number of shortest trees found at the same time. For most real data sets, this will tend to be a small number.

Note that the 1000 iterations involve a sequence of four commands. Each sequence is the "pipeline" mentioned above. The user interface updates the overall script execution progress, and estimates termination time for the set of pipelines instead of individual commands.

Parallel execution division. Note that in the previous example, each pipeline can be executed independently of the others, with the results being merged by the composable elements of the pipeline. Pipelines are the script components that are parallelized by POY 4.

If the previous script is executed in parallel with 1000 processors, each processor would have taken care of a single pipeline, and the selection of the shortest trees would have followed with only 11 ($\lceil \log_2 1000 \rceil$) rounds of messages between processors.

The general rules for parallelization are as follows:

1. Only the master process can print to files or screen.
2. Pipelines and support calculation pseudo-replicates are divided among all processes. If there are m processes and n pipelines, each process does at most ($\lceil n/m \rceil$) pipelines, to complete exactly n.
3. All processes synchronize execution at the end of each pipeline.

Using this strategy, the application shows linear scalability in the number of processors and number of trees evaluated (Fig. 6). The exact execution strategy of a particular script can be verified using the `report` (`script_analysis: "script.poy"`) command (Fig. 7).

Other features

There are many other new features in the program. The following are several highlighted functions.

Transformation between character types

POY 4 supports functions for the easy transformation of character types. For example, suppose a user would

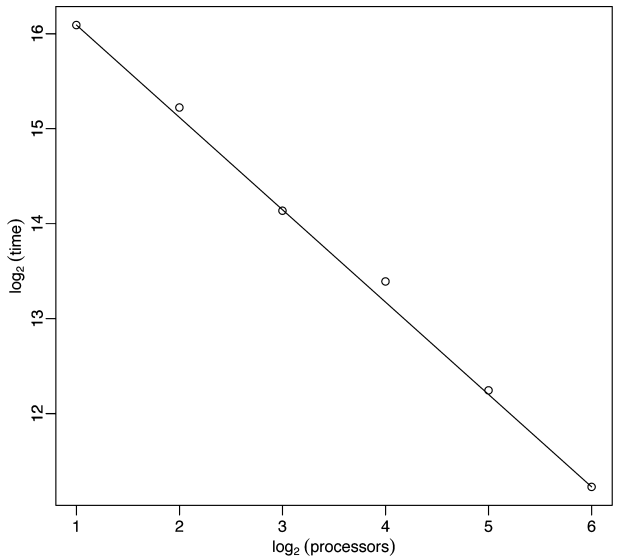

Fig. 6. POY 4 scales linearly in parallel execution. In this example, 64 RAS + TBR were tested with one to 64 processors in parallel. The speedup is linear in the number of processors, with a slope of ≈ 0.9.

like to run an analysis for one complete day, select the best tree, fix the alignment of the dynamic homology sequences implied by the best tree found, and compute the jackknife support values using the characters inferred by that alignment. In this example, we will assume that the program found only one tree during the analysis. The script in Fig. 8 shows this approach. Another example would show the effect that different alignment parameters may have on the implied alignment of a particular tree (Fig. 9).

An important effect of the `static_approx` transformation shown in Fig. 9 is the way it is performed for other distance functions. For instance, assume the user has defined an affine distance function, with cost two for every substitution, four for gap opening, and 1 for gap extension. (The total cost of an indel of length three would be seven in this example.) After performing `transform` (`static_approx`), the program will create characters corresponding to the individual columns of the implied alignment, representing the substitution events, and characters of different weight representing the individual indel blocks as inferred from the best tree in memory. For each indel block character, the program stores as the state names the sequence block that was inserted (or deleted) (e.g. an inferred insertion of the block AACTTG will have state names AACTTG and '-' representing the presence of the block, and its absence). All the characters can then be exported in Nona/TNT format for use in other programs, and generate the

```
beginning of the program
read an input file
I will calculate the following in separate processors (if available)
processor group 1:
  in parallel:
    build some trees from scratch
    swap the trees in memory
    while keeping the following invariant:
      eliminate repeated trees
      select the optimal trees
  eliminate repeated trees
  select the optimal trees
processor group 2:
  redraw the screen
  output the trees in memory
  close POY
```

