Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: Day Month Year
Application Note

OXFORD

Phylogenetics

Supplement: Two C++ Libraries for Counting Trees on a Phylogenetic Terrace

R. Biczok [1], P. Bozsoky [1], P. Eisenmann [1], J. Ernst [1], T. Ribizel [1], F. Scholz [1], A. Trefzer [1], F. Weber [1], M. Hamann [1], and A. Stamatakis [1,2]*

[1] Institute for Theoretical Informatics, Karlsruhe Institute of Technology, Karlsruhe, 76128, Germany and
[2] Scientific Computing Group, Heidelberg Institute for Theoretical Studies, Heidelberg, 69118, Germany.

*To whom correspondence should be addressed.

Associate Editor: XXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Abstract

In this supplement we provide an overview over the SUPERB algorithm. We also show that the number of rooted trees on a terrace as inferred with SUPERB is identical to the number of unrooted trees on a terrace if the unrooted input tree can consistently be rooted on a branch leading to a comprehensive taxon. In addition, we provide details on the test datasets used and discuss some noteworthy implementation details of terraphast I and terraphast II. Finally, we document the C and C++ interfaces and our compressed NEWICK format extension for writing all trees on a terrace to file.

1 SUPERB Overview

Terminology First we introduce some terminology and define what the terms we use mean. With 'SUPERB' we refer to the original algorithm by Constantinescu and Sankoff (1995) for reconstructing supertrees. We use the terms leaf nodes, leaves, and taxa synonymously. We consistently use 'partition' for subsets of MSA sites that evolve according to the same evolutionary model, whereas 'split' always refers to a split of leaf nodes (taxa) into subsets. Such splits are denoted by Σ or σ, respectively. The data presence/absence matrix is denoted by M and the comprehensive taxon for rooting by tax_C. The induced unrooted per-partition trees are denoted by $T|P_i$, their rooted counterparts by $T'|P_i$. Finally, the comprehensive input tree is denoted by T.

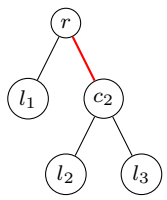
Original Superb Algorithm The original setting of the SUPERB algorithm is as follows: Given a set of rooted binary trees, construct – if possible – all rooted, binary so-called supertrees that are compatible with *all* given trees in the input tree set.

For our purposes, we use all induced per-partition trees as input tree set. These induced per-partition trees $T|P_1, ..., T|P_p$ are extracted from the given input tree T (supertree) by pruning all taxa for which no data is available for the specific partition. Therefore, we already know that the algorithm must find *at least* one such supertree. Note that, the input trees of the SUPERB algorithm must be rooted. We describe how the unrooted input trees $T|P_1, ..., T|P_p$ can be consistently rooted in Section 2.

SUPERB consists of two steps: Given the tree set as input, it first constructs a set of constraints that the supertree must fulfill/comply with to fully describe the induced per-partition trees. Then, given these constraints, SUPERB enumerates *all* binary rooted trees that *do* fulfill them.

1.1 Constraint Construction

For the constraint construction, the two children of each node in a tree $T|P_i$ are ordered (note that, the actual binary input trees are unordered leaf-labeled trees) such that there is a clearly determined left and a right child. The actual order chosen is not relevant as long as it is fixed. By $\mathrm{lca}(x, y)$ we denote the lowest common ancestor of the leaves (taxa) x and y, that is, the lowest node in the tree that is both in the path from x to the root of the tree and in the path from y to the root of the tree. Further, for an inner node x we denote by x^l/x^r the leftmost/rightmost leaf of x, that is, the leaf we reach if, starting a tree traversal at x, we always descend into the leftmost/rightmost child of a node. The constraints are of the form $\mathrm{lca}(i, j) < \mathrm{lca}(k, l)$. This form denotes that the lowest common ancestor of leafs i and j must be below the lowest common ancestor (LCA) of leafs k and l in the supertree we intend to construct. For a given rooted binary tree, it is sufficient to generate one constraint per inner edge (commonly referred to as branches in phylogenetics) (x, y), where x and y are inner nodes of the tree. This constraint for (x, y), where y is a child of x has the form $\mathrm{lca}(y^l, y^r) < \mathrm{lca}(x^l, x^r)$. Note that, depending on whether y is the left or right child of x, $y^l = x^l$ or $y^r = x^r$. Thus, due to the symmetric nature of the lca, the extracted constraints are actually of the form $\mathrm{lca}(i, j) < \mathrm{lca}(j, k)$. Figure 1 shows a simple example of this



$\mathrm{lca}(l_2, l_3) < \mathrm{lca}(l_1, l_3)$

Fig. 1. Example tree and the corresponding constraint (for the red edge/branch).

constraint construction procedure. In our example, there is only one inner edge (branch) and therefore just one constraint.

1.2 Tree Enumeration

The main part of the SUPERB algorithm recursively divides the set of taxa/leaves S of the entire tree set, respectively the input tree T together with a set of constraints C_S on these leaf nodes. The algorithm starts with all leaves/taxa. Then, for each leaf, it determines if it belongs to the left or right subtree of the root. In the recursion, the algorithm then again divides the leaves among the children of the next node. Therefore, each recursive step corresponds to one node in the supertree we intend to build. The basic insight for dividing the leaves is that for each constraint $\mathrm{lca}(i, j) < \mathrm{lca}(j, k)$ the leaves i and j must be located together in a subtree while j and k may be separated. Therefore, starting with a trivial split $\Sigma_0 = \{\{v\} | v \in \Sigma\}$ where every node is in its own part, the algorithm iteratively joins for each constraint $\mathrm{lca}(i, j) < \mathrm{lca}(j, k)$ the parts σ_a and σ_b that contain i and j, respectively. If a supertree exists, at the end of this procedure, there will be a split Σ_k with at least two parts. If there are exactly two parts, these are the leaves that are located below the two children of the current node. Otherwise, we must consider all possibilities to combine these parts such that we obtain exactly two parts. Each of these possibilities to combine parts results in a different supertree. Thus, enumerating all of them will generate all possible supertrees. For each of the two parts, we call the algorithm recursively with only the leaf nodes in the respective part *and* the constraints that only contain leaves of that specific part. If there are no constraints in one of the recursive calls, it suffices to enumerate all possible binary trees for the corresponding subset of leaf nodes of S. This can be implemented in a straight-forward way.

