```
                                                                    |
                                                                    |
                 (\                                                 |
        . ` ' {(((-](8                                              |
       '         (/                                                 |
       `.                                                           |
         .                  ANAGALLIS                               |
          `.                                                        |
            .          A PROGRAM FOR PARSIMONY ANALYSIS             |
           ,               OF CHARACTER HIERARCHIES              '  |  '
          .                                                       \ ^ /
         '                                                     `---[x]---'
         ` .  .                       _._                       ,'/°-°\`.
             ` .                   _ /   \ _                   / |     | \
                ,                .'  \   /  `.
                 .        , ' ` . \ _.[o]._ /
                  ` . .            /   |   \
                                   \__/ \__/
                                   '       '
```

```
                              v1.02a - 27 May 2020
                provided as is and without warranty of any kind
                    Jan De Laet, Gothenburg Botanic Garden

              This documentation dump was generated with command
                  'help dump of anagallis_1.02a_27MAY2020 w80'


             It is formatted to view with a non-proportional font
                          and 80 characters per line
```

```
                                        __
                                (\  ,--'  `--.
                                 '><    - ;°-<
                                (/  `--.__,.-'
```

```
USAGE anagallis [--help][-c][-h][-s][-v] [commands]
=====
 --help    Show usage information and program options, and exit
 -c        Start the program with context-sensitive help enabled (as if command
           'program set context =' ('psc =') is executed)
 -h        Show usage information and program options, and exit
 -s        Suppress display of program header at the start of an interactive
           session
 -v        Show program version and exit
 commands  A command sequence as it would be entered from the program prompt
           (use ';' as command separator). To pass the commands properly,
           characters with special meaning in the shell from which the program
           is called (such as ';' or '*' in most shells) must be escaped. It
```

```
          mostly suffices to put the sequence between single quotes.
```

After the command sequence is executed, the program prompt is displayed. To execute commands in file 'fname', put command 'script execute start fname' ('sestafname') in the command sequence. Full batch mode can be achieve by including command 'program quit' ('pq') in the command sequence or in the command file.

A list of known bugs and issues is maintained at www.anagallis.be/anagallis.

Use command 'help summary' ('hs') for an overview of commands and help topics ('hsc' for just commands, 'hst' for just topics), 'help dump f fname' ('hd f fname') for a dump of all documentation to file 'fname', 'help command - help' ('hc-h' or '?h') for more information about the help command.

Command 'program set context =' ('psc=') enables context sensitive help.

Some commands to get general information about character hierarchies:

```
   * basic theory                   'help topic - background theory'
   * outline of the main algorithm  'help topic - background algorithm'
   * state reconstructions          'ht - br'
   * scope of the current algorithm 'ht - bs'
```

(Use 'help topic r - background' to view all this with one command.)

Some help commands to get more information about how to work with character hierarchies:

```
   * read character data                          '? crn'
   * define character hierarchies                 '? cps'
   * search for optimal trees                     '? tseam'
   * list summary scores of character hierarchies '? csc'
   * plot character hierarchy optimizations       '? cdp'
```

OVERVIEW OF HELP TOPICS AND COMMANDS

```
====================================
```

> name, shortest unambiguous abbreviation, short description

Help topics

```
-----------
  background          b      Background
    theory            bt     Basic theory
    algorithm         baa    Outline of the main optimization algorithm
    reconstructions   br     Aggregate and non-aggregate final statesets
    scope             bs     Scope and limits of the current algorithm
  program             p      Program
    commands          pc     Command structure
    treebuffer        ptr    Tree buffer
    input             pi     Input
      native          pinn   anagallis data formats
      import          pii    Importing other data formats
    tntmode           ptn    TNT mode
  references          r      References
```

```
Regular commands
----------------
  "                 "       echo the input until an end-of-command character
  #                 #       skip input until the end of the input line
  >                 >       provide context-sensitive help (only available
                            from the command prompt under 'program set context
                            =' or 'psc=')
  ?                 ?       shorthand for 'help command -': use '?xyz' for
                            basic help about command 'xyz'
  characters        c       current number of characters and terminals; has
                            optional subcommands
    diagnose        cd      show character state optimizations on tree nodes;
                            requires a subcommand
      plot          cdp     show tree plots of character state optimizations
      tabulate      cdt     show tables of character state optimizations
    properties      cp      set/show basic character properties or show
                            derived character properties; requires a
                            subcommand
      minmax        cpm     list minimum and maximum number of steps for all
                            characters outside character hierarchies
      set           cps     set/show basic character properties (defaults:
                            prior weight 1, active, non-additive, not part of
                            a character hierarchy)
    read            cr      read character data; requires a subcommand (use
                            command 'import' to import TNT or nexus character
                            data)
      alphanumeric  cra     read character data with up to 30 regular
                            character states coded as 0-9 and a-t (or A-T)
      numeric       crn     read character data with up to ten regular
                            character states coded as 0-9
    score           csc     a summary of the scores of all characters and
                            character hierarchies on one or more trees
    show            csh     show the current dataset
  help              h       show basic usage information and program options;
                            has optional subcommands to get more detailed
                            information
    command         hc      show information about a specific regular command
    dump            hd      show all built-in program documentation
    summary         hs      overview of available commands and/or help topics
    topic           ht      get information about a specific help topic
  import            i       import data from other file formats; requires a
                            subcommand
    nexus           in      execute a nexus datafile (supported nexus subset
                            still empty in this version)
    tnt             it      execute a TNT datafile (limited support for a tiny
                            subset of TNT commands)
  log               l       name and status of log file, if there is one; has
                            optional subcommands
    pause           lp      suspend output to the logfile
    resume          lr      resume output to the logfile
    start           lsta    open a log file (append mode by default)
    stop            lsto    close the current logfile
  optimality        o       set/show optimality criterion; requires a
                            subcommand
    set             os      overview of settings that relate to the optimality
                            criterion; has optional subcommands
      deviation     osd     set/show allowed deviation from optimality in tree
                            searches and tree buffer cleaning
      searchmode    oss     set/show search mode: look for best (=, default)
```

```
                                or worst (-) trees
program             p           quit the program or set/show general settings;
                                requires a subcommand
  quit              pq          quit the program
  set               ps          overview of general settings; has optional
                                subcommands
    context         psc         set/show if context-sensitive help (command '>')
                                is available from the command line (=) or not (-,
                                default)
    longlists       psl         set/show if lists of command completions are
                                multilevel (=) or not (-, default)
    randomseed      psr         set/show seed for generator of pseudorandom
                                numbers
    shortcommands   pss         set/show if command abbreviations are allowed (=,
                                default) or not (-)
    tntmode         pst         set/show if TNT mode is on
    unicode         psu         set/show if tree plotting uses multibyte UTF-8
                                characters (=, default) or not (-)
script              s           overview of open script files; has optional
                                subcommands
  execute           se          overview of script files that are opened for
                                execution; has optional subcommands
    pause           sep         temporarily suspend execution of current script
                                file and get interactive input
    resume          ser         resume reading from the current script file that
                                is open for execution
    start           sesta       open a script file and start executing its
                                commands
    stop            sesto       close the current script file that is open for
                                execution
  record            sr          name and status of the file that is open for
                                recording commands, if there is one; has optional
                                subcommands
    pause           srp         temporarily suspend writing to the current file
                                for recording commands
    resume          srr         resume recording commands to the script file for
                                recording
    start           srsta       open a file for recording commands (append mode by
                                default)
    stop            srsto       close the current file for recording commands
trees               t           current number of trees in memory; has optional
                                subcommands
  consense          tc          calculate consensus trees; requires a subcommand
    majority        tcm         majority rule consensus tree
    strict          tcs         strict consensus tree
  read              trr         read trees in parenthetical notation (use command
                                'import' to import TNT or nexus trees)
  score             tsc         list the score of the current data on one or more
                                trees
  search            tsea        search trees; requires a subcommand
    mult            tseam       do one or more replicates of building a tree and
                                swapping it (spr or tbr)
    swap            tseas       swap trees from the tree buffer (spr or tbr)
  select            tsel        manipulate trees in the tree buffer; requires a
                                subcommand
    best            tselb       discard suboptimal trees
    delete          tseld       discard the trees in the specified tree scopes
    keep            tselk       discard the trees that are outside the specified
                                tree scopes
```

```
      unique          tselu   discard duplicate trees
    set               tset    overview of tree related settings; has optional
                              subcommands
      current         tsetc   set/show the default tree that is for example used
                              when showing trees or character optimizations on
                              trees
      outgroup        tseto   set/show terminal(s) to be used as outgroup(s)
                              when showing trees
      width           tsetw   set/show default maximum width of a line when
                              plotting trees
      zerocollapse    tsetz   set/show the rule for collapsing zero-length
                              branches
    show              tsh     show trees; requires a subcommand
      plot            tshp    plot trees using character graphics
      write           tshw    write trees in parenthetical notation
```

TNT mode commands

```
> only available in TNT mode (command 'program set tntmode =' or 'pst=') and in
imported TNT files (command 'import tnt' or 'it')
  ccode         c     set/show character settings
  help          h     show documentation for the commands that are available in
                      TNT mode and in imported TNT files
  nstates       n     set the TNT default datatype
  program       p
    set         ps
      tntmode   pst   set/show if TNT mode is on
  quit          q     leave TNT mode, go back to regular mode
  tread         t     read trees in parenthetical notation (numbering of
                      terminals starts from 0)
  xread         x     read alphanumeric or dna data (no support for interleaved
                      data)
```

Use 'help topic - topicname' ('ht-topicname') for more information about topic 'topicname' (topicname can be abbreviated).
Use 'help command - commandname' ('hc-commandname') for more information about regular command 'commandname' (commandname can be abbreviated).

HELP TOPICS

1 Background

```
--> Subtopics
      background theory           bt   Basic theory
      background algorithm        baa  Outline of the main optimization
                                       algorithm
      background reconstructions  br   Aggregate and non-aggregate final
                                       statesets
      background scope            bs   Scope and limits of the current algorithm
```

1.1 Basic theory

The basic theory behind the main optimization algorithm in anagallis can be found in De Laet (2005, 2015; see also De Laet 2017). In these papers it is argued that the problems with missing characters or inapplicable data (Maddison 1993) disappear when parsimony is not seen as an approach that minimizes evolutionary changes but as an approach that maximizes homology. These two points of view are operationally equivalent in the absence of inapplicables, but that equivalence no longer holds in general with data that contain cases of inapplicability. This is mainly argued and discussed in the context of the analysis of unaligned sequence data, but also holds for inapplicables as they arise in the classic setting of morphological data (De Laet 2005: 110-111; De Laet 2015: 552-556). With inapplicables in the analysis of unaligned sequence data, the computational complexity of the optimization of homology on a given tree makes the use of heuristic approximations unavoidable. With inapplicables as they arise in morphology, the computational complexity is greatly reduced and an exact algorithms for the optimal homology score of character hierarchies with inapplicables become practically feasible. Anagallis is a program that provides tree evaluation and tree searches with an algorithm that yields exact scores for basic absence/presence character hierarchies.

The problems with inapplicable data can be exemplified using Maddison's (1993) well-known reatment of how to deal with tails in a group of terminals where a tail is either absent or present; and that, when present, can have different colors, textures, and so on. This can be seen as a character hierarchy with a more inclusive level of homology (absence or presence of a tail) and a less inclusive level of homology (tail color when tail is present; tail texture when tail is present, ...). In general, the less inclusive level can have homology statements about entire substructures that can be either absent or present, and each such substructure can in turn be at the root of a subhierarchy, with an even less inclusive level of homology at which further variability (or lack thereof) of the substructure is applicable. Further treatment only applies to empirical data that from a biological point of view can be argued to be properly conceptualized in such hierarchies.

Maddison's tail color example deserves some important caveats in that respect. First, color is in general not restricted to tails, and might equally well be conceptualized as part of one or more hierarchies (such as absence/presence of particular pigments that require certain precursors to be present, ...) that are independent from tail hierarchy. Such discussions are important but outside the scope of how to deal with a character hierarchy once it is accepted that it constitutes a prior hypothesis that deserves further examination, which is the focus here. Much depends on the specific group under study. Second, there is also the implicit assumption that there are good prior reasons to consider tails in this group of terminals homologous on a prior basis, even if they may differ in color, texture, and so on. If there would be prior grounds to reject homology of, for example, red tails and blue tails in this group of terminals, there would be two independent absence/presence character hierarchies: one for red tails, another for blue tails. Once again these are important issues but beyond the scope of this introduction. Related to the first issue is the general concern is that variability, or better lack thereof, should be treated at the right morphological level. One could for example choose to include the observation that the living cells of the observed tails contain ribosomes. While not wrong by itself, it is of wider applicability and the correct level to include this observation would be the living cells in the entire organism, not just in the tails. Without inapplicables (all terminals under study have tails), this would just be an uninformative character and no harm would be done by including it, even if coded at the wrong level. As will be clear, that does no longer hold when inapplicables are present.

Using Sereno's (2007, p. 573) distinction between neomorphic and transformational characters, the absence/presence of (sub)structures in a given set of terminals can be represented using neomorphic characters. Further variability of (sub)structures can be represented using transformational characters. So the general concept of a basic character hierarchy can be represented as a hierarchy with a single neomorphic character at its root and one or more transformational and/or neomorphic characters at each next level. Each such next level represents a less inclusive level of applicability, and at each level a special token such as a dash can be used to indicate inapplicability. This representation provides unambiguous and non-redundant descriptions of such hierarchies (at least as long no additional constraints are required; see the section on the Scope and limits of the current algorithm for a refinement of the basic model discussed here). Discussion of a simple concrete example (though not in terms of Sereno's terminology) can be found in De Laet (2015, Fig. 2).

Having a way to unambiguously describe a character hierarchy is one thing, how to optimize such a hierarchy on a tree another. Maddison (1993) well established that algorithms that can properly deal with characters that are applicable throughout a given set of terminals are in general insufficient for that, not just for often used coding strategies such as contingent coding, but for any way of coding he could think of (to be sure, he didn't to that in these terms and only dealt with hierarchies one level deep: a root level such as absence/presence of a tail; and a lower level describing further aspects of that structure; but that's what it amounts to). Examples of such algorithms are Farris' (1970) algorithm for linearly ordered characters, Fitch' (1971) algorithm for unordered characters, and Sankoff and Rousseau's (1975) step matrix algorithm. In practice, implementations of these algorithms often allow the user to treat a dash either as missing data or as an additional character state, but these are just two different ways to shoehorn inapplicability into algorithms that are themselves not in general applicable with inapplicable data, and that were not designed to treat such cases. Maddison described and discussed a number of cases in which different coding approaches for such algorithms give results that are hard to defend from a theoretical/biological/logical point of view. Because he couldn't find any combination of coding approach and algorithm that that would always give satisfactory results, he concluded that the ultimate solution would require the development of algorithms that would keep track of interactions among characters and that would count steps in characters only in regions of the tree where these characters are applicable (Maddison 1993, p. 580).

Before moving to the development of such algorithms, a fundamental question must be answered: what is it, from a biological and theoretical point of view, that one should optimize in such cases? As will be discussed right away, this is so because the notion of minimization of evolutionary changes or steps, often seen as the hallmark of parsimony, is fundamentally problematic when inapplicables are present. Discussions of coding techniques and algorithms for data with inapplicables in terms of evolutionary steps are therefore just operational guidelines in search of deeper ground. It is not a coincidence that, in the absence of a clear answer to the above question, about all papers that deal with inapplicability in terms of evolutionary steps, including Maddison's, conclude with practical recommendations that are known to be problematic in some cases but hoped to give reasonable results under most.

As an answer to this question I have proposed to maximize the number of independent pairwise homology statements that trees allow given the data at hand (De Laet 2005). The theoretical basis for this criterion is that it maximizes agreement between observational data (homology hypotheses as coded in the data set, ultimately based on observed similarity; see De Laet 2005 p. 83-84 for some discussion and pointers to the literature when it comes to the notion of similarity in this context) and explanation (any single tree with uniquely optimized inner nodes that explains the data by inheritance and common descent, a crucial notion in parsimony). The requirement that homology statements as coded in a dataset must be similarity-based ensures that the whole enterprise has an empirical basis. Given such prior tree-independent hypotheses of homology, the notion of homology on a given tree has this precise meaning: a feature that is observed to be shared among a group of terminals and that constitutes a prior hypothesis of homology can be considered homologous on that tree if that shared presence can be explained by inheritance and common descent on that tree; if it cannot be so explained, it is not homologous (Farris 1983, p. 18; see also Farris 2008). As such, it covers both presence and absence of structures and substructures (provided that these absences are considered at the appropriate level; De Laet 2015, p. 551) and further aspects of these structures and substructures when they are present.

The best trees for a given dataset are then the trees on which the highest number of independent pairwise similarity-based prior homology statements can be simultaneously interpreted as homology. Maximization is over independent pairwise such homology statements because these constitute the units of comparative content of a dataset (De Laet 2005, p. 89-91). When no inapplicables are present, this reduces to Farris's (1983) rationale for parsimony analysis (see e.g. De Laet 2015, p. 553). With inapplicables, it can be seen as an extension of that rationale that covers such cases as well, an extension in which the problems that Maddison (1993) described disappear (De Laet 2005, p. 110).

