

Constructing Camin-Sokal Phylogenies via Answer Set Programming

Jonathan Kavanagh, David Mitchell, Eugenia Ternovska, Jan Manuch,
Xiaohong Zhao, and Arvind Gupta

Simon Fraser University, Burnaby, B.C. V5A 1S6, Canada,
{jkavanag,mitchell,ter,jmanuch,xzhao2,arvind}@cs.sfu.ca

Abstract. Constructing parsimonious phylogenetic trees from species data is a central problem in phylogenetics, and has diverse applications, even outside biology. Many variations of the problem, including the cladistic Camin-Sokal (CCS) version, are NP-complete. We present Answer Set Programming (ASP) models for the binary CCS problem, as well as a simpler perfect phylogeny version, along with experimental results of applying the models to biological data. Our contribution is three-fold. First, we solve phylogeny problems which have not previously been tackled by ASP. Second, we report on variants of our CCS model which significantly affect run time, including the interesting case of making the program “slightly tighter”. This version exhibits some of the best performance, in contrast with a tight version of the model which exhibited poor performance. Third, we are able to find proven-optimal solutions for larger instances of the CCS problem than the widely used branch-and-bound-based PHYLIP package.

Keywords: phylogeny; maximum parsimony; Camin-Sokal; answer set programming

1 Introduction

Phylogenetics is the taxonomical classification of organisms based on their evolutionary distance. The central problem is that of constructing phylogenies (evolutionary trees) which postulate the most likely evolution of a set of extant species. This problem, and its variations, are widely applicable. For example, they play an important role in homology determination [1] and haplotyping [2], and can even be applied to the evolution of natural languages [3]. These variations are, for the most part, NP-complete, however their wide applicability requires the development of tools that help to overcome this intractability.

One of the most general and widely used forms of the problem is the maximum parsimony problem, where the goal is to find the smallest evolutionary tree (called the most parsimonious tree) that accounts for the diversity of the given species. The problem is specified as follows: A set of characters, each of which can take on a number of possible states, characterizes a group of species. The input is a set of species, given as character vectors. The goal is to construct a tree, with nodes labeled as character vectors, such that the node labels include all species, and the total number of character changes

along edges is minimized. Variations of the problem arise from different restrictions on character changes and different metrics on the minimization.

One traditional approach to solving maximum parsimony phylogeny problems, proposed by Hendy and Penny [4], is branch-and-bound (BNB). The BNB method involves constructing candidate trees in a depth-first manner, keeping track of the best trees found so far. The method benefits from heuristics which direct the search toward promising trees, and often finds optimal or near-optimal trees quickly. However, in the worst case, finding optimal phylogenies requires enumerating all trees. For data sets with many species, this is a daunting task since the number of trees grows exponentially in the number of species.

Due to the success of BNB, most present day software packages for phylogeny construction use this approach. The two most common packages, PHYLIP [5] and PAUP [6], use BNB to solve several variations of the phylogeny construction problem. Many of these packages sacrifice optimality in an effort to improve running times and it can be difficult to know if and when optimality has been reached. Other methods are needed, especially when optimality is required. In particular, we would like to develop a tool which can quickly compute optimal phylogenies for the maximum parsimony problem.

One strategy for dealing with NP-complete problems is to specify a solution's properties declaratively, and solve the declarative model with a general-purpose solver. This contrasts with the procedural approach used in BNB packages such as PHYLIP. In recent years, Answer Set Programming (ASP), a declarative approach, has gained increasing attention in tackling combinatorial search problems. Based on the stable model semantics of logic programming [7], it was identified as a new programming paradigm in 1999 [8, 9]. Problems are modeled in extended logic programming notation so that models of the logic program correspond to problem solutions. An answer set solver, such as *smodels* [10] or *Cmodels* [11] is used to compute answer sets, which represent models of the program and hence solutions. Phylogeny problems, with their combinatorially large search spaces, seem ideal candidates for ASP formulation.

In attempting to solve the maximum parsimony problem using ASP, we must choose a metric space in which to work. The ultimate goal is to use the Wagner metric [12], the most general metric, in which arbitrary mutations are allowed. In general, this metric yields extremely large search spaces and it seems difficult to find ASP models which perform satisfactorily on substantial data sets. Another well-established metric for the maximum parsimony problem is the cladistic Camin-Sokal (CCS) metric [13]. Though other models are more common, the CCS version is in use for specific applications (see [14–16]). In this version, the states of each character are ordered and all changes are to the next state in the order. These changes are irreversible. In the binary version, each character has two states. The CCS problem, even in the binary case, is NP-complete [17].