Let us consider an example with $S = \{1, 2, 3, 4, 5\}$ and $C_S = \{\mathrm{lca}(1, 2) < \mathrm{lca}(3, 2), \mathrm{lca}(4, 5) < \mathrm{lca}(4, 2)\}$. Then $\Sigma_0 = \{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$. In the next step, we merge 1 and 2: $\Sigma_1 = \{\{1, 2\}, \{3\}, \{4\}, \{5\}\}$. Then, we merge 4 and 5: $\Sigma_2 = \{\{1, 2\}, \{3\}, \{4, 5\}\}$. There are now 3 different ways to split these subsets into two: $(\{1, 2\}, \{3, 4, 5\})$, $(\{1, 2, 3\}, \{4, 5\})$ and $(\{1, 2, 4, 5\}, \{3\})$. So far, this yields three distinct trees. Now let us consider the recursion into all three partitions:

1. $(\{1, 2\}, \{3, 4, 5\})$: For $\{1, 2\}$, there are no constraints left and we obtain exactly one tree. For $\{3, 4, 5\}$ there are also no constraints left. We therefore need to enumerate all possible rooted binary trees with 3 leaves. Those trees all have the form of the tree in Figure 1 and there are exactly 3 of them as we have 3 possibilities for choosing l_1. Thus we obtain 3 trees from this recursion.
2. $(\{1, 2, 3\}, \{4, 5\})$: For $\{1, 2, 3\}$, there is the constraint that $\mathrm{lca}(1, 2) < \mathrm{lca}(3, 2)$. Therefore, we join 1 and 2 and obtain $(\{1, 2\}, \{3\})$ as a new split. From the next recursion we obtain exactly one result as there is only one binary tree with two leaves and 1 with one leaf (the leaf itself). For $\{4, 5\}$ there is also only one tree. Thus, we obtain exactly 1 tree from this recursive step.
3. $(\{1, 2, 4, 5\}, \{3\})$: For $\{1, 2, 4, 5\}$ we have the constraint that $\mathrm{lca}(4, 5) < \mathrm{lca}(4, 2)$. Thus, we obtain the splits $\{1\}, \{2\}, \{4, 5\}$. Again, there are three different ways of splitting these subsets into two: $(\{1, 2\}, \{4, 5\})$, $(\{1\}, \{2, 4, 5\})$, $(\{2\}, \{1, 4, 5\})$. The first yields exactly one tree from the recursion, the second one also returns only one as the constraint is being used, and the third yields three subtrees as there is no constraint left. Thus we obtain $1 + 1 + 3 = 5$ trees from this recursion.

In total, there are $3 + 1 + 5 = 9$ possible supertrees for the given set of constraints.

Next, we address the problem of how to root the unrooted input trees for executing SUPERB.

2 Rooting by comprehensive taxa

The original SUPERB algorithm is defined on rooted trees. However, all 'classic' likelihood- and parsimony-based phylogenetic inference programs and criteria return unrooted trees, except if an outgroup is specified, which however, merely constitutes a drawing option. Therefore, our library function specification explicitly requires a fully bifurcating unrooted phylogenetic tree as input. Depending on the API (Application Programming Interface) parameter settings, the output is specified to be the number of unrooted phylogenetic trees that reside on the same terrace as the input tree, potentially also including the topologies of all those trees written to a file in NEWICK format.

As the original algorithm is specified on rooted trees we need to devise a method to (i) consistently root our unrooted trees and (ii) ensure that the number of rooted trees reported by the algorithm is identical to the number of unrooted trees. This can be achieved by requiring the input dataset to contain at least one so-called comprehensive taxon tax_C, that is, a taxon that *has* data for *all* partitions of the MSA. In other words, the binary input matrix M that contains the presence/absence information about sequence data per species (rows) and partition (columns) needs to contain at least one row that entirely consists of `1`s.

If a tax_C exists, every induced tree $T|P_i$ for each partition i will contain a branch leading to tax_C. Consequently, each $T|P_i$ can be rooted on the branch/edge leading to tax_C and all induced rooted trees $T'|P_i$ can therefore be consistently rooted and provided as input to SUPERB.

Given this consistent rooting, we now need to show that the number of trees *and* actual tree topologies rooted at tax_C as returned by SUPERB are identical to the number of unrooted trees *and* unrooted tree topologies containing tax_C. Without loss of generality, we can order the leaves of the rooted induced trees passed to SUPERB as tax_C first and then all other taxa by some fixed, numerical index (e.g., a lexicographic order).

Given this ordering and rooting the $\mathrm{lca}(tax_C, y)$ will always be the root of the tree for any $T'|P_i$ and any taxon $y \in S, y \neq tax_C$ in the tree.

To prove the equivalence of rooted and unrooted supertrees, we show that the mapping π_u: $\mathcal{T}' \rightarrow \mathcal{T}$ is bijective. The function π_u maps any rooted tree to its unrooted counter-part by contracting the root. If we restrict the mapping to all supertrees that are enumerated by the SUPERB algorithm, this mapping is surjective onto the set of all unrooted supertrees of the induced subtrees $T|P_i$. This is because every unrooted supertree is also a rooted supertree of $T'|P_i$ when we place the root into the branch leading to tax_C.

However, the mapping is not necessarily injective (see Section 8.1). To ensure that it is injective, we need to restrict the SUPERB algorithm to supertrees where tax_C is a direct descendant of the root node. This can be achieved by fixing the trivial split in the first recursion step such that tax_C is placed in one set and all remaining leaves are placed in the other set. This modification of the SUPERB algorithm does not alter the results and is hence correct based on the following observations: As mentioned above, every constraint involving tax_C is of the form

$\text{lca}(i,j) < \text{lca}(j, tax_C)$. Thus, tax_C will always be placed into a singleton set when we apply the constraints. Therefore, the fixed split between tax_C and all remaining leaves constitutes a valid split in the original SUPERB algorithm. Since we ignore all other possible splits, the modified SUPERB algorithm enumerates exactly all those rooted supertrees that contain tax_C as direct descendant of the root. Note that, the above modification maintains surjectivity as every tree unrooted at tax_C can be re-rooted at tax_C.

Since we have shown injectivity and surjectivity of π_u, the set of unrooted supertrees is equivalent to the set of rooted trees returned by our modified version of SUPERB.

2.1 Relaxing the Requirement for a Comprehensive Taxon

In practice, and as was observed when integrating `terraphast-I` with `RAxML-NG` as well as in our experiments, a large proportion of empirical datasets does not contain a comprehensive taxon. In the following, we sketch how this limitation can be relaxed.

For this, we first introduce some additional definitions:

- Labelled leaf/taxon set $L = \{1, \ldots, n\}$
- Taxon subsets $\tilde{L}_1, \tilde{L}_2, \ldots, \tilde{L}_m$ representing the data that is present/available for different partitions
- Set of unrooted trees containing all leaves/taxa $\mathcal{T}_L$
- Set of rooted trees containing all leaves/taxa $\mathcal{T}_L^r$
- A split is a bipartition $(S, L \setminus S)$ of the leaf/taxon set where $S \subseteq L$

Next, we define some operations on the above data structures.

