Faster Phylogenetic Inference with MXG

David G. Mitchell, Faraz Hach and Raheleh Mohebali

Computational Logic Laboratory
Simon Fraser University, Burnaby BC, CANADA
{mitchell,fhach,rmohebal}@cs.sfu.ca

Abstract. We apply the logic-based declarative programming approach of Model Expansion (MX) to a phylogenetic inference task. We axiomatize the task in multi-sorted first-order logic with cardinality constraints. Using the model expansion solver MXG and SAT+cardinality solver MXC, we compare the performance of several MX axiomatizations on a challenging set of test instances. Our methods perform orders of magnitude faster than previously reported declarative solutions. Our best solution involves polynomial-time pre-processing, redundant axioms, and symmetry-breaking axioms. We also discuss our method of test instance generation, and the role of pre-processing in declarative programming.

Keywords: Phylogeny, Declarative Programming, Model Expansion.

1 Introduction

We apply a declarative programming approach, based on the logical task of model expansion (MX), to a problem in phylogenetic inference. In the approach, an instance is a finite structure; a solution is a finite structure; a problem specification is an axiomatization, in a suitable logic, of the relationship between instances and solutions [16].

A phylogenetic tree is a directed graph representing the evolutionary relationships among a collection of species. Phylogenetic inference (or re-construction) is the task of constructing a phylogenetic tree (or other network) from species data. It has many applications in biology and elsewhere, producing a variety of particular computational problems. Our interest in these problems is primarily as developers of declarative programming tools: In trying to make our tools more effective, it is useful to work on challenging applications, especially those where success may not be immediate, but benefits may be significant. Phylogenetic inference is interesting because of the following observations. Most interesting variants are NP-hard optimization problems, and there are many data sets too hard to solve well in practice; Many particular problems are variants of, or combinations of, a few basic problems; And, the optimality metrics often do not precisely match subjective solution quality, so users could benefit from a method to interactively add ad-hoc constraints, which is not possible with current tools.

Contributions We describe a method that is faster, by many orders of magnitude, than the only previous declarative solution we know of. In doing so, we demonstrate that MX-based tools can be effective on more realistic domains than has previously been shown. Effectively measuring progress in solving NP-hard problems is tricky, and we

believe the method we use here is interesting. Our pre-processing method, in addition to benefiting our own solution, could improve the performance of existing software packages. We point out that instance pre-processing, often important in problem solving, can be done declaratively.

The Problem The particular problem we study is the binary cladistic Camin-Sokal (CCS) problem, which we chose because it is a simple problem which is NP-hard; to which standard tools apply; for which we have suitably challenging real data; and for which there is a previous declarative programming solution to compare with. Our primary solving mechanism is the model expansion grounder/solver MXG [16], with the SAT+cardinality solver MXC [2]. Our test set consists of several hundred instances of graduated difficulty derived from biological data. We compare the performance of:

- MXG and MXC, with various axiomatizations using cardinality constraints;
- MXG and Minisat [6], a high-performance SAT solver, with non-cardinality MX axiomatizations;
- clasp [9], a high-performance answer set programming (ASP) solver, with the best-performing ASP axiomatization from [13];
- MXG and MXC, aided by polynomial-time instance pre-processing;
- PAUP, a widely-used phylogeny software package [19].

Related Work Kavanagh et al [13] reported answer set programming (ASP) based solutions to binary CCS. Their best solution established optimal trees for instances for which the phylogeny software package PENNY [8] could not, but could not solve their largest instance (which is identical to our largest instance), or even moderate-sized subsets. ASP solutions to some other phylogeny problems, which are not directly comparable, are reported in [3, 7, 1]. [3] use "large compatibility", where the goal is to find the maximum number of characters, for a given set of species, for which there is a perfect phylogeny. In contrast, we use "large parsimony", where we find a (perhaps not perfect) phylogeny for the input species with the minimum number of evolutionary changes. The task in [7] is to construct a "perfect phylogenetic network", from given phylogenetic trees (the "species" there are natural languages). [1] studied the "Maximum Quartet Consistency" problem, and evaluated an ASP solution on synthetic data.

2 The Binary Camin-Sokal Phylogeny Problem

We study a simple "large parsimony" problem in character-based cladistics. A group of species is characterized by a set of *characters*. Each character can take one of several *states*, and each species is described by a vector giving a state for each character. The input is a set of species vectors and the goal is to construct a tree with nodes labeled by character vectors, so that the vector of every input species labels some node. Changes of a character's state along an edge are mutations. Problem variations result from differing cost metrics and restrictions on character changes. A tree minimizing the cost metric is a "most parsimonious tree". In the cladistic Camin-Sokal (CCS) problem, the states of each character are ordered and mutations must be increasing on this order. This is

appropriate when the direction of evolutionary change of characters is assumed to be known. The goal is to minimize the total number of mutations. (Examples of biological application of CCS include [5, 18]). The decision version of the problem, even in the binary case where each character has just two states, is NP-complete [4].

Definition 1. *The binary cladistic Camin-Sokal problem (binary CCS) is:*

> **Instance:** *Set S of n distinct vectors from $\{0,1\}^m$; natural number B.*
> **Question:** *Is there a directed tree $T = (V, E)$, such that: 1) T is rooted at 0^m; 2) $S \subseteq V \subseteq \{0,1\}^m$; 3) Every leaf of T is in S; 4) For each directed edge $(v_1, v_2) \in E$, v_1 and v_2 differ in exactly one character, which is 0 at v_1 and 1 at v_2; 5) $|V| \leq B$.*

Remark 1. An alternate definition allows multiple mutations on an edge. The definition we use here was also used in [13], and is easier to axiomatize in MXG.

