

A peer-reviewed version of this preprint was published in PeerJ on 1 March 2017.

[View the peer-reviewed version](https://peerj.com/articles/3058) (peerj.com/articles/3058), which is the preferred citable publication unless you specifically need to cite this preprint.

Redelings BD, Holder MT. (2017) A supertree pipeline for summarizing phylogenetic and taxonomic information for millions of species. PeerJ 5:e3058 <https://doi.org/10.7717/peerj.3058>

A supertree pipeline for summarizing phylogenetic and taxonomic information for millions of species

Benjamin D Redelings^{1,2}, Mark T Holder^{Corresp. 2,3,4}

¹ Department of Biology, Duke University, Durham, NC, United States

² Department of Ecology and Evolutionary Biology, University of Kansas, Lawrence, KS, United States

³ Biodiversity Institute, University of Kansas, Lawrence, KS, United States

⁴ Heidelberg Institute for Theoretical Studies, Heidelberg, Germany

Corresponding Author: Mark T Holder

Email address: mtholder@ku.edu

We present a new supertree method that enables rapid estimation of a summary tree on the scale of millions of leaves. This supertree method summarizes a collection of input phylogenies and an input taxonomy. We introduce formal goals and criteria for such a supertree to satisfy in order to transparently and justifiably represent the input trees. In addition to producing a supertree, our method computes annotations that describe which grouping in the input trees support and conflict with each group in the supertree.

We compare our supertree construction method to a previously published supertree construction method by assessing their performance on input trees used to construct the Open Tree of Life version 4, and find that our method increases the number of displayed input splits from 35,518 to 39,639 and decreases the number of conflicting input splits from 2,760 to 1,357. The new supertree method also improves on the previous supertree construction method in that it produces no unsupported branches and avoids unnecessary polytomies.

This pipeline is currently used by the Open Tree of Life project to produce all of the versions of project's "synthetic tree" starting at version 5. This software pipeline is called "*propinquity*". It relies heavily on "*otcetera*" - a set of C++ tools to perform most of the steps of the pipeline. All of the components are free software and are available on GitHub.

A supertree pipeline for summarizing phylogenetic and taxonomic information for millions of species

Benjamin D. Redelings^{1,2} and Mark T. Holder^{2,3,4}

¹Department of Biology, Duke University, Durham NC, US

²Department of Ecology and Evolutionary Biology, University of Kansas, Lawrence KS, US

³Biodiversity Institute, University of Kansas, Lawrence KS, US

⁴Heidelberg Institute for Theoretical Studies, Heidelberg, Germany

Corresponding author:

Mark T. Holder^{2,3,4}

Email address: mtholder@ku.edu

ABSTRACT

We present a new supertree method that enables rapid estimation of a summary tree on the scale of millions of leaves. This supertree method summarizes a collection of input phylogenies and an input taxonomy. We introduce formal goals and criteria for such a supertree to satisfy in order to transparently and justifiably represent the input trees. In addition to producing a supertree, our method computes annotations that describe which grouping in the input trees support and conflict with each group in the supertree.

We compare our supertree construction method to a previously published supertree construction method by assessing their performance on input trees used to construct the Open Tree of Life version 4, and find that our method increases the number of displayed input splits from 35,518 to 39,639 and decreases the number of conflicting input splits from 2,760 to 1,357. The new supertree method also improves on the previous supertree construction method in that it produces no unsupported branches and avoids unnecessary polytomies.

This pipeline is currently used by the Open Tree of Life project to produce all of the versions of project's "synthetic tree" starting at version 5. This software pipeline is called "*propinquity*". It relies heavily on "*otcetera*" - a set of C++ tools to perform most of the steps of the pipeline. All of the components are free software and are available on GitHub.

1 BACKGROUND

The Open Tree of Life project seeks to build a platform for summarizing what is known about phylogenetic relationships across all of Life (Hinchliff et al., 2015). One primary goal of the project is to build a summary tree from a comprehensive taxonomic tree and a set of published trees. The summary tree is intended to transparently and justifiably represent phylogenetic information from these inputs. The taxonomic tree is derived from the Open Tree Taxonomy (OTT hereafter, publication in preparation). The phylogenetic inputs are published trees that have been curated to align the tips to OTT and to identify the correct rooting (see McTavish et al. 2015 for further details of the curation tools). Unlike OTT, these phylogenetic trees do not include all leaf taxa. The inputs (taxonomy and phylogenetic trees)

38 and the output summary supertree are all rooted. Here we describe the software pipeline
39 (*propinquity*) that summarizes and integrates these smaller source trees and the taxonomy
40 tree into a single supertree and the noteworthy tools for manipulating and solving supertrees
41 in the *otcetera* package.

42 1.1 Goals

43 Translating the goals of the Open Tree of Life's summary tree into an explicit set of criteria
44 is not trivial. The summary supertree should represent the phylogenetic information from
45 source trees in a transparent and justifiable fashion. We would like to allow users to correct
46 errors in the supertree by improving the input information rather than requiring modification
47 to the supertree algorithm. The pipeline was designed to create a tree which:

- 48 1. displays no unsupported groups,
- 49 2. defers to groupings from higher ranked trees in the case of conflict,
- 50 3. contains no unnecessary polytomies, and
- 51 4. displays as many groupings from input trees as possible.

52 These goals are described more fully below. In order to accomplish transparency and
53 justification, our pipeline also produces annotations files with information about conflict and
54 support.

55 1.1.1 Goal 1: Each grouping is supported by at least one input

56 We require that each edge in the supertree be supported by at least one input tree edge.
57 In addition to aiding interpretability, this requirement keeps the supertree from arbitrarily
58 representing information that comes from none of the input trees. Of course, in a supertree
59 analysis, the full tree will imply some relationships for subsets of the taxa that are not found
60 in any input tree. So, the meaning of "supported by" needs some clarification.

61 **Notation, terminology, and the definition of "supported by"** Let \mathbb{S} denote a supertree, and
62 T_i denote the i th input tree. The set of taxa that are mapped to the tips of the tree T_i is
63 $\mathcal{L}(i)$. $\mathbb{S}(i)$ denotes the summary tree induced by tip nodes that are mapped to taxa in $\mathcal{L}(i)$
64 and the most recent common ancestor of those leaves, and any other node that is an ancestor
65 of some but not all of these leaves. We say that edge j of the supertree is compatible with an
66 input tree, T_i if edge j either is not included in the induced tree $\mathbb{S}(i)$ or none of the edges in
67 T_i are in conflict with edge j in the induced tree.

68 We can consider whether or not a node in an input tree is displayed by \mathbb{S} . For any such
69 node j there is a set of taxa that are mapped to the tips that descend from the node. This
70 set of taxa is the "cluster" of taxa corresponding to node j ; it can be denoted $\mathcal{L}(i, j)$. It will
71 also be referred to as the "include set" of the node. The "exclude set" of j in T_i is the set of
72 taxa in $\mathcal{L}(i)$ but not in $\mathcal{L}(i, j)$. If the cluster of taxa for any supertree node in $\mathbb{S}(i)$ is identical
73 to $\mathcal{L}(i, j)$, then we say that \mathbb{S} displays node j of T_i . We say that the summary tree *displays*
74 edge j if the summary tree displays the child node of edge j . Operationally, we can find the
75 most recent common ancestor (MRCA) node of $\mathcal{L}(i, j)$ in \mathbb{S} ; the summary tree displays j
76 if and only if that MRCA node is not an ancestor of any member of the exclude set of j .

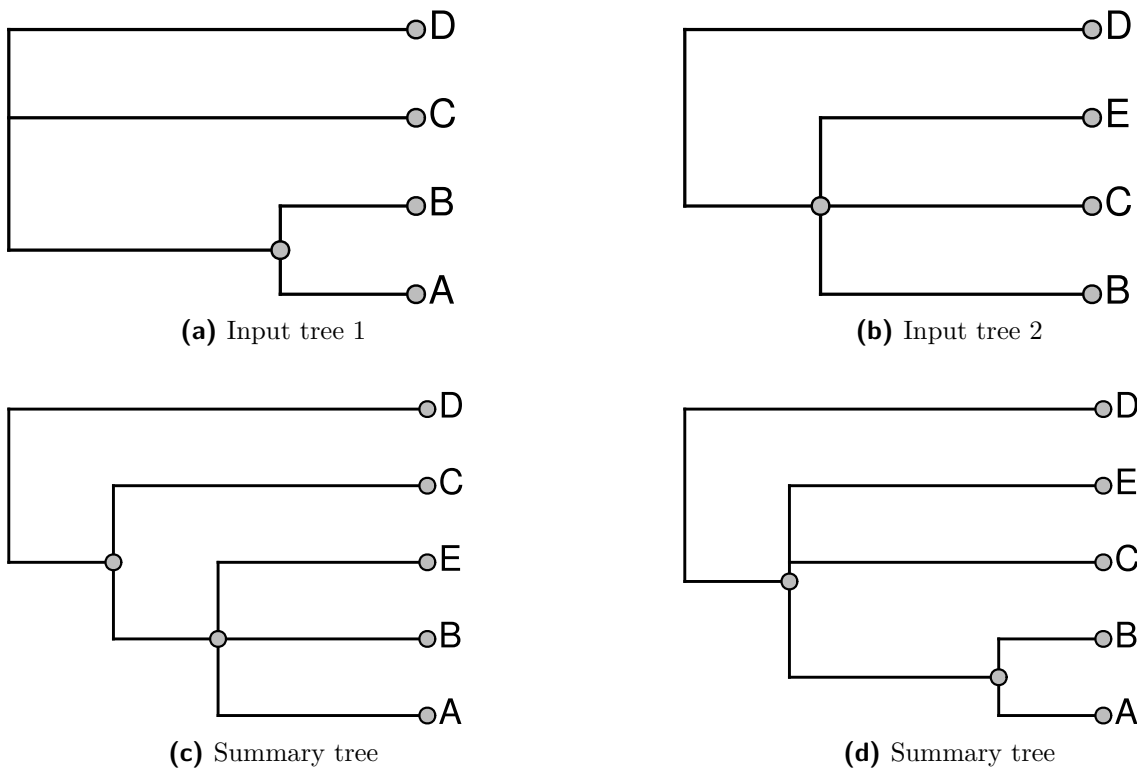


Figure 1. An example demonstrating that our definition of “supported by” does not imply entire composition of a grouping. (a) and (b) show 2 input trees and (c) and (d) depict trees that each display each of the groupings in the input trees and which have no unsupported nodes. The BUILD algorithm (section 8) would choose tree (d) that floats taxon E closer to the root.