- The mapping that takes a rooted tree $t \in \mathcal{T}_L^r$ and unroots it by contracting the root node is again called $\pi_u : \mathcal{T}_L^r \to \mathcal{T}_L$
- Since every branch in an unrooted tree corresponds to a split/bipartition $(S, L \setminus S)$ (it separates two subtrees with leaf/taxon sets S and $L \setminus S$), we can use splits and induced splits to generalize the rooting at leaves to inner branches:
  The mapping that takes an unrooted tree $t \in \mathcal{T}_{\tilde{L}}$ containing the split induced by $(S, L \setminus S)$ in $\tilde{L}$ and inserts a root node in the middle of the corresponding branch is denoted by $\pi_{r(S)}$
- For a split $(S, L \setminus S)$ of the taxon set and a subset $\tilde{L} \subseteq L$, we denote by $\pi_{\tilde{L}}(S) = (S \cap \tilde{L}, \tilde{L} \setminus S)$ the induced split on $\tilde{L}$
- Constructing induced subtrees: For a given unrooted tree $t \in \mathcal{T}_L$ and a taxon set $\tilde{L} \subseteq L$, the mapping $\pi_{\tilde{L}} : \mathcal{T}_L \to \mathcal{T}_{\tilde{L}}$ constructs the induced subtree as the union of all paths between taxa in $\tilde{L}$ after contracting all nodes of degree 2. The same operations apply to rooted trees with the limitation that

  1. the root node remains uncontracted
  2. there must be a path between taxa/leaves in $\tilde{L}$ that passes through the root node, that is, the induced split on $\tilde{L}$ at the root must be non-trivial.

In the following, we sketch a proof that shows the equivalence of this adapted rooting algorithm to the one requiring a comrehensive taxon.

We know that the (unmodified) SUPERB algorithm enumerates all rooted supertrees $t_i \in \mathcal{T}_L^r$ that are compatible with the input tree $t \in \mathcal{T}_L^r$ in accordance with the missing data pattern (i.e., $\forall p = 1, \ldots, m\colon \pi_{\tilde{L}_p}(t_i) = \pi_{\tilde{L}_p}(t)$).

We now need to show that we can establish a correspondence between these rooted supertrees and the unrooted supertrees.

One ingredient of this proof is that the unrooting and construction operations of induced subtrees commute, that is, for a rooted tree $t \in \mathcal{T}_L^r$ we have $\pi_u(\pi_{\tilde{L}}(t)) = \pi_{\tilde{L}}(\pi_u(t))$. This is true because

1. the unrooting operation contracts the root node and
2. the only difference between the construction of induced subtrees for rooted and unrooted trees is that the root node remains uncontracted in the rooted case

The remainder of the proof consists of the following steps:

1. Show that every unrooted supertree has exactly one corresponding rooted supertree (this requires the existence of a comprehensive split/bipartition)
2. Show that every rooted supertree has exactly one corresponding unrooted supertree (this requires a modification of the SUPERB algorithm to only output rooted supertrees containing this comprehensive split)
3. Show that unrooting the rooted supertree obtained in step 1 returns our original input tree.

unrooted to rooted
We require that there exists a comprehensive split $(S, L \setminus S)$, that is, a split that corresponds to an edge in every supertree and every induced subtree. This means that $\pi_{L_i}(S)$ must be non-trivial for all partitions L_i. This can, for instance, be guaranteed by rooting at the edge leading to a comprehensive taxon. A more general, sufficient condition for the existence of a comprehensive taxon is the subject of ongoing research.

Then, for any unrooted supertree $t_i \in \mathcal{T}_L$ compatible with the induced subtrees of $t \in \mathcal{T}_L$, we can root at the edge that corresponds to the comprehensive split. If we compute the induced subtrees of this rooted tree, we observe that they are rooted versions of the induced subtrees of t where the root corresponds to the induced split $\pi_{L_i}(S)$. Thus, the rooted tree $\pi_{r(S)}(t_i)$ is also a (rooted) supertree of $\pi_{r(S)}(t)$. If we formulate this with our operators, we obtain:

$$\forall p = 1, \ldots, m\colon \pi_{r(S)}(\pi_{\tilde{L}_p}(t)) = \pi_{\tilde{L}_p}(\pi_{r(S)}(t_i))$$

rooted to unrooted
This is straight-forward as we obtained the rooted tree by rooting our input tree at an inner edge. The reverse operation simply returns our initial unrooted tree. However, we also need to ensure that every rooted tree has exactly one corresponding unrooted supertree. To prove this we need to proceed in two steps:

1. By the argument in the paragraph 'unrooted to rooted', the induced subtrees used as input for SUPERB are rooted versions of the induced subtrees of our input tree. Thus, if we unroot any tree returned by SUPERB, it will be compatible with the induced rooted subtrees as a rooted tree and will therefore also be compatible with the induced unrooted subtrees as an unrooted tree.
2. To prove that the mapping 'rooted to unrooted' is injective, we need to ensure that all output trees of SUPERB are rooted at the comprehensive split. This can be achieved by fixing the leaf split during the first recursion level of the SUPERB algorithm to this comprehensive split.

3 Implementation Overview

We initially discuss the parts that are common to both implementations before describing some specific implementation details of `terraphast I/II`. Algorithm 1 illustrates the unoptimized pseudo code for both libraries.

3.1 Pre-calculation and Constraint Construction

Before the tree enumerations, our implementations check via function `root_tree` if a comprehensive taxon tax_C exists. If it exists, we root the unrooted input tree T on the branch leading to taxon tax_C. We unroot

Algorithm 1: Pseudo code of **terraphast I/II**

```
Data: Comprehensive tree T, missing data matrix M
Result: Information R about terrace composition ,
        e.g., terrace size, trees in NEWICK format, …
begin
    T' ← root_tree(T, M);
    S ← extract_leaves(T, M);
    C_S ← compute_constraints(T', M);
    R ← enumerate_trees(S, C_S);
end
Function compute_constraints(T', M)
    C_S ← ∅;
    foreach Partition P_i in M do
        T'|P_i ← extract_partition_tree(T',P_i);
        C_S ← C_S ∪ extract_constraints(T'|P_i);
    end
    return Set of (unique) constraints C_S
end
Function enumerate_trees(S, C_S)
    if C_S = ∅ then
        return enumerate_binary_trees(S)
    end
    R ← init_result();
    Σ ← apply_constraints(S, C_S);
    foreach split σ = (S'_1, S'_2) in Σ do
        // Combine results R' from each split
        C'_S1 ← filter_constraints(S'_1, C_S);
        R'_1 ← enumerate_trees(S'_1, C'_S1);
        C'_S2 ← filter_constraints(S'_2, C_S);
        R'_2 ← enumerate_trees(S'_2, C'_S2);
        R' ← R'_1 ⊙ R'_2;
        // Accumulate per-split results R' to R
        R ← R ⊕ R'
    end
    return Combined result R of each split σ
end
```

and re-root the tree on tax_C if the input tree is given as rooted tree. This defines a fixed traversal order for the rooted tree T'. If more than one tax_C exists, we select the first valid tax_C that appears when reading M line by line. The SUPERB algorithm is supposed to operate on a set of leaves S extracted by the call to $extract\_leaves$. This step is, depending on the implementation, carried out implicitly by `root_tree`.