The difference between parsimony as an approach that minimizes evolutionary changes/events/transformations and parsimony as an approach that maximizes homology can be illustrated by example. Consider a clade with two deeply nested sister species that share an insert of 30 nucleotides that is not present in any other species in that clade. The inserted sequence is largely conserved but two base transformations have occurred nevertheless. Minimizing evolutionary changes, the shared presence of that insert cannot be considered a synapomorphy that unites these two species. This is so because a single insert of length 30 in the common ancestor of these sister species (which would constitute the synapomorphy that unites these species) followed by two base transformations amounts to three evolutionary events. If it is evolutionary events that are to be minimized, two independent such inserts, along the two terminal branches leading to these species, each of the exact observed 30 nucleotides, provides a better explanation (one less evolutionary event: just two insertions, no base transformations). So considering the insert as a synapomorphy that unites both species is suboptimal when just minimizing evolutionary events, even if it can explain why 28 out of these 30 bases are identical between the two species, a series of empirical observations that the 'optimal' solution of two independent inserts cannot explain. Similar examplesi but in the context of unaligned sequence data are discussed on pp. 111-114 of De Laet (2005). An example in the context of morphological data can be found in De Laet (2015, Fig. 2). Each such example can be 'fixed' by assigning different weights to different events, but such differential weighting is ad hoc and no general weighting approach seems to be able to cover all cases.

When it comes to explanation of empirical data, such examples show that minimization of evolutionary events or transformations in general breaks down when inapplicables are present. Maximization of homology, on the other hand, performs well in such examples. Postulating two independent origins of the sequence in the above example, for example, is way suboptimal from this point of view: none of the 28 shared presences of a specific base can be explained by common descent and inheritance, and neither can the shared presence of the insert of length 30. All these observed shared similarities can be explained as homology though when a single insert of length 30 in the common ancestor of these species is postulated.

Once it is accepted that most parsimonious trees are the trees that best explain empirical data in terms of common descent and inheritance, the question arises of how to proceed operationally to identify those trees. This is the technical side of the question of which optimality function to use to evaluate any given tree, a criterion that can then be optimized in a tree search. With inapplicables, the evaluation of a given tree itself already involves a non-trivial optimization: to maximize homology in a character hierarchy on a given tree, one has to minimize a self-consistent total cost or score on that tree that has three components (De Laet 2015, pp. 553; see also De Laet 2005: 105-108):

1. gains and losses of (sub)structures where these are applicable;
2. transformations in further characters of these (sub)structures where applicable;
3. subcharacters (regions of applicability for (sub)structures and further characters that describe them).

That optimal score on a tree is simply called the cost or score of the given character hierarchy on that tree. It reflects or implies one or more scenarios of evolutionary change that result in one or more optimal mixes of homology statements on that tree. Some of these are about absences and presences of entire (sub)structures and some about states in further characters of (sub)structures when present. The mix is optimal in the sense that any other evolutionary scenario will imply that the total amount of observed similarity that can be explained as homology on that tree will decrease (as just illustrated by example, the scenarios that lead to such optimal explanations are not necessarily minimum evolution scenarios). The best trees are then the trees with the lowest cost thus defined.

A somewhat surprising quantity in the above minimization is the number of subcharacters or regions in a tree where a character is applicable. Subcharacters have no biological meaning by themselves, they are just a quantity that pops up when deriving an expression for the amount of similarity that can be explained as homology on a given tree. Keeping track of numbers of subcharacters for all characters that are part of a character hierarchy is a major part of what the main optimization algorithm in anagallis does.

The notion of self-consistency here means that the overall explanation of the observed data must be free of internal contradictions. As pointed out by Maddison, it boils down to keeping track of interactions between the characters that are logically related. That's a second major job that the optimization in anagallis does. This is also a main difference with the algorithm of Brazeau et al. (2017). Theirs is, for reasons of computational simplicity, by design a single-character algorithm. So by definition it cannot guarantee overall internal consistency across entire character hierarchies.

The minimization that has to be performed on any given tree is a real co-minimization. Whenever evaluating a given tree for a given character hierarchy, it is for example not sufficient to first optimize gains and losses at the root level of that hierarchy and to use the regions of applicability thus defined as fixed constraints to count gains/losses, transformations and subcharacters at less inclusive levels. The best overall solutions for the hierarchy as a whole may well require optimizations of gains and losses at the root level of the hierarchy that, considered in isolation, are suboptimal.

The program provides a flexible way to define hierarchic relationships among the columns of a conventional dataset, using parenthetical notation with angle brackets. The syntax is such that the first character following an opening bracket is an absence/presence character that determines (in)applicability at less inclusive levels. Next follow simple nested characters or nested subhierarchies. There is no restriction on the number of nested levels or on the number of characters or subhierarchies at any level.

The absence/presence characters at any level are referred to as root characters because they are at the root of a (sub)hierarchy. The root character of the complete hierarchy is called the main or global root character of the hierarchy. Within each level, the characters that follow the root character are called the (directly) subordinate characters at that level. These include root characters of one level down: a root character of a nested level is a (directly) subordinate character one level up. Such nested root characters are called complex subordinate characters (as opposed to simple subordinate characters).

As an example,

```
    <1 5 6 <7 8>>
```

is a character hierarchy with main root character 1. Characters 5, 6 are simple subordinate characters at that level. Character 7 is a complex subordinate character at that level. It is the root character of a subhierarchy that has character 8 as a simple directly subordinate character.

Once a hierarchy is defined, subsequent tree optimizations and tree searches will take these relationships into account and provide scores according to the above criterion.

1.2 Outline of the main optimization algorithm

So to find the score of a character hierarchy on a tree such that homology is maximized, one has minimize the self-consistent total cost that is the sum of these three components:

1. gains and losses of (sub)structures where these are applicable;
2. transformations in further characters of these (sub)structures where applicable;
3. subcharacters (regions of applicability for (sub)structures and further characters that describe them).

This section provides an outline of the algorithm to calculate that score for character hierarchies with equally weighed characters (weighting factor 1) as currently implemented. The program allows different characters of a character hierarchy to be differentially weighted using user-specified weights. Such weighted scores are obtained by applying these weights to the scores of the individual characters of a hierarchy.

The main root character of a properly defined character hierarchy is by definition applicable in all terminals, so in any optimal solution it will be applicable in all inner nodes as well (if not, the score can be improved, leading to a contradiction). So the search can be constrained to solutions where the main root character is in a single subcharacter. The algorithm starts with a preliminary full (downpass and uppass) and unconstrained Fitch optimization of the root absence/presence character on that tree (under the given conditions, unconstrained Fitch optimization always results in a single subcharacter).

Fitch optimization of an absence/presence character can result in final statesets that include both absence and presence. If that is the case, the algorithm preliminary resolves these ambiguities as absence (this does not affect the calculated cost). This results in a number of inner nodes that are reconstructed as absence and a number of nodes that are reconstructed as presence, thus defining initial or preliminary current regions of absence and initial or preliminary current regions of presence on the given tree. The nodes that are reconstructed as presence at this point will be reconstructed as presence in any optimal solution for the hierarchy as whole (it can be shown that the cost of the full hierarchy will increase if any combination of these nodes is switched to absence). For nodes that are reconstructed as absence at this point this only holds in initial regions of absence that are bounded by at most one neighbouring region of presence. In initial regions of absence that are bounded by more than one region of presence, some or all nodes may have to be switched to presence in order to obtain the optimal score for the full hierarchy. Optimal re-assignments to presence in such initial regions of absence can be calculated one region at a time (I was too harsh on my algorithm in the documentation of v1.01 when it came to this, high time to get that paper done...). In other words, there are no interactions between different such current preliminary regions of absence for this optimization problem.

The optimal reassignments within any given preliminary region of absence, if any, are determined as follows. First, calculate the preliminary cost of the full hierarchy under the current constraints (the root character as preliminary optimized at this point; this includes optimal re-assignments in preliminary regions of absence that have already been looked at). Use this score to initialize the current best global score in the search that follows. That search is through the possible set of unit switches in the current region that have the potential to improve the current best global score. In this, a unit switch is defined as a subset of nodes of the current region that are switched from absence to presence (the initial situation corresponds to the empty unit switch; the nodes that are not switched are the complement of the unit switch, so the unit switch fully defines the assignments in the region). Those unit switches that yield the best improvements of the current best score will determine the final statesets in the current preliminary region of absence. To find the optimal score, it is sufficient to consider just one optimal unit switch in the current region of absence (and to effectively perform its re-assignments) before moving to the next such region. In that way, the current best cost after all initial regions of absence have been optimized in this way is the optimal cost of the full hierarchy on the given tree.

Here are some details about the search space in any preliminary region of absence. Consider a unit switch and assume that there is at least one switched node that does not have at least two neighbouring nodes that have also been switched or that belong to neighbouring regions of presence. It can be shown that the derived unit switch in which that node is not switched will have a better score. This procedure of switching such nodes back can be repeated until al switched nodes have at least two neighbouring nodes that have also been switched or that belong to neighbouring nodes of presence (including the case where no switched nodes remain). This final derived switch is a unit switch for which the following holds: for each switched node, there exists a path through the original region of absence that connects two neighbouring regions of presence such that all nodes on that path have been switched (a neighbouring region of presence for a given region of absence is a region of presence that is connected to that region of presence by a single edge). To find all optimal unit switches, it is then sufficient to search through the set of unit switches that have this property (note that, as a corollary, any such unit switch can increase the number of regions of absence and will decrease the number of regions of presence). For ease of further discussion, unit switches with this property are called proper unit switches. This still amounts to a search space that grows more than linearly with the number of neighbouring regions of presence. This explains why the program in general takes measurably more time to optimize random trees than it takes to optimize optimal or near-optimal trees. Some remarks on how the set of relevant unit switches is generated in the current implementation follow later in this section.

The cost of the full hierarchy under the current constraints is the sum of the current cost of the main root character, the costs of its simple subordinate characters, and the costs of the subhierarchies that start at its complex subordinate characters.

The current cost of the main root character is the sum of its number of subcharacters (always equal to one for a main root character) plus the cost of its initial Fitch optimization as adapted to reflect unit switches that may already have been performed. The current preliminary optimization of this main root character implies one or more preliminary regions of applicability or subcharacters for its directly subordinate characters, both simple and complex.

The score of a simple subordinate character is the sum of that number of subcharacters and the number of transformations in that subordinate character within those subcharacters. Depending on user-specified settings for the character, the number of transformations in a subcharacter is obtained as the regular Fitch cost or as the regular Farris cost within that subcharacter. In practice, the total number of transformations within all subcharacters can be obtained by a traditional downpass that is modified to detect boundaries of subcharacters (similar and somewhat more elaborate modifications are required in the uppass when, in a later stage, all optimal statesets are calculated).

The cost of a subhierarchy that starts at a complex subordinate character is calculated in the same way as the cost of the full hierarchy but with the added twist that the root character of a subhierarchy can have more than one subcharacter, a dynamic constraint from one level up in the hierarchy. So the full Fitch optimization of the root character of a subhierarchy, the first step in such calculations, has to be performed within each of those subcharacters.

To check if a given unit switch in a given region of absence as implied by a preliminary optimization of a root character improves the score of the (sub)hierarchy that starts at that root character, the current implementation does a complete re-optimization of that (sub)hierarchy. It could re-use the previous calculations outside the given region of absence, but that algorithmic optimization is not programmed yet.

The following paragraphs provide some notes on the procedure used to generate the set of unit switches that have to be checked in a given region of absence. As discussed, these are all unit switches for which the following holds: for each switched node, there exists a path through the original region of absence that connects two neighbouring regions of presence such that all nodes on that path have been switched. This property can be used to generate that set, an approach that is followed in the current implementation of the algorithm.

Given a region of absence and its neighbouring regions of presence as currently optimized, consider a set of non-singleton and non-intersecting subsets of neighbouring regions of presence. For each such subset, change the preliminary assignments (from absence to presence) in the region of absence for all nodes that are on the path between any two members of that subset (for any set of subsets, that can be done in a single downpass through that region). By generating and examining all sets of non-singleton and non-intersecting subsets of neighbouring regions of presence, it is guaranteed that each relevant unit switch will be generated at least once. The drawback of this approach is that some unit switches can be generated more than once because different sets of subsets can result in the same unit switch. The current implementation therefore checks some rules to identify sets of subsets that can be skipped because there exists a set of subsets that will not be skipped and that results in the same unit switch.

In the current implementation of the algorithm, all possible sets of non-singleton and non-intersecting subsets of neighbouring regions of presence up to a cardinality of five are checked, the smallest number that suffices to guarantee that each relevant unit switch will be generated at least once as long as no more than eleven neighbouring regions of presence are involved during any stage of the optimization of any of its absence/presence characters. This condition is certainly met when no unconstrained Fitch optimization of an absence/presence character has such a region, an overly strict condition that is easily checked a posteriori. With more than eleven neighbouring regions of presence, an unlikely situation for empirical datasets on optimal or near-optimal trees, there may be optimal solutions that that can only be found with sets of subsets that have a cardinality higher than five. When this is the case, the current implementation can be considered to provide a heuristic approximation, even in the absence of interactions between different initial regions of absence: reconstructed statesets at inner nodes and reported scores may be optimal, but there is no guarantee. Whenever this happens during tree search or tree evaluation, it is flagged in the output.

1.3 Aggregate and non-aggregate final statesets

At the point where the algorithm identifies a single optimal solution for a given character hierarchy on a given tree, each initial region of absence of the main root character of that hierarchy reflects a single unit switch. In combination with the initial regions of presence, these unit switches amount to an optimal reconstruction of the main root character. This reconstruction defines current sets of subcharacters or regions of applicability for the subordinate characters, both simple and complex.

Within each current subcharacter of a simple subordinate character, final statesets can initially be calculated in the same way as done for characters outside character hierarchies. Some additional steps are then required to detect and properly deal with boundaries between regions of applicability (subcharacters) and regions of inapplicability, but these do not affect the logic of calculating the final statesets themselves.

A complex subordinate character is by definition the root character of a subhierarchy. Within each current subcharacter of such a subordinate root character, the subhierarchy that starts there has the same recursive structure, including inner node state reconstructions, as the hierarchy as a whole. There's just the additional twist that the reconstruction of the subordinate root character requires some extra steps to detect and properly deal with boundaries between regions of applicability and regions of inapplicability. These are not explicitly required at the outer level because there are no regions of inapplicability at that level.

For each character of the hierarchy, the inner node reconstructions at that point in the algorithm are called the non-aggregate (final) statesets. With multiple global solutions of a character hierarchy, there are one or more inner nodes where multiple non-aggregate statesets exist for one or more characters. The union of all these for a given character at a node yield the aggregate (final) stateset for that character at that node. Calculation of zero-length branches is currently done on the basis of such aggregate statesets.

Below, the difference between aggregate and non-aggregate statesets is illustrated with some examples of increasing complexity. The first three consists of a dataset that contains a character hierarchy, and such that there is just a single most parsimonious tree under equal weights. Lines with output of the program (non-aggregate and aggregate statesets on the optimal tree) are slightly indented and marked with '>' (for lines that would wrap, some manual editing has been added to maintain a clear layout). Final statesets are plotted using command characters diagnose plot (cdp), a command that by default plots aggregate statesets. Plots of non-aggregate statesets are obtained by adding option 'u'. The final example just contains anagallis statements to read a dataset, read a character hierarchy, read four trees, and generate non-aggregate and aggregate statesets for the hierarchy on those trees. When executed, the output contains some more complex examples of the visualization problems that can arise when plotting non-aggregate statesets.

Example 1

This is a simple example of a character hierarchy that has more than one optimization on a tree.

Data set:

```
  out 00000 11
  A   10000 11
  B   11000 11
  C   11100 0-
  D   11111 0-
  E   11111 12

  character hierarchy <6 7>
```

Optimal tree:

```
  ┌─ A
  ├─ out
  └───┬─ B
      └───┬─ C
          └───┬─ D
              └─ E
```

Below are the non-aggregate statesets for the character hierarchy on the the most parsimonious tree. As pointed out in the output, the subordinate character is shown first. It includes the number of subcharacters and transformations for that character on that plot. The optimization of the root character that provides the proper context for that nested optimization follows. This one includes the number of subcharacters and gains/losses. There are two different global solutions, so this simple hierarchy of optimizations is shown twice.

```
 >* non-aggregate final statesets for character hierarchy <6 7> on tree 1
 >  selected characters: all
 >
 >  the optimization is presented depth first
 >  at each level, the local context for the optimization of the nested level
 >  is provided by the optimization of the parent level that follows
 >
 >     + non-aggregate statesets for simple subordinate character 7 on tree 1
 >       local optimization
 >       2 subcharacters, 0 transformations
 >       ┌[1] out
 >       └[1]──┬[1] A
 >             └[1]──┬[1] B
 >                   └[-]──┬[-] C
 >                         └[-]──┬[-] D
 >                               └[2] E
 >  + non-aggregate statesets for main root character 6 on tree 1
 >    optimization 1 of 2
 >    1 subcharacter, 2 gains/losses
 >    ┌─[1] out
 >    └─[1]─┬─[1] A
 >          └─[1]─┬─[1] B
 >                └─[0]─┬─[0] C
 >                      └─[0]─┬─[0] D
 >                            └─[1] E
 >  ______________________________________________________________
 >
 >
 >     + non-aggregate statesets for simple subordinate character 7 on tree 1
 >       local optimization
 >       1 subcharacter, 1 transformation
 >       ┌─[1] out
 >       └─[1]─┬─[1] A
 >             └─[1]─┬─[1] B
 >                   └─[12]─┬─[-] C
 >                          └─[12]─┬─[-] D
 >                                 └─[2] E
 >
 >  + non-aggregate statesets for main root character 6 on tree 1
 >    optimization 2 of 2
 >    1 subcharacter, 2 gains/losses
 >    ┌─[1] out
 >    └─[1]─┬─[1] A
 >          └─[1]─┬─[1] B
 >                └─[1]─┬─[0] C
 >                      └─[1]─┬─[0] D
 >                            └─[1] E
```

The second plot of the subordinate character contains an example of the special provisions that may be required along the boundaries between a region of applicability and its neighbouring regions of inapplicability. In this case there is one region of applicability and two neighbouring regions of inapplicability (leaf node C and leaf node D). These two neighbouring regions of inapplicability connect to two nodes of the region of inapplicability that are directly connected themselves. The net effect is that the required transformation between state 1 to state 2 in the region of applicability cannot be pinpointed to a single branch in that region, resulting in an ambiguous reconstruction in the two inner nodes near that boundary. This, in turn, has consequences for the detection and collapsing of zero-length branches (see command 'trees set zerocollapse').

