SOLVING NP SEARCH PROBLEMS WITH MODEL EXPANSION

by

Faraz Hach
B.Sc., Sharif University of Technology, 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
Fall 2007

APPROVAL

Name:	Faraz Hach
Degree:	Master of Science
Title of thesis:	SOLVING NP SEARCH PROBLEMS WITH MODEL EXPANSION
Examining Committee:	Dr. Eugenia Ternovska Chair
	Dr. David Mitchell, Assistant Professor, Computing Science Simon Fraser University Senior Supervisor
	Dr. Uwe Glasser, Associate Professor, Computing Science Simon Fraser University Supervisor
	Dr. Arvind Gupta, Professor, Computing Science Simon Fraser University SFU Examiner
Date Approved:	Dec. 7, 2007

Abstract

We explore the application of MXG, a declarative programming solver for NP search problems based on Model Expansion (MX) for first order logic with inductive definitions. We present specifications for several common NP-complete benchmark problems in the language of MXG, and describe some modeling techniques we found useful in obtaining good solver performance. We present an experimental comparison of the performance of MXG with Answer Set Programming (ASP) solvers on these problems, showing that MXG is competitive and often better. As an extended example, we consider an NP-complete phylogenetic inference problem. We present several specifications for this problem, employing a variety of techniques for obtaining good performance. Our best solution, which combines instance pre-processing, redundant axioms, and symmetry breaking axioms, performs orders of magnitude faster than previously reported declarative programming solutions using ASP solvers.

To my mom and dad.

Acknowledgments

I would like to thank Dr. David Mitchell, my senior supervisor, for his guidance and support throughout my Masters studies. I thank Dr. Uwe Glasser and Dr. Arvind Gupta for their comments on the final document. I am grateful to Eugenia Ternovska, Jan Manuch, and Sharon (Xiao-hong) Zhong for helpful discussions. My thanks to Jonathan Kavanagh for providing his ASP programs. Last but not least I would like to thank Raheleh Mohebali, my beloved partner, and my parents for their endless support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Informally, declarative programming involves stating what is to be computed, but not necessarily how it is to be computed. From a programmers point of view, programming is lifted to a higher level of abstraction. A declarative programming language for search problems provides a syntax to specify the relationship between instances of the problem and solutions. A "solver" for the language takes such a specification, together with an instance, and produces a solution for the instance if there is one. The use of such tools can greatly reduce the effort required to obtain effective practical solutions to a wide variety of problems, which otherwise would require a significant investment in development of problem-specific algorithms and implementations.

For many declarative programming approaches, the language is a theory in some logic, and the choice of logic determines the expressive power of the language. For example, "Answer Set Programming" (ASP) [35, 43] is based on the language of logic programs under the stable model semantics [25]. The language of the ASP solver DLV [32] can describe any problem in the complexity classes $\sum_2^p$ and $\prod_2^p$ [19].

Fagin's theorem [21] states that the classes of finite structures definable in existential second order logic ($\exists$SO), are exactly those in the complexity class NP. The theorem suggests a natural declarative problem solving approach for NP-complete problems: Represent a problem with an $\exists$SO formula ϕ, and solve instances by reduction to SAT or some other fixed NP-complete problem. In the case of search problems, we must find interpretations of the existentially quantified second order variables, which provide a solution. So, the task becomes that of expanding a given structure to give a suitable interpretations for those relation symbols. For a specific logic $\mathcal{L}$, the task is called *$\mathcal{L}$-Model Expansion*, abbreviated

as $\mathcal{L}$-MX.

In [40], Mitchell and Ternovska proposed a new declarative programming framework, the "Model Expansion (MX) Framework", as a formal basis for tools for solving search problems based on the task of model expansion. They also proposed using First Order logic (FO) extended with inductive definitions [15, 13, 14], FO(ID), for solving NP search problems. The framework was further developed in [41]. MXG [42] is an MX-based solver for solving NP search problems. The language of MXG is based on FO(ID) enriched with Cardinality Constraints, FO(ID+Card). The goal of MXG is to establish whether solver technology based on FO(ID+Card) Model Expansion and grounding can be practically effective.

We explore the question of whether MXG is effective, in terms of performance and convenience, as a solver for NP-complete problems. In our effort to demonstrate that MXG can be an effective solver technology, our emphasis will be on this observation: It is a general property of declarative problem solving approaches that particular choices of problem representation can greatly affect solver performance. While this might not be entirely desirable, it can also be used to advantage to obtain good performance in solving particular problems.

In particular, we partially answer the following questions regarding the use of MX-based tools, and MXG in particular, in representing and solving NP-hard search problems:

- The language of MXG has a number of features, whose use could affect performance, including: sorts, order, bounded quantifiers, inductive definitions and cardinality constraints. Can these be used to positively affect performance?

- A number of techniques are used in many other related approaches to obtain performance benefits. These included adding redundant constraints [52] and adding symmetry-breaking constraints [46, 4, 5]. Can these techniques be naturally and effectively applied in MXG solutions?

- Often poly-time preprocessing of instances of NP-hard problems can be used to improve performance of general-purpose solving technology. Is this the case with MXG, and, more interestingly, could we carry out this kind of pre-processing declaratively using features of MXG?

- Often, general-purpose approaches are regarded as being convenient but never as effective overall as special-purpose tools in solving particular problems. Can we contribute

to a specific application domain by applying the knowledge we aquire about writing effective axiomatizations?

We present MXG specifications of a number of benchmark problems. In refining these specifications, we have explored the use of the features and techniques mentioned above. We observe a number of cases where we can use the features and techniques to obtain performance benefits. We compare the performance of:

- MXG, using MXC [6], a high performance SAT solver capable of handling CNF formulas extended with cardinality constraints, as its ground solver (Denoted MXG+MXC). A ground solver finds a satisfying assignment for a set of ground clauses, that are propositional clauses in our case;
- Three high-quality ASP solvers (clasp [26], smodels [49] and DLV). clasp and DLV were the top-performing solvers in the "2006 ASP Solver Competition" [1];
- MidL [37] another FO(ID)-MX solver developed independently and concurrently with MXG.

Our performance results show MXG to be competitive with these systems, sometimes being less effective but often being more effective.

As an extended example, we tackle a challenging problem and set of instances in phylogenetic inference [38]. A phylogenetic tree is a directed graph representing the evolutionary relationships among a collection of species. Phylogenetic inference (or re-construction) is the task of constructing a phylogenetic tree (or other network) from species data. It has many applications in biology [17, 44] and elsewhere [20], producing a variety of particular computational problems. Phylogenetic inference is interesting to us because of the following observations.

- Most interesting variants are NP-hard optimization problems which can be axiomatized by MXG, and there are many data sets too hard to solve well in practice, which makes it interesting to show that our solver can be used as a practical tool;
- Many particular problems are variants of, or combinations of, a few basic problems, so having declarative solutions which can easily be combined or revised would be useful;

- The optimality metrics often do not precisely match subjective solution quality, so users could benefit from a method to interactively add ad-hoc constraints, which is possible with declarative solutions but not possible with current tools;
- There are special purpose tools for different variants of this problem, so we can compare performance of our axiomatization and tools with those state-of-art special purpose tools.

The particular problem we study is the binary Cladistic Camin-Sokal (CCS) problem, which we chose because it is a simple NP-hard problem; to which standard tools apply; for which we have suitably challenging real data; and for which there is a previous declarative programming solution with which to compare. Our test set consists of several hundred instances of graduated difficulty derived from biological data [29]. We compare the performance of:

- MXG+MXC, with various axiomatizations using cardinality constraints;
- MXG with Minisat [18], a high-performance SAT solver, with non-cardinality MX axiomatizations;
- clasp, a "native" ASP solver with clause learning, with the best-performing ASP axiomatization from [30];
- MXG+MXC, aided by polynomial-time instance pre-processing;
- PAUP [47], a widely-used phylogeny software package.

We describe a method that is faster, by many orders of magnitude, than the only declarative solution of which we are aware. In doing so, we demonstrate that MX-based tools can be effective on more realistic domains than has previously been shown.

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

This thesis makes the following contributions:

- We demonstrate that FO(ID+Card)-MX, and in particular the technology used to produce MXG, is feasible as the basis for practical solving of NP-hard search problems.

In particular, the overall performance of the MXG on the problems we have studied is competitive with best ASP solvers and MidL.

- In solving these problems, we answer the questions posed earlier about the performance and convenience of MXG, and provide a number of examples illustrating techniques for producing specifications with effective performance.
- We push the state-of-the-art in declarative problem solving for phylogenetic inference substantially, to the point where bio-informaticians are interested, and where producing tools useful to bio-informaticians is within reach.

