Pseudocode for some tree search algorithms in phylogenetics

Jan De Laet
Koninklijk Belgisch Instituut voor Natuurwetenschappen
Vautierstraat 29, B-1000 Brussel, België

History and prospects

I wrote the first complete version of this text from scratch in the early summer of 2003, as a chapter draft for a multi-authored book project on POY (see Wheeler et al. 2003 and De Laet and Wheeler 2003 for POY) that later stalled. After Peter Schols drew my attention to the book "An Introduction to Bioinformatics Algorithms" by Jones and Pevzner in January 2005, I decided to release a slightly reworked version of the 2003 text available under the GNU free documentation license. Some sunny day I plan to take up the to-be-done-list on the next page. In the meantime, comments and/or criticisms are welcome at jdelaet@naturalsciences.be or jan.delaet@lid.kviv.be. Thanks to Ward Wheeler for suggesting this topic to me and for comments on early drafts; and to Kevin Nixon for making the document available as algora.pdf at http://www.plantsystematics.org/publications/jdelaet.

Jan De Laet
Brussels, 15 February 2005

1. Introduction
2. Implicit enumeration for *nt* terminals (*nt* >=2)
3. Find optimal binary trees using branch and bound, for *n*t terminals (*nt* >=2)
4. Build a tree by stepwise addition (n terminals, n >= 3)
5. Branch swapping
    5.a. Introduction
    5.b. A tree search strategy using SPR rearrangements of given trees
    5.c. A tree search strategy using TBR rearrangements of given trees
6. Ratcheting
7. Tree drifting
8. Tree fusing
9. Static approximation
10. An integrated approach
11. Some quick comments on time complexity
References

To be done:

- Remove inconsistencies in use of *italics* and Capitals
- Check correctness of code / revise code (my apologies for typo's and incomplete variable name changes)
- Use queues as queues; and stacks where stacks are the better choice
- Add sectorial searches, genetic algorithms, and MCMC-based search strategies
- Add jackknifing?
- Sort out local and global variables – too messy now
- Many things are different for single-column characters and sequence characters (see De Laet 2005) – sequence characters are not yet in here, apart from static approximation. Elaborate on that. This is strictly not about tree search but about optimization of data on a given tree. But there's an interaction when it comes to heuristics though.

1. Introduction

The basic problem that is addressed here is the problem of finding, among all possible trees for a given number of terminals, those trees that have an optimal value for a given function that computes a number from (1) a tree and (2) some comparative data for these terminals (this is called evaluation of the data on the tree). Examples of such functions, in the realm of parsimony analysis, are the Fitch-Hartigan algorithm (Fitch 1971, Hartigan 1973) for regular unordered characters and Sankoff's (1975) minimal mutation algorithm for unaligned sequences (or one of the many approximations of the latter; see De Laet 2005 for a discussion).

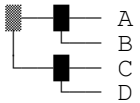
Throughout, the following representation for trees is used. An unrooted tree consists of inner nodes and leaf nodes (terminals) such that leaf nodes have one edge and inner nodes have three edges. Inner nodes are unlabeled while leaf nodes are labeled with a terminal name. Such trees are often called binary trees in literature about phylogenetic tree search, as opposed to non-binary trees in which some inner nodes have more than three incident edges (trees with polytomies) or only two incident edges. For algorithmic convenience (to give the tree a direction) an additional node, the root node, is often inserted along a (random) branch of the unrooted tree. The root node has two descendants, and these descendants are either inner nodes (with, in turn two descendants; this explains the name 'binary tree') or leaf nodes (no descendants). The root node of an isolated leaf node is the leaf itself. A rooted binary tree is represented in the same way, but here the root node has a distinct meaning: it marks the basal split in the rooted tree.

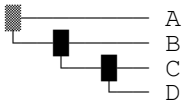
For all tree evaluation algorithms of interest, it can be shown that for any tree with a polytomy, there exists at least one binary tree in which that polytomy has been binary resolved such that this binary tree has at least an equally good value for that function. For this reason, searches for optimal trees can be restricted to binary trees. Whether or not there exist optimal trees with polytomies is next easily detected starting from optimal binary trees, by checking if such binary trees display so-called zero-length branches. A zero-length branch is roughly a branch that can be removed from a tree (resulting in a polytomy) such that the score of data on the tree does not increase (see, e.g., Coddington and Scharff 1994). Removal of zero-length branches from optimal binary trees does not only makes sense from a biological point of view (no data are present that support such branches), it can also significantly diminish the total time required for a tree search because a single optimal tree with polytomies can have many different optimal binary resolutions (if all these are kept as different best trees, the algorithms will spend a lot of time comparing trees that are basically undistinguishable by the data at hand).

In general, the only way to identify the best trees for a given data set is to evaluate all possible trees for that data set, either explicitly or implicitly by using a branch-and-bound approach. Unfortunately the number of trees grows so fast as the number of terminals increases that this only practically feasible for small numbers of terminals (e.g., using the Fitch-Hartigan algorithm and depending on the structure of the data set at hand, roughly up to 15-25 terminals). Most empirical datasets that are being used today have so many terminals that such exact algorithms are no longer practical. For such datasets, several heuristic or approximate tree search algorithms have been developed that try to avoid wasting computing effort on trees that are manifestly not optimal.

The algorithms as presented here are intended to clarify the logical structure of various procedures that are involved in tree searching. They do not represent the state of the art in coding efficiency when it comes to execution time or memory usage.

As an example of the tree representation that is used here, and using ▒ for the root node and ▉ for other inner nodes, the two following trees for terminals A, B, C, and D are identical as unrooted trees but different as rooted trees:





With this convention about the root node, a node always (rooted and unrooted) unambiguously represents a subtree (in the case of a terminal node, the trivial subtree of one terminal; in the case of the root node, the whole tree), so the terms ‘node‘ and ‘tree’ are freely interchangeable. Nodes can have various values attached to them that can be set and read. These are indicated using dot notation; e.g. if variable *node* points to subtree (C D) in the above tree,

node.cost <- Evaluate (*node*)

sets the cost of that subtree to the value as returned by the call of function *evaluate(node),* while

node.cost

retrieves that value afterwards.

Here are a number of variables and functions that are used throughout:

nt terminals, numbered from *1* to *nt* (*nt* > 2)
nc characters, numbered from *1* to *nc* (*nt* > 0)
obs : matrix of state distributions of a number of characters for the *nt* terminals; *obs[i, j]* refers to the character state(s) observed for character *i* and terminal *j;* these are the data that are optimized on trees.

left (tree) : returns left (or upper or first) subtree of *tree*(NIL if is_leaf(*tree*)) [dot notation could be used here as well]

right (tree) : returns right (or lower or second) subtree of *tree* (NIL if is_leaf(*tree*))

root (tree) : returns the root node of *tree*

ancestor(tree) : returns the node that has *tree* as a descendant (NIL when *tree* is the root node)

is_leaf (tree) : returns true if tree is a leaf node, returns false otherwise

evaluate (tree, obs) : returns the cost of *obs* on *tree,* using some character optimization algorithm that is further mostly left unspecified in this paper.