77 We say that node k of the summary tree is *supported by* node j in T_i if the summary tree
 78 displays node j , but if we contracted edge that separates k from its parent then the modified
 79 summary tree would no longer display node j .

80 Note that stating that a node in the summary tree is supported by an input does not
 81 imply that every descendant of that node must be present in the input nor that every taxon
 82 that is not a descendant must be excluded in order to display the node. Consider the problem
 83 shown in figure 1; panels (1a) and (1b) show two input trees. Because taxa A and E do
 84 not occur together in either input, there is some uncertainty about where to place them.
 85 By our terminology, either output shown in (1c) or (1d) would be characterized as a tree
 86 that displays all of the input groupings and which has no unsupported groups. Clearly these
 87 criteria are insufficient to specify a unique solution, and users of the output tree need to
 88 be aware that it may be possible for some taxa to “float” to multiple positions. In figure 1,
 89 taxon E floats to different positions in (1c) and (1d), whereas taxon A does not.

90 One of our aims in supertree construction is to minimize the amount of information in
 91 the supertree that does not come from input trees. We permit information that comes from
 92 combinations of input trees, but not any single input tree. However, we seek to exclude
 93 information that comes from none of the input trees. This motivates the criterion of not

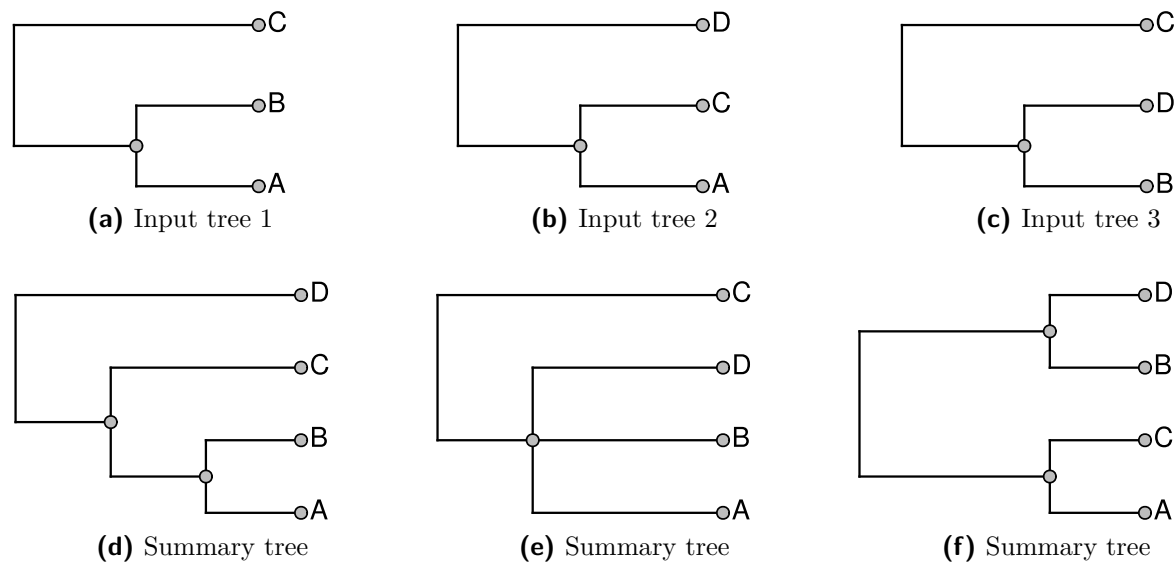


Figure 2. An example of 3 input trees shown in (a), (b), and (c) which do not conflict in a pairwise manner, but cannot be jointly displayed in one tree. The 3 solution trees are shown in panels (d-f). Panel (d) for ranking the tree in (c) lowest. Panel (e) shows the solution if the tree in (b) has the lowest rank. Panel (f) shows the solution if the tree in (a) is ranked lowest. Each of the solutions displays 2 of the 3 input groupings.

94 having any unsupported edges, since these edges could be removed without decreasing the
 95 support from any input tree.

96 1.1.2 Goal 2: Tree ranking

97 An appealing goal for the summarization would be to find the supertree that displays the
 98 largest number of input tree edges. As discussed in Huson et al. (pages 92 and 131; 2010) the
 99 maximum compatibility problem is known to be *NP*-hard via a reduction to Max-Clique
 100 (Karp, 1972). In addition to being computationally daunting, this formulation of the supertree
 101 problem does not provide biologists who use the summarization tool with an obvious avenue
 102 for fixing perceived problems with the summary tree. For example, a grouping that a biologist
 103 expected may not be present in the supertree, but it may not conflict with any of the input
 104 groupings which are displayed. This can happen because displaying both node *a* from T_1 and
 105 node *b* from tree T_2 in a summary tree may only be possible by displaying a grouping that is
 106 present in no input tree. All other factors being equal, if this implied grouping conflicts with
 107 input node *c* in tree T_3 , then *c* will not be displayed in the summary tree, but a biologist will
 108 not necessarily know how to fix this problem. One solution is to use a ranking of groupings. If
 109 an expert were quite confident in the *c* grouping, then she could assign that input node a high
 110 ranking. A supertree that used ranks could then recover this grouping even if its inclusion
 111 did not increase the total number of input nodes that are displayed by the summary tree.
 112 Figure (2) shows an example of 3 input trees for which there is no pairwise incompatibility,
 113 but no solution displays all of the input groups. Alternative rankings of inputs can result in
 114 one of three summary trees shown in panels (d-f).

115 Any approach to supertree construction must deal with the need to adjudicate between
116 conflicting input trees. We choose to deal with conflict by ranking the input trees, and
117 preferring to include edges from higher-ranked trees. The merits of using tree ranking are
118 questionable because the system does not mediate conflicts based on the relative amount of
119 evidence for each alternative. However, it is a reasonable starting point. It has the benefits
120 of making it easy to see why some groups are included or not (transparency), and it allows
121 simpler and cleaner algorithms.

122 Note that if some edge c conflicts with a higher-ranked edge b , then c may still be included
123 in the supertree. This can occur when the higher ranked edge b conflicts with a yet-higher
124 ranked edge a , and thus b is not included. In that case, it will be possible for c to be
125 represented in the summary tree. Thus, the fact that the summary tree displays an input
126 edge does not imply that none of the higher ranked input trees conflict with that edge.

127 In order to produce a comprehensive supertree, we also require a rooted taxonomy tree in
128 addition to the ranked list of rooted input trees. Unlike other input trees, the taxonomy tree
129 is required to contain all taxa, and thus has the maximal leaf set. We make the taxonomy
130 tree the lowest ranked tree. In our current formulation, the taxonomy tree is also unique
131 in that the taxonomy is the only source of taxonomic names. Each node in the taxonomy
132 tree corresponds to a named group. Taxonomic groups may have the same name, but each
133 node in the taxonomy tree is identified by a unique number (its OTT ID). Taxonomic groups
134 are identified in the summary supertree by finding a branch (or “node”) that has exactly
135 the same include|exclude relationship. The taxonomy supertree can meaningfully possess
136 degree-two nodes. Although these nodes can be removed without affecting the relationships
137 of the leaves, they do represent nested taxonomic groups that contain exactly one subgroup.
138 The taxonomy is also used to determine which tips are terminal taxa.

139 **1.1.3 Goal 3: Contain no unnecessary polytomies**

140 The supertree should be as resolved as possible - in other words, it should have no unnecessary
141 polytomies. Thus, for each input edge that is *not* included, we can point to a reason for
142 non-inclusion by showing that the input edge conflicts with some edge of the summary tree.
143 Note, that the requirement to not display unsupported groups leads to some “necessary”
144 polytomies. For example any resolution of the polytomy shown in figure (2e) would continue to
145 display the same two input groups. However, the additional grouping would be unsupported,
146 because the unresolved tree already displays both input groups. Thus, the unresolved tree
147 would be preferred by our criterion. However, collapsing either internal edge of the tree
148 shown in figure (2d) would result in a tree which displays only one input grouping. This tree
149 would contain an unnecessary polytomy, because the polytomy would permit refinement to
150 the depicted tree which displays more input groupings.

151 **1.1.4 Goal 4: Display as many input nodes as feasible**

152 We also seek to construct a supertree that represents as many input tree nodes as possible.
153 Since non-included input tree nodes must conflict with the supertree (or they would have
154 been added), this criterion is the same as minimizing the number of input nodes that conflict
155 with the supertree.

