CABIOS

Vol.10. no.1. 1994
Pages 41 – 48

fastDNAml: a tool for construction of phylogenetic trees of DNA sequences using maximum likelihood

Gary J.Olsen, Hideo Matsuda[1,2], Ray Hagstrom[1] and Ross Overbeek[1]

Abstract

We have developed a new tool, called fastDNAml, for constructing phylogenetic trees from DNA sequences. The program can be run on a wide variety of computers ranging from Unix workstations to massively parallel systems, and is available from the Ribosomal Database Project (RDP) by anonymous FTP. Our program uses a maximum likelihood approach and is based on version 3.3 of Felsenstein's dnaml program. Several enhancements, including algorithmic changes, significantly improve performance and reduce memory usage, making it feasible to construct even very large trees. Trees containing 40–100 taxa have been easily generated, and phylogenetic estimates are possible even when hundreds of sequences exist. We are currently using the tool to construct a phylogenetic tree based on 473 small subunit rRNA sequences from prokaryotes.

Introduction

Tools for the construction of phylogenetic trees play a central role in evolutionary analysis. In particular, methods for inferring relationships from molecular sequence data are especially valuable, given the enormous increases in DNA sequence data. For many groups of organisms, the molecular fossil record provides the only clear insights we have into their origins and histories (e.g. Woese, 1987; Sogin *et al.*, 1989). In addition, the relationships between the various genes in contemporary organisms reflect the processes by which today's complex genomes have arisen from a simpler ancestral genome by duplications, rearrangements and sequence changes that have given rise to new functions.

A number of distinct approaches to phylogenetic inference exist, and a substantial volume of literature comparing their relative merits has developed (e.g. Nei, 1987; Jin and Nei, 1990; Li and Grauer, 1991; Swofford and Olsen, 1991). Approaches based on maximum likelihood (Cavalli-Sforza and Edwards, 1967; Felsenstein, 1973a,b, 1982, 1989; Thompson, 1975; Saitou, 1990) are based on concrete models of the evolutionary process and are well motivated statistically, but their use has been hindered by the computational costs involved. Until recently, such costs have limited the use of maximum likelihood techniques to trees of under ~20 taxa.

One of the primary goals of the Ribosomal Database Project (Olsen *et al.*, 1992) is to produce and distribute a phylogenetic tree based on its alignment of ribosomal RNA (rRNA) sequences. In particular, the initial alignment of small subunit rRNA sequences from prokaryotes included sequences from 473 taxa, and we wished to produce a high-quality phylogenetic tree with lengths on the branches. To achieve this objective using a maximum likelihood criterion, we developed fastDNAml, which we are now making available. This work builds directly upon previous work: fastDNAml is an enhancement of version 3.3 of the dnaml program distributed by Felsenstein as part of the PHYLIP package (Felsenstein, 1989, 1990). We have used fastDNAml to develop an initial version of the phylogenetic tree of prokaryotic microorganisms based on the rRNA alignment; we will present that tree in detail elsewhere. In this paper, we discuss the enhancements we have introduced into the program and its use on machines ranging from Unix workstations to massively parallel supercomputers. We also provide an outline of how the tool can be used to study large alignments of sequence data.

Algorithm

General considerations

In the inference of relationships between gene sequences, maximum likelihood provides a well-defined objective criterion under which to compare the relative merits of alternative phylogenetic hypotheses. Specifically, one seeks the tree and branch lengths that have the greatest probability of giving rise to the present-day sequences. This is usually broken into two parts, defining the tree topology to be tested, and then optimizing the branch lengths on that particular topology. If the number of possible topologies for a given set of taxa is not too large, one could generate all unrooted trees containing the given taxa, and compute the branch lengths for each that maximize the likelihood of the tree giving rise to the observed sequences. One then retains the best tree (perhaps, along with a set of trees that have nearly optimal likelihoods).

Department of Microbiology, University of Illinois, Urbana, IL 61801 and [1]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439-4801, USA

[2]Present address: Department of Systems Engineering, Faculty of Engineering, Kobe University, 1-1 Rokkodai, Nada, Kobe, 657 Japan

Unfortunately, the number of bifurcating unrooted trees for n taxa is

$$\frac{(2n-5)!}{2^{n-3}(n-3)!}$$

which rapidly leads to numbers of trees that are well beyond what can be examined practically. Thus, a situation arises in which some type of heuristic search is made through a (one hopes) well-chosen subset of the possible trees attempting to locate that with maximum likelihood. This type of search is typical of optimization problems in which an exhaustive search is impractical.