add_to_queue (element, a_queue_of_elements) : adds element *element* to queue *queue_of_elements* (e.g., add a tree to a queue of trees)

cond_add_to_queue (element, a_queue_of_elements) : as add_to_queue, but only add the element if it is not present yet

is_in_queue (element, a_queue_of_elements) : returns true if element *element* is present in queue *queue_of_elements* , false otherwise

take_from_queue (a_queue_of_elements) : removes the last element that was added to queue *queue_of_elements* and returns that element (returns EmptyQueue when there are no elements in queue *queue_of_elements*)

concat_queues(queue1, queue2) : take, one by one, all elements from *queue2*, add them to *queu1*, and return *queue1*

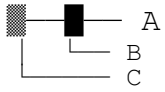
clear_queue (a_queue) : makes queue *a_queue* empty

initialize_tree (terminal_1, terminal_2): returns a tree *t* with *t = root(t)*, *left(root(t)) = terminal_1*, and *right(root(t)) = terminal_2*.
E.g., with A and B two terminals, *initialize_tree A, B)* will reurn the root node of the following tree:

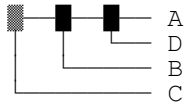
tree_copy (*node*) : returns a copy of tree *node* (this implies that subsequent changes to that copy will not be visible in the original)

insert_node (to_insert, insert_at, direction): if *isleaf (insert_at)* nothing happens, else this call has as a side effect that node *insert_at* is changed such that it has an additional inner node along the branch from *insert_at* to *left (insert_at)* [direction '*left*'] or *right (insert_at)* [direction '*right*']; the left (right) descendant of that new node is the original left (right) descencant of *insert_at*, the other descendant of the new node is *to_insert*

As an example, with *t* the root node of the tree below, *n1* the inner node at (A B) in that tree, and D, E, and F terminals



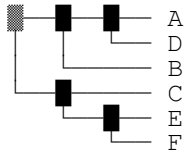
the statement
insert_node (*n1*, D, left)
will change *t* into



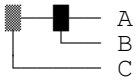
Subsequently, the statement

insert_node (*t*, (E, F), right)

will change *t* into (note presence of D in the result of this second call)



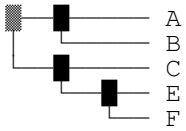
cp_insert_node (to_insert,insert_at, direction) : works as insert_node, but operates on a fresh copy of *root(insert_at)*, and returns this modified copy (i.e., *insert_at* and *root(insert_at)* are not changed directly). With *t'* the root node of the tree below, *n1'* the inner node at (A B) in that tree, and D, E, and F terminals



the sequence

tree1 <- insert_node((A B), D, left)
tree2 <- insert_node(t', (E F), right)

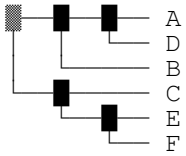
results in a tree2 that looks like this (it does not have terminal D):



insert_node_below (to_insert, insert_at) : as *insert_node (to_insert, insert_at, direction)* but insertion happens along the branch leading from the ancestor of *insert_at* to *insert_at*. If *insert_at = root(insert_at)*, nothing happens

cp_insert_node_below (to_insert, insert_at) : as *insert_node_below (to_insert, insert_at)* but operates on a fresh copy of *root(insert_at)*

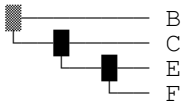
chop_tree (chop_at, direction) : if *is_leaf(chop_at)*, nothing happens; else this function removes a subtree from *root(chop_at)* and returns that subtree in a way that is illustrated in the following example.
With *chop_at* the inner node at ((A D) B) in the following tree *root(chop_at)*:



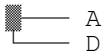
the statement

pruned_branch <- chop_tree (*chop_at, left*)

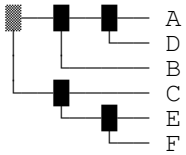
will have as a result that the tree *root(chop_at)* is turned into



while *pruned-branch* looks like this:



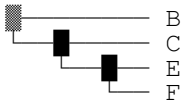
cp_chop_tree (chop_at, direction) : works as *chop_tree* but operates on a copy of *root(chop_a*), which itself remains unchanged.
The result is a pair of trees *(trunk, pruned_branch).*
As an example, with *chop_at* set to the the inner node at ((A D) B) in the following tree *root(chop_at)*:



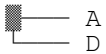
the statement

(trunk, pruned_branch) <- chop_tree (*chop_at, left*)

will leave the tree at *root(chop_at)* unchanged while trunk looks like:



and *pruned*-branch looks like this:



Other variables and functions are introduced as they are needed.

2. Implicit enumeration for *nt* terminals (*nt* >=2)

This algorithm generates and processes all binary trees for *nt* taxa. It starts out by creating an initial tree of two terminals. Next, the remaining terminals are added one by one, each time in all possible positions on the growing tree. There are two levels of recursion: firstly, the next terminal is added to all positions in the growing tree in recursive function Add_next_terminal_everywhere. Second, each time a next taxon is added somewhere in a growing tree (in function Add_next_terminal_here), a recursive call of All_trees with the resulting tree and the remaining terminals will add the first of these remaining terminals in all possible positions in the new tree (and then further recurse). The algorithm below deals with unrooted trees, but it is indicated where and which changes have to be made for rooted trees. The unrooted algorithm is sufficient when the cost calculations are root independent.

current_best_cost : the best estimate for the optimal length available at any point in the algorithm, initialized as positive infinity
best_trees : queue of all trees of *cost current_best_cost* that have already been found; is empty initially

```
Process (tree)
    tree.cost <- evaluate (tree, obs)
    if tree.cost < current_best_cost then begin
        current_best_cost = tree.cost;
        clear_queue (best_trees);
    end
    if tree.cost = current_best_cost then Add_to_queue (best_trees, tree)   (*
        1. alternatively, and mostly more efficiently, the cost of the growing tree can be calculated/updated using incremental optimization in function
           Add_next_terminal_here;   in this way, the global evaluation (for every tree) in the preceding line becomes superfluous (see '2. Branch and bound' for an
           example). The version as presented here (evaluation only when all terminals have been added) is implemented in poy (command -alltrees).
        2. by in turn rerooting (and re-evaluating) tree tree to all its branches in this function, this algorithm will generate all rooted binary trees.  This is necessary
           if the algorithm for evaluating a tree is root dependent (otherwise the best trees might not be found).
        3. all kinds of other things can be done here, like updating a cost frequency table *)


Add_next_terminal_here (num_terminalsleft, tree)
  if not is_leaf (tree) then begin
        All_trees (num_terminalsleft - 1, cp_insert_node (num_terminalsleft, tree, left));
        If not tree = root(tree) then All_trees (num_terminalsleft - 1, cp_insert_node (num_terminalsleft, tree, right));
          (* note that this is very demanding on memory because at each call of Add_next_terminal_here (with tree not a leaf), two tree are made. Alternatively, the
          new node and terminal in could be added in the original tree (using insert_node); in that case, the tree would have to be restored to its original state before
          going on after the recursive All_trees calls. This approach would be vastly less demanding on memory. *)
    end
```

```
Add_next_terminal_everywhere (num_terminals_left,  tree)
        if not is_leaf (tree) then begin
                Add_next_terminal_here (num_terminals_left, tree)
                Add_next_terminal_everywhere (num_terminalsleft, left(tree))
                Add_next_terminal_everywhere (num_terminalsleft, right(tree))
        end


All_trees (num_terminals_left, tree) =
        if num_terminals_left = 0 then Process (tree)
        else add_next_terminal_everywhere (num_terminals_left, tree)
```

