CONSTRUCTING CAMIN-SOKAL PHYLOGENIES VIA ANSWER SET PROGRAMMING

by

Jonathan Kavanagh

B.Sc. (Honours), Memorial University of Newfoundland, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
Summer 2006

APPROVAL

Name:	Jonathan Kavanagh
Degree:	Master of Science
Title of thesis:	Constructing Camin-Sokal Phylogenies Via Answer Set Programming
Examining Committee:	Dr. Veronica Dahl Chair
	Dr. Arvind Gupta, Senior Supervisor
	Dr. David Mitchell, Co-Senior Supervisor
	Dr. Eugenia Ternovska, Supervisor
	Dr. Andrei Bulatov, Examiner
Date Approved:	August 4, 2006

Abstract

The problem of constructing a most parsimonious phylogenetic tree from species data, the maximum parsimony problem, is central to phylogenetics and has diverse applications elsewhere. Most natural variations of the problem, including the cladistic Camin-Sokal (CCS) version studied here, are NP-complete. The usual approach to solving these problems is branch-and-bound (BNB); packages using BNB often find approximate solutions quickly, but can establish optimality only for small instances.

We present a new approach to solving the CCS problem based on Answer Set Programming (ASP), a declarative approach based on stable model semantics of logic programming. ASP proves useful in tackling hard, combinatorial search problems. Along with our base model, we describe several variations which significantly affect performance. We compare our best versions with a commonly used BNB-based approach (PHYLIP's PENNY package), and conclude that ASP offers a viable approach to solving phylogeny problems, especially when optimality is relevant.

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

For my mother.

Acknowledgments

The work contained in this thesis was very much a collaborative effort, and I would like to thank all of the individuals involved. First and foremost, I would like to thank my supervisors: senior supervisor Dr. Arvind Gupta, co-senior supervisor Dr. David Mitchell, and supervisor Dr. Eugenia Ternovska. Their teaching and guidance during weekly meetings helped keep me on the right track, and their extra help editing and revising during paper submissions and thesis writing made such events worthwhile endeavors. I would also like to extend thanks to the two other members of our research team, Dr. Jan Manuch and Xiaohong Zhao, for their dedication and efforts in this project. They provided large amounts of their time to this project, and their help is greatly appreciated. I would also like to thank Dr. Nikolay Pelov for his preliminary work, for providing his initial ASP programs and tools, and for helping me make a smooth transition into this project. I would also like to thank Dr. Andrei Bulatov for volunteering to be the examiner for my thesis defense. As well, I would like to thank the three groups who supplied their data to us: Hellman et al., Pacak et al., and Edwards-Ingram et al. Their generosity enabled us to compare our variant models with existing techniques on actual biological data.

I would also like to acknowledge my family for their constant love and support. I would especially like to thank my parents, John and Geraldine, for teaching me the joy of learning and for providing the environment in which I grew.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 General 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, the process of determining if biological structures are alike due to shared ancestry [57], and haplotyping, the process of separating haplotypes from genotypes [32]. They can even be applied to the evolution of natural languages [17]. These variations are, for the most part, NP-complete. However, their wide applicability requires the development of tools that help to solve instances occurring in practice, in spite of 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 [37], 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 [19] and PAUP [64], use BNB to solve several variations of the phylogeny construction problem. For a given phylogeny problem, each software package is likely to return different trees. Analysis of these trees is required to determine which solution best describes the input data. 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 step-by-step procedural approach used in software packages such as PHYLIP and other traditional imperative programs. 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 [27], it was identified as a new programming paradigm in 1999 [52, 49]. Problems are modeled in extended logic programming notation, which looks similar to a Prolog program. The main difference between the two is that Prolog is used to answer queries (for program P and atom q, does $P \models q$?) while ASP generates models which satisfy the program. The syntax is simple and easy to use, and programs can often be written compactly using variables and recursion. Answer sets for the logic program correspond to problem solutions. After modeling, an answer set solver such as smodels [53], Cmodels [44], or dlv [43] is used to compute the stable models of the program and hence problem solutions. As faster solvers are developed, this method will continue to see performance improvements.

Phylogeny problems, with their combinatorially large search spaces, seem ideal candidates for ASP formulation. ASP algorithms can be viewed as a form of branch-and-bound, with the branching on truth values of ground atoms. However, the truth values of many

atoms are pre-computed efficiently by computing the well-founded model of the program; branch-and-bound is applied only where the truth values are still undefined. Moreover, solvers such as Cmodels are based on clause-learning SAT solvers, which recall derived information during the search to reduce the portion of the search space which is explicitly examined. A number of authors have suggested using ASP to construct character-based phylogenies. ASP-based approaches have been used for the construction of maximum compatibility phylogenies [6] and perfect phylogenetic networks [17]. In this thesis, we take the first steps towards finding maximum parsimony phylogenies within an ASP framework.

In attempting to solve the maximum parsimony problem using ASP, we must choose a metric space in which to work. Many metrics have been proposed and are currently in use. The ultimate goal is to use the Wagner metric [14], the most general metric, in which arbitrary mutations are allowed. In general, this metric yields extremely large search spaces and it is 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 [7]. Though other models are more common, the CCS version is in use for specific applications (see [15, 54, 55]). 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; a character cannot mutate back to a previous state in the order. In the binary version, each character has two states. The CCS problem, even in the binary case, is NP-complete [10]. 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 characters or character states is constant, perfect phylogenies can be constructed in polynomial-time [50, 1].

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. This is not uncommon in modeling NP-hard problems (for example, integer programming formulations also exhibit similar behaviour [66]). To achieve a speed-up, we experimented with a considerable number of variations of the basic model, of which the seven most interesting are reported in Chapter 4. Indeed, one of our major contributions is a deeper understanding of how ASP models can be modified to potentially reduce running times. In particular, we develop the notion of *slightly tighter* models (see Section 4.5), and show that a slightly tighter model can obtain better performance where a completely tight program fails to do so.

All of our experiments were performed on actual biologically significant data. In practice,

such data sets are quite large and, to the best of our knowledge, this is the first attempt to use ASP in finding any phylogenies for such data. We compare the performance of our ASP-based approach with PENNY [21], the BNB-based program from the PHYLIP package which constructs CCS phylogenies. The results show that the BNB method can quickly construct phylogenies with a low number of species and a high number of characters. This approach falters for larger numbers of species, even with relatively few characters. Our method does not perform as well for instances with a low number of species, but finds optimal trees for data with high numbers of species for which PENNY cannot establish optimality.

1.2 Overview

- In Chapter 2, we introduce the biological field of phylogenetics and the central problem of phylogeny construction. We formally define the binary CCS problem and give a survey of the BNB approach to the maximum parsimony problem.
- In Chapter 3, we introduce the key notions of answer set programming.
- In Chapter 4, we present our basic models and several of our most interesting variations.
- In Chapter 5, we give our experimental results and discuss their significance versus existing methods.
- In Chapter 6, we discuss related work and finish with some concluding remarks.

Chapter 2

Phylogenetics

2.1 Overview

In biology, *phylogenetics* is the study of the evolutionary relationships between a group of organisms, such as a set of species or populations of the same species. A *phylogeny* is a representation of the evolution of some set of organisms based on some mathematical model, often depicted by a tree. Aside from its central role in (evolutionary) biology, phylogenetic applications have been found in such fields as ecology [61], linguistics [22], and forensic studies [2].

There are many different approaches to phylogenetic tree construction, which arise from varying metrics of what a "good" phylogeny should look like. For example, should the phylogeny reinforce what is already believed by biologists, should it make use of all the properties of evolutionary history, or should it just be simple? Hence, the mathematical models used to construct phylogenies vary with one's needs. The most common approaches used today are cladistics, phenetics, and maximum likelihood. Cladistics, which makes use of shared traits to infer relationships amongst species, is the topic of Section 2.2. *Phenetics* [63], or *numerical taxonomy*, uses overall similarity, rather than evolutionary relations, to classify organisms. The most important algorithm to come from phenetics is *neighbor-joining*: a polynomial-time greedy algorithm which joins closely related species into clusters in a step-by-step fashion [59]. This method optimizes locally at each step and there is no guarantee that the fully constructed trees will be optimal. Although phenetics has been somewhat replaced by cladistics in recent years, neighbor-joining is still commonly used as it is a fast algorithm which often produces near-optimal results. *Maximum likelihood* is a

general statistical method which uses a pre-specified likelihood function and the probability distribution of the given data set to make inferences about the unknown parameters. The values of these parameters which maximize the likelihood are called Maximum Likelihood Estimates (MLEs). This purely analytic maximization procedure was first developed by Fisher between 1912 and 1922 [3]. The phylogeny construction problem can be worded as a maximum likelihood problem as follows: Given a model of sequence evolution and a tree, what is the likelihood that this tree accounts for the given species data? This approach to phylogenetic estimation has been extensively studied in recent years (eg. [20, 26, 39]).

When dealing with molecular data, the binary characters are often the most important [29]. Binary characters in genetic data are called Single Nucleotide Polymorphisms (SNPs). An SNP is a variation between members of the same species which occurs in a single nucleotide (A, C, T, or G) in a DNA sequence. SNPs make up approximately 90% of all human genetic variations and thus are often the focus of data analysis. In some situations, biologists remove characters which are not SNPs from their data before constructing phylogenies.

The first widely used software package for phylogeny construction was PHYLIP (PHYLogeny Inference Package) [19]. PHYLIP was first distributed in 1980 and features various programs for different problems and different algorithmic techniques. PENNY, one such program from the PHYLIP package, finds all most parsimonious trees for binary character data, using Camin-Sokal, Wagner, or mixed parsimony criteria. In Chapter 5, we will compare our results to PENNY. PHYLIP remains popular today due to the fact that it is freely distributed, can run on many different kinds of computer systems, and competes with PAUP* as the package responsible for the largest number of published trees. PAUP* (Phylogenetic Analysis Using Parsimony (and Other Methods)) [64] is a commercial software package which continues to use research advances to improve its performance. Version 4.0 introduced maximum likelihood and distance methods alongside its standard maximum parsimony approaches. MacClade, a Mac-based tool for phylogenetic analysis, provides elaborate visual representations of character-based phylogenies and allows for manual manipulation of the trees [47]. There are many other phylogenetic software packages in existence: The PHYLIP website provides a list of 265 known packages which solve some sort of phylogeny problem.

2.2 Character-based Cladistics

One of the most commonly used approaches to infer phylogenies is *cladistics*. Developed by Hennig [38], cladistics uses derived similarities, or shared traits, to infer evolutionary relationships amongst species (or *taxa*). These relations can be represented by a tree (called a *phylogenetic tree* or simply a *phylogeny*), where parent nodes denote ancestors of their child nodes, and edges denote genetic relationships. Edges often represent a genetic mutation which accounts for the difference in species. The standard convention is to place all species at the leaves of the tree with internal nodes denoting common (possibly extinct) ancestors.