An overview of the thesis is as follows. Chapter 2 outlines MXG, its mathematical background, the underlying logic of its language, and the syntax of its language. In Chapter 3, we present a few techniques for writing axiomatizations which we have found to be useful to improve solver performance. We then give a comparison of MXG+MXC with several ASP solvers and MidL on some benchmark problems. In Chapter 4, we study the binary Cladistic Camin-Sokal (CCS). We provide a number of axiomatizations for this problem with MXG and compare our performance with previous work with ASP tools and a state-of-art phylogeny software, PAUP. In Chapter 5, we present systems related to MXG, and the techniques used in other systems for improving performance. In Chapter 6, we present conclusions and future work.

Chapter 2

MXG: A First Order Model Expansion Solver

In [40], Mitchell and Ternovska proposed the Model Expansion (abbreviated as MX) framework, as a declarative programming framework in which search problems are cast as the logical task of model expansion. In this chapter we explain the mathematical background of the MX framework, and present MXG [42], a solver for the MX framework for NP search problems.

2.1 MXG Mathematical Background

A vocabulary σ is a set of relation and function symbols, each with an associated arity. Constant symbols are zero-ary function symbols. A structure $\mathcal{A}$ for vocabulary σ, (σ-structure) is a tuple containing a universe A, and a relation (function) for each relation (function) symbol of σ, denoted by $\mathcal{A} = (A; \sigma^{\mathcal{A}})$. A structure $\mathcal{A}$ is called finite if its universe A is a finite set. For example, if σ has constant symbol 0 and a binary relation symbol $<$, then one possible finite structure for σ is $\mathcal{A} = (A; 0^{\mathcal{A}}, <^{\mathcal{A}})$, with universe of discourse $A = \{0, \ldots, 100\}$ where $0^{\mathcal{A}} = 0$, $<^{\mathcal{A}}$ has its natural meaning of less than on elements of universe A.

For a formula ϕ, we write $vocab(\phi)$ for the collection of exactly those vocabulary symbols which occur in ϕ. If $\mathcal{A} = (A; \sigma^{\mathcal{A}})$ and $\mathcal{B} = (A; \sigma^{\mathcal{A}}, \varepsilon^{\mathcal{B}})$, then $\mathcal{B}$ is an *expansion* of $\mathcal{A}$ to vocabulary $\sigma \cup \varepsilon$.

Definition: For any logic $\mathcal{L}$, the $\mathcal{L}$ *Model Expansion* problem, $\mathcal{L}$-MX is:
Given a pair $(\phi, \mathcal{A})$ where

- ϕ is an $\mathcal{L}$ formula, and
- $\mathcal{A}$ is a finite σ-structure for vocabulary $\sigma \subset vocab(\phi)$.

Find a structure $\mathcal{B}$ such that

- $\mathcal{B}$ is an expansion of $\mathcal{A}$ to $vocab(\phi)$, and
- $\mathcal{B} \models \phi$.

We call $\mathcal{A}$ instance structure, σ the instance vocabulary, and $\varepsilon = vocab(\phi) \setminus \sigma$ the expansion vocabulary.

A natural way to model a search problem is to view ϕ as a problem description, with an instance as a given finite structure. The formula, which is fixed, specifies the relationship between an instance and its solutions. The solutions for an instance $\mathcal{A}$ are given by the interpretations of expansion vocabulary symbols for any expansion structure $\mathcal{B}$ which is constructed by expanding $\mathcal{A}$ to $vocab(\phi)$ such that it satisfies ϕ. This is the idea of the MX framework, which clearly separates problem specifications, which are formulas in some logic $\mathcal{L}$, from instances, which are finite structures. (In contrast, some other declarative programming approaches, such as ASP, do not make a formal distinction between the two.)

Example: We want to find cliques in a graph $G = (Vtx, Edge)$ where Vtx is the set of vertices and $Edge$ is the binary edge relation of graph G. ϕ is the following formula, of first order logic (FO), defining a clique in the graph:

$$\phi : \forall xy\,((Clique(x) \wedge Clique(y)) \supset (x = y \vee Edge(x, y)))$$

The instance vocabulary is $\sigma = \{Edge\}$, the expansion vocabulary is $\varepsilon = \{Clique\}$, and an instance structure is $\mathcal{A} = (Vtx,\ Edge^{\mathcal{A}})$. The interpretation of $Clique$ in any structure $\mathcal{B} = (Vtx,\ Edge^{\mathcal{A}}, Clique^{\mathcal{B}})$, which is an expansion of $\mathcal{A}$, and satisfies ϕ, represents a clique in graph G.

In the MX framework, different logics can be chosen for writing the problem specification formulas, which gives us different expressive power for describing problems. We denote MX for a logic $\mathcal{L}$, by $\mathcal{L}$-MX. In the task of model expansion, symbols of the expansion vocabulary behave as existentially quantified second order variables. Fagin's theorem [21] says that NP

consists exactly of those problems that can be axiomatized in existential second order logic ($\exists$SO). Thus first order (FO) model expansion, denoted by FO-MX, has the same power as ($\exists$SO) over finite structures, and can express any problem in NP.

2.1.1 The Logic FO(ID+Card)

The logic FO(ID+Card) is classical FO logic augmented by inductive definitions [15, 13, 14], cardinality constraints, sorts and orders. These new features added to FO do not increase expressiveness, in that FO(ID+CARD)-MX, like FO-MX, captures NP. However, these extensions provide some facilities to improve convenience of modeling, and basing the language of MXG on this logic makes it more convenient to use.

Sorts

The logic underlying the MXG language is multi-sorted. The universe is partitioned into a set of sorts and we specify for each variable the sort which it is to range over. Having sorts in modeling problems usually helps in better understanding of axiomatizations, and helps the solver by reducing the search space. A multi-sorted FO formula can be easily tranformed to an ordinary FO formula by using domain predicates.

Order:

All structures in MXG are considered to be ordered. Ordering on elements of each sort is specified by the instance structure. Ordering relations $<, >, =, \neq, \leq, \geq$, constants MIN, MAX and binary relation SUCC are built in to MXG. For each sort they are interpreted with their natural semantics regarding the ordering of elements of the sort. Constants MIN and MAX denoting the first and last element of each sort, and the binary relation SUCC represents the successor relation.

Cardinality Constraints

Cardinality constraints are not easy to express in FO-MX, and usually require introducing auxiliary relations for counting elements. MXG provides a notation for expressing simple cardinality properties more conveniently. A SAT+Card solver, a SAT solver extended to handle cardinality constraints, is then used as the ground solver.

Inductive Definitions

ID-logic [15, 13, 14] extends classical logic with inductive definitions. Both monotone and non-monotone induction are formalized in a natural way in ID-logic. The classical part of the logic has the usual classical semantics. The semantics of inductive definitions is based on the two-valued well-founded semantics of logic programs. FO has no feature for expressing recursive properties such as graph reachability. Recursive properties can be expressed in FO-MX, but axiomatizations of this sort are often not intuitive. MXG uses FO(ID), a fragment of ID-logic with FO as the classical logic, in which these properties can be conveniently be expressed with inductive definitions.

Reductions of inductive definitions to SAT are not trivial, and the question of how to obtain good performance in a ground solver with inductive definitions is not resolved (but see [36]). In the current version of MXG, two fragments of inductive definitions of FO(ID) are supported:

- *Horn-ID:* A definition is a Horn-ID if the defined predicate occurs only positively in body of each rule, and all predicate symbols in its body are instance predicates or are effectively instance predicates (as they have already been computed during grounding). The interpretation of a predicate defined by a Horn-ID is computed at grounding time. This is done by re-writing the rules as an FO implication, grounding to propositional Horn clauses, and computing the minimum model in polynomial time. This model is the well-founded model of the inductive definition. The defined predicate is then treated as an instance predicate for the remainder of the grounding of this specification.

  Example: To find the distance of vertices in a graph $G = (Vtx; Edge)$ from a particular vertex $Start \in Vtx$ we can use the following inductive definition:

$$\begin{array}{ll} \{ & Dist(a,b) \leftarrow a = Start \wedge b = 0 \\ & Dist(a,b) \leftarrow Dist(a',b') \wedge Edge(a',a) \wedge SUCC(b',b) \quad \} \end{array}$$

  $Dist(a,b)$ is true iff the distance of a from $Start$ is b. The definition is a Horn-ID. MXG rewrites the rules as the following FO formulas and produces its grounding and then finds its minimal model.

$$\forall ab : ((a = Start \wedge b = 0) \supset Dist(a,b))$$
$$\forall ab : ((\exists a'b' : (Dist(a',b') \wedge Edge(a',a) \wedge SUCC(b',b))) \supset Dist(a,b))$$

- *Comp-ID:* A definition is a Comp-ID iff it is defined over a well-founded order. If a definition is not a Horn-ID, MXG replaces it with its *Completion* [9]. The substitution is correct if the definition is a Comp-ID, but not in general.