In a *perfect phylogeny*, each character mutation occurs only once. For the binary CCS, this is equivalent to setting $B = m$, provided that both states of every character occur in S. (Note that if some character has only one state, we may safely delete it.) When an instance does not have a perfect phylogeny, some character mutations must occur more than once in the tree. Since mutations are irreversible in CCS, the same character cannot change more than once on a directed path from the root, so the same mutation will occur in distinct subtrees. The goal is to find a tree that minimizes the number of these "extra mutations". We allow only one mutation per edge, so the number of extra mutations is equivalent to the number of "extra vertices" or "extra edges" needed to construct a phylogeny. Since mutation is irreversible, we may assume that all mutations are from state 0 to state 1, and the tree is rooted at the zero vector.

3 Model Expansion and MXG Basics

We give a minimal and informal description of MXG. For further details, including formal aspects of MX, the MXG language and grounding algorithm, examples and performance on other problems, see [16]. MXG is a model expansion grounder/solver. It takes as input a problem specification S and an instance I. The problem specification is essentially an axiomatization in multi-sorted first-order logic (FO) extended with inductive definitions and cardinality constraints.

Vocabulary symbols in an axiomatization have three distinct roles: Instance vocabulary consists of symbols whose interpretation is given by an instance; Solution vocabulary is symbols whose interpretations comprise a solution; Auxiliary vocabulary is symbols that are not part of the instance or solution. The solution and auxiliary vocabulary together form the *expansion vocabulary*, the symbols whose interpretations must be constructed by the solver. Problem specifications contain declarations of types, typed declarations of vocabulary symbols, and axioms. They have three parts: Given: has declarations of all types and instance vocabulary; Find: has the declaration of solution vocabulary; Satisfying: has axioms, plus declaration of auxiliary vocabulary, if any.

As an example, Figure 1 is an MXG specification for the graph colouring problem. The sorts are Vtx (vertices) and Clr (colours); the instance vocabulary is the binary

```
Given:   type  Vtx Clr;
         Edge(Vtx, Vtx)

Find:    Colour(Vtx, Clr)

Satisfying:
         ∀ x : ∃ y : Colour(x, y)
         ∀ x y : (Edge(x,y) ⊃ (∀ z : ¬(Colour(x, z) & Colour(y,z))))
         ∀ x y z : ((Colour(x, y) ∧ Colour(x, z)) ⊃ (y=z))
```

Fig. 1. An MXG problem specification for graph colouring

relation Edge; the solution vocabulary is the binary relation Colour. The axioms say that the interpretation of Colour must give a proper colouring of the given graph. The MXC instance file for the instance with colours $\{1, 2\}$, vertices $\{1, 2, 3\}$, and edges $(1, 2)$ and $(1, 3)$ contains: Vtx = [1..3]   Clr = [1; 2]   Edge = $\{$1, 2; 1, 3$\}$.

MXG combines a specification and instance, producing a propositional formula ϕ which is a "reduced grounding" of S with respect to I. That is, satisfying assignments of ϕ correspond one-to-one with solutions of I. Currently, ϕ is a CNF formula, possibly extended with cardinality constraints. A propositional solver searches for a satisfying assignment to ϕ, and if found MXG maps the assignment back to the FO language to produce a solution. Other relevant aspects of the (current) MXG language are:

- Vocabulary symbols are typed, by declarations in the specification. The domain of each variable is inferred from its use, which must be consistent.
- Each type is an ordered finite set given by the instance. The ordering is determined by the form of the instance description: Numerical if expressed as a range of integers; otherwise as enumerated. For each type, constant symbols MIN and MAX, binary relation symbols $<$, $\leq$, etc., and binary relation SUCC are all implicitly defined, with the natural semantics. Types are disjoint, so two elements are comparable only if of identical type.
- Bounded quantifiers are supported: $\forall$x y $<$ x : ϕ(x,y) is equivalent to $\forall$x $\forall$y : (y$<$x $\supset$ ϕ(x,y)); $\exists$x y $<$ x : ϕ(x,y) is equivalent to $\exists$x $\exists$y : (y$<$x $\wedge$ ϕ(x,y)).
- Simple cardinality constraints are supported, which are universal sentences of the form $\forall \boldsymbol{x} : \odot(n; \boldsymbol{y}; \phi(\boldsymbol{x}, \boldsymbol{y}))$, where $\odot$ is one of UB, LB, or CARD, for upper bound, lower bound, and equivalence, respectively. For example $\forall$u : UB(1;v;Edge(v,u)) says the in-degree of every vertex is at most 1.
- A limited form of inductive definition is supported (see [16] for details).