The algorithm we use is a direct outgrowth (see below) of dnaml, version 3.3 (Felsenstein, 1990); the intellectual debt to Felsenstein will be obvious to anyone who studies the two programs. Following the algorithm of Felsenstein, we use sequential addition of taxa and tree rearrangement to search for the optimal tree topology. In the following outline of the strategy, n is the final number of taxa, i is the number of taxa in the current tree and T_i is the current estimate of the best tree of size i.

1. Compute the optimal tree, T_3, for the first three taxa (only one topology is possible).
2. Pick the next taxon A to be inserted. We consider, in turn, the topologies that result from attaching a new branch from A to each of the $2i - 3$ branches in tree T_i (i.e. we examine all the topologies that result from T_i by attaching a new branch point along an existing branch). For each of these topologies, compute the optimal branch lengths and corresponding likelihood. Set T_{i+1} to the best of this set.
3. Increment i (the tree now has one more taxon). If all the taxa have been added to the tree (i.e. $i = n$), then skip to step 5. Otherwise, continue with step 4.
4. Since we have evaluated only a small portion of the possible trees containing the same taxa as T_i, we may well not have the optimal tree. At this stage, we perform a partial tree check to see whether minor rearrangements lead to a better tree. By default, these rearrangements can move any subtree to a neighboring branch (often called nearest neighbor interchanges). Reset T_i to the best tree resulting from this set of rearrangements. Repeat this step until none of the alternatives tested is better than the starting tree. Return to step 2.
5. Perform a full tree check in an attempt to improve T_n via rearrangements. The only difference between the full tree check and a partial tree check is that the user can elect to search a greater number of rearrangements by increasing the number of branch points that can be crossed while moving a subtree. After testing the set of rearrangements, reset T_n to the best. Repeat this step until none of the alternatives tested improves T_n. Then terminate, reporting T_n as the best tree found.

The partial tree check in step 4 and the full tree check in step 5 both are basically hill-climbing optimizations. In each case, a set of trees is generated by performing simple operations on T and each of the trees in the set is evaluated. If any is better than T, T is equated to the best of the set, and the process repeats from this new starting point. This will produce a tree that is a local optimum under the given set of simple operations.

Improving performance

Our primary goal was to arrive at the same answer as dnaml version 3.3, but to do so faster. In order to achieve this, our method differs from Felsenstein's in several ways.

The primary gain in speed was achieved by changing the algorithm used during the computationally intensive operation of finding optimal branch lengths. Felsenstein chose an EM algorithm to make successive steps towards the optimal length of a given branch; we chose to use a Newton–Raphson method based on analytic calculation of first and second derivatives of the likelihood. There is no guaranty that this will converge on the true optimum, but so far it seems to work in practice (we have not yet found a tree for which we arrive at branch lengths significantly different from those found by dnaml. In dnaml version 3.4 Felsenstein has made a similar change (personal communication).

A second change in calculation strategy involves the approach to 'simultaneously' optimizing branch lengths over the whole tree. In both programs this is done by making a number of passes over the tree, adjusting branches one at a time. The passes are continued until no branch changes by more than a given value, or until a maximum number passes is reached. In dnaml version 3.3, Felsenstein made a substantial effort to drive each branch toward its optimum before moving on to the next branch. We limit the effort invested in a given branch on each pass. The rationale is that subsequent changes to other branches can invalidate the extra effort; thus, one is better off making a more limited effort on each pass. The result of the change is a net decrease in recomputation, though we have neither measured the extent to which the limit improves performance, nor determined its optimal value.

One final feature of fastDNAml is a quick add (Q) option that provides a rapid estimate of the location at which to insert a new sequence into the growing tree. In essence, if the conceptual basis of a tree with branch lengths is sound, then splitting one branch by inserting a new taxon at its proper location in the tree should minimally perturb the optimal branch lengths elsewhere in the tree. Thus, if only the new branch (with the new taxon at its tip) and the two adjacent branches (the two halves of the branch that was just split) are reoptimized, the likelihood should still be fairly close to optimal. By not reoptimizing all the branches in the set of alternative new trees, much time is saved. Once a specific insertion point is selected as best, its branch lengths are then refined across the whole tree. Though it might seem that errors due to this approximation

would propagate through the reset of the tree-building process, this may not be so, since the next operation in the tree-building algorithm is performing local rearrangements on the tree, thereby providing an opportunity to correct an erroneous placement.

Implementation

Program functions

For the full functionality of dnaml, we refer the reader directly to Felsenstein (1989, 1990). Options that are retained from the original program include: use of either user-supplied or empirical base frequencies (F option); a user-specified ratio of transition substitutions to transversion substitutions (T option); randomizing (jumbling) the order of sequence addition to the tree (J option); evaluating the likelihood of a user-supplied tree (with or without optimizing the branch lengths) (U and L options); and writing the inferred tree to a file (Y option). Below we discuss other program options that have been added or extended.

