

The Ultrametric Constraint and its Application to Phylogenetics

Neil Moore

Computing Science, University of St. Andrews, Scotland

NCAM@CS.ST-ANDREWS.AC.UK

Patrick Prosser

Computing Science, Glasgow University, Scotland

PAT@DCS.GLA.AC.UK

Abstract

A phylogenetic tree shows the evolutionary relationships among existing species. Internal nodes of the tree represent speciation events and leaf nodes correspond to existing species. A goal of phylogenetics is to combine such trees into larger trees, called supertrees, whilst respecting the relationships in the original trees. A rooted tree exhibits an ultrametric property; that is, for any three leaves of the tree it must be that one pair has a deeper most recent common ancestor than the other pairs, or that all three have the same most recent common ancestor. This inspires a constraint programming encoding for rooted trees. We present an efficient constraint that enforces the ultrametric property over a symmetric array of constrained integer variables, with the inevitable property that the lower bounds of any three variables are mutually supportive. We show that this allows an efficient constraint-based solution to the supertree construction problem, an important problem in phylogenetics [MS07, BE04, Pen03]. We demonstrate that the versatility of constraint programming can be exploited to allow solutions to variants of the supertree construction problems.

1. Introduction

One of the grand challenges of phylogenetics is to build the *Tree of Life* (ToL), a representation of the evolutionary history of every living thing. To date, biologists have catalogued about 1.7 million species, yet estimates of the total number of species range from 4 to 100 million. Of the 1.7 million species identified only about 80,000 species have been placed in the ToL [Pen03]. There are applications for the ToL: to help understand how pathogens become more virulent over time, how new diseases emerge, and to recognise species at risk of extinction [Pen03, MGP03]. One approach to building the ToL is divide and conquer, combining smaller trees into so called “supertrees” [BE04]. Phylogenetic trees have been created for relatively small sets of species, such as in TreeBASE [Tre03], and one of the challenges is to combine together existing trees into larger supertrees [BE04].

Supertree construction has been dominated by imperative techniques such as [SS00, SDH⁺04, Dan03, BES06b, NW96, BS95, Pag02] but recently new declarative approaches have emerged using constraint programming and answer set programming [GPSW03, Pro06, WYL07, BFL08]. One of the properties of rooted trees that suits these approaches is that trees are by their nature ultrametric. That is, in rooted trees the root node has depth 0, and the depth of other nodes is 1 plus the depth of their parent. Taking any three leaves a , b and c of the tree it must be that one pair has a deeper *most recent common ancestor*

(mrca) than the other pairs, or that all three have the same mrca. And this is what we mean by ultrametric, that for any three variables there is a tie for the minimum. In fact, if we know the depth of the mrca of all pairs of leaves we uniquely determine the structure of the tree. This inspires a constraint programming encoding for rooted trees, using the ultrametric constraint that we will define later. We explore solutions to the phylogenetic supertree problem and its variants. In so doing, we show the practicality of an ultrametric encoding for rooted tree problems, as well as to argue that this is a valuable addition to the set of techniques solving for the supertree problem and its variants.

The paper is organised as follows. First, we briefly introduce constraint programming and the supertree construction problem. We then propose a specialised ultrametric constraint, in terms of its propagation procedures, that maintains bounds(Z) consistency [Bes06a] on three variables. We show that this specialised constraint is required because models that use toolkit primitives cannot guarantee the ultrametric property on the supertree problem via propagation alone. Furthermore, the space complexity of such models becomes prohibitive. The ultrametric constraint is then extended to maintain the property over a symmetric matrix of variables. We then go on to show that this constraint can be efficiently applied to the problem of supertree construction, in particular that applying propagation to this model gives a polynomial time decision procedure for supertree construction. We then demonstrate this on real data and give justification that there has been an improvement in time and space over previous constraint encodings. One of the benefits of the constraint programming approach is that variants of the supertree problem can be addressed within this one model. We justify this assertion by proposing a constraint solution finding essential relations in the supertree [Dan03], addressing ancestral divergence dates [SDH⁺04, BSS04], modelling nested taxa [Pag04, DS04] and coping with conflicting data.

2. Background

In this section we give some necessary definitions and descriptions of the Constraint Satisfaction Problem [Tsa93], Constraint Programming, and the Supertree problem.

2.1 Constraint Programming and the CSP

Constraint Programming (CP) [RvBW07] is a declarative style of programming where problems are modelled as a CSP, i.e., as a set of variables that have to be assigned values from those variables' domains to satisfy a set of constraints. Values might typically be integers drawn from finite domains, real numbers from ranges, or more complex entities like sets or graphs. We will only be considering integers.

Definition 1. A constraint satisfaction problem (CSP) is a triple (V, D, C) where V is a set of n variables $\{v_1, \dots, v_n\}$; $D = \{dom(v_1), \dots, dom(v_n)\}$ is a collection of ordered domains; and $C = \{c_1, \dots, c_e\}$ is set of e constraints, each with a scope of variables $scope(c) = (v_{c_1}, \dots, v_{c_k})$ and a relation $rel(c) \subseteq dom(v_{c_1}) \times \dots \times dom(v_{c_k})$. An assignment to a CSP is a set of pairs of the form (v, x) where $v \in V$ and $x \in dom(v)$. A constraint $c \in C$ is satisfied by an assignment $\{(v_{c_1}, x_{c_1}), \dots, (v_{c_k}, x_{c_k})\}$ when $scope(c) = \{v_{c_1}, \dots, v_{c_k}\}$ and $(x_{c_1}, \dots, x_{c_k}) \in rel(c)$.

$rel(c)$. An assignment $\{(v_1, x_1), \dots, (v_n, x_n)\}$ is a solution when that assignment satisfies all the constraints in C .

A constraint solver finds a solution to a CSP via a process of constraint propagation and search. Constraint propagation takes place when a variable loses values (or loses a bound or is given a final value, as special cases). Propagation then maintains a level of consistency, such as arc-consistency [Mac77], across the variables, removing values from domains that cannot occur in any solution (i.e., removing unsupported values). Each constraint has an associated propagation algorithms. The search process is more often than not chronological backtracking, but might be something more exotic such as a discrepancy based search [HG95] or local search. For example, when modelling a routing problem we might have a constrained integer variable for each location to be visited, with a domain of values corresponding to the index of the next location to be visited (the so called “single-successor” model). The subtour elimination constraint [CL97] might then be used to ensure that only legal tours are produced, and Régin’s all-different constraint [R94] could be added to increase domain filtering. Side constraints could be added to ensure that some locations are visited before others, as in a pick up and delivery problem. For a job shop scheduling problem we might have a model that uses 0/1 variables to decide the order between pairs of activities that share a resource, and we might increase propagation by adding the edge finding constraint [CP94].

The constraint programming approach is general and practical for modelling and solving problems, and provides a framework for the combination of problem specific algorithms in one solver. This allows us to solve classes of problems efficiently and to model rich problem variants via the addition of side constraints.

2.2 The Supertree Problem

Supertree construction is a problem in phylogenetics where we are to combine leaf labelled species trees, where the sets of leaf labels intersect, into a single tree that respects all arboreal relationships in each input tree [BE04]. Species trees describe part of the evolutionary history of a set of species. Labels on leaves correspond to existing species and internal nodes represent divergence events in evolutionary history where one species split into at least two other species. Species trees may also be annotated with dates on internal nodes, representing the time at which the divergence event happened.

We define the term *displays*, which makes precise what we mean by “respects arboreal relationships”: supertree $T1$ *displays* a tree $T2$ if and only if $T2 = T4$ where $T4$ is obtained by the following steps (see [SS00]):

1. Let L be the set of leaves of $T1$ that are in $T2$.
2. Obtain $T3$ by removing leaves from $T1$ that are not labelled by L as well as any edges incident to them.
3. To obtain $T4$: wherever there is a path consisting of degree 2 interior nodes in $T3$, contract this into a single edge.

The problem is then to produce from a forest of input trees F a rooted species tree T that contains all the species in F and displays every tree in F . Figures 1 and 2 illustrate the displays property.

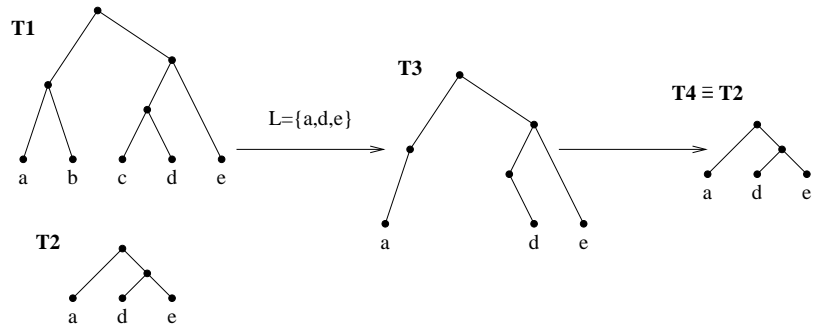


Figure 1: An example of a tree **T1** that displays a tree **T2**

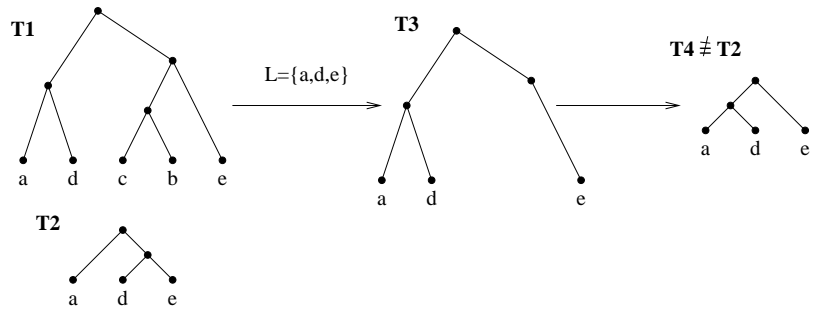


Figure 2: An example of a tree **T1** that does not display tree **T2**

Variants on the supertree problem that have previously been published and solved in the specialist bioinformatics literature include finding all solutions¹, counting solutions, finding conserved relationships in all supertrees [Dan03], incorporating nested taxa [SDH⁺04], incorporating ancestral divergence dates [SDH⁺04] and the possibility of contradictory input data [SS00].

3. The ultrametric constraint

The ultrametric constraint was first proposed in [GPSW03] within the context of supertree construction [BE04], and was implemented using toolkit primitives. We review this encoding and show that in most constraint toolkits this is inefficient in terms of both space and time. This motivates the creation of a specialised ultrametric propagator over three variables, that maintains the ultrametric property over the bounds of those variables. This is presented in terms of its propagation methods. This is then extended to a specialised propagator that maintains the ultrametric property on a symmetric matrix of variables.

3.1 Previous work on the ultrametric constraint

First, we start with a definition of the ultrametric constraint.