After the comprehensive input tree has been re-rooted, we extract all constraints by invoking `compute_constraints`. For each partition P_i of the missing data matrix M, we first calculate the induced per-partition tree $T'|P_i$ via a post-order traversal of T' (`extract_partition_tree`). Then, we construct the LCA constraints for each $T'|P_i$ (`extract_constraints`) and combine them into a de-duplicated list.

Duplicate constraints *can* arise when identical subtrees (see Fig. 1 for an example) are induced by more than one partition. Such duplicate constraints can be removed because they provide no additional information on the topology of the $T'|P_i$. Avoiding unnecessary constraints is another reason for de-duplicating the constraint list. Although SUPERB allows for arbitrary constraints $\mathrm{lca}(a,b) < lca(c,d)$, the constraints extracted from the $T'|P_i$ can only be of the form $\mathrm{lca}(a,b) < \mathrm{lca}(b,c)$. Note that, SUPERB does not need to distinguish between constraints $\mathrm{lca}(a,b) < \mathrm{lca}(b,c)$ and $\mathrm{lca}(b,a) < \mathrm{lca}(a,c)$ as they are applied and filtered equivalently. By generating constraints for each $T'P_i$, we may encounter a specific subtree x times and thus generate x instances of the constraint $\mathrm{lca}(l_2,l_3) < \mathrm{lca}(l_1,l_3)$.

3.2 Tree Enumeration

The function `enumerate_trees` performs the actual terrace analysis. To support various execution modes, this part of our implementations is generic. This means that, function `init_result`, function `enumerate_binary_trees`, operator $\odot$, and operator $\oplus$ are placeholders for specialized variants of `enumerate_trees`. For instance, if the user wishes to retrieve the exact terrace size, the result to be returned is an integer. In this case, function `init_result` will initialize the result variable R as arbitrary precision integer with a starting value of 0, function `enumerate_binary_trees` calculates the number of binary trees for a leaf set S if no constraint are left, and the operator $\odot$ / $\oplus$ adds/multiplies intermediate results from the recursive calls. If the user wishes to enumerate all tree topologies on the terrace, all aforementioned functions and operators will generate corresponding tree data structures instead.

Both, `terraphast I` and `terraphast II` provide the following four `enumerate_trees` execution modes:

- *Terrace detection* for checking if the given comprehensive tree T resides on a terrace or not.
- *Tree counting* for calculating the number of distinct trees that reside on the terrace.
- *NEWICK tree enumeration* for printing all trees on the terrace to file in NEWICK format.
- *Compressed NEWICK tree enumeration* for printing the compressed NEWICK tree format to file (see Section 9.1)

The `apply_constraints` function in the tree enumeration phase is important, because it accounts for a large fraction of overall runtime. Function `apply_constraints` applies a given set of constraints S_C to a set of sets of leaves generated from the given input set S. It combines, for instance, a set S_1 containing a leaf l_x and a set S_2 containing a leaf l_y, *iff*, there is a constraint satisfying $\mathrm{lca}(l_x,l_y) < \mathrm{lca}(r_x,r_y)$ for any leaf pair (r_x,r_y). Since combining sets, and checking, if a particular leaf is present within a specific set of leaves are frequently invoked operations, they should ideally require constant runtime. To this end, we experimented with two alternative data structures for this task in both implementations. One of them is the union-find with the conventional *union by rank* and *path compression* heuristics. As described by Tarjan (1975), the union-find data structure with both heuristics allows for searching and combining sets in amortized $O(\alpha(n))$ time where $\alpha(n)$ is the inverse Ackerman function. This results in an almost constant time operation as $\alpha(n) < 5$ for any relevant number $n < 10^{80}$ in our context. Alternatively, one can use bit vectors where a set bit indicates if a specific leaf node is contained in the set or not.

After applying the constraints, the algorithm iterates over all possible splits of those leaf sets into pairs of disjunct leaf sets (a specific split) denoted by Σ. For both parts S'_1 and S'_2 of a particular split, `filter_constraints` determines all constraints C'_{S1} and C'_{S2} that are still applicable. We then recursively apply `enumerate_trees` on S'_1 and S'_2 with their corresponding constraint sets C'_{S1} and C'_{S2} Finally, the $\odot$ operation combines the results from both recursive calls, while the $\oplus$ operation accumulates the results for all possible splits in a single recursive call.

4 Implementation of terraphast I

4.1 Constraint Construction

We extract the per-partition trees in a two-step process: For every partition P_i, we determine which subtrees of T' occur in the induced tree $T'|P_i$. This is calculated simultaneously for all partitions i and for every inner node v that represents/roots a subtree. More specifically, we compute the bitwise `or` of the entries in matrix M for both children of v via a post-order traversal. The result is then stored in an augmented presence/absence matrix M'.

After extracting the induced per-partition trees, we calculate the constraints as follows. First, we traverse the $T'|P_i$ in post-order to determine the outermost descendant nodes for every node v. These outermost nodes have the LCA v. We then traverse all edges/branches of all trees $T'|P_i$ once more and use the above information to establish LCA relationships. Constraint calculation is completed by calculating a de-duplicated list of LCA constraints from the above LCA relationships.

4.2 Tree Enumeration

Before the actual tree enumeration, we first use the extracted LCA constraints to construct a so-called multitree. A multitree is a single tree that complies with the constraints and represents all ambiguous (multi-furcating) nodes in the tree via a dedicated node type. Such ambiguities/multi-furcations occur when no constraints exist for a specific subset of nodes. For instance, we represent a three-taxon subtree `{a,b,c}` as a single node in the multitree. When we generate all possible supertrees based on the multitree, for every possible partial topology not containing `{a,b,c}`, the `{a,b,c}` node yields the three subtrees `(a,(b,c))`, `(b,(a,c))`, and `(c,(a,b))`. In other words, a multitree represents the set of all possible supertrees (trees on the terrace) that can be generated from the constraints via SUPERB. A supertree iterator construct (based on regular C++ Standard Template Library iterators) uses this multitree to iterate over, and generate *all* (if the option is set) such supertrees.

4.3 Data Structures

4.3.1 Bitvectors

Subsets of leaves and constraints are represented by packed bitvectors. We implemented an efficient operation to iterate over all set bits in these bitvectors by using bitmasks and the BSF (bitscan forward) instruction. As mentioned above, we calculate the node set of every $T'|P_i$ via a post-order traversal. Here, the bit vector of an inner node is the bit-wise `or` of its children. The bit vector of a leaf node i contains only one set bit at position i that corresponds to the respective taxon ID. Our bitvector implementation also provides efficient set operations like union, complement, and symmetric difference. They correspond to the bit-wise operations `or`, `xor`, and `not` which are used throughout our implementation.