156 **1.1.5 Summary of goals**

157 These optimality criteria help to define what it means for the supertree to represent the
158 input trees, as well as justifying and explaining why various features of the supertree exist.
159 The pipeline described below produces a supertree that satisfies the first three optimality
160 criteria and is a greedy approximation of a solution to the fourth goal. It is not guaranteed
161 to display as many input nodes as possible. Even if the summary tree does accomplish goal
162 4, it is not necessarily a *unique* optimum. The pipeline takes a greedy approach to producing
163 a summary tree by attempting to add groupings from the trees in order of the trees ranking.
164 This can be viewed as a greedy solution to the problem of finding the tree with the maximum
165 sum of displayed groups' weighted scores criterion (MSDGWS, described in the appendix
166 A) where the weights from the trees are so extreme that displaying one group from a highly
167 ranked tree is preferred to displaying all of the groupings from lower ranked trees.

168 **2 DESCRIPTION OF THE SUPERTREE METHOD**

169 **2.1 Preprocessing steps**

170 Propinquity was designed to function as a part of the Open Tree of Life software architecture,
171 so the first few steps of the pipeline involve transforming artifacts from that project into a set
172 of rooted trees and a phylogenetic taxonomy. The phylesystem API (McTavish et al., 2015)
173 of Open Tree allows users to curate published estimates of trees and create ranked collections
174 of these trees. Early steps in the propinquity pipeline manipulate the phylogenetic input
175 trees to improve their usability and reliability. The first steps of the pipeline (see Figure
176 3) collect a list of trees to include (in the `phylo_input` subdirectory) and store copies of
177 these files (in the `phylo_snapshot` subdirectory) to make it easier to replicate the operation
178 (because the collection of trees and the tree files change due to curation).

179 **2.1.1 Pruning questionable taxa from the taxonomy**

180 OTT is a hierarchy of taxonomic names that implies a phylogenetic taxonomy. An OTT ID
181 has a position in the hierarchy, a taxonomic name, and set of references to the same name
182 in different taxonomies. In addition, the ID may also be associated with a set of flags that
183 can indicate that the taxon may be questionable. These flags can either encode information
184 taken from an input taxonomy (for example, taxa the NCBI refers to as “unplaced” are
185 assigned an “unplaced” flag) or can arise because of some form of conflict during taxonomy
186 construction (for example, if two taxonomies disagree on the name for a taxon, then the
187 taxon can be merged and the name will be retained without any descendants; this name will
188 have an OTT ID, but will be flagged as “barren”). Propinquity prunes the OTT down to
189 a more reliable taxonomy by pruning off parts of the tree that are flagged with suspicious
190 flags. The set of flags that lead to a subtree of the taxonomy being pruned is under the
191 control of the user (the set of flags used by the Open Tree of Life project can be found in
192 the `config.opentree.synth` file in the propinquity repository). For the purpose of the rest
193 of the pipeline, an OTT ID that has been pruned from the taxonomy will be treated in the
194 same way as invalid OTT ID. The output of this step is stored in propinquity's `cleaned_ott`
195 subdirectory; this operation only needs to be performed when the OTT or the pruning flags
196 change.

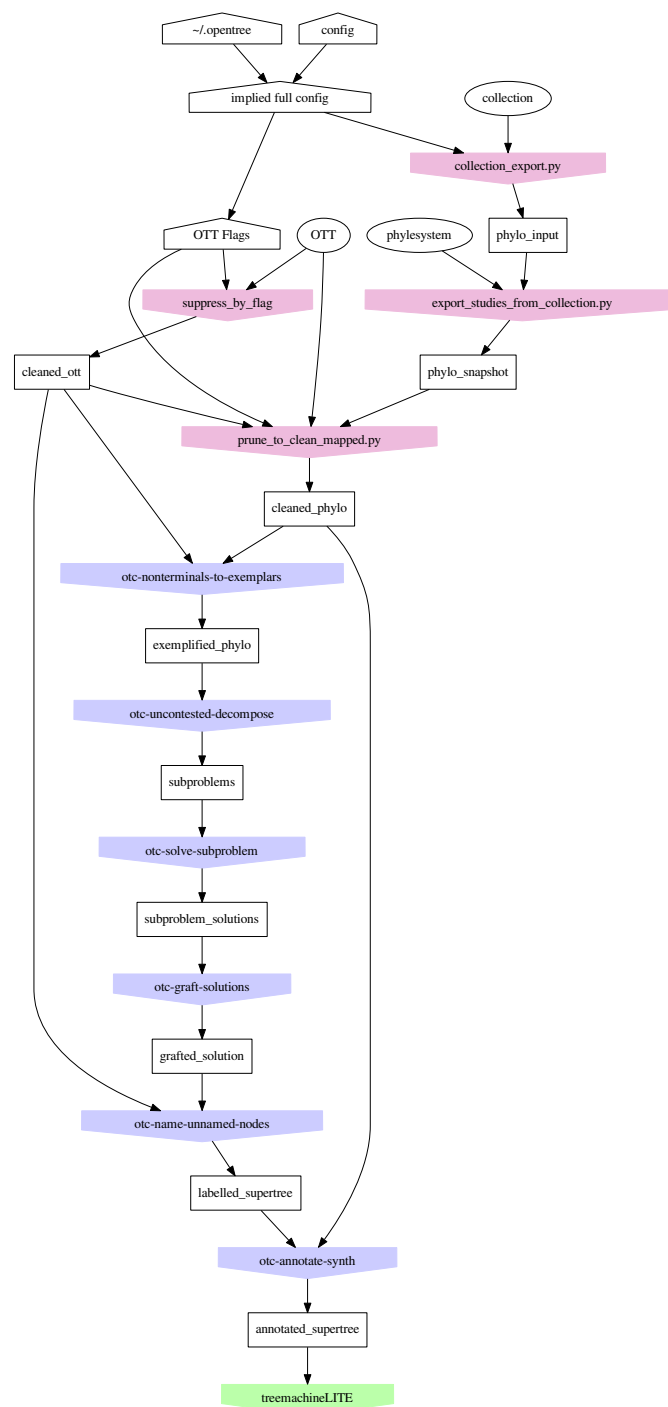


Figure 3. Organization of the *propinquity* pipeline. Each colored pentagon labels a program (blue for *otcetera*-based tools and red for python scripts in the *propinquity* or *peyotl* repository) that performs the important operations in each step; the number before the tool name refers to the section in this paper that describes the operation. The output of each step corresponds to a subdirectory of the *propinquity* system which will hold the output artifacts for the step. Ovals are resources that are required (OTT and Open Tree’s phylesystem repository). White pentagons are user-controlled inputs.

197 **2.1.2 Pruning problematically mapped tips from input phylogenetic trees**

198 Frequently, phylogenetic estimates are rooted using the outgroup criterion, which is an
199 assumption about the monophyly of the ingroup taxa. Because the rooting of the branches
200 in the outgroup portion of the tree is often uncertain, data curators can identify the ingroup
201 node of the tree; propinquity uses this annotation to prune off the outgroup taxa.

202 Frequently, not all tips in a phylogenetic input will have been mapped to a taxon in the
203 current version OTT. Unmapped leaves are pruned from each phylogenetic input. In some
204 cases, the OTT has changed and a taxon has been unambiguously mapped to another taxon.
205 This can occur when multiple species in one version of the taxonomy are “lumped” into a
206 single taxon in a subsequent version. OTT maintains a set of “forwarding” statements about
207 IDs that have been removed but can be mapped to an existing taxon; propinquity uses these
208 statements to update the OTU mapping of input trees.

209 Finally some leaves are mapped to taxa that occur more than once in the tree, or taxa
210 that have ancestors represented as tips of the tree. In these cases, leaves are pruned to assure
211 that tips are mapped to unique taxa that are not nested. In the case of nested taxa, the
212 tip mapped to the higher level taxon is pruned, and one of the lower level tips is retained.
213 In the case of duplicate occurrences of an OTT taxon, propinquity checks to see if a data
214 curator has selected one of the taxa to be the exemplar for the taxon. If this selection has
215 not been made, then the node with the lexicographically lowest ID is chosen to exemplify the
216 taxon. This choice is arbitrary, but repeatable. The pruned phylogenetic inputs are stored in
217 a `cleaned_phylo` subdirectory of propinquity.

218 **2.1.3 Exemplifying tips mapped to higher taxa**

219 Many input trees have tips that are not terminal taxa, but higher-order taxonomic groups.
220 It is not clear how to interpret a tip in a phylogenetic estimate that is labeled with the
221 name of a higher taxon. Several scenarios can lead to these cases: the data for the tip
222 could have been created by merging a chimeric set of character scores from constituent taxa;
223 the species sampled may not have been identified to the lowest taxonomic rank; or the
224 researcher may simply have used a higher taxonomic name because he/she assumed that the
225 taxon is monophyletic and the higher level name would be more recognizable. Rather than
226 allowing the ambiguity about interpretation of the higher-taxon mapped tips to propagate
227 throughout the entire pipeline, we transform the input trees by replacing higher taxa at tips
228 with a set of terminal-taxon exemplars for each taxon. One approach would be to simply
229 determine all descendant terminal taxa and attach them as children of the problematic tip.
230 However, this would create a clade rather than a tip; subsequent steps in the supertree would
231 interpret the clade as a claim of monophyly for the taxon. The input tree may not have tested
232 monophyly of the clade, so this interpretation is unwarranted. We avoid it by attaching
233 exemplar taxa as child nodes of the higher taxonomic tip but then collapsing the edge between
234 the former tip node and its parent. Thus, if A is a non-terminal taxon containing terminal
235 descendants a_1 and a_2 and B is a non-terminal taxon containing terminal descendants b_1
236 and b_2 we would replace the subtree $((A, B), c)$ with $((a_1, a_2, b_1, b_2), c)$ instead of the subtree
237 $((a_1, a_2)A, (b_1, b_2)B), c)$.