In the testing of tree rearrangements, we have generalized Felsenstein's local and global search operations to allow moving subtrees across an arbitrary (user-defined) number of nodes. The user can separately specify (using an extension of the G option) the number of nodes that are crossed during rearrangement of partial trees (the default is 1) and the full tree (default is 1). By crossing one node, the user gets local rearrangements (nearest-neighbor interchanges). By crossing many nodes, the user gets the same global rearrangements as in dnaml. By crossing an intermediate number of nodes, the user can get regional rearrangements. It is even possible to set these values to zero, so that taxa are added to the tree without re-examining earlier decisions. We have not fully determined the effect of altering the range of permitted rearrangements on the frequency of converging on the global maximum rather than a local optimum. However, comparisons on ~ 10 data sets suggest that increasing the exploration of partial tree rearrangements is computationally expensive, and gives little if any increase in the frequency of finding the optimal tree.

We allow more rate categories to be associated with different columns in the alignment when using the C option. Felsenstein allows nine in dnaml versions 3.3 and 3.41; we chose to extend this to 35 categories (represented by the series 1, 2, 3, . . ., 9, A, B, . . ., Z). If a 1000-fold range of rates were distributed over 35 categories, any desired rate value would fall within 11% of one of the category rates. Thus, mapping a continuous distribution of rate estimates onto the discrete set of category rates will not entail significant degradation (especially when it is appreciated that the most common practice in molecular phylogenetic inference is to 'discretize' all rates to 1).

We permit alignment position weights other than one and zero when using the W option. In particular, we allow the positions to take any integer weight in the range 0−35. This was originally introduced to define bootstrap samples of the input data without rewriting the entire sequence alignment. However, it can also be used as a method for lowering the individual contributions of highly correlated positions, such as those involved in base-pairing in RNA secondary structure.

We have added checkpoint and restart capabilities to the program. Since runs can go for days when executed on single processors, it can be distressing to lose several days' work when a run is interrupted prior to completion. The restart function (R option) permits continuation of the calculation from the last completed insertion or rearrangement. A useful side-effect of the restart function is the ability to add new sequences to an existing tree easily and rapidly. To do this, the user adds one or more sequences to the old data table, and then uses the final tree of the smaller data set as the starting point for the new larger tree.

One more application of the restart capability is the ability to increase the set of rearrangements tested on a previously computed tree. For example, a tree produced with nearest neighbor rearrangements of the final tree could be restarted with a global search (G) option, and the additional tree rearrangements would be generated and tested. Because it requires 2−3 times as much CPU time to do global rearrangements on a tree of n taxa than to construct it from scratch, we sometimes build a large number of trees using different orders of sequence addition, and then select only the best of these as starting points for global rearrangements.

Felsenstein provides the seqboot program to generate bootstrap samples for dnaml (and the other sequence-based programs in his package). We have added a bootstrap (B) option to fastDNAml that generates an appropriate sample of the columns (L samples drawn with replacement from the L columns of the alignment) without rewriting the entire data table. The bootstrap sample is based on a user-supplied random number seed, separate from that used by the jumble option.

These enhancements and changes help considerably with performance and convenience of use. For Unix environments, we have created an array of shell scripts that can be used to add program options to the input file before passing it to the program, thereby minimizing the need to edit the file prior to each program run (or to interactively edit options, as in version 3.4 of the dnaml program).

Coding and adding parallel processing

To compute the phylogenetic tree of 473 microorganisms, it was critical that we be able to exploit the computational resources offered by parallel processors. In our case, the computational demands were high enough to warrant using a massively parallel machine (the Intel Touchstone DELTA System, comprising 576 i860 processors); more commonly, researchers will find it useful to be able to conveniently use a set of available Unix workstations.