The corresponding aggregate statesets are given below. The non-aggregate statesets on which each of these plots is based have in general different numbers of subcharacters and transformations or gains/losses. No attempt is made to summarize the possible ranges of these numbers in these plots.

```
 >* aggregate final statesets for character hierarchy <6 7> on tree 1
 >  selected characters: all
 >
 >     + aggregate statesets for simple subordinate character 7 on tree 1
```

```
>       ┌[1] out
>       └[1]──┬[1] A
>             └[1]──┬[1] B
>                   └[12-]──┬[-] C
>                           └[12-]──┬[-] D
>                                   └[2] E
>  + aggregate statesets for main root character 6 on tree 1
>    ┌[1] out
>    └[1]──┬[1] A
>          └[1]──┬[1] B
>                └[01]──┬[0] C
>                       └[01]──┬[0] D
>                              └[1] E
```

Example 2

This is an example where the root character has two initial regions of absence that are optimized independently. In general, each initial region of absence for a root character can have multiple optimizations of the subhierarchy that starts at that root character. To evade the combinatorial explosion that would follow from showing every possible combination of optimizations in these regions, these optimizations are shown one region at a time, using the '+' character as a placeholder for the possible optimizations in the other regions. The drawback is that the plots can no longer be accompanied by their numbers of subcharacters and transformations or gains/losses because these numbers depend on specific values for the placeholder. For regions that have only a single optimization, such as region two in this example, that optimization could be directly plugged in into the plots for the other regions, making the presentation of the results less complex. That algorithmic refinement is not available in this version.

Data set:
```
  out 0000000000 11
  A   1000000000 11
  B   1100000000 11
  C   1110000000 0-
  D   1111100000 0-
  E   1111100000 12
  F   0000010000 11
  G   0000011000 11
  H   0000011100 0-
  I   0000011111 0-
  J   0000011111 11

  character hierarchy <11 12>
```

Optimal tree:
```
 ┌── out
 ├──┬─── A
 │  └──┬─── B
 │     └──┬─── C
 │        └──┬─── D
 │           └─── E
 └──┬─── F
    └──┬─── G
       └──┬─── H
          └──┬─── I
             └─── J
```

Below are the non-aggregate statesets for the character hierarchy on the the most parsimonious tree.

```
 >* non-aggregate final statesets for character hierarchy <11 12> on tree 1
 >  selected characters: all
 >
 >  the optimization is presented depth first
 >  at each level, the local context for the optimization of the nested level
 >  is provided by the optimization of the parent level that follows
 >
 >     + non-aggregate statesets for simple subordinate character 12 on tree 1
 >       local optimization
 >       '+': placeholder in region(s) as defined at level 1 of the hierarchy
 >            valid values are nested under the optimizations at that level
 >       ┌─[1] out
 >       └─[1]─┬─[1]─┬─[1] A
 >             │     └─[1]─┬─[1] B
 >             │           └─[-]─┬─[-] C
 >             │                 └─[-]─┬─[-] D
 >             │                       └─[2] E
 >             └─[1]─┬─[1] F
 >                   └─[1]─┬─[1] G
 >                         └─{+}─┬─[-] H
 >                               └─{+}─┬─[-] I
 >                                     └─[1] J
 >
 >  + non-aggregate statesets for main root character 11 on tree 1
 >    there are 2 regions that are optimized independently using the
 >    final optimal statesets at the borders of these regions as plotted below
 >
 >    such regions may have multiple non-trivial non-aggregate final statesets
 >    these statesets are therefore plotted separately for each such region
 >    '+' serves as a placeholder for optimizations in the other regions
 >
 >    the regions are defined in the first plot that follows
 >    nodes that are not labelled in that definition are outside
 >    such regions and have the same optimization in all following plots
 >
 >    ┌─ out
 >    └──┬──┬─ A
 >       │  └──┬─ B
 >       │     └─1─┬─ C
 >       │         └─1─┬─ D
 >       │             └─ E
 >       └──┬─ F
 >          └──┬─ G
 >             └─2─┬─ H
 >                 └─2─┬─ I
 >                     └─ J
 >
 >    character 11, region 1: optimization 1 of 2
 >    '+': placeholder in other region(s) as defined at this level (1)
```

```
>    ┌─[1] out
>    └─[1]─┬─[1]─┬─[1] A
>          │     └─[1]─┬─[1] B
>          │           └─[0]─┬─[0] C
>          │                 └─[0]─┬─[0] D
>          │                       └─[1] E
>          └─[1]─┬─[1] F
>                └─[1]─┬─[1] G
>                      └─{+}─┬─[0] H
>                            └─{+}─┬─[0] I
>                                  └─[1] J
>    ____________________________________________________________
>
>
>      + non-aggregate statesets for simple subordinate character 12 on tree 1
>        local optimization
>        '+': placeholder in region(s) as defined at level 1 of the hierarchy
>             valid values are nested under the optimizations at that level
>       ┌─[1] out
>       └─[1]─┬─[1]─┬─[1] A
>             │     └─[1]─┬─[1] B
>             │           └─[12]─┬─[-] C
>             │                  └─[12]─┬─[-] D
>             │                         └─[2] E
>             └─[1]─┬─[1] F
>                   └─[1]─┬─[1] G
>                         └─{+}─┬─[-] H
>                               └─{+}─┬─[-] I
>                                     └─[1] J
>
>  + non-aggregate statesets for main root character 11 on tree 1
>    character 11, region 1: optimization 2 of 2
>    '+': placeholder in other region(s) as defined at this level (1)
>    ┌─[1] out
>    └─[1]─┬─[1]─┬─[1] A
>          │     └─[1]─┬─[1] B
>          │           └─[1]─┬─[0] C
>          │                 └─[1]─┬─[0] D
>          │                       └─[1] E
>          └─[1]─┬─[1] F
>                └─[1]─┬─[1] G
>                      └─{+}─┬─[0] H
>                            └─{+}─┬─[0] I
>                                  └─[1] J
>    ____________________________________________________________
>
>
>      + non-aggregate statesets for simple subordinate character 12 on tree 1
>        local optimization
>        '+': placeholder in region(s) as defined at level 1 of the hierarchy
>             valid values are nested under the optimizations at that level
```

```
 >       ┌─[1] out
 >       └─[1]─┬─[1]─┬─[1] A
 >             │     └─[1]─┬─[1] B
 >             │           └─{+}─┬─[-] C
 >             │                 └─{+}─┬─[-] D
 >             │                       └─[2] E
 >             └─[1]─┬─[1] F
 >                   └─[1]─┬─[1] G
 >                         └─[1]─┬─[-] H
 >                               └─[1]─┬─[-] I
 >                                     └─[1] J
 >  + non-aggregate statesets for main root character 11 on tree 1
 >    character 11, region 2: optimization 1 of 1
 >    '+': placeholder in other region(s) as defined at this level (1)
 >    ┌─[1] out
 >    └─[1]─┬─[1]─┬─[1] A
 >          │     └─[1]─┬─[1] B
 >          │           └─{+}─┬─[0] C
 >          │                 └─{+}─┬─[0] D
 >          │                       └─[1] E
 >          └─[1]─┬─[1] F
 >                └─[1]─┬─[1] G
 >                      └─[1]─┬─[0] H
 >                            └─[1]─┬─[0] I
 >                                  └─[1] J
```

The aggregate statesets for this example are plotted below. By design, they do not suffer from the combinatorial explosion that follows when some characters have multiple initial regions of absence, and no special provisions are required to deal with that situation when plotting the statesets. The drawback is that the statesets themselves may be harder to intuit.

```
 >* aggregate final statesets for character hierarchy <11 12> on tree 1
 >  selected characters: all
 >
 >       ┌─[1] out
 >       └─[1]─┬─[1]─┬─[1] A
 >             │     └─[1]─┬─[1] B
 >             │           └─[12-]─┬─[-] C
 >             │                   └─[12-]─┬─[-] D
 >             │                           └─[2] E
 >             └─[1]─┬─[1] F
 >                   └─[1]─┬─[1] G
 >                         └─[1]─┬─[-] H
 >                               └─[1]─┬─[-] I
 >                                     └─[1] J
 >  + aggregate statesets for main root character 11 on tree 1
 >    ┌─[1] out
 >    └─[1]─┬─[1]─┬─[1] A
 >          │     └─[1]─┬─[1] B
 >          │           └─[01]─┬─[0] C
 >          │                  └─[01]─┬─[0] D
 >          │                         └─[1] E
 >          └─[1]─┬─[1] F
 >                └─[1]─┬─[1] G
 >                      └─[1]─┬─[0] H
 >                            └─[1]─┬─[0] I
 >                                  └─[1] J
 >
```

Example 3

This example is included to illustrate that subordinate root characters also may require some extra steps to detect and properly deal with boundaries between a region of inapplicability and its neighbouring regions of inapplicability when plotting statesets. During the search for the optimal cost, the non-aggregate statesets for a subordinate root character are uniquely determined in its subcharacters: any inner node in these subcharacters has been assigned either absence or presence. This is sufficient to find the optimal cost, but some implied gains/losses that are assigned to single branches in this way may actually occur along a series of branches near boundaries with regions of inapplicability. This is the case in the second and the fourth plot of the non-aggregate statesets for subordinate root character 12 in this example. The net effect is, just as it was the case for a non-root character as illustrated above, an ambiguous reconstruction in some inner nodes. There is the additional complexity though that, in this case, the ambiguity has to extend down into the subhierarchy that starts at that root character (the non-aggregate statesets in the subordinate characters at the nodes involved are then something like 'semi-aggregate statesets'). This is illustrated in the second and fourth plot of the non-aggregate statesets for simple subordinate character 13.

Data set:

```
  Out 0000000000 111
  A   1000000000 111
  B   1100000000 110
  C   1110000000 0--
  D   1111100000 0--
  E   1111100000 10-
  F   0000010000 111
  G   0000011000 111
  H   0000011100 0--
  I   0000011111 0--
  J   0000011111 10-

  character hierarchy <11 <12 13>>
```

Optimal tree:

```
 ┌─ out
 ├─────── A
 │     └─────── B
 │           └─────── C
 │                 └─────── D
 │                       └─ E
 └─────── F
       └─────── G
             └─────── H
                   └─────── I
                         └─ J
```

Here are the non-aggregate statesets on the optimal tree:

```
 >* non-aggregate final statesets for character hierarchy <11 <12 13>>
    on tree 1
 >  selected characters: all
 >
 >  the optimization is presented depth first
 >  at each level, the local context for the optimization of the nested level
```

```
>  is provided by the optimization of the parent level that follows
>
>        + non-aggregate statesets for simple subordinate character 13
           on tree 1
>          local optimization
>          '+': placeholder in region(s) as defined at level 1
                of the hierarchy
>               valid values are nested under the optimizations at that level
>          ┌─[1] out
>          └─[1]─┬─[1]─┬─[1] A
>                │     └─[01]─┬─[0] B
>                │            └─[-]─┬─[-] C
>                │                  └─[-]─┬─[-] D
>                │                        └─[-] E
>                └─[1]─┬─[1] F
>                      └─[1]─┬─[1] G
>                            └─{+}─┬─[-] H
>                                  └─{+}─┬─[-] I
>                                        └─[-] J
>     + non-aggregate final statesets for subordinate root character 12
        on tree 1
>       local optimization 1 of 1
>       '+': placeholder in region(s) as defined at level 1 of the hierarchy
>            valid values are nested under the optimizations at that level
>       ┌─[1] out
>       └─[1]─┬─[1]─┬─[1] A
>             │     └─[1]─┬─[1] B
>             │           └─[-]─┬─[-] C
>             │                 └─[-]─┬─[-] D
>             │                       └─[0] E
>             └─[1]─┬─[1] F
>                   └─[1]─┬─[1] G
>                         └─{+}─┬─[-] H
>                               └─{+}─┬─[-] I
>                                     └─[0] J
>
>  + non-aggregate statesets for main root character 11 on tree 1
>    there are 2 regions that are optimized independently using the
>    final optimal statesets at the borders of these regions as plotted below
>
>    such regions may have multiple non-trivial non-aggregate final statesets
>    these statesets are therefore plotted separately for each such region
>    '+' serves as a placeholder for optimizations in the other regions
>
>    the regions are defined in the first plot that follows
>    nodes that are not labelled in that definition are outside
>    such regions and have the same optimization in all following plots
>
>    ┌─ out
>    └──┬──┬─ A
>       │  └──┬─ B
>       │     └─1─┬─ C
>       │         └─1─┬─ D
>       │             └─ E
>       └──┬─ F
>          └──┬─ G
>             └─2─┬─ H
>                 └─2─┬─ I
>                     └─ J
>    character 11, region 1: optimization 1 of 2
>    '+': placeholder in other region(s) as defined at this level (1)
>    ┌─[1] out
>    └─[1]─┬─[1]─┬─[1] A
>          │     └─[1]─┬─[1] B
>          │           └─[0]─┬─[0] C
>          │                 └─[0]─┬─[0] D
>          │                       └─[1] E
>          └─[1]─┬─[1] F
>                └─[1]─┬─[1] G
>                      └─{+}─┬─[0] H
>                            └─{+}─┬─[0] I
>                                  └─[1] J
>    ____________________________________________________________
>
>
>        + non-aggregate statesets for simple subordinate character 13
           on tree 1
>          local optimization
>          '+': placeholder in region(s) as defined at level 1
                of the hierarchy
>               valid values are nested under the optimizations at that level
>          ┌─[1] out
>          └─[1]─┬─[1]─┬─[1] A
>                │     └─[01]─┬─[0] B
>                │            └─[01-]─┬─[-] C
>                │                    └─[01-]─┬─[-] D
>                │                            └─[-] E
>                └─[1]─┬─[1] F
>                      └─[1]─┬─[1] G
>                            └─{+}─┬─[-] H
>                                  └─{+}─┬─[-] I
>                                        └─[-] J
>
>     + non-aggregate final statesets for subordinate root character 12
        on tree 1
>       local optimization 1 of 1
>       '+': placeholder in region(s) as defined at level 1 of the hierarchy
>            valid values are nested under the optimizations at that level
>       ┌─[1] out
>       └─[1]─┬─[1]─┬─[1] A
>             │     └─[1]─┬─[1] B
>             │           └─[01]─┬─[-] C
>             │                  └─[01]─┬─[-] D
>             │                         └─[0] E
>             └─[1]─┬─[1] F
>                   └─[1]─┬─[1] G
>                         └─{+}─┬─[-] H
>                               └─{+}─┬─[-] I
>                                     └─[0] J
>
>  + non-aggregate statesets for main root character 11 on tree 1
>    character 11, region 1: optimization 2 of 2
>    '+': placeholder in other region(s) as defined at this level (1)
>    ┌─[1] out
>    └─[1]─┬─[1]─┬─[1] A
>          │     └─[1]─┬─[1] B
>          │           └─[1]─┬─[0] C
>          │                 └─[1]─┬─[0] D
>          │                       └─[1] E
>          └─[1]─┬─[1] F
>                └─[1]─┬─[1] G
>                      └─{+}─┬─[0] H
>                            └─{+}─┬─[0] I
>                                  └─[1] J
>    ____________________________________________________________
>
>
>        + non-aggregate statesets for simple subordinate character 13
           on tree 1
>          local optimization
>          '+': placeholder in region(s) as defined at level 1
                of the hierarchy
>               valid values are nested under the optimizations at that level
>          ┌─[1] out
>          └─[1]─┬─[1]─┬─[1] A
>                │     └─[01]─┬─[0] B
>                │            └─{+}─┬─[-] C
>                │                  └─{+}─┬─[-] D
>                │                        └─[-] E
>                └─[1]─┬─[1] F
>                      └─[1]─┬─[1] G
>                            └─[-]─┬─[-] H
>                                  └─[-]─┬─[-] I
>                                        └─[-] J
>
>     + non-aggregate final statesets for subordinate root character 12
        on tree 1
>       local optimization 1 of 1
>       '+': placeholder in region(s) as defined at level 1 of the hierarchy
>            valid values are nested under the optimizations at that level
>       ┌─[1] out
>       └─[1]─┬─[1]─┬─[1] A
>             │     └─[1]─┬─[1] B
>             │           └─{+}─┬─[-] C
>             │                 └─{+}─┬─[-] D
>             │                       └─[0] E
>             └─[1]─┬─[1] F
>                   └─[1]─┬─[1] G
>                         └─[-]─┬─[-] H
>                               └─[-]─┬─[-] I
>                                     └─[0] J
>
>  + non-aggregate statesets for main root character 11 on tree 1
>    character 11, region 2: optimization 1 of 2
>    '+': placeholder in other region(s) as defined at this level (1)
>    ┌─[1] out
>    └─[1]─┬─[1]─┬─[1] A
>          │     └─[1]─┬─[1] B
>          │           └─{+}─┬─[0] C
>          │                 └─{+}─┬─[0] D
>          │                       └─[1] E
>          └─[1]─┬─[1] F
>                └─[1]─┬─[1] G
>                      └─[0]─┬─[0] H
>                            └─[0]─┬─[0] I
>                                  └─[1] J
>    ____________________________________________________________
>
>
>        + non-aggregate statesets for simple subordinate character 13
           on tree 1
>          local optimization
>          '+': placeholder in region(s) as defined at level 1i
                of the hierarchy
>               valid values are nested under the optimizations at that level
>          ┌─[1] out
>          └─[1]─┬─[1]─┬─[1] A
>                │     └─[01]─┬─[0] B
>                │            └─{+}─┬─[-] C
>                │                  └─{+}─┬─[-] D
>                │                        └─[-] E
>                └─[1]─┬─[1] F
>                      └─[1]─┬─[1] G
>                            └─[1-]─┬─[-] H
>                                   └─[1-]─┬─[-] I
>                                          └─[-] J
>
>     + non-aggregate final statesets for subordinate root character 12
        on tree 1
>       local optimization 1 of 1
>       '+': placeholder in region(s) as defined at level 1 of the hierarchy
>            valid values are nested under the optimizations at that level
>       ┌─[1] out
>       └─[1]─┬─[1]─┬─[1] A
>             │     └─[1]─┬─[1] B
>             │           └─{+}─┬─[-] C
>             │                 └─{+}─┬─[-] D
>             │                       └─[0] E
>             └─[1]─┬─[1] F
>                   └─[1]─┬─[1] G
>                         └─[01]─┬─[-] H
>                                └─[01]─┬─[-] I
>                                       └─[0] J
>
>  + non-aggregate statesets for main root character 11 on tree 1
>    character 11, region 2: optimization 2 of 2
>    '+': placeholder in other region(s) as defined at this level (1)
```

```
 >    ┌─[1] out
 >    └─[1]─┬─[1]─┬─[1] A
 >          │     └─[1]─┬─[1] B
 >          │           └─{+}─┬─[0] C
 >          │                 └─{+}─┬─[0] D
 >          │                       └─[1] E
 >          └─[1]─┬─[1] F
 >                └─[1]─┬─[1] G
 >                      └─[1]─┬─[0] H
 >                            └─[1]─┬─[0] I
 >                                  └─[1] J
```