In this paper, we define an ASP model for the binary CCS problem. As a first step towards modeling this problem, we construct a model for the so-called *perfect phylogeny* problem. So long as the number of character states is constant, perfect phylogenies can be constructed in polynomial-time [18]. Our model to construct perfect binary CCS phylogenies performs well and we base our general CCS model on it. A

straightforward implementation gives a model with unsatisfactory running time, which is not uncommon in modeling NP-hard problems. To achieve a speed-up, we experimented with a considerable number of variations of the basic model, of which the seven most interesting are presented here. We present empirical evidence of the performance of the variations tried. Indeed, one of our contributions is a deeper understanding of how ASP models can be modified to potentially reduce running times. In particular, we propose the notion of *slightly tighter* models and show that a slightly tighter model can obtain better performance where a completely tight program fails to do so. We compare the performance of our ASP-based approach with PENNY, the BNB-based program from the PHYLIP package which constructs CCS phylogenies, on two biological data sets, including haplotype data for *Poecilia reticulata* (guppies) which was recently determined experimentally [19]. Our method finds, and proves optimality of, trees for subsets of this data with up to 27 species and 24 characters, or 36 species and 18 characters. PENNY cannot establish optimality of trees for subsets with more than 18 species within a two hour cutoff period. As the number of species grows, our methods seem to yield a more viable technique than PENNY to obtain optimal solutions. This work shows that by refining an original, basic declarative model, one can achieve performance comparable to, or faster than, procedural approaches.

2 Related Work

There are two main approaches (both yielding NP-complete problems) in character-based phylogeny construction: the “maximum parsimony” approach and the “maximum compatibility” approach. The goal of the former approach is to find a phylogeny for the input species with the minimum number of evolutionary changes, and is the approach we use in this paper. The goal of the latter approach is to construct a phylogeny using the maximum number of compatible characters. Brooks *et al.* present an ASP-based method for the “maximum compatibility” approach [20]. ASP solvers have also been used to construct “perfect phylogenetic networks”, from phylogenetic trees, to explain the evolution of Indo-European languages [3]. These networks extend given phylogenies with extra edges to create perfect phylogenetic networks which explain how languages are related, both through evolutionary changes and contact amongst different cultures.

This work focuses on the binary CCS version of the “maximum parsimony” approach. Recent biological uses of binary CCS include finding phylogenetic trees of *Saccharomyces sensu stricto* complex of yeast [14]. The model was also applied to DNA fragment data for individuals from *Pellia* genus, where state 1 represents the presence of the particular DNA fragment and state 0 represents its absence [16]. Nozaki *et al.* utilized the irreversibility in the CCS model because they observed that regaining of plastid genes is generally impossible during evolution [15].

As mentioned earlier, BNB is the most common method used in the “maximum parsimony” approach. In recent years, advances have been made to the basic BNB procedure, such as developing tighter lower bounds and better branching heuristics, e.g. [21–23]. These advances have helped improve the speed of BNB algorithms for finding the most parsimonious evolutionary trees. We are not aware of any recent developments in BNB techniques which are tailored directly to the CCS version.

3 Cladistic Camin-Sokal Problem

The general large parsimony problem is to construct a tree with the minimum number of mutations, or state changes. In binary CCS, a given character may mutate only once on any directed path from the root. Since this mutation is irreversible, we may assume that all mutations are from state 0 to state 1, and the root of the tree is the zero vector.

The usual convention for phylogenetic tree construction is to construct binary trees, to place all species at the leaves of the tree, and allow any number of characters to change along each edge. For our purposes, it is easier to model phylogenetic trees by dropping the assumptions that the tree is binary and that species appear only in leaves of the tree and enforcing exactly one character state change along each edge. Every leaf will be labeled by a species, but not necessarily vice versa. Since we limit ourselves to one change per edge rather than grouping changes on a single edge, there are more phylogenies possible with our convention. However, each of our phylogenies maps to a unique phylogeny in the standard convention, and each phylogeny in the standard convention can be mapped to a group of isomorphic phylogenies in our convention.

Consider the small 6 species, 5 character example in Figure 1. Notice that exactly one character changes on each edge (labeled by c_i to denote the i -th character has changed) and that species may occur at internal nodes. Both of these properties differ from the phylogenies produced by PENNY, but it is easy to convert trees from either format to the other.

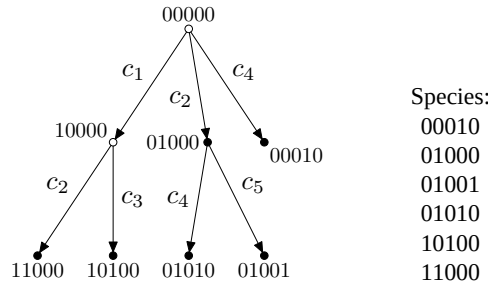


Fig. 1. Example of a binary CCS phylogenetic tree with minimum number of extra vertices. Vertices for species are marked with black dots.