We used the basic approach outlined in Boyle *et al.* (1987), using an updated version of the p4 tools developed by E.Lusk and R.Butler. We developed the implementation as follows:

1. We developed a functioning sequential version of the code in C (dnaml version 3.3 was written in Pascal) running on Sun SPARCstations under SunOS 4.01. The sequential code has also been compiled and run on other hardware systems and software environments including the following: an i486-based PC (MS-DOS), Alliant FX/8, Cray II, Cray Y-MP, Digital MicroVax (VMS), Digital DEC station, Digital Alpha (VMS), IBM RS/6000, Silicon Graphics R4000 and various Sun SPARCstations. Generally, we have used the compiler supplied with the operating system, although we have also used GNU C version 1.40 (SPARC).
2. We split the computation into two processes: a master process co-ordinated the search for an optimal tree, invoking the services of a slave process whenever it was time to optimize branch lengths and evaluate a set of topologies. Initially, the co-ordination between the master and slave processes was achieved by communication using Unix files. This restructuring of the sequential program was achieved in an afternoon working on a Sun workstation. Currently, the sequential, master and slave codes are a single file, and the different versions are produced by a compile-time switch. This approach ensures functional consistency of the sequential and parallel codes.
3. The file-based communication was replaced with the p4 routines distributed by Lusk and Butler. Again, this was a trivial modification, requiring a few hours of effort.
4. Once the software was structured in such a way that problems were sent to a slave and the master waited for the results, it was straightforward to introduce a general-purpose dispatcher between the master and slave. This process initiates an arbitrary number of copies of the slave, distributes problems to them, and collects the results to be sent back to the master. This part of the code could have been developed on a Sun workstation, but we found it convenient to use a Sequent Symmetry, a parallel processor that offers superior debugging support. Coding and debugging the dispatcher required only a couple of days. The product was a more-or-less portable program that could be run on a number of machines.
5. We moved the program to an Encore Multimax, a BBN TC2000, an Intel GAMMA, an Alliant FX/8 and an Intel DELTA. Once the p4 toolkit is completely installed on a system, such ports are achievable in a matter of minutes. In some cases, when we attempted to use p4 toolkit versions that were still under development, we did experience occasional delays. In particular, such delays occurred when moving to a relatively primitive environment in which neither the hardware nor the software was yet completely stable—as was inevitable with the Intel DELTA, on which we were fortunate to gain access before the delivered machine was even officially accepted.
6. Once the system was operational on the Intel DELTA, we made the runs required to construct a tree of the 473 organisms in just a few days, rather than the years that would have been required on a workstation. The p4 environment became stable, and the program was easily debugged (since all but one bug had been detected and solved in the far more benevolent environments supported on machines such as the Sequent and Encore).
7. Finally, in preparing the system for distribution, we ported it to a network of Sun workstations. This task required less than a day and largely involved building in options that allow a few minor conveniences when one has a shared file system.

We now have a system that can be run effectively on single-processor workstations, on a number of medium-scale parallel processors (such as the Alliant, Sequent and Encore machines), on massively parallel machines (such as the Intel DELTA) and on workstations communicating over a network.

Results

Performance of the sequential code

We compared the performance of fastDNAml (without and with the quick add option) to three versions of dnaml (dnaml 3.3 compiled with the Sun Microsystems PC-O3 command and the Sun SPARC executable files for dnaml 3.41 and dnaml 3.5c, as supplied by Felsenstein). Two rRNA data sets were examined: easy__10, a set of 10 sequences with a clearly defined branching order; and hard__16, a set of 16 sequences with numerous nearly equivalent branching orders. We used the jumble option of the programs to vary the order of sequence addition to the tree and monitored the CPU time required per replicate. The results of these comparisons are in Table I.

When the difficulty of finding the optimal tree is low, fastDNAml and dnaml 3.41 have similar performances. The quick add (Q) option of fastDNAml provides some advantages. In contrast, dnaml 3.3 and dnaml 3.5c were substantially slower, so they were not included in further tests. When the tree was harder to infer, fastDNAml found the correct tree more quickly than dnaml 3.41, but the distinction in raw rates might not have been significant given the uncertainty in the probability of finding the correct tree (Table I and below).

In conclusion, the speed improvements made by Felsenstein between dnaml versions 3.3 and 3.41 are comparable in magnitude to the speed improvements we have made in deriving fastDNAml from the former program. With the Q option, fastDNAml was about twice as fast as dnaml 3.41 at finding trees using local rearrangements.

Efficiency in finding globally optimal trees

The factor of greatest interest in comparing tree inference programs is not how long it takes to find a tree, but how long it takes to find the best (globally optimal) tree. This requires scaling the time per addition order by the fraction of the addition

Table I. Performance of DNA maximum likelihood tree programs on a Sun SPARCstation II