Here are the corresponding aggregate statesets:

```
 >* aggregate final statesets for character hierarchy <11 <12 13>> on tree 1
 >  selected characters: all
 >
 >        + aggregate statesets for simple subordinate character 13 on tree 1
 >          ┌─[1] out
 >          └─[1]─┬─[1]─┬─[1] A
 >                │     └─[01]─┬─[0] B
 >                │            └─[01-]─┬─[-] C
 >                │                    └─[01-]─┬─[-] D
 >                │                            └─[-] E
 >                └─[1]─┬─[1] F
 >                      └─[1]─┬─[1] G
 >                            └─[1-]─┬─[-] H
 >                                   └─[1-]─┬─[-] I
 >                                          └─[-] J
 >
 >     + aggregate statesets for subordinate root character 12 on tree 1
 >       ┌─[1] out
 >       └─[1]─┬─[1]─┬─[1] A
 >             │     └─[1]─┬─[1] B
 >             │           └─[01-]─┬─[-] C
 >             │                   └─[01-]─┬─[-] D
 >             │                           └─[0] E
 >             └─[1]─┬─[1] F
 >                   └─[1]─┬─[1] G
 >                         └─[01-]─┬─[-] H
 >                                 └─[01-]─┬─[-] I
 >                                         └─[0] J
 >
 >  + aggregate statesets for main root character 11 on tree 1
 >    ┌─[1] out
 >    └─[1]─┬─[1]─┬─[1] A
 >          │     └─[1]─┬─[1] B
 >          │           └─[01]─┬─[0] C
 >          │                  └─[01]─┬─[0] D
 >          │                         └─[1] E
 >          └─[1]─┬─[1] F
 >                └─[1]─┬─[1] G
 >                      └─[01]─┬─[0] H
 >                             └─[01]─┬─[0] I
 >                                    └─[1] J
```

Example 4

This example just contains an anagallis script. When executed, the output contains some more complex examples of the vizualization problems that can arise when plotting non-aggregate statesets, such as placeholders for regions that are defined at different levels.

```
characters read numeric
'example data to illustrate non-aggregate statesest in character
 hierarchies'
4 32
A   0---
B   1114
C   1111
D   1111
E   1114
F   0---
G   0---
H   0---
I   10--
J   10--
K   10--
L   10--
M   0---
N   0---
O   1112
P   1111
Q   0---
R   1114
S   1111
T   1111
U   1114
V   0---
W   0---
X   0---
Y   10--
Z   10--
1   10--
2   10--
3   0---
4   0---
5   1112
6   1111
;
characters properties set <1 <2 3 4>>;
    trees read
(1 (10 (((5 31)(28 26))((19 30)(((15 21)((((16 11)27)8)13))((7 (((25
  ((6 17)((14 ((29 9)((4 (((18 20)22)12))23)))24)))32)3))2))))))*
(1 (((26 ((4 ((((((22 (((3 (((6 5)10)((19 (20 23))(13 15))))(28 ((9
  14)((((32 18)25)11)12))))24))8)16)31)30)17))21))29)(27 (7 2))))*
(1 (20 ((8 18)(15 ((6 (26 ((((2 (22 24))16)27)7)))((32 (3 ((17 ((21
  (((10 ((5 29)28))25)(30 (11 13))))14))((12 ((31 19)4))9))))23))))))*
(((((((4 8)17)26)30)(10 ((2 12)((7 ((31 ((13 29)(((22 24)(((11 15)
  ((32 20)(((16 (14 ((23 28)19)))27)25)))9))21)))5))6))))18)3)1);

characters diagnose plot u r1 t. bk
characters diagnose plot   r1 t. bk
characters score .
```

1.4 Scope and limits of the current algorithm

At any level in a defined character hierarchy, the current algorithm properly takes into account basic logical dependences of subordination between root absence/presence characters on the one hand and their subordinate characters (simple and complex) on the other. But whenever different characters at the same level in a character hierarchy treat (sub)structures that are homologous at the parent level, constraints beyond mere hierarchic subordination have to be taken into account. This is something that anagallis does not yet do.

Consider this example, inspired by Coddington et al.'s (2018) paper on web evolution in spiders. In a group of spiders being studied, some have no web, some have a web of a type w1, and some a web of type w2. On prior grounds, web types w1 and w2 are considered to be homologous as webs (part of a transformation series) and mutually exclusive. There are also several characters that describe the web types where applicable: characters w1_1, w1_2 and w1_3 for web type w1, and character w2_1 for web type for w2. Lastly, both web types have a substructure ws that may be absent or present and that is thought to evolve independently of webtype. Characters ws_1 and ws_2 further describe it where applicable. This could be modelled in this character hierarchy (a/p for absence/presence):

```
  <web_a/p <ws_a/p ws_1 ws_2> <w1_a/p a/p w1_1 w1_2 w1_3> <w2_a/p w2_1>>.
```

This is a technically valid definition of a character hierarchy in anagallis. As such, reconstructed absence of a web (main root character) implies inapplicability of the three subordinate root characters, a hierarchic constraint that is guaranteed to be met.

But in the regions of applicability as defined by the globally best optimization of web absence/presence, subhierarchies <ws_a/p ws_1 ws_2> <w1_a/p a/p w1_1 w1_2 w1_3> <w2_a/p w2_1> will be independently optimized. That is ok for <ws_a/p ws_1 ws_2> but not for <w1_a/p a/p w1_1 w1_2 w1_3> and <w2_a/p w2_1>. The problem is that web types w1 and w2 are not independent characters but part of the transformation series of the structure 'web' that is found at the parent level of w1_a/p and w2_a/p (see Brazeau 2011 for a good discussion of similar cases in the context of classic parsimony algorithms). As a result, there is the additional constraint that all inner nodes in all regions of web presence must be optimized as either having a web of type w1 or a web of type w2. Optimizations of such inner nodes where both w1 and w2 are optimized as presence and optimizations where both w1 and w2 are optimized as absence violate the assumptions of the analysis. In addition, as absence of for example type w1 in a region of web presence automatically means presence of another type of web, such absences cannot be taken into account as independent evidence.

The current algorithm is not designed to treat such cases (see next section for a generalization to such cases). The above character hierarchy could of course be analyzed with the program, but then, for the reasons just discussed, a wrong model of the data is being analyzed and the results should therefore be interpreted with caution: whether or not presence of w1 and presence of w2 are mutually exclusive on optimal trees can be verified a posteriori, but those trees may just be optimal because absence of w1 and absence of w2 has been counted as independent evidence - which it isn't.

The additional constraints of this example could be included in the definition of character hierarchies for use in a generalized algorithm by grouping the relevant parts using, for example, square brackets (not available in the program):

```
  <web_a/p <ws_a/p ws_1 ws_2> [<w1_a/p a/p w1_1 w1_2 w1_3> <w2_a/p w2_1>]>.
```

The meaning of this notation would then be that the structures at the bracketed level (w1 and w2 types of web in this example) are part of a single transformation series and hence homologous at the parent level (w1 and w2 webs are homologous as webs, in this case). Another way to express the same information (not available either) is by allowing what can be called complex root characters: root characters that can have several different states of presence in addition to missing information and inapplicability (absence is explicitly dealt with at a higher level in the hierarchy):

```
  <web_a/p <ws_a/p ws_1 ws_2> <webtype_w1_or_w2 w1_1 w1_2 w1_3 w2_1>>.
```

In this, subhierarchy <webtype_w1_or_w2 w1_1 w1_2 w1_3 w2_1> is a subhierarchy with complex root character webtype_w1_or_w2. Its states can be '?', '-', 'w1' or 'w2'. Subordinate characters w1_1, w1_2, and w1_3 are then only applicable where the state is 'w1', character w2_1 where the state is 'w2'. This second notation results in a more concise matrices and lends itself better to algorithmically enforce the additional constraints (see next section).

Even if these two notations differ in the exact matrix that will be constructed in concrete cases, they are equivalent in terms of data and prior assumptions that they convey. This is in line with Brazeau's (2011, p. 494) observation that "matrices and their accompanying character lists should be viewed as formatted data, and not just a table of observations. That is, they should be constructed with an understanding of how that information will be interpreted by the algorithm that is receiving them".

Note that the complication in this particular case arises because there are additional characters that describe at least one web type when applicable, resulting in a series of non-independent characters at the same level of a character hierarchy with the notation that is used. If we had just observed absence/presence of webs and two different types of web, this can be correctly coded as as the following simple hierarchy that fully captures all logical dependences and that is correctly optimized by the current algorithm (it uses a single character that codes if eiher w1 or w2 has been observed):

```
  <web_a/p webtype_w1_or_w2>.
```

Similarly, if we had observed just one web type with several characters that describe it, there are no multiple characters at the same level about (sub)structures that are homologous at the parent level either (character abbreviations as in the first example):

```
  <w1_a/p <ws_a/p ws_1 ws_2> w1_1 w1_2 w1_3>.
```

So this case will also be treated correctly in the current version of the program.

Another example where such additional constraints can pop up and be useful is in the analysis of aligned sequence data. Consider this dataset of aligned sequence data:

```
  A  A--A
  B  AAGA
  C  AAGA
  D  AGAA
  E  AGAA
  F  ATCA
  G  ATCA
  H  ACTT
  I  ACTT
```

Using square brackets, the subsequence that consists of positions 2-3 can be modelled as follows:

```
  c1   subsequence absent(0) present (1)
  c2   purine at first position of subsequence absent (0) present (1)
  c3   purine A at first position of subsequence absent (0) present (1)
  c4   purine G at first position of subsequence absent (0) present (1)
  c5   pyrimidine at first position of subsequence absent (0) present (1)
  c6   pyrimidine C at first position of subsequence absent (0) present (1)
  c7   pyrimidine T at first position of subsequence absent (0) present (1)
  c8   purine at second position of subsequence absent (0) present (1)
  c9   purine A at second position of subsequence absent (0) present (1)
  c10  purine G at second position of subsequence absent (0) present (1)
  c11  pyrimidine at second position of subsequence absent (0) present (1)
  c12  pyrimidine C at second position of subsequence absent (0) present (1)
  c13  pyrimidine T at second position of subsequence absent (0) present (1)

  <c1 [<c2 [c3 c4]> <c5 [c6 c7]>] [<c8 [c9 c10]> <c11 [c12 c13]>]>

  A  0------------
  B  11100--1010--
  C  11100--1010--
  D  11010--1100--
  E  11010--1100--
  F  10--1010--110
  G  10--1010--110
  H  10--1100--101
  H  10--1100--101
```

Or equivalently, but with complex root characters c2 and c5:

```
  c1   subsequence absent(0) present (1)
  c2   first position of subsequence purine (R) or pyrimidine (Y)
  c3   first position of subsequence purine A or purine G
  c4   first position of subsequence pyrimidine C or pyrimidine T
  c5   second position of subsequence purine (R) or pyrimidine (Y)
  c6   second position of subsequence purine A or purine G
  c7   second position of subsequence pyrimidine C or pyrimidine T

  <c1 <c2 c3 c4> <c5 c6 c7>>

  A  0------
  B  1RA-RG-
  C  1RA-RG-
  D  1RG-RA-
  E  1RG-RA-
  F  1Y-TY-C
  G  1Y-TY-C
  H  1Y-CY-T
  H  1Y-CY-T
```

In both cases, the structure at the outer level of the hierarchy (a subsequence of length 2) consists of two substructures (first and second position 2 of that subsequence) that make up a sequence of two independent transformation series at the nested level. With the defined constraints properly enforced, both representations will non-redundantly extract the phylogenetic signal that is present in the indel itself and the phylogenetic signal that is present in the composition of the subsequence that is involved. In this model, the latter is, in turn, the sum of the phylogenetic signal at the purine/pyrimidine level and the phylogenetic signal at the level of fully individuated nucleotides. Similar coding can be applied at different levels of generality. Different codons that code for the same amino acid, for example, can be modelled in a similar way. So rather than to have to choose which level of phylogenetic signal to include in an analysis (see for example Simmons 2017; or, for morphological data, Torres-Montúfar 2018), this approach, once programmed, will allow to non-redundantly combine all levels into a single model.

2 Program

```
--> Subtopics
      program commands    pc   Command structure
      program treebuffer  ptr  Tree buffer
      program input       pi   Input
      program tntmode     ptn  TNT mode
```

2.1 Command structure

Commands are not case-sensitive, options are. Options are parsed from left to right.

Commands are structured in a hierarchy. Take command 'trees'. By itself, it just gives the number of trees in the tree buffer. In addition it has a number of subordinate commands to do all kinds of things with these trees. Command 'trees select', for example, can be used to manipulate trees in the tree buffer. It does nothing by itself, but requires a further subcommand. Examples are 'trees select best' to select all optimal trees, or 'trees select unique' to remove duplicate trees from the tree buffer. Likewise, 'trees search', the command to search for trees, requires a further subcommand (trees search mult or trees search swap). Command 'trees set', as a last example, can be used to set a number of properties related to trees, or to show the current values of those settings. By itself, it lists all tree-related settings. With additional subcommands, those different settings can either be set or displayed individually. 'Trees set collapse', for example sets or displays the way in which zero-length branches are treated. Or 'trees set width' sets the maximum width (in characters) that is used when trees are displayed using character graphics (trees that are wider are chopped into fitting pieces).

'help summary' (hs) gives an overview of the full command hierarchy. For multilevel commands, unambiguous abbreviations for the different levels do not require white space in between the different levels. Ambiguous abbreviations can be disambiguated by lengthening the abbreviation or by inserting white space (or in some cases even by shortening the abbreviation). 'Program quit', for example, can also be entered as 'p q' or as 'pq'. This leads to some subtleties when it comes to determining the shortest unambiguous abbreviation for a multi-level command. The numeric code for the shortest unambiguous abbreviations of, say, a two-level command is of the following form: 'i[.+]j': an integer i followed by one of the characters ' ', '.', or '+', followed by an integer j. The first integer indicates the number of characters that are required for the top-level command, the second for the subcommand. When the character in between is ' ', then a shortest unambiguous abbreviation exists that has a space in between the abbreviations of the two levels (but there might be an alternative equally short unambiguous abbreviation without a space and with an additional character from the top-level command). When the character is a '+', the shortest unambiguous abbreviation of the full command consists of the direct juxtaposition of the indicated abbreviations for the top-level command and the subcommand (without a blank in between); in addition, no ambiguities will arise if longer chunks are used. When the character is a '.', the shortest unambiguous abbreviation of the full command still consists of the direct juxtaposition of the indicated abbreviations for the top-level command and the subcommand, but ambiguities will arise for some larger chunks of the top-level command.

It can happen that there are multiple shortest abbreviations. In those cases, and when not using numeric codes, the abbreviation shown is always the one that does not contain spaces.

No blanks are required to separate commands from their options or to separate different options.

Filenames are case-sensitive or not depending on the underlying operating system (Linux only right now, so filenames are case-sensitive). Filenames are not allowed to contain blanks, irrespective of the underlying operating system. White space is used to separate a filename argument from subsequent options. Likewise, names of terminals are not allowed to contain blanks (white space is currently used to separate a terminal name from character data when reading data matrices). Names of terminals are case_sensitive.

2.2 Tree buffer

The tree buffer always reflects the current character settings (command 'characters properties set') and the current setting of collapsing mode for zero-length branches (command 'trees set zerocollapse') (using a lazy re-evaluation approach). So by changing any of these, a tree buffer with no duplicate trees may end up with duplicate trees. Or a tree buffer in which all trees have the same score may end up with trees of different scores.

