WILEY-BLACKWELL

Cladistics (2013) 1–9

Cladistics

10.1111/cla.12056

# Oblong, a program to analyse phylogenomic data sets with millions of characters, requiring negligible amounts of RAM

Pablo A. Goloboff*

*Consejo Nacional de Investigaciones Científicas y Técnicas, Instituto Superior de Entomología, Miguel Lillo 205, S. M. de Tucumán, 4000, Argentina*

Accepted 19 August 2013

**Abstract**

Oblong, a program with very low memory requirements, is presented. It is designed for parsimony analysis of data sets comprising many characters for moderate numbers of taxa (the order of up to a few hundred). The program can avoid using vast amounts of RAM by temporarily saving data to disk buffers, only parts of which are periodically read back in by the program. In this way, the entire data set is never held in RAM by the program—only small parts of it. While using disk files to store the data slows down searches, it does so only by a relatively small factor (4× to 5×), because the program minimizes the number of times the data must be accessed (i.e. read back in) during tree searches. Thus, even if the program is not designed primarily for speed, runtimes are within an order of magnitude of those of the fastest existing parsimony programs.


## Introduction

Many phylogeny computer programs are currently available and in wide use, but as sequencing methods continue to improve and vast quantities of data become available, they begin to have serious limitations on regular desktop computers, especially in terms of memory. This is, in part, a consequence of most of the research in computational phylogenetics having been aimed, not at minimizing memory usage, but instead at improving the speed for larger numbers of taxa. This is logical, as the time needed for computing solutions increases exponentially with numbers of taxa, but only linearly with numbers of characters. For larger numbers of taxa, even heuristic solutions can be very time consuming, and several methods to improve the speed of branch-swapping have been proposed (see Goloboff, 1993, 1996, 1999; for parsimony calculations, and Guindon and Gascuel, 2003 and Stamatakis, 2006; for maximum-likelihood). Those speed-up methods are based on avoiding redundant calculations, through the use of bookkeeping and buffering different stages of the analysis. Thus, the speed is obtained mostly at the expense of memory. This is the approach taken by all the main programs for phylogenetic analysis, which in general prioritize speed of calculations over memory usage. Alphabetically ordered examples of such memory-demanding programs are FastTree (Price et al., 2009), Garli (Zwickl, 2006), Mega (Kumar et al., 2012), MrBayes (Huelsenbeck and Ronquist, 2001), PAUP* (Swofford, 1998), Phylip (Felsenstein, 2005), PhyML (Guindon et al., 2010), RaxML (Stamatakis, 2006), and TNT (Goloboff et al., 2003b, 2008). As some of these programs have become more efficient at dealing with large numbers of taxa, the size of the trees that are commonly published has increased enormously in the last few years (e.g. Goloboff et al., 2009, analysed a data set of 73 060 taxa and 10 145 characters).

However, given that the amount of RAM memory needed for standard phylogenetic programs is always several times the size of the data set on disk, the analysis of data sets with millions of characters

*Corresponding author:
*E-mail address*: pablogolo@csnat.unt.edu.ar

becomes prohibitive on regular computers, not because of speed reasons, but instead because of RAM limitations. Currently, genomic data sets with numbers of characters in the order of millions comprise relatively small sets of taxa (normally within the hundreds), as the complete sequences required are still expensive and have been obtained for few species. Therefore, it is ironic that the extra memory used for speeding up analyses for larger taxon sets ends up being rather unnecessary—the rearrangements to complete branch swapping are relatively few, and could be accomplished relatively rapidly even without the use of special shortcuts.

This paper describes Oblong, a new program specifically designed for parsimony analysis of matrices with very large numbers of characters using very low amounts of RAM. The program is designed for data sets with relatively small numbers of taxa (up to a few hundred), but with many characters (possibly hundreds of millions). The name of the program derives from these matrices: rectangles that are much wider than long. To avoid using vast amounts of RAM, Oblong buffers data to disk and then uploads the data by parts. By minimizing the number of times the data are uploaded into the program during branch-swapping, a tree search can be completed using only 4–5 times longer than if holding all data in memory, yet using only a few Mb of RAM—even for data sets that occupy several Gb on disk.

Oblong is a minimalist program, with very little code (below 2500 lines), designed for the analysis of data sets with up to four states (i.e. DNA data). Small portions of the code have been taken from TNT (a much larger, closed source program with 125 000 lines of code; Goloboff et al., 2008); these portions include the core bit operations for parsimony calculations (first used in J. S. Farris's 1994 program Jac, and subsequently incorporated into TNT) as well as some tree-handling functions.