Alignment[a]	Performance	Program (compile command)				
		fastDNAml (gcc-O)	fastDNAml with Q option (gcc-O)	dnaml 3.3 (pc-O3)	dnaml 3.41 (compiled by Felsenstein)	dnaml 3.5c (compiled by Felsenstein)
easy__10	time ± SD (s)	201 ± 16	126 ± 17	1142	244 ± 39	10 506
	no. optimal[b]/trials	100/100	100/100	1/1	100/100	1/1
	time per optimum (s)	201	126	1142	244	10 506
easy__10__G	time ± SD (s)	592 ± 118	518 ± 124	3051	568 ± 133	–
	no. optimal/trials	50/50	50/50	1/1	50/50	–
	time per optimum (s)	596	522	3051	564	–
hard__16	time ± SD (s)	2986 ± 448	2077 ± 460	–	4937 ± 750	–
	no. optimal/trials	5/400	5/400	–	5/400	–
	time per optimum (s)	238 880	166 160	–	394 960	–
hard__16__G	time ± SD (s)	22 225 ± 10 237	20 918 ± 9299	–	24 246 ± 11 608	
	no. optimal/trials	0/17	3/30	–	0/17	–
	time per optimum (s)	>377 825	209 180	–	>412 182	–

[a]Alignment: easy__10, an alignment (1892 columns, but only 385 distinct patterns of nucleotides in the columns) of 10 16S rRNA sequences (from *Haloferax volcanii, Halococcus morrhua, Halobacterium cutirubrum, Halobacterium halobium, Methanospirillum hungatei, Methanocorpusculum parvum, Methanoculleus marisnigri, Methanoculleus thermophilicus, Methanomicrobium mobile* and *Methanoplanus limicola*) for which one tree is substantially better than all alternatives; easy__10__G, the easy__10 data with the G (global search) option; hard__16, an alignment (2413 columns with 962 distinct data patterns) of 16 16S rRNA sequences (from *Agrobacterium tumefaciens, Bacillus subtilis, Chlorobium vibrioforme, Clamydia psittaci, Cytophaga heparina, Deinococcus radiodurans, Fusobacterium nucleatum, Heliobacterium chlorum, Leptonema illini, Leuconostoc oenos, Planctomyces staleyi, Synechococcus* sp. PCC 6301, *Thermococcus celer, Thermomicrobium roseum, Thermotoga maritima* and *Zea mays* mitochondrion) for which numerous branching orders have nearly the same likelihood, making the tree search very difficult; and hard__16__G, the hard__16 data with the G (global search) option.

[b]'Optimal' refers to trees equivalent to the (presumptive) global optimum. All orders of sequence addition yield a tree that is at least locally optimal.

orders yielding the globally optimal tree. These values are also reported in Table I. In the case of the easy__10 alignment, all sequence addition orders tested lead to the same tree, so the scaled comparison is trivial.

Differences in performance of the programs at finding the globally optimal tree, as opposed to a locally optimal tree, should be most obvious in the case of hard__16 alignment. In particular, since the Q option of fastDNAml is a heuristic to speed the evaluation of tree branch insertion locations, it might, at least in principle, lead to inferior decisions during the process of adding taxa to the tree. With the hard__16 alignment, fastDNAml, fastDNAml Q and dnaml found the optimal tree the same number of times, five out of 400 orders of addition. In examining the behavior more closely, we found that the programs did not find the global maximum with the same orders of addition. That is, the results of individual addition orders were sometimes different, but none of the alternatives were systematically worse (or better).

Because the globally optimal tree for the hard__16 data was found so rarely, there is a large uncertainty in the frequency of finding the best tree, and hence in the scaling that should be applied to the time per addition order. However, the overall distributions of scores (locally optimal trees, whether they are globally optimal or not) were not observably different either. For the collection of 400 locally optimal trees found by fastDNAml, fastDNAml Q and dnaml, the mean log-likelihoods (± standard deviation) were −17937.34428 ± 11.68394, −17937.25228 ± 11.75695 and −17937.27159 ± 12.05913 respectively. Similarly, the median log-likelihoods were −17933.63079, −17934.19920 and −17933.63282 respectively.

The global (G) search option of the programs is intended to increase the fraction of the time that a tree search locates the globally optimal tree. If this frequency can be increased by a factor greater than the increase in time required per addition order, then there will be a net decrease in the time to find the optimal tree. In the case of the easy__10 data, the optimal tree was always found, so there was a 2- to 4-fold increase in time per optimal tree (Table I). With the hard__16 data the time required per addition order increased 4- to 10-fold. The few addition orders that we have run suggest that there is not enough of an increase in the frequency of finding the globally optimal tree to compensate for the increase in time per cycle. For many of the problems that we have examined, the global search option decreases the overall rate of finding the optimal tree. As noted above, the context in which we have found global (or regional) searches most useful is to generate a large number of locally optimal solutions without the G option, and then to combine the fastDNAml restart and global options to try to search more broadly the vicinities of the several best trees.