Program:

All_trees(*nt – 2* , initialize_tree (*nt*, *nt-1*))

After execution, all optimal trees are in queue best_trees, and the optimal cost is *current_best_cost*

3. Find optimal binary trees using branch and bound, for *n*t terminals (*nt* >=2)

This algorithm, a straightforward modification of (1), finds optimal trees using a process of branch and bound (a general optimization procedure first applied to phylogenetic tree search by Penny and Hendy 1982). It consists of calculating tree costs of growing trees and stopping search paths that start from partial trees with a length that already exceeds the length of the current best (complete) tree. Compared to implicit enumeration, branch and bound can be substantially faster in finding the optimal length and the optimal trees. On the other hand, it does not allow, e.g., calculation of frequency plots of tree lengths (because not all trees are evaluated). As an intermediate, branch and bound could be implemented such that search paths are not aborted until the partial tree exceeds the current length + *n*. This would still allow to calculate an exact length frequency distribution between the optimal length and trees that are up to *n* more costly. The algorithm below deals with unrooted trees, and remarks indicate where changes have to be made for rooted trees.

current_best_cost : the best estimate for the optimal length available at any point in the algorithm; see below for initialization
best_trees : queue of all trees of *cost current_best_cost* that have already been found; is empty initially

```
Process_tree
    if tree.cost < current_best_cost then begin
        current_best_cost = tree.cost; // tree.cost has been set in Insert_node
        Clear_queue (best_trees); (* slop value can be applied here *)
    end
    if tree.cost = global_best_cost then add_to_queue (best_trees, tree) (* slop value can be applied here *)


Add_next_terminal_here (num_terminalsleft, tree)
    if not is_leaf (tree) then begin
        newtreeleft = cp_insert_node (num_terminalsleft, tree, left);  // see 1. for some remarks on memory usage
        newtreeleft.cost = evaluate (newtreeleft, obs);
        (* This could be a full evaluation of the tree, but a more efficient alternative would be: (1) calculate the cost of inserting the node and terminal by means of
        incremental optimization (see Gladstein1997); (2) update the node.cost for all subtrees of newtreeleft (root inclusive) for which the cost has changed. Under
        this approach, the initial tree has to be evaluated before Branch_and_bound is called, and function cp_insert_node has to copy all required information from
        tree to newtreeleft *)
        if newtreeleft.cost <= current_best_cost then Branch_and_bound (num_terminalsleft - 1, newtreeleft);  // search paths can get aborted here

        If not (tree = root(tree)) then begin
                newtreeright = cp_insert_node (num_terminalsleft, tree, right);
                newtreeright.cost = evaluate (newtreeright, obs);
                if newtreeright.cost <= current_best_cost then Branch_and_bound (num_terminalsleft - 1, newtreeright);  // search paths can get aborted here
        end (* This is sufficient when the cost calculation algorithm is root independent (i.e., only unrooted trees have to be checked); with cost calculation
                algorithms that are root sensitive, newtreeleft and newtreeright have to evaluated with all possible positions of the root (can be done efficiently using
                incremental optimization); for each position of the root, the conditional recursive call of Branch_and_bound must be made *)
    end
```

```
Add_next_terminal_everywhere (num_terminals_left,  tree)
        if not is_leaf (tree) then begin
                Add_next_terminal_here (num_terminals_left, tree)
                Add_next_terminal_everywhere (num_terminalsleft, left(tree))
                Add_next_terminal_everywhere (num_terminalsleft, right(tree))
        end


Branch_and_bound (num_terminals_left, tree) =
        if num_terminals_left = 0 then process_tree (tree)
        else add_next_terminal_everywhere (num_terminals_left, tree)
```

Program:

initialize_cost (*current_best_cost*)

(* No pseudocode provided for this one. The current best cost can be initialized with positive infinity, but for optimizing performance the length of a tree that is obtained using a quick search (e.g., a single random addition sequence, or a single random addition sequence followed by SPR) will do much better; this is because the lower the current best length, the more and the earlier search paths can be aborted; *)

initial_tree = initialize_tree (*nt, nt-1*)

Branch_and_bound(*nt – 2* , initial_tree)

After execution, all optimal trees are in queue best_trees, and the optimal cost is *current_best_cost*

4. Build a tree by stepwise addition (n terminals, n >= 3)

This algorithm builds a tree (or set of trees) by adding new terminals one by one to a growing tree. The initial tree is the single tree for the two first terminals. Each time, the next terminal is added along that branch that makes the increase in cost (of adding that terminal to the tree) minimal. The algorithm is such that, in the case of ties of adding a taxon to a tree, all equally good insertion points will be used as starting points for adding the next terminal; these different starting points are evaluated separately (as an example, with a tie between two insertion points for adding the previous terminal, the trees that result from those two insertion points will be used independently as a starting point for adding the next terminal; so, when adding that next terminal, the best insertion points for the first tree and the best insertion points for the second tree will both be kept for the next round, whether or not these are equally good). Some possible variations are indicated in comments. The resulting tree or trees are not guaranteed to be globally optimal and in fact they can be quite suboptimal. The algorithm is widely used nevertheless because it allows to fast and easily generate trees that are most probably not too bad (note that different runs of the algorithm with a different addition sequence of terminals can and often results in different trees). These trees can next be used as starting trees for swapping, as discussed further.

current_best_cost : the best estimate for inserting the next terminal to one particular tree
trees : queue of all different trees with all terminals that have already been found; is empty initially
terminals : array[1..*nt*] of terminal identification numbers 1..*nt*; if, for all *i*, terminals[*i*] = *i*, the addition sequence is 'as is'; to have a random addition sequence, it is sufficient to randomize this array before calling Build_a_tree

```
Try_insertion_here (node_to_insert, insert_below)
    If not (insert_below = root(tree) OR root(tree)).right = tree) then //second cond. for unrooted only (because insertion there has already been done, through root.left)
        newtree = cp_insert_node_below (node_to_insert, insert_below);
        insert_below.cost = evaluate (newtree, obs);
          (* for efficiency, use incremental optimization; note that node.cost here is the cost that would apply if insertion were at the branch leading to node *)
        if insert.below.cost < current_best_cost then current_best_cost <- insert.below.cost
    end
```

```
Try_insertion_everywhere (node_to_insert, insert_below)
    Try_insertion_here (node_to_insert, insert_below)
    if not is_leaf (insert_below) then begin
        Try_insertion_everywhere (node_to_insert, left (insert_below))
        Try_insertion_everywhere (node_to_insert, right (insert_below))
    end
```

```
Get_nodes_at_cost (tree, cost) // returns a queue of all non-leaf subtrees of tree (inclusive) with cost equal to cost (* a slop value can be applied here *)
    // alternatively this queue can directly be build in Try_insertion_here; see 4. for an example
    if is_leaf (tree) then return EmpyQueue
    else if tree.cost = cost then return add_to_queue (tree, concat_queues (Get_nodes_at_cost (left (tree)), Get_nodes_at_cost (right (tree))))
    else  return concat_queues (Get_nodes_at_cost (left (tree)), Get_nodes_at_cost (right (tree))))
```

```
Build_a_tree (num_terminals_left, tree) =
    if num_terminals_left = 0 then cond_add_to_queue (tree, trees) // alternatively, only keep trees of lowest cost here, and clear trees if a new best cost is obtained
    else begin
        current_best_cost <- positive infinity
        tree.cost <- positive infinity
        Try_insertion_everywhere (terminals[nt + 1 - num_terminals_left], tree)
        queue_of_best_insertion_points <- Get_nodes_at_cost (tree, current_best_cost)
        while queue_of_best_insertion_points <> EmptyQueue do  // alternatively this can be done for only one insertion point
            an_insertion_point <- take_from_queue (queue_of_best_insertion_points)
            starting_point <- cp_insert_node_below (an_insertion_point, num_terminals_left)
            Build_a_tree (num_terminals_left – 1, starting_point) // alternatively, only do this when for starting points that are optimal at this stage
        done;
    end


Build (obs)
    Build_a_tree( nt – 2 , initialize_tree(nt, nt-1))
    Return trees
```