  Example: To find the even and odd numbers of the set $N = \{0, 1, 2, \ldots 100\}$, one might use the following inductive definitions, where *Even* and *Odd* are expansion predicates which denote the even and odd numbers, respectively.

$$\begin{array}{ll} \{ & Even(n) \leftarrow n = 0 \\ & Even(n) \leftarrow \neg Odd(n) \\ & Even(n) \leftarrow Odd(n') \wedge SUCC(n', n) \quad \} \\ \{ & Odd(n) \leftarrow \neg Even(n) \\ & Odd(n) \leftarrow Even(n') \wedge SUCC(n', n) \quad \} \end{array}$$

  As these definitions are not Horn-IDs, MXG replaces each with its completion, which is the following classical formula:

$$\forall n : (Even(n) \Leftrightarrow (n = MIN \vee \neg Odd(n) \vee [\exists n' : (Odd(n') \wedge SUCC(n', n))]))$$
$$\forall n : (Odd(n) \Leftrightarrow (\neg Even(n) \vee [\exists n' : (Even(n') \wedge SUCC(n', n))]))$$

  The model for these FO formulas is the well-founded model of the inductive definitions.

2.2 MXG

Cook's theorem [10] states that every problem in the complexity class NP can be reduced in polynomial time to SAT, which suggests a general solving scheme: reduce NP problems to SAT, and run the best SAT solver available on the SAT problem corresponding to each given instance.

MXG [42] is a solver for FO(ID+Card)-MX, for modeling and solving any NP search problem. It takes as input a problem specification file, and an instance description file, and reduces the problem, for the given instance, to a set of propositional and cardinality constraint clauses (denoting by SAT+Card). The reduction method is a polynomial time grounding (instantiation) procedure, called "Gnd-Hidden" [42]. It then runs a SAT+Card solver on the SAT+Card problem. The general scheme of MXG is illustrated in Figure 2.2

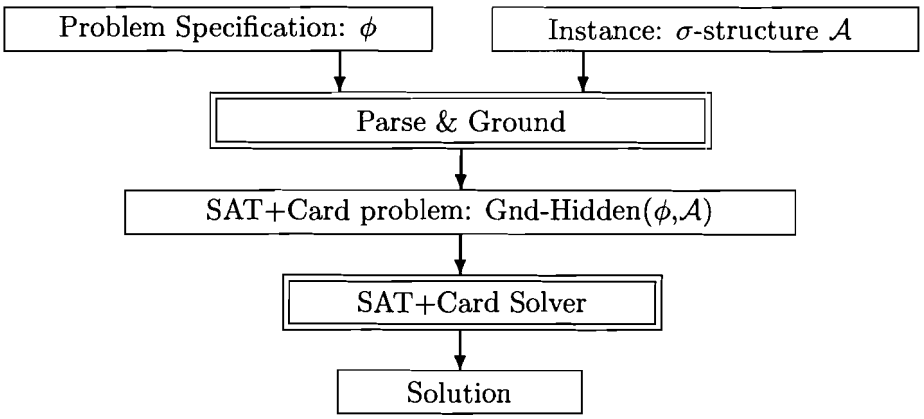

Figure 2.1: MXG High-level architecture

2.2.1 MXG Grounding Method

The grounding procedure of MXG, "Gnd-Hidden", is sound and complete, which means that satisfying assignments for the SAT+Card problem are in one-to-one correspondence with solutions of the original FO(ID+Card) model expansion problem. It is based on applying relational algebra operations to "extended-hidden relations"[42, 45]. We do not intend to explain the "Gnd-Hidden" algorithm here. We just give some basic ideas that we are need in Chapter 3.

For ϕ, a formula over vocabulary $\sigma \cup \varepsilon$, a reduced grounding of ϕ with respect to finite σ-structure $\mathcal{A}$ is a formula ψ over ε only, such that, for any structure $\mathcal{B} = (\mathcal{A}; \sigma^{\mathcal{A}}; \epsilon^{\mathcal{B}})$, $\mathcal{B} \models \phi$ iff $\mathcal{B} \models \psi$. A reduced grounding exactly defines the set of solutions for the instance structure $\mathcal{A}$. One may be obtained by producing a grounding and then "evaluating out" the instance vocabulary. MXG performs the grounding and evaluating out simultaneously.

The MXG algorithm[42] is based on "extended relations" and a generalization of the relational algebra. An extended relation $T_{\overline{x}}$ is a table with attributes $\overline{x}$, and a reduced ground formula associated to each entry. Tuples may be represented by pairs $(\overline{a}, \psi_T(\overline{a}))$, where $\psi_T(\overline{a})$ is the ground formula associated with the tuple $\overline{a}$. If $\overline{a}$ does not appear in the table, this is equivalent to $(\overline{a}, false)$ appearing in the table.

The grounding algorithm is recursive, operating on the structure of the formula being grounded. It produces an "answer" for each sub-formula. The extended relation $T_{\overline{x}}$ is the "answer" to formula $\phi(\overline{x})$ with respect to structure $\mathcal{A}$, iff for all $\overline{a} : \overline{x} \rightarrow A$, $\psi_T(\overline{a})$

is the reduced ground formula for $\phi(\overline{a})$. The notation $\overline{a} : \overline{x} \rightarrow A$, means that each value a_i, $1 \leq i \leq k$ in tuple $\overline{a} = (a_1, \ldots, a_k)$ is taken from the sort of variable x_i in attributes $\overline{x} = (x_1, \ldots, x_k)$. The answer for an atomic formula is obtained from the given relation when the predicate involved is an instance symbol, and from the universal relation when it is an expansion symbol. The answer for a sentence is an extended relation containing only the empty tuple; the formula associated with that tuple is the reduced grounding of the sentence.

2.3 MXG Language Syntax

In this section we describe the language for problem specification and instance description, which are distinct, in MXG. The MXG language syntax is, in part, a co-operative effort involving the developers of MXG [41], the developers of the solver MidL [37] and others. Differences between the languages of MXG and MidL primarily reflect differences in what is implemented in the respective systems.

Vocabulary symbols in an axiomatization have three distinct roles:

- 'Instance vocabulary' consists of symbols whose interpretation is given by an instance;
- 'Solution vocabulary' is symbols whose interpretations comprise a solution;
- 'Auxiliary vocabulary' is symbols that are not part of the instance or solution.

The solution and auxiliary vocabulary together form the 'expansion vocabulary', the symbols whose interpretations must be constructed by the solver. In the MXG problem specification file, each vocabulary symbol must be declared prior to its use. A problem specification file for MXG consists of 3 sections:

Given: has declarations of all types and of all instance vocabulary symbols;

Find: has declarations of the solution vocabulary symbols;

Satisfying: has axioms, and declarations of auxiliary vocabulary, if any.

As an example, Figure 2.2 gives an MXG specification, together with a sample instance description, for the clique problem. The first line of each is a comment. MXG comments come within /*, */ or in a line after //. Vtx is the sort for vertices; the instance vocabulary

```
/* "Clique" Problem Specification */
Given:    type Vtx;
          Edge(Vtx, Vtx)

Find:     Clique(Vtx)

Satisfying:
          ! x y : (Clique(x) & Clique(y)) => (Edge(x,y) | x=y)


// "Clique" Sample Instance File
Vtx  =  [1..3]
Edge =  { 1, 2; 1, 3 }
```

Figure 2.2: An MXG problem specification and instance description

Logical Symbol	$\forall$	$\exists$	$\wedge$	$\vee$	$\neg$	$\supset$	$\equiv$
ASCII Representation	!	?	&	!	~	=>	<=>

Table 2.1: ASCII equivalents for Logical Symbols

is the binary relation Edge over Vtx $\times$ Vtx; the solution vocabulary is the unary relation Clique over Vtx.

An instance description specifies the elements for each type, and the interpretation of each instance vocabulary symbol. In the instance description of Figure 2.2:

Vtx = [1..3] in the instance description specifies elements of Vtx to be [1..3].

Edge = {1, 2; 1, 3} in the instance description specifies tuples of relation Edge to be {(1,2), (1,1)}.

The axioms say that the interpretation of Clique is a clique in the graph. The Satisfying part consists of ASCII representations of FO formulas. The mapping from logical operators to ASCII symbols is provided in Table 2.1

Other relevant aspects of the language of the current version of MXG are:

- A predicate or constant symbol, as well as a type name, is a string of $[A-Za-z0-9\_]^*$ starting with an upper case letter $[A-Z]$. Variables are strings starting with a lower case letter.