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

Instance: A set S of n distinct species vectors from $\{0, 1\}^m$, and natural number B .

Question: Is there a directed tree $T = (V, E)$, such that: T is rooted at 0^m ; $S \subseteq V \subseteq \{0, 1\}^m$; every leaf in T is in S ; if $(v_1, v_2) \in E$, v_1 and v_2 differ in exactly one character; $\forall (v_1, v_2) \in E$, if v_1 has character state 1 for character c , then v_2 has state 1 for c (irreversibility); and $|V| \leq B$.

In a perfect phylogeny, each character mutation occurs only once in the tree. For binary CCS, this is equivalent to setting $B = m$.

4 Models

Before giving our ASP model for binary CCS, we describe a simpler model for the perfect phylogeny version. This model will help us illustrate our general model, given in Section 4.2, since we constructed the general model by extending the perfect phylogeny model to allow for individual characters to mutate multiple times.

4.1 Perfect Phylogeny Model

Our ASP model (program) takes as input a set of facts:

$$a(P, C, S).$$

where P is the species number ranging from 1 to n , C is the character number ranging from 1 to m , and S is the state, either 0 or 1, of that character.

Let predicate $c(C)$ be true if C is a character. We name each vertex, other than the root, by the character that has just changed from 0 to 1 (with the rule “ $v(C) :- c(C).$ ”). This naming scheme identifies each vertex, other than the root, with a character. Since the root has only character states of 0, and no character/vertex associated with it, we ignore it in our model. Thus we generate a forest. Connecting the root of each tree in the forest to our real root in post-processing produces the solution. To enforce this forest structure, we define relation $edge$ on vertices so that each vertex has at most one incoming edge:

$$\{ \text{edge}(V1, V) : c(V1) : V1 \neq V \} \vdash c(V).$$

It remains to ensure that each species is represented by a vertex in the tree. We do not have an explicit mapping between species and vertices, but rather ensure that exactly those characters with state 1 in a species appear as vertices along the path to it from the root. For this, we introduce two new relations: $above$ and $comparable$. Relation $above$ is the transitive closure of $edge$:

$$\begin{aligned} \text{above}(V, V1) &:- v(V), v(V1), V \neq V1, \text{edge}(V, V1). \\ \text{above}(V, V2) &:- v(V : V1 : V2), V \neq V1, V \neq V2, V1 \neq V2, \text{above}(V, V1), \text{edge}(V1, V2). \end{aligned}$$

Two vertices (characters) are comparable if one is above the other:

$$\begin{aligned} \text{comparable}(C, C1) &:- v(C), v(C1), C \neq C1, \text{above}(C, C1). \\ \text{comparable}(C, C1) &:- v(C), v(C1), C \neq C1, \text{above}(C1, C). \end{aligned}$$

We say two characters are shared if, for some species, they both take the value 1. We require that each pair of shared characters is comparable:

$$\vdash a(P, C, 1), a(P, C1, 1), C \neq C1, \text{not comparable}(C, C1).$$

The constraint ensures that, for each species, all characters with state value 1 must appear along a single path in the forest. However, this path should not contain characters which take value 0 for this species. For any species, we must prevent a character with state 0 occurring above a character with state 1:

$$:- a(P,C,0), a(P,C1,1), \text{above}(C,C1).$$

To complete the model, we must ensure a graph is a forest. To eliminate cycles, we add a constraint to prevent any two different characters from being above each other:

$$:- c(C), c(C1), \text{above}(C,C1), \text{above}(C1,C).$$

4.2 General Model

To handle the general problem, we modify our simple model for the perfect version. Predicates $c(C)$ and $sp(P)$ indicate that C is a character and P is a species. In this model, $c(C)$ no longer represents vertices. Since characters can change states multiple times, there is no longer a one-to-one correspondence between vertices and characters. We let $v(V)$ denote that V is a vertex. For m characters, we have $m + k$ vertices, where k is a number of extra vertices:

$$\begin{aligned} v(C) &:- c(C). \\ v(m + i) & \text{ (for } 1 \leq i \leq k) \end{aligned}$$

We create a mapping $m(V,C)$ from vertices to characters. Since each character must change to state 1 at least once, the first m vertices are mapped identically to the characters. The k extra vertices are free to be mapped to any character:

$$\begin{aligned} m(C,C) &:- c(C). \\ 1 \{ m(V,C) : c(C) \} & 1 :- v(V), V > m. \end{aligned}$$