4 MX Axiomatizations of Binary CCS

Here we give three MX axiomatizations of Binary CCS. One we find natural and simple, using cardinality constraints; one uses no cardinality constraints, and can be solved by straightforward reduction to SAT; and one is a translation to MX of the best ASP encoding from [13]. We produced other distinct axiomatizations, but since none performed better than our basic one (except when using enhancements such as described in Section 6), we do not report them.

```
Given:  type  Char Vertex State;
        A(Vertex, Char, State)
        NSpecies: Vertex
        NEdges: Vertex

Find:   Edge(Vertex, Vertex)
        Vector(Vertex, Char)

Satisfying:
        ∀ s ≤ NSpecies c : (A(s,c,MAX) ⇔ Vector(s, c))              (1)
        ∀ u v : UB(1; c; (Edge(u,v) ∧ ¬ Vector(u,c) ∧ Vector(v,c)))   (2)
        ∀ u v : (Edge(u,v) ⊃ (∃c : (¬Vector(u,c) ∧ Vector(v,c))))    (3)
        ∀ u v : UB(0; c; (Edge(u,v) ∧ Vector(u,c) ∧ ¬ Vector(v,c)))   (4)
        CARD(NEdges; v, u; Edge(v,u))                                 (5)
        ∀ v>MIN : CARD(1; u; Edge(u, v))                              (6)
```

Fig. 2. Basic MX axiomatization of binary CCS phylogeny re-construction

Basic MX Axiomatization The types are Vertex, vertices of the tree; Char, the set of characters; and State ($=\{0,1\}$), the set of states. We identify the n species with the first n vertices. The (too simple) type system requires that variables and constant symbols which are to range over species must be of type Vertex. The instance vocabulary consists of:

- A(Vertex, Char, State): the set of triples specifying the matrix of species of data. (The first argument is the species.)
- NSpecies: a constant symbol denoting the number of species.
- NEdges: a constant symbol which is always set to $|\text{Vertex}| - 1$.

The solution vocabulary has two binary relation symbols: Edge, the set of edges, and Vector, which labels vertices with character vectors. Vector(v,c) holds if character c has state 1 in the vector labeling v. The axioms (see Figure 2) state:

- The label of vertex i, for $i \in \{1,\ldots,n\}$, must be species vector i (Axiom 1);
- Each edge has exactly one character changing from 0 to 1, and no characters changing from 1 to 0 (Axioms 2–4);
- Every node, except the root, has in-degree exactly one, and the number of edges is exactly the number of vertices less one (Axioms 5,6).

Axioms 2 through 4 ensure edges have only allowed mutations, and in particular that every path is monotone increasing in the set of characters with state 1; Axioms 5 and 6 ensure the graph is a tree, which is rooted at the zero vector by a convention that species 1 is the zero vector.

Non-Cardinality Axiomatization To determine if we obtain a speed-up over pure SAT solving by using MXC with cardinality constraints, we produced several axiomatizations without cardinality constraints, which MXG grounds to SAT. Figure 3 shows the best-performing of these. The axioms state: The input species vector i must label vertex i (Axiom 1 - as before); Each edge has exactly one character changing from 0 to 1 (Axiom 2); On a directed path the set of 1-characters is monotone increasing (Axiom

```
Given:  type  Char Vertex State;
        A(Vertex, Char, State)
        NSpecies: Vertex
        NEdges: Vertex

Find:   Edge(Vertex, Vertex)
        Vector(Vertex, Char)

Satisfying:
        TC(Vertex, Vertex)   // TC will be the transitive closure of Edge.
        ∀ s ≤ NSpecies  c :  (A(s,c,MAX) ⇔ Vector(s,c))                                    (1)
        ∀ v1 v2: (Edge(v1,v2)⊃(∃ c1 : (¬Vector(v1,c1)∧Vector(v2, c1)                      (2)
                      ∧ (∀c2 : ((¬Vector(v1,c2) ∧ Vector(v2, c2)) ⊃ (c1=c2))))))
        ∀ u v c : ((TC(u, v) ∧ Vector(u, c)) ⊃ Vector(v,c))                                (3)
        ∀ u>MIN : ∃ v : (Edge(v,u) ∧ (∀v2 : (Edge(v2, u) ⊃ (v2 = v))))                     (4)
        ∀ u v>u : ¬(TC(u, v) ∧ TC(v, u))                                                   (5)
        ∀ u v : (TC(u,v) ⇔ ((u = v) ∨ Edge(u,v) ∨ (∃x : (TC(u, x) ∧ Edge(x,v)))))          (6)
```

Fig. 3. MXG axiomatization with no cardinality constraints

3), so there are no reverse mutations; The graph is a tree (Axioms 4 and 5), since every node but the root has in-degree one and there are no cycles in the transitive closure of Edge; TC is the transitive closure of Edge (Axiom 6 provides the lower bound on TC; Axiom 5 the upper bound). We report results based on the SAT solver minisat, arguably the best all-around SAT solver available.

Translation of ASP to MX We also used an MX axiomatization based on the best ASP encoding from [13] (denoted "A+" there). It differs from the previous two in that: 1) We have a type Species, distinct from Vertex. 2) Rather than identify the n species with the first n vertices, we construct a function P (represented as a binary relation) from species to vertices. 3) We construct a function M from vertices to characters. The mutation on edge (u,v) is the character c such that M(v,c) holds. In contrast, in our previous axiomatizations the mutation on edge (u,v) is implicit in the difference between the vectors labeling u and v. 4) The (root) vector **0** is left implicit, so a solution is a forest. A tree is obtained by adding an edge from **0** to the root of each forest component.

Figure 4 gives the axioms, which state: Each vertex is mapped to exactly one character (Axiom 1); Each species is mapped to a vertex (Axiom 2); The graph is a tree (Axioms 3–5); The characters which are 1 at species S must have mutated at some ancestor of the node S is mapped to (Axiom 6); If species S is mapped to vertex v, the character which mutated at v must not be 0 at S (Axiom 7); A character mutates at most once on any (directed) path (Axiom 8). Axioms 7 and 8 are redundant, but improve performance.