2.3 Input

```
--> Subtopics
      program input native  pinn  anagallis data formats
      program input import  pii   Importing other data formats
```

2.3.1 Anagallis data formats

To read character matrices, see command 'characters read'. To set character properties (weights, additive/non-additive, ...) and to define character hierarchies, see command 'characters properties set'. To read trees, see command trees read.

2.3.2 Importing other data formats

See command 'import' and its subcommands 'import tnt' and 'import nexus' (but nexus import is not supported yet). More information about support for TNT can be found in section 'program tntmode'.

2.4 TNT mode

In addition to its native mode, anagallis has a TNT mode, a mode that partially supports a tiny subset of TNT commands (Goloboff et al. 2008). Use command 'program set tntmode' ('pst') to toggle between both modes. The command prompt indicates which mode is on: in TNT mode it has a 't'. TNT mode is also implicitly entered when importing a TNT file with command 'import tnt' ('it').

The idea of TNT mode is to be able to read files with data matrices in TNT format. Commands that are currently (partially) supported are 'ccode', 'help', 'nstates', 'xread', and 'quit' (the last one as a synonym for 'program set tntmode -' or 'pst-'. An overview of the degree to which they are supported is obtained by entering 'help*' in TNT mode.

Shortest unambiguous abbreviations are determined on the basis of this set of supported commands. So, differently as in TNT, 'q' means 'quit', not 'qcollapse'.

Commands that are not supported (or plain wrong) are skipped with appropriate warnings (they are flagged as invalid commands) but don't trigger errors. Unsupported options of partially supported commands (as available in TNT version 1.1) are flagged as unsupported options but don't trigger errors either. So in both cases anagallis keeps reading a TNT datafile rather than closing it with an error message.

Data sets, character codes, and trees in the tree buffer persist across mode changes. So you can read a matrix in regular mode, change its character settings in TNT mode, and finally go back to analyze the data regular mode. Just switch between counting from one and counting from zero when doing so.

Changes of the default datatype for TNT (TNT command 'nstates') do not persist accross different TNT files or different TNT mode sessions.

3 References

Brazeau, M., 2011. Problematic character coding methods in morphology and their effects. Biol. J. Linn. Soc. 104, 489-498.

Brazeau, M.D., Guillerme T., Smith, M.R., 2017. Morphological phylogenetic analysis with inapplicable data. BioRxiv preprint first posted online October 26, 2017. DOI: http://dx.doi.org/10.1101/209775.

Coddington, J.A., Scharff, N., 1994. Problems with zero-length branches. Cladistics 10, 415-423.

Coddington, J.A., Agnarsson, I., Hamilton, C., Bond, J.E., 2018. Spiders did not repeatedly gain, but repeatedly lost, foraging webs. PeerJ Preprints 6:e27341v1 https://doi.org/10.7287/peerj.preprints.27341v1.

De Laet, J., 2005. Parsimony and the problem of inapplicables in sequence data. Pp. 81-116 in Albert, V.A. (ed.) Parsimony, phylogeny and genomics. Oxford

University Press, ISBN 0-19-856493-7.

De Laet, J., 2015. Parsimony analysis of unaligned sequence data: maximization of homology and minimization of homoplasy, not minimization of operationally defined total cost or minimization of equally weighted transformations. Cladistics 31, 550-567.

De Laet, J., 2017. A note on Brazeau et al.'s (2017) algorithm for characters with inapplicable data, illustrated with an analysis of their Fig. 3d using anagallis, a program for parsimony analysis of character hierarchies. Technical Report, November 5, 2017. DOI: 10.13140/RG.2.2.31309.54245

Farris, J.S., 1970. Methods for computing Wagner trees. Syst. Zool. 19, 83-92.

Farris, J.S., 1983. The logical basis of phylogenetic analysis. Pp. 7-36 in Platnick, N.I, Funk, V.A. (eds.) Advances in Cladistics Vol. 2 (eds. N.I. Platnick, V.A. Funk), pp. 7-36. Columbia University Press, New York, New York.

Farris, J.S., 1989. The retention index and the rescaled consistency index. Cladistics 6, 91-100.

Farris, J.S., 2008. Parsimony and explanatory power. Cladistics 24, 825-847.

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

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

Maddison, W.P., 1993. Missing data versus missing characters in phylogenetic analysis. Syst. Biol. 42, 576-581.

Page, R.D.M, 1993. COMPONENT: Tree comparison software for Microsoft Windows, version 2.0. Natural History Museum, London.

Platnick, N.I., Griswold, C.E., Coddington, J.A., 1991. On missing entries in cladistic analysis. Cladistics 7, 337-343.

Sankoff, D., Rousseau, P., 1975. Locating the vertices of a Steiner tree in an arbitrary metric space. Mathematical Programming 9, 240-246.

Sereno, P.C., 2007. Logical basis for morphological characters in phylogenetics. Cladistics 23, 565-587.

Simmons, P., 2017. Relative benefits of amino-acid, codon, degeneracy, DNA, and purine-pyrimidine character coding for phylogenetic analyses of exons. J. Syst. Evol. 55, 85, 109.

Torres-Montúfar, A., Borsch, T. Ochoterena, H., 2018. When homoplasy is not homoplasy: dissecting trait evolution by contrasting composite and reductive coding. Syst. Biol. 67, 543–551.

REGULAR COMMANDS

Command '"'

> Echo the input until an end-of-command character.

Command double-quote is useful to document script files with information that gets echoed to the the output of the script, or to insert comments in open log files. As currently implemented, the command ends when it encounters an end-of-line or a semi-colon. So there is no way to include a semicolon in the comment itself.

For comments that do not get echoed to the output, use command '#'.

Command '#'

> Skip input until the end of the input line.

Useful to insert comments in script files or to comment out commands in script files.

To insert comments that are echoed to the output of the script, use command '"' (double quote).

Command '>'

> Provide context-sensitive help (only available from the command prompt under 'program set context =' or 'psc=').

When context-sensitive help is enabled, the current location in the command hierarchy is indicated before the command prompt.

Experimental feature, remains to be thoroughly tested.

Command '?' <commandname>

```
> Shorthand for 'help command -': use '?xyz' for basic help about command 'xyz'.
> Argument
    commandname   the command name or program option for which help is requested
                  (required)
```

For the basic documentation of command or program option 'xyz', enter '? xyz' ('?xyz'). Use 'help summary c' ('hsc') for a list of command names and their shortest abbreviations.

Command 'characters' (c)   {cd, cp, cr, csc, csh}

```
> Current number of characters and terminals; has optional subcommands.
> Subcommands
    characters diagnose    cd   show character state optimizations on tree
                                nodes; requires a subcommand
    characters properties  cp   set/show basic character properties or show
                                derived character properties; requires a
```

```
                              subcommand
    characters read        cr   read character data; requires a subcommand (use
                                command 'import' to import TNT or nexus
                                character data)
    characters score       csc  a summary of the scores of all characters and
                                character hierarchies on one or more trees
    characters show        csh  show the current dataset

--------------------------------------------------------------------------------
Command 'characters diagnose' (cd)   {cdp, cdt}

> Show character state optimizations on tree nodes; requires a subcommand.
> Subcommands
    characters diagnose plot      cdp  show tree plots of character state
                                       optimizations
    characters diagnose tabulate  cdt  show tables of character state
                                       optimizations

--------------------------------------------------------------------------------
Command 'characters diagnose plot' (cdp)  [abcCdDfgGijJkKnorstTuvW] <scopes>

> Show tree plots of character state optimizations.
> Options:
    a          label the terminals with their names (this is the default; it is
               overwritten when option 'n' is present; this option is useful to
               have terminal names in such cases as well)
    b          suppress numbering of internal nodes
    c          interpret scopes that follow as characters to include; cannot be
               combined with option 'C'
    C          interpret scopes that follow as characters to exclude; cannot be
               combined with option 'c'
    d          dry run to set custom defaults: remember all other options in
               this invocation for use with the following invocations in this
               session (with no other options, the built-in defaults are
               restored)
    D          dry run to show the current custom defaults
    f fname    write output also to file fname (append mode by default)
    i          when plotting subtrees that branch at the same level, plot small
               subtrees last, and equally sized non-leaf subtrees sorted
               according to their decreasing numeric code; this option also
               inverses the plot order of leaves that branch at the same level
    j          indent increment for each next level in a character hierarchy
               (default 3; minimum 1, maximum 20)
    J          add a blank line between each plot header and the plot itself
    g n        use alternative ASCII glyph set 1 or 2 for plotting trees (has
               only effect under 'program set unicode -' or 'psu-')
    G n        use alternative glyph set 1 for delimiting placeholders (curly
               braces instead of square brackets)
    k          condensed output (shorter branches)
    K n        truncate terminal names (to a minimum of n characters, n > 0)
               when they would exceed the specified width (option 'W') for
               plotting
    n          label the terminals with their numeric code (their sequential
               number in the data matrix; see option 'a' for more information)
    r charnum  include all characters from character hierarchy wth main root
               character charnum
    o          modifies option 'f': use overwrite mode, not append mode
    s          sort terminals that branch at same level according to their
               increasing numeric code (default: ascending alphabetical order of
               terminal names)
    t          interpret scopes that follow as trees to include; cannot be
               combined with option 'T'; this is the default interpretation of
               scopes
    T          interpret scopes that follow as trees to exclude; cannot be
               combined with option 't' or with default scopes
    u          show non-aggregate statesets of optimizations with different
               subcharacters (may result in multiple trees per character); only
               (possibly) affects characters in character hierarchies
    v          verbose (only has effect with character hierarchies)
    W n        maximum width (in characters) of a single line (beyond this, the
               tree is broken into subtrees); use -1 for the width of the
               current window (default), 0 to turn off this feature (valid
               values 20 - 5000)
> Argument
    scopes   one or more scopes (character scopes by default, no default scope)
```

Shows the final state sets of one or more characters on one or more trees. There is no default for the character(s) to show. The default tree is the current tree.

Scopes and options may be intermingled. By default, scopes are interpreted as scopes of characters to include. This default interpretation can be changed with option 'C' (interpret scopes that follow as characters to exclude), option 't' (interpret scopes that follow as trees to include) and option 'T' (interpret scopes that follow as trees to exclude). Options 'c' and 'C' cannot be combined. Neither can 't' and 'T'.

Polytomies are not optimized as polytomies. Optimizations shown are the ones for the underlying dichotomous resolutions that the program happened to find first.

When a character that is part of a character hierarchy is specified as a character to plot, all characters that lead from the main root character of the hierarchy to that character are automatically included as well. As an example, consider character hierarchy <1 5 <7 8 9 <10 11 15>>>. Requesting an optimization plot for character 15 will automatically trigger plots for characters 1, 7, and 10 as well.

Character optimizations are not plotted in the order in which the characters are entered in this command, but, with the exception of subordinate characters in character hierarchies, in numeric character order. Subordinate characters in a character hierarchy are plotted as part of that hierarchy under the main root character of that hierarchy. With option 'v', a reference to that main root character is added at the numeric position of each character of the hierarchy for which a plot was requested.

To plot optimizations for a full character hierarchy, all characters of that hierarchy must be specified in the invocation of this command. Alternatively, option 'r charnum' can be used (r for recursive), with charnum the number of the main root character of that hierarchy. With this option, all characters of that hierarchy are automatically included. It does not change the interpretation of scopes as set with options 'c', 'C', 't' and 'T.

For each character that is part of a character hierarchy, the default statesets that are shown are the aggregate statesets. Non-aggregate statesets can be plotted using option 'u'.

```
--------------------------------------------------------------------------------
Command 'characters diagnose tabulate' (cdt)  [cCtT] <scope(s)>
```

```
> Show tables of character state optimizations.
> Options:
    c  interpret scopes that follow as characters to include; cannot be combined
       with option 'C'
    C  interpret scopes that follow as characters to exclude; cannot be combined
       with option 'c'
    t  interpret scopes that follow as trees to include; cannot be combined with
       option 'T'; this is the default interpretation of scopes
    T  interpret scopes that follow as trees to exclude; cannot be combined with
       option 't' or with default scopes
> Argument
    scopes   one or more scopes (character scopes by default, no default
             character scope)
```

Arguments and options can be intermingled.

Tabulate the final statesets of reconstructed inner nodes for one or more characters on the current tree. Use option 't' or option 'T' to select other trees.

Numbers of inner nodes correspond to the numbers as obtained with command 'trees show plot'

Polytomies are not optimized (yet). Optimizations shown are the ones for the underlying dichotomous resolutions that the program happened to find first.

For characters in a character hierarchy, the statesets shown are the aggregate statesets.

```
--------------------------------------------------------------------------------
Command 'characters properties' (cp)   {cpm, cps}

> Set/show basic character properties or show derived character properties;
  requires a subcommand.
> Subcommands
    characters properties minmax  cpm  list minimum and maximum number of steps
                                       for all characters outside character
                                       hierarchies
    characters properties set     cps  set/show basic character properties
                                       (defaults: prior weight 1, active,
                                       non-additive, not part of a character
                                       hierarchy)

--------------------------------------------------------------------------------
Command 'characters properties minmax' (cpm)

> List minimum and maximum number of steps for all characters outside character
  hierarchies.
```

For a character outside a character hierarchy, the maximum number of steps, max (its number of steps on a star-tree; g of Farris 1989), is calculated directly and exactly. The minimum number of steps for such a character, min (m of Farris 1989), is calculated using a single tree build using only those terminals that have a different state set. This gives correct results for non-additive characters or when no polymorphic terminals are present. It remains to be determined if this always works for additive polymorphic terminals.

These numbers can be considered derived or secondary character properties: they depend on the basic character properties but also on the set of terminals being considered.

For characters in character hierarchies, no such numbers are provided. For any character hierarchy of interest, they can be obtained heuristically as follows.

For the minimum numbers, first inactivate all characters except those of that character hierarchy. Next do a tree search under 'optimality set searchmode =' ('oss='). The score obtained provides the minimal score for the character hierarchy as a whole. The distribution(s) of this score over the individual characters of the hierarchy provide(s) the (range of possible) minimum numbers for these characters.

Likewise, for the maximum numbers, first inactivate all characters except those of that character hierarchy. Next do a tree search under 'optimality set searchmode -' ('oss-'). The score obtained provides the maximum score for the character hierarchy as a whole. The distribution(s) of this score over the individual characters of the hierarchy provide(s) the (range of possible) maximum numbers for these characters.

Command not active in anagallis v1.02a.

```
--------------------------------------------------------------------------------
Command 'characters properties set' (cps)  [/+-[]<>acefnopqv] <character scopes>

> Set/show basic character properties (defaults: prior weight 1, active,
  non-additive, not part of a character hierarchy).
> Options:
    /n       set character weight to integer weight n for the character scopes
             that follow
    +        set the character additive for the character scopes that follow
             (see -)
    -        set the character non-additive for the character scopes that follow
             (see +)
    [        activate the character scopes that follow (see ])
    ]        inactivate the character scopes that follow (see [)
    <>       define a character hierarchy (see ><)
    ><       undefine a character hierarchy (see <>)
    a        show the activity stiatus (active/inactive) of all characters
    c        show the character hierarchies that are currently defined
    e str    str must be 'tnt'; write the character codes as a readable
             statement for TNT (as far as they exist there); this implies a
             renumbering of characters to start counting from zero
    f fname  write output also to file fname (append mode by default)
    n        show the non-additive/additive status of all characters
    o        modify option 'f': use overwrite mode, not append mode
    p        show the character weights in use
    q        don't show any of the current settings (takes precedence over other
             output modifiers)
    v        increase verbosity (up to two levels)
> Argument
    scopes   one or more character scope (no default; scopes can be intermingled
             with options)
```

This is a multiline command, so it always needs a semicolon (';') to terminate. Numbering of characters starts from 1 (but see option 'e'). There are two groups of options:

1. character modifiers such as '+' or '[' that modify the interpretation of characters in the scopes that follow

2. options such as 'a', 'e', or 'n' that set/modify the output of the command.

By default characters are active, non-additive, have weight one, and no character hierarchies are defined. Character activity, additivity and weight can be changed with the options '[', ']', '+', '-', and '/'.

Character scopes and options may be intermingled, and the modifiers that are active are applied to the scopes as they are read. As with other commands, such changes get only activated after the whole command finishes without an error (as triggered; for example, by a non-existing option). An appropriate warning is shown when nothing gets changed because of such errors.

By default, all current settings (character additivities, weights and activities, and defined character hierarchies) are displayed in a program readable form after each run of the command. Option 'q' turns this off. By default, character additivities, weights, and activities are reported in one statement and in the most condensed form . This implies that the character scopes that are used will not necessarily show all characters in order. With one occurrence of option 'v', characters are reported in order but grouped in scopes as much as possible. With two occurrences of option 'v', characters are reported individually. When character hierarchies have been defined, these are always reported separately from activity, additivity and weight (see below for the meaning of option 'v' in that case).

Options 'a', 'c', 'n', and 'p' can be used to select what will be shown in the report as described above. From the moment one of these four is specified, only the character properties so selected will be shown. When none of these is specified, the command behaves as if all four are specified.

Outside character hierarchies, the dash character ('-') is treated (and shown) as missing information ('?'). The possibility to toggle this to treating it as a separate state is not available. The interpretation of a dash in a character hierarchy is explained in detail below.

Compared to the ccode statement of programs as Hennig86, Nona or TNT, an important extension is the possibility to define and undefine character hierarchies.