As before, we create a forest by allowing each vertex to have at most one incoming edge and by forbidding cycles. We again use relation above to define paths amongst directed edges, but we modify it to be reflexive (i.e., the reflexive transitive closure of edge):

$$\text{above}(V,V) :- v(V).$$

This will simplify the remaining specification that each species is properly mapped to a vertex in the tree. In the general case, it is not enough to insist that each pair of characters which take the value 1 for a particular species are comparable, as multiple vertices can be mapped to the same character. For each species, we need a path from a root to a particular vertex such that the vertices in the path map exclusively to all of the characters which take value 1 for that species. To do this, we first introduce a mapping $p(P,V)$ from species to vertices:

$$1 \{ p(P,V) : v(V) \} 1 :- sp(P).$$

Suppose that species s is mapped to vertex v (i.e., $p(s,v)$ holds). If character c has state 1 for species s , we require some vertex above v to map to c (above is now reflexive, so v itself could map to c). Similarly, if character c has state 0 for species s , we require that no vertex above v maps to c . To model these requirements, we introduce relation $g(P,C)$, which is true of all characters C which have changed along the path to species P . For example, suppose that species s maps to vertex v which maps to character c . Then $g(s,c)$ holds if there is a vertex v_1 , above v , which maps to c :

$$g(P,C) :- sp(P), c(C), v(V1), v(V), p(P,V), m(V1,C), above(V1,V).$$

For each species P , we require that all characters with state 1 to be exactly those satisfying $g(P,C)$:

$$\begin{aligned} &:- a(P,C,1), \text{ not } g(P,C). \\ &:- a(P,C,0), g(P,C). \end{aligned}$$

This completes the model.

To solve an instance of this problem, we consider a sequence of instances of the model. In the first instance, we have no extra vertices (i.e., require a perfect phylogeny). In each successive instance, we increase the number of extra vertices by one. We continue until a solution, which must be optimal, is found.

4.3 Model Variations

We were unable to solve the CCS problem for either of the full data sets we obtained. In attempting to improve performance, we tried a number of variations of the basic model just described. We report the most interesting, based on these four strategies: adding redundant constraints; rephrasing constraints; tightening the program; and adding pre-processing steps to reduce the size of the search space.

Model A - Redundant constraints. A common strategy to improve performance of declarative models is to add redundant constraints. Adding redundant constraints to a logic program does not change the resulting answer sets, but can reduce the running time of the solver to find these solutions as candidate solutions can possibly be discarded more quickly.

A species can only be mapped to a vertex which takes state 1 for this species, so we added a constraint which explicitly prevents a species from being mapped to a vertex which maps to a character that has state 0 for this species:

$$:- sp(P), v(V), c(C), p(P,V), m(V,C), a(P,C,0).$$

Note that this constraint does not change the requirements for a candidate tree to be a solution. Without this constraint, it is still impossible for a species to be mapped to a vertex which represents a character for which the species takes a zero.

Model B - Rephrase constraints. Our basic model from Section 4.2 makes use of the fact that *above* is reflexive. This allows for a more concise definition of predicate *g*. We tested the idea of making *above* irreflexive and altering the program to account for this.

We remove the reflexive constraint from our basic model, and replace the rule to remove cycles with our new irreflexive constraint (this will also serve to prevent cycles):

$$:- v(V), above(V,V).$$

To modify relation $g(P,C)$ so that reflexivity is not needed, we introduce a new predicate, $ch_above(C,V)$:

$$\begin{aligned} \text{ch_above}(C,V) &:- c(C), v(V), m(V,C). \\ \text{ch_above}(C,V) &:- c(C), v(V), v(V1), V \neq V1, m(V1,C), \text{above}(V1,V). \end{aligned}$$

$\text{ch_above}(c,v)$ is true if vertex v maps to character c or there is a vertex v_1 above v which maps to c . Predicate g can now be simply defined as:

$$g(P,C) :- \text{sp}(P), c(C), v(V), p(P,V), \text{ch_above}(C,V).$$

Model C - Make program tight. If a logic program satisfies a syntactic condition called “tightness” or being “positive-order-consistent”, then its stable models can be characterized as the models of its completion (for a discussion of tight logic programs, see [24]). In general, Cmodels may make exponentially many calls to a SAT solver during one execution, but on tight programs it will only make one such call [11]. This may improve the overall run time.

To make our program tight, we create an ordering on the vertices, by replacing our identity mapping rule with a general mapping rule so each vertex can map to any character:

$$1 \{ m(V,C) : c(C) \} 1 :- v(V).$$

We modify the edge selection rule so that only edges from smaller vertices to greater vertices are allowed:

$$\{ \text{edge}(V1,V) : v(V1) : V1 < V \} 1 :- v(V).$$

Model D - Use preprocessing to reduce search space. In an attempt to reduce the size of the search space, we determined which characters are in conflict by constructing a conflict graph, in a preprocessing stage. A conflict graph has a vertex for each character, and edges between characters which are in conflict. Two binary characters are in *conflict* if there are three distinct species in which they take the state pairs ‘0-1’, ‘1-0’, and ‘1-1’. If two characters are in conflict, at least one of them will require an extra vertex in any resulting phylogenies. This is due to the fact that, from some parent node which has value 0 for both characters, both characters must change state along different paths to produce the ‘0-1’ and ‘1-0’ state pairs. However, one of these two characters must change again to obtain the ‘1-1’ state pair. In this variation, we restrict the possible characters to which the extra vertices can be mapped to those which are in conflict.