Thus, there are no obvious differences in the abilities of these programs to find the globally optimal tree; when it is easy they all succeed, and when it is hard they do comparably well. To within the limits of our measurements, the overall rate of finding globally optimal trees is proportional to the rates at which the

programs find locally optimal trees, suggesting that fastDNAml with the Q option is about twice as fast as dnaml 3.41 in solving realistic problems.

Estimating the time required to infer trees from other alignments

Estimating the rate of tree inference for a new alignment is complex, but to a first approximation it is proportional to the number of unique data columns in the alignment times the cube of the number of taxa. Since 126 s were required for an alignment of 10 sequences with 385 distinct patterns of nucleotides in the columns, this would extrapolate to 2150 s for 16 taxa with 962 unique data columns. The average time for the hard__16 data set was 2986 s (see Table I), so this simple scaling of time required is ~39% low.

The time required to find a locally optimal tree is divided between the tasks of adding branches to the tree and performing tree rearrangements (looking for better trees). For the easy__10 alignment, there are seven branch addition steps and at least six rearrangement steps (each time a rearrangement improves the tree, then there is another rearrangement step, increasing the total). Because finding the optimal tree for the easy__10 tree was straightforward, the average number of rearrangement steps per tree was 6.4, very close to the minimum. In contrast, for the hard__16 alignment, there are 13 addition steps and at least 12 rearrangement steps. Because the tree search for this alignment was difficult (due to the number of trees with nearly the same likelihood), the average number of rearrangement steps per tree was 20.7. If the extra rearrangements are taken into account, the time estimate for finding a locally optimal tree for the hard__16 becomes

$$2150\left(\frac{7+6}{7+6.4}\right)\left(\frac{13+20.7}{13+12}\right)\text{ s}$$

or ~2812 s. This is much closer to the actual number, but it required knowledge about the intrinsic difficulty of finding a tree for a given set of data. In essence, finding even a locally optimal tree is harder for some data. Also, as seen above, finding the globally optimal tree in these cases can require numerous trials.

The quick add option speeds the addition steps, but has no impact on the time for rearrangement evaluation. As tree problems become more difficult, neglecting changes in the number of rearrangements will have a larger relative effect on the time underestimate since the rearrangements occupy a larger fraction of the total time. It is also possible that the option will increase the number of rearrangement steps (by making bad placements that must be corrected by rearrangements). For the hard__16 alignment, the Q option only increased the average number of rearrangement cycles from 20.7 to 21.1, so this latter effect is quite small. The simple scaling of timings from 10 taxa with 385 unique columns to 16 taxa with 962 unique columns gives a time estimate of 1290 s, which is 61% low (compared to being 39% low without the Q option)—as expected, it is harder to scale timings with the Q option than without it.

The comparison of timings for the easy__10 and hard__16 alignments might lead to concerns that the increase in difficulty is intrinsic in the size of the problem, and that a higher power of the number of sequences might give more realistic time estimates. Analysis of an alignment of 43 sequences with 1106 unique columns (and a relatively clear optimal topology) required an average of 36 088 s per tree inferred when using fastDNAml with the Q option. If we take the power law to be unknown and equate the 10 sequence and 43 sequence trees to the same rate constant, we get:

$$\text{rate} = \frac{1106\text{ columns (43 sequence)}^a}{36\,088\text{ s}} = \frac{385\text{ columns (10 sequences)}^a}{126\text{ s}}$$

$$a = 3.17$$

If we consider the extrapolation from 43 to 80 sequences, the difference in the timing estimates given by $a = 3$ versus $a = 3.16$ is only 10%. Thus, a slightly higher power might be better than a cube law, but other differences between problems contribute larger variations in observed timings.

The other obvious concern—after the poor performance on the hard__16 alignment—is that inordinate numbers of input orders will be required to find the optimal tree for larger numbers of sequences. For the 43 sequence alignment, ~10% of the input orders gave the optimal tree. Thus, when the data are good, even fairly large trees can be found with moderate patience.

Reproducibility of the optimal tree log-likelihood values

In analyzing the performances of fastDNAml and dnaml 3.41, we observed that the log-likelihood values reported by dnaml fluctuated for trees of the same topology. For example, the log-likelihoods reported for 100 analyses of the easy__10 alignment varied from −6536.60742 to −6536.61621, even though the tree topologies were the same. For each of the problems in Table I, the means and standard deviations of the optimal tree scores reported by fastDNAml, fastDNAml Q and dnaml are recorded in Table II.