To define or specify a character hierarchy, '<' and '>' can be used to set off levels of (in)applicability. Within each level, the first character that is specified is the absence/presence character that determines inapplicability further on. Because it is at the root of a character (sub)hierarchy it is called a root character. The root character of the complete hierarchy is called the main or global root character. Within each level, the characters that follow the root character are called subordinate characters at that level. These include root characters of one level down: a root character of a nested level is a subordinate character one level up. These are called complex subordinate characters (as opposed to simple subordinate characters).

As an example,

```
    <1 5 6 <7 8>>
```

is a character hierarchy with main root character 1. This character codes absence/presence of some feature. Characters 5, 6, and 7 are subordinate characters at that level: they code three aspects of that feature where it is present. Characters 5 and 6 are simple subordinate characters, character 7 is a complex subordinate character: it is at the root of its own subhierarchy (it codes absence/presence of a subfeature). Character 8, finally, is subordinate at that subordinate level and describes some feature of that subfeature.

Root characters can have at most four different state codes:
* a first one denoting absence
* a second one denoting presence
* a third one for inapplicability
* '?' for missing data.

Polymorphisms are not allowed in root characters. The main root character of a hierarchy, in addition, is not allowed to have inapplicable data. If this would be the case in an existing dataset, that datset must first manually be augmented with one or more additional outer levels of applicability. The outer additional level can then be used as the main root of the hierarchy.

By default, state code 0 is taken to indicate absence and state code - is taken to indicate inapplicability. Presence is coded by whatever third state code used (apart from '?' (but there are some constraints; see below). To let a different state code denote absence, specify this after the character using a '!' followed by that other state. To let a different state code denote inapplicability, specify this after the character using a ':' followed by that other state. So in root character 7 of hierarchy

```
    <1 5 6 <7 !1:0 8>>
```

state code 1 codes absence and state code 0 inapplicability; in character 1 it is still default state code 0 that codes absence.

In simple subordinate characters in a character hierarchy (characters 5, 6 and 8 in the example), state code '-' is by default taken to indicate inapplicability. To let a different state code denote inapplicability, specify this after the character using a ':' followed by that other state. So in

```
    <1 5 :2 6 <7 8 :0>>
```

state code 2 denotes inapplicability in character 5 and state code 0 denotes inapplicability in character 8; in character 6 it is still default state code - that codes inapplicability.

Polymorphisms that involve inapplicability are not allowed.

In verbose mode (option 'v'), all root characters in the hierarchy will be shown with their ':' and '!' modifiers; all simple subordinate characters with their ':'modifier. In non-verbose mode, these are only indicated where their values differ from the defaults.

Within the specification of a character hierarchy, the regular code modifiers and the options that modify output are not allowed, but the modifiers that are active are applied while reading the specification.

Root character that don't have subordinate characters are allowed (as long as their observed states can be interpreted as an absence/presence character), but such trivial hierarchies don't serve any meaningful purpose. From a practical point of view, they will slow down tree searches. Also note that the unweighted score of such a a trivial character hierarchy will be one higher than for that same character outside that trivial character hierarchy. This is so because subcharacters are only taken into account inside character hierarchies.

Character scopes that span multiple characters are not allowed at the start of a (sub)hierarchy: root characters must be explicitly specified as a single character. Character scopes that span multiple characters are allowed for simple subordinate characters. When such a scope is followed by ':' and a state code, that state code is taken to denote inapplicability in all characters of the scope.

A character hierarchy imposes a number of logical constraints on the character state distributions:

1. When a given terminal has missing data ('?') in a root character, it must have missing data in all subordinate characters as well. (If it is not known if a feature is present or not in a terminal, aspects or subfeatures are not known either).
2. When a given terminal has absence in a root character, all subordinate characters must have inapplicability for that terminal.
3. When a given terminal has inapplicability in a root character, all subordinate characters must have inapplicability as well.
4. When a given terminal has presence in a root character, none of the subordinate characters at that level can have inapplicability (but see below).

When a hierachy specification is submitted, it will only succeed when these constraints are met. Failure for one of those reasons means that there is an internal contradiction between the requested hierarchy and the state distribution of the characters involved. If that's the case, the program will point out which of these constraints is not met for which terminal and which character.

The idea behind this way of specifying character hierarchies is to provide a flexible way to adapt the specifications to existing datasets without having to edit these datasets (too much) rather than the other way around. For the same reason, even if the default values are different, it is not enforced that the state code for inapplicability (':' modifier) and the state code for absence ('!' modifier) in root characters must be different (when a hierarchy is properly defined, the program has sufficient information to figure out the correct interpretation). However, when they are the same, the fourth constraint is skipped in subordinate root characters because it can no longer be unequivocally tested

To undefine a character hierarchy, enclose its main root character between '>' and '<'. So

```
'>1<'
```

undefines the hierarchy of the example.

A character hierarchy can be (in)activated as a whole by (in)activating its main root character. It is currently not allowed to (in)activate part of a hierarchy. As a work-around, the hierarchy can be undefined and then redefined with the unwanted characters excluded. These can then be inactivated outside the hierarchy.

Alternatively, the prior weights of the characters to be excluded can be set to 0. This does not require redefinition of the hierarchy but will slow down tree searches: inactive characters are not optimized during tree searches but characters that are part of a hierarchy and have weight zero are (outside hierarchies, zero-weight characters are not optimized during tree searches). Be aware that this can have unexpected side effects. If, for example, one would give zero weight to tail absence/presence with non-zero weight for tail color, one can end up with an awful lot of tail gains and losses (that each have weight zero).

To avoid confusion or ambiguity, some additional constraints are imposed when setting character hierarchies:
state code '-' is not allowed as a regular state code;
'-' for absence in a character (!) is not allowed;
'?' is not accepted as code for inapplicability or as code for absence.

The restriction to allow '-' only to mean inapplicability in character hierarchies makes it possible to use it unequivocally as a character that separates subcharacters when showing optimizations of character hierarchies on trees with commands ('characters diagnose plot' and 'characters diagnose tabulate'). Whatever state code has been used for inapplicability when defining a hierarchy (:), these two commands will use '-' to indicate inapplicability in their output. This makes it easier to interpret the output of these commands, especially in root characters where inapplicability and absence have been coded with the same state code.

Characters that are inactive, whether part of a hierarchy or not, are not optimized during tree searches. Afterwards their optimizations are available for commands such as 'characters diagnose plot' though. The same is true for characters that have weight 0 and are not part of a character group. Neither are taken into account when collapsing zero-length branches.

Command 'characters read' (cr)   {cra, crn}

```
> Read character data; requires a subcommand (use command 'import' to import TNT
  or nexus character data).
> Subcommands
    characters read alphanumeric  cra  read character data with up to 30 regular
                                       character states coded as 0-9 and a-t (or
                                       A-T)
    characters read numeric       crn  read character data with up to ten
                                       regular character states coded as 0-9
```

Support for interleaved data sets and for concatenating multiple data sets is not available yet: all character data for an analysis have to be in a single non-interleaved dataset.

After a first data set has been read, it is possible to read a new data set, but it will replace the original one.

Command 'characters read alphanumeric' (cra)

```
> Read character data with up to 30 regular character states coded as 0-9 and
  a-t (or A-T).
```

Works mostly as NONA's xread command. Digits 0-9 and letters a-t (not case sensitive) are available for a total of up to 30 different states. In addition, '?' is available for missing data and '-' for inapplicable data.

A dash is by default interpreted and shown as missing data ('?'). It only acquires its special meaning of inapplicable data in character hierarchies (see command 'characters properties set'

As in Nona, there can be an optional comment between the command name and the number of characters between the command name, enclosed by single quotes. The parser of anagallis does not recognize escaped single quotes as such, so make sure the comment itself does not contain single quotes.

When the data have been read, characters are by default treated as not being part of character hierarchies. Character hierarchies can be defined on top of these data using command 'characters properties set'.

Command 'characters read numeric' (crn)

> Read character data with up to ten regular character states coded as 0-9.

Works mostly as NONA's xread command. Digits 0-9 are available for a total of up to 10 different states. In addition, '?' is available for missing data and '-' for inapplicable data.

A dash is by default interpreted and shown as missing data ('?'). It only acquires its special meaning of inapplicable data in character hierarchies (see command 'characters properties set').

As in Nona, there can be an optional comment between the command name and the number of characters between the command name, enclosed by single quotes. The parser of anagallis does not recognize escaped single quotes as such, so make sure the comment itself does not contain single quotes.

When the data have been read, characters are by default treated as not being part of character hierarchies. Character hierarchies can be defined on top of these data using command 'characters properties set'.

> Examples

```
     #read the data
     characters read numeric
     '
     Tail example of Maddison (1991).
     Characters 13 and 14 are tail a/p and tail color.
     Maximizing homology, there is no long distance effect of tail color between
     the non-homologous tails of A-D and K-N and two trees are obtained.
     '
     14 15
     out 000000000000 0-
     A   111100000011 11
     B   111100000001 11
     C   111100000010 12
     D   111100000011 12
     E   111000000000 0-
     F   110000000000 0-
     G   100000000000 0-
     H   000010000000 0-
     I   000011000000 0-
     J   000011100000 0-
     K   000011110000 12
     L   000011111000 12
     M   000011111100 11
     N   000011111100 11
     ;
     #define the character hierarchy
     characters properties set <13 14>;
     #do a tree search
     trees search mult *5;
     #give the number of trees found
     trees
     #plot all trees
     trees show plot;
```

```
--------------------------------------------------------------------------------
Command 'characters score' (csc)  [tT] <treescopes>

> A summary of the scores of all characters and character hierarchies on one or
  more trees.
> Options:
    t  interpret scopes that follow as trees to include; cannot be combined with
       option 'T'; this is the default interpretation of scopes
    T  interpret scopes that follow as trees to exclude; cannot be combined with
       option 't' or with default scopes
> Argument
    treescopes   trees to list the character scores of (defaults to the current
                 tree)

List the character scores on the trees in the specified tree scopes (default:
current tree). For characters with non-unity prior weight, the score is reported
as prior weight times basic score. The scores of inactive characters are
included in the output of this command (in the overview of all characters they
are put between square brackets).

For a character hierarchy, the total score of the hierarchy is listed for the
main absence/presence character of that hierarchy. For the other characters of
that hierarchy, a placeholder reference to their immediate parent
absence/presence character is provided. This is done because, in general,
character hierarchies have no unique distribution of their total score over
their constituent characters. So in a brief summary as provided here, it only
makes sense to give the score of the full hierarchy.

More detailed information about the score of character hierarchies can be
obtained with command 'characters diagnose plot' ('cdp').

--------------------------------------------------------------------------------
Command 'characters show' (csh)  [abefinoru]

> Show the current dataset.
> Options:
    a        show active characters only
    b n      set block size to n (insert a space every nth character; default is
             10)
    e str    str must be 'tnt' or 'fasta': show the data in TNT or fasta format
    f fname  write output also to file fname (append mode by default)
    i        show informative characters only
    n        no characters, just terminal names
    o        modify option 'f': use overwrite mode, not append mode
    r        show data as originally read (discard changes such as making a
             character state set continuous for an additive character in a
             polymorphic terminal)
    u        use upper case for alphabetical state codes (default: use lower
             case)

--------------------------------------------------------------------------------
Command 'help' (h)   {hc, hd, hs, ht}
```

```
> Show basic usage information and program options; has optional subcommands to
  get more detailed information.
> Subcommands
    help command  hc  show information about a specific regular command
    help dump     hd  show all built-in program documentation
    help summary  hs  overview of available commands and/or help topics
    help topic    ht  get information about a specific help topic

--------------------------------------------------------------------------------
Command 'help command' (hc)  [forsw] <- commandname>

> Show information about a specific regular command.
> Options:
    f fname    write output also to file fname (append mode by default); implies
               'w80'
    o          modify option 'f': use overwrite mode, not append mode
    r          include documentation of subcommands (no effect when 's' is
               specified as well)
    s          summary (only short description, completions, and list of options
               and arguments, no long description and no examples)
    w numcols  set line width (in characters); beyond this, lines are wrapped;
               defaults to the width of the current window
> Argument
    - commandname   a dash followed by the command name or program option for
                    which help is requested (required)

For the documentation of command or program option 'xyz', enter 'help command -
xyz' ('hc-xyz'). The dash before the command name is required to disambiguate
cases where a command starts with a leter that is also an option of this help
command. When using this command without options, command '? xyz' can be used as
a shorthand for 'help command - xyz'.

By default, the command name can be abbreviated (check command 'program set
shortcommands' to change this). Use 'help summary c' ('hsc') for a list of
command names and their shortest abbreviations.

Values for option 'w' must be in the range 66..10000. When using a w-value
higher than the width of the current window, output will not wrap correctly in
the window. With option 'f', wrapping is by default done at 80 characters (and
not according to the current window). This default can be changed with option
'w'.

--------------------------------------------------------------------------------
Command 'help dump' (hd)  [cfilostTuw]

> Show all built-in program documentation.
> Options:
    c          dump documentation of regular commands ('c', 'i', 't', 'T', and
               'u' are additive)
    f fname    write output also to file fname (append mode by default); implies
               'w80'
    i          dump an index of all commands and topics ('c', 'i', 't', 'T', and
               u are additive) (as with command 'help summary')
    l          use multilevel lists of command and topic completions
    o          modify option 'f': use overwrite mode, not append mode
    s          summary (show only show short description, completions, and list
               of options and arguments for the commands)
    t          dump topics ('c', 'i', 't', 'T', and 'u' are additive)
    T          dump documentation of TNT mode commands ('c', 'i', 't', 'T', and
               'u' are additive)
    u          dump program usage and options ('c', 'i', 't', 'T', and 'u' are
               additive)
    w numcols  set line width (in characters); beyond this, lines are wrapped;
               defaults to the width of the current window
```

By default, all built-in documentation is dumped. When at least one of options 'c', 'i', 't', 'T' is present, output follows absence/presence of these three options.

Values for the option 'w' must be in the range 66..10000. When using a w-value higher than the width of the current window, output will not wrap correctly in the window. With option 'f', wrapping is by default done at 80 characters (and not according to the current window). This default can be changed with option 'w'.

Command 'help summary' (hs) [bcCfilnostTvw]

```
> Overview of available commands and/or help topics.
> Options:
    b          brief: do not include the short descriptions
    c          show regular commands ('c', 't' and 'T' are additive)
    C c        use character 'c' as field delimiter (useful for creating csv
               output)
    f fname    write output also to file fname (append mode by default); implies
               'w80'
    i num      number of characters to indent for each next level in the command
               or topic hierarchy (2 by default, 100 at most; at least 0 with
               option 'v', at least 1 otherwise; numbers outside this range are
               set to the appropriate extreme)
    l num      number of levels to show (0 stands for all levels; this is the
               default when the option is not present)
    n          use numeric codes for shortest unambiguous abbreviations
    o          modify option 'f': use overwrite mode, not append mode
    s num      change the space between output fields (num is 2 by default, 1 at
               least, and 100 at most; numbers outside this range are set to the
               appropriate extreme)
    t          show topics ('c', 't' and 'T' are additive)
    T          show TNT mode commands ('c', 't', and 'T' are additive)
    v          verbose (spell out full command name at each level)
    w numcols  set line width (in characters); beyond this, lines are wrapped;
               defaults to the width of the current window
```

As long as options 'c', 't', and 'T' are all three absent, 'c' and 't' are the defaults.

Values for the option 'w' must be in the range 66..10000. When using a w-value higher than the width of the current window, output will not wrap correctly in the window. With option 'f', wrapping is by default done at 80 characters (and not according to the current window). This default can be changed with option 'w'.

Command 'help topic' (ht) [forw] <- topicname>

```
> Get information about a specific help topic.
> Options:
    f fname    write output also to file fname (append mode by default); implies
               'w80'
    o          modify option 'f': use overwrite mode, not append mode
    r          include subtopics
    w numcols  set line width (in characters); beyond this, lines are wrapped;
               defaults to the width of the current window
> Argument
    - topicname   a dash followed by the topic name for which help is requested
                  (required)
```

For the documentation of topic 'xyz', enter 'help topic - xyz' ('ht-xyz'). The dash before the topic name is required to disambiguate cases where a topic starts with a leter that is also an option of this help command.

By default, the topic name can be abbreviated (check command 'program set shortcommands' to change this). Use 'help summary t' ('hst') for a list of topics and their shortest abbreviations.

Values for option 'w' must be in the range 66..10000. When using a w-value higher than the width of the current window, output will not wrap correctly in the window. With option 'f', wrapping is by default done at 80 characters (and not according to the current window). This default can be changed with option 'w'.

Command 'import' (i)   {in, it}

```
> Import data from other file formats; requires a subcommand.
> Subcommands
    import nexus  in  execute a nexus datafile (supported nexus subset still
                      empty in this version)
    import tnt    it  execute a TNT datafile (limited support for a tiny subset
                      of TNT commands)
```

Command 'import nexus' (in) <nexusfilename>

```
> Execute a nexus datafile (supported nexus subset still empty in this version).
> Argument
    nexusfilename   name of the nexus file (required)
```

Under construction.

Command 'import tnt' (it) <tntfilename>

```
> Execute a TNT datafile (limited support for a tiny subset of TNT commands).
> Argument
    tntfilename   name of the file that contains TNT commands (required)
```

Use command 'help summary T' ('hsT') to get an overview an overview of supported TNT commands. These are only available in TNT mode (command 'program set tntmode =' or 'pst=') and in TNT files that are imported with this command.

More information about these supported TNT commands is available with command 'help' within TNT mode.