We define a predicate $\text{con}(C)$ to mean character C is in conflict. When adding extra vertices, we only choose among the characters in conflict:

$$1 \{ m(V,C) : \text{con}(C) \} 1 :- v(V), V > m.$$

Surprisingly, only Model A showed improved running time. We take Model A as our base model for these further improvements:

Model E Our attempt to make the program completely tight resulted in horrendous performance. Using a similar idea we made a ‘tighter’ program by creating an ordering on the extra vertices only. In this model, there can be edges from the first m vertices to any other vertices, but for each extra vertex numbered $k > m$, there can only be edges to vertices numbered greater than k .

To accomplish this, we create a predicate l which defines which pairs of vertices are allowed to have an edge between them. So $l(v_1, v)$ is true if $v_1 \leq m$ or $m < v_1 < v$:

$$\begin{aligned} l(V1, V) &:- v(V1), v(V), V1 \leq m. \\ l(V1, V) &:- v(V1), v(V), V1 > m, V1 < V. \end{aligned}$$

With l defined, we can define our new edge relation as:

$$\begin{aligned} \{ \text{edge}(V1, V) : v(V1) : V1 = V \} &1 :- v(V), V \leq m. \\ \{ \text{edge}(V1, V) : v(V1) : l(V1, V) \} &1 :- v(V), V > m. \end{aligned}$$

Model A+ A character cannot change to state 1 more than once along any given path, so we add another redundant constraint which prevents vertices which map to the same character from being comparable. Since the first m vertices map to unique characters, we only need to check a pair of vertices for this condition if at least one of the two vertices is an extra vertex. We add these rules in the preprocessing phase:

$$\begin{aligned} &:- v(V), v(V1), V > m, V1 \leq m, c(C), m(V, C), m(V1, C), \text{above}(V, V1). \\ &:- v(V), v(V1), V \leq m, V1 > m, c(C), m(V, C), m(V1, C), \text{above}(V, V1). \\ &:- v(V), v(V1), V > m, V1 > m, V = V1, c(C), m(V, C), m(V1, C), \text{above}(V, V1). \end{aligned}$$

We again note that these constraints do not alter the resulting answer sets.

Model MC For each pair of conflicting characters, at least one needs an extra vertex (i.e., the extra vertices must form a vertex cover of the conflict graph). Every vertex cover contains a minimal one. We make use of this fact and add mappings, in preprocessing, from the first extra vertices to characters so that a particular minimal vertex cover is a part of the extra vertices. Our iteration procedure begins with minimal covers with the smallest number of vertices, then continuing with these minimal covers and one additional extra vertex, and with minimal vertex covers with one more vertices than the minimum number, etc., until an optimal solution is found. Since we have explicitly chosen the characters some extra vertices map to, we have significantly reduced the number of possible mappings. However, we must (currently) manually find all minimal vertex covers in the conflict graph. For the reported data sets, this is not hard.

As mentioned, Models B, C, and D did not improve the performance of our basic model. The results of tests with the remaining variations are given in the next section.

5 Experimental Results

We present experimental results based on two sets of species data. One set, ‘fb65’, is haplotype information for *P. reticulata* containing 37 species and 468 characters with

5 states. Since the binary characters (SNPs) of this data are the most important, and our models are restricted to the binary case, we remove non-binary characters, leaving 449. The other set, ‘pin’, is from a taxonomic study of the *saccharomyces sensu stricto* complex [14], consisting of 20 species and 274 binary characters.

Identical characters form non-branching paths in phylogenies and all but one copy can be removed in preprocessing. Removed characters can easily be added to resulting trees in post processing. After this, the ‘fb65’ data set is left with 65 characters and ‘pin’ is left with 75.

Our ASP solver, in all reported results, was Cmodels-2 with zchaff. Cmodels regularly performed much better than smodels on our data. Our model, along with the input, was grounded using lparse [25]. This is always fast and is not included in the timing. Times shown are durations output by Cmodels. In the case of the general CCS problem, the time given is the sum of durations for the sequence of models as the number of extra vertices is increased. All runs were on a Sun Fire V20z, with Opteron 250 (2.4 GHz) CPU, and 4GB DDR1 RAM, running 64-bit Suse Linux Enterprise Server 9.