Fig. 7. Analysis of input script in Fig. 5 as generated by POY 4 using the `script_analysis` report.

```
(* We read three genes from three different fasta files *)
read ("input_file.fasta", "input_file2.fasta", "input_file3.fasta")
(* We will use substitutions and indels with cost 1 *)
transform (tcm:(1,1))
(* We will now search for one day, zero hours, zero minutes *)
search (max_time:1:0:0)
(* Select the shortest trees, and remove duplicated trees *)
select ()
(* Fix the alignment to compute the support values *)
transform (static_approx)
calculate_support (jackknife)
(* Output the best tree found and support values *)
report (trees, supports:jackknife)
```

Fig. 8. Computing the jackknife support values on the implied alignment of the best tree found after one day of search.

```
(* We read three genes from one fasta file, and a tree  *)
read ("18S.fasta", "input.tree")
(* We will use substitutions and indels with cost 1 *)
transform (tcm:(1,1))
(* Output the implied alignment on screen *)
report (implied_alignments)
(* Repeat the procedure with various alignment parameters *)
transform (tcm:(2,1))
report (implied_alignments)
transform (tcm:(3,1), gap_opening:2)
report (implied_alignments)
transform (tcm:(6,1), gap_opening:5)
report (implied_alignments)
exit ()
```

Fig. 9. Comparing the effect of various alignment parameters in the alignment implied by the same phylogenetic tree and the same locus.

apomorphy list [using the command `report (phastwinclad)`].

Input file formats

POY 4 supports the data and tree input and specification of Nona/TNT files, NEXUS files (Maddison et al., 1997), all GenBank sequence formats (the most commonly used is FASTA), as well as CLUSTAL file formats (Thompson et al., 1994). The program honours character and state names in the input, and all reports use them accordingly. This ensures that the user will be able to employ the application output more efficiently for publication.

Graphical output

Along with newick formatted trees, POY 4 can output graphical trees in PDF format, allowing modification in vector image editors, for screen and print layout.

Support calculation

POY 4 supports bootstrap, Bremer, and jackknife support calculations. It is important to note that bootstrap and jackknife will resample characters as listed by the command `report (data)`: that is, if dynamic homology characters are loaded, the characters themselves are sampled (e.g. the sequences), and not the elements within each state (e.g. the bases).

What the program cannot do

Although flexible, POY 4 has a number of limitations. The following are the most important functions that the program cannot do, yet users may assume.

Automatically detect non-homologous sequences. At the input level, the user provides a set of states that are believed to be homologous (e.g. fragments from the 5S subregions for all the species). For this reason, *in dynamic homology characters, any pair of elements from homologous states can be hypothesized homologous*. This assumption has two main implications:

1. The program cannot detect incomplete sequences and treat them as such. If the sequences are incomplete, the user may follow one of the procedures suggested by Wheeler et al. (2006) to treat sections as missing data, or simply accept that this will be a source of noise.
2. If a pair of sequences are random relative to each other (for example when reverse complements are included in the analysis), the program will do something with them, no matter how little sense that may make. The software is designed to help, but it is not a goalkeeper.

Automatically select alignment parameters. Parameter selection is an important problem in phylogenetic analysis. POY 4 provides functions to apply different parameters to each character, and assign default parameters in every analysis, but users must be careful in this respect.

Detect inversions and breakpoints in the same character. The functions provided in version 4 can detect either inversions, or translocations, or rearrangements in a particular character, but neither two nor three can be detected simultaneously on the same character. However, POY 4 can simultaneously analyse multiple characters, each of a different type, and using different parameters. It is possible then to detect inversions in one character, while rearrangement or translocations are detected in others during the execution of a combined analysis (see section on ''Supported character types'').