## Material and methods

### *Simulated data sets*

For the comparison with other programs, data sets with different sizes were created, under the Jukes–Cantor model, using model trees with a random topology and branch-lengths chosen at random in the interval 0.01–0.3. The data sets were created using a TNT script, *simul.run*, also found in the Oblong web site. The RAM comparisons were done with Windows TaskManager, recording the maximum memory usage for each program. To make comparisons of speed and memory more direct, the code for all parsimony programs except PAUP* and MEGA (which are not open source) was compiled with the same compiler (Open Watcom, available at http://www.openwatcom.org). All the tests were carried out on a machine running under 32-bit Windows 7, with an Intel i5 processor (3.2 GHz). The hard drive on this machine is a 4-year old disk (WDC WD 3200AAKX-001CA0 ATA); newer disks, with a higher read speed, will produce faster results than reported here.

### *General approach to saving memory*

Oblong uses several approaches to lessen memory requirements. The first is never holding the entire data file on RAM; the second is storing fewer separate variables for length calculations; the third is optionally using temporary files on disk to avoid holding all data in RAM. The temporary files created by Oblong at runtime are deleted on normal termination.

### *Identification of uninformative characters as the data set is read*

In most programs, the original data are placed in a single array as they are read from disk; uninformative characters (i.e. those that require the same number of steps on all trees) are identified and packed subsequently. In Oblong, there is a small buffer for the original data, and the data set is read in parts of no more than 1000 characters at a time (Fig. 1). Oblong thus reads the first 1000 characters for each of the taxa (characters 0–999), then the subsequent 1000 (characters 1000–1999), and so on, by repositioning the file buffers. For each set of 1000 characters, the informativeness of the characters is assessed, and the informative characters are saved to a temporary file, *oblong.tmp*. For characters with four states (the maximum allowed by Oblong), four bits per taxon are used, but Oblong eliminates uninformative states (those occurring in fewer than two taxa) and uses fewer bits per taxon for characters with fewer informative states (the program keeps track of the steps in uninformative states, which is a constant for all trees, so that it reports the same length as other programs). Once the entire data set has been processed in this way, the file *oblong.tmp* will contain all the informative characters, without ever having held the entire data in RAM. Subsequent program operations rely solely on the compressed file, *oblong.tmp*, and the original data file is closed.

### *Reading all informative characters back into RAM*

When all the data are to be held in RAM (the maximum memory that Oblong will use), the temporary data saved to *oblong.tmp* are read back into the program. The characters are then placed in a matrix,

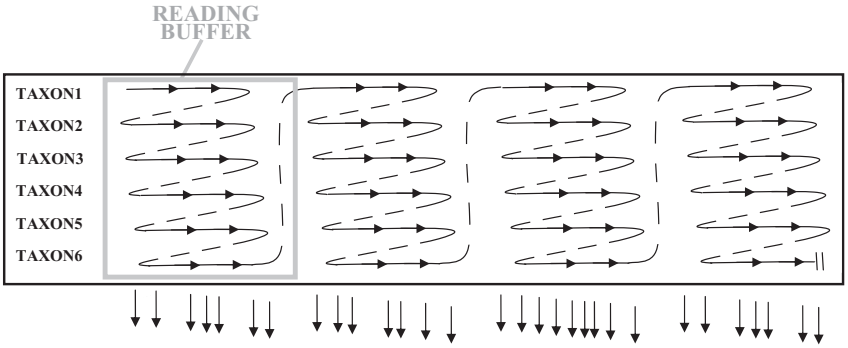


Fig. 1. Oblong never holds the entire data set in RAM; it reads the data in parts, and assesses informativeness of the characters as it reads each piece (saving the informative characters only, in compressed binary form, to a temporary file).

which requires at most as many bytes as the size of the dataset on disk. The characters are represented with the parallel approach used by TNT (similar to that of Goloboff, 2002; and White and Holland, 2011), so that every state is represented in a 4-bit field, with eight fields under 32-bit int (in Windows) or 16 fields under 64-bit int (in Linux/Mac). Because twice as many nodes as taxa are needed to optimize a tree, but each of the characters is represented with four bits (half a byte), then the size of this array (when every character is a four-state informative character) is the same as the size of the data set on disk (when many characters/states are uninformative, as in many real data sets, even a smaller array will suffice).

Branch-swapping consists of clipping the original tree in two, obtaining two subtrees, and then reinserting the clipped subtree at different branches in the main subtree. Oblong derives the length of each rearrangement using the method of Goloboff (1993), based on comparing the final states for the nodes delimiting the branches to be joined (this speeds up calculations for $T$ taxa by a factor of $T$, relative to a complete down-pass optimization for each rearrangement). Other shortcuts described by Goloboff (1996), to obtain further speed-ups by deriving the final states for the two subtrees from the separately stored down- and up-pass states require additional memory, and are avoided in Oblong. Thus, Oblong uses a single array, calculating first the down-pass for internal nodes, and then the up-pass on the same array (i.e. overwriting down-pass values in the up-pass). As described by Goloboff (1993, 1996), the node comparisons for length calculation should be done between the union of the final states for the nodes delimiting the target branch, and the union of the states for the nodes delimiting the branch where the clipped subtree is rerooted (with the proviso that, for polymorphic terminals, the states must be those that would be assigned to the terminal in an up-pass). In TNT, those unions are calculated and stored separately; Oblong instead recalculates the up-pass for the terminals and the unions of the final states for each rearrangement, without storing them separately. This, of course, has a negative effect on speed, but the main goal in Oblong is to lessen the memory required.