238 If a taxon is only present in the taxonomy (not in any of the input trees), then it can be
239 pruned from the taxonomy for the construction of the supertree and then grafted back on to
240 the summary tree later. Performing this pruning reduces the size of the supertree problem,

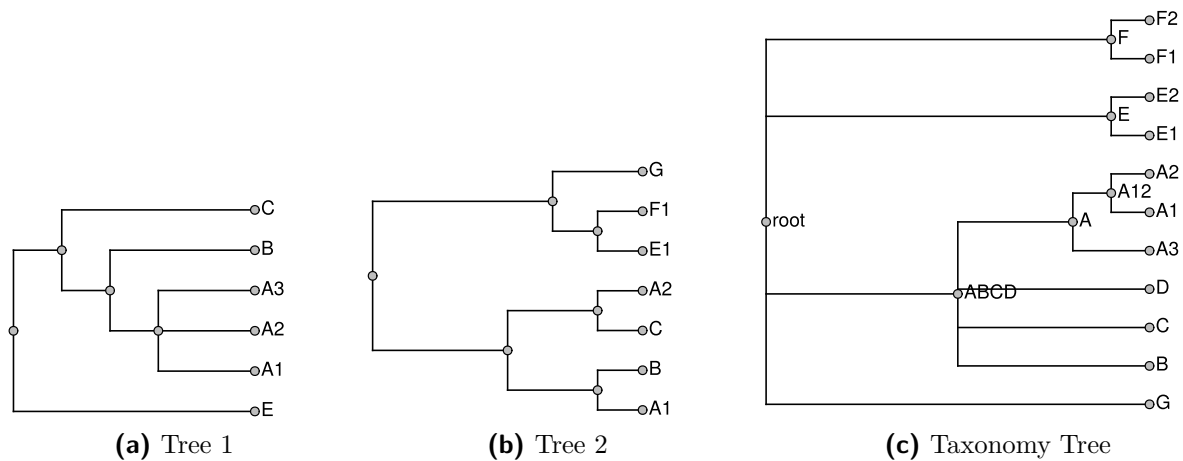


Figure 4. Input trees and taxonomy tree

241 reducing the running time of the pipeline. Similarly, when we expand a higher taxon in the
 242 exemplification step, we can omit members of the taxon if they do not occur in any of the
 243 phylogenetic inputs. If there are no members of the higher taxon sampled in any other input
 244 tree, then we arbitrarily choose one terminal taxon to represent the higher taxon. During
 245 the exemplification step, a tool from *otcetera* (*otc-nonterminals-to-exemplars*) reads the
 246 taxonomy and all of the “cleaned” phylogenetic estimates from the previous step. Reading
 247 all of the inputs is necessary to assure that each higher taxon is replaced with the same set
 248 of exemplars regardless of which tree the higher taxon occurs in, and that the exemplars for
 249 a higher taxon is the union on the set of descendant terminal taxa that have been sampled
 250 in a phylogenetic input. Figure 5 shows an example of how the trees in figure 4 would be
 251 exemplified.

252 We prune the taxonomy by removing tips that are not present in any input tree to produce
 253 the pruned taxonomy \mathbb{T}_p . The tips pruned in this step will be grafted back onto the skeleton
 254 of the summary tree in a subsequent step. A terminal taxon that is represented only in the
 255 taxonomy can be pruned and then regrafted onto the solution without affecting which nodes
 256 are displayed by the final summary tree. Thus, this procedure does not impede our ability to
 257 find a good summary tree. Removing these tips produces a smaller input to the rest of the
 258 pipeline, which reduces running times. After producing the set of “exemplified” phylogenetic
 259 inputs, this tool exports a pruned down version of the taxonomy that only contains tips that
 260 are present in at least one phylogenetic input.

261 2.2 Summary tree construction

262 After the preprocessing steps, the inputs have been converted to a set of rooted phylogenetic
 263 estimates in which each leaf is mapped to a terminal taxon in the exemplified taxonomic tree.
 264 The goal of the remainder of the pipeline is to construct a tree that maximizes the sum of
 265 displayed groups’ weighted scores (MSDGWS) criterion. This is accomplished in four steps:
 266 (1) dividing the full problem into subproblems based on uncontested taxa, (2) constructing a
 267 summary solution for each subproblem by greedily creating a maximally-sized list of groupings
 268 that can all be displayed simultaneously; (3) grafting the subproblem solutions into a single

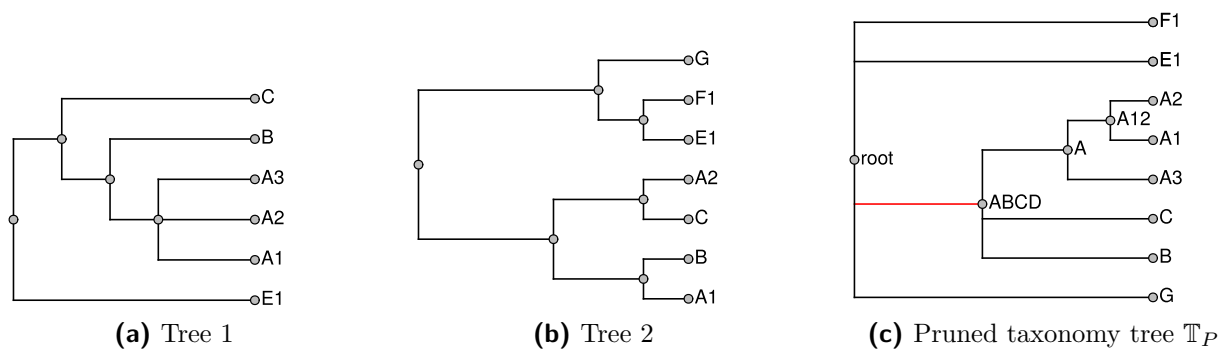


Figure 5. Exemplified input trees and taxonomy tree from figure 4. E in tree1 is exemplified by E1. Pruned taxa are E2, F2, and D. The taxa E and F are retained as monotypic taxa in the pruned taxonomy \mathbb{T}_P , but are not shown in panel c. The red edge in the pruned taxonomy tree is an uncontested higher taxon in the exemplified taxonomy (as explained in section 2.2.1)

269 supertree; and (4) grafting (or “unpruning”) the taxonomy-only taxa onto the solution to
 270 produce a complete summary tree.

271 2.2.1 Subproblem decomposition

272 For the sake of efficiency, propinquity uses a divide-and-conquer approach to construct the
 273 supertree. Subproblems are identified by searching through the taxonomy tree to find any
 274 taxa that are not contested by any single input tree. Here we say that input tree T_i contests
 275 taxon x in the pruned taxonomy, if x is not monophyletic in any resolution of tree T_i . Thus,
 276 polytomies in an input tree are treated as soft polytomies, and a taxon is not contested
 277 merely because it is not displayed by an input tree.

278 This operation is performed by the `otc-uncontested-decompose` tool in `otcetera`; see
 279 appendix B for a description of the algorithm. The output is a series of subproblems, each
 280 of which corresponds to a slice of the taxonomy and corresponding slices through each
 281 relevant input tree. Each uncontested non-terminal and non-root taxon will show up in two
 282 subproblems: it will be the root of its own subproblem and it will be tip in the subproblem
 283 that covers the next slice deeper in the tree. The red edge in figure 5c highlights the taxa
 284 that are not contested by the input shown in 5; figure 6 shows the subproblems that would
 285 be emitted as a result of this set of inputs. The supertree operation of Hinchliff et al. (2015)
 286 also used this `otcetera`-based decomposition step.

287 Note that decomposition into uncontested groups does not necessarily allow us to find
 288 the tree that maximizes the MSDGWS score. For example, see figure 7; that example is
 289 a variant of the situation shown in figure 2. In this case the groupings from each of the
 290 phylogenetic estimates, shown in panels 7a and 7b, could be displayed. That solution is
 291 shown in panel 7d, it displays two of the three input splits, but is optimal because no solution
 292 displays all three input groupings and the depicted solution displays the two highest ranked
 293 groupings. However, neither of the trees shown in panels 7a or 7b contest the taxon B shown
 294 in the taxonomy panel 7c. Thus, when using our decomposition, the branches leading to taxa
 295 $B1$ and $B2$ in the input phylogenetic trees would be sliced during the decomposition, and

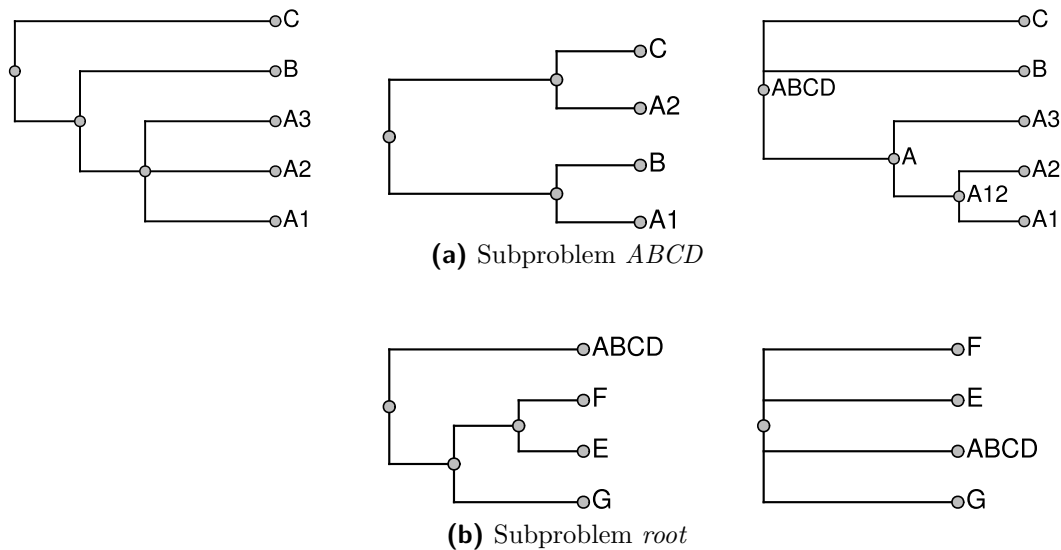


Figure 6. Subproblems generated from the exemplified trees shown in figure 5. A trivial statement from the first tree that a taxon labelled *ABCD* is sister to *E* has been omitted, because trees with only 2 leaves do not contain phylogenetic information.