- No function symbols are allowed.
- MXG has no variable declarations. The types (sorts) of each argument to each predicate symbol is declared, and the type of each variable, in a sentence, is inferred from its position as arguments to predicate symbols, which must be consistent.
- Each type is an ordered finite set given by the instance. The ordering is determined by the form of the instance description: Numerical if expressed as a range of integers; otherwise as enumerated. For each type, constant symbols MIN and MAX, binary relation symbols $<$, $\leq$, etc., and binary relation SUCC are all implicitly defined, with the natural semantics. Types are disjoint, so two elements are comparable only if of identical type.
- Bounded quantifiers are supported: $\forall$x y $<$ x : ϕ(x,y) is equivalent to $\forall$x $\forall$y : (y$<$x $\supset$ ϕ(x,y)); $\exists$x y $<$ x : ϕ(x,y) is equivalent to $\exists$x $\exists$y : (y$<$x $\wedge$ ϕ(x,y)).
- A cardinality constraint in MXG is a universal formula of the form $\forall \vec{x} : \odot(n; \vec{y}; \phi(x, \vec{y}))$, where $\odot$ is one of UB, LB, or CARD, for upper bound, lower bound, and equivalence, respectively. For each $\overline{a} : \overline{x} \rightarrow A$, the formula constrains the number of $\overline{b} : \overline{y} \rightarrow A$ for which $\phi(\overline{a}, \overline{b})$ is true in the expansion structure $\mathfrak{B}$:

$$\forall \overline{a} : lb \leq |\{\overline{b} : \overline{y} \rightarrow A, \mathfrak{B} \vDash \phi(\overline{a}, \overline{b})\}| \leq ub.$$

  In the above, values of lb, ub, the lower bound and upper bound, are set for UB, CARD, and LB as follows:

  - $lb = 0$, $ub =$ BOUND for UB,
  - $lb = ub =$ BOUND for CARD,
  - and $lb =$ BOUND, $ub = k$ for LB, where k is the size of set $\{\overline{b} : \overline{y} \rightarrow A\}$.

  The value of BOUND is:

  - n, if n is a natural number.
  - i, if n is a constant symbol c of type D where that element is the the i^{th} element in the order.

  For example $\forall$u : UB(1;v;Edge(v,u)) says the in-degree of every vertex is at most 1.

- MXG supports inductive definitions in the following form. Each inductive definition consists of a set of rules of the form *Head* $\leftarrow$ *Boby*, appearing within {}, where the head is an atomic formula and the body is a quantifier-free FO formula. Variables occurring in the head are implicitly universally quantified. Free variables in the body which do not also occur in the head are implicitly existentially quantified. For example, the definition of transitive closure can be written like:

$$\begin{array}{ll} \{ & \mathrm{TC(u,v)} \leftarrow \mathrm{Edge(u,v)} \\ & \mathrm{TC(u,v)} \leftarrow \mathrm{TC(u,w)} \wedge \mathrm{Edge(w,v)} \quad \} \end{array}$$

  MXG supports two "well-behaved" fragments of inductive definitions, which have a semantics that allows them to be easily handled by classical methods. These are explained in Section 2.1.1

Chapter 3

Writing Efficient Axiomatizations

There are many logically equivalent ways to axiomatize a problem in any particular language. A particular solver may perform very differently, in terms of time to produce a solution, with different (though logically equivalent) axiomatizations. In Section 3.1, we illustrate a few techniques in writing axiomatizations which we have found to be useful to improve solver performance, and give some explanation of why they might do so. Some of these techniques are related specifically to the speed of grounding, some to the performance of the ground solver, and some to both.

In Section 3.2, we compare the performance of MXG [42], using the best axiomatizations we found with these techniques, with several other solvers.

3.1 Techniques to Write Efficient Axiomatization

In this section, we will describe some techniques we have used in our axiomatizations of benchmark problems that we present in the following section.

3.1.1 Solution Symmetry Breaking

An important technique which is widely used in SAT and CSP (Constraint Satisfaction Problem) communities, is symmetry breaking [46]. This technique can affect SAT solver performance dramatically. An example of a so-called "solution symmetry" occurs with the graph k-coloring problem as follows. For any solution S, the set of vertices can be divided into k independent sets. It does not matter which color we assign to these sets as long as

any two sets have different colors. In this way, we can obtain $k!$ symmetric solutions for S. We frequently can add so-called "symmetry breaking" axioms to eliminate all but one representative solution for each such symmetry.

Symmetry breaking axioms often improve performance, even when only one solution is needed, presumably because they help the solver effectively to eliminate many symmetric "near-solutions". These axioms also improve performance when there is no solution because they help the solver to eliminate many symmetric "non-solutions".

In order to break symmetries, we have to identify symmetries and then add axioms to break them. For the k-coloring problem, we can remove the symmetries by defining a new predicate Min(c,v) which is true if v is the minimum vertex in ordering of vertices that has color c. The symmetry breaking axiom says that for any two colors c_1 and c_2 where $c_1 < c_2$, the minimum vertex which has color c_1 should precede the minimum vertex which has color c_2.

3.1.2 Eliminating Symmetric Clauses

Symmetric clauses are introduced by MXG when we have formulas such as:

$$\forall \text{ x y z: } ((\text{Color(x,y)} \wedge \text{Color(x,z)}) \supset \text{y=z})$$

For a fixed v, MXG produces $n^2 - n$ clauses of form $\neg$(Color(v,b) $\wedge$ Color(v,c)) where $b \neq c$ and size of the sorts for x and y is n. Half of these clauses are redundant because $\neg$(Color(v, b) $\wedge$ Color(v,c)) and $\neg$(Color(v, c) $\wedge$ Color(v, b)) are logically equivalent. These redundant clauses make CNF files bigger and also increase grounding time for MXG. They have a small effect on a SAT solver performance as well.

We can eliminate these symmetries by re-writing the axioms. For example, we can re-write the above formula as follows:

$$\forall \text{ x y z: } ((\text{Color(x,y)} \wedge \text{Color(x,z)}) \supset \text{y=z}) \equiv$$
$$\forall \text{ x y z: } (\text{y} \neq \text{ z} \supset \neg(\text{Color(x,y)} \wedge \text{Color(x,z)}))$$

If $y < z$ is used instead of $y \neq z$ then grounder produces $n^2(n-1)/2$ clauses instead of $n^2(n-1)$. It reduces the number of clauses by leaving out the symmetric clauses. It also reduces the grounding time because the size of the table for $y < z$ is almost half of the size of table for $y \neq z$. A more efficient formula is:

$$\forall \text{ x y z: } (\text{y<z} \supset \neg(\text{Color(x,y)} \wedge \text{Color(x,z)}))$$

$y < z$ can be moved to quantifier part, please see section 3.1.6 for details.

The current version of MXG keeps each clause as a set of propositional variables and ground clauses generated for each formula as a set. Because of the way MXG stores the clauses, it automatically removes these symmetric clauses. Of course, if MXG needs to introduce new Tseitin [50] variables, it cannot remove the symmetric clauses anymore.

3.1.3 Redundant Axioms

Adding redundant axioms is another technique used in SAT and CSP [52]. For a problem, a redundant axiom is an axiom which can be removed without changing the set of solutions. A redundant axiom is satisfied by all the solutions of the problem. Redundant axioms may help by adding more constraints on particular properties. It is not well-understood why particular redundant axioms help performance. Natural explanations are that they increase the amount of unit propagation performed for some partial assignments, or that they help a clause learning solver learn more useful clauses. For example, consider the following set of formulas defining a bijection on P(x,y).

$$\forall\ x : \exists\ y : P(x,y)$$
$$\forall\ x\ y\ z{<}y{:}\ \neg(P(x,y) \wedge P(x,z))$$
$$\forall\ x\ y\ z{<}y{:}\ \neg(P(y,x) \wedge P(z,x))$$

The following formula is a redundant axiom which can be added.

$$\forall\ y : \exists\ x : P(x,y)$$

In our experience, we found the provided axiom to improve performance when we define a bijection.

3.1.4 Auxiliary Predicates

Sometimes, a FO sub-formula appears more than once in different axioms. If MXG does not produce Tseitin variables for this sub-formula then it is fine to use the sub-formula itself several times. However, if MXG introduces Tseitin variables then this may slow down the solver performance because the Tseitin variables assigned by MXG for a sub-formula are different from those Tseitin variables assigned to a distinct occurrence of the same sub-formula. So, introducing an auxiliary predicate corresponding to the sub-formula not only reduces the grounding time but also improves the solving time.

3.1.5 Inductive Definition as a Pre-processing Technique

The use of inductive definitions seems to be obvious. The reason we are introducing it as a technique is that sometimes defining a problem does not need to be done with inductive definitions. However we can pre-process the instance by using inductive definitions. If the inductive definition is a Horn-ID (See Section 2.1.1) then it will be computed in a pre-processing step in grounding and the result will be used in the rest of grounding as if it was an instance predicate.

3.1.6 Bounded Quantifiers and Ordering

MXG allows use of bounded quantifiers, which help make axioms readable, but may at times slow down MXG. These orderings should be carefully chosen. Sometimes, we can re-write these qualifiers using implications and conjuctions and improve the performance

If we have a set of inequalities $\{e_1, ...e_n\}$ of form $l \odot r$, where l and r are terms and $\odot$ is one of $<, >, \leq, \geq, \neq$ and a formula such as:

$$\forall x_1 \dots x_k : (e_1 \wedge \dots \wedge e_n) \supset \phi$$

where ϕ contains both instance and expansion vocabulary symbols, the set of inequalities can be partitioned into two sub-sets:

- E_1: inequalities in which both terms (l, r) appear in ϕ as an argument to a predicate symbol of the instance vocabulary.
- $E_2 = \{e_1, ...e_n\} \setminus E_1$

The inequalities in E_2 can be written as bounded quantifiers, or can be written as a big conjunction at the beginning of the formula :

$$\forall x_1 \dots x_k : (e'_1 \wedge \dots \wedge e'_i) \supset (e''_1 \supset \dots (e''_j \supset \phi) \text{ where } e'_1, \dots e'_i \in E_2 \text{ and } e''_1 \dots e''_j \in E_1.$$

This reformulation is important because for grounding MXG creates a table for each sub-formula, with the set of attributes which are free variables of arguments to a predicate symbol of the instance vocabulary.