Definition 2. *An ultrametric constraint on three variables (henceforth, UM-3) x , y and z constrains them such that:*

$$(x > y = z) \vee (y > x = z) \vee (z > x = y) \vee (x = y = z) \quad (1)$$

This constraint ensures that there is a tie for the least element of the three, i.e., either all three are the same, or two are the same and the other is greater. The constraint was proposed by [GPSW03, Pro06] and was implemented as a literal translation of equation 1 using toolkit primitives. Evidence obtained from the JCHOCO, ECLIPSE and ILOG constraint programming toolkits shows that no propagation is done to lower bounds by this combination of primitive constraints. This is due to the disjunctive constraints. In many constraint programming toolkits propagation on disjunctive constraints is delayed until all but one of the disjuncts is disentailed, i.e., *delayed-disjunction* consistency is used [HSD98]. Consequently, in the above encoding of the ultrametric constraint a non-ultrametric instantiation can occur. Consider the case for three variables: $x \in \{1, 2, 3\}$, $y \in \{2, 3\}$ and $z \in \{3\}$. The domains of the variables are already at a fixed point with respect to delayed-disjunction consistency and a search process can instantiate these variables and fail (by assigning x the value 1). As we shall see later, finding a solution to the supertree problem using toolkit constraints can result in a backtracking search. Of course, higher levels of consistency would overcome this, such as constructive-disjunction consistency [HSD98], singleton arc-consistency [DB97] or the filtering algorithm of [Lho03]. However, the cost of these is greater in the average case than delayed-disjunction, preventing their use in toolkits. In fact for the UM-3 constraint it is especially unfortunate that the lower bounds may not be trimmed properly:

1. there may be multiple supertrees for the same set of input trees

Lemma 1. *In the UM-3 constraint, when lower bounds are supported (i.e., form an ultrametric instantiation with values in the other constrained variables), they support each other.*

Proof. Consider three supported lower bounds. Suppose for a contradiction that the two least of these are distinct. One of these is distinct lowest and it cannot be supported on account of the fact that it is not equal to anything or larger than anything. Therefore by contradiction the two least must be equal. However the other lower bound is at least as large as these, so the lower bounds are mutually supportive. \square

This Lemma will have important implications for the species tree model to be presented in detail in Section 4: in particular, the lower bounds of a BC(Z) model form a solution, where bounds consistency and bounds(Z) consistency [Bes06a] are defined as follows

Definition 3. *Given (V, D, C) and constraint $c \in C$, a tuple $\tau = (x_{c_1}, \dots, x_{c_k})$ is a bound support when $\tau \in \text{rel}(c)$ and for all $x_{c_i} \in \tau$, $\min(\text{dom}(v_{c_i})) \leq x_{c_i} \leq \max(\text{dom}(v_{c_i}))$. A constraint c with $\text{scope}(c) = (v_{c_1}, \dots, v_{c_k})$ is bounds(Z) consistent if for all $v_{c_i} \in \text{scope}(c)$ there exist bound supports involving both $\min(\text{dom}(v_{c_i}))$ and $\max(\text{dom}(v_{c_i}))$. A CSP is bounds(Z) consistent when every constraint $c \in C$ is bounds(Z) consistent.*

We will henceforth abbreviate “bound support” to “support” and bounds(Z) consistency to BC(Z).

3.2 Design of a UM-3 propagator

In this Section we describe the design of a UM-3 propagator enforcing BC(Z).

3.2.1 ANALYSIS OF LOWER AND UPPER BOUNDS

In this section we don’t take account of domains becoming empty. Since we analyse lower and upper bounds in isolation a lower bound may pass an upper bound or vice-versa, thereby emptying a domain. If this happens then the propagator described in Section 3.2.2 will not enforce BC(Z), rather it will terminate. This is not a problem, because a domain becoming empty means that there is no solution and to continue would be a waste of time. Concordantly, in this section, when we analyse lower bounds we will assume the upper bound is ∞ so that the domain cannot become null for this reason, and vice-versa.

The procedure LBFix in Figure 3 takes as input three variables and removes any unsupported values at the lower bounds of the domain. The intuition for the algorithm achieving this is that each one needs to be involved in a tie for least element, hence if the smallest lower bound is strictly less than the others then it must be unsupported.

The possible states of lower bounds when LBFix is invoked are summarised in Figure 4. Either all three are different (case 1), all three are the same (case 2) or two are the same and one different (cases 3 and 4). These give relationships between bounds at a point in time when some lower bound may be unsupported. The boxes in Figure 4 are shaded as follows: regions shaded black are removed by propagation whereas gray regions are supported. What the diagrams are *not* supposed to suggest is that, for example in case 1, the bounds differ by 1. Rather when two bounds are lined up they are the same bound and when one is

Code style: **SortOnLowerBounds**(x, y, z) returns a tuple of references to variables x , y and z in non-decreasing order by the lower bound of their respective domains; **SortOnUpperBounds** is analogous; **let** (S, M, L) $\leftarrow \dots$ names the references S , M and L ; $x.\mathbf{lb}$ and $x.\mathbf{ub}$ return references to the lower and upper bounds of x , respectively; as a result of $S.\mathbf{lb} \leftarrow M.\mathbf{lb}$, the lower bound of S is assigned equal to the value of the lower bound of M , although if $M.\mathbf{lb}$ subsequently changes they will be distinct again; the expression $x \cap_b y$ returns the intersection of the ranges $[\min(\text{dom}(x)) \dots \max(\text{dom}(x))]$ and $[\min(\text{dom}(y)) \dots \max(\text{dom}(y))]$.

Algorithm UM-3-BC(Z)

LBFix(v_1, v_2, v_3)

A1 **let** (S, M, L) \leftarrow SortOnLowerBounds(v_1, v_2, v_3)

A2 **if** ($S.\mathbf{lb} < M.\mathbf{lb}$) **then**

A2.1 $S.\mathbf{lb} \leftarrow M.\mathbf{lb}$

UBFix(v_1, v_2, v_3)

A3 **let** (S, M, L) \leftarrow SortOnUpperBounds(v_1, v_2, v_3)

A4 **if** ($S.\mathbf{ub} < M.\mathbf{ub}$) **then**

A4.1 **if** ($S.\text{dom} \cap_b L.\text{dom} = \emptyset$) **then**

A4.2 $M.\mathbf{ub} \leftarrow S.\mathbf{ub}$

A4.3 **else if** ($S.\text{dom} \cap_b M.\text{dom} = \emptyset$) **then**

A4.4 $L.\mathbf{ub} \leftarrow S.\mathbf{ub}$

Min event(v_1, v_2, v_3)

A5 LBFix(v_1, v_2, v_3)

A6 **if** all domains are non-empty **then**

A6.1 UBFix(v_1, v_2, v_3)

Max event(v_1, v_2, v_3)

A7 UBFix(v_1, v_2, v_3)

Fix event(v_1, v_2, v_3)

A8 LBFix(v_1, v_2, v_3)

A9 **if** all domains are non-empty **then**

A9.1 UBFix(v_1, v_2, v_3)

Figure 3: Algorithm for UM-3 propagator

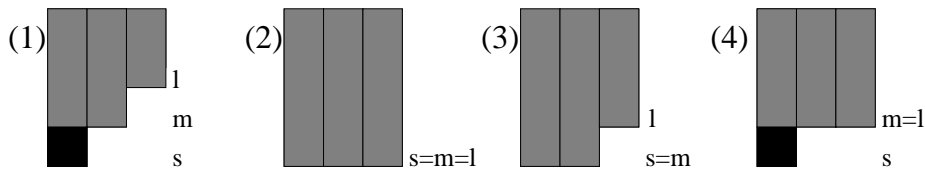


Figure 4: Cases in the analysis of LBFix

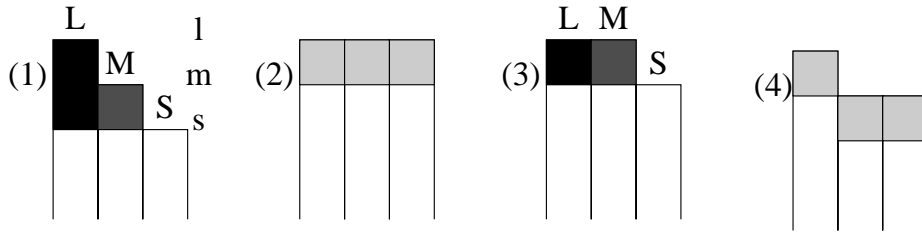


Figure 5: Cases in the analysis of UBFix

different from another they are different by some non-zero but unspecified amount. Hence they describe relationships and not actual values.

The following shows that LBFix removes all unsupported values and does not remove any supported values.

Lemma 2. *After LBFix is invoked all lower bounds of the argument variables are supported w.r.t. the UM-3 constraint and no supported values are removed.*

Proof. For cases 1 and 4 (Figure 4) the condition on line A2 is satisfied, so line A2.1 is executed, this results in the removal of the unsupported range and by inspection the remaining bounds are mutually supportive. In cases 2 and 3, the condition on line A2 is failed and so no changes are made to the domains; the bounds are mutually supportive. \square

The procedure UBFix in Figure 3 does the same job to upper bounds that LBFix does to lower bounds. The following Lemma justifies this assertion and the cases used in the proof are shown in Figure 5:

Lemma 3. *After UBFix is invoked either*

- *all upper bounds of the argument variables are supported w.r.t. the UM-3 constraint and no supported values are removed, or*
- *a domain is null as a result of removing unsupported values.*

Proof. Let S , M and L be the domains with smallest, middle and largest upper bounds, breaking ties arbitrarily. Let s , m and l be these upper bounds.

First we will prove that in case 1 (Figure 5) the shaded region in M is supported if and only if $L \cap_b S \neq \emptyset$: Potentially, the bound can be supported by

- equal values in S and L at least as small as it (i.e., $S \cap_b L \neq \emptyset$), or
- an equal value in either S or L , and a value at least as large in the remaining domain.

However, notice that the latter is impossible due to the fact that only L contains an equal value, and S has no value as large as this.

Similar arguments can establish that the shaded regions in L in case 1, M in case 3 and L in case 3 are supported if and only if $M \cap_b S \neq \emptyset$, $L \cap_b S \neq \emptyset$ and $M \cap_b S \neq \emptyset$, respectively.

Now we will establish the Lemma for each of the cases 1, 2, 3 and 4:

Cases 2 and 4 The condition on line A4 is false, and so no domains are changed. The upper bounds are mutually supportive in each case.

Case 1 and shaded region of M is unsupported From above $L \cap_b S = \emptyset$. Hence UBFix line A4.2 will be executed and the unsupported region removed. Now the upper bounds $l \in L$, $s \in M$ and $s \in S$ are mutually supportive.

Case 1 and shaded region of M is supported From above $L \cap_b S \neq \emptyset$. If the shaded region of L is also supported then $M \cap_b S \neq \emptyset$ and so neither line A4.2 nor A4.4 is executed and no changes are made to the domains. The upper bound of S is also supported, by $m \in L$, $m \in M$ and $s \in S$. If the shaded region of L is *not* supported then $M \cap_b S = \emptyset$ and so line A4.4 is executed resulting in the removal of the region. The new bounds $s \in S$, $m \in M$ and $s \in L$ are mutually supportive.

Case 3 and shaded regions of M and L supported From above, $L \cap_b S \neq \emptyset$ and $M \cap_b S \neq \emptyset$. Hence no domain changes result from executing UBFix. $l \in L$ and $s \in S$ are supported by $l \in L$, $s \in M$ and $s \in S$. $m \in M$ is supported by $s \in L$, $m \in M$ and $s \in S$.

Case 3 and shaded regions of M and L are unsupported From above, $L \cap_b S = \emptyset$ and $M \cap_b S = \emptyset$ and so line A4.2 of UBFix is executed and this results in M becoming null.

Case 3, shaded region of M supported but shaded region of L not supported From above $L \cap_b S \neq \emptyset$ and $M \cap_b S = \emptyset$, so that A4.4 is executed to remove the unsupported region. The new bounds of $s \in S$, $m \in M$ and $s \in L$ are mutually supportive.

Case 3, shaded region of L supported but shaded region of M not supported Symmetric with previous case.

□

Note that there is no analog of Lemma 1 for upper bounds since, for example, the bounds of $x = \{1, 2, 3\}$, $y = \{1, 2\}$ and $z = \{1\}$ are all supported, but *not* mutually supportive.