5 Evaluating Progress in Performance

Evaluating performance of solvers for NP-hard problems has many pitfalls, especially when there is no base-line provided by well-established benchmarks and solvers. Our goal is to have a clear measure of *progress* in performance. Direct comparison of run-times does not work here, because run-times for the methods we test vary by many

```
Given:  type Chars Vertex State Species;
        A(Species, Char, State)

Find :  Edge(Vertex, Vertex)

Satisfying :
        M(Vertex, Char)
        P(Species, Vertex)
        TC(Vertex, Vertex)
        ∀ v : CARD(1; c; M(v,c))                                                        (1)
        ∀ s : CARD(1; v; P(s,v))                                                        (2)
        ∀ v : UB(1; u; Edge(u, v))                                                      (3)
        ∀ u v : (TC(u,v) ⇔ ((u = v) ∨ Edge(u,v) ∨ (∃x : (TC(u, x) ∧ Edge(x,v)))))       (4)
        ∀ u v>u : ¬(TC(u,v) ∧ TC(v,u))                                                  (5)
        ∀ s c: ( A(s,c,MAX) ⇔ (∃ u v : (TC(u, v) ∧ M(u,c) ∧ P(s, v))))                  (6)
        ∀ s v c : ¬(P(s,v) ∧ M(v,c) ∧ A(s,c, MIN))                                      (7)
        ∀ v v1>v c : ¬(TC(v, v1) ∧ M(v1, c) ∧ M(v, c))                                  (8)
```

Fig. 4. MXG axiomatization based on ASP encoding "A+" from [13].

orders of magnitude (see Figure 8). A better measure is the number of instances that can be solved within reasonable time. For this, a collection of related instances of graduated difficulty is needed, but in practice this is often hard to arrange. For example, [13] obtained three real data sets: Two were too easy and the third was too hard. Randomly generated instances are easily graduated, but their use requires care (see, e.g., [17]), and may be irrelevant to practice.

Instances Here, we produce a set of suitably graduated instances from the one challenging real data set we have for our problem. This is possible because, if we view a set of n species vectors of length m as an $n \times m$ matrix M, any sub-matrix of M is a valid set of data, as real (or not) as the full matrix. For illustration: {eye-colour,hair-colour} is as valid a set of characters as {eye-colour,hair-colour,handedness}. (Not every such matrix is scientifically interesting, of course.) Our initial instance is a 36×63 matrix obtained from the experimentally obtained haplotype data of [12] (for Poecilia reticulata – guppies) as described in [13]. Following [13], we produced a set of instances from upper-left sub-matrices of size $k \times l$, for k, l multiples of 3. Thus, we view performance as a function of two natural instance parameters: number of species and number of characters. Unfortunately, the resulting instances were not nicely graduated, as most moderate-size instances were very easy for our methods. The problem was that most sub-matrices had all-zero columns and duplicate rows, which we solved as follows. Keeping the zero vector as species 1, we put all other species in reverse lexicographic order. Thus, the first row was all zero, but the second row had many ones. The set of instances produced from this initial matrix by the scheme described above satisfied our main criteria: the instances are smoothly graded in difficulty for our solvers, and they do not contain significant numbers of trivial or duplicate rows or columns.

Performance Measure As an objective measure of progress, we require the solver to establish the optimal phylogeny size within a fixed time bound. As with any optimization problem, one may trade solution quality for solving time. In phylogenetic inference,

user's often don't care about optimality *per se*, because the cost metrics do not exactly correspond their subjective notion of quality. But if optimality is not a precise measure of quality, surely being within some distance of optimal is not either, so relaxing the optimality requirement does not improve our measure. The requirement to solve to optimality would seem to better measure whether we are making progress in dealing with whatever it is that makes up the combinatorially hard aspect of our instances. For measuring progress toward being able to practically solve larger and harder instances than currently possible, we believe that establishing optimal solutions within a reasonable time cut-off is as good a measure as any we know of.

Evaluation MXG does not have a built-in optimization facility, so for each optimization instance we solve a sequence of search instances. The first asks for a perfect phylogeny (with the same number of mutations as characters). Successive instances allow one more mutation. We run the solver on the sequence, stopping when a solution is found – which must be optimal – or when the cumulative run time reaches two hours. Sequential search is faster than binary search because instances with too few mutations are typically easier than those with too many. Time for sequential search is dominated, almost without exception, by the two instances needed to establish optimality: the one producing an optimum solution and the one with one fewer mutations. Binary search is often dominated by the instances just beyond optimal, which sequential search never visits. This pattern of hardness also supports our argument in favour of using optimality in our measure of performance.

Tests were run on Sun Fire VZ20 Dual Opteron computers with 2.4 GHz AMD Opteron 250 processors, with 1MB cache and 2GB of RAM per processor, running Suse Enterprise Linux 2.6.11. The software versions were: MXG 0.17; MXC 0.5; minisat_v2s; clasp rc3; and paup4b10-opt-linux-a. Executables for clasp and PAUP were downloaded from the solver web sites, while MXG, MXC, and minisat_v2s were compiled with gcc version 3.3.4. [1]

Figure 5 shows the "frontier" for MXG with the axiomatizations of Section 4, and for the ASP solver clasp using the A+ axiomatization of [13]. We plot a curve for each solving method. A point at (x, y) denotes that x is the largest number of characters for which the method succeeded in solving instances with y species within the two-hour cut-off. Instances left of or below a curve were solved; those above and to the right were not. The basic MX axiomatization is best, except with very few species. MXG performs relatively poorly with few species because it must construct the whole vector for each vertex, and thus with many characters has quite a bit of work to do, while the ASP axiomatizations do not. We ran two ASP solvers, cmodels [14] and clasp [9], on the A+ axiomatization. Since the performance was similar, with clasp being slightly better, we show only the clasp curve. The no-cardinality axiomatization and minisat performed essentially the same as our basic MX axioms, except for being slightly weaker with few species. Our translation of the ASP axioms to MX performed poorly.