*Buffering data to disk*

When very low memory is to be used, Oblong uses a buffer where it can hold only a fraction of the packed characters at a time. The entire character matrix is then saved to disk, using a file called *oblong.pts*.

To enable a faster reading of these data from disk, the memory for the buffer holding a number $N$ of (packed) characters for $T$ terminal taxa is allocated sequentially, on the same memory segment. In C code:

```
int * at;
int ** matrix;
…
matrix = (int **) malloc ((2 * T – 1) * sizeof (int *));
at = (int *) malloc (N * T * sizeof (int));
for (j = 0; j < T; ++j, at += N)
  matrix [j] = at;
for (j = T; j < MAXNOD; ++j)
  matrix [j] = (int *) malloc (N * sizeof (int));
```

Note that for the internal nodes (from T to MAXNOD), there is no need for sequential allocation—those matrix cells are filled by the program, not read from disk. After the matrix for the terminal taxa has been allocated as above, for each set of $N$ characters loaded onto the matrix from the file *oblong.tmp*, the entire two-dimensional buffer can be saved to the file *oblong.pts* before the search begins, as if it was a single one-dimensional array, using the standard C function *fwrite*:

```
fwrite ((void *) matrix, 1, sizeof (int) * T * N, fileptr);
```

More importantly, the character matrix can also be read back quickly into the program with *fread*, the sister function of *fwrite*. By doing successive reads in the same order as the successive writes, the entire

character matrix is reconstructed identically. Note that the file *oblong.tmp*, created as the original data set is read, could not be read in this way, because it lists the states of each of the taxa, for each of the characters—the appropriate sequence of data can only be written to the file *oblong.pts* once all the data have been processed. For simplicity, when using a disk-buffered character matrix, all the characters are packed into 4-bit fields (i.e. Oblong does not distinguish in this case between two-, three- and four-state characters).

Recall that the full tree-length calculation with all characters held in RAM would involve visiting all tree nodes as in the C-style pseudocode shown on the left side of Fig. 2 (with nodes assumed to be appropriately numbered, and tree stored as lists of descendants for each internal node, *left_desc* and *right_desc*; multi-character calculations are avoided for simplicity).

The equivalent calculation for the matrix loaded in pieces (a total of *num_pieces* pieces, of *temp_nchar* characters each) is shown on the right side of Fig. 2. While the states for the internal nodes are overwritten as the different parts of the matrix are loaded and optimized, the length calculations themselves are correct.

Obviously, the approach described requires that the entire matrix be loaded again for each tree to be evaluated. But tree bisection and reconnection (TBR) branch swapping requires that thousands of trees are optimized; as the tree search is done in a sequential manner, this order allows us to evaluate many rearrangements loading the entire matrix only once (as shown on the pseudocode of the right side of Fig. 3). For comparison, the standard procedure for TBR (i.e. with the full character matrix stored in RAM) is also illustrated on the left side of Fig. 3. When the entire character matrix cannot be held in RAM, the matrix can be loaded only once per clipping, and the evaluation of all possible rerootings and targets repeated for every matrix piece loaded (storing in the array *partial_length* the cumulative lengths for each of the rearrangements). In this way, loading the entire matrix back into RAM only once, it is then possible to evaluate dozens of destinations, as shown in the pseudocode of the right side of Fig. 3. For $T$ taxa, the number of rearrangements that can be evaluated with a single matrix load ranges from ca. $2T$ (when a terminal is clipped) to ca. $T^2$ (when an internal node with half the taxa is clipped).

By using this approach, completing branch-swapping requires only 4–5 times longer than when the entire character matrix is held in memory, but using truly negligible amounts of RAM. On a Windows 32-bit machine, for example, a simulated data set with 15 million characters for 50 taxa (about 750 Mb on disk) required 497 Mb and 665 s for completing a random addition sequence holding all the character matrix in RAM, but only 6.24 Mb and 2587 s when buffering the matrix to disk (with a *–m5* switch). That is, the program becomes about four times slower, but it uses 80 times less memory. Increasing the size of the data set simply requires a larger number of reads from the disk buffer, so that the larger data set can still be analysed without increasing the amount of RAM needed. Note also that because the rearrangements are examined using the same sequence as when the full character matrix is held in RAM, exactly the same final results are obtained when buffering the character matrix to disk.