5.1 Perfect Phylogeny Model Results

To test the performance of our model, we derived subsets of ‘fb65’ which must have perfect phylogenies. We compared each pair of characters to find conflicting character pairs. For ‘fbpp1’, the first character in each conflicting pair was removed before continuing, leaving 38 characters. In ‘fbppr’, a random character from each conflicting pair was removed. This data set has 39 characters. For each, we solved subsets with 12, 24, 36, and 37 species (see Table 1). These results verified the correctness of the model. We found solutions to our full test data sets quickly, as hoped.

Sp	12	24	36	37
fbpp1	0.01	0.09	0.25	0.27
fbppr	0.01	0.07	0.28	0.29

Table 1. Running times (in seconds) to construct phylogenies using our perfect phylogeny model on two different data sets, as the number of species (Sp) is varied.

5.2 General Model Results

We also used the ‘fb65’ data to test the performance of our CCS model and its variations. The 37-species 65-character data set is too large for any solver we tested, so we considered a number of subsets, varying both the number of characters and the number of species. Our notation for these subsets is of the form ‘15s,39c’, for example, which represents the data set comprised of the first 39 characters from each of the first 15 species in our data.

To obtain the minimum number of extra vertices needed to produce a solution, we proceed incrementally from zero, adding a vertex whenever the solver returns false. We

do not use binary search since the most time consuming computation often occurs when the number of extra vertices is one less than optimal (see Table 2). To prove a solution is optimal, we must show that no trees can be constructed with one less than the optimal number of vertices. Adding more vertices to the optimal number increases the running time of the solver. For this particular example, where the optimal solutions have four extra vertices, our algorithm requires just 28 seconds to reach three extra vertices when proceeding linearly. Solving even one instance with more than four vertices would be less efficient.

#EV	0	1	2	3	4	5	6
Time	0	1	27	11931	4845	8930	9011

Table 2. Running times (in seconds) as the number of extra vertices (#EV) is increased, for ‘24s,24c’. Optimal solution has 4 extra vertices.

Table 3 lists the largest data sets for which we could find solutions, together with run times for our various models. These represent a frontier for the size of problems we are able to solve. We can construct a phylogeny for ‘36s,18c’ in seconds, but are unable to construct one for ‘36s,21c’ in two hours.

Data	#EV	Basic	A	E	A+	MC
3s,63c	0	0	0	0	0	0
6s,51c	0	3159	2503	1701	2398	2503
9s,39c	3	3191	4099	4850	1996	1500
12s,39c	3	670	579	496	863	131
15s,39c	3	1107	755	797	4093	73
18s,39c	3	1077	947	1050	3787	191
21s,21c	3	23	5	5	4	10
24s,24c	4	16803	10784	8634	1916	440
27s,24c	4	1990	1180	913	239	6
30s,18c	3	4	3	3	3	2
33s,18c	3	5	4	4	3	2
36s,18c	3	6	5	5	4	4

Table 3. Model comparison on ‘fb65’ data. Column ‘Data’ shows the largest data sets for which we could find solutions. Column ‘#EV’ shows the optimal number of extra vertices needed. The remaining columns give the running times, in seconds, for each model on each data set.

Since Model A+ generally has the best performance (Model MC performs best, but we exclude it as it requires manual steps), we used it to compare the performance of our ASP-based approach to that of PENNY. Table 4 gives run times for the two methods on our two data sets. For each data set, we construct several subsets, as before. The tables list the largest data sets for which a solution was found with our method (Data),

the optimal number of extra vertices (#EV), the total length of time, rounded to the nearest second, to find a solution with Model A+, and the time for PENNY, in seconds (PENNY). Times for PENNY are wall-clock times, as this package does not provide a timing function. An ‘X’ in the table denotes that PENNY did not halt within the two hour cutoff period.

Data	#EV	A+	PENNY	Data	#EV	A+	PENNY
3s,63c	0	0	0	3s,75c	0	0	0
6s,51c	0	2398	0	6s,57c	1	3	0
9s,39c	3	1996	0	9s,30c	4	317	0
12s,39c	3	863	1	12s,21c	4	10	46
15s,39c	3	4093	8	15s,10c	2	0	1350
18s,39c	3	3787	5700	18s,9c	3	8	X
21s,21c	3	4	X	20s,9c	3	7	X
24s,24c	4	1916	X				
27s,24c	4	239	X				
30s,18c	3	3	X				
33s,18c	3	3	X				
36s,18c	3	4	X				

(a)

(b)

Table 4. Comparison of performance of Model A+ and PENNY on two data sets: (a) fb65 - 37 species, 65 characters; (b) pin - 20 species, 75 characters. Running times are given in seconds. An ‘X’ in the table represents failure to return a solution within the two hour cutoff period.