3.2.2 THE PROPAGATION ALGORITHM

Having presented LBFix and UBFix we are now in a position to present the complete propagation algorithm. The propagator works with arbitrary domains and it enforces $BC(Z)$, except when a domain becomes empty, in which case it does no further work. The algorithm is described by the action taken when any of three domain events occur:

min The domain has lost its lower bound since propagator was last invoked.

max The domain has lost its upper bound since propagator was last invoked.

fix The domain is a singleton, i.e., the variable is instantiated.

The algorithm is listed in lines A5-A9 of Figure 3. Intuitively these procedures work because, as we will show, a change to an upper bound can affect the support for other upper bounds, but a change in a lower bound can affect support for both lower and other upper bounds. Hence we need only run LBFix when a lower bound may have changed, but UBFix must be run for a change of either lower or upper bounds. Whilst it would be correct to cycle between trimming upper and lower bounds until a fixed point is reached, we can guarantee a fixed point more easily.

Lemma 4. *It is possible for a change in a lower bound to result in the loss of support for another lower bound.*

Proof. All the bounds in the diagram have support, but when the black shaded lower bound is lost the dark gray shaded lower bound loses support.



□

Lemma 5. *It is possible for a change in a lower bound to result in the loss of support for an upper bound.*

Proof. All the bounds in the diagram have support, but when the black shaded lower bound is lost the dark gray shaded upper bound loses support.



□

Lemma 6. *It is possible for the loss of an upper bound to cause the loss of support for another upper bound.*

Proof. All the bounds in the diagram have support, but when the black shaded upper bound is lost the dark gray shaded upper bound loses support.



□

Corollary 1. *It is impossible for a change in an upper bound to result in the loss of support for a lower bound.*

Proof. By Lemma 1 a lower bound retains support as long as the other lower bounds are intact, hence losing an upper bound has no effect. □

Why the asymmetry between upper and lower bounds? It is due to the asymmetry in the definition of UM-3 and has the practical repercussion that BC(Z) lower bounds must be mutually supportive whereas BC(Z) upper bounds may not be and may require support from other values including lower bounds.

Corollary 1 suggests that a further improvement on the algorithm in Figure 3 is to execute Line A6 and A9 if and only if any lower bound lost is the only remaining support for an upper bound. However, the conditionals intrinsic in UBFix amount to much the same thing and there is little point in repeating them.

The point of these theorems has been to build up a complete proof of correctness and BC(Z) status:

Theorem 1. *The code for min, max and fix events listed in Figure 3 does not remove any values involved in bound supports for the UM-3 constraint, and, if all the domains are non-null after propagation, the resultant domains will be $BC(Z)$.*

Proof. First we must establish that no values are removed during propagation that could be involved in a support, and that the result domains are subsets of the input domains. The former is immediate from Lemmas 2 and 3, because values are only removed as a result of executing LBFix and UBFix. The latter is immediate from inspection of LBFix and UBFix, because they only ever make lower bounds larger and upper bounds smaller.

The final thing to establish is that $BC(Z)$ is enforced, unless a domain becomes null.

If any domain becomes empty as a result of running the algorithm then the Theorem is trivially true.

If no domain becomes empty then we must show that all the bounds are supported. For lower bounds, by Lemma 4 and Corollary 1 we know that only the loss of a lower bound can result in the need to change a lower bound during propagation. Lower bounds can change as a result of either fix or min events, hence the propagator in Figure 3 runs LBFix in either event. When LBFix runs it leaves all lower bounds supported, as was shown in Lemma 2. For upper bounds, by Lemmas 5 and 6 we know that the loss of either a lower or upper bound can result in the loss of an upper bound. Hence upper or lower bounds can change as a result of any event, and lower bounds can also change as a result of LBFix, hence the propagator runs UBFix in all events, and it runs after LBFix has finished, if necessary. When UBFix runs it leaves all upper bounds supported provided no domain becomes empty, as was shown in Lemma 3. \square

The propagation algorithm runs in $\Theta(1)$ time. This is because all of the operations in LBFix, UBFix, min, max and fix events are $\Theta(1)$, provided that our domain representation allows access to the upper and lower bounds in $\Theta(1)$. This can be guaranteed if domain reductions only occur at the bounds, as is the case here, or if domains are represented using one of the structures proposed in [vHDT92].

3.3 Entailment

A constraint is entailed when all possible subsets of the variables in its scope are consistent. When this happens we can stop running the propagator henceforth, since it cannot prune any more values until we next backtrack (since domains can only get smaller except for backtracking). We now describe a sufficient condition for UM-3 to be entailed:

Definition 4. *A propagator is entailed by domains $D = \{d_1, \dots, d_n\}$ when any set of domains that are subsets of these, i.e., any $E = \{e_1, \dots, e_n\}$ s.t. $\forall i. e_i \subseteq d_i$, are at a fixed point (see [SC06]).*

UM-3 constraint becomes entailed as soon as two domains contain single values:

Theorem 2. *UM-3- $BC(Z)$ becomes entailed as soon as two domains are singleton.*

Proof. Consider the possible scenarios: either the two singletons are distinct or they are the same. The domains before propagation are shown in Figure 6 as boxes; the domains after propagation are shaded gray. Clearly all remaining choices for the third variable are

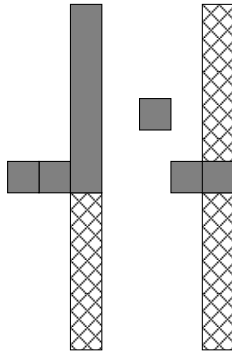


Figure 6: Propagation done when 2 domains are singleton

valid instantiations and since the propagation algorithm is safe they cannot be removed by further propagation and by definition the propagation will be at a fixed point. \square

3.4 Ultrametric matrix constraint

The supertree model presented in Section 4 makes use of the ultrametric constraint, however in this context the desired end product is to constrain a whole matrix to be an *ultrametric matrix*, and not merely to constrain three variables.

Definition 5. *A symmetric matrix M is an ultrametric matrix if and only if for every set of three distinct indices i, j and k , there is a tie for the minimum of M_{ij} , M_{ik} and M_{jk} ; and $M_{ii} = 0$ for all i .*

The ultrametric matrix constraint can be achieved for matrix M by posting the constraint $\text{UM-3}(M_{ij}, M_{ik}, M_{jk})$ over all choices of distinct i, j and k , but at a cost of introducing $\binom{n}{3}$ constraints. In practical constraint solvers any model containing this constraint will have $\Theta(n^3)$ space complexity. However, when a domain event occurs for a matrix variable M_{ij} it is straightforward to iterate over all k indices doing the same propagation as $\text{UM-3-BC}(Z)$ from Figure 3. This replaces a $\Theta(n^3)$ space list representation of the set of UM-3 constraints by a $\Theta(1)$ code representation. Hence we propose the ultrametric matrix constraint propagator $\text{UM-MATRIX-BC}(Z)$ in Figure 7.

This propagator mimics part of the AC-3 algorithm[Mac77] since it (a) receives a propagation event on a variable, (b) identifies which constraints are over that variable, and (c) arranges for the propagation to be carried out. However any events caused as a result are queued and dispatched by the underlying propagator as normal. Each variable can be involved in up to $n - 2$ constraints, since each variable has *two* indices in the matrix and we have a constraint involving each choice of three different indices.

The algorithm propagates in $\Theta(n)$ time, which is more expensive per event than using $\binom{n}{3}$ UM-3 constraints, but a factor of n fewer propagators wake up as a result of each event.

Code style: **let** $(i, j) \leftarrow \text{index}(v)$ declares i and j to be the indices of variable v in the matrix M that the constraint is over.

```

Algorithm UM-MATRIX-BC(Z)


---


Min event( $v$ )
A1  let  $(i, j) \leftarrow \text{index}(v)$ 
A2  for  $k \leftarrow 1 \dots n$  do
A2.1 if  $k \neq i$  and  $k \neq j$  then
A2.2     Min event( $M_{ij}, M_{ik}, M_{jk}$ )
A2.3     Max event( $M_{ij}, M_{ik}, M_{jk}$ )


---


Max event( $v$ )
A3  let  $(i, j) \leftarrow \text{index}(v)$ 
A4  for  $k \leftarrow 1 \dots n$  do
A4.1 if  $k \neq i$  and  $k \neq j$  then
A4.2     Max event( $M_{ij}, M_{ik}, M_{jk}$ )


---


Fix event( $v$ )
A5  let  $(i, j) \leftarrow \text{index}(v)$ 
A6  for  $k \leftarrow 1 \dots n$  do
A6.1 if  $k \neq i$  and  $k \neq j$  then
A6.2     Min event( $M_{ij}, M_{ik}, M_{jk}$ )
A6.3     Max event( $M_{ij}, M_{ik}, M_{jk}$ )


---



```

Figure 7: Algorithm for UM matrix propagator

4. Supertree Construction

We now review imperative solutions to the supertree construction problem, the first constraint programming solution [GPSW03], and present a new encoding that exploits the specialised UM-MATRIX-BC(Z) constraint.

4.1 Imperative solutions to the supertree problem

The earliest imperative techniques are due to Bryant and Steel [BS95] and Ng and Wormald [NW96]. Both present a ONETREE algorithm which is based on the BUILD algorithm of [ASSU81]. ONETREE is based on the observation that in a tree any three leaf nodes define a unique relation with respect to their most recent common ancestor (mrca), such that $mrca(a, b)$ is the interior node furthest from the root that has both leaf nodes a and b as descendants. We abuse notation by writing $mrca(a, b) > mrca(c, d)$ when the former has a greater depth than the latter, and similarly $mrca(a, b) = mrca(c, d)$ if they have the same depth. Given three different leaf nodes/species (labelled a , b , and c) one of the following four relations must hold:

- (1) $mrca(a, b) > mrca(a, c) = mrca(b, c)$
- (2) $mrca(a, c) > mrca(a, b) = mrca(c, b)$
- (3) $mrca(b, c) > mrca(b, a) = mrca(c, a)$
- (4) $mrca(a, b) = mrca(a, c) = mrca(b, c)$

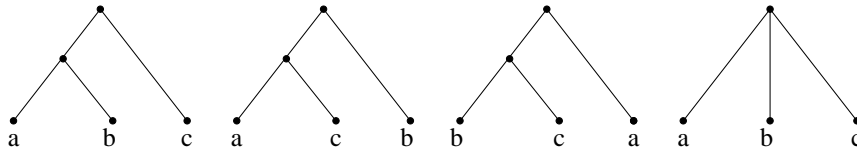


Figure 8: The four possible relationships between three leaf nodes in a tree: i.e. the three triples $(ab)c$, $(ac)b$, and $(bc)a$, and the fan (abc) .

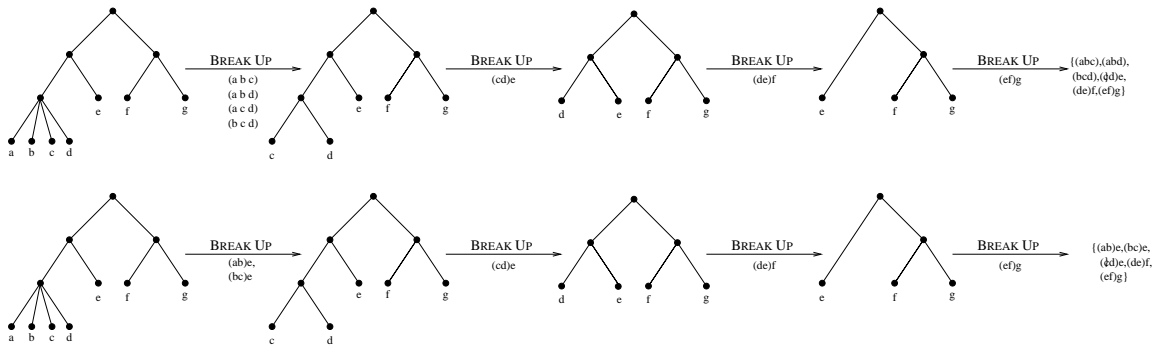


Figure 9: Example execution of the BREAKUP algorithm. On the top, a hard breakup, and below a soft breakup (no fans produced)

This is shown in Figure 8. Using the terminology from [NW96] we can say that in (1), (2) and (3) we have the *triples* $(ab)c$, $(ac)b$, and $(bc)a^2$ and in (4) we have the *fan* (abc) , i.e., in a fan the relationship between these leaves is unresolved. Prior to applying the ONETREE algorithm two (or more) species trees are broken up into triples and fans via the BREAKUP algorithm [NW96], resulting in a linear sized encoding of those trees. The supertree is then constructed (if possible) using this encoding as input.