For formula $\phi(\overline{x})$, with free variables $\overline{x} \subset \{x_1, \dots, x_k\}$, MXG creates a table with attributes $\overline{x'}$. $\overline{x'}$ is the set of free variables in $\phi(\overline{x})$ that appears as arguments to a predicate symbol of the instance vocabulary. MXG creates a table for each of inequalities e_d,

$1 \leq d \leq n$, with attributes of l_d and r_d if they appear as arguments to an instance predicate symbol.

To ground $\psi_j = (e''_j \supset \phi) \equiv \neg(e''_j \wedge \neg\phi)$, MXG joins table of $\neg\phi$ with table of e''_j. As attributes of table for e''_j are subset of attributes of table for $\neg\phi$, in this join the number of tuples in the result table is reduced. Continuing with the new table, and computing table for $\psi_{j-1} = (e''_{j-1} \supset \psi_j)$, ..., $\psi_1 = (e''_1 \supset \psi_2)$, at each step the number of tuples is reduced, so the actual computation time for joins is reduced at each step.

3.1.7 Cardinality Constraints

Use of cardinality constraints increases the readability of the axioms and also improves performance of the grounder and solver. For example, we want to define a bijection on the predicate P(x,y):

$$\forall\ \mathrm{x} : \exists\ \mathrm{y} : \mathrm{P(x,y)}$$
$$\forall\ \mathrm{x\ y\ z{<}y}: \neg(\mathrm{P(x,y)} \wedge \mathrm{P(x,z)})$$
$$\forall\ \mathrm{x\ y\ z{<}y}: \neg(\mathrm{P(y,x)} \wedge \mathrm{P(z,x)})$$

The three formulas define a bijection on P(x,y). Suppose the size of sorts for x and y is n. If n is big then it will take huge amount of time to ground these formulas. There will be n CNF clauses of size n and $n^2(n-1)$ binary clauses. The above property can be written in the following cardinality constraint:

$$\forall\ \mathrm{x} : \mathrm{CARD(1;\ y;\ P(x,y))}$$

which takes less time to ground. The number of clauses produced this way is n ground cardinality clauses of size n.

It is hard to define bounds with pure FO formulas. For example, to bound the number of children for each node to be at most b is hard to axiomatize without cardinality constraints, especially if b is given with the instance.

3.1.8 Handling Negations and Disjunctions

Negations and disjunctions are costly for MXG. For a predicate of arity k where the size of each domain is n, to compute the negation n^k tuples are examined.

Disjunctions are costly because each tuple in the table of left hand side of the disjunction, even *false*, must be disjuncted with each tuple in the table of the right hand side. So for

a formula like $\neg a \vee \neg b \vee \neg c \vee \phi$, it is useful to apply the "DeMorgan" rule to produce $\neg(a \wedge b \wedge c) \vee \phi$.

MXG handles negation in front of the whole formula in an efficient way which is much faster than applying the negation to the formula directly. This feature should be carefully used such that, having a negation outside the whole formula does not leave lots of disjunctions and negations inside.

3.2 Results and Axiomatizations

Here we present an empirical comparison of the performance of MXG using MXC [6], a high performance SAT solver capable of handling cardinality constraints, as SAT solver (Denoted MXG+MXC) and our best axiomatizations, with MidL [37], an FO(ID) model expansion solver produced at the Katholieke Universitat Lueven (KU Leuven), and with several ASP solvers (clasp [26], smodels [49] and DLV [32]). We compare with ASP solvers for several reasons, including

- ASP is the most widely known and most developed framework which is comparable in goals and techniques to ours;
- The Asparagus repository [2] of ASP solvers, axiomatizations, and benchmark instances provides a useful resource for efficiently carrying out such a comparison.

The following solvers were compared:

- MXG 0.171 with MXC 0.5 for the SAT solver.
  Available from "http://www.cs.sfu.ca/research/groups/mxp";
- MidL 2.2.0, an "native" FO(ID) model expansion solver.
  Available from "http://www.cs.kuleuven.be/~dtai/krr/software/midl.html";
- smodels 2.32, a "native" ASP solver.
  Available from "http://www.tcs.hut.fi/Software/smodels/";
- clasp 1.0.4, a "native" ASP solver with clause learning.
  Available from "http://www.dbai.tuwien.ac.at/proj/dlv/";
- DLV 2006-7-14, an ASP solver for disjunctive logic programs.
  Available from "http://www.cs.uni-potsdam.de/clasp/";

Grounding for smodels and clasp was done using Lparse [48] version 1.0.17. MidL uses a variant of Lparse which comes with the MidL download and DLV does its own grounding.

Our choice of benchmark problems reflects three goals:

- to demonstrate use of techniques we provided;
- to use a variety of problems with non-trivial instance collections;
- and to use problems for which ASP axiomatizations have been provided (presumably by people with some ASP expertise).

We study the following problems:

- Graph K-coloring
- Latin Square Completion
- Social Golfer
- Bounded Spanning Tree
- Blocked Queens

We define the problems, specify the instances used, and give our best MXG axiomatizations in section 3.2.4.

3.2.1 Overall Solver Performance

When we consider total solving time (time for grounding plus time for the ground solver), MXG+MXC had the best performance of the five solvers tested on three of the five problems studied (K-Coloring and Latin Square and Social Golfer), was second best on Blocked Queens, and third on Bounded Spanning Tree. Thus, overall, MXG+MXC performance is competitive with ASP solvers and MidL on the benchmark problems we have studied.

3.2.2 Cumulative Performance Plots

The performances plots have the following format. The X-axis is time in seconds; the Y-axis is the cumulative number of instances solved within a given time bound. For example, a point at (5,10) indicates that among the instances tested, 5 were solved in 10 seconds or less each, and the remaining all required more than 10 seconds each. We plot a point for each

instance solved, so if the i^{th} instance solved required n seconds, there is a point at (i,n). We connect the points for each solver with a curve, as a visual aid.

Presenting solver times for non-trivial sets of instances of NP-hard problems is not always simple. Solvers of essentially the same quality often succeed on different instances even within a collection of very similar instances. (Even a tiny change to a solver can affect *which* instances are solved, while not affecting the number solved.) Thus, tables of run-times are hard to interpret, especially for large collections of instances. Also, we are typically interested in *scaling* - how performance with problem size - for a general *category of instances*, more than with performance on particular small instances. Often the solvers with the best scaling performance are not the fastest on small instances. Further, run-times typically exhibit very high variance, so the mean run time may be dominated by a few extreme instances and not reflect typical performance. The cumulative plot format reduces emphasis on particular run-times, and gives a good overall picture of relative performance and scaling of solvers on a collection of instances. (Looking at run-times for particular instances often is very interesting, it is just not the best way to see the overall trend.)

Observe that the X-axis of all plots is logarithmic, and thus some care is required in interpreting the curves. The curve for a sequence of instances with linear growth in run-time will show up as a curve with large and rapidly increasing slope. A straight line suggests an exponential increase in run-times, and the smaller the slope the higher the exponent.

We ran all tests with a time cut-off of 30 minutes (1800 seconds), so the righthand edge of each plot is at 1800 seconds. The upper edge of each plot is at a value a bit larger than the number of instances in the relevant collection, so we can see the points when all instances were solved within the cut-off time. If (x,y) is the extreme upper-right point of the curve for solver A, then A solved y instances in x seconds or less, and failed to solve any of the remaining instances within the 30-minute cut-off. In all cases, we report the total time for both grounding and solving.

3.2.3 Test Platform

All tests were run on Sun Fire VZ20 Dual Opteron computers with two 2.4 GHz AMD Opteron 250 processors having 1MB cache and 2GB of RAM per processor. The machines were running Suse Enterprise Linux 2.6.11. The executables for DLV, MidL were downloaded from the respective solver sites Executables for smodels, clasp and MXG were compiled with gcc version 3.3.4, using the default settings of the makefiles provided with the solver sources.

3.2.4 Problem Specifications and Performance Plots

In this section, we present the plots showing the relative performance of the various systems on the chosen benchmark problems, and also the MXG axiomatizations we used. Most of the techniques we described in previous sections are used in these axiomatizations.

For the ASP solvers, we used axiomatizations downloaded from Asparagus [2]. For MidL, we used axiomatizations for K-Coloring, Bounded Spanning Tree and N-Queens included in the MidL download package. To the latter, we added an axiom enforcing the blocked cells to produce a Blocked Queens axiomatization. For the remaining problems we used a direct translation of our MXG axiomatizations from [41] into the MidL language, which primarily amounted to replacing bounded quantifiers with their definitions.