Program:

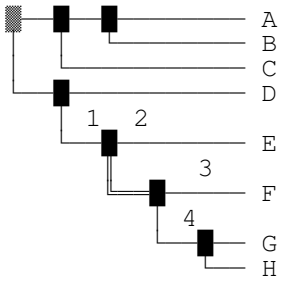
```
    tree_queue <- Build (obs)
```

5. Branch swapping

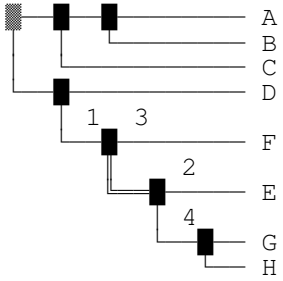
5.a. Introduction

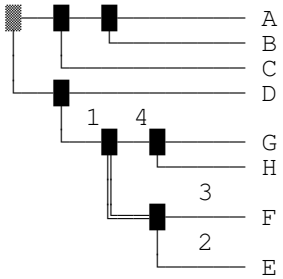
Branch swapping is general heuristic procedure in which an existing tree is modified (‘rearranged) by moving some branches around, mostly with the aim of obtaining a tree with a better cost, or additional trees with equally good cost. Well-known types of branch swapping are NNI (nearest neighbor interchange), SPR (subtree pruning and regrafting), and TBR (tree bisection and reconnection).

In NNI the rearrangements that are tried are the ‘nearest neighbor interchanges’. Nearest neighbors in a tree are defined as illustrated in the following figure: each inner branch of a tree connects two inner nodes, and these two inner nodes each have two additional branches connected to them. These four additional branches are the nearest neighbors relative to the inner branch in between.



By interchanging the nearest neighbors along a given internal branch, two new trees can be generated. With the previous example:

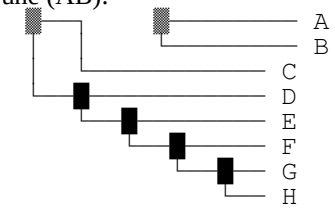




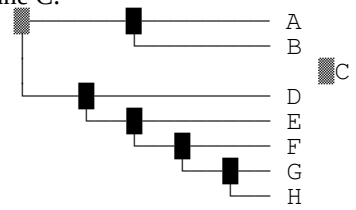
NNI as a heuristic tree search strategy, starting from a set of initial trees (see, e.g., (3)) could be done as follows: try out all NNI rearrangements of the initial trees as to find better trees. If such trees are found, start a next round of trying out NNI rearrangements starting from these new trees. Keep doing this until no better trees are found.

SPR considers rearrangements that can be made as follows: remove a clade from a tree such that two trees are obtained. Next insert a new inner node along one of the branches of the tree that contains the original root node and attach the other tree as the second sibling of that new inner node. Some examples of SPR rearrangements of the previous tree that are not NNI rearrangements are shown below.

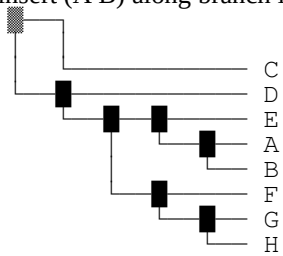
Prune (AB):



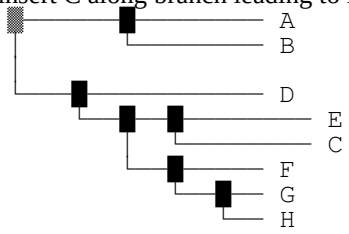
Prune C:



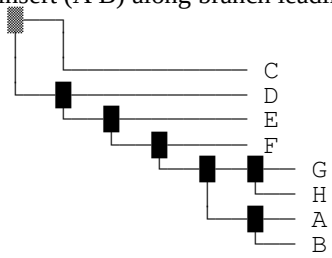
Re-insert (A B) along branch leading to E:



Re-insert C along branch leading to E:



Re-insert (A B) along branch leading to (G H):

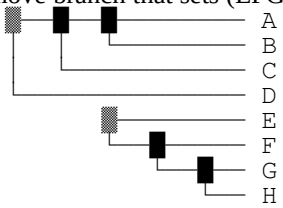


Re-insert C along branch leading to (G H):

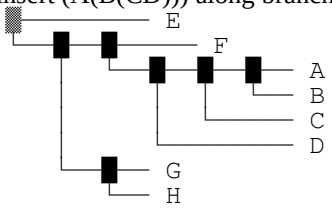
Pseudocode for a tree search strategy using SPR rearrangements as defined here is given in 5.b.

Note that the set of possible SPR rearrangements of a tree as defined above depends on the position of the root node. A broader definition of an spr rearrangement that removes this dependency is as follows: remove a branch from a tree such that two trees are obtained. Next insert an new inner node along a branch of either tree and attach the other tree as the second sibling of that new inner node. Below is a rearrangement of the tree of the previous example that satisfies the broad but not the narrow definition:

Remove branch that sets (EFGI) apart:

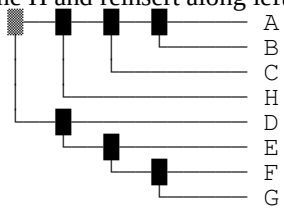


Re-insert (A(B(CD))) along branch leading to F:

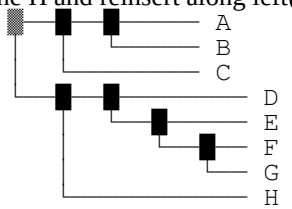


In both cases (narrow and broad definition), there is a difference whether or not the original tree is considered rooted or unrooted when the root node is involved:

Prune H and reinsert along left(root node):



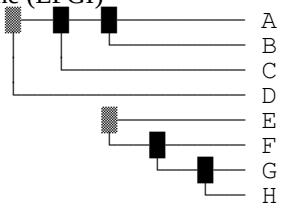
Prune H and reinsert along left(root node):



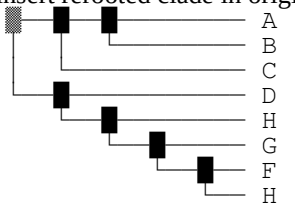
The tree that results from re-inserting along left(root node) and right(root node) are identical as unrooted trees but different as rooted trees. So it may be (marginally) useful to try both alternatives when using a cost calculation algorithm that is root dependent but it is wasted effort when using a root-independent cost algorithm.