[1] MXG and MXC are at `www.cs.sfu.ca/research/groups/mxp/`; minisat at `www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/`, clasp at `www.cs.uni-potsdam.de/clasp/` and PAUP at `http://paup.csit.fsu.edu/`.

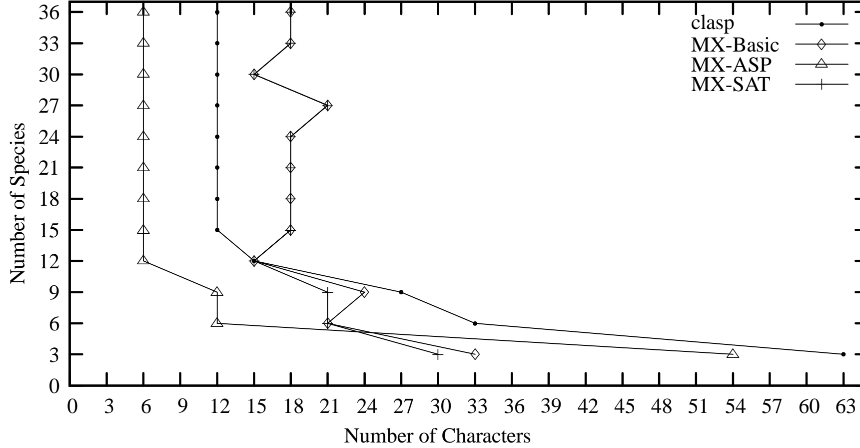

Fig. 5. Frontier comparison of three MX axiomatizations and an ASP solution.

We conclude that our basic MX axiomatization, which is already substantially better than the best solution in [13], is a good starting point for further work.

6 Enhancing Performance

In this section, we report on two ways we refined our basic axiomatization that dramatically improved performance. Adding redundant axioms is a standard method in SAT and CSP encodings, and [13] reported significant speedups with this method. It is not well understood why particular redundant axioms help (or hurt) performance. Natural explanations are that they increase the amount of unit propagation performed for some partial assignments, or that they help a clause-learning solver learn more useful clauses. Symmetry-breaking axioms eliminate some - but not all - solutions among a set of symmetric solutions. They often improve performance, even when only one solution is needed, presumably because they help the solver effectively eliminate many symmetric "near-solutions".

Axioms 13 of Figure 6 states that no extra vertex is a leaf. It is neither symmetry-breaking nor redundant, but has a similar flavour in that it removes only solutions that are not very interesting, and improves performance.

Computing Vertex Depth In a binary CCS tree, each vector labeling a vertex at depth k has exactly k 1's. Thus, if K is the maximum number of 1's in any species vector, the tree has height at most K. We can add axioms requiring labels of vertices to respect this property. These are axioms seven through ten and thirteen of Figure 6, which state:

- Each vertex must be assigned a unique depth (Axiom 8), which must be one greater than that of its parent (Axiom 9).
- The depth of each vertex is the number of 1's in its label (Axiom 10).

```
Given:  type  Char Vertex State Depth;
        A(Vertex, Char, State)
        NSpecies: Vertex
        NEdges: Vertex

Find:   Edge(Vertex, Vertex)
        Vector(Vertex, Char)

Satisfying:
        // Axioms 1-6 are the Basic MX Axioms of Figure 2

        VtxDepth(Nodes, Depth)
        SpcDepth(Nodes, Chars, Depth)
        { SpcDepth(u,c,d)  ← c=MIN ∧ d=MIN ∧ s = MIN ∧ A(u,c,s)                        (7)
          SpcDepth(u,c,d)  ← c=MIN ∧ SUCC(MIN, d) ∧ s = MAX ∧ A(u,c,s)
          SpcDepth(u,c,d)  ← SpcDepth(u,c1, d) ∧ SUCC(c1, c) ∧ A(u,c,s) ∧ s=MIN
          SpcDepth(u,c,d)  ← SpcDepth(u,c1, d1) ∧ SUCC(c1, c) ∧ SUCC(d1, d)
                              ∧ A(u,c,s) ∧ s=MAX
        }
        ∀ u : CARD(1; d; VtxDepth(u, d))                                                (8)
        ∀ u v d1 d2 : ((Edge(u,v) ∧ VtxDepth(u, d1) ∧ VtxDepth(v,d2)) ⊃ SUCC(d1, d2))  (9)
        ∀ u≤NSpecies d: (VtxDepth(u,d) ⇔ SpcDepth(u,MAX,d))                            (10)
        ∀ u>MIN : ¬ VtxDepth(u,MIN)                                                     (11)
        ∀ u>NSpecies v>u d1 d2 : ((VtxDepth(u,d1) ∧ VtxDepth(v,d2)) ⊃ d2 ≥ d1)         (12)
        ∀ u>NSpecies : ∃ v : Edge(u,v)                                                  (13)
```

Fig. 6. MX-Depth (Axioms 1-10) and MX-Depth+ (Axioms 1-13) axiomatizations.