The most common way to represent species' traits is with characters. In *character-based cladistics*, a set of characters is used to describe the similarities and differences between species. Each character has a set of possible states, and each species assumes some state for each character. For example, to construct a phylogeny of different species of birds, one of the characters could be feather-color with possible states red, blue, white, etc. Another character could represent the ability to fly. This is considered a binary character, as it has only two possible states: yes (for most birds) and no (for penguins). Each input species is defined by its particular states for each characteristic. From a set of input species, we must construct phylogenies. If the characters are chosen well, we may be able to deduce meaningful partial evolutionary trees [31]. In character-based cladistics, the characters change states on the edges of the tree.

With the set of taxa described in terms of character states, there are two main approaches for constructing meaningful phylogenies: the *maximum parsimony* criterion and the *maximum compatibility* criterion. The problem is NP-hard, even when the characters are binary, for either criterion [23, 11]. Our focus will be on the maximum parsimony approach, which attempts to construct a phylogeny with the minimum number of character state changes. This will be discussed in detail in the next section. The goal when using the maximum compatibility criterion is to construct a phylogeny with the maximum number of "compatible" characters. A character with k states is compatible if it changes state exactly $k-1$ times. Equivalently, a character is compatible if, for each state, each vertex labeled by that state forms a connected subtree. The score for a particular tree is the number of compatible characters it contains, and an optimal solution would be a tree with the highest possible score. An ASP-based method for the maximum compatibility problem was presented in [6].

2.3 Maximum Parsimony Problem

As mentioned in the previous section, the aim of the maximum parsimony approach is to construct a phylogeny with the minimum number of character state changes. The score for a candidate tree is the total number of character state changes that occur in the tree, and an optimal solution would be a phylogeny which accounts for the data and has the fewest possible number of state changes (called a *most parsimonious tree*). Viable mutations are a low probability event. Getting the same mutation twice independently is rare, especially in eukaryotic organisms (organisms in which the genetic material is located in the nuclei of its cells). However, on the time scales for which the problem is considered, mutations are quite possible (especially for molecular data). So the maximum parsimony assumption has a reasonable basis.

In the general maximum parsimony problem, the trees are unrooted, with the extant taxa labeled at the leaves. Each vertex is labeled by a set of character states, one for each character. The score of the tree is obtained by considering each adjacent pair of vertices, and scoring one for each character where the states differ. Let us assume we have a set of n taxa with m different characters $\{c_1, \ldots, c_m\}$. Each character c_i has $k_i \geq 2$ states, all of which must be represented in the taxa. A lower bound on the score of any candidate tree is $\sum_{i=1}^{m}(k_i - 1)$. This score corresponds to the situation where each character takes on each state value exactly once. If a phylogeny exists with such a score, we call it a *perfect phylogeny*, and the problem of constructing a perfect phylogeny, if one exists, from a set of species is known as the perfect phylogeny problem. If the number of characters [50] or character states [1] is constant, than the perfect phylogeny problem can be solved in polynomial-time. However, in the general sense, the perfect phylogeny problem is NP-complete [5].

Variations in the maximum parsimony problem can arise when different restrictions are placed on the allowable state changes. For example, each character could be assigned an *ancestral* state; a state this character must have in the common ancestor to all input taxa. This condition would force a root node, labeled by each character's ancestral state, and imply a direction on the edges, emanating from this root. A further restriction could be added so that a character cannot change back to a state it has previously visited (called a *back mutation*). Each variation in criteria can severely alter the resulting phylogenies, and many different versions have been used for different tasks.

2.4 Cladistic Camin-Sokal Problem

The cladistic Camin-Sokal (CCS) problem is one such variation of the general maximum parsimony problem, and is the version we will be focused on in this work. In the CCS version, every character has an ordering on its states (we can assume each character c_i has states ordered $0 < 1 < \cdots < k_i$), and changes are only permitted to the next state in the order. Hence, we require a directed tree, rooted at 0^m, the vector where each character is in state 0. Due to the ordering, each change is irreversible. A character cannot change from state 1 to state 0, for example. In the binary version of the CCS problem, each character has exactly two possible state values, 0 (the ancestral state) and 1 (the derived state). The CCS problem, even in the binary case, is NP-complete [10]. Several packages, including PENNY, PAUP*, and MacClade, discussed earlier, can be used to construct CCS phylogenies.

As mentioned, the usual convention for phylogenetic tree construction is to place all species at the leaves of the tree. If a species has the same states for each character as an internal node, they are linked by an unlabeled edge. For our purposes, it is easier to model phylogenetic trees without such changeless edges (we will see why in Chapter 4). We will construct trees with exactly one character state change along each edge. To guarantee equivalence of this definition to the common definition of phylogenetic trees, we drop the assumption that species appear only in leaves of the tree; every leaf will be labeled by a species, but not necessarily vice versa. By standard conventions, any number of characters can change along a single edge. Assume we have an edge where j characters change. Since we limit ourselves to one change per edge, we must represent this original single edge by a non-branching path with j nodes and j edges. This can be done in $j \times (j-1) \times \cdots \times 1 = j!$ ways. Hence there are more phylogenies possible with our convention, although 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.

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

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

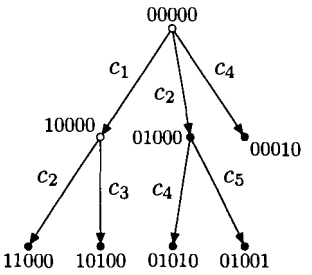
Question: Is there a directed tree $T = (V, E)$, such that: T is rooted at 0^m; $S \subseteq V \subseteq \{0,1\}^m$; every leaf in T is in S; if $(v_1, v_2) \in E$, v_1 and v_2 differ in

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 $B = m$, provided both states of each character occur in S.

2.4.1 Example

Consider the small 6 species, 5 character example in Figure 2.1. Notice that exactly one character changes on each edge (labeled by c_i to denote the i-th character has changed) and that species '01000' occurs at an internal node. Both of these properties differ from the standard phylogenies produced by most software packages (such as PHYLIP), but it is easy to convert trees from one format to the other.



Species:
00010
01000
01001
01010
10100
11000

Figure 2.1: Example of a binary CCS phylogenetic tree with minimum number of extra vertices. Vertices for species are marked with black dots.

2.5 Conflict Graphs

An important concept in maximum parsimony phylogeny construction is that of character conflicts. Two binary characters are in *conflict* if there are four distinct species in which they take the state pairs '0-0', '0-1', '1-0', and '1-1'. When the tree is rooted with ancestral states, as in the binary CCS problem, only the three state pairs '0-1', '1-0', and '1-1' are required to cause a conflict. For example, c_1 and c_2, from the example in Section 2.4.1, are in conflict since they contain the pair '0-1' in species 2, the pair '1-0' in species 5, and the pair '1-1' in species 6. Since both c_1 and c_2 start in state 0 at the root, they both must mutate to 1 along different paths to reach the states of '1-0' and '0-1', respectively.

At least one of c_1 and c_2 must mutate again to reach the '1-1' state pair. Thus, when two characters are in conflict, at least one of them must mutate twice. The only other conflict in the example is between c_2 and c_4.

The conflicts between characters can be represented visually with a *conflict graph*. In a conflict graph, each character is denoted by a vertex, and an edge between distinct characters denotes a conflict. Figure 2.2 presents the conflict graph for the example in Section 2.4.1.

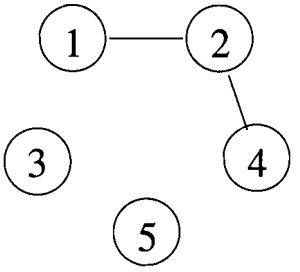


Figure 2.2: The conflict graph for example from Section 2.4.1.

Since a perfect phylogeny only allows a particular character to mutate to a particular state once, a perfect phylogeny cannot be constructed if any of the input characters are in conflict. Hence, for a particular set of input data, a perfect phylogeny can be constructed if, and only if, there are no edges in the conflict graph. This is one formulation of the Perfect Phylogeny Theorem [30, 31].

2.6 Branch and Bound

The branch-and-bound (BNB) problem solving method was first proposed by Land and Doig in 1960 for the linear programming problem [40]. Since then, it has become a very popular algorithmic method for finding solutions to combinatorial optimization problems. It has been applied to several NP-hard problems, including the knapsack problem [60], the travelling salesman problem [34], and, more recently, the maximum parsimony phylogeny problem [21]. The BNB method implicitly enumerates every point in the search space. The aim of BNB is to prune subspaces of the search space using bounds, so that not every candidate solution need be considered. The term 'branch' refers to the notion of splitting the search space into smaller subspaces, and recursively searching these smaller subspaces. The term 'bound' is used to denote the idea of finding upper and/or lower bounds on the costs of possible solutions within each subregion. For example, if the upper bound on the score of

some subregion A is less than the score of an already computed subregion in a maximization problem, we can disregard A. This idea, called *pruning*, is key to the success of the BNB method. If no branches get pruned, we will have effectively performed an exhaustive search.

BNB has been, and remains, a very popular approach to phylogeny construction problems. Several phylogenetic software packages, including the most widely used PHYLIP and PAUP* packages, use BNB to construct phylogenetic trees. PENNY uses BNB to solve the CCS problem.

Branch-and-bound can be applied to (rooted) phylogeny construction as follows: Take the first two species in the input set (say, S_1 and S_2) and construct a tree with them. The tree will have a root, two leaves, and (possibly) one internal node D. The internal node will contain all the character states common to both species, and the edges from D to the leaves will contain the changes unique to each respective species. There are three possibilities as to where to place the third species S_3: Add an internal node between D and S_1 and branch S_3 off of that; add an internal node between D and S_2 and branch S_3 off of that; or just simply branch S_3 from D. For simplicity, we add S_3 to the first possible place. Each new species added will have several possibilities, but we keep adding to the first such possibility until all species are placed, in a depth-first manner. The number of character state changes which have occurred in the tree represents the minimal score, m. We remove the last species from its first location and attach it to its second possible location. If we have reduced the score, we update m. We continue recursively, constructing candidate trees and checking their score against the best score found thus far. We keep track of the cost of each tree as we are building it. As soon as a species is added which increases the score of the partial tree to more than m, we can eliminate all possible trees which can be built from that state on, thus pruning a part of the search space. If we fully construct a tree with score m, we add it to the list of our candidate optimal trees. If we construct a tree with cost less than m, we update m and begin a new list of optimal trees. We continue until all trees have been constructed or discarded.

Obviously, this is the most simple way to traverse through the search space. An algorithm can use smarter heuristics to improve the performance, both for deciding which species to place next, as well as where to place it. Performance can be improved through branching (eliminating large regions of the search space) or bounding (lowering m as quickly as possible). The PENNY algorithm uses a heuristic based on the 'apparent promise' of each possibility [21]. When adding to a partial tree, PENNY tries adding each remaining

species to each possible place, and chooses the species which adds (at a minimum) the most to the current score and places it to cause the least increase in the score.