TBR rearrangements, lastly, are like SPR rearrangements but the tree that is inserted can be rerooted before reinsertion. So all SPR rearrangements are TBR rearrangements but not the other way around. Here are some examples of tbr rearrangements that are not spr rearrangements:

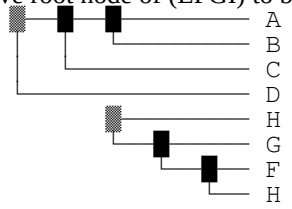
Prune (EFGI)



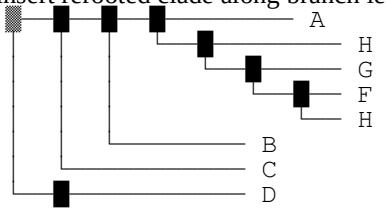
Re-insert rerooted clade in original position:



Move root node of (EFGI) to between (EFG) and H



Re-insert rerooted clade along branch leading to A



Pseudocode for a search strategy using TBR rearrangements is given in 5.c.

5.b. A tree search strategy using SPR rearrangements

This algorithm describes a heuristic tree search strategy (and some possible modifications) based on SPR rearrangements (narrow definition) of a set of given unrooted trees.

trees_to_swap : queue of trees to swap; initially it contains a set of given trees
results : queue of best trees found during swapping; initially empty
current_best_cost : the best estimate found while rearranging a particular tree
global_best_cost : the best cost over all trees that are to be swapped or have already been swapped; initialized as the best cost over trees_to_swap
chopped_at : indicates where the tree that is being swapped has been cut (reinsertion at that site does not have to be tried)
stop_swapping_this_tree : Boolean, cf. alternative 4 in function Try_insertion_here (see below)

```
Try_insertion_here (node_to_insert, insert_below)
    If not (insert_below = root(tree) OR root(tree)).right = tree) then begin
        newtree = cp_insert_node_below (node_to_insert, insert_below);
        newtree.cost = evaluate (newtree, obs);   (* for efficiency, use incremental optimization *)

        (* three alternatives follow, each one is a different search strategy – many more alternatives exist *)
        (* alternative 1: keep (and subsequently swap) all new trees that are at least as good as the tree being swapped; probably not a good idea *)
        if newtree.cost =< current_best_cost then cond_add_to_queue newtree trees_to_swap (* a slop value can be applied *)

         (* alternative 2: only keep (and subsequently swap) trees if they are at least as good as the best trees found globally thus far *)
        if newtree.cost < global_best_cost then clear_queue trees_to_swap (* a slop value can be applied *)
        if newtree.cost =< global_best_cost then cond_add_to_queue newtree trees_to_swap  (* a slop value can be applied *)

        (* alternative 3: as 2., but stop swapping this tree right away *)
        if newtree.cost < global_best_cost then begin
            clear_queue trees_to_swap (* a slop value can be applied *)
            stop_swapping_this_tree <- true
        end
        if newtree.cost =< global_best_cost then cond_add_to_queue newtree trees_to_swap  (* a slop value can be applied *)

        (* for all alternatives: *)
        if newtree.cost < current_best_cost then current_best_cost <- newtree.cost
        if newtree.cost < global_best_cost then global_best_cost <- newtree.cost
    end
```

```
Try_insertion_everywhere (node_to_insert, insert_below)
        If not (insert_below = chopped_at) then Try_insertion_here (node_to_insert, insert_below)
        if not is_leaf (insert_below) AND not stop_swapping_this_tree then begin
                Try_insertion_everywhere (node_to_insert, left (insert_below))
                Try_insertion_everywhere (node_to_insert, right (insert_below))
        end


Swap_a_tree_spr (tree) =
        if not root (tree) = tree then begin
                (trunk, pruned_branch) <- cp_chop_tree_below (tree) // make sure statesets in the two trees are set correctly
                chopped_at <- tree
                Try_insertion_everywhere (pruned_branch, trunk)
        end
        if not is_leaf (tree) AND not stop_swapping_this _tree then begin
                Swap_a_tree_spr (left(tree))
                Swap_a_tree_spr (right(tree))
        end


Swap_trees_spr (intrees, obs)
        trees_to_swap <- intrees
        while trees_to_swap <> EmptyQueue do
                tree_to_swap <- take_from_queue trees_to_swap
                current_best_cost <- evaluate (tree_to_swap)
                stop_swapping_this_tree <- false
                Swap_a_tree_spr (tree_to_swap)
                if tree_to_swap.cost < global_best_cost then clear_queue results (* a slop value can be applied here *)
                if tree_to_swap.cost <= global_best_cost then cond_add_to_queue tree_to_swap results (* slop value can be applied here *)
        done
        return results

Get_initial_trees (obs) // just an example
        return Build_a_tree (obs, weights)                    // see (4)
```

Program:

```
        tree_queue <- Swap_trees (Get_initial_trees (obs), obs)
```

5.c. A tree search strategy using TBR rearrangements

This algorithm describes a heuristic tree search strategy (and some possible modifications) based on TBR rearrangements of a set of given unrooted trees. Very similar to 5.b, but all possible rerootings of pruned trees are now tried out as well (as opposed too just the original root in 5.b.).

trees_to_swap : queue of trees to swap
results : queue of best trees found during swapping; initially empty
current_best_cost : the best estimate found while rearranging a particular tree
global_best_cost : the best cost over all trees that are to be swapped or have already been swapped; initialized as the best cost over trees_to_swap
chopped_at : indicates where the tree that is being swapped has been cut (reinsertion at that site does not have to be tried)
original_root : the original root node of a pruned clade
stop_swapping_this_tree : Boolean, cf. alternative 4 in Try_insertion_here (see (5.b))

Try_insertion_here (node_to_insert, insert_below) -> as in 4.b

Reroot (tree, direction) : if is_leaf (tree) or root(tree) <> tree, tree is returned unchanged; else

1. direction = 'left': the root node of tree is moved to left(tree) and the original branch from the root to left(root) becomes the left descendant of the new root node.
2. direction = 'right: the root node of tree is moved to right(tree) and the original branch from the root to right(root) becomes the right descendant of the new root node.