296 relabeled to refer to taxon *B*. This taxonomically-informed interpretation of the inputs views
 297 the two phylogenetic inputs as in conflict; so the solution returned by propinquity would
 298 defer to the higher ranked tree. The tree shown in panel 7e would be returned. This example
 299 arises from the fact that the trees in 7a and 7b jointly contest taxon *B*, but neither contests
 300 taxon *B* when the trees are considered in isolation.

301 Despite the fact that the use of *otc-uncontested-decompose* can worsen the final score
 302 of the summary tree, we use this approach in propinquity because it makes the construction
 303 of the tree faster and it is easy for users to correct issues caused by incorrect taxa being
 304 constrained to be monophyletic. By adding a tree (even a low-ranked tree) that contests
 305 a taxon to the corpus of input trees, then the next synthetic tree will no longer consider
 306 the taxon to be uncontested. Thus the procedure encourages curation of more phylogenetic
 307 inputs as a means of improving the summary tree.

308 2.2.2 Subproblem solution

309 When solving sub-problems, we sequentially incorporate splits from trees in order of ranking,
 310 retaining splits that are compatible with the current set of splits (Alg 1). The order of
 311 splits from the same tree is not specified by this approach, and we incorporate splits using
 312 one of the possible post-order traversals of the tree. We make use of the BUILD algorithm
 313 (Aho et al., 1981) to assess compatibility. This strategy avoids unnecessary polytomies,
 314 since splits of later input trees are only rejected from the summary supertree if they conflict
 315 with higher-priority splits. Finally, we use the BUILD algorithm to construct a supertree
 316 displaying all of the splits in the set of compatible splits. Using the BUILD algorithm to
 317 construct the subproblem summary tree satisfies criterion 3, because trees from the BUILD
 318 algorithm do not contain unsupported branches.

319 The BUILD algorithm as originally stated by Aho et al. (1981) applies to a collection of

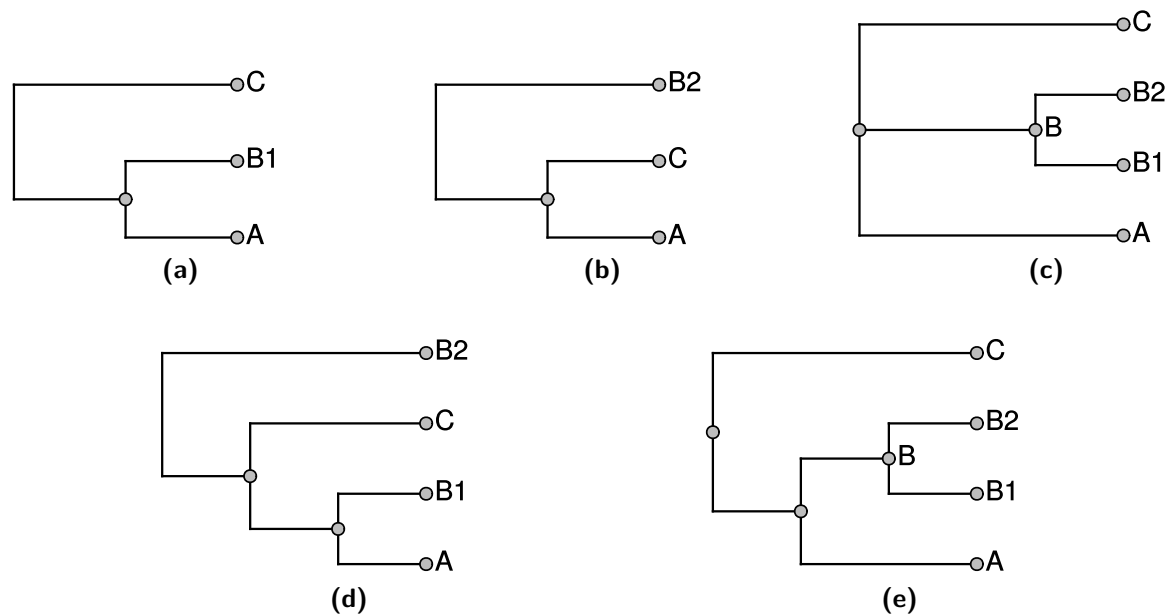


Figure 7. An example with three input trees: the highest ranked phylogenetic input panel (a), the second ranked phylogenetic input (b), and the taxonomy in panel (c). The summary tree in panel (d) has the highest possible score, but the summary shown in panel (e) would be returned from the pipeline that uses uncontested taxon decomposition.

Algorithm 1 ConsistentSplitsFromRankedList

Require: An ordered list of M splits, $\mathcal{R} = [R_1, R_2, R_3, \dots, R_3, \dots, R_M]$

$\mathcal{C} = [R_1]$

for each split i in $[2, 3 \dots M]$ **do**

$\mathcal{T} \leftarrow \mathcal{C} + R_i$

▷ where ‘+’ means concatenating 2 lists

if BUILD(\mathcal{T}) does not return null **then**

$\mathcal{C} \leftarrow \mathcal{T}$

end if

end for

return \mathcal{C}

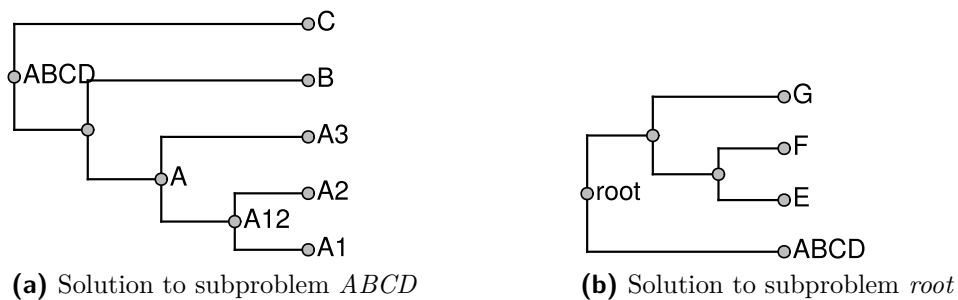


Figure 8. Solutions to the subproblems depicted in figure 6

320 rooted triplets. Instead of decomposing each input split into a collection of rooted triplets, we
 321 instead modify the BUILD algorithm to apply directly to larger rooted splits. The modified
 322 BUILD algorithm constructs a tree compatible with a collection of rooted splits, and returns
 323 failure if such a tree does not exist. This modified algorithm recovers the original BUILD
 324 algorithm if only rooted triplets are supplied as input. When larger splits are supplied as
 325 input, the results are the same as if each was decomposed into all implied triplets. The
 326 modified build algorithm has order $O(N^2 + N^2E + NL)$ where N is the number of splits
 327 passed in, E is the average size of the exclude group, and L is the total number of leaves. This
 328 simplifies to $O(N^2)$ if all splits are triplets. In this approach splits are either entirely retained
 329 or entirely discarded - consistent rooted triplets from conflicting splits are not retained.
 330 However, when unpruning taxonomy-only taxa (see below), we make an attempt to break
 331 ties in a way that preserves some partial information from conflicting splits by attaching taxa
 332 from conflicting splits at their common ancestor. Figure 8 shows the solutions that would
 333 be obtained by applying our modified version of the BUILD algorithm to the subproblems
 334 shown in figure 6.

335 2.2.3 Solution grafting

336 To produce a tree that spans all of the taxa sampled in the exemplified set of
 337 input trees, we graft the subproblem solutions into a single tree (stored as the
 338 `grafted_solution/grafted_solution.tre` by propinquity). Recall that each non-root
 339 uncontested taxon used for decomposition occurs as a leaf taxon in one subproblem and as a
 340 root taxon in one other subproblem. Thus, the grafting operation simply consists of reading
 341 all of the subproblem solutions into memory and then merging the nodes that are labeled
 342 with the same OTT ID.

343 2.2.4 Unpruning unsampled taxa

344 As described above, taxa that do not have any descendants in a sampled phylogenetic input
 345 are pruned from the taxonomy for the sake of efficiency. These taxa are reattached by an
 346 “unpruning step.” For those taxa that are compatible with the grafted tree, this step simply
 347 amounts to adding any unsampled taxonomic children to the node that represents the taxon
 348 in the grafted solution tree.