The BNB method gives a general approach for tackling combinatorial optimization problems, however there is no universal algorithm which can be used to efficiently branch in any such search space. Each particular problem requires its own branching and bounding algorithms to perform efficiently. As the maximum parsimony problem is widely studied, there has been much active research in the development of better BNB algorithms for constructing these phylogenies. The traditional approach, as used in PENNY, is to assign a cost equal to the minimum number of changes, or discrepancy, of each partial phylogenetic tree as it is being constructed. One recent approach involves using a single column discrepancy heuristic, which increases the cost by predicting the minimal additional cost needed to attach the species yet to be added to the partial tree [56]. That work also involved using a "dynamic Max-mini order of sequence addition" to allow for quick pruning of suboptimal subspaces, hence improving on both the branching and bounding. Since the scores of partial trees must be calculated many times during a single phylogeny construction, improving the speed at which these scores are computed could substantially improve the performance of BNB algorithms. In 2003, Yan and Bader proposed such an algorithm for fast, exact computation of tree cost [67].

Chapter 3

Answer Set Programming

3.1 ASP for Normal Logic Programs

Answer Set Programming (ASP) is a form of declarative programming oriented towards combinatorial search problems.

We begin our formal look[1] at ASP by first restricting our attention to *normal logic programs*, which consist only of rules of the form

$$p_0 \leftarrow p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\ , \tag{3.1}$$

where $n \geq m \geq 0$, and each p_i $(0 \leq i \leq n)$ is an *atom*. An atom consists of a predicate symbol followed by a parenthesized list of terms. A *term* is a constant, variable, or a function $f(t_1, \ldots, t_n)$ where f is a function symbol and $t_1, \ldots, t_n$ are terms (the standard convention is for variables to begin with a capital letter, and constants to begin with a lower case letter). For example, the atom $color(ball, red)$ could mean that the object "ball" has the color "red". The atom $color(V, red)$ could mean that some variable V is red. An atom with no variables is known as a *ground atom*. Atoms have two possible values, "True" and "False".

Given (3.1) as rule r, we let $head(r)$ denote the head, p_0, of r and $body(r)$ denote the body, $\{p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\}$, of r. The body can be broken into $body^+(r) = \{p_1, \ldots, p_m\}$ and $body^-(r) = \{not\ p_{m+1}, \ldots, not\ p_n\}$. Consider a set of atoms X. We can intuitively read rule r as: If all atoms in $body^+(r)$ are in X, and no atoms in $body^-(r)$ are in

[1] This discussion is based largely on [4] and [65].

X, then $head(r)$ must be included in X. If the body of a rule is empty, such as in "$p_0 \leftarrow$", the rule is called a *fact*, and is interpreted as "$p_0 \leftarrow True$". A rule is called a *constraint* if it has no head, as in

$$\leftarrow p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\ . \tag{3.2}$$

This rule r implicitly has $head(r) = False$. It is actually shorthand notation for the rule $f \leftarrow not\ f, p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n$, where f is a new atom distinct from each p_i.

A logic program is called *basic* if $body^-(r) = \emptyset$ for all its rules. For example, the program

$$\Pi_1 = \{p \leftarrow;\ r \leftarrow p;\ r \leftarrow q;\ s \leftarrow q\}$$

is basic. A set of atoms X is closed under a basic program Π if for any $r \in \Pi$, $head(r) \in X$ whenever $body^+(r) \subseteq X$. The *answer set* of a basic program Π, denoted by $Cn(\Pi)$, is the smallest set of atoms which is closed under Π. So $Cn(\Pi_1) = \{p, r\}$. Note that $\{p, r, s\}$ is also closed under Π_1, but is not an answer set since it is not the smallest such set.

Things are a little more complicated in the general case, so we introduce the concept of a reduct. The *reduct* of a program Π relative to a set of atoms X is

$$\Pi^X = \{head(r) \leftarrow body^+(r) \mid r \in \Pi, body^-(r) \cap X = \emptyset\}.$$

Hence, the reduct of a program is formed by removing all rules which contain a $not\ p_i$ for some $p_i \in X$ and removing $body^-(r)$ from all remaining rules. This reduct is obviously basic. A set X of atoms is an answer set (or *stable model*) of a program Π if $Cn(\Pi^X) = X$. Using reducts gives an intuitive meaning to the answer sets of a program. We would like an answer set X to satisfy a program Π, and for all atoms in X to be justifiable. For an atom $p \in X$, we can remove all rules with "$not\ p$" in the body, since they cannot be satisfied. For any "$not\ q$" in the remaining rules, we know $q \notin X$, and hence it can be dropped from the body. Hence we can reduce Π to the basic program Π^X. An answer set for Π^X will satisfy Π, so if $Cn(\Pi^X) = X$, then we can conclude that X satisfies Π, and contains only justifiable atoms. The heads of all fact rules (or just simply facts) of a program must appear in any answer set.

We illustrate the concept of reduct and answer set with an example (from [4]). Consider the program $\Pi_2 = \{p \leftarrow p;\ q \leftarrow not\ p\}$. There are four candidate sets for X, which we consider here:

X	Π_2^X	$Cn(\Pi_2^X)$
$\emptyset$	$p \leftarrow p$	$\{q\}$
	$q \leftarrow$	
$\{p\}$	$p \leftarrow p$	$\emptyset$
$\{q\}$	$p \leftarrow p$	$\{q\}$
	$q \leftarrow$	
$\{p,q\}$	$p \leftarrow p$	$\emptyset$

There is only one set X for which $X = Cn(\Pi_2^X)$, namely $X = \{q\}$, and this is our only answer set for this program. In general, a program can have many answer sets. It is also possible for a program to have no answer sets. Such is the case for $\Pi_3 = \{p \leftarrow not\ p\}$:

X	Π_3^X	$Cn(\Pi_3^X)$
$\emptyset$	$p \leftarrow$	$\{p\}$
$\{p\}$		$\emptyset$

We are now able to clarify why we call rules of form (3.2) "constraints". We know that $\{f \leftarrow not\ f\}$ has no answer sets, so if a set X contains all of the atoms in $p_1, \ldots, p_m$ and none of the atoms in $not\ p_{m+1}, \ldots, not\ p_n$, it cannot possibly satisfy the rule

$$f \leftarrow not\ f, p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n.$$

Hence constraint rules are used to eliminate sets of atoms which violate some property of our desired solutions.

3.2 Constants, Variables, and Grounding

As mentioned earlier, an atom is an n-ary predicate containing constants and/or variables. Variables allow programs to be written compactly and clearly. For example, we could have a large number of fact rules defining relationships between people and their parents, i.e. "$parent(frank, tom) \leftarrow$ ", "$parent(frank, luke) \leftarrow$ ", etc. If we want to establish a connection between siblings, we can do so using a single rule with variables, rather than writing a series of rules with constants:

$$sibling(X, Y) \leftarrow parent(Z, X), parent(Z, Y), X \neq Y$$

This can be read as "X and Y are siblings if they both have the same parent, Z". Variables make it easier to write programs, as a single rule with variables can be used to replace a set of rules with the same form but different constants. ASP solvers first replace variables in rules by the appropriate constants. This procedure is known as grounding. A *grounding* transforms a normal logic program into a ground logic program (a program with no variables) such that both programs have the same answer sets.

The *Herbrand universe* of a logic program is the set of all ground terms which can be constructed using the constants and function symbols of that program. The *Herbrand base* of a program is the set of all ground atoms which can be constructed using the predicates of the program and terms in the Herbrand universe. The set of ground instances of the rules of a program, which is produced by taking all terms in the Herbrand universe and substituting them into the variables in the rules, is called the *Herbrand instantiation.* Consider our parent-sibling program example:

$$parent(frank, tom) \leftarrow$$
$$parent(frank, luke) \leftarrow$$
$$sibling(X, Y) \leftarrow parent(Z, X), parent(Z, Y), X \neq Y$$

The Herbrand universe of this program is $\{frank, tom, luke\}$, since these are the only constants of the program. The Herbrand base contains 18 elements: 9 elements are comprised of the *parent* predicate, using each of the three constants in each of the two positions of the predicate ($parent(frank, frank), parent(frank, tom), parent(frank, luke)$, etc.), and similarly 9 elements for the *sibling* predicate. The Herbrand instantiation of this program is the first two rules of the program (since they are already ground), and the $3^3 = 27$ ground instances of the third rule (formed by choosing one of $frank$, tom, and $luke$ for each of the variables X, Y, and Z). For example, binding $luke$ to X, tom to Y, and $luke$ to Z gives the rule "$sibling(luke, tom) \leftarrow parent(luke, luke), parent(luke, tom), luke \neq tom$".

The answer sets of a logic program are equivalent to the answer sets of the Herbrand instantiation of the program. Hence each answer set of a program is a subset of its Herbrand base. It would appear that the simplest approach to computing answer sets of a logic program would be to construct the Herbrand instantiation of the program and then find its answer sets. This is not possible in practice, however, since the size of the Herbrand instantiation is often exponential in the size of the original program. Thankfully, many rules in the Herbrand instantiation have unsatisfied bodies, and hence can be discarded without

affecting the set of answer sets. Consider our parent-sibling example once again. Any of the grounded rules with, say, the atom $parent(tom, tom)$ in the body, will be unsatisfiable since there is no way to deduce $parent(tom, tom)$. As another example, say we add three facts which give the ages of the three people, say "$age(frank, 52) \leftarrow$; $age(tom, 20) \leftarrow$; $age(luke, 17) \leftarrow$ ". Then the constants 17, 20, and 52 would be added to the Herbrand universe and hence rules such as "$sibling(17, luke) \leftarrow parent(20, 17), parent(20, luke)$" would appear in the Herbrand instantiation. Obviously rules such as these will be unsatisfied.

To take advantage of this fact, Lparse [65], a program which is used to ground logic programs before a solver is used on the ground instantiation, divides predicates into two categories: domain and non-domain predicates. Domain predicates are essentially used to define the domain of a particular instance of a problem, while non-domain predicates are usually the interesting predicates which must be deduced or eliminated to give us solutions to our initial problem. In our example, *parent* and *age* are domain predicates, as they specify the possible values a variable can take (technically, *sibling* is also a domain predicate, because it can easily be deduced from *parent*). Only applicable domain predicates are used to ground each rule, and hence the size of the actual grounding is usually much smaller than the Herbrand instantiation.

To precisely define the notion of a domain predicate, we first need the concept of a dependency graph. The *dependency graph* of a program is a directed graph with signed edges: The nodes are the predicates of the program; there is an edge from p to q if p is the head and q is in the body of some rule in the program; this edge is negative if q is preceded by a *not*, and positive otherwise. A domain predicate p of a program is a predicate for which there are no cycles in the dependency graph which include p and contain a negative edge. Domain predicates are those which are defined either without recursion or through positive recursion, while non-domain predicates are those defined using negative recursion.

3.3 Example

Let us consider a practical example of writing a program in ASP: the graph 3-colorability problem (from [65]). For this problem we are given a graph as input, and must determine if it is possible to color each node of the graph, using one of three colors, so that no two adjacent nodes have the same color. This example will illustrate the standard "generate-and-test" approach used in writing ASP programs. In this approach, two sets of rules are

constructed. The first set of rules is used to generate all possible candidate solutions, and the second set, which usually consists of constraint rules, is used to eliminate the candidates which violate the conditions of the problem. The remaining candidates will be answer sets of the program and hence solutions to the problem.