The pseudocode shown in Fig. 2 avoids re-loading the part of the character matrix that had been loaded in the last clip (using the *last_part_loaded* variable); while this saves no significant time when the matrix is divided in many parts, it does when the RAM buffer

```
length = 0 ;
for( n = 0 ; n < all nodes ; ++ n ) {
   node_ptr = matrix [ n ] ;
   left_ptr = matrix [  left_desc [ n ] ] ;
   right_ptr = matrix [ right_desc [ n ] ] ;
   for ( c = 0 ; c < num_chars ; ++ c ) {
      if ( ( * left_ptr & * right_ptr ) == 0 ) {
         ++ length ;
         * node_ptr = * left_ptr | * right_ptr ; }
      else  * node_ptr = * left_ptr & * right_ptr ;
      ++ left_ptr ;
      ++ right_ptr ;
      ++ node_ptr ; }}
```

```
length = 0 ;
for ( k = curc = 0 ; k < num_pieces ; ++ k , curc += temp_nchar ) {
   load_matrix ( curc ) ;  /*** load matrix starting at curc ***/
   for( n = 0 ; n < all nodes ; ++ n ) {
      node_ptr = part_matrix [ n ] ;
      left_ptr = part_matrix [  left_desc [ n ] ] ;
      right_ptr = part_matrix [ right_desc [ n ] ] ;
      for ( c = 0 ; c < temp_nchar ; ++ c ) {
         if ( ( * left_ptr & * right_ptr ) == 0 ) {
            ++ length ;
            * node_ptr = * left_ptr | * right_ptr ; }
         else  * node_ptr = * left_ptr & * right_ptr ;
         ++ left_ptr ;
         ++ right_ptr ;
         ++ node_ptr ; }}
```

Fig. 2. C-style pseudocode to compare the calculation of tree length in a down-pass holding the entire character matrix in RAM (left), and the calculation of tree length reading the character matrix in *numpieces* parts of *temp_nchar* characters each (right). The data are stored as A = 0001, G = 0010, C = 0100, and T = 1000 (multi-character calculations avoided here for simplicity). See text for details.

```
for ( clip = 0 ; clip < all clips ; ++ clip  ) {
   divide_tree ( clip ) ;
   /*** find final states for all nodes ***/
   two_pass_optimization( ) ;
   best_target = init_position ;
   best_root = init_rooting ;
   /*** Get initial value ***/
   best_cost =
      cost_of_joining ( clip , init_root , init_position ) ;
   for ( root = 0 ; root < all rootings ; ++ root ) {
      for ( target = 0 ; target < all targets ; ++ target ) {
         /***  Compare branches to join ****/
         cost = cost_of_joining ( clip , root , target ) ;
         if ( cost < best_cost ) {
            best_cost = cost ;
            best_root = root ;
            best_target = target ; }} /** end targets **/
          }  /** end roots  **/
   insert ( clip , best_root , best_target ) ; } /** end clips **/
```

```
partial_length = ( int * ) malloc ( T * T * sizeof ( int ) ) ;
...
curc = 0 ;
last_part_loaded = –1 ;
for ( clip = 0 ; clip < all clips ; ++ clip  ) {
   divide_tree ( clip ) ;
   num_insert = best_cost = 0 ;
   /*** Initialize length of all rearrangements to 0 ***/
   for ( root = 0 ; root < all rootings ; ++ root )
      for ( target = 0 ; target < all targets ; ++ target )
         partial_length [ num_insert ++ ] = 0 ;
   /*** Load matrix in pieces and, for each piece,
        do  all rootings and insertion points...  ****/
   for ( k = 0 ; k < num_pieces ; ++ k )  {
      if ( curc !=  last_part_loaded ) load_matrix ( curc ) ;
      last_part_loaded = curc ;
      curc += temp_nchar ;
      split_two_pass_optimization( ) ;
      best_cost +=
         split_cost_of_joining ( clip , init_root , init_position ) ;
      num_insert = 0 ;
      for ( root = 0 ; root < all rootings ; ++ root ) {
         for ( target = 0 ; target < all targets ; ++ target ) {
            cost = split_cost_of_joining ( clip , root , target ) ;
            partial_length [ num_insert ++ ] += cost ; } /** end targets */
            }  /** end roots **/
         }  /** end matrix pieces **/
   curc = last_part_loaded ;
   /**** Now that length calculations are complete for all
        rearrangements, find best rooting/target       ****/
   num_insert = 0 ;
   for ( root = 0 ; root < all rootings ; ++ root ) {
      for ( target = 0 ; target < all targets ; ++ target ) {
         cost = partial_length [ num_insert ++ ] ;
         if ( cost < best_cost ) {
            best_cost = cost ;
            best_root = root ;
            best_target = target ; }}}
   insert ( clip , best_root , best_target ) ; }  /** end clips **/
```

Fig. 3. C-style pseudocode to compare the process of branch-swapping when the entire character matrix is held in RAM (left), and branch-swapping reading the character matrix in *num_pieces* parts of *temp_nchar* characters each (right). See text for details.

is large enough to hold the character matrix in just a few pieces.

*Available options and usage*

Oblong can read either TNT, Phylip, or Fasta format data files. For Phylip data sets, the *–p* switch must be used; for Fasta files, the *–f* switch instead. It is also possible to read multiple data sets, by listing them in a file, and then giving Oblong the name of the file preceded by *–@*.