We will say that a solver "solved" an instance if it produced a solution or correctly reported it unsatisfiable, and "failed" to solve the instance if it did not halt within the 30 minute time cut-off. We verified every solution produced during our tests with both MXG and smodels. For each problem, we estimate an order of preference of solvers based on performance. Since we prefer to reward good scaling over fast solving of small instances, our ordering is as follows: we prefer those solvers that solved the most instances within the cut-off time, and among those we prefer the one that minimized the maximum time for solving an instance. Solver order could change with an increased cut-off time, but this will be the case with any preference scheme that rewards good scaling.

Graph K-coloring

Graph K-coloring problem is a classic and well-studied NP-hard search problem. The instance is a graph and a number K, and the solution is a proper K-coloring of the graph. We want to find a function mapping the set of vertices to set of colors such that no two adjacent vertices have the same color. The current version of MXG does not support functions so we use a binary relation for our purpose.

Axiom(2), in Figure 3.1, states that no more than one color can be assigned to a node. Axiom(3) states two ends of an edge cannot have the same color. Axiom(4) states that every node has a color. Axiom(1) is a solution symmetry-breaking axiom. In Axiom(2) we used bounded quantifiers to remove symmetric clauses.

Our test set consisted of 17 instances, five 3-coloring instances from Asparagus and twelve instances from the LEI category of the graph coloring benchmark collection at [3].

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

Find: Color(Vtx, Clr)

Satisfying:

Color(MIN,MIN)	(1)
$\forall$ x y z < y :$\neg$(Color(x, y) $\wedge$ Color(x, z))	(2)
$\forall$ x y : (Edge(x,y) $\supset$ ($\forall$ z : $\neg$(Color(x, z) & Color(y,z))))	(3)
$\forall$ x: $\exists$ y : Color(x, y)	(4)

Figure 3.1: MX Problem Specification for Graph K-Coloring

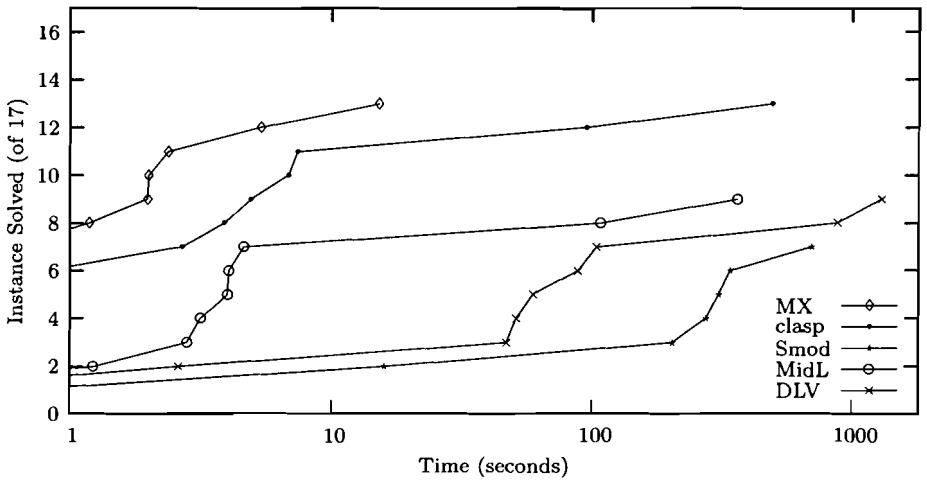

Figure 3.2: Performance on Graph K-Coloring

These latter are are challenging instances on graphs of 450 vertices, variously with 5, 15 and 25 colors. All 17 instances are colorable with the alloted number of colors.

The performance of the solvers is shown in Figure 3.2. The figure clearly shows the following order of solvers, from best to worst: MXG+MXC, clasp, MidL, DLV, smodels. Notice that no solver was successful on all instances: MXG+MXC and clasp solved 13 of the 17 instances, while others solved 10 or fewer. The four instances that were not solved by MXG+MXC went unsolved by all solvers tested.

Given : type Num;
Preassigned(Num, Num, Num)

Find : Cell(Num, Num, Num)

Satisfying:

$\forall$ x y z : (Preassigned(x, y, z) $\supset$ Cell(x, y, z))	(1)
$\forall$ x z: $\exists$ y : Cell(x, y, z)	(2)
$\forall$ y z: $\exists$ x : Cell(x, y, z)	(3)
$\forall$ x y: $\exists$ z : Cell(x, y, z)	(4)
$\forall$ x z y1 y2<y1 : $\neg$(Cell(x, y1, z) $\wedge$ Cell(x, y2, z))	(5)
$\forall$ y z x1 x2<x1 : $\neg$(Cell(x1, y, z) $\wedge$ Cell(x2, y, z))	(6)
$\forall$ x y z1 z2<z1 : $\neg$(Cell(x, y, z1) $\wedge$ Cell(x, y, z2))	(7)
$\forall$ x y z : (Preassigned(x, y, z) $\supset$ Cell(x, y, z))	(1')
$\forall$ x y : CARD(1, z; Cell(x, y, z))	(2')
$\forall$ x z : CARD(1, y; Cell(x, y, z))	(3')
$\forall$ z y : CARD(1, x; Cell(x, y, z))	(4')

Figure 3.3: MX Problem Specification for Latin Square Completion

Latin Square Completion

A latin square[16] is an n by n matrix with elements of $\{1, \ldots, n\}$ such that every row and every column has all elements. In the Latin Square Completion problem, an instance is the number n plus prescribed values for certain elements.

Figure 3.3 shows the axiomatization of the latin square completion problem. Axioms (1) to (4) is enough by itself to define the problem. However if we only use these four axioms and try to solve the instances MXG may fail even for the easiest ones. Here, we use redundant axioms to improve our performance. Axioms(5) and (6) require that each row and column to be a bijection on $\{1, \ldots, n\}$. Axiom(7) says that each cell cannot have more than one element. The three redundant axioms increase the grounding time a little bit, but reduces the solving time significantly. Here, also we used bounded quantifications to remove symmetric clauses.

In figure 3.3, axioms (1') to (4') are the axiomatization of Latin Square Completion with the cardinality constraints.

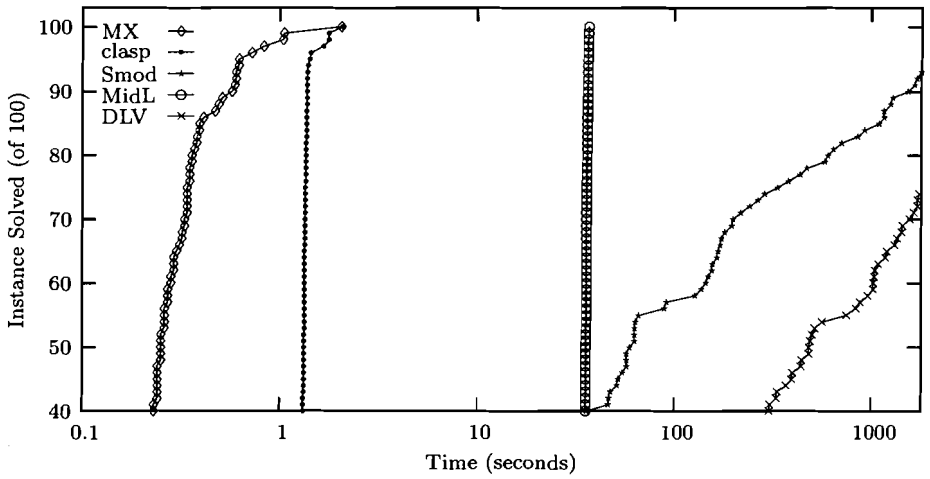

Figure 3.4: Performance on Latin Square

The test set consists of 100 instances from Asparagus. All are of size 30-by-30, and all have solutions. The apparent ordering is: MXG+MXC, clasp, MidL, smodels, DLV. The performance plot is shown in Figure 3.4.

Social Golfer

In the Social Golfer problem, the goal is to find a schedule for $g * s$ golfers into g groups of s players over w weeks, such that no two golfers play in the same group more than once. Figure 3.5 shows axiomatization of the problem using cardinality constraints. The axiomatization of the social golfer without using cardinality constraints is more complicated (See [41]).

One of the techniques we used in axiomatization of Figure 3.5 is introducing the auxiliary predicate Soc. $\mathsf{Soc}(w, p_1, p_2)$ is true when players p_1 and p_2 play in the same group in week w. Introduction of this predicate has a significant improvement on the number of instances solved. As previous examples, bounded quantification is used to remove symmetric clauses.