To "generate" candidate solutions we write rules which define the domain. In the case of the 3-colorability problem (we can easily extend to the n-colorability problem), the domain is the input graph, given as a set of *node* and *edge* domain predicates, and the colors to be used:

$$node(a) \leftarrow ; \; node(b) \leftarrow ; \; node(c) \leftarrow ; \; node(d) \leftarrow .$$
$$edge(a,b) \leftarrow ; \; edge(b,c) \leftarrow ; \; edge(c,d) \leftarrow ; \; edge(d,a) \leftarrow .$$
$$c(red) \leftarrow ; \; c(blue) \leftarrow ; \; c(green) \leftarrow .$$

The first two rows of rules define the nodes and edges of the input graph. The last three fact rules define c, the unary predicate which denotes color. The possible values for c here are red, blue, and green. These facts vary with respect to the input. Also used as generators are the rules which assign a color to each node:

$$color(X,red) \leftarrow node(X), not\ color(X,blue), not\ color(X,green)$$
$$color(X,blue) \leftarrow node(X), not\ color(X,red), not\ color(X,green)$$
$$color(X,green) \leftarrow node(X), not\ color(X,red), not\ color(X,blue)$$

These three rules are used to ensure each node is colored. The rule "$color(X,red) \leftarrow node(X), not\ color(X,blue), not\ color(X,green)$" says that if X is a node, and it is not colored blue and not colored green, then it is colored red. These three rules ensure that for each node X in the candidate set, exactly one of $color(X,red)$, $color(X,blue)$, and $color(X,green)$ will be in the set.

We have now generated all possible colorings. What remains is to eliminate those candidate colorings which have colored adjacent nodes the same color. For this we define a single constraint rule, in the "test" group of the program:

$$\leftarrow edge(X,Y), c(C), color(X,C), color(Y,C)$$

This constraint rule disallows the situation in which two nodes are connected by an edge and colored by the same color. Any candidate solution which satisfies all of these rules is an answer set to the program and a solution to the 3-colorability problem, for the graph defined by the *node* and *edge* predicates.

3.4 Cardinality constraints

ASP solvers such as smodels and Cmodels allow the use of cardinality constraints, an extension of normal logic programs, which allow for more succinct programs. They are of the form

$$l\ \{q_1, \ldots, q_r\}\ u,$$

where $r \geq 1$, $q_1, \ldots, q_r$ are atoms, and l and u are lower and upper bounds on the cardinality of subsets of $\{q_1, \ldots, q_r\}$ which are to be satisfied in any corresponding answer sets. By default, $l = 0$ and $u = \infty$. Cardinality constraints can appear in the head or body of a rule. When a cardinality constraint appears in the head of a rule, it is called a *choice rule*. It is possible to abbreviate cardinality constraints using domain predicates. So, for example, we could use the expression

$$1\ \{coach(X) : parent(X, Y)\}\ 2$$

to mean "choose one or two coaches from all possible parents", rather than enumerating each parent as a possibility for coach:

$$1\ \{coach(frank), coach(bob), \ldots, coach(joe)\}\ 2.$$

The first predicate must be satisfied between l and u times, and the predicate(s) after the first colon are used to define the set of choices. When multiple predicates are used, each is separated by a colon. For example, the expression

$$1\ \{match(X, Y) : num(X) : letter(Y) : X \neq 6\}\ 1$$

means "match some number, other than 6, with some letter". A satisfying model must have exactly one pair of atoms for which the *match* predicate is true.

Recall our 3-colorability program from Section 3.3. In it, we used three rules to ensure that each node was colored exactly one color. Using a cardinality constraint, we can enforce this requirement with a single choice rule:

$$1\{color(X, C) : c(C)\}1 \leftarrow node(X)\ .$$

Here, if X is a node, then exactly one value of C must satisfy $color(X, C)$, where the possible values of C are defined by the domain predicate c. In general, the choice rule

$$1\{q_1, \ldots, q_r\}1 \leftarrow p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n$$

can be rewritten as the following normal logic program rules:

$$q_1 \leftarrow not\ q_2, \ldots, not\ q_r, p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n$$
$$\ldots$$
$$q_j \leftarrow not\ q_1, \ldots, not\ q_{j-1}, not\ q_{j+1}, not\ q_r, p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n$$
$$\ldots$$
$$q_r \leftarrow not\ q_1, \ldots, not\ q_{r-1}, p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n$$

The "at most one" choice rule

$$0\{q_1, \ldots, q_r\}1 \leftarrow p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n$$

can be rewritten as these normal logic program rules (where f is a unique atom):

$$f \leftarrow not\ q_1, \ldots, not\ q_r, p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n$$
$$q_1 \leftarrow not\ f, not\ q_2, \ldots, not\ q_r, p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n$$
$$\ldots$$
$$q_j \leftarrow not\ f, not\ q_1, \ldots, not\ q_{j-1}, not\ q_{j+1}, not\ q_r, p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n$$
$$\ldots$$
$$q_r \leftarrow not\ f, not\ q_1, \ldots, not\ q_{r-1}, p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n$$

3.5 Tight Logic Programs

Let Π be a finite normal logic program and a be an atom. We use $Comp(\Pi, a)$ to denote the propositional formula[2]

$$a \equiv body(r_1) \vee \cdots \vee body(r_k),$$

where the disjunction extends over all rules r_i in Π where a is the head ("$a \leftarrow body(r_i)$"). If the body of any of these rules is empty, then $a \equiv True$. If a does not appear in the head of any rule in Π, then $Comp(\Pi, a)$ is $a \equiv False$ or $\neg a$. The *completion* of Π is the set of all formulas $\{Comp(\Pi, a) | a$ is an atom in $\Pi\}$ [16, 46]. We can extend the definition of completion to programs with constraint rules by adding $\neg body(r_i)$ to $Comp(\Pi)$ for each constraint rule r_i (where $body(r_i)$ has been converted to a propositional formula) [45]. For example, if program Π contains these three rules:

[2]The body of a rule can be considered propositional if we interpret the commas as conjunctions and "*not*"s as "$\neg$"s. So $body(r_i) = p_1, p_2, not\ p_3$ becomes $body(r_i) \equiv p_1 \wedge p_2 \wedge \neg p_3$.

$$p \leftarrow q, r, not\ s$$
$$p \leftarrow q, not\ s$$
$$\leftarrow q, not\ r\ ,$$

then $Comp(\Pi) = \{p \equiv (q \wedge r \wedge \neg s) \vee (q \wedge \neg s), \neg q, \neg r, \neg s, \neg(q \wedge \neg r)\}$. The completion of a program is written in standard propositional logic, and hence negative literals can be derived using standard rules and the completion can be computed using a SAT[3] solver.

The completion concept is one way of dealing with inferring negative information in a program [33]. It is one way of making explicit the Negation as Failure rule [8]: $\neg A$ succeeds iff A fails. This is performed by turning the "if"s of the program into "if and only if"s, as described in the definition. The completion semantics, along with answer set semantics, are two of the most widely used interpretations of the meaning of negation as failure. Clark showed that every answer set of a logic program is also a model of the completion of the program [8]. The converse is not true, in general. However, if a program is tight, then every model of its completion is also an answer set.

To define the tightness property, we will first need the concept of a *parent*. Let X be a set of atoms and $p_1, p_2 \in X$. We call p_1 a *parent* of p_2 relative to program Π and set X if there is a rule r in Π such that

1. $X \models body(r)$
2. $p_1 \in body^+(r)$
3. $head(r) = p_2$.

Consider, for instance, the program $\Pi = \{p \leftarrow not\ q;\ q \leftarrow not\ p;\ p \leftarrow p, s\}$. The parents of p relative to Π and the set $X = \{p, q, s\}$ are p and s (all three conditions are satisfied for the third rule, for both p and s). However, p has no parents relative to Π and $\{p, q\}$.

A program Π is *tight*[4] on a set of atoms X if there is no infinite sequence $p_1, p_2, \ldots$ of elements of X such that, for every i, p_{i+1} is a parent of p_i relative to Π and X. In other words, a (finite) program is tight on a set of atoms X if its parent relation on elements of X contains no cycles. We say a program is tight if it is tight on the set of all atoms

[3]The propositional satisfiability problem (SAT) is to decide whether a given propositional formula can be satisfied by some assignment of the boolean variables in the formula. It was the first known NP-complete problem [9] and still receives much research attention.

[4]The concept of tightness was originally called *positive-order-consistent* when first defined by Fages[18].

contained in the program. For instance, in the program $\Pi_1 = \{p \leftarrow q;\ q \leftarrow p\}$, p is a parent of q and q is a parent of p (relative to $\{p, q\}$). This cycle prevents the program from being tight. The completion of Π_1 is $\{p \equiv q, q \equiv p\}$, which has two models ($p \equiv q \equiv True$ and $p \equiv q \equiv False$), but the only answer set of Π_1 is $\{\}$ (i.e. p and q are false).

When a program is tight, any set of atoms which satisfies the program's completion will also be an answer set of the program [18]. Hence, one way to compute the answer sets for a tight program is to form the completion of the given program, and use a SAT solver to compute the models of this completion. These models will correspond to the answer sets of the original program. In fact, this is the approach used by Cmodels when computing answer sets of tight programs.

If a program Π is not tight, the completion of Π can be modified with a set of loop formulas (LF) so that the models of $Comp(\Pi) \cup LF$ correspond to the answer sets of Π [45, 42]. We saw above that program Π_1 had a cycle, or "loop", containing p and q. In propositional logic, one can make any assumptions about the truth values of the atoms in the loop so long as the constraints are still satisfied. We can assign both p and q to "True" or both to "False". In answer set semantics, one cannot assume an atom is true without justification. If we added the rule "$(p \vee q) \supset False$" to the completion, then we could no longer assign p and q to "True", and the only remaining model would correspond with the single answer set: both p and q are false. This added rule is known as a loop formula. In general, if $p_1, \ldots, p_n$ are in a loop, then "$(p_1 \vee \cdots \vee p_n) \supset False$" is the corresponding loop formula. If we add a loop formula for each loop in the parent relation to the set LF, we can construct a set of propositional formulas, $Comp(\Pi) \cup LF$, for which the models are in one-to-one correspondence with the answer sets.

It would appear that we can take any non-tight program, compute its completion and loop formulas, and use these to find models which correspond to answer sets. However, this may be impractical as there are potentially an exponential number of loops in a logic program [45]. SAT-based ASP solvers like Cmodels and ASSAT use iterative procedures to handle non-tight programs. The basic procedure is to set the completion of the program to set T and find a model, X, for T. If there is no such model, terminate with failure. Otherwise, check if the model is an answer set, and halt if so. If not, selectively add one or more loop formulas to T and start again. Determining which loop formulas to add is the main difference between ASSAT and Cmodels. ASSAT chooses to add a single loop formula which is not satisfied by X. Cmodels determines which set of atoms, X^-, in X are not

derivable from the reduct Π^X. It adds loop formulas for each of the maximal (under subset inclusion) loops in X^-. Eventually, a model for T will be obtained which corresponds to an answer set.

Since a tight program requires just one call to a SAT solver, it potentially can perform better than a non-tight program which has the same answer sets. Converting a non-tight program into a tight program, without changing the answer sets, is a common technique employed when trying to improve the performance of an ASP model. We discuss the application of this technique to our problem in Section 4.4.