Maximum likelihood. POY 3 supported phylogenetic analyses using maximum likelihood (ML) as optimality criterion. This optimality criterion is not supported in POY versions 4.0 to 4.1.2. However, ML is currently in development and will be supported again soon in a future release of the software.

Performance example

As an example of the overall application performance, and as a comparison between POY versions 3 and 4, a random subset of 100 published anurans (Faivovich et al., 2005) was analysed. The data set includes 12S rRNA, tRNA valine, 16S rRNA, and fragments of cytochrome b, rhodopsin, tyrosinase, 28S rRNA, and RAG 1, and a small set of 38 morphological, non-additive characters.

To compare the performance of POY version 3 and version 4, we performed 1000 independent repetitions consisting of one randomized Wagner build (section on "Phylogenetic tree search"), followed by a default local search using TBR on the tree initially constructed (section on "Phylogenetic tree search"), and reported the resulting tree cost. This procedure can be executed in POY 3 with the command:

```
poy -replicates 1 -seed -1 -maxtrees 1
-nooneasis -minterminals 0 -terminalsfile
ranNamesPH.txt *.fas *.ss.
```

In POY 4, the script would be:

```
read ("*.fas", "*.ss")
select ("ranNamesPH.txt")
build (1)
swap ()
report (treestats)
exit ()
```

The scores of the trees found by each program were plotted in a density histogram (Fig. 10). The results show that one repetition of the previous procedure in POY 4 outputs a tree which is expected to belong to the top 15% of the best trees found by this very simple search strategy. For POY 3 to expect a tree within the same percentile, it would be necessary to run more than 2000 repetitions. It follows that due to heuristic improvements, POY version 4 is more than 2000 times faster than POY 3.

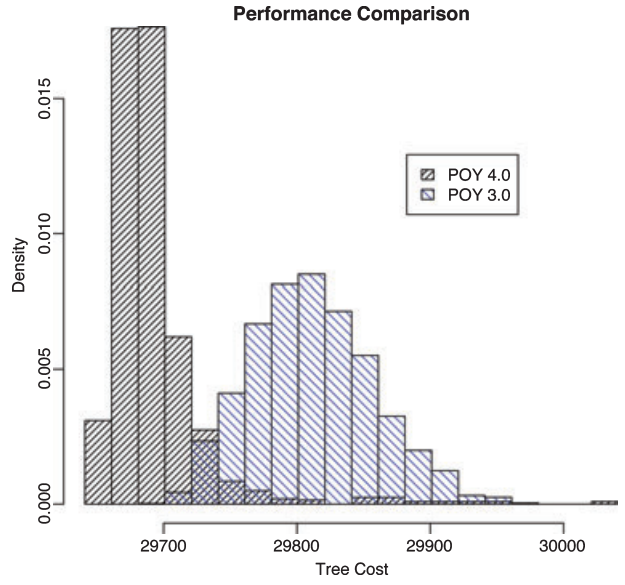

Fig. 10. Density histogram of the frequency of occurrence of different tree costs in POY version 3 and version 4 for the example data set.

To evaluate the execution time spent on each iteration, we allowed each application to perform the previous simple search as many times as possible within a 24-h time boundary. POY 3 could perform 28 repetitions and POY 4 could perform 38 repetitions. That is, POY 4 performs roughly 30% more Wagner tree builds and local searches than POY 3.

Considering these results, we can see that, for this particular data set, POY version 4 is more than 2600 times faster than POY 3. To put these numbers in perspective, the expected results of an analysis that could be previously performed in a cluster of more than 5000 processors using POY 3 can now be done on a personal computer with a dual core processor.

When we analyse this data set in the same 24-h time limit using the `search` (`max_time:1:0:0`) command with only one processor, POY 4 found the shortest tree four times (29 649 steps), a result that was not possible to reproduce with POY 3.