We do not know the source of variation in the optimal scores. The most obvious thought was that it might be due to variations in the optimized branch lengths. Although the maximum differences observed between corresponding branch lengths of the inferred trees were only ±0.00002 (about one part in 10 000), this might be sufficient to introduce variations in the log-likelihood. However, this cannot explain *increases* in the log-likelihood, since non-optimal branch lengths would always lower it. If noise in the branch lengths could raise the likelihoods, then the branch lengths were not optimal. Thus, we remain uncertain as to the cause of this variation in reported values.

Table II. Log-likelihoods reported for globally optimal trees

Data set	Log-likelihood ± SD[a]		
	fastDNAml	fastDNAml with Q option	dnaml 3.41
easy__10	−6536.61237 ± 0.00011	−6536.61232 ± 0.00000	−6536.61162 ± 0.00185
easy__10__G	−6536.61236 ± 0.00010	−6536.61232 ± 0.00000	−6536.61139 ± 0.00183
hard__16	−17 923.24918 ± 0.00000	−17 923.24918 ± 0.00000	−17 923.24297 ± 0.00450

[a]The number of optimal trees were 100, 50 and 5 for the easy__10, easy__10__G and hard__16 data sets. The variations are among trees with the same topology.

Discussion

Computational cost has hindered the use of maximum likelihood for inferring DNA-based phylogenetic trees with more than ~20 taxa. We have built upon the work of Felsenstein to develop a tool that enables any researcher with access to a workstation to generate DNA-based trees containing 40−50 taxa (e.g. about 12 h on a Sun SPARCstation II when the branching order is relatively well defined by the data). On workstations that support high-performance floating-point operations, it is practical to compute trees of 60−80 taxa (and possibly, with patience, to compute trees with >100 taxa). For cases in which multiple trees containing 50−100 taxa are needed, one can use a set of workstations on a network. In the fairly limited cases in which collections of hundreds of sequences exist, one can attain estimates by using a massively parallel machine.

However, it is generally necessary to compute a number of trees with different orders of taxon addition in order to gain some confidence that the optimal branching order has been found. This was dramatically illustrated with the data in the hard__16 alignment, which were chosen to include multiple lineages arising within a particularly short interval. The search for the optimal branching order is non-trivial, with only slightly more than 1% of the random addition orders finding the optimal tree. In addition, specific suboptimal trees were found more often than the maximum likelihood tree. This emphasizes the need to replicate any presumptive optimum, preferably several times.

In developing fastDNAml, there were two main advantages to starting with an existing program. First, we were able to use direct comparisons with dnaml outputs to verify the function of our program (a very significant issue in dealing with methods for which it is difficult to calculate the correct answer by hand). Second, the approach yields a program that is well integrated with an array of related tools. This array includes the rest of the PHYLIP package (Felsenstein, 1989, 1990), programs such as PAUP (Swofford, 1990) that generate or accept PHYLIP data file formats, and programs that generate or accept Newick format tree descriptions.

When we began our effort to develop a program for analyzing the alignment of 473 small subunit ribosomal RNA sequences, we estimated that we might be able to construct one estimate of such a tree using a few thousand hours of workstation time, or a few days of time on a machine such as the DELTA. This estimate was based on a number of graduated runs of dnaml and an informal observation that the execution time of dnaml 3.3 seemed to be proportional to the cube of the number of taxa.

As we developed more efficient versions of our program, we gradually became able to examine its behavior in solving larger problems. Although we were aware that the search for the optimal topology could become trapped in local maxima, we did not know how this scaled with tree size. When we could construct trees involving 60−100 taxa, it became clear that at these sizes local maxima change from a nuisance to a major problem. In addition, we found that multiple runs with differing orders of sequence addition (using different jumble option random number seeds) could yield discrete local maxima more often than the presumed global optimum. These observations threw into question the notion of making one long run constructing a large tree of 473 taxa. A more suitable approach appeared to be one with four steps: (i) use existing knowledge (based on phylogenetic analysis of smaller groups of sequences) to break the complete set of sequences into a number of partially overlapping subgroups; (ii) make multiple runs to analyze each group to find its global optimum; (iii) merge the derived trees into a single large tree; and finally (iv) conduct additional smaller runs to help resolve any specific problem areas.

In our case, we created 11 groups of taxa, which were selected to include a limited number of common organisms to provide landmarks for use in merging the resulting trees. These groups ranged from 22 to 67 taxa. Using 5−10 runs for each group, we were able repeatedly to produce a maximum likelihood value that we believe is probably optimal. We merged the best tree for each group into a single 473-organism tree, using a few runs of 40−50 selected organisms to resolve ambiguities in how the trees should be joined. These extra runs could have been eliminated or reduced, had we chosen the original overlapping sets more carefully. Finally, to develop data on which to base an estimate of confidence, we ran 15−20 trees of randomly selected taxa; each sample included 60−100 taxa. In addition, we ran an extensive bootstrap resampling analysis of one of the groups (70 runs were made of 43 taxa, which included the Archaea sequences). We are currently analyzing these data to determine which sections of the overall tree appear to be clearly determined and which are still in question. We intend to report on this analysis elsewhere.