Figure 9 shows an example of the BREAKUP algorithm process. Two variants of the process are shown; at the top we have a hard breakup, where fans are considered as hard evidence (i.e., hard polytomies) and must be respected (as in [NW96]), and below a soft breakup where fans are taken as a lack of evidence (i.e. soft polytomies, as in [BS95]). For hard breakup the algorithm [NW96] is modified such that when encountering a k -fan this is broken up into $\binom{n}{3}$ 3-fans, and in a soft breakup a fan is broken into a linear number of rooted triples. Algorithms for hard and soft breakups are given in Figures 10 and 11, and are used by the imperative ONETREE algorithm here and the constraint programming models.

A toy set of input triples and single solution are shown in Figure 12. The triples have been drawn to reflect how the solution is compatible with them.

2. ... where $(xy)z$ can be read as “x is closer to y than z”

Code style: The function `sortedInteriorNodes(T)` delivers the set of interior nodes of the tree T in non-increasing order of depth in that tree; `degree(v)` delivers the out degree of node v ; function `child(v, i)` delivers the i^{th} child of interior node v ; `uncleOrCousin(l)` delivers a leaf node that is descended from a sibling of the parent of leaf node l ; function `becomesLeaf(v, l)` transforms interior node v into a leaf node labelled as l ; `removeChild(l, v)` removes the leaf node l from the list of children of interior node v .

Algorithm **HARDBREAKUP**

HardBreakup(T)

```

1  let  $V \leftarrow \text{sortedInteriorNodes}(T)$ 
2  let  $S \leftarrow \emptyset$ 
3  let  $i \leftarrow 0$ 
4  while  $\text{notRoot}(V[i]) \vee \text{degree}(V[i]) > 2$  do
5      let  $v \leftarrow V[i]$ 
6      let  $c_0 \leftarrow \text{child}(v, 0)$ 
7      if  $\text{degree}(v) = 2$ 
8          then let  $c_1 \leftarrow \text{child}(v, 1)$ 
9              let  $c_2 \leftarrow \text{uncleOrCousinOf}(c_0)$ 
10              $S \leftarrow S \cup \{\text{triple}(c_0, c_1, c_2)\}$ 
11              $v \leftarrow \text{becomesLeaf}(v, c_0)$ 
12              $i \leftarrow i + 1$ 
13         else for  $j \leftarrow 1$  to  $\text{degree}(v) - 2$  do
14             for  $k \leftarrow j + 1$  to  $\text{degree}(v) - 1$  do
15                 let  $c_1 \leftarrow \text{child}(v, j)$ 
16                 let  $c_2 \leftarrow \text{child}(v, k)$ 
17                  $S \leftarrow S \cup \{\text{fan}(c_0, c_1, c_2)\}$ 
18                  $v \leftarrow \text{removeChild}(c_0, v)$ 
19  return  $S$ 

```

Figure 10: Hard breakup of a tree T , producing triples and fans.

Algorithm **SOFTBREAKUP**

SoftBreakup(T)

```

1  let  $V \leftarrow \text{sortedInteriorNodes}(T)$ 
2  let  $S \leftarrow \emptyset$ 
3  let  $i \leftarrow 0$ 
4  while  $\text{notRoot}(V[i])$  do
5      let  $v \leftarrow V[i]$ 
6      let  $c_0 \leftarrow \text{child}(v, 0)$ 
7      let  $c_1 \leftarrow \text{child}(v, 1)$ 
8      let  $c_2 \leftarrow \text{uncleOrCousinOf}(c_0)$ 
9       $S \leftarrow S \cup \{\text{triple}(c_0, c_1, c_2)\}$ 
10     if  $\text{degree}(v) = 2$ 
11         then  $v \leftarrow \text{becomesLeaf}(v, c_0)$ 
12          $i \leftarrow i + 1$ 
13     else  $v \leftarrow \text{removeChild}(c_0, v)$ 
14  return  $S$ 

```

Figure 11: Soft breakup of a tree T , producing only triples.

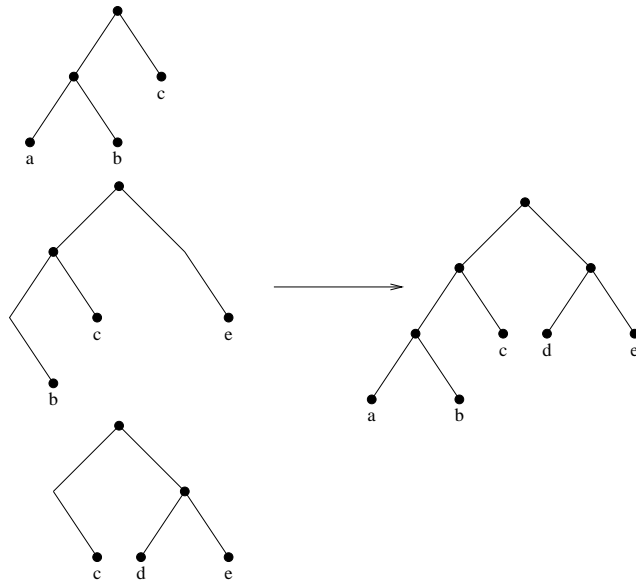


Figure 12: A toy input (left) and single solution to the supertree problem (right). Input trees are distorted to make relationships in resultant supertree more obvious.

The complexity of ONETREE is given in [NW96] as $O(h(n))$ where $h(n) = n(n + t + bn)\alpha(n + t + f)$, n is the number of labels, t the number of triples, f the number of fans, b is the sum of the squares of the number of leaves in the fans, and α is the inverse Ackermann function (and is less than 4 for all conceivable inputs and so behaves like a constant). Therefore if the input trees are fully resolved (i.e., have no fans) complexity is $O(n^2)$ but in the worst case complexity grows to $O(n^4)$. This should be contrasted with the complexity of Bryant and Steel's ONETREE[BS95], and that is $O(t.n)$.

4.2 A Constraint Encoding using Toolkit Constraints

This second stage, i.e., ONETREE equivalent, was first proposed as a constraint program in [GPSW03]. The encoding takes advantage of an equivalence between *ultrametric trees* and *ultrametric matrices*:

Definition 6. Let M be a real symmetric $n \times n$ matrix. An ultrametric tree for M is a rooted tree T such that:

1. T has n leaves, each corresponding to a unique row of M ;
2. each internal node of T has at least 2 children;
3. for any two leaves i and j , M_{ij} is the label of the most recent common ancestor of i and j ; and
4. along any path from the root to a leaf, the labels strictly increase.

Theorem 3. *A symmetric matrix M has an ultrametric tree T if and only if it is an ultrametric matrix. Furthermore, the tree T uniquely determines the matrix M and the matrix M uniquely determines the tree T .*

Proof. A proof is given in [Gus97]. □

There is a clear correspondence between Definition 6 and the description of a species tree given in Section 4: a species tree T is an ultrametric tree for matrix M , where M_{ij} is the depth of the mrca of species i and j or M_{ij} is the divergence date of those two species. For this reason we can use an ultrametric matrix model to solve the supertree problem.

4.2.1 THE MODEL OF [GPSW03]

Given as input a forest F with n distinct leaf labels, a symmetric $n \times n$ matrix M of constrained integer variables is created with domains $\{1, \dots, n-1\}$ or $\{0\}$ on the main diagonal. Variable M_{ij} is the depth of the mrca of species i and j . Initially, constraints are posted to make the whole matrix ultrametric thus ensuring that any resulting tree is ultrametric:

$$\begin{aligned} & M_{ij} > M_{ik} = M_{jk} \\ \vee & M_{ik} > M_{ij} = M_{jk} \\ \vee & M_{jk} > M_{ij} = M_{ik} \\ \vee & M_{ij} = M_{ik} = M_{jk} \end{aligned} \tag{2}$$

for each $i < j < k$. The input trees are then broken up into triples and fans using either of the breakup algorithms of Figures 10 and 11. For each triple $(ij)k$ produced the constraint

$$M_{ij} > M_{ik} = M_{jk} \tag{3}$$

is posted and for each 3-fan (ijk)

$$M_{ij} = M_{ik} = M_{jk} \tag{4}$$

is posted. These constraints break the disjunctions of Equation 2. The model has $(n^2 - n)/2$ variables and

$$t + f + \binom{n}{3} = O(n^3) + O(n^3) + \Theta(n^3) = \Theta(n^3) \tag{5}$$

constraints, where t is the number of triples and f the number of fans. There are $O(n^3)$ of each because each one breaks the disjunction in at most one constraint from Equation 2, and there are $\Theta(n^3)$ of those.

4.2.2 CONVERTING BACK TO TREE REPRESENTATION

The final step is to use an algorithm based on the constructive proof of the \Leftarrow direction of Theorem 3 of [Gus97] to produce a tree from the matrix M produced by a constraint solver from the above model. We will not describe this algorithm in detail, but for the sake of intuition it works by

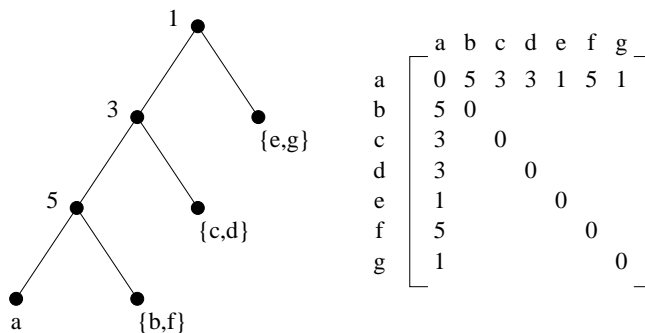


Figure 13: One iteration of an algorithm to convert an ultrametric matrix to a tree

- Picking an arbitrary leaf s . Let the number of distinct entries in row s be d .
- Partition the other leaves into sets p_1, \dots, p_d based on their entry in row s .
- Solve the problem recursively on each p_i by ignoring all the rows and columns in the matrix D not in p_i .
- Combine into overall solution by attaching subproblem solutions at the correct depth on the path to s .

Figure 13 shows one recursion of the algorithm with a choice of leaf a and shows that row a fully describes the path to a in the corresponding tree.

4.2.3 TIME COMPLEXITY OF THE MODEL OF [GPSW03]

BREAKUP, and the procedures to build a constraint model and to convert an ultrametric matrix to a tree are all polynomial time. However the complexity of backtracking search over $O(n^2)$ variables with $O(n)$ -size domains is worst case $O(n^{n^2})$. This is an upper bound on the time taken to solve the supertree problem. We have not attempted to derive a lesser upper bound on the time complexity, since, as we will show in the following section, our new model has provably achieved a polynomial time bound.

4.3 A constraint encoding using the new propagator design

This issue of potentially exponential solution time in the [GPSW03] model is worrying, but in our experiments the time taken to solve instances has not been a major issue. Conversely the memory requirements are a problem in practice, but not in theory! The model requires $\Theta(n^3)$ space for n species, but the constant factor is inhibiting. Posting the constraint of Equation 1 literally (using toolkit propagators) as described in Section 3 uses 23 propagators in the JCHOCO toolkit. This requires roughly 23 times the runtime memory of a single propagator, since each corresponds to a single JAVA object, and each of these have comparable footprints. As we will show in the empirical study of Section 5, this prevents modest instances from being loaded on typical current workstations.