349 However, a taxon may be incompatible with the grafted solution; we refer to such taxa
 350 as “broken taxa.” If a broken taxon contains some unsampled children, it is not clear
 351 where these unsampled children should be attached to the grafted solution. One approach

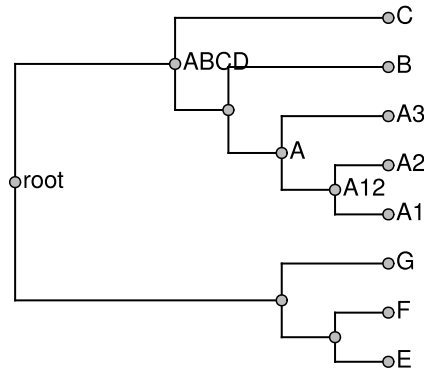


Figure 9. Grafted solution. The nodes E and F are monotypic here, but will end up being polytypic after unpruning is performed.

would be to mimic the application of Algorithm 1 to the full (unpruned) taxonomic tree. This would be equivalent to collapsing each edge in the taxonomy that attaches a broken taxon to its parent. The unsampled children of broken taxa would attach at their least inclusive ancestral taxon which is unbroken. In cases of several adjacent taxa are broken, this can lead to polytomies of very high degree deep in the tree. This can make the summary tree difficult to navigate. Thus, we have adopted an alternative solution. The `otc-unprune-solution-and-name-unnamed-nodes` tool from `otcetera` attaches the unsampled children of a broken taxa to the grafted solution as children of the MRCA of the sampled children.

Figure 10 illustrates the two approaches to unpruning. Taxa G, M, and R (Fig 10a) are broken because they conflict with the grafted solution (Fig 10b); among these, only taxon R has children that were unsampled in the grafted solution. Ignoring all broken taxa when unpruning would cause the unsampled children (R4, R5, and R6) to attached directly at taxon N (as in the tree shown in Fig 10c), because that is the least inclusive unbroken ancestor of R. The tree illustrated in Fig 10d shows the tree that would be produced by propinquity; the children of the broken taxon R and instead attached at the MRCA of sampled children (R1, R2, and R3). Their attachment point does not correspond to any taxon in the taxonomic tree.

2.2.5 Naming unnamed nodes

In order to annotate each node in the summary supertree, it is first necessary that each node have a unique identifier. Nodes whose include group correspond exactly to the include group of a node in the taxonomy are given the same identifier as the corresponding taxonomy node. These identifiers are of the form *ottX* where *X* is an integer OTT ID. We generate a label of the form *mrcaottX₁ottX₂* for a non-taxonomic node *n* where *X₁* and *X₂* are the OTT IDs for two leaves, *n* is the MRCA of these leaves, and *X₁* is the numerically smallest OTT ID that is a descendant of *n*, and *X₂* is the next the smallest ID that can be chosen to designate *n* as the MRCA. Because new taxa added to OTT will be given higher OTT IDs, the use of the lowest numbered OTT IDs as designators increases the chance that a node label can be encountered in a subsequent version of the tree (though the taxonomic content may change). The deterministic choice of designators also makes the labeling insensitive to branch rotation

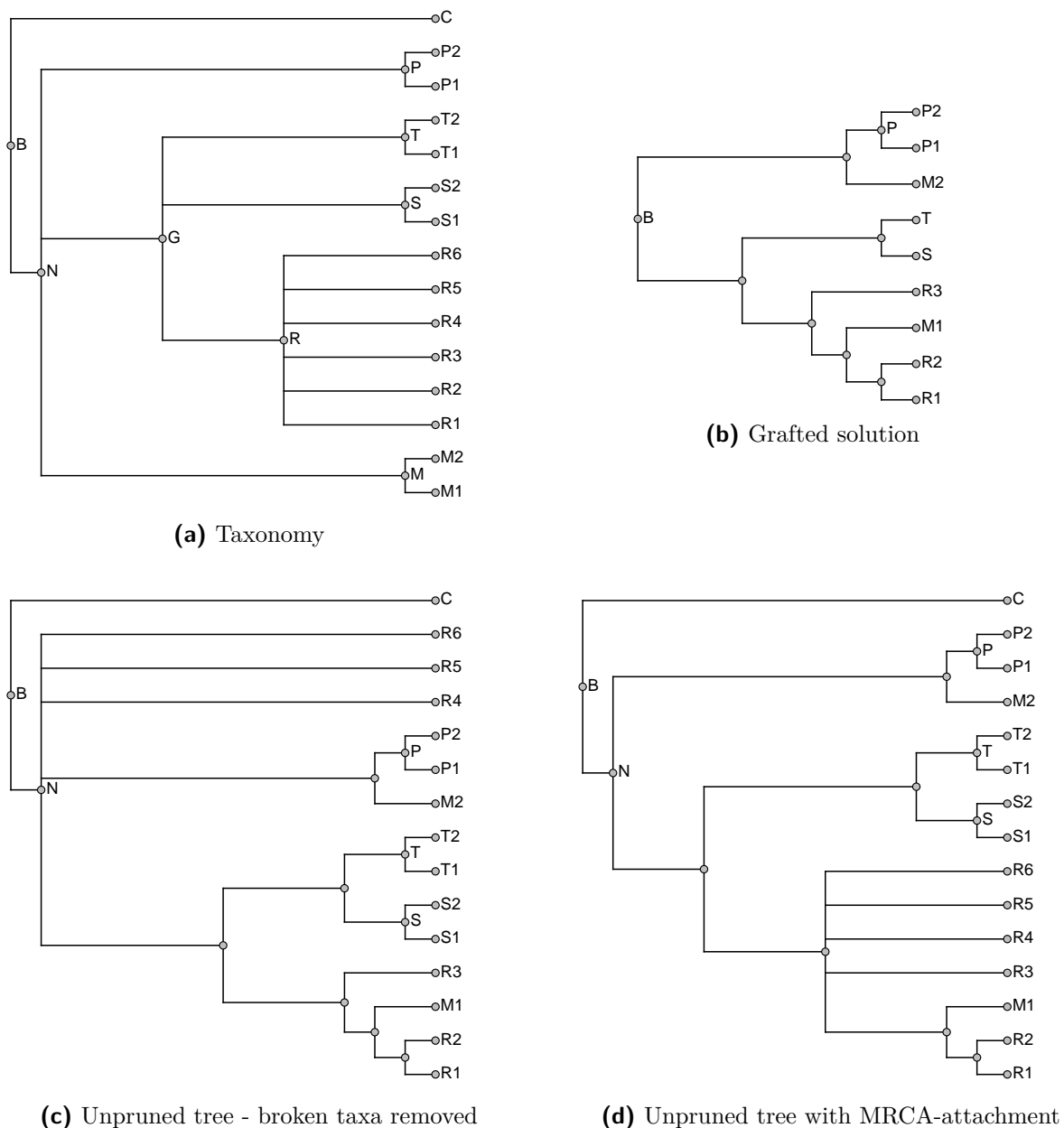


Figure 10. Two approaches to unpruning. Taxa G and R in the taxonomy (a) are broken because they conflict with the grafted solution (b). Removing these broken taxa from the taxonomy before unpruning leads to taxa R4, R5, and R6 being attached directly at taxon N, as in tree (c). In tree (d), the children of the broken taxon R are instead attached at the MRCA of R1, R2, and R3. Our method follows the second approach.

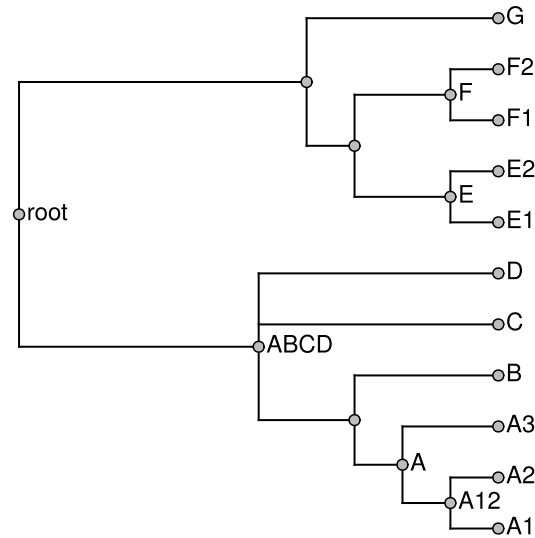


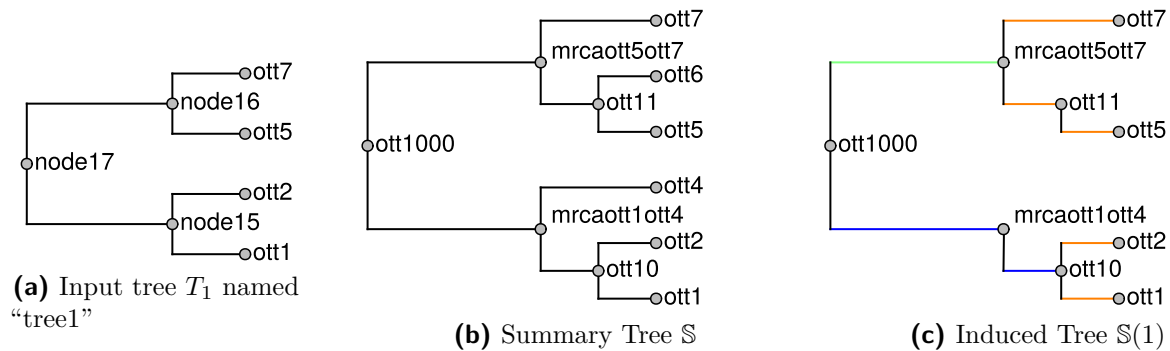
Figure 11. Unpruned tree with internal taxa. Nodes E2, F2, and D have been re-added to the tree. The nodes E and F are no longer monotypic.