Both TBR and subtree prunning and regrafting (SPR) branch-swapping are available; searches can start from either random addition sequence Wagner trees (Farris, 1971) or random trees. The program can read trees in parenthetical notation as well, and use them as starting point for searches. Nixon's (1999) method for improving parsimony searches, the ratchet, is invoked with *–x*; Oblong only perturbs the original data set by deactivating some characters (instead of re-weighting); as in TNT, the original procedure described by Nixon (1999) was modified so that the perturbation phase (i) accepts rearrangements of equal score, and (ii) stops when a number of moves equal to $T/2$ have been accepted. When using the ratchet, the program allocates additional memory to keep track of temporarily active/inactive characters in each ratchet cycle. While it would be possible to handle these in the same manner as the matrix, by recalculating character eliminations on the same memory space (see Material

and methods), the present version of Oblong does not incorporate such an approach. The amount of memory needed is much less than needed for the matrix—as many bytes as half the number of informative characters in the matrix. For 100 million informative characters, this would mean using 50 Mb of RAM; if the matrix has 50 taxa, this additional memory represents 1% of the size of the data set on disk.

For measuring group supports, Oblong can perform jackknifing (with *–j*) as well as Bremer supports (with *–q*). The program uses the independent deletion probability proposed by Farris et al. (1996) to avoid the influence of uninformative characters; the default probability is $P_{(del)} = 0.36$, but this can be changed by the user. For each replication, jackknifing uses the same algorithms specified for searches, and saves the resulting tree to the output file (for subsequent calculation of group frequencies with TNT or other program). The trees obtained in each replication can optionally be collapsed (with *–k*) under TBR (or SPR), as in Goloboff and Farris (2001) and Goloboff (1999), so as to approximate the results obtained if saving multiple trees and producing their strict consensus for each replicate (see Goloboff et al., 2003a).

Bremer support values are calculated by TBR (or SPR) swapping the best (optionally, all) of the trees found, and recording the score difference for each move (as done in the TNT command *bsupport !*). Both the absolute Bremer support (Bremer, 1994) and the relative Bremer support (Goloboff and Farris, 2001; Farris and Goloboff, 2008) are available, as well as a new measure, the combined Bremer support. The latter combines the absolute and relative Bremer supports, approximating the results of jackknifing for simple cases. The formula used to combine the Bremer supports is $(Q.R)^{1/A}$, where $Q$ equals $1 - P_{(del)}$, $R$ the relative Bremer support, and $A$ the absolute Bremer support. As $Q.R$ is always smaller than unity, elevating it to $1/A$ will produce a number which approaches unity as the absolute Bremer support is larger. For conflicting characters, this measure strictly tends to zero as the support tends to zero (since it is based on comparisons of tree scores, and thus on optimality; see discussion in Wheeler, 2010). The correlation between resampling values and this combined measure is much better than the correlation between resampling values and either the absolute or the relative Bremer support, especially for groups with resampling frequencies above 0.50. A perfect correlation, however, seems neither possible nor desirable, as some actually supported groups can have a resampling frequency below 0.50 (see Goloboff et al., 2003a, for examples), thus having lower frequencies than unsupported groups (for which the frequency may approach 0.50, as in the case of groups supported and contradicted by the same numbers of characters, or exceed 0.50, in the case of more complex patterns). In addition to providing an evaluation of supports which is sometimes more sensible than that produced by resampling, the combined Bremer supports can be obtained much more quickly than jackknifing values, as they do not require repeating numerous searches.

The program does not draw trees, but relies instead on other programs to view the results. Oblong outputs (with *–o*) its results to a file that contains no character matrix, only the tree, in a format that is ready to be input to TNT. A publication-quality tree (with supports indicated on the branches) that can be viewed on most web browsers can be obtained (say, in the file *tree.svg*), with the following TNT commands after execution of Oblong:

```
oblong –idataset –ooutfile –q <enter> [for combined Bremer supports]
tnt; p outfile; ttag "&" tree.svg; zzz <enter>
oblong –idataset –ooutfile –j –r100 <enter> [for jackknifing]
tnt; p outfile; ttag=; majority; ttag "&" tree.svg; zzz <enter>
```

Note that in Linux or Mac the TNT commands would have to be given once inside the program, or the semicolons would have to be substituted by commas (see the online help of TNT on the *ttag* command for other options in plotting the tree). Optionally (with *–N*) the data can be saved in Nexus format, for reading and plotting with other tree-drawing programs [such as TreeView (Page, 1996) or Dendroscope (Huson et al., 2007)].

For memory usage, the *–m* switch determines (in Mb) the amount of memory to be used for temporary matrix buffers (the default is holding the entire character matrix in RAM). The number of characters to store in each piece (see Material and methods) is obtained by calculating (from the required amount of memory) the number of characters that can be held for the current number of taxa.