– Only the root has depth 0 (Axiom 11);

These axioms are redundant, but significantly improve performance. They use two auxiliary relation symbols, SpcDepth and VtxDepth. VtxDepth(v,d) holds if vertex v is at depth d. SpcDepth(s, c, d) holds if the number of 1's among the first c characters for species s is d. Axiom 7 is an inductive definition which plays a special role. The form of this definition is such that MXG can compute the relation SpcDepth *before* grounding Axioms 8, 9 and 10. Thus, it is as if a pre-processor computed this relation and added it to the instance. Since SpcDepth(s, MAX, d) says that species s is at depth d, the grounder has computed the depth for each species. (For simplicity of axiomatization, we also added a new type Depth, which is a set the size of the maximum number of ones in species vector. We added this to our instances in a simple pre-processing step, although this could be avoided with a more complex axiomatization.)

Symmetry Breaking Our final example is a symmetry-breaking axiom. It states that the depth of "extra vertices" (those which allow extra mutations), respects their numerical order (Axiom 12).

Instance Pre-processing We found that instances (including the largest) often satisfied easily-checked properties that could be used to simplify them with a pre-processing step, which greatly improved performance. We recursively applied the following rules: 1. Delete any all-zero column: The character does not mutate, so we need no node for it. 2. Delete any column having exactly one 1: If c is 1 only in s, we construct a tree without c, then add $c = 0$ to every vector on the tree, adding one new edge and vertex

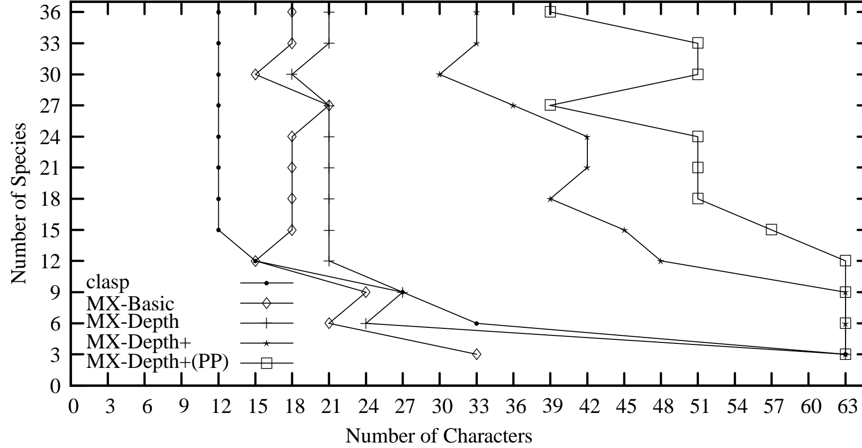

Fig. 7. Frontier plot showing performance improvement with refined axiomatizations (MX-Depth and MX-Depth+) and instance pre-processing (MX-Depth+(PP)).

where s appears. 3. Delete any column having exactly one 0: The 0 occurs in the root zero vector. We construct a solution without c, and insert one new node beneath the root in the solution, setting $c = 1$ everywhere except the root. 4. Delete any duplicate species. 5. Delete s_2 for any pair s_1, s_2 of species such that: (a) $s_1 \subset s_2$, i.e., every character that is 1 for s_1 is also 1 for s_2; (b) $|s2 - s1| = k$, i.e., s_2 has k more 1's than s_1; (c) $\forall s_3 \notin \{s_1, s_2\}, |s_2 \backslash s_3| \geq k$. Solve the instance without s_2, then add it to a path of length k below s_1.

Performance with Refined Axioms and Pre-processing Figure 7 is a frontier plot showing performance improvements obtained with enhanced axiomatizations and pre-processing. For comparison, we included the curves for clasp and MX-Basic, and in addition three new curves:

- MX-Depth: MX-Basic axioms extended with Axioms 7–10 of Figure 6;
- MX-Depth+: MX-Depth further extended with Axioms 11–13 of Figure 6;
- MX-Depth+(PP): Pre-processing of instances, and solving with the MX-Depth+ axiomatization (all axioms of Figure 6).

Remark 2. The "dip" in performance of MX-Depth+(PP) at 27 species is the consequence of pre-processing being less effective on these.

How Much Better: Frontier *vs.* **Run-time** The frontier plots show that we have progressed in terms of our chosen measure, but do not show the (dramatic) corresponding changes in run-time. Figure 8 illustrates, showing run-times as a function of number of characters, with number of species fixed at 24. Analogous curves for fewer or more species are similar, except for very small numbers of species. The y (time) axis is log

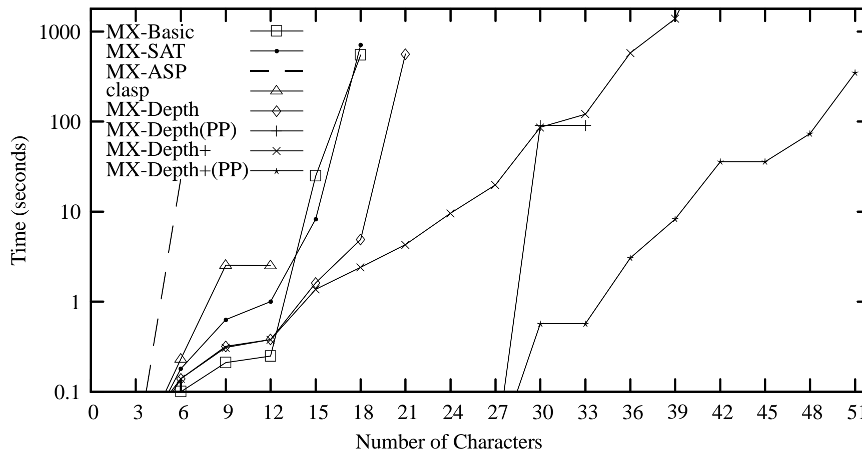

Fig. 8. Run-times for instances with 24 species, as a function of number of characters.

scale, so these run-times appear to be exponential in the number of characters. Notice that the curves have very different slopes, suggesting that the run-time curves for MX-Depth+ and MX-Depth+(PP) have much smaller exponents than the other solutions. We cannot really extrapolate the curves for the ASP or Basic MX solutions to compare run-times for large instances with the best methods, but unless the curves here are completely mis-leading the difference is certainly many orders of magnitude. Solving the hardest instances with those methods is completely infeasible in practice.