Using the new propagator of Section 3 we replace these $\binom{n}{3}$ propagators with a single compact propagator and as a result memory usage is reduced asymptotically from $\Theta(n^3)$

Algorithm CPBUILD

```

CPBuild( $F$ )
1  let ( $V, D, C$ )  $\leftarrow$  CPMoDel( $F$ )
2  for  $T \in F$  do
3      for  $t \in \text{BreakUp}(T)$  do  $C \leftarrow \text{post}(t, C)$ 
4  if propagate( $V, D, C$ ) then return UMTToTree( $V, D$ )
5  else fail()

```

Figure 14: Build a supertree from forest F using the ultrametric constraint model.

to $\Theta(n^2)$ since now the model memory is dominated by the $\Theta(n^2)$ space needed for the matrix M . Also reducing the amount of space to be initialised delivers a proportional saving in build time. But most importantly, using the new constraint provides a solution to exponential time complexity, because enforcing $\text{BC}(Z)$ on the model allows a solution to be read out of the lower bound of each domain. Theorem 4 is a proof of correctness for this algorithm.

Figure 14 gives a schema for the constraint programming algorithm for supertree construction, CPBUILD. The algorithm takes as input a forest F of trees. In line 1 a constraint model is produced, i.e., an $n \times n$ symmetric array of constrained integers variables is created, where there are n unique species in the forest, and the $\text{UM-MATRIX-BC}(Z)$ constraint is then posted over those variables. Lines 2 and 3 breaks these input trees into their triples and fans using either of the breakup algorithms given in Figures 10 and 11, and posts them into the model as constraints. Propagators for the constraints are executed to a fixed point in line 4; if this succeeds a tree is created from the lower bounds of the ultrametric matrix otherwise we fail.

Lemma 7. *If the propagator for every constraint in a model enforces $\text{BC}(Z)$ and furthermore the lower bounds are mutually supportive, then after executing all propagators to a fixed point, either the lower bounds are a solution, or we have an empty domain and fail.*

Proof. If we reduce each domain to just the lower bound after a fixed point is obtained then each bound is supported because they are mutually supportive by supposition. Hence every constraint is simultaneously satisfied by these singleton domains and, by definition, we have a solution. \square

Theorem 4. *CPBUILD is a polynomial time solution to the supertree problem.*

Proof. The only constraints involved in the model are those for triples and fans and ultrametric constraints. By Theorem 1 we know that all lower bounds are supported after the propagators run, and by Lemma 1 we know that the lower bounds are mutually supportive. The same is true of the disjunction breaking propagators. Hence by Lemma 7, and as shown in Figure 14, we can either read out a solution or fail. We can enforce $\text{BC}(Z)$ on the problem in polynomial time as shown below. \square

Immediate from this is that we can preserve the polynomial time solution with the addition of a polynomial number of side-constraints, so long as these additional constraints preserve the property that lower bounds are mutually supportive. In fact, CSPs such as these with ordered domains where all constraints have the property that lower bounds are

mutually supportive belong to a known tractable class called min-closed. See [JC95] for more details.

4.3.1 TIME COMPLEXITY OF CPBUILD

The algorithm can be implemented to run in $O(n^4)$ time using a variation on the AC3 algorithm. AC3[Mac77] begins with a queue containing all constraints. It repeatedly removes a constraint until none remain and runs the associated propagator. Any constraints over affected variables are re-queued, if necessary. Once the queue empties, all propagators are at a fixed point. We need $O(n^3)$ constraints so the worst case complexity is

$$\underbrace{O(n^3)}_{\text{build initial Q}} + \underbrace{O(n)O(n^3)}_{\text{worst case re-queues with 1 value removed at a time}} \cdot \underbrace{O(1)}_{\text{propagation time}}$$

or $O(n^4)$ overall. This matches the worst case complexity of ONETREE [NW96]. Our constraint solution has its worst case when the problem is unsolvable, since when it is unsolvable domains are *emptied* by propagation, whereas for solvable instances propagation reaches a fixed point sooner. This is the opposite of the imperative solution referred to earlier, which discover inconsistencies during the build process and hence unsolvable instances finish sooner.

5. Empirical Study

We present an empirical study to determine if any practical improvements have been achieved in constraint solutions to the supertree problem and, if so, what size of improvement. Experiments were run using a 1.7GHz Pentium 4 processor with 768MB of memory, using Sun Java build 1.5.0_06-b05. The constraint toolkit used was JChoco version 1.1.04. Input trees were broken up using a hard breakup, consequently in all cases fans were treated as hard polytomies³.

Real-life seabird data from [KP02] is our benchmark (also used in [BFL08]) and we present statistics on various techniques for producing supertrees, namely ONETREE, and the CP solutions of Section 4 (entries Toolkit and Matrix). For completeness we reproduce the recent results reported in [BFL08] over the same data set, and tabulate this as TREECON. The TREECON results were encoded in the same constraint programming toolkit as ours but on a processor that was approximately twice as fast (3GHz). We do not correct the times to compensate for this factor. We mark in **bold** results that differ very significantly between the Matrix and TREECON results, specifically those whose runtimes would undoubtedly be a factor of 10 different on the same processor. Results are reported for combinations of seabird trees (input trees named A to G) and the following data is tabulated below:

Data The combination attempted.

n Total distinct species in input trees

Technique Type of algorithm used to solve

Sol T iff supertree is possible

3. In a later section we use soft breakup.

Build Time in milliseconds to initialise CP model

Solve Time in milliseconds to first solution, if any

Total = Build + Solve

Nodes Number of nodes in search tree

Mem Model memory in MB

In the table, DNL means that the model could not be loaded and DNF means that it could not be solved within 30 mins, but succeeded in loading. We have not provided the memory usage of ONETREE; however it is smaller than that of any of the constraint encodings.

Data	n	Technique	Sol	Build	Solve	Total	Nodes	Mem
AB	23	Toolkit	T	2056	374	2430	23	26.92
		Matrix		183	131	314	23	0.24
		TreeCon				302		
		ONETREE				13		
AC	32	Toolkit	F	2670	327	2997	0	36.34
		Matrix		189	153	342	0	0.34
		TreeCon				406		
		ONETREE				12		
AD	47	Toolkit	T	8235	946	9181	38	118.51
		Matrix		220	248	468	38	0.70
		TreeCon				398		
		ONETREE				22		
AE	95	Toolkit	F	DNL	DNL	DNL	DNL	> 629
		Matrix		340	1477	1817	0	2.79
		TreeCon				10393		
		ONETREE				37		
AF	31	Toolkit	T	2497	379	2876	19	32.99
		Matrix		188	137	325	18	0.32
		TreeCon				127		
		ONETREE				20		
AG	46	Toolkit	T	7671	871	8542	31	111.07
		Matrix		222	252	474	31	0.68
		TreeCon				409		
		ONETREE				21		
BC	29	Toolkit	F	2056	21931	23987	171	26.90
		Matrix		171	107	278	0	0.27
		TreeCon				32		
		ONETREE				8		
BD	42	Toolkit	T	5833	930	6763	33	84.26
		Matrix		201	251	452	33	0.55
		TreeCon				301		
		ONETREE				17		
BE	94	Toolkit	F	DNL	DNL	DNL	DNL	> 629
		Matrix		335	16340	16675	0	2.71
		TreeCon				892		
		ONETREE				11		
BF	30	Toolkit	T	2405	343	2748	29	29.83
		Matrix		174	99	273	29	0.28
		TreeCon				144		
		ONETREE				8		
BG	40	Toolkit	T	5098	651	5749	30	72.71
		Matrix		203	353	556	30	0.51
		TreeCon				1440		
		ONETREE				13		
CD	45	Toolkit	T	10056	1134	11190	45	143.91
		Matrix		224	276	500	45	0.77
		TreeCon				630		
		ONETREE				14		

CE	68	Toolkit Matrix TreeCon ONETREE	T	DNL 516	DNL 1451	DNL 1967 27180 36	DNL 68	> 629 2.72
CF	34	Toolkit Matrix TreeCon ONETREE	T	3101 180	563 133	3662 313 393 11	30 30	43.72 0.36
CG	44	Toolkit Matrix TreeCon ONETREE	F	6683 210	587 215	7270 425 1530 14	0 0	97.10 0.61
DE	104	Toolkit Matrix TreeCon ONETREE	F	DNL 360	DNL 2021	DNL 2381 1126 34	DNL 0	> 629 3.31
DF	44	Toolkit Matrix TreeCon ONETREE	T	6613 203	987 250	7600 453 630 17	37 37	97.10 0.60
DG	56	Toolkit Matrix TreeCon ONETREE	F	14090 252	2280 640	16370 892 910 19	1 0	201.42 0.99
EF	94	Toolkit Matrix TreeCon ONETREE	F	DNL 331	DNL 9546	DNL 9877 1035 12	DNL 0	> 629 2.71
EG	97	Toolkit Matrix TreeCon ONETREE	F	DNL 344	DNL 8900	DNL 9244 1211 15	DNL 0	> 629 2.89
FG	38	Toolkit Matrix TreeCon ONETREE	F	4299 195	DNL 212	DNL 407 62 10	DNL 0	61.41 0.46
ABDF	72	Toolkit Matrix TreeCon ONETREE	T	27032 277	5291 722	32323 999 8139 34	63 59	382.52 1.48
ABDG	78	Toolkit Matrix TreeCon ONETREE	F	60847 301	DNF 3633	DNF 3934 347 29	DNF 0	553.49 1.91
ACDF	72	Toolkit Matrix TreeCon ONETREE	F	31067 286	1931 649	32998 935 8690 28	0 0	434.84 1.61
ACDG	81	Toolkit Matrix TreeCon ONETREE	F	DNL 307	DNL 1711	DNL 2018 12650 35	DNL 0	> 629 2.06
ACE	97	Toolkit Matrix ONETREE	F	DNL 737	DNL 1632	DNL 2369 38	DNL 0	> 629 2.91

The most obvious thing to note is how much faster the imperative approach is compared to the constraint techniques. Why is this? Primarily it is due to the lower complexity of ONETREE in the absence of fans, and partly due to the generality of the constraint programming approach. The imperative approach is highly specialised to only one class of problem whereas the constraint approach sits within a toolkit, and runs on top of a

general purpose constraint maintenance system. We should not expect that the constraint approach will compete in raw speed but what we later demonstrate (in Section 6) is that the approach benefits from its versatility, i.e., the costs of space and time is repaid by the ease of accommodating variants of the problem into the same model.

The most impressive aspect of the matrix constraint over that of [GPSW03] is the improvement in memory requirements, so that all instances can now be loaded comfortably. This also has a dramatic impact on the build time. These improvements dominate the reduction in solve time in practice. The toolkit model is outperformed by the matrix model by an order of magnitude on each instance; however there are two cases of search occurring in the toolkit model, i.e., data sets BC and ABDG.

Our results also compare well against those of [BFL08]. There is one case, BE, where Matrix is an order of magnitude slower than TREECON, and we attribute this to $O(n^2)$ variables using in the matrix model compared to the $O(n)$ variables used in TREECON, and four cases when TREECON is significantly worse. No results are available for TREECON over the data set ACE. It should be noted that in [BFL08] it is stated that they do not yet have a complete filtering algorithm for this problem based on their constraint model and, via personal communication, that although the TREECON model never backtracked over the birds data set there is as yet no proof that the complexity of their model is polynomial.