In addition, the leaf bitvector is augmented by a constant-time rank support data structure, that is, the rank of an element in the set l can be computed efficiently using but a few CPU cycles. The rank operation index only needs to be updated once per recursive call.

4.3.2 Union-find

The union-find data structure uses the *union by rank* heuristic as well as *path compression* to achieve almost constant-time operations. We also implemented an explicit path compression method for the entire data structure to obtain a thread-safe `find` operation(`simple_find`). Therefore, we do not need to duplicate these data structures for the threads in the parallel version of the code.

4.4 Memory Allocation

In an early version of `terrphast I`, memory-management represented the largest performance bottleneck due to the high number of recursive calls and the frequent allocation and deallocation of the respective data structures. Thus, we developed a dedicated memory manager that leverages the LIFO (Last In First Out) structure of the memory allocations in conjunction with the predictable size of the allocations. It maintains a free-list of memory blocks and uses the worst-case (largest) size for every data structure. Note that, the RAM requirements of our implementation are generally low (see Table 2) such that this slight over-allocation of memory is justified.

4.5 Leaf-based Indexing

Since the constraints only contain leaf nodes, we remap all index values (constraints and comprehensive taxon index) from the original indexing based on their node index in the tree to leaf-based indexing. More specifically, we replace every leaf node index by its rank in the leaf set. This allowed us to reduce the space requirements for the leaf bitvectors by a factor of two, and thus further improved spatial locality and, as a consequence, runtime.

4.6 Generalized Implementation

We use an approach similar to the *Strategy pattern* where the SUPERB implementation `tree_enumerator` relies on a `Callback` object which implements several callback methods. These callback methods are used for status information (`enter`, `exit`, …), execution control (`fast_return`, `continue_iteration`), or to provide the elementary operations (`combine` for $\odot$, `accumulate` for $\oplus$) and base cases (`base_*`, `null_result`,…). Due to static polymorphism, the compiler is able to remove all potential overhead induced by empty method calls.

Analogously to the description in Section 3.2, our implementation provides four different variants of these callback objects: The `count_callback` and `clamped_count_callback` callbacks simply count the supertrees. Note that, the results are clamped in case of an integer overflow in the `clamped_` variant. The `multitree_callback` callback constructs a multitree structure that represents all trees on the terrace in a compressed format. Finally, the `check_callback` callback only checks if there are at least two trees on the terrace. This is accomplished by stopping the split iteration either when the accumulated number of trees is at least two or when a recursive call returns at least two trees (see below).

4.7 Fast Check Heuristic

When checking whether a tree resides on a terrace, we try to prune the recursions of the regular tree counting to the largest extent possible. This is achieved by stopping the iteration over all leaf splits as soon as our accumulated count is ≥ 2. In fact, it is possible to stop the recursion even earlier by using a straight-forward lower bound on the number of equivalent trees generated by a subset of the leaves. Every recursive call returns at least one tree. Thus, the number of possible leaf splits is a lower bound on the number of possible trees at every recursion level. Using this lower bound, we can stop the recursion as soon as we encounter more than two possible leaf splits after the constraints have been applied in a recursive call by invoking the `fast_return` callback.

4.8 Parallelization

The `should_resume_parallel` method decides whether the current recursive call should enumerate its different splits in parallel. If this is the

case, it prepares all parameters for the recursive calls and aggregates the results after completion of the parallel invocations.

Our initial parallelization approach was subsequently improved as follows:

- All input parameters for the parallel recursive calls were previously computed in the main thread. By deferring this work to the worker threads, we were able to reduce the overhead of the parallel implementation.
- Since the recursive calls often exhibit load imbalance, distributing the workload from two subsequent recursion levels via more sophisticated work-stealing approaches yielded improved load balance.

5 Implementation of terraphast II

5.1 Constraint Construction

The function `compute_constraints` is implemented as described in the Algorithm 1 except that it also omits tax_C and all constraints containing it. This is a valid optimization, because the first iteration of the tree enumeration phase will always generate a split between tax_C and all other leaves. The implementation is aware of the fact, that tax_C only exists implicitly, that is, it is added back after `enumerate_trees` has been completed.

5.2 Tree Enumeration

We use a conventional `for` loop to iterate over the splits σ. This is achieved by enumerating all possible splits for a specific leaf set/constraint set combination from 1 to $2^{c-1} - 1$ where c is the number of sets that is left after executing `apply_constraints`. The method `get_nth_split` computes the nth split by interpreting the number n as a bit vector of length c. Each bit i set to 1 means that leaf set c_i is supposed to be merged with the set S'_1. Otherwise, if $i = 0$, the leaf set c_i is merged with set S'_0 instead. For both parts of the split, `filter_constraints` determines all constraints that are still applicable for the respective part.

5.3 Data Structures

Benchmarks comparing the two alternative implementations suggest that the union-find data structure is on average 24.10% slower in tree counting mode than the bit vector structure. Therefore, `terraphast II` uses bitvectors by default. Users can chose to switch between union-find and bit vectors by editing the header file `leaf_set.h`.

5.4 Generalized Implementation

We implemented the four execution modi of the SUPERB algorithm by using `C++` templates and static polymorphism, similar to `terraphast I`. There exists one C++ class for each mode, where each has specialized operator implementations such as `combine_split_results` ($\oplus$ operator) or `combine_part_results` ($\odot$ operator).

- `CountAllRootedTrees` only counts the number of (sub-) trees by using the arbitrary precision integer type `mpz_class`. The implementation of `combine_split_results`, for example, only calculates the sum over the terrace sizes for individual splits. The recursion stops when no additional constraints can be fulfilled by a leaf set.
- `CheckIfTerrace` determines if T' resides on a terrace. This is the case when invoking `combine_split_results` in any step of the recursion yields more than one split. This version of the algorithm stops as soon as a recursive step identifies a terrace for a leaf subset.
- `FindAllRootedTrees` generates all trees residing on the terrace by representing each tree thereon as a corresponding binary data structure. Here, each recursion returns a dynamic array (`std::vector`) containing these binary trees.
- `FindCompressedTree` behaves analogously to version `FindAllRootedTrees`, but only maintains one dedicated data structure that represents the compressed NEWICK tree representation (see Section 9.1) of the terrace.

For each possible split of leaf nodes, the algorithm is then recursively applied to the respective subset of leaf nodes. The intermediate results (terrace detection flag, number of subtrees, or subtree structures) are then combined by `combine_split_results`. The run time contributions and return values of `combine_split_results` vary with the specified execution mode.