7 MXG *vs.* PAUP

The two most widely used phylogeny software packages, PHYLIP [8] and PAUP [19], both use two methods to carry out phylogenetic inference (for CCS and other models). One method is based on heuristic search, which cannot guarantee optimality, and one is based on branch-and-bound, which can. The branch-and-bound program for CCS in the PHYLIP package is called PENNY (after the second author of [11], where branch and bound was proposed for this task). In [13], the performance of PENNY was compared with the ASP-based solutions developed there. PENNY was unable to prove optimality of solutions for any instances with more than 18 species.

We compare the performance of our method against the branch and bound implementation in PAUP. (We might expect PAUP (which is not free) to be faster than PHYLIP (which is free), because it has had more development effort, and this seems to be the case.) Figure 9 shows the frontier plots for the PAUP branch and bound implementation (PAUP-BnB), along with that for MX-Depth+(PP), and MX-Basic for comparison. Our MX-based solution is similar overall to PAUP, but comes closer to solving our largest – and presumably hardest – instances. For completeness, we also ran PAUP branch-and-bound on the instances produced by our pre-processing algorithm. Interestingly, PAUP performance improved, and with our pre-processing it solved all instances.

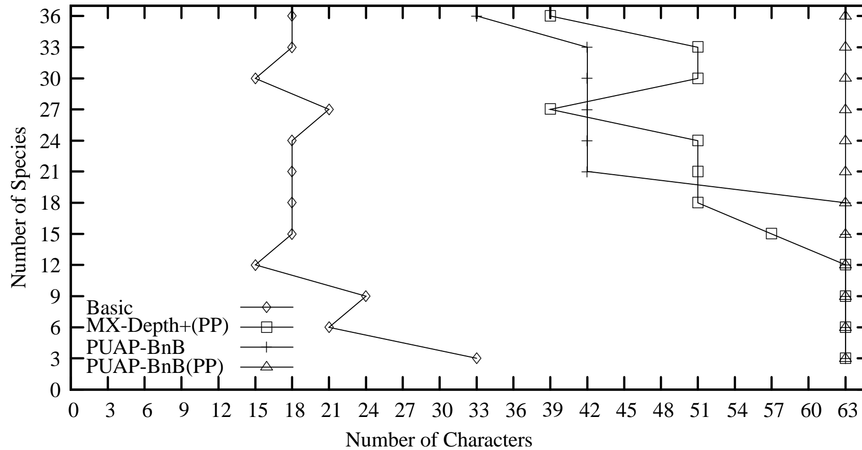

Fig. 9. Frontier for MX-Basic, PAUP-BnB, PAUP-BnB(PP), and MX-Depth+(PP)

8 Discussion

We have developed MX-based solutions to a phylogenetic inference problem. A simple and natural axiomatization gave much better performance than the only other declarative solution to this problem we know of, and more refined efforts produced a solution scheme with dramatically better performance. Ultimately, the kinds of methods and tools we use must be validated by demonstrating good performance on a wide range of problems and instances. Here we have tackled one interesting and non-trivial problem, and we believe what we have learned here will usefully inform more general solutions to a variety of phylogeny problems. Our instances here are derived from a single source of data, but we have taken pains to ensure that our instances and performance measures provide a good measure of performance progress. The improvements in running time, which are of many orders of magnitude, strongly suggest that our methods will be significant improvements by any other reasonable measure of performance.

MX *vs.* **ASP** We remind the reader that, while our Model Expansion based solutions, using the grounder MXG with ground solver MXC, perform dramatically better than the ASP solution we evaluate, our results here do not justify a claim of superiority of MX methodology or solvers over those of ASP. The best ASP solvers are quite powerful, and alternate approaches to ASP axioms, combined with pre-processing of the sort we do, might yield effective ASP-based solutions. (A comparison of MXG with several ASP solvers appears in [16].)

PAUP Heuristics An easy criticism of the work presented here is that the heuristic methods implemented in PAUP and PHYLIP often perform very well, and we have not compared our methods with those. In fact, the heuristic search method of PAUP finds optimum solutions for all of our instances well within our time limit. PAUP, of

course, does not know if they are optimal or not, and neither would a PAUP user. But, if optimality *per se* is not of much value to a biologist, why would they care?

One reply is that declarative solutions are potentially very useful. For example, user's don't worry about optimality because they are interested in criteria that are not captured by the cost function. With standard tools they are limited in how they can address these other preferences. Good declarative tools would allow them to add specific constraints, say that certain species should not be in the same sub-tree, and find solutions satisfying these (see also [7]). Another reason good declarative techniques could pay off is that problems of interest are often variants of a few core problems, and in some cases these are much more complex than the simple problem we studied here. An example is the Galled Tree Network Haplotyping Problem [10]. An instance is genotype data, which consists of vectors of conflated pairs of haplotypes. The task is to infer a set of haplotype vectors from the genotype data for which a parsimonious galled tree network exists. A galled tree network is a significantly more complex phylogeny than our binary CCS trees. The task of inferring small sets of haplotypes from genotype data, without worrying about phylogenies, is itself NP-hard (although SAT solvers do well at this [15], so MXG should also). Implementing a special-purpose program for this problem would be some effort, and finding simple heuristics which work reliably on large instances of such a problems seems unlikely. However, if we had effective declarative solutions for haplotype inference and construction of galled-tree networks, it would be easy to combine them and have a good start toward a solution for the larger problem.