[each time make sure appropriate statesets are set correctly and as efficiently as possible]
return the new root node is returned

```
Try_insertion_everywhere (node_to_insert, insert_below)
    If not (insert_below = chopped_at AND original_root = node_to_insert) then Try_insertion_here (node_to_insert, insert_below)
    if not is_leaf (insert_below) AND not stop_swapping_this_tree then begin
        Try_insertion_everywhere (node_to_insert, left (insert_below))
        Try_insertion_everywhere (node_to_insert, right (insert_below))
    end

Try_unrooted_insertion_everywhere (to_insert, insert_below)
    Try_insertion_everywhere(to_insert, insert_below)
    if not is_leaf (to_insert) then begin
        Try_unrooted_insertion_everywhere (reroot (to_insert, left), insert_below)
        to_insert <- reroot (to_insert, left)
        Try_unrooted_insertion_everywhere (reroot (to_insert, right), insert_below)
        to_insert <- reroot (to_insert, right)
    end
```

```
Swap_a_tree_tbr (tree) =
    if not root (tree) = tree then begin
        (trunk, pruned_branch) <- cp_chop_tree_below (tree) // make sure statesets in the two subtrees are set correctly
        chopped_at <- tree
        original_root <- pruned_branch
        Try_unrooted_insertion_everywhere (pruned_branch, trunk)
    end
    if not is_leaf (tree) AND not stop_swapping_this_tree then begin
        Swap_a_tree_tbr (left(tree))
        Swap_a_tree_tbr (right(tree))
    end


Swap_trees_tbr (intrees, obs)
    trees_to_swap <- intrees
    while trees_to_swap <> EmptyQueue do
        tree_to_swap <- take_from_queue trees_to_swap
        current_best_cost <- evaluate (tree_to_swap)
        stop_swapping_this_tree <- false
        Swap_a_tree_tbr (tree_to_swap)
        if tree_to_swap.cost < global_best_cost then clear_queue results (* a slop value can be applied here *)
        if tree_to_swap.cost <= global_best_cost then cond_add_to_queue tree_to_swap results (* slop value can be applied here *)
    done
    return results
```

Program:

```
    tree_queue <- Swap_trees_tbr (Get_initial_trees (obs), obs)   // see (5.b) for Get_initial_trees
```

After execution, the trees that result from the search strategy are in queue results.
Note that the new trees are being swapped in the order that they come in. This can be reversed using stacks instead of queues.

6. Ratcheting

The ratchet (Nixon 1999) is a tree search strategy that is built on top of simple branch swapping as in (5): it uses Swap_trees_spr (trees) or Swap_trees_tbr (trees) (or the like) in an iterative approach in which characters are periodically reweighted. The reweighting consists of changing the weight of a random subset of characters. The idea is that iterating between original weights and weights that are randomly changed will make it possible to escape local minima in which simple swapping might get stuck. Examples and benchmarks can be found in Nixon (1999) and Goloboff (1999).
In POY, ratcheting is requested with the commands –ratchetspr and –ratchettbr (ratcheting using spr or tbr). Both commands take the number of iterations to perform as an argument. The size of the random subset to be reweighted is determined with command –ratchetpercent, and reweighting consists of multiplying the original weight by a factor that can be set with command –ratchetseverity.

weights: array of *nc* character weights
num_iterations: number of iterations to perform
num_to_reweight: number of characters to reweight (num_to_reweight <= *nc*).
weight_factor: weight multiplier to be used in Perturb_weights
swap_type : 'spr' or 'tbr'

Reset_weights (*weights*): restore the original character weights

Perturb_weights (*weights*): select a random subset of *num_to_reweight* characters and multiply the weights of the characters in this set by *weight_factor*

Get_initial_swapped_trees (*obs*) // just an example
trees <- Build_a_tree (*obs*, *weights*) // see (4)
trees <- Swap_trees (*obs*, trees, *weights*, *swap_type*) // see (5)
return trees

Ratchet(*obs*, *swap_type*, *weights*, *num_to_reweight*, *weight_factor*, intrees)
trees <- intrees
for iteration <- 1 to *num_iterations* do
Perturb_weights (*weights*)
trees <- Swap_trees (*obs*, trees, *weights*, *swap_type*)
Reset_weights (*weights*)
trees <- Swap_trees (*obs*, trees, *weights*, *swap_type*)
done
return trees

Program:
tree_queue <- Ratchet (*obs*, *swap_type*, *weights*, *num_to_reweight*, *weight_factor*, Get_initial_swapped_trees (*obs*))

7. Tree drifting

Like the ratchet, tree drifting (Goloboff 1999) is an iterative search strategy one level above simple branch swapping. In this case, iterations are between regular branch swapping, and swapping in which suboptimal trees are accepted with a probability that depends on their degree of suboptimality. This is an application of *simulated annealing*, a general heuristic optimization procedure in which suboptimal solutions may be temporarily accepted as a means to escape local optima.

Goloboff (1999) described an acceptance probability that is based on the length difference and on the RFD or Relative Fit Difference (Goloboff and Farris 2001) between a candidate tree and the current best tree. Poy uses a simpler acceptance probability that is based on the length difference only (see De Laet and Wheeler 2005).

In POY, drifting is requested with the commands –driftspr and –ratchetspr (drifting using SPR or TBR). Both commands by default perform one iteration, but more iterations can be requested with commands –numdriftspr and –numdrifttbr. In the drift step of an iteration, swapping goes on until a specified number of equally good or suboptimal trees have been accepted (or until completion, if this number is not reached); this number is set with command –numdriftchanges.

num_iterations: number of iterations to perform
num_to_accept: number of equally good or suboptimal trees to accept before stopping the drift step of an iteration
num_accepted: number of equally good or suboptimal trees already accepted in the drift step of an iteration
swap_type : 'spr' or 'tbr'
accept_function: a function that takes as arguments *obs* and two trees: the tree being swapped and a candidate tree; it returns true if the candidate tree is accepted, false otherwise; this function hides the specifics of the acceptance probability (in POY, e.g., the function would apply different probability functions for equally good versus suboptimal trees).

```
Drift_try_insertion_here (node_to_insert, insert_below, accept_function) // also uses current_tree, the current tree being swapped
    If not (insert_below = root(tree) OR root(tree)).right = tree) then begin
        newtree = cp_insert_node_below (node_to_insert, insert_below);
        newtree..cost = evaluate (newtree, obs);   (* for efficiency, use incremental optimization *)
         (* this is a modification of alternative three of (5) *)
        if newtree.cost < global_best_cost then begin
            clear_queue trees_to_swap (* a slop value can be applied, or suboptimal trees kept using a probability function *)
            stop_swapping_this_tree <- true
            cond_add_to_queue newtree trees_to_swap  (* a slop value can be applied *)
        end
        else if accept_function (obs current_tree, newtree) AND num_accepted < num_to_accept then begin
            stop_swapping_this_tree <- true
            cond_add_to_queue newtree trees_to_swap  (* a slop value can be applied *)
            num_accepted <- num_accepted + 1 (* or only do this if the new tree was not yet in the trees_to_swap queue *)
        end
        if newtree.cost < current_best_cost then current_best_cost <- newtree.cost
        if newtree.cost < global_best_cost then global_best_cost <- newtree.cost
    end
```

```
Drift_swap_trees_tbr (trees_to_swap, obs)
    while trees_to_swap <> EmptyQueue AND num_accepted < num_to_accept do  // the second part of the condition is the only difference with Swap_trees_tbr (5)
        tree_to_swap <- take_from_queue trees_to_swap
        current_best_cost <- evaluate (tree_to_swap)
        stop_swapping_this_tree <- false
        Swap_a_tree_tbr (tree_to_swap)
        if tree_to_swap.cost < global_best_cost then clear_queue results (* a slop value can be applied here *)
        if tree_to_swap.cost <= global_best_cost then cond_add_to_queue tree_to_swap results (* slop value can be applied here *)
    done


Drift_swap_trees (obs, trees, swap_type, num_to_accept, accept_function)
    (* mostly as in Swap_trees in 5; just stop swapping when the num_to_accept limit is reached (see Drift_swap_trees_tbr above for an example) and use
    Drift_try_insertion_here (see above) instead of Try_insertion_here *)


Drift (obs, swap_type, num_to_accept, num_iterations, accept_functionI, intrees)
    trees <- intrees
    for iteration <- 1 to num_iterations do
        num_accepted <- 0
        trees <- Drift_swap_trees (obs, trees, swap_type, num_to_accept, accept_function)
        trees <- Swap_trees (obs, trees, swap_type)                    // as in 5
    done
    return trees
```

Program:

```
    tree_queue <- Drift (obs, swap_type, num_to_accept, num_iterations, accept_function, Get_initial_swapped_trees (obs))
        // see (6) for an example of Get_initial_swapped_trees
```

8. Tree fusing

Tree fusing (Goloboff 1999; see also Moilanen 1999, 2001) is based on the idea that one might escape local optima by exchanging, between different trees, subgroups of identical composition but not resolution.