The best performance we have obtained, to date, is with our Model MC, which, in pre-processing, modifies the ASP program to reduce the search space based on properties of the data. This approach drastically reduced running times, enabling us to solve larger problems within reasonable amounts of time. Figure 2 shows the frontier of largest subsets of our ‘fb65’ data solvable within two hours, for Models A+ and MC, as well as PENNY.

6 Discussion and Future Work

We presented ASP models for the binary CCS phylogeny problem, and for the restriction to the perfect phylogeny case. We examined the performance of a method using these models and the Cmodels solver on experimentally obtained biological data.

As we would hope, solutions for the polytime perfect phylogeny case were found very quickly. The general problem is much harder, and we know of no method, including ours, that can determine optimal phylogenies for the full data sets we use. We tried several ideas for improving the performance of our model. The best of our model variants can determine optimal phylogenies for larger fragments of the data (measured by number of species included) than can PENNY, the standard branch-and-bound program for the problem.

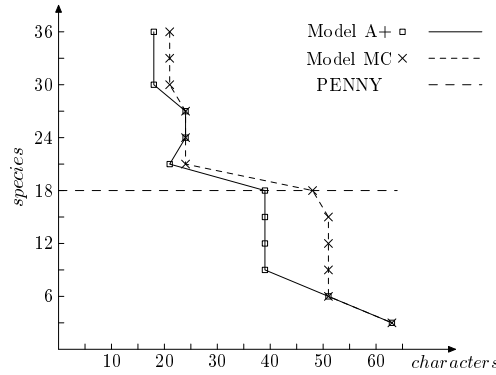


Fig. 2. Frontier for PENNY and Models A+, MC.

For data with n species, PENNY essentially enumerates all phylogenetic trees on n nodes, keeping track of the most parsimonious ones found so far (measured by total number of character changes on the tree). PENNY relies on good heuristics which direct the search toward the most promising trees, and trims subspaces which cannot contain optimal solutions. This approach has three main effects: it is very effective at finding good trees quickly, at least on small data sets; its performance is largely unaffected by the number of characters in the data; and it can establish optimality only for data sets with few species, since a large number of species implies too many trees to search.

Our ASP models fix a limit on the number of mutations in the tree to be found. Our method solves a series of models, beginning with the one which requires a perfect phylogeny, and then adding extra vertices until a phylogeny is found. This phylogeny must be optimal. In contrast with PENNY: Our general method relies on no domain-specific heuristics, and searches for a “perfect” tree, rather than enumerating trees; Our model involves mappings between both species and vertices, and vertices and characters, so performance is significantly affected by number of characters as well as number of species; The first phylogeny we find is optimal, and because we do not have to enumerate all trees, we can prove optimality for cases involving larger numbers of species.

Based on our experimental results and our understanding of the performance of the two methods, we conclude that declarative methods, and ASP in particular, are promising for solving hard phylogeny problems, especially when optimality is relevant. We understand that advances have been made in BNB algorithms since PENNY was developed, and that commercial software such as PAUP* 4.0 make use of these more advanced approaches. However, we also feel that time will permit advances in the declarative approach, both through the development of better models (perhaps with non-declarative components) and the development of faster ASP solvers.

Among our model variations, a few deserve attention. Model C was obtained by looking for a straightforward way to make the ASP program tight. Models for a tight program coincide with models for its completion, so some solvers could perform better on an equivalent tight program. Our change involved ordering the vertices. However, off-setting any benefit of tightness is the fact that the solver must also guess the ordering.

This significantly enlarges the search space. The resulting performance was very poor. However, Model E, a model which involved ordering the extra vertices only, exhibited good performance in general, and the best performance for some particular data sets. We have simultaneously increased the search space slightly and made the program slightly tighter. In doing so, we have exhibited better performance than a purely tight model or a model with a smaller search space.

The second best model, A+, was the result of adding two sets of redundant constraints to the model. This is a standard technique in constraint satisfaction (CSP) practice, for example, but less used in ASP.

The best performance was with Model MC. This model was based on pre-processing the data to obtain information which was used to revise the model, on an instance-by-instance basis, to reduce the search space. The fact that the performance of this version was significantly better than the others highlights a general problem for declarative approaches, namely, can we always solve problems with a purely declarative approach, or will we always need to consider such non-declarative components when tackling hard problems? Put another way, can we find a way to capture ideas such as the one used in version MC declaratively, or not; if so, how, and if not, under what conditions should we look beyond declarative methods?

Future work includes exploring more ways to improve the performance of our binary CCS models. Possible directions include further attempts at reformulations, incorporating more recent, faster SAT solvers into Cmodels, and exploring hybrid techniques which combine BNB and declarative approaches. The generation of minimal vertex covers for Model MC could be automated, perhaps by using another ASP program. Our CCS model could be extended to the non-binary case. We have made some initial explorations in this direction, but the straightforward model we constructed yields unsatisfactory performance and considerable work remains. Continuing to remove restrictions from our CCS model, until we reach the more general Wagner version of the problem, would make this technique applicable to a wider range of applications.