Command 'log' (l)   {lp, lr, lsta, lsto}

```
> Name and status of log file, if there is one; has optional subcommands.
> Subcommands
    log pause   lp    suspend output to the logfile
    log resume  lr    resume output to the logfile
    log start   lsta  open a log file (append mode by default)
    log stop    lsto  close the current logfile

--------------------------------------------------------------------------------
Command 'log pause' (lp)

> Suspend output to the logfile.

--------------------------------------------------------------------------------
Command 'log resume' (lr)

> Resume output to the logfile.

--------------------------------------------------------------------------------
Command 'log start' (lsta)  [nof]

> Open a log file (append mode by default).
> Options:
    n              suppress logging of a header with time and date
    f logfilename  name of the logfile to open (required, append mode by
                   default)
    o              open the file in overwrite mode

When a logfile is open, all screen output is also written to the logfile.

This is a general logging file that logs output of all commands until it is
paused or closed. In addition, several commands have options to specify logging
of their output to a file. Such command-specific logging happens in addition to
this general logging.

--------------------------------------------------------------------------------
Command 'log stop' (lsto)

> Close the current logfile.

--------------------------------------------------------------------------------
Command 'optimality' (o)   {os}

> Set/show optimality criterion; requires a subcommand.
> Subcommands
    optimality set  os  overview of settings that relate to the optimality
                        criterion; has optional subcommands

--------------------------------------------------------------------------------
Command 'optimality set' (os)   {osd, oss}

> Overview of settings that relate to the optimality criterion; has optional
  subcommands.
> Subcommands
    optimality set deviation   osd  set/show allowed deviation from optimality
                                    in tree searches and tree buffer cleaning
    optimality set searchmode  oss  set/show search mode: look for best (=,
                                    default) or worst (-) trees
```

```
--------------------------------------------------------------------------------
Command 'optimality set deviation' (osd)  <n>

> Set/show allowed deviation from optimality in tree searches and tree buffer
  cleaning.
> Argument
    n   a non-negative integer
```

This value is used during swapping (command 'trees search') and when selecting optimal trees (command 'trees select best'). It has only limited influence while building initial trees for swapping.

This has consequences that at first may seem counterintuitive. Assume a strongly structured dataset for which 20 replicates of tree building and swapping (using command 'trees search mult') all return the same tree of score 100. Next the deviation from optimality is set to 50 and 20 new replicates also just return that same tree. This does not necessarily mean that there are no trees of lengths 101-150. What happens most likely is that the build itself already finds that tree in all replicates. And because it is already in the tree buffer, it is not passed on to swapping in the replicates.

The best thing to do then after increasing the deviation and before initiating new search replicates, is to explicitly swap the trees in the tree buffer (command 'trees search swap').

When the deviation is decreased, the tree buffer is left as it is. An explicit invocation of command 'trees select best' is required to make it reflect the new situation.

When looking for worst trees (see command 'optimality set searchmode'), the requested level of suboptimality is reported as a negative integer.

```
--------------------------------------------------------------------------------
Command 'optimality set searchmode' (oss)  [-=]

> Set/show search mode: look for best (=, default) or worst (-) trees.
> Options:
    =  search best trees
    -  search worst trees
```

Looking for worst trees may be useful to get an indication of the spread of possible tree scores

```
--------------------------------------------------------------------------------
Command 'program' (p)   {pq, ps}

> Quit the program or set/show general settings; requires a subcommand.
> Subcommands
    program quit  pq  quit the program
    program set   ps  overview of general settings; has optional subcommands

--------------------------------------------------------------------------------
Command 'program quit' (pq)

> Quit the program.
```

Close all open files and quit the program.

Command 'program set' (ps)   {psc, psl, psr, pss, pst, psu}

> Overview of general settings; has optional subcommands.
> Subcommands

program set context	psc	set/show if context-sensitive help (command '>') is available from the command line (=) or not (-, default)
program set longlists	psl	set/show if lists of command completions are multilevel (=) or not (-, default)
program set randomseed	psr	set/show seed for generator of pseudorandom numbers
program set shortcommands	pss	set/show if command abbreviations are allowed (=, default) or not (-)
program set tntmode	pst	set/show if TNT mode is on
program set unicode	psu	set/show if tree plotting uses multibyte UTF-8 characters (=, default) or not (-)

Command 'program set context' (psc) [-=]

> Set/show if context-sensitive help (command '>') is available from the command line (=) or not (-, default).
> Options:

```
    -  turn of availability of context-sensitive help command '>'
    =  turn off availability of context-sensitive help command '>'
```

When context-sensitive help is on, commands that have no possible subcommands, options, and arguments are still terminated by hitting the 'enter' key. Likewise for commands that only have '-' and '=' as options and for which at least one option has been provided. In all other cases, commands have to be terminated explicitly with a semicolon.

Still an experimental feature. It seems to be working quite well but needs to be thoroughly tested.

When context-sensitive help is off, a semicolon is only required for the few commands that possibly take a long (structured) argument list.

Command 'program set longlists' (psl)

> Set/show if lists of command completions are multilevel (=) or not (-, default).

Affects the list of command completions that is shown when a command is specified as an ambiguous abbreviation, with a missing required subcommand, or with an invalid subcommand. For the completion lists that are used in the help commands, check the options for those commands.

Command 'program set randomseed' (psr) <n>

> Set/show seed for generator of pseudorandom numbers.
> Argument

```
    n   a strictly positive integer
```

Without an argument, the current seed for the pseudorandom number generator is shown (initially 1 by default). With a strictly positive argument n, the current seed is set to n.

The program uses pseudorandom numbers on such various occasions as generating a pseudorandom addition sequence to add terminals to a growing tree or to resample datasets (the latter is not available in this version).The generator used is the same as in Component 2.0 (Page 1993). It calculates the next pseudorandom number from the current one using this recursion:

```
    X(n + 1) = a * X(n) mod p
```

with

```
    p = 2 power 31 -1 (a Mersenne prime)
    a = 7 power 5
```

At each iteration, the current number is the seed for the next one.

By setting the seed to a specific value at the start of the program, it is guaranteed that, say, a tree search that is performed after starting the program will by default return the same trees every time. This default behavior ensures reproducibility of default searches after program startup but it is not suited for parallelization of replicates for a given data set by starting a number of parallel anagallis batch sessions, either manually or script-driven. In such cases, each parallel invocation has to set its own random seed.

Command 'program set shortcommands' (pss) [-=]

```
> Set/show if command abbreviations are allowed (=, default) or not (-).
> Options:
    =  abbreviations allowed (default)
    -  abbreviations not allowed
```

This setting only applies in regular program mode when context-sensitive help is off (command 'program set context'). It does not apply in TNT mode.

Command 'program set tntmode' (pst) [=-]

```
> Set/show if TNT mode is on.
> Options:
    =  switch to TNT mode
    -  switch to regular mode
```

Command 'program set unicode' (psu) [=-]

```
> Set/show if tree plotting uses multibyte UTF-8 characters (=, default) or not
  (-).
> Options:
    =  enable unicode characters when plotting trees (default)
    -  disable unicode characters when plotting trees (just use ASCII
       characters)
```

When plotting trees, the program by default uses some UTF-8 encoded unicode characters that are longer than one byte.

Most terminals properly deal with such characters, but when exporting the output to a word processor make sure to use a non-proportional font that properly deals with UTF-8 encoded unicode characters. In openoffice 3.2, for example, font

```
'Courier 10 Pitch' is problematic but 'FreeMono' is ok.

This command can be used to disable the use of UTF-8 encoded unicode characters
when plotting trees.

Note that input is always assumed to be 100% ASCII. This command does not change
that.

--------------------------------------------------------------------------------
Command 'script' (s)   {se, sr}

> Overview of open script files; has optional subcommands.
> Subcommands
    script execute  se  overview of script files that are opened for execution;
                        has optional subcommands
    script record   sr  name and status of the file that is open for recording
                        commands, if there is one; has optional subcommands

> Examples
     Script files can be opened for execution or for recording commands that are
entered from the command prompt.

--------------------------------------------------------------------------------
Command 'script execute' (se)   {sep, ser, sesta, sesto}

> Overview of script files that are opened for execution; has optional
  subcommands.
> Subcommands
    script execute pause   sep    temporarily suspend execution of current
                                  script file and get interactive input
    script execute resume  ser    resume reading from the current script file
                                  that is open for execution
    script execute start   sesta  open a script file and start executing its
                                  commands
    script execute stop    sesto  close the current script file that is open for
                                  execution

--------------------------------------------------------------------------------
Command 'script execute pause' (sep)

> Temporarily suspend execution of current script file and get interactive
  input.

--------------------------------------------------------------------------------
Command 'script execute resume' (ser)

> Resume reading from the current script file that is open for execution.

--------------------------------------------------------------------------------
Command 'script execute start' (sesta)  [f]

> Open a script file and start executing its commands.
> Options:
    f fname  name of the file that contains anagallis commands to execute
             (required)

Script files for execution can be nested up to 16 levels deep.

--------------------------------------------------------------------------------
```

```
Command 'script execute stop' (sesto)

> Close the current script file that is open for execution.

--------------------------------------------------------------------------------
Command 'script record' (sr)   {srp, srr, srsta, srsto}

> Name and status of the file that is open for recording commands, if there is
  one; has optional subcommands.
> Subcommands
    script record pause   srp    temporarily suspend writing to the current file
                                 for recording commands
    script record resume  srr    resume recording commands to the script file
                                 for recording
    script record start   srsta  open a file for recording commands (append mode
                                 by default)
    script record stop    srsto  close the current file for recording commands

When there is an open unsuspended script file for recording commands to, all
commands that are entered from the command prompt (or supplied with the program
invocation) are written to that file. Command abbreviations are expanded before
doing so. Commands that are read from a scriptfile are not included. Useful to
keep a history of an interactive session, to edit/replay the session later on,
or to expand abbreviated commands.

--------------------------------------------------------------------------------
Command 'script record pause' (srp)

> Temporarily suspend writing to the current file for recording commands.

--------------------------------------------------------------------------------
Command 'script record resume' (srr)

> Resume recording commands to the script file for recording.

--------------------------------------------------------------------------------
Command 'script record start' (srsta)  [fon]

> Open a file for recording commands (append mode by default).
> Options:
    n                 suppress logging of a header with time and date
    f scriptfilename  name of the scriptfile to open ifor recording commands
                      (required, append mode by default)
    o                 open the file in overwrite mode

When a file for recording anagallis commands is open, all commands entered from
the command prompt are written to this file. To enhance readability, all
abbreviated commands are expanded.

This command can also be used to expand abbreviations in an existing scriptfile,
albeit indirectly: first open a file to record to, next paste the contents of
that scriptfile to the command prompt.

--------------------------------------------------------------------------------
Command 'script record stop' (srsto)

> Close the current file for recording commands.

--------------------------------------------------------------------------------
```

```
Command 'trees' (t)   {tc, trr, tsc, tsea, tsel, tset, tsh}

> Current number of trees in memory; has optional subcommands.
> Subcommands
    trees consense  tc    calculate consensus trees; requires a subcommand
    trees read      trr   read trees in parenthetical notation (use command
                          'import' to import TNT or nexus trees)
    trees score     tsc   list the score of the current data on one or more
                          trees
    trees search    tsea  search trees; requires a subcommand
    trees select    tsel  manipulate trees in the tree buffer; requires a
                          subcommand
    trees set       tset  overview of tree related settings; has optional
                          subcommands
    trees show      tsh   show trees; requires a subcommand

--------------------------------------------------------------------------------
Command 'trees consense' (tc)   {tcm, tcs}

> Calculate consensus trees; requires a subcommand.
> Subcommands
    trees consense majority  tcm  majority rule consensus tree
    trees consense strict    tcs  strict consensus tree

--------------------------------------------------------------------------------
Command 'trees consense majority' (tcm)  [abcdDefgikKnosStTW] <scopes>

> Majority rule consensus tree.
> Options:
    a        label the terminals with their names (this is the default; it is
             overwritten when option 'n' is present; this option is useful to
             have terminal names in such cases as well)
    b        suppress numbering of internal nodes
    c n      n is the cutoff percentage (between 0 and 99): ony clades with
             higher occurrence are shown; for building a tree (default, option
             'p' and option 'w'), values below 50 are interpreted as 50 (but
             clades theat occur less frequently are still shown with the option
             'i')
    d        dry run to set custom defaults: remember all other options in this
             invocation for use with the following invocations in this session
             (with no other options, the built-in defaults are restored)
    D        dry run to show the current custom defaults
    i        when plotting subtrees that branch at the same level, plot small
             subtrees last, and equally sized non-leaf subtrees sorted according
             to their decreasing numeric code; this option also inverses the
             plot order of leaves that branch at the same level
    g n      use alternative ASCII glyph set 1 or 2 for plotting trees (has only
             effect under 'program set unicode -' or 'psu-')
    i        when plotting subtrees that branch at the same level, plot small
             subtrees last, and equally sized non-leaf subtrees sorted according
             to their decreasing numeric code; this option also inverses the
             plot order of leaves that branch at the same level
    k        condensed output (shorter branches)
    K n      truncate terminal names (to a minimum of n characters, n > 0) when
             they would exceed the specified width (option 'W') for plotting
    n        label the terminals with their numeric code (their sequential
             number in the data matrix; see option 'a' for more information)
    f fname  write output also to file fname (append mode by default)
    o        modifies option 'f': use overwrite mode, not append mode
    s        sort terminals that branch at same level according to their
             increasing numeric code (default: ascending alphabetical order of
             terminal names)
    S        silent (suppress summary statement at start of output)
    t        interpret scopes that follow as trees to include; cannot be
             combined with option 'T'; this is the default interpretation of
             scopes
    T        interpret scopes that follow as trees to exclude; cannot be
             combined with option 't' or with default scopes
    W n      maximum width (in characters) of a single line (beyond this, the
             tree is broken into subtrees); use -1 for the width of the current
             window (default), 0 to turn off this feature (valid values 20 -
             5000)
> Argument
    scopes   one or more tree scopes (defaults to all trees)

Scopes and options may be intermingled.

--------------------------------------------------------------------------------
Command 'trees consense strict' (tcs)  [abdDfgikKnosStTwW] <scopes>

> Strict consensus tree.
> Options:
    a        label the terminals with their names (this is the default; it is
             overwritten when option 'n' is present; this option is useful to
             have terminal names in such cases as well)
    b        suppress numbering of internal nodes
    d        dry run to set custom defaults: remember all other options in this
             invocation for use with the following invocations in this session
             (with no other options, the built-in defaults are restored)
    D        dry run to show the current custom defaults
    i        when plotting subtrees that branch at the same level, plot small
             subtrees last, and equally sized non-leaf subtrees sorted according
             to their decreasing numeric code; this option also inverses the
             plot order of leaves that branch at the same level
    g n      use alternative ASCII glyph set 1 or 2 for plotting trees (has only
             effect under 'program set unicode -' or 'psu-')
    k        condensed output (shorter branches)
    K n      truncate terminal names (to a minimum of n characters, n > 0) when
             they would exceed the specified width (option 'W') for plotting
    n        label the terminals with their numeric code (their sequential
             number in the data matrix; see option 'a' for more information)
    f fname  write output also to file fname (append mode by default)
    o        modifies option 'f': use overwrite mode, not append mode
    s        sort terminals that branch at same level according to their
             increasing numeric code (default: ascending alphabetical order of
             terminal names)
    S        silent (suppress summary statement at start of output)
    t        interpret scopes that follow as trees to include; cannot be
             combined with option 'T'; this is the default interpretation of
             scopes
    T        interpret scopes that follow as trees to exclude; cannot be
             combined with option 't' or with default scopes
    w        write the consensus tree in parenthetical notation (and ignore the
             options to tweak plotting; see option 'p')
    W n      maximum width (in characters) of a single line (beyond this, the
             tree is broken into subtrees); use -1 for the width of the current
             window (default), 0 to turn off this feature (valid values 20 -
             5000)
```

```
> Argument
    scopes   one or more tree scopes (defaults to all trees)

Scopes and options may be intermingled.

--------------------------------------------------------------------------------
Command 'trees read' (trr)  [a]

> Read trees in parenthetical notation (use command 'import' to import TNT or
  nexus trees).
> Options:
    a  the terminals in the trees are indicated with their alphanumeric names,
       not with their sequential number in the current dataset

By default, terminals are indicated using numbers that correspond to the current
dataset. The first terminal in the dataset is terminal 1 (not 0). Use option 'a'
to indicate the terminals with their names.

Parentheses within a tree must match and an outer pair of parentheses is
required (so '((1 2) (3 4))' is ok, '(1 2)(3 4)' not). Different trees within a
single statement must be separated using '*', the last tree must be followed by
a semicolon. This is so because reading trees is a command that can span
multiple lines, so the semicolon is required to indicate that no more input
trees will follow.

Polytomies are internally resolved in a further unspecified way. After being
read, the trees are evaluated according to the current dataset and the current
settings for collapsing zero-length branches (and therefore the number of
terminals and their labels must match). Only those that are not yet in the tree
buffer are retained.

Trees in other formats can be imported with command 'import'.

> Examples
     characters read numeric
      1 4
      a 0
      b 0
      c 1
      d 1
     ;
     trees read (1 2 (3 4)) * (1 4 (2 3));
     trees read a
     (a c (b d))
     ;


--------------------------------------------------------------------------------
Command 'trees score' (tsc)  <treescopes>

> List the score of the current data on one or more trees.
> Argument
    tree scopes   trees to show the score of (defaults to the current tree)

List the total (weighted) score of all active characters on the trees in the
specified scopes (defaults to the current tree).

--------------------------------------------------------------------------------
Command 'trees search' (tsea)   {tseam, tseas}
```

```
> Search trees; requires a subcommand.
> Subcommands
    trees search mult  tseam  do one or more replicates of building a tree and
                              swapping it (spr or tbr)
    trees search swap  tseas  swap trees from the tree buffer (spr or tbr)
```

```
Command 'trees search mult' (tseam)  [*-ahkr] <n>

> Do one or more replicates of building a tree and swapping it (spr or tbr).
> Options:
    *    do tbr swapping (spr by default)
    -    skip swapping, just build an initial tree
    a    use addition sequence 'as is' in first replicate (default: random
         addition sequence)
    h n  hold at most n trees for each repetion
    k    keep the best trees of each replicate (even if they are not the best
         overall)
    r    use a random tree to initiate swapping
> Argument
    n   repeat this n times (n > 0; defaults to 1)
```

Do a number of repetitions (1 by default) of building a tree and (by default) swapping it. By default the build is done with a random terminal addition sequence. With the 'r' option, the build just selects a random tree. (In both cases, the reported random seed for a replicate is the current seed at the start of the build.) With the 'a' option, the build of the first replicate uses the terminal addition sequence as laid out in the dataset ('as is'). Documentation of details of swap process (when switching to a next tree to swap and things like that): to be done.