### *Comparison with other programs*

*Comparison with parsimony programs.* The speed comparisons are shown on Table 1. TNT was the fastest program in all cases. SSE-optimized versions of Parsimonator (Stamatakis, 2011) are faster than the version used here, and could conceivably run faster than TNT for small numbers of taxa. Parsimonator, announced as the "fastest open-source parsimony function implementation", does in fact perform a simplified SPR, examining trees within a restricted SPR neighbourhood, moving the clipped clade no further than 20 nodes away from the original position. As a result, part of the "speed" of Parsimonator for larger

Table 1
Running times (in seconds) for different parsimony programs and different numbers of characters and taxa

| | Data set size | | | | | |
|---|---|---|---|---|---|---|
| Program | $10^5 \times 30$ | $10^5 \times 40$ | $10^5 \times 50$ | $2.5 \times 10^5 \times 50$ | $5 \times 10^5 \times 50$ | $10^6 \times 50$ |
| TNT | 2.29 | 3.51 | 5.96 | 17.13 | 35.88 | 69.56 |
| | 1.53 | 2.45 | 4.35 | 11.84 | 22.21 | 55.68 |
| Oblong | 1.39 | 3.31 | 7.05 | 18.19 | 36.13 | 73.23 |
| | 0.89 | 2.17 | 4.31 | 11.40 | 23.79 | 46.11 |
| Oblong *–m2* | 6.66 | 17.32 | 30.87 | 67.13 | 158.50 | 338.26 |
| | 2.62 | 9.92 | 9.92 | 26.24 | 56.99 | 110.17 |
| PAUP* | 2.40 | 4.79 | 9.81 | 25.83 | 50.29 | 112.50 |
| | 2.17 | 3.81 | 7.44 | 18.83 | 38.22 | 86.20 |
| MEGA | 70 | 145 | 110 | 825 | ?? | ?? |
| | 15 | 25 | 25 | 205 | ?? | ?? |
| Parsimonator | – | – | – | – | – | – |
| | 2.81 | 6.35 | 15.94 | 24.07 | 54.96 | 105.55 |
| Phylip | – | – | – | – | – | – |
| | 770 | 2310 | 3865 | 9400 | ?? | ?? |

For most of the programs, two rows are given; the first corresponds to the time needed for completing five random addition sequences plus TBR; the second row corresponds to the time for five random addition sequences plus SPR. Two of the programs (Phylip, Parsimonator) cannot perform TBR. In the case of MEGA, an "MP search level" of 5 was used (which presumably does full SPR and TBR). Cases that were not run with some programs (due to memory constraints) are indicated with "??"

numbers of taxa comes from doing a more superficial search—it tries only a fraction of the SPR rearrangements, and the trees it delivers may perfectly well lead to better trees if subjected to true SPR in TNT or PAUP*. The simplified SPR used by Parsimonator probably did not have a strong effect on the data sets examined here (which consist of relatively low numbers of taxa, so that any move within 20 nodes of the original position can span the entire tree). A further complication for comparing TNT and Parsimonator runtimes is that what is highly optimized in TNT is not the SPR swapper, but instead the TBR swapper (to the point that completing TBR for large numbers of taxa takes even less than SPR).

Surprisingly, of the programs/versions tested, the second fastest branch swapper is that of Oblong with default settings—although the emphasis of the program is on memory usage, not speed, it still performs at very reasonable speeds. When using restricted amounts of memory (with the *–m2* switch), Oblong is slowed down only by a factor of about 4—runs take longer than in the fastest programs, but still in the same order of magnitude, and faster than the widely used Phylip and MEGA.

The memory usage of parsimony programs is shown on Table 2. Even with default settings (i.e. holding the entire character matrix in RAM), Oblong has the lowest RAM requirements of all programs (using on average about as many bytes as the size of the data set on disk). The closest competitor is Parsimonator, which on average uses twice that much memory. PAUP* and TNT are far more demanding in terms of memory, requiring 8–12 times more memory than Oblong under defaults. The most memory-demanding programs are

Table 2
Memory consumption (in thousands of kb) of different parsimony programs and different numbers of characters and taxa

| | Data set size | | | | | | |
|---|---|---|---|---|---|---|---|
| Program | $10^5 \times 30$ | $10^5 \times 40$ | $10^5 \times 50$ | $2.5 \times 10^5 \times 50$ | $5 \times 10^5 \times 50$ | $10^6 \times 50$ | Ratio |
| TNT | 43.30 | 47.91 | 54.34 | 123.05 | 244.29 | 486.39 | 11.3× |
| Oblong | 3.52 | 4.29 | 6.12 | 10.04 | 19.62 | 38.24 | 0.99× |
| Oblong –m2 | 2.96 | 3.57 | 3.36 | 3.82 | 3.80 | 3.84 | 1.01–0.08× |
| PAUP* | 31.60 | 46.06 | 52.99 | 104.20 | 236.68 | 465.89 | 10.2× |
| MEGA | 115.0 | 149.9 | 192.6 | 447.17 | ?? | ?? | 38× |
| Parsimonator | 6.22 | 9.62 | 11.86 | 25.48 | 51.48 | 101.56 | 2.2× |
| Phylip | 261.3 | 440.9 | 641.7 | 1553 | ?? | ?? | 61× |

The rightmost column corresponds to the average proportion of data set size on disk that the program uses as RAM (except for Oblong –*m2*, where the range is indicated—the smaller proportion corresponds to the larger data sets). For all programs that can use both algorithms, the memory required to run SPR and TBR is the same, except for MEGA (the table shows the memory needed for TBR). In the case of TNT, the options "cost <" (to avoid allocating memory to be used for step-matrix characters) and "nstates dna" were used.