5.5 Parallelization

Our parallelization approach is straight-forward and fairly similar to the one used in `terraphast I` (see Section 4.8). The loop, that iterates over all splits returned by `get_nth_split`, is parallelized via the `parallel for` OpenMP pragma. This can be done, because every split represents a different set of trees. Therefore, these iterations are independent of each other. One performance problem is that, the trees from the splits assigned to a thread may be easy/fast to compute, so that this thread finishes long before the others. Hence, parallel efficiency is reduced by load imbalance. To avoid potential overhead by invoking too many threads, the OpenMP parallelization is only applied to the first recursion level at the root. Subsequently, the aforementioned `for`-loop is executed sequentially. As for `terraphast I`, parallel performance could potentially be improved by deploying work-stealing concepts, but is outside the scope of this work.

6 Performance Differences of terraphast I & II

In the following, we briefly discuss the reasons for the performance differences between the two implementations, thatis, why `terraphast I` is about a factor of two faster than `terraphast II`.

In general, it is difficult to assess the exact reasons for the performance difference, as both implementations were developed completely independently. Nonetheless, below, we list some key aspects which are highly likely to have contributed to the observed performance discrepancy. We list these aspects in decreasing order of estimated impact:

1. `terraphast I` relies on a custom memory allocator that reuses almost all memory (except for GMP integers) by leveraging the LIFO (Last In First Out) structure of memory allocations and deallocations in the recursion of SUPERB. Thus, we were able to almost entirely eliminate the overhead of the highly frequent `malloc()` and `free()` calls. In contrast to this, `terraphast II` uses the standard memory allocators that internally rely on `malloc()` and `free()`.
2. `terraphast I` was extensively optimized and almost all performance-critical functions were inlined.
3. Most data structures used by `terraphast I` in its performance-critical functions either deploy branchless (in the sense of not containing conditional statements) constant-time operations or rely on dedicated compiler intrinsics (e.g., `BSF/TZCNT` for iterating over the set bits in a bitvector).

Table 1. Empirical test datasets used

name	# partitions	# taxa	subsampled yes/no	terrace size
Allium	4	57	yes	8038035
Allium_Reduced	4	30	yes	730680125
Allium_Tiny	3	6	yes	35
Asplenium.1	3	132	yes	1
Asplenium.2	2	133	yes	95
Bouchenak	3	298	no	61261515
Burleigh.birds.small	29	627	yes	4.12×10^{50}
Caryophyllaceae	6	224	yes	7.18×10^{11}
Eucalyptus.1	4	136	yes	229
Eucalyptus.2	4	136	yes	267
Euphorbia.1	6	131	yes	759
Euphorbia.2	5	131	yes	759
Ficus.1	4	110	yes	283815
Ficus.2	4	108	yes	851445
Ficus.3	4	110	yes	851445
Iris	4	137	yes	1
Meredith.mammals	26	169	yes	1
Meusemann	97	117	no	1
Miadlikowska.fungi	9	1317	yes	11655
Misof.insects	479	144	yes	1
Primula	5	185	yes	2835
Pyron	5	767	no	2205
Rabosky.scincids	6	213	yes	3
Ranunculus	6	170	yes	3
Rhododendron	4	117	yes	81
Rosaceae	7	529	yes	1.72×10^{23}
Shi.bats	25	797	yes	2.42×10^{35}
Solanum	6	187	yes	211865625
Springer.primates	77	372	yes	70840575
Szygium.1	3	106	yes	45
Szygium.2	3	106	yes	45
Tolley.chameleons	6	202	yes	1
Wick.1kp.few.genes	8	102	yes	1
Wick.1kp.many.genes	619	102	yes	1
Yang.caryo.1122	1115	95	yes	1
Yang.caryo.209	209	95	yes	1

Table 2. RAM consumption (MB) in tree counting mode

Dataset	terraphy	terraphast I	terraphast II
Allium	20.7	3.83	4.36
Allium_Reduced	20.3	3.80	4.27
Allium_Tiny	19.7	3.87	4.22
Asplenium.1	23.3	3.96	4.42
Asplenium.2	22.8	3.89	4.34
Bouchenak	28.5	4.01	4.71
Burleigh.birds.small	50.3	4.63	5.77
Caryophyllaceae	28.2	4.00	4.56
Eucalyptus.1	23.7	3.90	4.41
Eucalyptus.2	23.8	3.94	4.78
Euphorbia.1	24.2	3.93	4.27
Euphorbia.2	24.3	3.89	4.81
Ficus.1	23.4	3.89	4.36
Ficus.2	23.3	3.88	8.37
Ficus.3	23.3	3.91	4.38
Iris	23.7	3.96	4.43
Meredith.mammals	61.7	3.99	4.44
Meusemann	99.0	4.18	4.50
Miadlikowska.fungi	76.4	4.68	7.52
Misof.insects	680.1	8.24	4.47
Primula	26.0	3.98	4.46
Pyron	43.7	3.98	5.03
Rabosky.scincids	30.2	3.96	4.53
Ranunculus	25.2	3.95	4.40
Rhododendron	23.4	3.94	4.29
Rosaceae	36.4	4.14	6.11
Shi.bats	59.3	4.75	5.54
Solanum	25.8	4.00	4.45
Springer.primates	130.4	5.36	4.89
Szygium.1	22.8	3.91	4.35
Szygium.2	22.8	3.89	4.41
Tolley.chameleons	31.6	4.02	4.54
Wick.1kp.few.genes	28.0	3.94	4.24
Wick.1kp.many.genes	610.4	7.97	4.54
Yang.caryo.1122	1001.9	12.36	4.70
Yang.caryo.209	210.9	5.21	4.39

7 Test Datasets

For testing, we used all empirical test datasets provided at `https://github.com/BDobrin/data.sets`. The repository contains several recently published partitioned phylogenomic datasets with missing data. As already mentioned in the main text, some of these datasets did not contain a comprehensive taxon tax_C. To this end, we sub-sampled the datasets by applying the following procedure: First, we determined the number of partitions every taxon contains data for. By selecting the taxa with data for the largest number of partitions, we determined candidate taxa which are comprehensive for a large subset of the partitions. We then generated the subsampled data sets by using the `cut` utility for pruning partitions.

Overall, we generated 36 test datasets from the 26 empirical datasets which are described in Table 1.

In addition, we used some simple artificial datasets for initial testing and verification.