3.6 Notation

For the remainder of this work, we will use the notation of the standard ASP solving tools (Lparse, smodels, Cmodels, dlv, etc.) when discussing logic programs. The main differences are that ":-" is used instead of "$\leftarrow$", and a period is used to denote the end of a rule. So the rule

$$color(X, blue) \leftarrow node(X), not\ color(X, red), not\ color(X, green)$$

will now be written in the form

```
color(X,blue) :- node(X), not color(X,red), not color(X,green).
```

For facts, we drop the arrow altogether. So, for example, "$parent(frank, tom) \leftarrow$ " becomes "parent(frank,tom).". The solvers we used allow for other convenient notations which appear in our programs, including "N < M" and "N <= M", to represent the mathematical inequalities $N < M$ and $N \leq M$, respectively, and "N != M" to represent $N \neq M$.

Chapter 4

Models

4.1 Model Overview

Due to the pervasive need to solve NP-hard problems, a number of modeling frameworks have been developed. One of the best known and most widely used is integer linear programming (ILP). More recently, constraint satisfaction problem (CSP) solvers and now, ASP solvers, have been developed. There is extensive on-going research in determining the scope of their performance.

It is often challenging to find a model, within any of these frameworks, which performs well. Often, the most straightforward formulation can lead to excessive computation time, and "good" reformulations could potentially have negative affects on run times [66]. The ILP community has developed many techniques, including relaxation and mixed integer linear programming (MILP), to improve the performance of their models. A similar effort is underway for CSP and ASP, and some general techniques, such as adding redundant constraints, have already been explored.

Our strategy was to first add a further restriction to the binary CCS problem (recall the formal definition from Section 2.4); one where each character can mutate only once. This version, known as the perfect phylogeny problem, can be solved in polynomial time provided the number of character states is constant [1]. After formulating and some reformulating, we arrived at the model presented next in Section 4.2. This model will help us illustrate our general model, given in Section 4.3, since we constructed the general model by extending the perfect phylogeny model to allow for individual characters to mutate multiple times.

We found that our initial model for the binary CCS problem did not exhibit great performance. A major thrust of our work is experimenting with model variations to reduce running times. Initially, we were guided by techniques that are commonly employed in CSP. We also explored models which exploited tightness, a technique unique to ASP. While these models were also unsatisfactory, they led to the development of a new technique: the construction of slightly tighter models. The most interesting model variations are highlighted in Sections 4.4 and 4.5.

4.2 Perfect Phylogeny Model

Our ASP model (program), for the perfect binary CCS phylogeny problem, takes as input a set of domain predicate 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:

```
c(C) :- a(P,C,1).
```

We name each vertex, other than the root, by the character that has just changed from 0 to 1. (Recall that changes from 1 to 0 are not allowed):

```
v(C) :- c(C).
```

This 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. Hence, the answer sets of our program will correspond to forests. Connecting the root of each tree in the forest to our actual root in post-processing produces the solution to the original problem. To enforce this forest structure, we define relation edge on vertices so that each vertex has at most one incoming edge:

```
{ edge(V1,V): v(V1): V1 != V } 1 :- v(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:

```
above(V,V1) :- v(V), v(V1), V != V1, edge(V,V1).
above(V,V2) :- v(V), v(V1), v(V2), V != V1, V != V2, V1 != V2, above(V,V1), edge(V1,V2).
```

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

```
comparable(C,C1) :- v(C), v(C1), C != C1, above(C,C1).
comparable(C,C1) :- v(C), v(C1), C != C1, above(C1,C).
```

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:

```
:- a(P,C,1), a(P,C1,1), C != C1, 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 being above a character with state 1:

```
:- a(P,C,0), a(P,C1,1), above(C,C1).
```

Since we want a forest, we add a constraint to prevent any two different characters being above each other:

```
:- v(C), v(C1), above(C,C1), above(C1,C).
```

This constraint removes cycles. This completes the model. The entire code for this model is given in Appendix A.1.

4.3 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[1]:

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

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:

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

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):

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

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:

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

[1] Throughout this chapter, italicized letters, such as m, k and i, are used to denote numeric variables which change for each problem instance. These values are instantiated with numeric constants during preprocessing, before each call to the ASP solver.

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 some vertex v. Then, if c is a character, 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).
```

Figure 4.1 shows the situation in which g(s,c) is true:

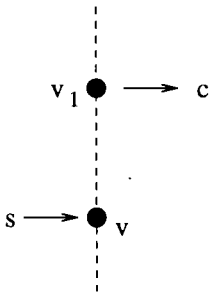


Figure 4.1: Species s maps to vertex v, and vertex v_1 maps to character c. Vertex v_1 is somewhere above v. Note that v_1 could equal v.

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

```
:- a(P,C,1), not g(P,C).
:- a(P,C,0), g(P,C).
```

This completes the model. The complete code for this program is given in Appendix A.2.

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.

4.4 Basic Model Variations

In an attempt to improve the performance of our method, we tried a number of variations of the basic model from the previous section. Our variations were based on the following four strategies: adding redundant constraints; rephrasing constraints; adding preprocessing steps to reduce the size of the search space; and tightening the program. The first three of these are common in the constraint satisfaction field and the last applies only to ASP. Each variation was tested extensively, using our test data, to determine if any performance improvements had been obtained.

4.4.1 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. To add a redundant constraint to a program, one must find a property which is implicitly true in the rules and add a constraint rule which explicitly states it.

Notice that, in our model, it is impossible for a species to be mapped to a vertex which maps to a character for which the species has state 0. This property is implicitly prevented by predicates g and above and the constraint ':- a(P,C,0), g(P,C).'. While the solver would never return a solution with such an occurrence, making this property explicit can potentially help the solver catch and discard these bad mappings sooner.

Hence, in this variation, we add 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.

4.4.2 Model B - Rephrase constraints.

At times, performance can be improved by changing the way properties are formulated in a program. Our basic model from Section 4.3 makes use of the fact that above is reflexive. This allows for a more concise definition of predicate g. However, we could have kept above

reflexive. This would reduce the number of atoms derived from the above predicate, but would require a slightly more complicated definition of g. We tested whether making above irreflexive and altering the program to account for this would improve performance.

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):

```
ch_above(C,V) :- c(C), v(V), m(V,C).
ch_above(C,V) :- c(C), v(V), v(V1), V != V1, m(V1,C), above(V1,V).
```

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) :- sp(P), c(C), v(V), p(P,V), ch_above(C,V).
```

4.4.3 Model C - Make program tight.

Recall from Section 3.5 that, if a logic program satisfies the syntactic condition called "tightness", then its answer sets can be characterized as the models of its completion. When Cmodels is run on a tight program, it need only call the SAT solver once[44]. This could potentially improve the overall run time.

Our basic model is not tight since the above predicate, the transitive closure of edge, causes cycles in the parent relation. This happens despite the fact that we have a constraint which removes cycles from our forest. Edges can connect any pair of nodes, since there is no ordering on our set of vertices. For two vertices v1 and v2, above(v1,v2) will be a parent for above(v2,v1) and vice versa. In fact, any set of distinct vertices $\{v1, v2, \ldots, vn\}$ causes a loop, $\{above(v1,v2), above(v2,v3), \ldots, above(vn,v1)\}$. To remove these loops and make our program tight, we create an ordering on the vertices. First, we replace 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).
```

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

```
{ edge(V1,V): v(V1): V1 < V } 1 :- v(V).
```

The conversion to a tight program comes at a cost. Previously, the only non-trivial part of our vertex-character mapping was the extra vertices, as they were free to be mapped to any character. By allowing each vertex to map freely to any character, we are greatly increasing the possible number of mappings and hence greatly increasing the size of the search space.

4.4.4 Model D - Use preprocessing to reduce search space.

By reducing the number of rules we pass to the solver, we can potentially reduce the amount of work the solver must do to find answer sets. Our first attempt at this was, in a preprocessing stage, to determine characters which are in conflict by constructing a conflict graph (as defined in Section 2.5). If two characters are in conflict, at least one of them will require an extra vertex in any phylogeny. In this variation, we attempt to restrict the possible characters to which the extra vertices can be mapped. We only allow each extra vertex to be mapped to a character which is in conflict, although any number of extra vertices can map to a particular character.

We define a predicate 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) : con(C) } 1 :- v(V), V > m.
```

4.5 Variations on Model A

Surprisingly, of the four variations mentioned above, only Model A showed improvements in running time on our data sets. We take Model A as our base model for these further improvements:

4.5.1 Model E

Our attempt to make the program completely tight resulted in significantly worse performance. Using a similar but less extreme idea, we made a "slightly tighter" variation 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. This model is slightly tighter than our base model since it has fewer loops. In the basic model, loops can be formed with above, and any set of vertices. Only the m base vertices can cause loops in Model E. The cost of this slight tightening is a slightly larger search space.

To implement this idea, 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 $m < v_1 < v$:

```
l(V1,V) :- v(V1), v(V), V1 <= m.
l(V1,V) :- v(V1), v(V), V1 > m, V1 < V.
```

With l so defined, we redefine our edge relation as:

```
{ edge(V1,V): v(V1): V1 = V } 1 :- v(V), V <= m.
{ edge(V1,V): v(V1): l(V1,V) } 1 :- v(V), V > m.
```

4.5.2 Model A+

Another property implicit in our model is that a character cannot change to state 1 more than once along any given path. If two comparable vertices did map to the same character, we would need one more than the optimal number of extra vertices to account for this redundant change. Hence, the solver will not be able to violate this property unless more than the optimal number of extra vertices are added to the model. In an attempt to help the solver notice these violations sooner, we make this property explicit.

In this variation, we add another redundant constraint which explicitly prevents comparable vertices from mapping to the same character. Since the first m characters 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. Since m, the number of characters, varies, we add these rules in the preprocessing phase:

```
:- v(V), v(V1), V > m, V1 <= m, c(C), m(V,C), m(V1,C), above(V,V1).
:- v(V), v(V1), V <= m, V1 > m, c(C), m(V,C), m(V1,C), above(V,V1).
:- v(V), v(V1), V > m, V1 > m, V = V1, c(C), m(V,C), m(V1,C), above(V,V1).
```

We again note that these constraints are redundant, and do not alter the resulting answer sets.

4.5.3 Model MC

For each pair of conflicting characters, at least one needs an extra vertex. Hence, the extra vertices must form a vertex cover of the conflict graph. We can modify our model to find a phylogeny using this idea, in the following way: First, we find a vertex cover of the conflict graph. Assume the cover contains j vertices. We then take the first j extra vertices in our model and explicitly map them to these j characters represented by the vertex cover. We now perform our iterative procedure as before: we attempt to construct a phylogeny with these j extra vertices; if one does not exist, we add one more extra vertex, which is free to map to any character, and use the solver to attempt to find a solution, and so on. Eventually, the solver will find a phylogeny with the j explicit extra vertices and, say, q more. Since a number of the extra vertices are mapped explicitly to particular characters, this procedure tends to construct a phylogeny much quicker than the variations discussed previously. However, though this procedure is guaranteed to find a phylogeny, the solution will not necessarily be optimal.