During the build stage, character hierarchy definitions are skipped: the/a best insertion point for the next terminal is searched as if no character hierarchies have been defined (but the reported score of the full tree that is obtained does take them into account afterwards). During swapping, defined character hierarchies are properly taken into account.

Options and argument may be intermingled. When more than one number is specified, the last one is used.

By default, only the best trees (according to current settings) over all replicates are retained. With option 'k', the best trees of each replicate will be retained.

See command 'trees search swap' for some comments on tree buffer maintenance.

During tree searches, the dot ('.') and the comma (',') have special meaning: hitting the dot during a tree search ends the current replicate, hitting the comma ends the complete search.

```
Command 'trees search swap' (tseas)  [*k] <tree scopes>

> Swap trees from the tree buffer (spr or tbr).
> Options:
    *  do tbr swapping (spr by default)
    k  for each original tree being swapped, keep its best trees (even if they
       are not the best overall)
```

```
> Argument
    tree scopes   trees to swap (required)

There is no default tree scope, so at least one tree scope must be specified
(this can be a trival scope of just one tree). Multiple scopes may be specified.
Scopes and options may be intermingled. Use '.' to swap all trees in memory.

Swapping any single starting tree from the specified scopes proceeds similarly
as swapping the starting tree that results from an initial build for a single
replicate with command 'trees search mult': all new trees that are derived from
the starting tree are themselves recursively swapped before proceeding to the
next starting tree; while swapping a starting tree, the numbers reported refer
to trees derived from that tree, not to all trees in the tree buffer; the best
length reported while swapping a starting tree refers to the best length
starting that starting tree, not from the current globally best length.

A difference with command 'trees search mult' is in tree buffer maintenance.
With command 'trees search mult', the tree buffer is checked and maintained
against the current best global length after each replicate. With command 'trees
search swap', global maintenance of the tree buffer is only performed after all
starting trees have been swapped.

While swapping, the dot ('.') and the comma (',') have special meaning: hitting
the dot during a tree search ends swapping of the current tree from the
specified tree scopes, hitting the comma ends the complete search.

--------------------------------------------------------------------------------
Command 'trees select' (tsel)   {tselb, tseld, tselk, tselu}

> Manipulate trees in the tree buffer; requires a subcommand.
> Subcommands
    trees select best    tselb  discard suboptimal trees
    trees select delete  tseld  discard the trees in the specified tree scopes
    trees select keep    tselk  discard the trees that are outside the specified
                                tree scopes
    trees select unique  tselu  discard duplicate trees

--------------------------------------------------------------------------------
Command 'trees select best' (tselb)

> Discard suboptimal trees.

--------------------------------------------------------------------------------
Command 'trees select delete' (tseld)  <scopes>

> Discard the trees in the specified tree scopes.
> Argument
    scopes   one or more tree scopes

--------------------------------------------------------------------------------
Command 'trees select keep' (tselk)  <scopes>

> Discard the trees that are outside the specified tree scopes.
> Argument
    scopes   one or more tree scopes

--------------------------------------------------------------------------------
Command 'trees select unique' (tselu)
```

```
> Discard duplicate trees.

When the level of collapsing zero-length branches is increased to a more severe
level (see command 'trees set zerocollapse'), trees that were different before
may no longer be different. This command can then be used to weed out
duplicates.

--------------------------------------------------------------------------------
Command 'trees set' (tset)   {tsetc, tseto, tsetw, tsetz}

> Overview of tree related settings; has optional subcommands.
> Subcommands
    trees set current       tsetc  set/show the default tree that is for example
                                   used when showing trees or character
                                   optimizations on trees
    trees set outgroup      tseto  set/show terminal(s) to be used as
                                   outgroup(s) when showing trees
    trees set width         tsetw  set/show default maximum width of a line when
                                   plotting trees
    trees set zerocollapse  tsetz  set/show the rule for collapsing zero-length
                                   branches

--------------------------------------------------------------------------------
Command 'trees set current' (tsetc)

> Set/show the default tree that is for example used when showing trees or
  character optimizations on trees.

--------------------------------------------------------------------------------
Command 'trees set outgroup' (tseto)  [a] <terminal number(s) or name(s)>

> Set/show terminal(s) to be used as outgroup(s) when showing trees.
> Options:
    a  force interpretation of argument(s) as terminal name(s)
> Argument
    terminal(s)   name(s) or numeric code(s) of terminal(s)

When one or more arguments are specified, this command sets the terminal(s) to
be used as outgroup(s) when showing trees. All current tree evaluation
algorithms are unrooted, so outgroups are extraneous to the analyses in the
strict sense.

An outgroup terminal can be specified using its numeric code (its sequential
number in the dataset, starting from one) or using its name, but both ways
cannot be mixed in a single call. So if the first outgroup terminal is specified
using its numeric code, all following outgroup terminals must be specified using
numeric codes as well; and if the first outgroup terminal is specified using its
name, then all following outgroup terminals must be specified using their names
as well.

By default, numeric arguments are interpreted as numeric terminal codes. But
anagallis allows terminal names to be numbers as well. In that case and if so
required, option 'a' can be used to skip the default interpretation of numbers.

When multiple outgroups are specified, the first terminal is considered the
primary outgroup. When showing any particular tree, the branch that is used for
rooting is the branch that divides that tree in (1) the largest group that
contains only outgroups and that includes the primary outgroup and (2) a group
that also or uniquely contains non-outgroups.
```

Without arguments, the current settings are shown.

Command 'trees set width' (tsetw)

> Set/show default maximum width of a line when plotting trees.

Beyond the requested width, trees are broken into subtrees of appropriate sizes when plotted. The value as set here can be overwritten with option 'W' of the various commands that plot trees.

The default is the width of the window in which a tree is plotted. To restore this default, set the width to -1. Set the width to zero to turn off this feature (the program actually uses a large built-in maximum then). Lines that are longer than the current window will then just wrap to the next line or lines.

Command 'trees set zerocollapse' (tsetz) [012]

```
> Set/show the rule for collapsing zero-length branches.
> Options:
    0  don't collapse branches
    1  consider a branch supported when there is at least one character that may
       have a step on that branch (as 'ambiguous =' in Nona)
    2  consider a branch supported when there is at least one character that
       must have a step on that branch (as 'ambiguous -' in Nona). Default.
```

See Coddington and Scharff (1994) for background information. As known, the default rule for collapsing that is used here may occasionally overcollapse trees. But not to the degree that their strict consensus is affected (the majority rule consensus tree may be).

Whether or not a branch is collapsed is determined by operations on the final statesets at both ends of the branch. For a character that is part of a character hierarchy, those operations are currently done using the aggregate statesets. It's probably better to loop over the non-aggregate statesets but the program doesn't do that yet.

Whether using aggregate statesets or non-aggregate statesets for characers in character hierarchies, characters with inapplicables pose yet another problem for collapsing branches. Assume state distribution ((1 -) (- 2)) on tree ((A B) (C D)). It has two optimizations: a first in which the two inner nodes have state '-', a second in which neither inner node has state '-'. The first optimization has two subcharacters and no transformations, the second has one subcharacter and one transformation. That transformation, between state 1 and state 2, has to occur on the path between A and D. So it can occur on the branch leading to A, on the inner branch, or on the branch leading to D. Depending on where the transformation is assumed to occur, both inner nodes can have state 1 or state 2. This is reflected in their (identical) optimized statesets as calculated by this program: [12] and [-] non-aggregate, [-12] aggregate. If support for the inner branch is calculated from such statesets (as is the case here), that inner branch will be collapsed, even if there exists an optimization on which a transformation occurs on that branch. That is ok under 'ambiguous -' but amounts to overcollapsing under 'ambiguous ='.One could alternatively mark that branch as supported under 'ambiguous -', but doing so leads to what might be called 'undercollapsing' in more complex cases. As an illustration, assume state distribution (1 (- (? (- 2)) on tree (A (B (C (D E))). Reasoning as above,

```
all inner nodes have [-12] as their aggregate optimal stateset. The single
transformation that may be present can now happen on the branch to A, on the
branch to E, or on both inner branches. So these inner branches might be
considered supported under 'ambiguous -' and left uncollapsed. But they cannot
be supported simultaneously by this character.

When switching to a more severe mode of collapsing, the tree buffer may end up
with duplicate trees. Command 'trees select unique' can be used to remove the
duplicates.

-------------------------------------------------------------------------------
Command 'trees show' (tsh)   {tshp, tshw}

> Show trees; requires a subcommand.
> Subcommands
    trees show plot   tshp  plot trees using character graphics
    trees show write  tshw  write trees in parenthetical notation

-------------------------------------------------------------------------------
Command 'trees show plot' (tshp)  [abdDfgikKnorsStTW] <scopes>

> Plot trees using character graphics.
> Options:
    a        label the terminals with their names (this is the default; it is
             overwritten when option 'n' is present; this option is useful to
             have terminal names in such cases as well)
    b        suppress numbering of internal nodes
    d        dry run to set custom defaults: remember all other options in this
             invocation for use with the following invocations in this session
             (with no other options, the built-in defaults are restored)
    D        dry run to show the current custom defaults
    i        when plotting subtrees that branch at the same level, plot small
             subtrees last, and equally sized non-leaf subtrees sorted according
             to their decreasing numeric code; this option also inverses the
             plot order of leaves that branch at the same level
    g n      use alternative ASCII glyph set 1 or 2 for plotting trees (has only
             effect under 'program set unicode -' or 'psu-')
    k        condensed output (shorter branches)
    K n      truncate terminal names (to a minimum of n characters, n > 0) when
             they would exceed the specified width (option 'W') for plotting
    n        label the terminals with their numeric code (their sequential
             number in the data matrix; see option 'a' for more information)
    f fname  write output also to file fname (append mode by default)
    o        modifies option 'f': use overwrite mode, not append mode
    r        show the trees as fully resolved, using a further undefined
             minimal-score resolution of polytomies
    s        sort terminals that branch at same level according to their
             increasing numeric code (default: ascending alphabetical order of
             terminal names)
    S        silent (suppress summary statement at start of output)
    t        interpret scopes that follow as trees to include; cannot be
             combined with option 'T'; this is the default interpretation of
             scopes
    T        interpret scopes that follow as trees to exclude; cannot be
             combined with option 't' or with default scopes
    W n      maximum width (in characters) of a single line (beyond this, the
             tree is broken into subtrees); use -1 for the width of the current
             window (default), 0 to turn off this feature (valid values 20 -
             5000)
```

```
> Argument
    scopes   one or more tree scopes, defaults to the current tree)

Scopes and options may be intermingled. When no scopes are present, the current
tree is shown.

Default order of subtrees within a set of subtrees that branch at the same level
is as follows: subtrees with less terminals are shown first; equally sized
non-terminal subtrees are sorted by their increasing numeric code (their
sequential order in the dataset). Terminals are shown in increasing alphabetical
order.

Option 's' can be used to sort terminals that branch at the same level according
to their increasing numeric code.

Option 'i' inverses ordering of subtrees that branch at the same level: smaller
subtrees are plotted last, equally sized non-leaf subtrees are sorted according
to their decreasing numeric code, and terminals either according to decreasing
numeric code of the terminals (option 's' specified) or descending alphabetical
order.

The sequential numeric labels of the inner nodes start from one more than the
number of terminals. The sequence is determined using the in-order traversal of
the complete tree with the default ordering of subtrees that branch at the same
level, and starting from 1 more than the number of terminals. These labels keep
being used when changing the default ordering.

--------------------------------------------------------------------------------
Command 'trees show write' (tshw)  [adDeforRsStTuv] <scopes>

> Write trees in parenthetical notation.
> Options:
    a        label the terminals with their names (default: label them with
             their numeric code, their sequential order in the dataset)
    d        dry run to set custom defaults: remember all other options in this
             invocation for use with the following invocations in this session
             (with no other options, the built-in defaults are restored)
    D        dry run to show the current custom defaults
    e str    str must be 'ana' or 'tnt'; write the trees(s) as a readable
             statement for anagallis (ana) or TNT (tnt)
    f fname  write output also to file fname (append mode by default)
    o        modifies option 'f': use overwrite mode, not append mode
    R        add a space between all tree elements, not just between terminals
    r        show the trees as fully resolved, using a further undefined
             minimal-score resolution of polytomies
    s        sort terminals that branch at same level according to their
             increasing numeric code (default: ascending alphabetical order of
             terminal names)
    S        silent (suppress summary statement at start of output)
    t        interpret scopes that follow as trees to include; cannot be
             combined with option 'T'; this is the default interpretation of
             scopes
    T        interpret scopes that follow as trees to exclude; cannot be
             combined with option 't' or with default scopes
    u        modifies r: use curly braces for unsupported nodes; no effect for
             TNT export (see option 'e')
    v        verbose (include tree scores in output); no effect for TNT export
             (see option 'e')
> Argument
```

```
    scopes   one or more tree scopes, defaults to the current tree)
```

Scopes and options may be intermingled. When no scopes are present, the current tree is shown.

By default only supported nodes are shown. When such trees are used as input for anagallis later on, any polytomies that may occur will at that time be internally resolved using a further unspecified criterion. As a consequence, the reported scores may then be higher than the scores for the original trees. This can be avoided with option 'r': the trees are then written as they are internally resolved. Using such trees as input for anagallis later on will then restore the original internal resolution of any polytomies that may occur. As a result, reported tree scores will then be identical to the tree scores of the original trees.

By default space is only added between terminal names. With option 'R' extra space is added such that all tree elements (parentheses and terminals) are separated by a space.

TNT MODE COMMANDS

These are only available in TNT mode (command 'program set tntmode =') and in imported TNT files (command 'import tnt').

TNT mode command 'ccode' (c) [+-[]/!()=]

```
> Set/show character settings.
> Options:
    +   set the character additive for the following scopes (see -)
    -   set the character non-additive for the following scopes (see +)
    [   activate the following scopes (see ])
    ]   inactivate the following scopes (see [)
    /n  set prior character weight to weight n for the following scopes
    !   not supported
    (   not supported
    )   not supported
    =   not supported
```

Only the basic code specifiers are supported. The unsupported options are recognized but then ignored. This enables anagallis to read TNT files that have ccode statements with these options without flagging an error. Remember to start counting characters from zero when using this command interactively. The changes to character codes that are made in TNT mode persist when moving back to native mode (and the other way around as well).

TNT mode command 'help' (h) [+*[]

```
> Show documentation for the commands that are available in TNT mode and in
  imported TNT files.
> Options:
    +  not supported
    *  show complete documentation of all commands
    [  not supported
```

```
By default, only a brief description is provided for each command. Option '*'
gives more information. Use 'help xyz' for help on command 'xyz'.

--------------------------------------------------------------------------------
TNT mode command 'nstates' (n)  [*&/]

> Set the TNT default datatype.
> Options:
    *  not supported
    &  not supported
    /  not supported

Only 'nstates num n' and 'nstates n' are supported, with n an integer between 1
and 30. The number of available states depend on the value of n: for n up to 8,
ten states 0-9 will be available; for larger n, 30 states are available, coded
as 0-9 and a-t (case insensitive). This is not completely the same as in TNT,
but it does mostly ensure that sufficient states are available when importing
TNT files (in TNT, for n up to 8 there are 8 available states 0-7; for n from 9
up to 16 16 states, and from 17 onwards 32 states).

--------------------------------------------------------------------------------
TNT mode command 'program set tntmode' (pst)  [=-]

> Set/show if TNT mode is on.
> Options:
    =  switch to TNT mode
    -  switch to regular mode

--------------------------------------------------------------------------------
TNT mode command 'quit' (q)

> Leave TNT mode, go back to regular mode.

--------------------------------------------------------------------------------
TNT mode command 'tread' (t)

> Read trees in parenthetical notation (numbering of terminals starts from 0).

Read trees in parenthetical notation. This command just reads the basic format
of parenthetical trees: none of TNT's special features are supported or even
parsed correctly (they will flag an error). A further restriction is that
terminals must be identified using their sequential number in the dataset
(starting from 0), not by their names. When moving back to native mode,
numbering shifts back to counting from 1 onwards (and likewise the other way
around).

--------------------------------------------------------------------------------
TNT mode command 'xread' (x)  [=*-[!/+>]

> Read alphanumeric or dna data (no support for interleaved data).
> Options:
    =  not supported
    *  not supported
    -  not supported
    [  not supported
    !  not supported
    /  not supported
    +  not supported
    >  not supported
```

TNT's xread options are recognized but then ignored. This enables anagallis to read basic character data from a TNT xread statement without flagging an error. These character data remain available when moving back to native mode (works the other way around as well). Interleaved input is not supported (an error will be triggered).