For instance, the following M matrix exhibits a structure where the terrace comprises all possible trees with 5 taxa:

```
taxon1 1 0 //taxon1 has data for partition 0 only
taxon2 1 0
taxon3 1 1
taxon4 0 1
taxon5 0 1
```

The following M matrix does not exhibit any terraces as there is no missing data:

```
taxon1 1 1
taxon2 1 1
taxon3 1 1
taxon4 1 1
taxon5 1 1
```

8 Additional Experimental Results

In Table 2 we show the memory consumption of `terraphy` and `terraphast I/II` for all test datasets in tree counting mode. Note that, the RAM consumption of `terraphast I/II` is one to two orders of magnitude lower than that of `terraphy`. The larger variance of the RAM consumption in `terraphast I` is due to a memory-wise not fully optimized data structure for storing the induced per-partition subtrees $T|P_i$. Therefore, this slight waste of RAM becomes more apparent on datasets with a larger number of partitions.

Table 3. Run times in seconds for terrace detection, counting, and enumeration modes.

	detection		counting			enumeration		
Dataset	terraphast I	terraphast II	terraphy	terraphast I	terraphast II	terraphy	terraphast I	terraphast II
Allium	0.00040	0.0041	0.50	0.0088	0.0159	–	–	–
Allium_Reduced	0.00024	0.0037	54.53	1.7701	3.4969	–	–	–
Allium_Tiny	0.00014	0.0038	0.34	0.0030	0.0034	0.34	0.0004	0.0038
Asplenium.1	0.00063	0.0045	0.47	0.0030	0.0056	0.49	0.0010	0.0059
Asplenium.2	0.00053	0.0056	0.44	0.0031	0.0079	0.84	0.0047	0.0114
Bouchenak	0.00099	0.0100	0.86	0.0045	0.0189	–	–	–
Burleigh.birds.small	0.00281	0.0322	4099.76	147.74	301.0850	–	–	–
Caryophyllaceae	0.00059	0.0076	0.92	0.0096	0.0271	–	–	–
Eucalyptus.1	0.00037	0.0053	0.66	0.0062	0.0225	1.70	0.0115	0.0273
Eucalyptus.2	0.00034	0.0053	0.65	0.0061	0.0210	1.81	0.0155	0.0372
Euphorbia.1	0.00043	0.0062	0.51	0.0025	0.0068	2.47	0.0264	0.0286
Euphorbia.2	0.00034	0.0050	0.63	0.0030	0.0159	2.92	0.0340	0.0395
Ficus.1	0.00035	0.0054	0.51	0.0020	0.0076	–	–	–
Ficus.2	0.00032	0.0052	0.47	0.0020	0.0072	–	–	–
Ficus.3	0.00028	0.0026	0.65	0.0058	0.0159	–	–	–
Iris	0.00044	0.0059	0.52	0.0021	0.0058	0.51	0.0008	0.0064
Meredith.mammals	0.00134	0.0108	2.13	0.0030	0.0133	2.16	0.0021	0.0132
Meusemann	0.00249	0.0184	3.85	0.0040	0.0199	3.86	0.0037	0.0223
Miadlikowska.fungi	0.00690	0.1289	7.82	0.0155	0.1406	–	–	–
Misof.insects	0.01560	0.1054	28.53	0.0191	0.0861	28.58	0.0259	0.1033
Primula	0.00083	0.0082	0.67	0.0022	0.0088	11.25	0.1253	0.1254
Pyron	0.00323	0.0223	2.33	0.0042	0.0253	32.59	0.3375	0.2832
Rabosky.scincids	0.00090	0.0096	0.88	0.0027	0.0091	0.91	0.0020	0.0070
Ranunculus	0.00061	0.0073	0.61	0.0033	0.0058	0.65	0.0013	0.0076
Rhododendron	0.00048	0.0060	0.49	0.0026	0.0049	0.72	0.0034	0.0083
Rosaceae	0.00136	0.0122	2.32	0.0334	0.0867	–	–	–
Shi.bats	0.00402	0.0470	6.34	0.0146	0.0810	–	–	–
Solanum	0.00057	0.0068	0.77	0.0050	0.0174	–	–	–
Springer.primates	0.00476	0.0489	8.53	0.0072	0.0489	–	–	–
Szygium.1	0.00038	0.0056	0.43	0.0020	0.0058	0.56	0.0020	0.0059
Szygium.2	0.00048	0.0042	0.47	0.0020	0.0059	0.57	0.0017	0.0074
Tolley.chameleons	0.00123	0.0091	0.92	0.0025	0.0081	0.97	0.0012	0.0105
Wick.1kp.few.genes	0.00068	0.0064	0.69	0.0020	0.0064	0.67	0.0007	0.0060
Wick.1kp.many.genes	0.01708	0.0981	26.28	0.0222	0.0751	26.34	0.0150	0.0749
Yang.caryo.1122	0.02313	0.1154	43.36	0.0369	0.1282	43.37	0.0324	0.1118
Yang.caryo.209	0.00464	0.0283	7.90	0.0092	0.0255	7.93	0.0079	0.0299

In Table 3 we show run-times for all datasets in terrace detection, tree counting, and tree enumeration modes for `terraphy` and `terraphast I/II`. Note that, `terraphy` does not offer a terrace detection mode. The results for the tree enumeration mode are incomplete due to excessive run-times.

8.1 Differences between terraphy and our implementations

While conducting our experiments, we noticed that for the Allium_Tiny dataset, `terraphy` enumerated 37 rooted trees, while there are only 35 unrooted trees on the terrace. This difference stems from the rooting, as in these initial tests the `terraphy` input tree was *not* rooted at a comprehensive taxon tax_C. When we re-rooted the `terraphy` input tree at a comprehensive taxon, `terraphy` also enumerated 35 rooted trees which correspond to our 35 unrooted trees due to the consistent rooting.

To further elucidate this, consider the following small example:

```
((((s3,s5),s2),s1),(s6,s4));
((((s3,s5),s2),(s6,s4)),s1);
(((s3,s5),s2),(s1,(s6,s4)));
```

The three trees above become topologically identical if they are consistently re-rooted at taxon $s1$.

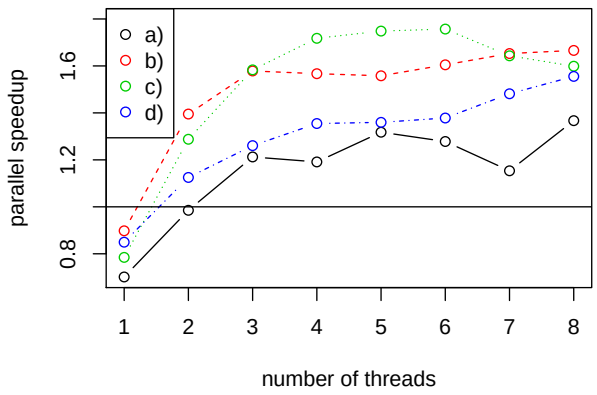


Fig. 2. Parallel speedup in tree counting mode for (a) Allium_Reduced with GMP, (b) Burleigh.birds.small with GMP, (c) Allium_Reduced without GMP and (d) Burleigh.birds.small without GMP