Figure 15 shows the supertree produced from the largest compatible forest $\{A, B, D, F\}$. This supertree has 72 leaves and takes about 1 second to produce. Although the result is not printed in the table, finding the forest $\{A, B, C, D, E, F, G\}$ incompatible takes about 12 seconds in total (1.5 seconds to build the model and 10 seconds to determine incompatibility).

6. Versatility of the Constraint Model

One of the strengths of constraint programming is its versatility: given a constraint model of a core problem this model can then be enhanced to address variants of the original pure problem. We demonstrate this versatility with respect to the ultrametric model, presenting four variants of the supertree problem (a) incorporating ancestral divergence dates into the model, (b) nested taxa, (c) determining if an induced triple or fan is common to all supertrees, and (d) coping with incompatibilities.

6.1 Ancestral Divergence Dates

In [SDH⁺04, BSS04] temporal information is added to the input trees. Interior nodes may be labelled with integer *ranks* such that if interior node v_2 is a proper descendant of v_1 then $rank(v_1) < rank(v_2)$, resulting in a ranked phylogenetic tree. Additionally relative divergence dates may be expressed in the form “div(c,d) *predates* div(a,b)” and this is interpreted as “the divergence of species c and d predates that of species a and b ”. The RANKEDTREE algorithm proposed in [BSS04] takes as input a collection of precedence constraints derived from input ranked species trees and *predates* relations. The algorithm outputs a ranked tree that respects those relations or returns “not compatible”.

This is trivial to incorporate into the constraint model. If trees have been ranked then for each pair of species (i, j) in the leaf set we instantiate the constraint integer variable M_{ij} to the value of $mrca(i, j)$. For a *predates* relation “div(c,d) *predates* div(a,b)” we post

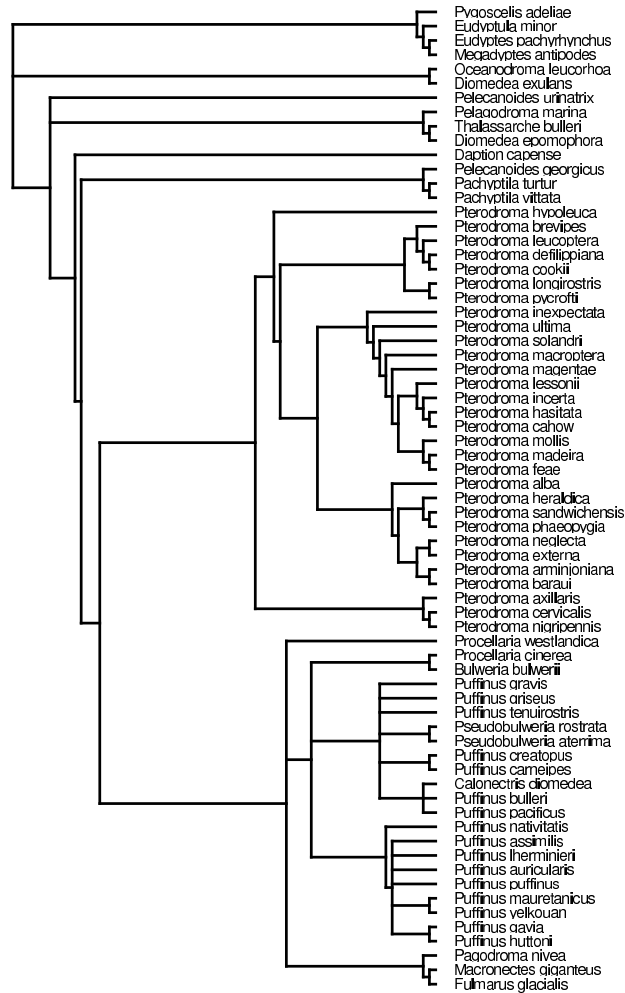


Figure 15: Supertree with largest compatible data sets of birds ABDF. This took 737ms to model and 270ms to solve using CPBUILD. Displayed using Rod Page's treeView [Pag96].

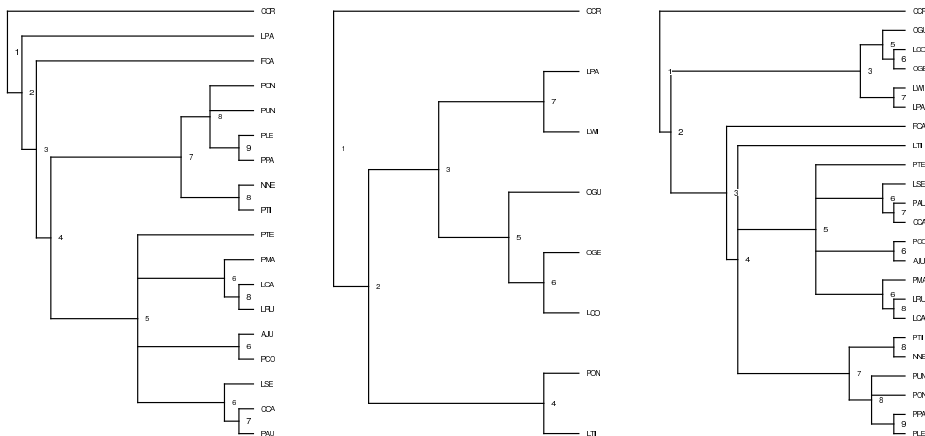


Figure 16: Two ranked trees of cats taken from [SDH⁺04] and on the right one of the 17 possible supertrees produced by CPBUILD. Displayed using Rod Page’s treeView [Pag96].

the constraint $M_{cd} < M_{ab}$. This is done before step 4 of CPBUILD (Figure 14), i.e., ranks and *predates* relations become side constraints. Similarly time bounds on speciation events are posted as unary constraints, i.e. in a *dated phylogenetic tree* upper and lower divergence bounds are given on interior nodes, such that $l(a, b)$ and $u(a, b)$ give respectively the lower and upper bounds on the divergence dates of species a and b . In the constraint program the following two side constraints are then posted (again, before step 4): $l(a, b) \leq M_{ab}$ and $M_{ab} \leq u(a, b)$.

A demonstration of ranked trees is given in Figure 16. On the left we have two ranked species trees of cats taken from [SDH⁺04] and originally from [JMSO95], where cat names were given three-letter abbreviations. The branch lengths of the source trees have been translated into rankings and added to the interior vertices of those trees. On the right we have one of the 17 possible resultant supertrees. In total, 7 of the 17 solutions contain interior nodes with ranges. Without this 30 solutions are produced, some of which are identical but for the divergence dates. This goes some way to addressing the issue of enumerating all supertrees compactly, raised in [SDH⁺04]. In Figure 17 we show the effect of adding a *predates* constraint to a supertree construction. The data used is from [BSS04], Figures 5 and 6.

6.2 Nested taxa

A *taxon* (plural, *taxa*) is a group of organisms comprising a single common ancestor and its descendents [DW04]. For example the species “lion” and the class “birds” are taxa. So far, all our species trees have been leaf labelled, however this is restrictive because trees may be annotated with taxa names on both leaves and internal nodes, giving *nested taxa*. For example, Figure 18 shows tree T_1 with an internal node labelled P which has descendents a and b , i.e., the a and b taxa are nested within the P taxon. Problems related to creating compatible supertrees for this type of data were raised in [Pag04] and defined and solved in

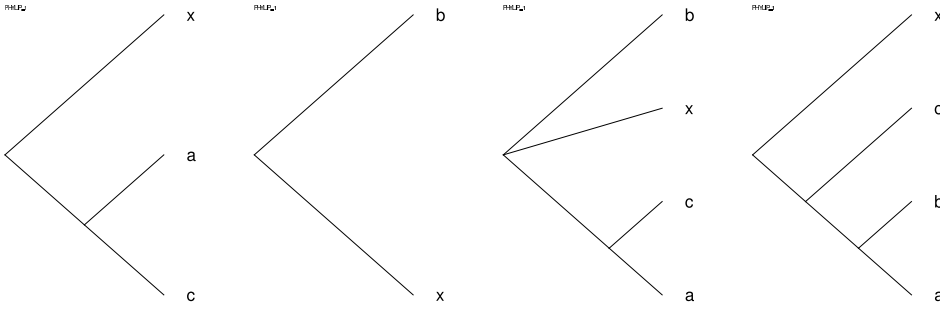


Figure 17: The two input trees from Figure 5 of [BSS04] $T_1 = ((a, c), x)$ and $T_2 = (b, x)$ with a resultant supertree shown in the 3rd position. The tree on the far right is a supertree from T_1 and T_2 with the side constraint “ $\text{div}(a,c)$ predates $\text{div}(a,b)$ ”, produced by CPBuild and displayed using Rod Page’s treeView [Pag96].

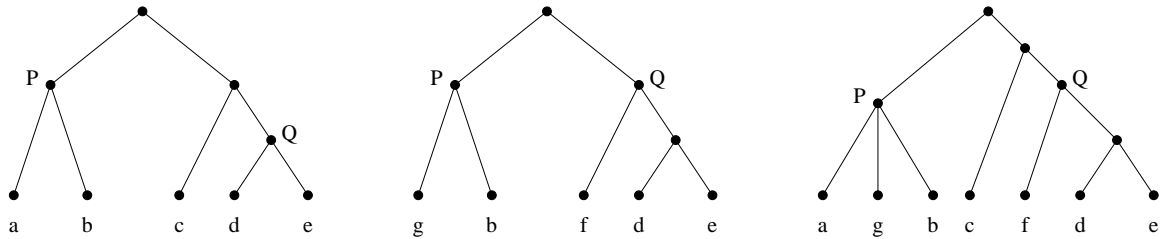


Figure 18: Two input rooted X -trees T_1 and T_2 (left) and an output tree T_3 (right) that perfectly displays them.

[DS04]. A set of input trees and possible solution to the problem are shown in Figure 18: notice that all labels are conserved in the solution, all ancestral relationships are conserved and, for any labels l_i and l_j from the same input tree, l_i is ancestor of l_j in the input tree if and only if l_i is ancestor of l_j in the solution tree. This is an instance of the problem *Higher Taxa Compatibility* defined in [DS04] and [SDH⁺04], where the result tree must *perfectly display* all of the input trees. We now define the problem more formally.

Definition 7. A rooted X -tree [DS04] is a species tree where internal nodes as well as leaves may be labelled from the set X .

In the following we will be slightly loose and may use a label l to identify the labelled node, as well as the label itself, e.g., “descendants of l ” means the same as “descendants of the node labelled l ”.

Definition 8. A rooted X -tree T perfectly displays a rooted X' -tree T' when

1. $T' \subseteq T$;
2. T' displays T , neglecting internal labels;
3. if a is a descendant of b in T' then a is a descendant of b in T ; and

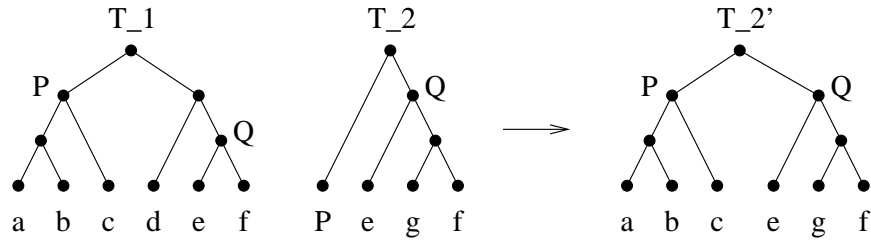


Figure 19: Two input trees T_1 and T_2 with an enclosing taxon P . By a process of substitution T_2 is replaced with T'_2 .

- 4. if a is not a descendant of b in T' then a is not a descendant of b in T .

A rooted X -tree T perfectly displays a forest of phylogenetic trees $F = \{T_1, \dots, T_n\}$ when it perfectly displays every T_i .