Social golfer is a highly symmetric problem. For any valid scheduling for players, permuting players of a group, groups in a week, and weeks gives a new valid schedule, which is symmetric to the original one. Each solution is symmetric to $(g * s)! * w! * g!$ others by changing weeks, players and groups. These symmetries affect solving time dramatically if

Given : type Players Groups Weeks;
Groupsize : Players

Find : Plays(Players, Weeks, Groups)

Satisfying:

MP(Weeks, Groups, Players)
SP(Weeks, Players)
Soc(Weeks, Players, Players)
$\forall$ p1 p2>p1 w : (Soc(w,p1,p2) $\Leftrightarrow$ ($\forall$ g : (Plays(p1,w,g) $\wedge$ Plays(p2,w,g)))) (1)
$\forall$ w g: CARD(Groupsize, p; Plays(p,w,g)) (2)
$\forall$ p w : CARD(1, g; Plays(p, w, g)) (3)
$\forall$ p1 p2>p1 : UB(1, w; Soc(w,p1,p2)) (4)
$\forall$ g1 g2>g1 p1 p2<p1 : $\neg$(Plays(p1, MIN, g1) $\wedge$ Plays(p2, MIN, g2)) (5)
$\forall$ w p : (SP(w,p) $\supset$ (p > MIN $\wedge$ Plays(p, w, MIN))) (6-1)
$\forall$ w p2>MIN p1>p2 : $\neg$(SP(w,p1) $\wedge$ Plays(p2, w, MIN)) (6-2)
$\forall$ w : CARD(1; p; SP(w,p)) (6-3)
$\forall$ w1 w2>w1 p1 p2$\leq$p1 : $\neg$(SP(w1, p1) $\wedge$ SP(w2, p2)) (6-4)
$\forall$ w : Plays(MIN, w, MIN) (7-1)
$\forall$ w g : CARD(1; p; MP(w,g,p)) (7-2)
$\forall$ w g p : (MP(w,g,p) $\supset$ (! p1 < p : $\neg$(Soc(w, p1, p)))) (7-3)
$\forall$ w p1 p2$\leq$p1 g1 g2>g1 : $\neg$(MP(w, g1, p1) $\wedge$ MP(w, g2, p2)) (7-4)

Figure 3.5: MX Problem Specification for Social Golfer

the problem does not have a solution. Proving the unsatisfiability of the problem may take exponential time to check each and every one of these solutions. The symmetry of players, groups and weeks are removed by axioms 4, 5 and 6 respectively in Figure 3.5.

The test set consists of 174 instances from Asparagus, spanning (but not covering) the parameter range: number of weeks from 2 to 8; group size from 2 to 6; number of groups from 2 to 8. We know 72 instances to have solutions and 94 to have no solution, leaving 7 of unknown status. The order of solvers is: MXG+MXC, clasp, smodels, DLV, MidL. The performance plot is shown in Figure 3.6.

Bounded Spanning Tree

A spanning tree of a graph is a sub-graph of it, which is a tree and covers every vertex. $K-$Bounded Spanning tree is a spanning tree in which the out-degree of every vertex is not

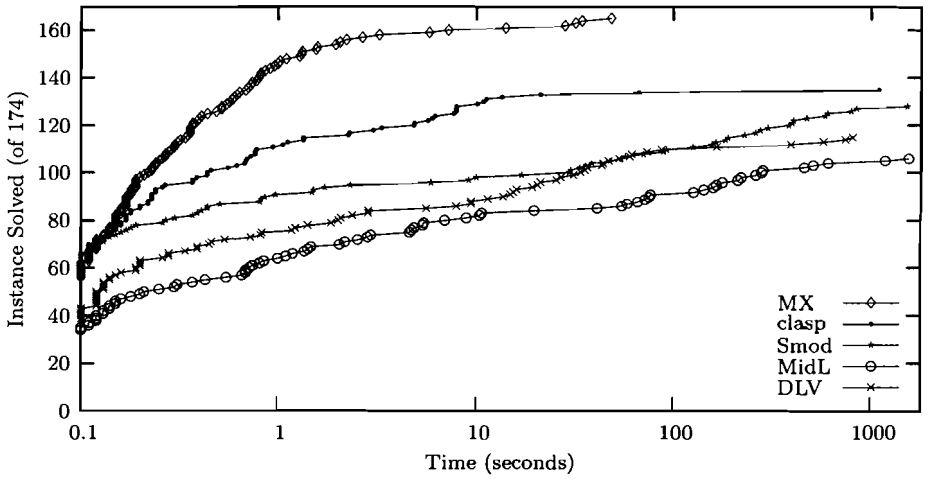

Figure 3.6: Performance on Social Golfer

Given : type Vtx;
Edge(Vtx,Vtx);
Bound : Vtx

Find : Bstedge(Vtx,Vtx)

Satisfying:
Map(Vtx,Vtx)

∀ x: CARD(1, y; Map(x,y))	(1)
∀ y: CARD(1, x; Map(x,y))	(2)
∀ v u : (Bstedge(u,v) ⊃ Edge(u,v))	(3)
∀ x : UB(1, y; Bstedge(y,x))	(4)
∀ u : UB(Bound, v; Bstedge(u, v))	(5)
∀ u v x y≤x : (Map(x,u) ∧ Map(y,v) ∧ Bstedge(u,v))	(6)
∀ v f>MIN: (Map(v, f) ⊃ (∃u:Bstedge(u,v)))	(7)

Figure 3.7: MX Problem Specification for Bounded Spanning Tree

greater than the bound K.

Figure 3.7 illustrates the axiomatization of the bounded spanning tree in MX. We gain a huge improvement on grounding time by using cardinality constraints to restrict the bounds.

MXG cannot find the model of an inductive definition, which is not "well-behaved". To

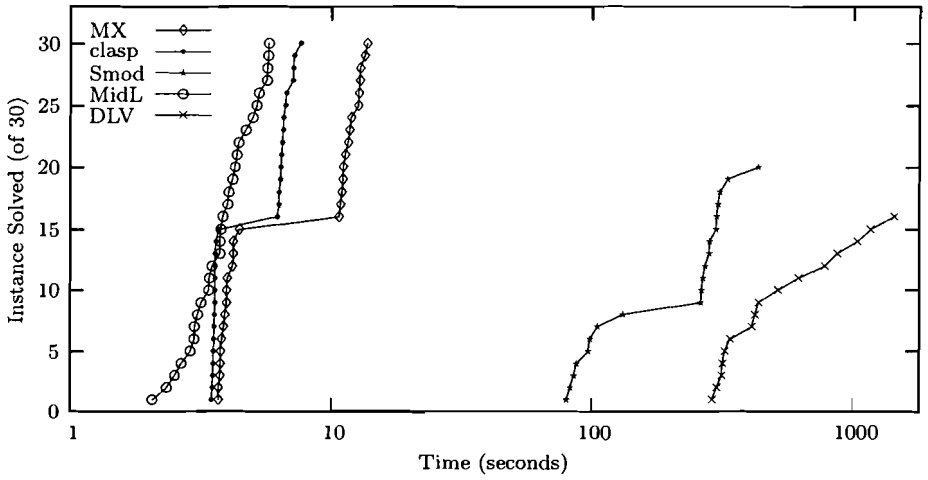

Figure 3.8: Performance on Bounded Spanning Tree

define recursive properties, auxiliary predicates we need to introduce auxiliary predicates. In axiomatization of Figure 3.7 we define Map to be a a one-to-one and onto mapping function from vertices of instance graph to natural numbers. Our intention is to define a total ordering on vertices of graph by this mapping. Having a total ordering on vertices of graph, it is not hard to express properties of a *tree*.

The test set is 30 instances from Asparagus, some with 35 and some with 45 vertices. All have solutions. The solver order is: MidL, clasp, MXG+MXC, smodels, DLV. The performance plot is shown in Figure 3.8.

Blocked Queens

The Blocked Queens problem is a generalized version of n-queens problem. A chessboard is provided which has x prescribed queens. The goal is to place the remaining $n - x$ queens on the chessboard such that no two queens attack each other.

In this problem, we have implemented lots of techniques to write the simplest and the most efficient axiomatization. Figure 3.9 shows the axiomatization of this problem.