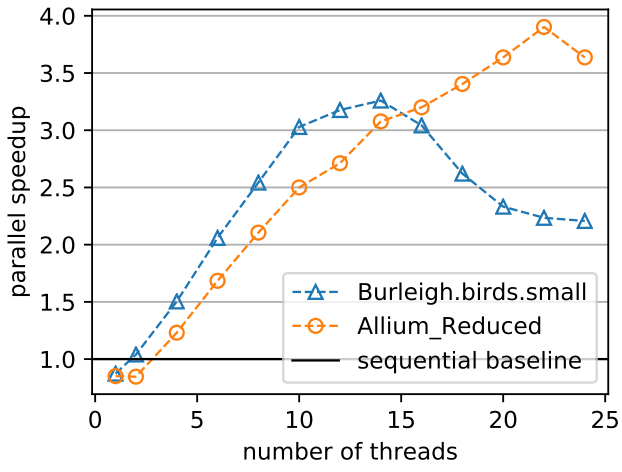


Fig. 3. Improved parallel speedup in tree counting mode for (a) Allium_Reduced with GMP, (b) Burleigh.birds.small with GMP using a system with 16 physical cores and jemalloc

Parallel performance Finally, in Figure 2 we provide the initial parallel speedup of `terraphast I` for up to 4 physical cores and up to 8 threads (using hyper-threading) on the reference test system.

The highly frequent memory allocations and deallocations in the algorithm constitute a potential parallel performance bottleneck. To this end, we deployed the dedicated lockless parallel memory allocator `jemalloc` (see `http://jemalloc.net/`) which yielded up to 25% run time improvement for the GMP-based version that executes a higher number of memory allocations to implement arbitrary precision integers.

In addition, we also assessed if thread pinning, that is, specific thread-to-core assignments, have a notable impact on performance. This is because it is known that thread pinning can substantially affect parallel efficiency on distributed shared memory systems (Klug *et al.*, 2011). In Figure 2 we show speedups for the respective optimal pinning, albeit distinct pinnings did not exhibit a substantial performance impact.

As mentioned before, parallel efficiency could be further improved via appropriate load balancing and work stealing concepts.

Following the initial submission of this paper, we took some steps toward improving parallel performance using the OpenMP `task` mechanism. On an Intel Xeon E5-2670 (Sandy Bridge) running at 2.6 GHz with 16 physical cores and equipped with 64 GB RAM, we attained speedups of up to a factor of 4 as shown in Figure 3.

9 C and C++ Interfaces

In the following we briefly present the C and C++ interfaces.

C interface

```
int terraceAnalysis(
    missingData *m,
    const char *newickTreeString,
    const int ta_outspec,
    FILE *allTreesOnTerrace,
    mpz_t terraceSize
    );
```

Here `m` represents the binary data input matrix M that also contains a list of taxon/species names for each row. `newickTreeString` is the tree string of the comprehensive tree T in NEWICK format that is passed from the application program to the library. The library will then internally compute the induced per-partition trees $T'|P_i$. `ta_outspec` specifies the desired output (execution mode), that is, if the function shall only determine whether the tree is on a terrace or not, if it shall return the number of trees on the potential terrace, or if it shall also enumerate and print to file (in compressed/uncompressed format), all trees on the respective terrace. `allTreesOnTerrace` is a file pointer for printing all trees on the terrace. Finally, `terraceSize` is used to store the number of trees on the terrace, where `mpz_t` is the respective GNU multi-precision library integer type.

The function returns 0 in case of success and a negative error code to handle errors (e.g., no $taxa_C$ could be found, problem parsing NEWICK tree string, mismatch between taxon names in NEWICK tree and `m` etc.).

C++ interface The C++-interface consists of four families of functions:

```
bool is_on_terrace(nwk, matrix)
std::uint64_t get_terrace_size(nwk, matrix)
mpz_class get_terrace_size_bigint(nwk, matrix)
mpz_class print_terrace(nwk, matrix, out)
```

The argument-types of `nwk` and `matrix` are `const std::string&` and `std::istream&` (four overloads are provided with all possible combinations). The `out`-argument of the `print_terrace`-function has the type `std::ostream&`. Finally, there is a `*_from_file`-variant of every function-family that takes two filenames as `const std::string&` and reads its data from those files while the other arguments remain unchanged.

If the terrace size exceeds the maximum integer value that can be represented by `std::uint64_t`, the `get_terrace_size` family of functions will simply return the maximum integer value $(2^{64} - 1)$. If the exact terrace size is required nonetheless, the `_bigint`-variants of the functions can be deployed to obtain it. Errors are handled via exceptions.

9.1 Compressed NEWICK representation of a terrace

As enumerating and printing all trees on a terrace to file can easily dominate run-times and require large amounts of disk space, we propose a compressed NEWICK representation that defines all trees on the terrace, but requires substantially less disk space. In Table 4 we provide the compression ratios for output tree files on three representative

Table 4. Size of enumerated NEWICK trees in bytes.

Dataset	Uncompressed	Compressed	Space Saving
Allium_Tiny	4,096	1,103	73.071 %
Primula	10,653,930	5,551	99.946 %
Pyron	37,612,890	20,407	99.945 %

empirical test datasets. The compressed representation is an extension of the NEWICK format. It relies on the following two extension: First, we use curly brackets to identify a subset of taxa that has no applicable constraint left. For instance, we write `{s1,s2,s3}` instead of `(s1,(s2,s3));` `(s2,(s1,s2));` `(s3,(s1,s2));` to denote all (rooted) binary trees for these 3 taxa. Second, we use the `|` symbol to list all subtrees that can be inserted at a specific position in the tree. The expression `((a,(b,c)),(d,(e,f))|(e,(d,f)))`, for example, is a compressed representation of the two alternative trees `((a,(b,c)),(d,(e,f));` and `((a,(b,c)),(e,(d,f)));`. This compressed NEWICK extension could be used, for instance, by tools for post-processing terraces.

Acknowledgements

Part of this work was financially supported by the Klaus Tschira Foundation and the DFG grant WA 654/22-2. We thank Olga Chernomor, Bui Quang Minh, and Derrick Zwickl for discussions on the interface definition, Barbara Dobrin for access to her empirical dataset repository, Alexey Kozlov for integration with RAxML-NG, and support by the state of Baden-Württemberg through bwHPC.

References

Constantinescu, M. and Sankoff, D. (1995). An efficient algorithm for supertrees. *Journal of Classification*, **12**(1), 101–112.

Klug, T. *et al.* (2011). *autopin – Automated Optimization of Thread-to-Core Pinning on Multicore Systems*, pages 219–235. Springer Berlin Heidelberg, Berlin, Heidelberg.

Tarjan, R. E. (1975). Efficiency of a good but not linear set union algorithm. *J. ACM*, **22**(2), 215–225.