6.2.1 CONSTRAINT ENCODING

Our constraint encoding is implemented by the addition of variables and side constraints to the standard model of Section 4. We describe how to transform the input to make the constraint solution simpler, and then describe the variables and constraints needed.

Substitution Taxon P in Figure 19 appears on an internal node of T_1 , we will call such a label an *enclosing taxon*. Note also that it appears *on a leaf* in T_2 . The input trees are preprocessed to replace any tree with an enclosing taxon P on a leaf by the same tree with any single subtree rooted at P substituted in its place. There must be such a subtree elsewhere in the input forest, or we have a contradiction that M is an enclosing taxon. This process does not add or remove any information, since the relationships between M and everything in the tree still holds, and the new relationships between the taxa in the subtree at M and the rest of the tree were always implicit in the input.

The aim of this process was just to obtain a set of input where enclosing taxa appear on internal nodes only, because without loss of generality our constraint encoding assumes that this is the case. Figure 19 shows an example of the substitution process applied to trees T_1 and T_2 , and T_2 would be replaced with T'_2 .

Variables and constraints The variables added are one integer variable v_l per enclosing taxa/label l , each with a domain of $\{1, \dots, n - 1\}$. The value of this variable in a solution is the tree depth of the internal node which it labels, and the label's position in the final tree is determined because it must be at the unique node of that depth on a path from one of its nested taxa to the root. See Figure 13 and suppose for the sake of argument that we have an enclosing taxa M which labels b , and the variable $l_M = 1$ in a solution. The unique location where the label M can go is at the root node.

Properties (1) and (2) from Definition 8 above are immediate from the properties of the earlier model which is the foundation for this one. Before explaining how to enforce property (3) we introduce some notations for convenience. Function $desc(l, F)$ returns the

set of all descendants of label l in any tree $T \in F$, and $notDesc(l, T)$ returns the set of labels that are not descendants of l in tree T .

We first need a constraint that l must label every single species that it labels in an input. For every enclosing label l , post the following set of constraints:

$$\{v_l \leq M_{ij} \mid i \in desc(l, F) \wedge j \in desc(l, F) \wedge i \neq j\} \quad (6)$$

so that the label must settle at least as shallow as any mrca of its descendants in input, and hence they must remain descendants. Notice that it is necessary to consider pairs of species from distinct input trees. An alternative of taking pairs from the same tree does not work, because it is necessary for all pairs to be under the *same* internal node l , rather than two distinct nodes that happen to be at the correct depth. Next, the label must be constrained so that no label that was not already a descendant becomes one. For each X -tree T and enclosing label $l \in X$, post the following set of constraints:

$$\{v_l > M_{ij} \mid i \in desc(l, \{T\}) \wedge j \in notDesc(l, T)\} \quad (7)$$

so that the label l must be placed strictly deeper than any mrca of a descendant and something that's not a descendant, i.e., no non-descendants of l can be a descendant in the result. As an illustration we list the generated constraints for the example of Figure 18.

1. Equation 6 and $l = P$: $\{v_P \leq M_{ab}, v_P \leq M_{ag}, v_P \leq M_{bg}\}$
2. Equation 6 and $l = Q$: $\{v_Q \leq M_{de}, v_Q \leq M_{df}, v_Q \leq M_{ef}\}$
3. Equation 7, $l = P$ and $T \in \{T_1, T_2\}$: $\{v_P > M_{ac}, v_P > M_{ad}, v_P > M_{ae}, v_P > M_{bc}, v_P > M_{bd}, v_P > M_{be}, v_P > M_{gf}, v_P > M_{gd}, v_P > M_{ge}, v_P > M_{bf}\}$
4. Equation 7, $l = Q$ and $T \in \{T_1, T_2\}$: $\{v_Q > M_{ad}, v_Q > M_{ae}, v_Q > M_{bd}, v_Q > M_{be}, v_P > M_{cd}, v_Q > M_{ce}, v_Q > M_{gf}, v_P > M_{bf}, v_P > M_{gd}, v_P > M_{ge}\}$

The number of new constraints created by both Equations 6 and 7 is bounded by the number of distinct pairs of species, i.e. $\Theta(n^2)$ new constraints.

6.3 Necessity

There may be many possible supertrees for a given input forest. One question is then, what relationships are common to all supertrees? The problem of determining if a derived induced triple (or fan) in a supertree is necessary (i.e., common to all possible supertrees) is introduced in [Dan03] along with the polynomial time decision procedure NECESSITY.

The algorithm NECESSITY in Figure 20 takes as arguments a forest F of trees, assumed to be compatible, and a rooted triple or fan τ and determines if τ occurs in every supertree that displays the trees in F . The algorithm is a simple modification of CPBUILD, where lines 1 to 3 are essentially the same. In line 4 the negation of the triple τ is posted to the problem, where $\neg\tau$ is posted as

$$M_{ik} \neq M_{jk} \vee M_{ij} \leq M_{ik} \vee M_{ij} \leq M_{ik} \quad (8)$$

when $\tau = (ij)k$ and posted as

$$M_{ij} \neq M_{ik} \vee M_{ij} \neq M_{jk} \vee M_{ik} \neq M_{jk} \quad (9)$$

Algorithm NECESSITY

```

Necessity( $F, \tau$ )
1  let ( $V, D, C$ )  $\leftarrow$  CPMoel( $F$ )
2  for  $T \in F$  do
3      for  $t \in \text{BreakUp}(T)$  do  $C \leftarrow \text{post}(t, C)$ 
4   $C \leftarrow \text{post}(\neg\tau, C)$ 
5  return  $\neg\text{propagate}(V, D, C)$ 
    
```

Figure 20: Does the triple/fan τ occur in every supertree that displays the trees in F ?

when $\tau = (ijk)$. A call is then made to propagate to make the problem arc-consistent (line 5), and if this fails then τ is necessary, otherwise it is not necessary. The algorithm has the same complexity as CPBUILD.

6.4 Coping with conflict

When a supertree cannot be produced from a pair of trees some of the input triples and fans must be in conflict with one another, either directly or indirectly. Junker's QUICKXPLAIN method [Jun04] discovers a minimal subset of constraints that when posted and propagated result in a failure. This set is not necessarily the smallest possible set but is minimal in the sense that the removal of any element from this set will not constitute a sound explanation, and the addition of any constraint would be redundant. When the set of constraints are input triples and fans, this minimal set is semantically a collection of input data that is incompatible. [Jun04] states that this method can be achieved by a worst case of $2k \cdot \log_2(n/k) + 2k$ propagations⁴, where k is the size of the minimal explanation found and n is the number of constraints.

An alternative approach is to satisfy as many of the input triples and fans as is possible within a reasonable amount of time, i.e., polynomial time. Semple and Steel propose such an algorithm, MINCUTSUPERTREE [SS00], and this has been refined by Page [Pag02]. We now propose a similar scheme within the constraint programming framework. We call this algorithm GREEDYBUILD and it works as follows. We associate a constrained integer variable x , with a domain of $\{0, 1\}$, to each triple and fan. If the variable is assigned the value 0 then the triple (or fan) is respected, otherwise it is ignored. Therefore for a triple $(ij)k$ we post the constraint of equation 10 and for a 3-fan (ijk) the constraint of equation 11.

$$(x = 0 \wedge M_{ij} > M_{ik} = M_{jk}) \vee (x = 1 \wedge (M_{ik} \neq M_{jk} \vee M_{ij} \leq M_{ik} \vee M_{ij} \leq M_{ik})) \quad (10)$$

$$(x = 0 \wedge M_{ij} = M_{ik} = M_{jk}) \vee (x = 1 \wedge (M_{ij} \neq M_{ik} \vee M_{ij} \neq M_{jk} \vee M_{ik} \neq M_{jk})) \quad (11)$$

GREEDYBUILD then instantiates in turn each of the x variables, i.e. the decision variables, preferring the value 0 to the value 1, and after each instantiation the problem is made arc-consistent. The algorithm is shown in Figure 21. In line 1 a constraint model is produced,

4. our BC(Z) propagator is $O(1)$ however, so the time complexity is the same as the number of propagations

Algorithm GREEDYBUILD

```

GreedyBuild( $F$ )
1  let  $(V, D, C) \leftarrow \text{CPModel}(F)$ 
2  let  $X \leftarrow \emptyset$ 
3  for  $T \in F$  do
4      for  $t \in \text{BreakUp}(T)$  do
5          let  $x \leftarrow \text{newVar}(0, 1)$ 
6          let  $c \leftarrow \text{newConstraint}((t \wedge x = 0) \vee (\neg t \wedge x = 1))$ 
7           $X \leftarrow X \cup \{x\}$ 
8           $V \leftarrow V \cup \{x\}$ 
9           $C \leftarrow \text{post}(c, C)$ 
10 for  $x \in X$  do
11     instantiate( $x$ )
12     propagate( $V, D, C$ )
13 return  $\text{UMToTree}(V, D)$ 

```

Figure 21: Greedily Build a supertree from forest F using the ultrametric constraint model.

i.e., an $n \times n$ symmetric array of constrained integers variables is created, where there are n unique species in the forest, and the UM-MATRIX-BC(Z) constraint is then posted over those variables. The variable X of line 2 is then the set of decision variables. The input trees are broken up as before, and a new variable x is created for each triple or fan. In line 6 the constraints of equations 10 and 11 are posted into the model. The loop of lines 10 to 12 in turn select a decision variable, set it to its lowest possible value, and then make the problem arc-consistent. This might in turn cause uninstantiated variables to have the value 0 removed from their domain if their associated triple or fan conflicts with the triple or fan that has just been enforced. This process terminates without failure, because any conflicting triples or fans are essentially ignored. In line 13 the ultrametric matrix is then converted to a tree. The complexity of GREEDYBUILD is then $O((t + f).n^4)$ where there are t triples and f fans.

GREEDYBUILD was applied to the forest of bird data $\{A, B, C, D, E, F, G\}$ from section 5, using a soft breakup. This data is incompatible when we use CPBUILD, however GREEDYBUILD produces the supertree in Figure 22. This supertree contains 121 species. SOFTBREAKUP produced 201 triples, and of those 17 were rejected. It took less than 2 seconds to build the model and about 100 seconds to solve that model. This should be compared to CPBUILD over the same data set, taking 1.5 seconds to build the model and 10 seconds to determine incompatibility. GREEDYBUILD was also applied to the data set ABDF, producing the identical supertree to CPBUILD, in comparable time (890ms to build the model and 578ms to solve).

Having executed GREEDYBUILD the decision variables in the set X (lines 2, 7, 10 and 11) can be analysed to identify the set of triples and fans that have been excluded from the supertree, i.e., if an x variable has been instantiated with the value 1 then its corresponding triple or fan has been ignored.