Phylip and MEGA, requiring 35–60 times as much as Oblong with default settings. Note that with the *–m2* switch, the amount of RAM used by Oblong remains almost constant as the number of characters increases, so that the difference between Oblong *–m2* and the other programs widens as there are more characters; for the largest data set that could be analysed with other programs on the present machine, Oblong used 26 times less memory than Parsimonator, the closest competitor. For the largest data set tested (30 million characters and 50 taxa, or 1500 Mb on disk), the memory used by Oblong *–m5* was 6388 kb—about 0.004× the size of the data set on disk (550 times less than expected for Parsimonator, 2500 times less than for TNT).

*Model-based programs*. Maximum-likelihood, Bayesian, or distance-based programs cannot be strictly compared with parsimony programs, as their search techniques are often very different, and the memory and CPU requirements of maximum-likelihood are bound to be much more intense than for equal weights parsimony. Although direct comparisons are difficult to make, the amounts of memory used by those programs are shown in Table 3. The program with the most modest memory requirements is RAxML (as already shown by Stamatakis et al., 2008), but even in that case, it is clear that phylogenomic data sets cannot be analysed on standard computers, as they would require very long processing times and extremely large amounts of memory.

Table 3
Memory consumption (in thousands of kb) of several model-based programs, for a data set of 100 000 characters and 50 taxa, and ratio of RAM used to the data set size on disk (in parentheses)

| Garli | RAxML | PhyML | FastTree | MrBayes | MEGA |
|---|---|---|---|---|---|
| 457.76 (93×) | 171.54 (35×) | 1247.71 (255×) | 505.64 (103×) | 416.07 (85×) | 1466.59 (299×) |

In the case of Garli vers. 2.0 the JC69 model and the genthreshfortopoterm = 100 option were used. In the case of RAxML vers. 7.2.6 (the most recent available Windows version) the analysis (a single replication) was done with –f d, with the GTRCAT model (the least memory-consuming option, according to the program documentation). In the case of PhyML vers. 3.1 the JC69 model with a single rate was used, starting from a BioNJ tree and swapping with SPR. Fasttree vers. 2.1.7 was run with the default settings. For MrBayes vers. 3.2.1 all the settings were default, except "lset nst=1" and the number of generations was set to just 1000 (as the goal was only measuring memory consumption, not the time for a full analysis). For MEGA vers. 5.1 the model was set to JC69 with uniform rates, SPR level 3, starting from an MP tree, and a "very strong branch swap filter".

## Conclusions

Oblong is the first program for phylogenetic analysis designed to reduce to a minimum the amount of RAM needed for running data sets with very large numbers of characters. Important efforts at lessening the memory requirements have been done by some programmers, but clearly not as the primary goal of the program, and not to the degree that millions of characters can be analysed on normal computers. For example, Stamatakis et al. (2012) used in RAxML-light a "subtree equality vector technique" which reduced the RAM required to calculate the likelihood of a tree from 66 Gb to 26 Gb, and a "recomputation technique" that does not store all ancestral probability vectors and allowed the RAM requirement to be reduced from 1 Tb to "only" 256 Gb.

All the ideas used in Oblong to reduce the need for RAM are simple, although it is not clear whether they can be adapted for methods of phylogenetic analysis other than parsimony. The main reason why periodically reloading the matrix from disk does not excessively slow down calculations in Oblong is because many rearrangements can be derived for each load—and this is most easily achieved under TBR or full SPR. Programs that use other methods (such as the "local" SPR moves of RAxML, with a tree-neighborhood size increasing linearly with $T$ as in NNI, not with $T^2$ as in full SPR or $T^3$ as in TBR) may perhaps suffer a more serious decrease in performance.

Although Oblong is not designed primarily for speed, it is nonetheless relatively fast, and it should greatly facilitate analyses with large numbers of characters on inexpensive computers. The program and code (open under GPL) are available at http://www.zmuc.dk/public/phylogeny/oblong; precompiled binaries for 32-bit Windows, and 64-bit Linux and Mac, are also included in the package.

## Acknowledgements

I thank Salvador Arias, Santiago Catalano, Joel Cracraft, and Claudia Szumik for discussion and for their interest in this project; the comments by reviewers (W. Wheeler and anonymous) also helped improve the manuscript. The research was carried out as part of the projects PICT 01314 (from Agencia Nacional de Promoción Científica y Tecnológica), PIP 0687 (from Consejo Nacional de Investigaciones Científicas y Técnicas), and "Assembly and evolution of the Amazonian biota and its environment: an integrated approach" (from National Science Foundation, National Aeronautics and Space Administration, and Fundaçao de Amparo à Pesquisa do Estado de São Paulo).