382 of the grafted solution tree.

383 2.2.6 Annotation

384 To reveal the connections between the groupings found in the a summary supertree and
 385 the input trees, propinquity uses a few Python scripts and the `otc-annotate-synth` tool
 386 from `otcetera` to create an annotations file describing the pipeline used and the connections
 387 between phylogenetic information in the inputs and the summary. The JSON file produced
 388 by `otc-annotate-synth` encodes a “nodes” property that holds a mapping between a node
 389 name for the summary tree (using the naming convention described in the previous section)
 390 and a node provenance object that categorizes the relationship between the node and the
 391 inputs. The node provenance object for node x uses several properties to categorize the
 392 relationship between the node and the inputs; each property in the node provenance object
 393 maps to a structure storing the tree ID and node IDs for the input tree nodes.

394 Conceptually, this annotation operation is equivalent to considering every node j in
 395 each input tree i and the summary tree node x . Because the vast majority of nodes in the
 396 input studies will be compatible but not directly relevant to node x we do not list all of the
 397 compatible groupings. If node x is not included in the induced tree $\mathbb{S}(i)$, then none of the
 398 nodes of tree i will be referred to in the annotations for node x . Even if x is included in $\mathbb{S}(i)$,
 399 many of the nodes of T_i will be compatible with x while being relevant to other parts of the
 400 summary tree. The only input nodes listed for node x are with rooted taxon bipartitions
 401 which conflict with, are displayed by, or are resolved by the the rooted taxonomic bipartition
 402 associated with node x . All input nodes that cannot be displayed by any supertree that
 403 contains x are stored in a “conflicts_with” property of the node provenance object. If
 404 node j of T_i is displayed by the summary tree and x is part of the path of $\mathbb{S}(i)$ that displays
 405 the split between descendants of j and other taxa, then a reference to the node j will be in
 406 the node provenance object. The exact categorization of this annotation will depend on the



```
{
  "nodes": {
    "mrcaott1ott4": { "partial_path_of": { "tree1": "node15", ... } },
    "ott10": { "partial_path_of": { "tree1": "node15", ... } },
    "mrcaott5ott7": { "supported_by": { "tree1": "node16", ... } },
    "ott1": { "terminal": { "tree1": "ott1", ... } },
    "ott2": { "terminal": { "tree1": "ott2", ... } },
    "ott5": { "terminal": { "tree1": "ott5", ... } },
    "ott11": { "terminal": { "tree1": "ott5", ... } },
    "ott7": { "terminal": { "tree1": "ott7", ... } }
  },
  ...
}
```

(d) JSON annotations relating edges of (c) to edges of (a)

Figure 12. The relationship of edges in summary tree \mathbb{S} (b) to edges in the input tree T_1 (a). Only edges of \mathbb{S} that are present in the induced tree $\mathbb{S}(1)$ are represented by JSON annotations (d). Taxon names are here suppressed in favor of OTT IDs, and edges are referenced via their tipward nodes. Edges in $\mathbb{S}(1)$ that correspond to terminal edges of T_1 are orange; edges of $\mathbb{S}(1)$ that are supported by edges of T_1 are blue; where multiple edges of $\mathbb{S}(1)$ correspond to the same edge of T_1 they are green. There is no conflict in this example. Also, if this were output from propinquity, then each internal node of \mathbb{S} would be supported by other inputs trees that are not shown here.

407 configuration of node x on the induced tree $\mathbb{S}(i)$:

- 408 • if x is on a terminal path in the induced tree then node j will be listed in the “terminal”
409 property;
- 410 • if x is along an internal path that contains some nodes with out-degree equal to 1, then
411 node j will be listed in the “partial_path_of” property; and
- 412 • if x is along an internal path without any node of out-degree 1, then node j will be listed
413 in the “supported_by” property, because node j supports the existence of grouping x
414 in the sense that collapsing the edge that separates x from its parent would cause the
415 summary tree to no longer display node j .

416 These three relationships are illustrated in Figure 12. Finally, if T_i does not display x from
417 $\mathbb{S}(i)$, but there exists an unresolved node j in T_i which could be resolved such that the tree
418 would then display x , then a reference to node j will be listed in the “resolves” property of
419 node x .

420 The otceera annotation tool can also detect cases in which including information from
421 node j in T_i could further resolve a polytomy x in the summary tree; such a case would
422 be annotated using the “resolved_by” property of x . However, because of our goal of
423 excluding unnecessary polytomies, none of the nodes in propinquity’s summary tree will use
424 this annotation when they are annotated with the set of input trees.

425 2.2.7 Self-documentation

426 An optional step in the propinquity pipeline (triggered by the executing the “make html”
427 target) can compose an “index.html” file for each directory created during the pipeline
428 to explain the artifacts held in that directory and report summary statistics about the
429 summarization run.

430 3 RESULTS

431 We seek to assess the performance of our new supertree method by comparing it to the
432 supertree method of Hinchliff et al. (2015). The method of Hinchliff et al. (2015) was used to
433 construct the Open Tree of Life v4 (OTLv4). Therefore, in order to facilitate comparison, we
434 applied our method to the same input trees and taxonomy used by OTLv4. We refer to the
435 resulting supertree as OTLv4’ since it was constructed by applying the propinquity pipeline
436 to the same inputs as OTLv4.

437 3.1 Inputs

438 The flag-cleaned version of OTT used in the construction of both supertrees contained
439 2,424,255 leaves. The OTLv4 supertree was constructed from 482 phylogenetic inputs,
440 containing a total of 45,385 leaves, of which 41,029 were unique. After flag-cleaning and
441 exemplification by propinquity, these trees contained 40,323 unique tips. We used the same
442 cleaning flags to trim the taxonomy and input trees when constructing OTLv4’, so OTLv4
443 and OTLv4’ contain the same number of leaves.

Table 1. Representation of input splits in the OTLv4 tree and the OTLv4' tree.

	supported_by	partial_path_of	terminal	conflicts_with	resolved_by	resolves
OTLv4	34,595	923	45,385	2,760	2,718	473
OTLv4 only	745	54	0	2,055	2,718	0
OTLv4'	38,521	1,118	45,385	1,357	0	515
OTLv4' only	4,671	249	0	652	0	42

Table 2. Representation of taxonomy splits in the OTLv4 tree and the OTLv4' tree.

	supported_by	partial_path_of	terminal	conflicts_with	resolved_by	resolves
OTLv4	125,384	0	2,424,255	1,998	4	3,676
OTLv4 only	296	0	0	19	4	17
OTLv4'	125,107	0	2,424,255	2,279	0	3,883
OTLv4' only	19	0	0	300	0	224

3.2 Subproblems

In the OTLv4' summary tree, the decomposition procedure produced 5,406 subproblems, but only 1,422 of these were non-trivial to solve. If a subproblem contains only two tips it is trivial; 2,362 subproblems were trivial in this way. Similarly, if a subproblem contains only 2 trees it is trivial to solve because the solution will simply be all of the groupings from the first tree combined with all of the groupings from the second tree that are compatible with the first tree; 3,052 subproblems were trivial in this way. The subproblem with the largest number of tips contained 946 tips. The largest subproblem, in terms of the number of input trees (including the taxonomic tree) that were relevant, had 16 trees. Without decomposition, the supertree problem would have had 482 input trees and 41,226 leaves.

3.3 Representing input splits

We performed an annotation of both the OTLv4 tree and the OTLv4' tree as described in section 2.2.6 to assess the ability of our new supertree method to represent splits from input phylogenies. Table 1 classifies the input phylogeny splits according to how they relate to a summary tree, so that each input edge falls in one of **supported_by**, **partial_path_of**, **terminal**, **conflicts_with**, or **resolved_by**. For example, the numbers in the **conflicts_with** column indicate the number of input splits j with at least one summary edge x such that the relation “ x conflicts with j ” holds. The total number of non-terminal input phylogeny splits considered was 40,996.

The number of displayed input splits (**supported_by** + **partial_path_of**) increased from 35,518 (for OTLv4) to 39,639 (for OTLv4'); an 11% increase. When examining which splits are displayed, we find that the OTLv4' tree displays 4,920 input splits that are not displayed by the OTLv4 tree, whereas the OTLv4 tree displays only 799 input splits that are not displayed by the OTLv4' tree. The number of input splits that conflict with the summary (**conflicts_with**) dropped from 2,760 to 1,357, a decrease of 1,403, or 51%. In accordance with the goal of not containing unnecessary polytomies, the number of input splits that do not conflict with the summary tree, but are not incorporated (**resolved_by**) dropped from 2,718 to 0. We also find that the number of polytomies in input phylogenies that are resolved

472 by the summary tree increases from 473 for OTLv4 to 515 for OTLv4'.

473 We also performed an annotation of the OTLv4 tree and the OTLv4' tree to assess the
474 degree to which these trees represent taxonomy splits (Table 2). The OTLv4' tree conflicts
475 with 281 more taxonomy splits than the OTLv4 tree. Since the taxonomy is the lowest ranked
476 input tree, this increased conflict with the taxonomy is an expected result of incorporating
477 more splits from higher-ranked input phylogenies.