To ensure we find an optimal phylogeny, we must (potentially) perform the above procedure multiple times. Since the extra vertices must form a vertex cover on the conflict graph, we will need at least as many extra vertices as the size of a minimum vertex cover of the conflict graph. Assume the minimum vertex cover is of size w. If we take a minimum vertex cover and find a phylogeny using the above procedure which only uses the base w extra vertices, we know we have found an optimal solution. If we use each minimum vertex cover as a starting point, and all of the returned phylogenies have more than w extra vertices, then we take the phylogeny with the smallest number of extra vertices amongst them, say $w + i$, as our potential optimal solution. If $i = 1$, the solution is optimal since any non-minimal vertex cover that we start with will have at least $w + 1$ extra vertices. If $i > 1$, then we must try all other vertex covers with size less than $w + i$ as starting sets, as one of these could potentially produce a smaller solution.

Consider the conflict graph in Figure 4.2. It has two different minimum vertex covers: $\{13, 18\}$ and $\{13, 20\}$. Taking the first cover and explicitly mapping the first two extra vertices to characters 13 and 18, our procedure constructs a phylogeny with three extra vertices, 13, 18, and 20. Starting with the second cover, our procedure again constructs a phylogeny which uses the same three extra vertices. Since any other vertex cover for this graph (such as $\{16, 18, 20\}$) has at least three vertices, we know our phylogenies with three

extra vertices are optimal.

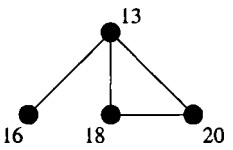


Figure 4.2: The conflict graph for the 21 species, 21 characters subset of 'fb65' data. This data and corresponding results will be discussed in Chapter 5.

Though we potentially have to construct many phylogenies to prove optimality with this method, each individual phylogeny construction tends to occur much quicker since we have reduced the number of possible mappings. If the conflict graph has a small number of vertices or edges, as is the case with most of our real world data, this method proves to be a very viable approach (as will be shown in the next chapter). However, in general, the problem of finding all vertex covers of a graph is very difficult. The problem of finding the smallest vertex cover, the vertex cover problem, alone, is NP-complete. Currently, we manually find all relevant vertex covers in the conflict graph, so this method is impractical for very large examples, or when automation is necessary.

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 Section 5.4.

Chapter 5

Experimental Results

5.1 Implementation

We present experimental results based on three sets of species data. Our first set, 'fb65', is haplotype information for guppy fish, *Poecilia reticulata*, containing 37 species and 468 characters with 5 states [35]. Specifically, the sequences control long-wave sensitive opsins, which control the visual sensitivity of the fish to different wave lengths of light. Of these 468 characters, 449 are binary. To apply our binary model, we delete the nineteen non-binary characters. This is scientifically justified by the observation that, as discussed in Section 2.1, molecular biologists often limit their consideration to binary characters (or SNPs). The 'gen' data set was used for taxonomic studies of the liverworts genus, *Pellia epiphylla*-complex [55]. The results support the distinction of recently discovered sibling species. It contains 8 species and 150 binary characters. The final set, 'pin', is from a taxonomic study of the *Saccharomyces* sensu stricto complex of yeast species [15], consisting of 20 species and 274 binary characters. It is based on all of the protein-encoding genes revealed by the complete genome sequence of the paradigmatic species, S. cerevisiae, and the results are used to give a detailed explanation of evolutionary events underlying the phylogeny.

Our ASP solver, in all reported results, was Cmodels-2 with SAT solver zchaff. Initial tests were performed using smodels and Cmodels, and Cmodels regularly performed much better than smodels on our data. Our model, along with the input, was grounded using Lparse [65]. This is always fast and is not included in the timing. Times shown are durations output by Cmodels. In the case of the general program, multiple runs of Cmodels-2 must be performed for each input: One run for each value of k, the number of extra vertices in the

required solution, must be carried out until a solution is found. The time given is the sum of durations for each run in the sequence of models. 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.

The general procedure for constructing phylogenies for a set of species is handled using a Perl script, 'findnpp.pl'. The program takes as input the species-character-state data of the given instance, as well as the solver to use, the model variation to use, and the number of solutions to return. It reads the input and determines the value of m. It then begins by adding no extra vertices to the model and uses Lparse and Cmodels to determine if a phylogeny exists. It stores the time taken to return (with or without an answer). If a phylogeny exists, it is output, along with the total number of vertices needed to construct the phylogeny and the total time taken. If no phylogeny exists for the input data with the specified number of vertices, an extra vertex is added to the model and the solver is run again. This script also handles the specific alterations to the model which are required by each variation. The complete code for 'findnpp.pl' is given in Appendix B.1.

5.2 General Preprocessing

We can reduce the number of characters we have to deal with for a particular problem instance if we use the technique of trimming identical characters. Identical characters form non-branching paths in phylogenies. If a set of characters is identical (i.e. each character takes the same state for each species), they will appear along a path with no branching points in any valid phylogeny. The order of the character changes does not matter, and any permutation of characters along the path will produce an equally valid phylogeny. Consider a small phylogeny example with three species and four characters, represented by the strings "1000", "0111", and "1111", as shown in Figure 5.1. Notice that characters 2, 3, and 4 are identical (they each have value 0 for species 1 and value 1 for species 2 and 3), and form a non-branching path in the phylogeny. This will be true of any phylogeny constructed with these identical characters. Hence, all but one copy of the identical character can be removed in preprocessing. Removed characters can easily be added back to resulting trees in post processing to form these non-branching paths.

By applying this procedure to our input data, we significantly reduce the size of the problem to be handled by the ASP solver, without changing the core of the problem. In

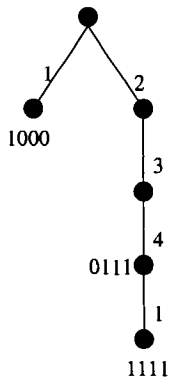


Figure 5.1: A phylogeny for the species "1000", "0111", and "1111". Characters 2, 3, and 4 are identical.

doing so, the 'fb65' data set is left with 65 characters, 'gen' is left with 26 characters, and 'pin' is left with 75.

5.3 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, and in 'fbpp2' the second character from each conflicting pair was removed. Both data sets are left with 38 characters. In 'fbppr', a random character from each pair was removed. This data set has 39 characters. For each, we solved subsets with 12, 18, 24, 30, 36, and 37 species (see Table 5.1). These results verified the correctness of the model and we found solutions to our full test data sets quickly, as hoped.

Sp	12	18	24	30	36	37
fbpp1	0.01	0.03	0.09	0.15	0.25	0.27
fbpp2	0.01	0.03	0.06	0.13	0.26	0.30
fbppr	0.01	0.04	0.07	0.14	0.28	0.29

Table 5.1: Time in seconds to construct phylogenies using the perfect phylogeny model. The row labeled "Sp" is the number of species in the data set. The next three rows give the running times for the solver on the three data sets.

5.4 General Model Results

To test the performance of our CCS model, and its variations, we again used the 'fb65' data. The 37-species 65-character data set is too large for any solver we tested, so we considered a number of subsets by varying both the number of characters and the number of species (in multiples of 3). 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, adding a vertex whenever the solver returns false. We do not use binary search since the most time consuming computation, in every tested instance, occurs when the number of extra vertices is one less than optimal (see, for example, Table 5.2). To prove a solution is optimal, we must show that no trees can be constructed with one less than the optimal number of vertices, so this time consuming step must always be performed. As was also observed during testing, adding more than the optimal number of extra vertices causes an increase in the running time of the solver, compared to the running time when the optimal number is used. The search space of the problem grows as the number of extra vertices increases, since each extra vertex can be mapped freely to any character. Hence the solver is usually very fast when the number of extra vertices is very small. As a particular example, our algorithm requires just 28 seconds to reach three extra vertices when proceeding linearly for the '24s,24c' data set, which has optimal solutions at four extra vertices. Solving even one instance with more than four vertices would be less efficient.

#EV	0	1	2	3	4	5	6
'24s,24c'	0	1	27	11931	**4845**	8930	9011
'27s,24c'	0	1	6	1344	**639**	636	1187

Table 5.2: Running times (in seconds) as the number of extra vertices (#EV) is increased, for the basic model on the '24s,24c' and '27s,24c' data sets. The optimal solution in both cases has 4 extra vertices.

Table 5.3 summarizes the results of our attempts to construct phylogenies on subsets of the 'fb65' data. To determine the frontier of the size of the problems we could solve in reasonable amounts of time, we took each value of n (the number of species) and, starting from 3 characters, ran the solver with our basic model on data sets with incrementally more

characters until we did not obtain a solution within a two hour cutoff period. The table 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 then used each of our variant models to construct phylogenies for these frontier data sets.

Data	#EV	Basic	A	D	E	A+	MC
3s,63c	0	0	0	0	0	0	0
6s,51c	0	3159	2503	3708	1701	2398	2503
9s,39c	3	3191	4099	4856	4850	1996	1500
12s,39c	3	670	579	881	496	863	131
15s,39c	3	1107	755	1088	797	4093	73
18s,39c	3	1077	947	1284	1050	3787	191
21s,21c	3	23	5	33	5	4	10
24s,24c	4	16803	10784	14620	8634	1916	440
27s,24c	4	1990	1180	1581	913	239	6
30s,18c	3	4	3	4	3	3	2
33s,18c	3	5	4	5	4	3	2
36s,18c	3	6	5	6	5	4	4

Table 5.3: Model comparsion 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 along our frontier to that of PENNY, the program which computes Camin-Sokal phylogenies from the commonly used PHYLIP package. Table 5.4 gives run times for the two methods on three separate 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 it takes PENNY to halt with optimal solutions, 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.

The best performance we have obtained 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

Data	#EV	A+	PENNY
3s,63c	0	0	0
6s,51c	0	2398	0
9s,39c	3	1996	0
12s,39c	3	863	1
15s,39c	3	4093	8
18s,39c	3	3787	5700
21s,21c	3	4	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)

Data	Ex. V	Var G	PENNY
3s,21c	0	0	0
4s,21c	0	0	0
5s,15c	1	70	0
6s,12c	1	2	0
7s,9c	3	8	0
8s,9c	3	6	0

(b)

Data	#EV	A+	PENNY
3s,75c	0	0	0
6s,57c	1	3	0
9s,30c	4	317	0
12s,21c	4	10	46
15s,10c	2	0	1350
18s,9c	3	8	X
20s,9c	3	7	X

(c)

Table 5.4: Comparison of performance of Model A+ and PENNY on three data sets. Running times are given in seconds. An 'X' in the table represents failure to return a solution within the two hour cutoff period: (a) fb65 - 37 species, 65 characters; (b) gen - 8 species, 26 characters (c) pin - 20 species, 75 characters.

approach drastically reduced running times, enabling us to solve larger problems within reasonable amounts of time. Figure 5.2 shows the frontier of largest subsets of our 'fb65' data solvable within two hours, for PENNY and Models A+ and MC.