## References


Bremer, K., 1994. Branch support and tree stability. Cladistics 10, 295–304.

Farris, J., 1971. Methods for computing Wagner trees. Syst. Zool. 34, 21–24.

Farris, J., Goloboff, P., 2008. Is REP a measure of "objective support"? Cladistics 24, 1065–1069.

Farris, J., Albert, V., Källersjö, M., Lipscomb, D., Kluge, A., 1996. Parsimony jackknifing outperforms neighbor-joining. Cladistics 12, 99–124.

Felsenstein, J. 2005. PHYLIP (Phylogeny Inference Package) version 3.6. Distributed by the author. Department of Genome Sciences, University of Washington, Seattle, available at http://evolution.genetics.washington.edu/phylip.html.

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

Goloboff, P., 1996. Methods for faster parsimony analysis. Cladistics 12, 199–220.

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

Goloboff, P., 2002. Optimization of polytomies: state set and parallel operations. Mol. Phylogenet. Evol. 22, 269–275.

Goloboff, P.A., Catalano, S.A., Marcos Mirande, J., Szumik, C.A., Salvador Arias, J., Källersjö, M. and Farris, J.S. 2009. Phylogenetic analysis of 73 060 taxa corroborates major eukaryotic groups. Cladistics 25, 211–230.

Goloboff, P., Farris, J., 2001. Methods for quick consensus estimation. Cladistics 17, S26–S34.

Goloboff, P., Farris, J., Källersjö, M., Oxelmann, B., Ramírez, M., Szumik, C., 2003a. Improvements to resampling measures of group support. Cladistics 19, 324–332.

Goloboff, P., Farris, J., Nixon, K. 2003b. TNT: Tree Analysis Using New Technology. Program and documentation, available at http://www.zmuc.dk/public/phylogeny/TNT.

Goloboff, P., Farris, J., Nixon, K., 2008. TNT, a free program for phylogenetic analysis. Cladistics 24, 774–786.

Guindon, S., Gascuel, O., 2003. A simple, fast and accurate algorithm to estimate large phylogenies by maximum-likelihood. Syst. Biol. 52, 696–704.

Guindon, S., Dufayard, J., Lefort, V., Anisimova, M., Hordijk, W., Gascuel, O., 2010. New algorithms and methods to estimate maximum-likelihood phylogenies: assessing the performance of PhyML 3.0. Syst. Biol. 59, 307–321.

Huelsenbeck, J., Ronquist, F., 2001. MrBayes: Bayesian inference of phylogenetic trees. Bioinformatics 17, 754–755.

Huson, D., Richter, D., Rausch, C., Dezulian, T., Franz, M., Rupp, R., 2007. Dendroscope: an interactive viewer for large phylogenetic trees. BMC Bioinformatics 8, 460–465.

Kumar, S., Stecher, G., Peterson, D., Tamura, K., 2012. MEGA-CC: Computing Core of Molecular Evolutionary Genetics Analysis program for automated and iterative data analysis. Bioinformatics 28, 2685–2686.

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

Page, R., 1996. TREEVIEW: an application to display phylogenetic trees on personal computers. Comput. Appl. Biosci. 12, 357–358.

Price, M., Dehal, P., Arkin, A., 2009. FastTree: computing large minimum-evolution trees with profiles instead of a distance matrix. Mol. Biol. Evol. 26, 1641–1650.

Stamatakis, A., 2006. RAxML-VI-HPC: maximum-likelihood-based phylogenetic analyses with thousands of taxa and mixed models. Bioinformatics 22, 2688–2690.

Stamatakis, A. 2011. Parsimonator: a fast open-source parsimony program. available at http://sco.h-its.org/exelixis/countParsimonator102.php.

Stamatakis, A., Hoover, P., Rougemont, J., 2008. A rapid bootstrap algorithm for RAxML web servers. Syst. Biol. 57, 758–771.

Stamatakis, A., Aberer, A., Goll, C., Smith, S., Berger, S., Izquierdo-Carrasco, F., 2012. RAxML-Light: a tool for computing terabyte phylogenies. Bioinformatics 28, 2064–2066.

Swofford, D. 1998. PAUP*: Phylogenetic Analysis Using Parsimony (* and Other Methods), version 4. Sinauer Associates, Sunderland, MA.

Wheeler, W., 2010. Distinctions between optimal and expected support. Cladistics 26, 657–663.

White, W., Holland, B., 2011. Faster exact maximum parsimony search with XMP. Bioinformatics 10, 1359–1367.

Zwickl, D.J., 2006. Genetic algorithm approaches for the phylogenetic analysis of large biological sequence datasets under the maximum-likelihood criterion. PhD dissertation, The University of Texas, Austin, TX. available at http://garli.googlecode.com.