In POY, fusing is requested with the command –treefuse. As inputtrees for fusing, POY uses trees that are read from a file or from the commandline, and/or trees that have first been build (and swapped). The minimum size of subtrees to be exchanged (5 by default) is set with the command –fusemingroup. The maximum number of pairs of trees to fuse is set with –fuselimit (all different unordered pairs of inputtrees by default). Command –fusingrounds determines if new trees that are found during the current fusing round will be submitted to a next round of fusing (only two rounds allowed, currently) . Commands –treefusespr and –treefusetbr determine if a new tree that is found during fusing is submitted to spr or tbr branch swapping right away. The pseudocode below reflects how treefusing is implemented in poy, which is slightly different from Goloboff (1999: 419).

results : queue with trees that result from fusing; is empty initially
trees : queue with trees to be swapped
do_swap : boolean variable to determine if trees that originated by clade exchange (fuse_two_trees) are to be swapped
minfuse : minimum size of clades to be exchanged

```
get_group (tree, subtree)
    if tree has a group clade that has all terminals of subtree AND clade and subtree have different resolutions then return clade
    else return NIL


fuse_two_trees (target, source)
    local_queue <- EmptyQueue
    for all subtrees stree of target tree with more than minfuse terminals do
        new_stree <- get_group (target, tree)
        if new_stree <> NIL then begin // so no more than one group is exchanged
            newtree <- a copy of target in which stree is replaced with a copy of new_stree
            add_to_queue new_tree local_queue
        end
    done
    return local_queue
```

```
fuse (trees, do_swap, obs)
    pairs = get_pairs (trees)  // returns a queue of all ordered pairs of trees in queue trees
    while pairs <> EmptyQueue do      // can be shortcut with command –fuselimit in POY
        (target, source) <- take_from_queue pairs
        fusion <- fuse_two_trees (target, source)
        if do_swap then fusion <- SwapTrees (fusion, obs) (* see 5.b or 5.c and poy commands –treefusespr and –treefusetbr (when both are on, spr and tbr are
            done one after the other, each time starting from fuse (target, source) (so having them both on makes no sense)) *)
        results <- concat_queues (results, fusion)
    done
```

Program:

```
    tree_queue <- fuse (trees, do_swap, obs)
```

9. Static approximation

Static approximation (Wheeler 2003) is a tree search strategy designed specifically for use with non-prealigned sequence data. It is basically a branch swapping algorithm that iterates between different heuristics for tree alignment cladogram cost calculations (see De Laet 2005 for background on tree alignment). It starts out with calculating an implied alignment for the tree to be swapped. Next, cost calculations during branch swapping are performed with that static multiple alignment. Each time a better tree is found, that tree is evaluated using a complete down-pass of Direct Optimization (a tree alignment heuristic developed by Wheeler (1996)) or any other more involved cost calculation heuristic. If this downpass confirms that the tree is better indeed, a new implied alignment is generated on this better tree and swapping proceeds with this new alignment. The advantage of the approach that it allows to concentrate efforts for calculating cladogram costs on 'good' candidate trees: the cost using a fixed alignment is considered a quick and dirty approximation for the real cladogram cost, and on the basis of this approximation most rearrangements can be skipped without further ado.

In poy, the command –staticapprox turns on static approximation. With static approximation on, all branch swapping for non-prealigned sequences will use this strategy (if I understand correctly). Algorithmically, this is achieved by changing the function Try_insertion_here (or its variants, like Drift_try_insertion_here). Here is an example for regular swapping (5) that uses direct optimization:

```
Static_approximation_Try_insertion_here (node_to_insert, insert_below)
    // uses alig, a set of implied alignments for all sequence characters that are present in obs; initially, alig is set (outside this function) using the first tree to be swapped
    If not (insert_below = root(tree) OR root(tree)).right = tree) then
        newtree = cp_insert_node_below (node_to_insert, insert_below)
        quick_cost = evaluate_alignments (newtree, obs, alig);   (* calculate cost of the implied alignments on the tree; for efficiency, use incremental optimization *)
         (* cf alternative 3 of (5.b) *)
        if quick_cost < global_best_cost then begin (* slop value can be applied here to pick up a wider range of good candidate trees *)
            newtree..cost = DO_evaluate (newtree, obs); (* calculate cost using the full down-pass of Direct Optimization *)
            if newtree.cost < global_best_cost then begin
                clear_queue trees_to_swap (* a slop value can be applied *)
                stop_swapping_this_tree <- true
                alig <- calculate_implied_alignments (newtree, obs)
            end;
            if newtree.cost =< global_best_cost then cond_add_to_queue newtree trees_to_swap  (* a slop value can be applied *)
            if newtree.cost < current_best_cost then current_best_cost <- newtree.cost
            if newtree.cost < global_best_cost then global_best_cost <- newtree.cost
        end
    end
```

10. An integrated approach

Tree refinement algorithms suc as in (5) – (9) are often combined with each other and with tree build algorithms into yet another level of search strategies (see Goloboff 1999). Well-known is the process of building starting trees using an algorithm as in (4), submit these trees to a refinement algorithm, and repeating this a number of times. This requires to keep the *results* queue global to all replicates or repetitions, and to have an additional global variable *best_cost_over_all_replicates*, as shown in this example with simple spr_swapping (only changes compared to 5.b shown):

```
Try_insertion_here (node_to_insert, insert_below)
    If not (insert_below = root(tree) OR root(tree)).right = tree) then
        newtree = cp_insert_node_below (node_to_insert, insert_below);
        newtree..cost = evaluate (newtree, obs);   (* for efficiency, use incremental optimization *)

        (* three alternatives follow, each one is a different search strategy - many more alternatives exist *) [how do GetSPR ad GetQuickSPR fit in here xxx]
        (* alternative 1: keep (and subsequently swap) all new trees that are at least as good as the tree being swapped; probably not a good idea *)
        if newtree.cost =< current_best_cost then cond_add_to_queue newtree trees_to_swap (* a slop value can be applied *)

        (* alternative 2: only keep (and subsequently swap) trees if they are at least as good as the best trees found globally thus far *)
        if newtree.cost < global_best_cost then clear_queue trees_to_swap (* a slop value can be applied *)
        if newtree.cost =< global_best_cost then cond_add_to_queue newtree trees_to_swap  (* a slop value can be applied *)

        (* alternative 3: as 2., but stop swapping this tree right away *)
        if newtree.cost < global_best_cost then begin
            clear_queue trees_to_swap (* a slop value can be applied *)
            stop_swapping_this_tree <- true
        end
        if newtree.cost =< global_best_cost then cond_add_to_queue newtree trees_to_swap  (* a slop value can be applied *)

        (* for all alternatives: *)
        if newtree.cost < current_best_cost then current_best_cost <- newtree.cost        // best cost obtained thus far from swapping this tree (not used in alt. 3)
        if newtree.cost < global_best_cost then global_best_cost <- newtree.cost                                    // best cost obtained thus far in this replicate
        if newtree.cost < best_cost_over_all_replicates then best_cost_over_all_replicates <- newtree.cost                  // best cost thus far over all replicates
    end
```

```
Swap_trees_spr (intrees, obs)  // modification of Swap_trees_spr in (5.b)
        trees_to_swap <- intrees
        while trees_to_swap <> EmptyQueue do
                tree_to_swap <- take_from_queue trees_to_swap
                current_best_cost <- evaluate (tree_to_swap)
                stop_swapping_this_tree <- false
                Swap_a_tree_spr (tree_to_swap)
                if tree_to_swap.cost < best_cost_over_all_replicates then clear_queue results (* a slop value can be applied here *)
                if tree_to_swap.cost <= best_cost_over_all_replicates then cond_add_to_queue tree_to_swap results (* slop value can be applied here *)
        done
        return results


Do_replicates (obs, num_replicates)
        for replicate <- 1 to num_replicates do
                trees <- Build (obs)                            // see 4
                trees <- Swap_trees_spr_this_replicate (trees, obs)             // but can can be any of 5-9, or a sequence thereof
        done
        return results
```