Note that we do not claim any biological significance in the arbitrary order we use to suppress triples. GREEDYBUILD could be amended to follow the order of MINCUT-

THE ULTRAMETRIC CONSTRAINT AND PHYLOGENETICS

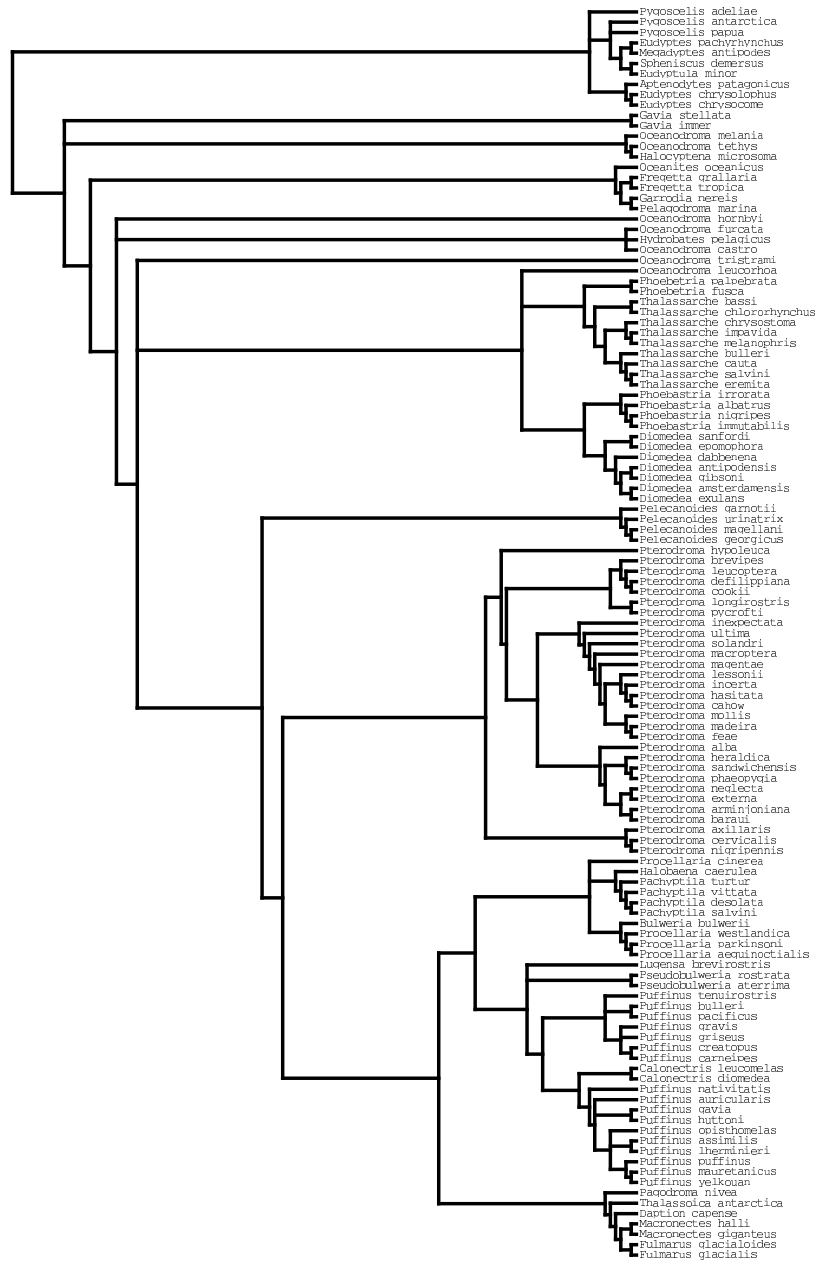


Figure 22: Supertree with largest data set of birds, ABCDEFG, with 121 species. This took about 2 seconds to model and 100 seconds to solve using GREEDYBUILD. Displayed using Rod Page's treeView [Pag96].

SUPERTREE but we have not investigated this. GREEDYBUILD can also be enhanced as follows. Currently if a triple or fan exists in multiple input trees then it occurs only once as a constraint. This information could be exploited by weighting the decision variables to take into consideration the relative weight of evidence for a triple or fan, e.g., the number of times that a triple or fan occurs as input. The decision variables can then be instantiated in non-increasing order of weight, i.e., a variable ordering heuristic can be used. In the extreme GREEDYBUILD can be modified to become OPTBUILD where a full backtracking search is performed with the objective of minimising the sum of the decision variables, but at a potentially exponential cost in time. This would return the tree with the fewest possible input triples suppressed.

7. Conclusion

We have presented a new constraint propagator for the ultrametric constraint over three integer variables, and shown how this can be extended to a symmetric array of constrained integer variables. When bounds(Z) consistency is established on the symmetric array the lower bounds of variables give mutual support. This is sufficient for modelling and solving the supertree construction problem in $O(n^4)$ time and $O(n^2)$ space, comparable to the complexity of Ng and Wormald’s ONETREE[NW96] but inferior to that of Bryant and Steel [BS95]. So, why bother with the CPBUILD approach when efficient imperative approaches already exist? The answer lies in the versatility of constraint programming. Rather than develop a new algorithm for a new variant of the supertree problem we add side constraints to our model, and we have shown that a polynomial time bound can often be achieved. We have done this for ancestral divergence data and nested taxa, and shown how our model can be used to deliver necessary triples and fans. Although not shown, it is obvious that input ancestral divergence data and nested taxa can be combined in the one model, simply by adding all the necessary constraint and auxiliary variables for both variants into the one model.

When input trees conflict we propose two options: use QUICKXPLAIN to determine the cause of that conflict or greedily build a supertree using GREEDYBUILD. In [BSS04] Bryant, Semple and Steel state that they have essentially an “all-or-nothing” approach to supertree construction when using RANKEDTREE and what is needed is something akin to MINCUTSUPERTREE, i.e. when trees are incompatible build a supertree that violates the minimum number of triples or fans, and do this in polynomial time [Pag02, SS00]. This has since been done by [BES06b] and can also be done in our constraint model by incorporating the constraints identified in section 6.1 into GREEDYBUILD [CPB].

Acknowledgements

We would like to thank Pierre Flener and Xavier Lorca; Barbara Smith, Ian Gent and Christine Wei Wu; Charles Semple, Mike Steel and Rod Page; Muffy Calder and Joe Sventek; Stanislav Zivny; and Chris Unsworth.

References

- [ASSU81] A.V. Aho, Y. Sagiv, T.G. Szymanski, and J.D. Ullman. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM J. Comput.*, 10(3):405–421, August 1981.
- [BE04] O.R.P. Bininda-Emonds. *Phylogenetic Supertrees: Combining information to reveal the tree of life*. Springer, 2004.
- [Bes06a] Christian Bessière. Constraint propagation. In *Handbook of constraint programming*. Elsevier, 2006. Chapter 3.
- [BES06b] Magnus Bordewich, Gareth Evans, and Charles Semple. Extending the limits of supertree methods. *Annals of combinatorics*, 10:31–51, 2006.
- [BFL08] Nicolas Beldiceaun, Pierre Flener, and Xavier Lorca. Combining tree partitioning, precedence, and incompatibility constraints. *Constraints*, 13:1–31, 2008.
- [BS95] D. Bryant and M. Steel. Extension Operations on Sets of Leaf-labeled Trees. *Advances in Applied Mathematics*, 16:425–453, 1995.
- [BSS04] David Bryant, Charles Semple, and Mike Steel. Supertree methods for ancestral divergence dates and other applications. In Olaf Bininda-Emonds, editor, *Phylogenetic Supertrees: Combining information to reveal the tree of life*, pages 151–171. Computational Biology Series Kluwer, 2004.
- [CL97] Yves Caseau and Francois Laburthe. Solving small TSPs with constraints. In *Proceedings International Conference on Logic Programming*, pages 1–15, 1997.
- [CP94] J. Carlier and E. Pinson. Adjustment of heads and tails for the jobshop scheduling problem. *European Journal of Operational Research*, 78:146–161, 1994.
- [CPB] CPBuild. <http://www.dcs.gla.ac.uk/pat/supertrees/> constraint programming supertree construction.
- [Dan03] Philip Daniel. Supertree methods: Some new approaches. Master’s thesis, Department of Mathematics and Statistics, University of Canterbury, 2003.
- [DB97] R. Debruyne and C. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of IJCAI’97*, pages 412–417, 1997.
- [DS04] Philip Daniel and Charles Semple. Supertree algorithms for nested taxa. In Olaf Bininda-Emonds, editor, *Phylogenetic Supertrees: Combining information to reveal the tree of life*, pages 151–171. Computational Biology Series Kluwer, 2004.
- [DW04] Richard Dawkins and Yan Wong. *The Ancestor’s Tale*. Weidenfeld and Nicholson, 2004.

- [GPSW03] I.P. Gent, P. Prosser, B.M. Smith, and W. Wei. Supertree construction with constraint programming. In *Principles and Practice of Constraint Programming*, pages 837–841. Springer, 2003.
- [Gus97] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA, 1997.
- [HG95] William D. Harvey and Matthew L. Ginsberg. Limited Discrepancy Search. In *Proceedings IJCAI'95*, 1995.
- [HSD98] P. Van Hentenryck, V.A. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). *Journal of Logic Programming*, 37:139–164, 1998.
- [JC95] Peter G. Jeavons and Martin C. Cooper. Tractable constraints on ordered domains. *Artif. Intell.*, 79(2):327–339, 1995.
- [JMSO95] D.N. Janczewski, W.S. Modi, J.C. Stephens, and S.J. O'Brien. Molecular evolution of mitochondrial 12S RNA and Cytochrome b sequences in the pantherine lineage of Felidae. *Mol. Biol. Evol.*, 12:690–707, 1995.
- [Jun04] Ulrich Junker. QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems. In *Proceedings AAAI2004*, pages 167–172, 2004.
- [KP02] M. Kennedy and R.D.M. Page. Seabird supertrees: Combining partial estimates of procellariiform phylogeny. *The Auk*, 69:88–108, 2002.
- [Lho03] Olivier Lhomme. An efficient filtering algorithm for disjunction of constraints. In *Principles and Practice of Constraint Programming*, pages 904–908. Springer, 2003.
- [Mac77] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
- [MGP03] Gorgina M. Mace, John L. Gittleman, and Andy Purvis. Preserving the Tree of Life. *Science*, 300:1707–1709, 2003.
- [MS07] D.R. Maddison and K.-S. Schulz. Tree of life web project. <http://www.tolweb.org/tree/>, 2007.
- [NW96] Meei Pyng Ng and Nicholas C. Wormald. Reconstruction of rooted trees from subtrees. *Discrete Appl. Math.*, 69(1-2):19–31, 1996.
- [Pag96] R.D.M. Page. TREEVIEW: An application to display phylogenetic trees on personal computers. *Computer Applications in the Biosciences*, 12:357–358, 1996.
- [Pag02] R.D.M. Page. Modified mincut supertrees. *WABI 2001, Workshop on Algorithms in BioInformatics*, 2002.

- [Pag04] Roderic D.M. Page. Taxonomy, supertrees, and the tree of life. In Olaf Bininda-Emonds, editor, *Phylogenetic Supertrees: Combining information to reveal the tree of life*, pages 247–265. Computational Biology Series Kluwer, 2004.
- [Pen03] Elizabeth Pennisi. Modernizing the Tree of Life. *Science*, 300:1692–1697, 2003.
- [Pro06] P. Prosser. Supertree construction with constraint programming: recent progress and new challenges. In *WCB06 - Workshop on Constraint Based Methods for Bioinformatics*, pages 75–82. N/A, 2006.
- [R94] J-C. Régim. A filtering algorithm for constraints of difference in CSP’s. In *Proceedings AAAI’94*, pages 362–367, 1994.
- [RvBW07] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier, 2007.
- [SC06] Christian Schulte and Mats Carlsson. Finite domain constraint programming systems. In *Handbook of constraint programming*. Elsevier, 2006. Chapter 14.
- [SDH⁺04] Charles Semple, Philip Daniel, Wim Hordijk, Roderic D.M. Page, and Mike Steel. Supertree algorithms for ancestral divergence dates and nested taxa. *Bioinformatics*, 20(15):2355–2360, 2004.
- [SS00] Charles Semple and Mike Steel. A supertree method for rooted trees. *Discrete Appl. Math.*, 105(1-3):147–158, 2000.
- [Tre03] TreeBASE. <http://www.treebase.org/> TreeBASE: a database of phylogenetic knowledge, 2003.
- [Tsa93] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.
- [vHDT92] P. van Hentenryck, Y. Deville, and C-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.
- [WYL07] Gang Wu, Jia-Huai You, and Guohui Lin. Quartet-based phylogeny reconstruction with answer set programming. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 4:139–152, 2007.