Program resources, availability, distribution, and licence terms

Obtaining the program

The released versions of POY 4 can be freely downloaded from http://research.amnh.org/scicomp/projects/poy.php as binaries and source code. The bleeding edge development version and bug tracking system can be found at http://code.google.com/p/poy4/.

Binaries for sequential and parallel execution are available for Microsoft Windows XP, Vista, and Mac OS X Tiger and Leopard (Universal binaries). Binaries for sequential execution in Linux ×86 are also available for download. For all other architectures, users can download the source code and compile themselves. POY 4 is highly portable, and has been successfully compiled in Linux AMD-64 and Itanium2, AIX, Sun OS, and Solaris, both for sequential and for parallel execution. For parallel environments of individual workstations or computers clusters, it is necessary to use any of the available implementations of the MPI version 1.0.

Licence

POY 4 is open source, distributed under GPL v. 2, written in OCaml and C. Inversion and breakpoint distance functions are provided by GRAPPA (Moret et al., 2002) (http://www.cs.unm.edu/~moret/GRAPPA/). PDF generation is provided by Camlpdf of Coherent Graphics (http://www.coherentgraphics.

co.uk). The three-dimensional alignment functions of versions 4.0 and 4.1 are provided using software from David R. Powell (Powell et al., 2000), (ftp://ftp.csse.monash.edu.au/software/powell/README.html).

Acknowledgements

Many thanks to all the people who tried POY 4 in all of its development stages, gave us, and continue providing excellent comments and bug reports. We would like to thank, in particular, Illya Bomash, Megan Cevasco, Louise Crowley, Torsten Dikow, Julian Faivovich, Gonzalo Giribet, Taran Grant, Christian Kehlmaier, Frederic Legendre, Nicholas Lucaroni, Kurt M. Pickett, Fernando Marques, Paola Pedraza-Peiialosa, Leo Smith, and Ilya Temkin for their comments, which have helped to improve the application. We would also like to thank Lone Aagesen, Pablo Goloboff, and an anonymous referee for their valuable comments regarding the manuscript. A.V., L.S.V. and W.C.W. were supported by the US Army Research Laboratory and the US Army Research Office [W911NF-05-1-0271]. A.V. and W.C.W. were also supported by the NSF-ITR grant "Building the tree of life: A national resource for phyloinformatics and computational phylogenetics" [NSF EF 03-31495]. W.C.W. was also supported by the National Science Foundation grants "An Integrated approach to the origin and diversification of protostomes" [NSF DEB 05-31677] and "Assembling the tree of life: phylogeny of spiders" [NSF EAR 02-28699].

References

Blanchette, M., Bourque, G., Sankoff, D., 1997. Breakpoint phylogenies. In: Proceedings of the Genome Informatics Workshop VIII. eds: Takagi, T. and Miyano, S. Universal Academy Press, Tokyo, pp. 25–34.

Bremer, K., 1994. Branch support and tree stability. Cladistics, 10, 294–304.

Caprara, A., 1997. Sorting by Reversals is Difficult. in RECOMB '97: Proceedings of the First Annual International Conference on Computational Molecular Biology. ACM, New York, NY, USA. pp. 75–83.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., 2001. Introduction to Algorithms, 2nd edn. The MIT Press, Cambridge, MA.

De Laet, J.E., 2004. Parsimony and the problem of inapplicables in sequence data. In: Parsimony Phylogeny and Genomics. ed: Albert, V.A. Oxford University Press, Oxford, pp. 81–116.

Faivovich, J., Haddad, C.F.B., Garcia, P.C.A., Frost, D.R., Campbell, J.A., Wheeler, W.C., 2005. Systematic review of the frog family Hylidae, with special reference to Hylinae: phylogenetic analysis and taxonomic revision. Bull. Am. Mus. Nat. Hist. 294, 240.