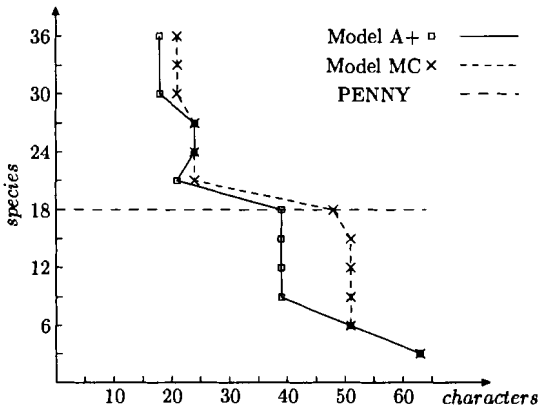


Figure 5.2: Frontier for PENNY and Models A+, MC.

5.5 Discussion

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 on experimentally obtained biological data, using the Cmodels-2 ASP solver.

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, one of most widely used branch-and-bound programs for this problem.

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. 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, which 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 vertices and species, 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 since 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 (in terms of the corresponding answer sets) tight program. Our change involved ordering the vertices. However, off-setting any benefit of tightness is the fact that the solver must attempt to find the proper mapping from vertices to characters. 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 in general, 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 often used in ASP.

The best performance we obtained 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?

Chapter 6

Related Work and Conclusions

6.1 Related Work

Recent biological uses of binary cladistic Camin-Sokal (CCS) include finding phylogenetic trees of *Saccharomyces sensu stricto* complex of yeast [15]. 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 [55]. In both cases, the CCS method which was used was a simple, general parsimony method for binary character data. The CCS model is also commonly used when the irreversibility constraint is essential. Nozaki *et al.* utilized the irreversibility in the CCS model because they observed that regaining of plastid genes is generally impossible during evolution [54].

As mentioned in Section 2.6, branch-and-bound (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. [56, 67, 51]. 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.

ASP has been used for phylogenetic problems before, although not for the maximum parsimony problem. In [6], they construct "maximum compatibility" phylogenies (see Section 2.2) with answer set programs. ASP solvers have also been used to construct "perfect phylogenetic networks", from phylogenetic trees, to explain the evolution of Indo-European languages [17]. These networks extend given phylogenetic trees with extra edges to account

for characteristics which are shared or borrowed rather than inherited from ancestors. Another approach to phylogeny construction which has appeared recently is the Maximum Quartet Consistency (MQC) problem. In this version of the problem, local phylogenies are built for every subset of four species. The goal is then to build a complete phylogeny which satisfies a maximum number of these predicted quartets. In [24], an equivalent representation of the MQC problem is given and is formulated in ASP. Computational experiments are conducted to confirm the efficiency of this approach.

As answer set programming is still a very young paradigm in computer science, and one filled with much promise, we can expect to see more robust, advanced, and faster solvers in the years to come. Cmodels has recently been upgraded to version 3.0, and new solvers appear regularly, including ASSAT [45], another solver which uses underlying SAT solvers, and aspps [13], a solver based on propositional schemata. Since some ASP solvers are SAT-based, new advances in SAT solving methods would potentially give improved performance to ASP solvers for free. Since 2002, yearly competitions have been held to promote the development of new state-of-the-art SAT solvers [62, 41]. From these competitions, promising new SAT solvers have been developed, including siege [58] and Zchaff2004 [48].

6.2 Conclusions

The main motivation of this work is to find optimal solutions for the maximum parsimony problem under commonly used metrics. In this thesis, we have presented an ASP-based approach to phylogeny inference using the binary cladistic Camin-Sokal metric. This is a logical first step towards phylogeny construction using the less restrictive Wagner metric. By using the expressive ASP language, we were able to present concise declarative models which clearly specify the requirements and constraints on desired solutions. The variations presented are motivated either by common procedures used to improve the performance of logic programs or through new ideas. Experimental results provide an interesting comparison of the benefits of each method considered. By testing our best models against the benchmark PENNY, a branch-and-bound technique from the commonly used PHYLIP package, we give evidence that our ASP-based approach can be a viable option in the phylogeny construction field. We feel that newer, faster ASP solvers and grounders will emerge, due to the high interest level in this new research area, and this will only help to improve our methods and strengthen our argument.

This thesis provides the basis for several logical next steps. Extensions to this work include exploring more ways to improve the performance of our binary CCS models. These include further attempts at rephrasing the constraints and adding redundant constraints, as well as incorporating more recent, faster SAT solvers into Cmodels. Hybrid techniques which combine the standard BNB and declarative approaches could also be explored. 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.

As outlined in Chapter 2, the CCS problem is a specific instance of the more general maximum parsimony problem, and modeling solutions for it has given us just a taste of the larger task at hand. Continuing to remove restrictions, such as the irreversibility property, until we reach the more general Wagner version of the problem, would make this technique applicable to a wider range of applications.

In the future, we also wish to explore other possible declarative approaches to the maximum parsimony problem. Aside from ASP, another similar declarative language, called *Inductive Definition Logic* (ID-Logic) [12], has recently generated a lot of research interest. ID-Logic extends first-order logic with inductive definitions, and early solvers have already been developed. As ID-Logic becomes a more viable technique, it would be an interesting project to attempt these same phylogeny problems with it and compare its results with those of ASP and the common procedural approaches.

Appendix A

Answer Set Programs

A.1 Program for perfect binary CCS problem

```
% phylogeny-a.lp
%
% Include species data from a separate file
% Input must only include binary states, 0 and 1.

%------------------------------------------------------------
% Handling the input

    c(C) :- a(P,C,1).

% c represents the set of all non-trivial characters. This set
%   is also the set of vertices, one for each character

%------------------------------------------------------------
% Enforce forest structure:

% For every vertex (char) V, there can be at most 1 incoming
%   edge and it should be from a vertex different than V.

    { edge(V1,V): c(V1): V1 != V } 1 :- c(V).
```

```
%-------------------------------------------------------------
% The "above" idea

% The edge-above relationship: C is above C1 if there is an
%   edge from C to C1

    above(C,C1) :- c(C), c(C1), C != C1, edge(C,C1).

% Transitivity of above
%
    above(C,C2) :- c(C), c(C1), c(C2), C != C1, C != C2,
        C1 != C2, above(C,C1), edge(C1,C2).

% For each species, each pair of characters with value 1 must
%   be comparable to each other.
%
    comparable(C,C1) :- c(C), c(C1), C != C1, above(C,C1).
    comparable(C,C1) :- c(C), c(C1), C != C1, above(C1,C).

    :- a(P,C,1), a(P,C1,1), C != C1, not comparable(C,C1).

% For a particular species, we can't have a character with
%   value 0 above a character with value 1
%
    :- a(P,C,0), a(P,C1,1), above(C,C1).

% To remove cycles
%
    :- c(C), c(C1), above(C,C1), above(C1,C).

#hide.
#show edge(V1,V2).
```

A.2 Program for general binary CCS problem

```
% phylogeny-np.lp
%
% Include species data from a separate file
% Input must only include binary states, 0 and 1.

%------------------------------------------------------------
% Handling the input

    c(C) :- a(P,C,1).
    sp(P) :- a(P,C,1).

    v(C) :- c(C).

    % Lines for extra vertices go here (using findnpp.pl).

    % mapping from vertices to characters
    m(C,C) :- c(C).

    % mapping from species to vertices
    1 { p(P,V) : v(V) } 1 :- sp(P).

% c represents the set of all non-trivial characters. m is the
%  mapping from vertex number to character number. v is the
%  set of vertices.

%------------------------------------------------------------
% Enforce forest structure:

% For every vertex V, there can be at most 1 incoming edge
%   and it should be from a vertex different than V.
```

```
    { edge(V1,V): v(V1): V1 != V } 1 :- v(V).

%-------------------------------------------------------------
% The "above" idea
%

% The edge-above relationship: Vertex V is above V1 if there is
%   an edge from V to V1.
%
    above(V,V1) :- v(V), v(V1), V != V1, edge(V,V1).

% Transitivity of above
%
    above(V,V2) :- v(V), v(V1), v(V2), V != V1, V != V2,
        V1 != V2, above(V,V1), edge(V1,V2).

% Make above reflexive
%
    above(V,V) :- v(V).

%-------------------------------------------------------------
% Checking the species mapping
%

% For each species, each character with value 1 must have a
%  vertex which is above the species' vertex which maps to
%  that character.
%
    g(P,C) :- sp(P), c(C), v(V1), v(V), p(P,V),
        m(V1,C), above(V1,V).

    :- a(P,C,1), not g(P,C).
```

```
    :- a(P,C,0), g(P,C).

%---------------
% Remove cycles
%
    :- v(V), v(V1), V != V1, above(V,V1), above(V1,V).

#hide.
#show edge(V1,V2).
#show m(V,C).
#show p(P,V).
```

Appendix B

Helper scripts for Binary CCS Model

B.1 'findnpp.pl'

This script is written in the Perl programming language.

```
#!/usr/bin/perl

# findnpp.pl
#
# written by: Jonathan Kavanagh
#
# To be used, in conjunction with phylogeny-np.lp or one of its
#  variants, for finding optimal phylogenies for CCS problem.
#
# Input is given with 3 arguments: data_file, the instance to
#  be solved; solver, which solver to use; var, which variation
#  of the model to use; number, the number of solutions to
#  return (0 for all solutions).
#
# Command line: findnpp.pl data_file solver var number
```

```
if ($#ARGV < 3)
{
    print("Proper usage: findnpp.pl data_file solver" .
        "var number\n");
    exit(0);
}
else
{
    $input = $ARGV[0];
    $solver = $ARGV[1];
    $var = $ARGV[2];
    $num = $ARGV[3];
}

if ($solver =~ m/^cmodels$/)    # Adjust solver for cmodels
{
    $solver = "cmodels-2/cmodels -zc";
}
if ($num == 0)
{
    $solver = "$solver 0";
}

$model = "phylogeny-np-$var.lp";

print "Using model: $model\n";

$initfound = 0;
$m = 0;

open(INFILE, "<$input");

while (<INFILE>)
```

```
{               # Parse input to get num of chars
    chomp;

    if (/a\(([a-z0-9]+), ?([a-z0-9]+), ?([a-z0-9]+)\)\./)
    {
        if (!$initfound) # We assign first species to $sp1
        {
            $initfound = 1;
            $sp1 = $1;
        }

        if ($1 == $sp1)
        {
            if ($2 > $m)    # Obtaining $m, the num of chars
            {
                $m = $2;
            }
        }
    }
}
close(INFILE);

$solfound = 0;
$totaldur = 0.0;
$k = $m;

while (!$solfound)
{           # Increment $k until solution is found
    open(PHYLFILE, "<$model");
    open(TEMPFILE, ">temp.lp");

    while(<PHYLFILE>)
    {
```

```
chomp;

print TEMPFILE $_, "\n";

if (/Lines for extra vertices go here/)
{       # Add $k - $m extra vertices
    for ($i = $m + 1; $i <= $k; $i++)
    {
        print TEMPFILE "    v($i).\n";
    }

    if ($var =~ m/^c$/)
    {
        # For this variation, each vertex gets mapped
        #  to any character and this line is found in
        #  the phylogeny-np-c.lp file
    }
    elsif ($k > $m)
    {
        print TEMPFILE "    1 { m(V,C) : c(C) } 1 " .
                           ":- v(V), V > $m.\n";
    }
}

if (/Lines for variation e go here/)
{
    print TEMPFILE "    l(V1,V) :- v(V1), v(V), " .
                       "V1 <= $m.\n";
    print TEMPFILE "    l(V1,V) :- v(V1), v(V), " .
                       "V1 > $m, V1 < V.\n";
    print TEMPFILE "    { edge(V1,V): v(V1): V1 != V }"
                       . " 1 :- v(V), V <= $m.\n";
    print TEMPFILE "    { edge(V1,V): v(V1): l(V1,V) }"
```

```
                        . " 1 :- v(V), V > $m.\n";
    }

    if (/Lines for variation a+ go here/ && $k > $m)
    {
        print TEMPFILE "    :- v(V), v(V1), V > $m, " .
                       "V1 <= $m, c(C), ";
        print TEMPFILE "m(V,C), m(V1,C), above(V,V1).\n";
        print TEMPFILE "    :- v(V), v(V1), V <= $m, " .
                       "V1 > $m, c(C), ";
        print TEMPFILE "m(V,C), m(V1,C), above(V,V1).\n";
        print TEMPFILE "    :- v(V), v(V1), V > $m, " .
                       "V1 > $m, V != V1, c(C), ";
        print TEMPFILE "m(V,C), m(V1,C), above(V,V1).\n";
    }
}

close(PHYLFILE);
close(TEMPFILE);

print "For k = $k:\n";
$res = `lparse temp.lp $input | $solver`;

if ($res =~ m/Duration: (\d+\.\d+)/)
{
    $dur = $1;
    $totaldur += $dur;
}

if ($res =~ m/Answer/)
{
    print $res;
    print "Total Duration: $totaldur\n";
```

```
        $solfound = 1;
    }
    else
    {
        print "False.\n";
        print "Duration: $dur\n";
        $k++;
    }
}

print "The end.\n";
```