Declarative Pre-Processing A point that may be argued against the progress we claim is that pre-processing of the instances before passing to the declarative solver was important, but this step is not declarative. Indeed, pre-processing is important in tackling many problems, seemingly a stumbling block for declarative methods. We point out the technique we used in our MX-Depth axiomatization (Figure 6), where we wrote an inductive definition to compute a set, and then used certain elements of that set in other axioms. MXG computes the defined set directly, while grounding, so the ground solver does not see this part of the axiomatization. Essentially any poly-time preprocessing can be carried out using this technique (not necessarily by the current version of MXG). With suitably refined languages, some users could accomplish such pre-processing more conveniently with declarative descriptions than with procedural code.

Conclusion We have not, yet, changed the way phylogenetic inference will be done in practice. But we have made progress that justifies our optimism regarding declarative approaches in general, and our MX-based tools in particular.

Acknowledgments We thank Jonathan Kavanagh for his instances, ASP programs, and bibtex file, and Eugenia Ternovska, Jan Manuch, and Sharon (Xiao-hong) Zhong for helpful discussions on axiomatizations and instances.

References

1. Gang Wu adn Jia-Huai You and Guohui Lin. Quartet-based phylogeny reconstruction with answer set programming. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 4(1):139–152, 2007.
2. David R. Bregman and David G. Mitchell. The sat solver mxc, version 0.5, 2007. Solver Description for the 2007 SAT Solver Competition.
3. D.R. Brooks, E. Erdem, J.W. Minett, and D. Rings. Character-based cladistics and answer set programming. In *Proc., Practical Aspects of Declarative Languages: 7th Int'l Symposium (PADL'05)*, pages 37–51, 2005.
4. W.H.E. Day, D.S. Johnson, and D. Sankoff. The computational complexity of inferring rooted phylogenies by parsimony. *Mathematical Biosciences*, 81:33–42, 1986.
5. L.C. Edwards-Ingram, M.E. Gent, D.C Hoyle, A. Hayes, L.I. Stateva, and S.G. Oliver. Comparative genomic hybridization provides new insights into the molecular taxonomy of the saccharomyces sensu stricto complex. *Genome Research*, 14:1043–1051, 2004.
6. N. Een and N. Sorensson. An extensible SAT-solver. In *Proc., Theory and Applications of Satisfiabiliy Testing, 6'th Int'l Conf. (SAT'03)*, pages 502–518. Springer, 2003. LNCS 2919.
7. E. Erdem, V. Lifschitz, L. Nakhleh, and D. Ringe. Reconstructing the evolutionary history of indo-european languages using answer set programming. *Proc., Practical Aspects of Declarative Languages: 5th Int'l Symposium*, pages 160–176, January 2003.
8. J. Felsenstein. Phylip home page, 1980. http://evolution.genetics.washington.edu/ phylip.
9. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. In *Proc., Ninth Int'l Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, pages 260–265. Springer, 2007. LNAI 4483.
10. Arvind Gupta, Jn Manuch, Xiaohong Zhao, and Ladislav Stacho. Characterization of the existence of galled-tree networks. *J. Bioinformatics and Computational Biology*, 4(6):1309–1328, 2006.
11. M.D. Hendy and D. Penny. Branch and bound algorithms to determine minimal evolutionary trees. *Mathematical Biosciences*, 59(2):277–290, 1982.
12. M. Hoffmann, N. Tripathi, S. R. Henz, A. K. Lindholm, D. Weigel, F. Breden, and C. Dreyer. Opsin gene duplication and diversification in the guppy, a model for sexual selection. *Proc. of the Royal Society of London Series B*, 274:33–42, 2007.
13. Jonathan Kavanagh, David G. Mitchell, Eugenia Ternovska, Ján Manuch, Xiaohong Zhao, and Arvind Gupta. Constructing Camin-Sokal phylogenies via answer set programming. In *Proc., LPAR 2006*, pages 452–466. Springer, 2006. LNCS 4246.
14. Yuliya Lierler and Marco Maratea. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In *Logic Programming and Nonmonotonic Reasoning, 7th International Conference*, volume 2923 of *LNCS*, pages 346–350, 2004.
15. I. Lynce and J.P. Marques Silva. Efficient haplotype inference with boolean satisfiability. In *Proc., AAAI'06*, 2006.
16. David Mitchell, Eugenia Ternovska, Faraz Hach, and Raheleh Mohebali. A framework for modelling and solving search problems. Technical Report TR 2006-24, School of Computing Science, Simon Fraser University, December 2006.
17. David G. Mitchell and Hector J. Levesque. Some pitfalls for experimenters with random SAT. *Artificial Intelligence*, 81(1,2), March 1996. *Special Issue* – Frontiers in Problem Solving: Phase Transitions and Complexity.
18. H. Nozaki, N. Ohta, M. Matsuzaki, O. Misumi, and T. Kuroiwa. Phylogeny of plastids based on cladistic analysis of gene loss inferred from complete plastid genome sequences. *J. Molecular Evolution*, 57:377–382, 2003.
19. D.L. Swofford. Paup* 4.0, 2001. Phylogenetic Analysis Using Parsimony (*and Other Methods).