The fastDNAml program, test data, documentation and utility

programs (implemented as Unix shell scripts) are available from the Ribosomal Database Project (Olsen *et al.*, 1992) by anonymous FTP to rdp.life.uiuc.edu (in directory pub/RDP/programs/fastDNAml). The files in the directory are sufficient for compiling and running the program in sequential mode. Since the p4 package is itself in the public domain and is available by anonymous FTP to info.mcs.anl.gov (in directory pub/p4), one can now easily install and use the program on a wide variety of parallel processing systems as well.

Acknowledgements

We are grateful to J.Felsenstein for his wonderful PHYLIP program package and for discussions regarding dnaml and to the Concurrent Supercomputing Consortium for access to the Intel Touchstone DELTA System. This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, US Department of Energy, under Contract W-31-109-Eng-38 and by National Science Foundation grant DIR 89-17863 to C.R.Woese, and G.J.O. G.J.O. is the recipient of a National Science Foundation Presidential Young Investigator Award.

References

Boyle,J., Butler,R., Disz,T., Glickfeld,B., Lusk,E.L., Overbeek,R.A., Patterson,J. and Stevens,R. (1987) *Portable Programs for Parallel Processors.* Holt, Rinehart, and Winston, New York.

Cavalli-Sforza,L.L. and Edwards,A.W.F. (1967) Phylogenetic analysis: models and estimation procedures. *Am. J. Hum. Genet.*, **19**, 233–257.

Felsenstein,J. (1973a) Maximum likelihood estimation of evolutionary trees from continuous characters. *Genetics*, **52**, 349–363.

Felsenstein,J. (1973b) Maximum likelihood and minimum-steps methods for estimating evolutionary trees from data on discreet characters. *Syst. Zool.*, **22**, 240–249.

Felsenstein,J. (1981) Evolutionary trees from DNA sequences: a maximum likelihood approach. *J. Mol. Evol.*, **17**, 368–376.

Felsenstein,J. (1982) Numerical methods for inferring evolutionary trees. *Q. Rev. Biol.*, **57**, 379–404.

Felsenstein,J. (1988) Phylogenies from molecular sequences: inference and reliability. *Annu. Rev. Genet.*, **22**, 521–565.

Felsenstein,J. (1989) PHYLIP—phylogeny inference package (version 3.2). *Cladistics*, **5**, 164–166.

Felsenstein,J. (1990) *PHYLIP Manual Version 3.3.* University Herbarium, University of California, Berkeley, CA.

Jin,L. and Nei,M. (1990) Limitations of the evolutionary parsimony method of phylogenetic analysis. *Mol. Biol. Evol.*, **7**, 82–102.

Li,W.-H. and Grauer,D. (1991) *Fundamentals of Molecular Evolution.* Sinauer, Sunderland, MA.

Nei,M. (1987) *Molecular Evolutionary Genetics.* Columbia University Press, New York.

Olsen,G.J., Overbeek,R., Larsen,N., Marsh,T.L., McCaughey,M.J., Maciukenas,M.A., Kuan,W.-M., Macke,T.J, Xing,Y. and Woese,C.R. (1992) The ribosomal database project. *Nucleic Acids Res.*, **20**, 2199–2200.

Saitou,N. (1990) Maximum likelihood methods. *Methods Enzymol.*, **183**, 584–598.

Sogin,M.L., Edman,U. and Elwood,H.J. (1989) A single kingdom of eukaryotes. In Fernholm,B., Breme,B. and Jornvall,H. (eds), *The Hierarchy of Life.* Elsevier Science Publishers, Amsterdam, pp. 133–143.

Swofford,D.L. (1990) *PAUP: Phylogenetic Analysis Using Parsimony, Version 3.0.* Illinois Natural History Survey, Champaign, IL.

Swofford,D.L. and Olsen,G.J. (1991) Phylogeny reconstruction. In Hillis,D. and Moritz,C. (eds), *Molecular Systematics.* Sinauer, Sunderland, MA, pp. 411–501.

Thompson,E.A. (1975) *Human Evolutionary Trees.* Cambridge University Press, Cambridge.

Woese,C.R. (1987) Bacterial evolution. *Microbiol. Rev.*, **51**, 221–271.

Accepted December 9, 1993