Farris, J.S., Kluge, A.G., Eckhardt, M.J., 1970. A numerical approach to phylogenetic systematics. Syst. Zool. 19, 172–189.

Felsenstein, J., 2004. Infering Phylogenies. Sinauer Associates, Sunderland, MA.

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

Goloboff, P., 1999a. Nona (no name). http://www.cladistics.com.

Goloboff, P.A., 1999b. Analyzing large data sets in reasonable times: solutions for composite optima. Cladistics, 4, 415–428.

Goloboff, P.A., Farris, J.S., Nixon, K.C., 2008. TNT, a free program for phylogenetic analysis. Cladistics, 24, 5.

Hamming, R.W., 1950. Error detecting and error correcting codes. Bell Syst. Tech. J. 26, 147–160.

Hendy, M.D., Penny, D., 1982. Branch and bound algorithms to determine minimal evolutionary trees. Math. Biosci. 60, 133–142.

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

Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P., 1983. Optimization by simulated annealing. Science, 220, 4598.

Liebecq, C., (ed.), 1992. Biochemical Nomenclature and Related Documents, 2nd edn. Portland Press, London.

Maddison, D.R., Swofford, D.L., Maddison, W.P., 1997. NEXUS: an extensible file format for systematic information. Syst. Biol. 46, 590–621.

Moret, B.M.E., Bader, D.A., Warnow, T., 2002. High-performance algorithm engineering for computational phylogenetics. J. Supercomputing, 22, 99–111.

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

de Pinna, M.C.C., 1991. Concepts and tests of homology in the cladistic paradigm. Cladistics, 7, 367–394.

Powell, D.R., Allison, L., Dix, T.I., 2000. Fast, optimal alignment of three sequences using linear gap costs. J. Theor. Biol. 207, 325–336.

Sankoff, D., Rousseau, P., 1975. Locating the vertices of a Steiner tree in an arbitrary space. Math. Program. 9, 240–246.

Sankoff, D., Cedergren, R.J., Lapalme, G., 1976. Frequency of insertion-deletion, transversion, and transition in the evolution of 5S ribosomal RNA. J. Mol. Evol. 7, 133–149.

Swofford, D.L., 1993. PAUP: Phylogenetic Analysis Using Parsimony, Ver. 3.1.1. Smithsonian Institution, Washington, DC.

Thompson, J.D., Higgins, D.G., Gibson, T.J., 1994. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions—specific gap penalties and weight matrix choice. Nucleic Acids Res. 22, 4673–4680.

Varón, A., Vinh, L.S., Bomash, I., Wheeler, W.C., 2008. POY 4.0.2900. http://research.amnh.org/scicomp/.

Varón, A., Wheeler, W., Bar-Noy, A., 2009. An efficient heuristic for the tree alignment problem, submitted.

Vinh, L.S., Varón, A., Wheeler, W.C., 2006. Pairwise alignment with rearrangements. Genome Inform. 17, 2.

Wang, L., Jiang, T., 1994. On the complexity of multiple sequence alignment. J. Comput. Biol. 1, 337–348.

Wang, L., Jiang, T., Lawler, E.L., 1996. Approximation algorithms for tree alignment with a given phylogeny. Algorithmica, 16, 302–315.

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., 2001. Homology and the optimization of dna sequence data. Cladistics, 17, S3–S11.

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

Wheeler, W.C., Gladstein, D., De Laet, J., 2003. POY, Phylogeny Reconstruction via Optimization of DNA and other Data version 3.0.11 (6 May 2003). American Museum of Natural History. ftp://ftp.amnh.org.

Wheeler, W.C., Aagesen, L., Arango, C.P., Faivovich, J., Grant, T., D'Haese, C., Janies, D., Smith, W.L., Varón, A., Giribet, G., 2006. Dynamic Homology and Phylogenetic Systematics: A Unified Approach Using POY. American Museum of Natural History.