Program:

```
        tree_queue <- Do_replicates (obs, num_replicates)
```

11. Some quick comments on time complexity

Time complexity of tree search strategies depends in the first place on complexity of algorithms to evaluate single trees or candidate trees. This in turn depends on

1. type of analysis; e.g., Fitch (1971) versus Sankoff (1975) makes a huge difference; and on
2. cleverness in evaluating candidate trees; e.g., when adding a next taxon to a growing tree at a particular position, one can add the taxon and evaluate the resulting tree from scratch (e.g., a complete downpass of Fitch 1971), or one can add the taxon and re-use lots of calculations the were done before (see e.g. incremental optimization, Gladstein 1997).

Decisions made at this level will influence complexity of whole tree search strategies. Below I conveniently forget about that (not a good idea when you actually write code – or do analyses) and concentrate on number of insertions of terminals/subtrees into another tree have to be made; to make this clear I use O'(f(n)) instead of O(f(n)). As an example, O'(n) for a search strategy, with n number of terminals, just means that the number of insertions that that have to be tried grows in O(n). To obtain the real big O complexity of the procedure, the complexity of the evaluation of one insertion has to be substituted for n in the O' expression (and that one may be different for different insertions or at different stages (cf static approximation). O' is useful nevertheless to get some rough idea of how different search strategies compare. N stands for number of terminals.

2. Implicit enumeration for nt terminals (nt >=2)

O' is of the order of all binary trees for n taxa. As an aside, the number of rooted trees for n terminals is assymptotically equal to n!/(2sqrt(pi))*2 power n * n power (-3/2) (Hamel and Steel 1997: 360). So O' is asymptotically n! * 2 power n * n power (-3/2), which illustrates the 'more than exponential' growth as n grows.

3. Find optimal binary trees using branch and bound, for nt terminals (nt >=2)

O' depends on structure of dataset and goodness of initial length estimate.

Best case scenario: assume a dataset for n taxa with no homoplasy for which the most parsimonious tree is fully resolved and every branch is supported by 2n binary characters.
Assume that the initial tree estimate is that most parsimonious length (would be hard to miss that one). Branch and bound in this case will never branch (or all but one side branch immediately get aborted at each level). In this case, the number of insertions to try at each level (of already having i terminals) is equal to the number of branches in the tree for i terminals. So O' (n square).

Worst case scenario: assume an indecisive dataset. There no search path will be aborted an O' is same as exhaustive serach

4. Build a tree by stepwise addition (n terminals, n >= 3)

Same as best case for branch and bound.

5. Branch swapping

5.b. A tree search strategy using SPR rearrangements of given trees

O' for narrow definition of a complete spr swap of one tree is number of branches that can be pruned times reinsertion points in remaining tree. Looks like O'(n square) to me. (versus O'n for NNI). Same for broad definition.
But O' for search strategy depends on many other factors. Some of them strategy dependent (do I skip swapping this tree when I have a better tree? Do I set a maxtrees during swapping? …) other also depend on data structure (how many times is a new tree found ? –cf worst vs best case analysis in branch and bound)

5.c. A tree search strategy using TBR rearrangements of given trees

O' for a complete tbr swap of one tree is as narrow spr but with each pruned tree reinserted in all its roots; so O'(n cube).
For global O' of full strategy, see comments at spr.

6. Ratcheting

Hard to tell, depends on the other factors mentioned at spr and tbr. Obviously a linear relationship with number of iterations.

7. Tree drifting

Cf. ratchet

8. Tree fusing

Hard to tell for global strategy with swapping of new trees. For one pair of trees to fuse, straight comparison of all groups with all groups to check if groups can be exchanged is O(n).

9. Static approximation

Cf spr/tbr

10. An integrated approach

For each replicate, see above at refinement algorithm used. Beyond that, a linear relationship with number of replicates.

Note: for 5-10: see also Goloboff (1999) and Nixon (1999) for benchmarks with various empirical datasets, and Goloboff (1999) for various other combined strategies.

References

Coddington, J.A. & Scharff, N. 1994. Problems with Zero-Length Branches. Cladistics *10:* 415-423.

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 (scheduled date of publication 31 March 2005).

De Laet, J. and W. Wheeler. 2003. POY version 3.0.11 (Wheeler, Gladstein and De Laet, May 6 2003). Command line documentation. Available from the first author and at ftp://ftp.amnh.org/pub/molecular/poy/version3-current/poy.3.0.11.pdf.

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

Gladstein, D. S. 1997. Efficient incremental character optimization. Cladistics 13: 21-26.

Goloboff P. A. 1999. Analyzing large datasets in reasonable times: solutions for composite optima. Cladistics 15: 415-428.

Goloboff, P. A, and J. S. Farris. 2001. Methods for quick consensus estimation. Cladistics 17: S26-S34.

Hartigan, J. A. 1973. Minimum mutation fits to a given tree. Biometrics 29: 53-65.

Hamel, A. M., and M. A. Steel. 1997. The length of a leaf coloration on a random binary tree. SIAM J. Discr. Math. 10: 359-372.

Hendy, M. D. and D. Penny. 1982. Branch and bound algorithms to determine minimal evolutionary trees. Mathematical Biosciences 59: 277-290.

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

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

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

Sankoff, D. 1975. Minimal mutation trees of sequences. SIAM J. Appl. Math. 28: 35-42.

Wheeler, W. C. 2003. Implied alignment: a synapomorphy-based multiple-sequence alignment method and its use in cladogram search. Cladistics 19: 261-268.

Wheeler, W., Gladstein, D., and J. De Laet. 2003. POY, Phylogeny Reconstruction via Optimization of DNA and other Data. Version 3.0.11 (May 6 2003). Software and documentation freely available at ftp://ftp.amnh.org/pub/molecular/poy.