478 4 CONCLUSIONS

479 Here we have described the motivation and methodology used by our new supertree method
480 that is currently used by the Open Tree of Life project to build summary supertrees on the
481 scale of millions of leaves. Our new method represented 11% more input phylogeny splits
482 with 51% less conflict compared to the Open Tree of Life version 4 summary tree, when
483 applied to the same inputs. Unlike the previous method (Hinchliff et al., 2015), our new
484 method is guaranteed to incorporate input splits unless they conflict with the summary tree.
485 The method is implemented in the Open Source software package propinquity. A modified
486 version of the treemachine software which built the summary tree described in the Hinchliff
487 et al. (2015) paper is used by the project to serve the tree produced by propinquity via Open
488 Tree of Life APIs.

489 The migration of summary tree construction from treemachine (used for version 4) to
490 propinquity (for all versions from v5.0 to present) has increased the pace of synthesis tree
491 releases from the Open Tree of Life project. This is partly because the newly available
492 annotations feature has made it possible to identify which input trees are responsible for
493 taxa being included or excluded from the summary tree. Additionally, the new propinquity
494 software pipeline has decreased the decreased the computational time required to construct a
495 supertree from several hours to about 8 minutes (after some preprocessing steps which only
496 have to be performed when the input taxonomy changes). The amount of RAM required
497 during tree construction has also decreased substantially.

498 5 ACKNOWLEDGEMENTS

499 Thanks to Emily Jane McTavish, Karen Cranston, Jonathan Rees, Jim Allman, Cody
500 Hinchliff, Stephen Smith, and Joseph Brown for discussions and feedback. Thanks to NSF
501 grant 1208393 (DEB), part of the collaborative Open Tree of Life awards, the University of
502 Kansas, and the Heidelberg Institute for Theoretical Studies for funding.

503 REFERENCES

- 504 Aho, A. V., Sagiv, Y., Szymanski, T. G., and Ullman, J. D. (1981). Inferring a tree from
505 lowest common ancestors with an application to the optimization of relational expressions.
506 *SIAM Journal on Computing*, 10(3):405–421.
- 507 Hinchliff, C. E., Smith, S. A., Allman, J. F., Burleigh, J. G., Chaudhary, R., Coghill, L. M.,
508 Crandall, K. A., Deng, J., Drew, B. T., Gazis, R., Gude, K., Hibbett, D. S., Katz, L. A.,
509 Laughinghouse, H. D., McTavish, E. J., Midford, P. E., Owen, C. L., Ree, R. H., Rees,
510 J. A., Soltis, D. E., Williams, T., and Cranston, K. A. (2015). Synthesis of phylogeny

511 and taxonomy into a comprehensive tree of life. *Proceedings of the National Academy of*
512 *Sciences*, 112(41):12764–12769.

513 Huson, D. H., Rupp, R., and Scornavacca, C. (2010). *Phylogenetic networks: concepts,*
514 *algorithms and applications*. Cambridge University Press.

515 Karp, R. M. (1972). Reducibility among combinatorial problems. In *Complexity of computer*
516 *computations*, pages 85–103. Springer.

517 McTavish, E. J., Hinchliff, C. E., Allman, J. F., Brown, J. W., Cranston, K. A., Holder,
518 M. T., Rees, J. A., and Smith, S. A. (2015). Phylesystem: a git-based data store for
519 community-curated phylogenetic estimates. *Bioinformatics*, page btv276.

520 **A THE MAXIMUM SUM DISPLAYED GROUPS' WEIGHTED** 521 **SCORES CRITERION**

522 Let \mathcal{W} is a weighting function that maps any input tree's internal node to a non-negative
523 number. If $I(\mathbb{S}, i, j)$ is an indicator function that is 1 if summary tree \mathbb{S} displays the node
524 $V(i, j)$ and 0 otherwise then: $SDGWS(\mathbb{S}) = \sum_i \sum_j I(\mathbb{S}, i, j) \mathcal{W}(i, j)$ is the “sum of displayed
525 groups' weighted scores” for a tree where i indexes all of the input trees and j indexes each
526 non-root internal node in tree i . Preference for this tree is referred to as the maximum
527 sum displayed groups' weighted scores criterion (MSDGWS criterion). The summary tree
528 constructed by the propinquity pipeline is a greedy heuristic for finding a tree that maximizes
529 this score when the weights for a node are determined by the tree's weight and the difference
530 in weighting is so large that displaying one node from a highly ranked tree is preferred to
531 displaying all of the nodes in the trees with lower rank.

532 **B DESCRIPTION OF THE DECOMPOSITION ALGORITHM OF** 533 **OTC-UNCONTESTED-DECOMPOSE**

534 The input is the a ranked list of input trees and comprehensive taxonomy.

535 **B.1 Creation of multigraph of the taxonomy with embedded trees**

536 The tool creates a multigraph by starting with a graph isomorphic to the taxonomic tree.
537 The nodes and edges created in this step will be referred to as the “taxonomic graph.” Next,
538 we add nodes and edges to that graph in a procedure that we refer to as “embedding” the
539 input trees into the taxonomy. A node is introduced for each node in an input tree, and
540 these nodes are mapped the MRCA nodes in the taxonomic graph. In other words, for any
541 node y in an input tree with a cluster of descendants, \mathcal{C} , we find the most tipward node
542 z in the taxonomic graph that is an ancestor to all of the taxa in \mathcal{C} ; let $m(y) = z$ refer to
543 this mapping, and $m'(z) = y$ refer to the reverse mapping. Each edge e_{ij} in a source tree i
544 connects ancestor to its descendant, $a(e_{ij}) \rightarrow d(e_{ij})$. The edges are introduced into the graph.
545 We also introduce new edges to create a from $m(a(e_{ij}))$ through its descendants to $m(d(e_{ij}))$;
546 we denote this path $p(e_{ij})$ and refer to the edges in the path as “embedding edges for tree i ”.
547 Note, that this is a path through the taxonomic nodes, while the edge e_{ij} connects source
548 tree nodes. The mapping between e_{ij} and $p(e_{ij})$ is stored, and the edges are labelled with
549 the index i so that it is clear which tree created them. Because the taxonomic tree is highly

550 unresolved, it is frequently the case that $m(a(e_{ij}))$ is the same node as $m(d(e_{ij}))$; in these
551 cases the embedding edge is a loop. This situation occurs whenever edge e_{ij} can resolve part
552 of the polytomy represented by a taxon.

553 We treat the taxonomy as the lowest ranked input tree. The next step will collapse
554 contested edges in the taxonomic graph. To retain all of the information from the taxonomy
555 we embed the taxonomy into the taxonomic graph as if it were another input tree

556 B.2 Detection of uncontested higher taxa

557 After every input tree and the taxonomy tree have been embedded into the taxonomic graph,
558 we perform postorder traversal over the taxonomic graph that underlies the multi-graph.
559 For any internal node (each of which corresponds to a non-terminal taxon) we determine
560 whether or not it is contested by examining each input tree. We can determine tree i contests
561 the taxon represented by taxonomic node x by looking at the parents of all of the “exiting”
562 embedding edges for tree i . These are the set of embedding edges that have x as a daughter
563 and have a parent node that is not x (ergo a parent node that is taxonomically higher than
564 x). If there are more than one parent nodes in this set of exiting embedding edges, then tree
565 i contests that taxon. If there is only one parent node, then the all of the constituent taxa
566 belonging to this taxon that are present in tree i have one parent that is more inclusive; this
567 means that the input tree does not contest monophyly of the taxon.

568 If the taxon is uncontested, then it may be the case that some input trees do not contest
569 the taxon, but contain polytomies that could be resolved to display the taxon. The cases can
570 be identified by finding multiple exiting embedding edges that have the same taxon y as their
571 parent node. In these cases, a pseudo input tree node is created and becomes the parent
572 node for these edges; this new node is then connected to y as if it had been an input edge.
573 This operation is equivalent to resolving an input tree’s polytomy in favor of the monophyly
574 of the uncontested taxon. This is the only way in which the input trees are modified during
575 the decomposition.

576 B.3 Collapsing contested taxa from the taxonomic graph

577 If a taxonomic node x fails the “uncontested” test described in the previous section, then
578 the node corresponding to the taxon is removed from the taxonomic graph and the set of
579 edges (and mappings between input edges and embedded paths) is updated as if this taxon
580 had not been present in when the taxonomic graph was created. This consists of detecting
581 changing any edge in an embedding path that is adjacent to x by replacing the reference to x
582 with a reference to its parent $a(x)$. Note that we do not collapse the edge corresponding to
583 this taxon in the part of the graph that represents the embedding of the taxonomy into the
584 taxonomic graph. Thus, the taxonomy will still claim the monophyly of the taxon. This is
585 relevant if the input grouping that contests taxon x is overruled (in the subproblem solution
586 step) by a higher ranked split. In other words, the fact that the a taxon is contested during
587 the decomposition is not a guarantee that the taxon will not be monophyletic in the final
588 supertree.

589 B.4 Emitting subproblems

590 Whenever an uncontested taxon is identified, the appropriate slice of each input trees that
591 intersect with the taxon is written to a file. Then the multigraph is simplified by slicing any

592 off the taxon. This slicing is accomplished by examining all of the exiting embedding edges
593 for the taxon. The descendant taxa of each input tree is relabeled with the identifier of the
594 contested taxa and all of that node's descendants are removed. Thus this input node will act
595 as as if it were a leaf mapped to the uncontested taxon. All descendants of the taxonomic
596 graph are also pruned off.