Bibliography

[1] R. Agarwala and D. Fernandez-Baca. A polynomial-time algorithm for the perfect phylogeny problem when the number of character states is fixed. *SIAM Journal on Computing*, pages 1216–1224, 1994.

[2] S. Agrawal and F. Khan. Reconstructing recent human phylogenies with forensic STR loci: A statistical approach. *BMC Genetics*, 6(47), 2005.

[3] J. Aldrich. R.A.Fisher and the making of maximum likelihood 1912-1922. *Statistical Science*, 12(3):162–176, 1997.

[4] C. Anger, K. Konczak, T. Linke, and T. Schaub. A glimpse of answer set programming. *Künstliche Intelligenz*, 19(1):12–17.

[5] H. Bodlaender, M. Fellows, and T. Warnow. Two strikes against perfect phylogeny. In *Proceedings of the 19th International Colloquium on Automata, Languages, and Programming*, pages 273–283. Springer Verlag, 1992.

[6] D.R. Brooks, E. Erdem, J.W. Minett, and D. Rings. Character-based cladistics and answer set programming. *PADL*, pages 37–51, 2005.

[7] J.H. Camin and R.R. Sokal. A method for deducing branching sequences in phylogeny. *Evolution*, 19(3):311–326, 1965.

[8] K. Clark. *Negation as Failure*. Plenum Press, 1978.

[9] S.A. Cook. The complexity of theorm-proving procedures. In *Proceedings of the third annual ACM Symposium on Theory of computing*, pages 151–158, 1971.

[10] 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.

[11] W.H.E. Day and D. Sankoff. Computational complexity of inferring phylogenies by compatibility. *Systematic Zoology*, 35(2):224–229, 1986.

[12] M. Denecker. Extending classical logic with inductive definitions. In *Proceedings of the 1st International Conference on Computational Logic*, pages 703–717. Springer, 2000.

[13] D. East and Truszczyński. Asp solver aspps, 2001. http://www.cs.uky.edu/aspps/.

[14] R.V. Eck and M.O. Dayhoff. *Atlas of protein sequence and structure.* National Biomedical Research Foundation, 1966.

[15] 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.

[16] E. Erdem and V. Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3:499–518, 2003.

[17] 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.

[18] F. Fages. Consistency of clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

[19] J. Felsenstein. Phylip home page, 1980. http://evolution.genetics.washington.edu/phylip.

[20] J. Felsenstein. Evolutionary trees from DNA sequences: A maximum likelihood approach. *Journal of Molecular Evolution*, 17(6):368–376, 1981.

[21] J. Felsenstein. Penny - branch and bound to find all most parsimonious trees, 1986. http://evolution.genetics.washington.edu/phylip/doc/penny.html.

[22] P. Forster and A. Toth. Toward a phylogenetic chronology of ancient gaulish, celtic, and indo-european. *Proceedings of the National Academy of Sciences*, 100(15):9079–9084, 2003.

[23] L.R. Foulds and R.L. Graham. The steiner tree problem in phylogeny is NP-complete. *Advanced Applied Mathematics*, 3:43–49, 1982.

[24] W. Gang, L. Guohui, and Y. Jia-Huai. Quartet based phylogeny reconstruction with answer set programming. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence*, pages 612–619, 2004.

[25] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete problems. In *Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 47–63, 1974.

[26] B.S. Gaut and P.O. Lewis. Success of maximum likelihood phylogeny inference in the four-taxon case. *Molecular Biology and Evolution*, 12:152–162, 1995.

[27] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. *Proc., Int'l Logic Programming Conference and Symposium*, pages 1070–1080, 1988.

[28] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proceedings of the 7th International Conference on Logic Programming*, pages 579–597. The MIT Press, 1990.

[29] Genome Management Information System (GMIS). SNP fact sheet, 2006. http://www.ornl.gov/sci/techresources/Human_Genome/faq/snps.shtml.

[30] D. Gusfield. Efficient algorithms for inferring evolutionary history. *Networks*, 21:19–28, 1991.

[31] D. Gusfield. *Algorithms on Strings, Trees, and Sequencesr; Computer Science and Computational Biology.* Cambridge University Press, 1997.

[32] D. Gusfield. Haplotyping as perfect phylogeny: conceptual framework and efficient solutions. In *RECOMB '02: Proceedings of the 6th annual international conference on Computational biology*, pages 166–175, 2002.

[33] J. Harland. A clausal form for the completion of logic programs. In *Proceedings of the International Conference on Logic Programming*, number 8, pages 711–725, 1991.

[34] M. Held and R.M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970.

[35] M. Hellman, N. Tripathi, S.R. Henz, A. Lindholm, D. Weigel, F. Breden, and C. Dreyer. Unpublished data. 2006.

[36] L. Helmuth. Genome research: Map of the human genome 3.0. *Science*, 293(5530):583–585, 2001.

[37] M.D. Hendy and D. Penny. Branch and bound algorithms to determine minimal evolutionary trees. *Mathematical Biosciences*, 59(2):277–290, 1982.

[38] W. Hennig. *Grundzuege einer Theorie der Phylogenetischen Systematik.* Deutscher Zentralverlag, 1950.

[39] J.P. Huelsenbeck and K.A. Crandall. Phylogeny estimation and hypothesis testing using maximum likelihood. *Annual Review of Ecology and Systematics*, 28:437–466, 1997.

[40] A.H. Land and A.G. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

[41] D. Le Berre and L. Simon. Fifty-five solvers in vancouver: The SAT 2004 competition. *Lecture Notes in Computer Science*, 3542:321–344, 2005.

[42] J. Lee and V. Lifschitz. *Loop Formulas for Disjunctive Logic Programs*, volume 2916, pages 451–465. 2003.

[43] N. Leone, W. Pfeifer, W. Faber, F. Calimeri, T. Dell'Armi, T. Eiter, G. Gottlob, G. Ianni, G. Ielpa, C. Koch, S. Perri, and A. Polleres. The dlv system. In *Proceedings of the 8th European Conference on Artificial Intelligence*, pages 537–540. Springer, 2002.

[44] Y. Lierler and M. 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.

[45] F. Lin and Y. Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[46] J. Lloyd and R. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 3:225–240, 1984.

[47] D.R. Maddison and W.P. Maddison. Macclade 4.08, 2005. http://macclade.org/macclade.html.

[48] Y.S. Mahajan, F. Zhaohui, and S. Malik. Zchaff2004: An efficient sat solver. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT2004)*, pages 360–375, 2004.

[49] V.W. Marek and M. Truszczynski. *Stable logic programming - an alternative logic programming paradigm.* Springer-Verlag, 1999.

[50] F.R. McMorris, T. Warnow, and T. Wimer. Triangulating vertex colored graphs. In *Proceedings of the 4th Annual Symposium on Discrete Algorithms*, 1993.

[51] B.M.E. Moret, J. Tang, L.S. Wang, and T. Warnow. Steps toward accurate reconstruction of phylogenies from gene-order data. *Journal of Compututational Systems Sciemces*, 65(3):508–525, 2002.

[52] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.

[53] I. Niemelä, P. Simons, and T. Syrjänen. Smodels: A system for answer set programming. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*, 2000.

[54] 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.

[55] A. Pacak, P. Fiedorow, J. Dabert, and Z. Szweykowska-Kulińska. RAPD technique for taxonomic studies of pellia epiphylla-complex (hepaticae, metzgeriales). *Genetica*, 104:179–187, 1998.

[56] W. Purdom, Jr., P.G. Bradford, K. Tamura, and S. Kumar. Single column discrepancy and dynamic max-mini optimization for quickly finding the most parsimonious evolutionary trees. *Bioinformatics*, 2(16):140–151, 2000.

[57] D. Roderic, M. Page, and E.C. Holmes. *Molecular Evolution: A Phylogenetic Approach.* Blackwell Science, Oxford, UK, 1998.

[58] L. Ryan. Siege satisfiability solver, 2004. http://www.cs.sfu.ca/ loryan/personal/.

[59] N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406–425, 1987.

[60] W. Shih. A branch and bound method for the multiconstraint zero-one knapsack problem. *The Journal of the Operational Research Society*, 30:369–378, 1979.

[61] J. Silvertown, M. Franco, and J.L. Harper. *Plant Life Histories: Ecology, Phylogeny and Evolution.* Cambridge University Press, 1997.

[62] L. Simon, D. Le Berre, and E.A. Hirsch. The SAT2002 competition report. *Annals of Mathematics and Artificial Intelligence*, 43(1-4):307–342, 2005.

[63] P. H. A. Sneath and R.R. Sokal. *Numerical taxonomy The principles and practice of numerical classification.* W.H. Freeman, 1973.

[64] D.L. Swofford. Paup* 4.0, 2001. Phylogenetic Analysis Using Parsimony (*and Other Methods).

[65] T. Syrjänen. Lparse user's manual, 1998. http://www.tcs.hut.fi/Software/smodels/.

[66] M. Trick. Formulations and reformulations in integer programming. *Lecture Notes in Computer Science*, 3524:366–379, 2005.

[67] M. Yan and D.A. Bader. Fast character optimization in parsimony phylogeny reconstruction. Technical report, 2003.