Acknowledgements We would like to thank Dr. Nikolay Pelov for his preliminary work in this project. We also wish to thank the three groups who supplied us with their data: Hellman *et al.*, Pacak *et al.*, and Edwards-Ingram *et al.*

References

1. Roderic, D., Page, M., Holmes, E.: Molecular Evolution: A Phylogenetic Approach. Blackwell Science, Oxford, UK (1998)
2. Gusfield, D.: Haplotyping as perfect phylogeny: conceptual framework and efficient solutions. In: RECOMB '02: Proc. of the sixth annual int'l conf. on Comp. biology. (2002) 166–175
3. Erdem, E., Lifschitz, V., Nakhleh, L., Ringe, D.: Reconstructing the evolutionary history of indo-european languages using answer set programming. Proc., Practical Aspects of Declarative Languages: 5th Int'l Symposium (2003) 160–176
4. Hendy, M., Penny, D.: Branch and bound algorithms to determine minimal evolutionary trees. Mathematical Biosciences **59**(2) (1982) 277–290
5. Felsenstein, J.: Phylip home page (1980) <http://evolution.genetics.washington.edu/phylip>.

6. Swofford, D.: Paup* 4.0 (2001) Phylogenetic Analysis Using Parsimony (*and Other Methods).
7. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. Proc., Int'l Logic Programming Conference and Symposium (1988) 1070–1080
8. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* **25** (1999) 241–273
9. Marek, V., Truszczyński, M.: In: Stable logic programming - an alternative logic programming paradigm. Springer-Verlag (1999) In: *The Logic Programming Paradigm: A 25-Year Perspective*, K.R. Apt, V.W. Marek, M. Truszczyński, D.S. Warren, Eds.
10. Niemelä, I., Simons, P., Syrjänen, T.: Smodels: A system for answer set programming. In: Proc. 8th Int'l Workshop on Non-Monotonic Reasoning, April 9–11, 2000, Breckenridge, Colorado. (2000)
11. Lierler, Y., Maratea, M.: 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. (2004) 346–350
12. Eck, R., Dayhoff, M.: Atlas of protein sequence and structure. National Biomedical Research Foundation (1966)
13. Camin, J., Sokal, R.: A method for deducing branching sequences in phylogeny. *Evolution* **19**(3) (1965) 311–326
14. Edwards-Ingram, L., Gent, M., Hoyle, D., Hayes, A., Stateva, L., Oliver, S.: Comparative genomic hybridization provides new insights into the molecular taxonomy of the *saccharomyces sensu stricto* complex. *Genome Research* **14** (2004) 1043–1051
15. Nozaki, H., Ohta, N., Matsuzaki, M., Misumi, O., Kuroiwa, T.: Phylogeny of plastids based on cladistic analysis of gene loss inferred from complete plastid genome sequences. *J. Molecular Evolution* **57** (2003) 377–382
16. Pacak, A., Fiedorow, P., Dabert, J., Szweykowska-Kulińska, Z.: RAPD technique for taxonomic studies of *Pellia* epiphylla-complex (hepaticae, metzgeriales). *Genetica* **104** (1998) 179–187
17. Day, W., Johnson, D., Sankoff, D.: The computational complexity of inferring rooted phylogenies by parsimony. *Mathematical Biosciences* **81** (1986) 33–42
18. Agarwala, R., Fernandez-Baca, D.: A polynomial-time algorithm for the perfect phylogeny problem when the number of character states is fixed. *SIAM Journal on Computing* (1994) 1216–1224
19. Hellman, M., Tripathi, N., Henz, S., Lindholm, A., Weigel, D., Breden, F., Dreyer, C.: Unpublished data. (2006)
20. Brooks, D., Erdem, E., Minett, J., Rings, D.: Character-based cladistics and answer set programming. *PADL* (2005) 37–51
21. Purdom, Jr., W., Bradford, P., Tamura, K., Kumar, S.: Single column discrepancy and dynamic max-mini optimization for quickly finding the most parsimonious evolutionary trees. *Bioinformatics* **2**(16) (2000) 140–151
22. Yan, M., Bader, D.A.: Fast character optimization in parsimony phylogeny reconstruction. Technical report (2003)
23. Moret, B., Tang, J., Wang, L., Warnow, T.: Steps toward accurate reconstruction of phylogenies from gene-order data. *J. Comput. Syst. Sci.* **65**(3) (2002) 508–525
24. Erdem, E., Lifschitz, V.: Tight logic programs. *Theory and Practice of Logic Programming* **3** (2003) 499–518
25. Syrjänen, T.: Lparse user's manual (1998) <http://www.tcs.hut.fi/Software/smodels/>.