Current version of MXG does not support arithmetic. But we are able to define most of the arithmetic operations by using inductive definitions and built-in predicate SUCC. MXG pre-computes the interpretation of a predicate defined by an inductive definition

Given : type Num;
Block(Num, Num);
Find : Queen(Num, Num)

Satisfying:
Diff(Num,Num,Num)
{Diff(i,x2,y2) <- (x2 = MIN ∧ y2 = i) (1)
Diff(i,x2,y2) <- (Diff(i,x1,y1) ∧ SUCC(x1,x2) ∧ SUCC(y1,y2))} (2)
∀ x y: (Queen(x,y) ⊃ Block(x,y)) (3)
∀ i x1 y1 x2 y2: (y1<y2) ⊃ ((x1<x2) ⊃
(Queen(x1,y1) ∧ Queen(x2,y2) ∧ Diff(i,x1,x2) ∧ Diff(i,y1,y2))) (4)
∀ i x1 y1 x2 y2: (y2<y1) ⊃ ((x1<x2) ⊃
(Queen(x1,y1) ∧ Queen(x2,y2) ∧ Diff(i,x1,x2) ∧ Diff(i,y2,y1))) (5)
∀ x : CARD(1;y;Queen(x,y)) (6)
∀ y : CARD(1;x;Queen(x,y)) (7)

Figure 3.9: MX Problem Specification for Blocked Queens

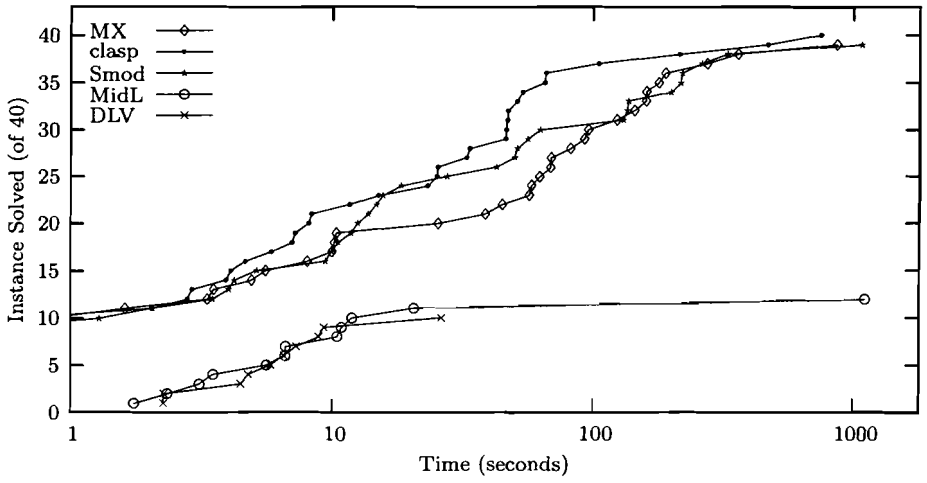

Figure 3.10: Performance on Blocked Queens

during grounding time, if body of each rule is a conjunction of positive predicates, and no expansion predicate other than head predicate appears in body. The predicate is then used as an instance predicate. In Figure 3.9, Diff is precomputed by MXG. Diff(i, x, y) is true if and only if $i = y - x + 1$. In blocked queens no two queens can be on the same row, column,

or diameter. (x_1, y_1) and (x_2, y_2) are on the same diameter if and only if $|x_1 - x_2| = |y_1 - y_2|$. Axiom (3) forces that every prescribed queen is a queen in the final solution. Axioms (4) to (7) enforce the constraints about diameters, rows and columns.

The way we put inequalities in axioms(4) and (5) reduces the grounding time, although the same set of clauses will be generated. We explained the reason in section 3.1.6.

The test set consists of 40 instances from Asparagus, of sizes from 28-by-28 to 56-by-56, 20 of which have solutions. While having a similar flavor to Latin Square completion, the performance profile (at least on the Asparagus instances) is different, in that all solvers found the instances to vary considerably in difficulty. Moreover, the order has changed considerably, to: clasp, MXG+MXC, smodels, MidL, DLV. The performance plot is shown in Figure 3.10.

Chapter 4

Extended Example: Phylogenetic Inference

4.1 The Binary Camin-Sokal Phylogeny Problem

We study a simple "large parsimony" problem in character-based cladistics. A group of species is characterized by a set of *characters*. Each character can take one of several *states*, and each species is described by a vector giving a state for each character. The input is a set of species vectors and the goal is to construct a tree with nodes labeled by character vectors, so that the vector of every input species labels some node. Changes of a character's state along an edge are mutations. Problem variations result from differing cost metrics and restrictions on character changes. A tree minimizing the cost metric is a "most parsimonious tree". In the cladistic Camin-Sokal (CCS) problem, the states of each character are ordered and mutations must be increasing on this order. This is appropriate when the direction of evolutionary change of characters is assumed to be known. The goal is to minimize the total number of mutations. The decision version of the problem, even in the binary case where each character has just two states, is NP-complete [11].

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

Instance: Set S of n distinct vectors from $\{0,1\}^m$; natural number B.

Question: Is there a directed tree $T = (V, E)$, such that:

1) T is rooted at 0^m;

2) $S \subseteq V \subseteq \{0,1\}^m$;

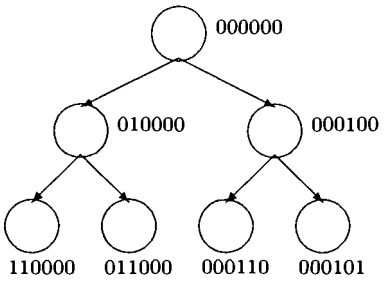

Figure 4.1: Perfect Phylogeny Tree for the given set of species

3) Leaves of T are labeled exactly by S;

4) For each directed edge $(v_1, v_2) \in E$, v_1 and v_2 differ in exactly one character, which is 0 at v_1 and 1 at v_2;

5) $|V| \leq B$.

An alternate equivalent definition allows multiple mutations on an edge but counts the number of mutations not the nodes. The definition we use here was also used in [30], and is easier to axiomatize in MXG.

In a *perfect phylogeny*, each character mutation occurs only once. For the binary CCS, this is equivalent to setting B to maximum of n and $m + 1$, provided that both states of every character occur in S. (Note that if some character has only one state, we may safely delete it.) Figure 4.1 is an example of a perfect phylogeny for a set of 7 species with 6 characters. When an instance does not have a perfect phylogeny, some character mutations must occur more than once in the tree. Since mutations are irreversible in CCS, the same character cannot change more than once on a directed path from the root, so the same mutation will occur in distinct subtrees. The goal is to find a tree that minimizes the number of these "extra mutations". We allow only one mutation per edge, so the number of extra mutations is equivalent to the number of "extra vertices" or "extra edges" needed to construct a phylogeny. Since mutation is irreversible, we may assume that all mutations are from state 0 to state 1, and the tree is rooted at the zero vector. Figure 4.2 shows two phylogenies for a set of 5 species with 5 characters, one is with one extra node (optimal), the other has two extra nodes. Solid black nodes are the extra ones.

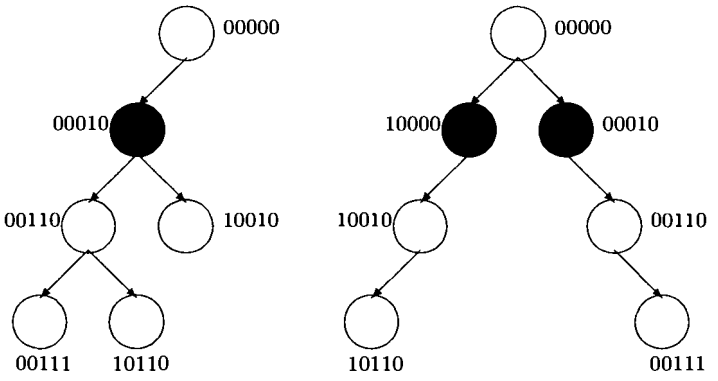

Figure 4.2: Optimal Phylogeny(LHS) - Phylogeny with two extra nodes (RHS)

4.2 MX Axiomatizations of Binary CCS

Here we give three MX axiomatizations of Binary CCS.

- One we find natural and simple, using cardinality constraints (Denoted as MX-Basic);
- One uses no cardinality constraints, and can be solved by straightforward reduction to SAT (Denoted as MX-SAT);
- and one is a translation to MX of the best ASP encoding from [30] (Denoted as MX-ASP).

We produced other distinct axiomatizations, but since none performed better than our basic one (except when using enhancements such as described in Section 4.4), we do not report them.

4.2.1 Basic MX Axiomatization

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

- A(Vertex, Char, State): the set of triples specifying the matrix of species of data. (The first argument is the species.)

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

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

Satisfying:

$\forall$ s $\leq$ NSpecies c : (A(s,c,MAX) $\Leftrightarrow$ Vector(s, c)) (1)
$\forall$ u v : UB(1; c; (Edge(u,v) $\wedge$ $\neg$ Vector(u,c) $\wedge$ Vector(v,c))) (2)
$\forall$ u v : (Edge(u,v) $\supset$ ($\exists$c : ($\neg$Vector(u,c) $\wedge$ Vector(v,c)))) (3)
$\forall$ u v : UB(0; c; (Edge(u,v) $\wedge$ Vector(u,c) $\wedge$ $\neg$ Vector(v,c))) (4)
CARD(NEdges; v, u; Edge(v,u)) (5)
$\forall$ v>MIN : CARD(1; u; Edge(u, v)) (6)

Figure 4.3: MX-Basic axiomatization of binary CCS phylogeny re-construction

- NSpecies: a constant symbol denoting the number of species.
- NEdges: a constant symbol which is always set to |Vertex| $-$ 1.

The solution vocabulary has two binary relation symbols: Edge, the set of edges, and Vector, which labels vertices with character vectors. Vector(v,c) holds if character c has state 1 in the vector labeling v. The axioms (see Figure 4.3) state:

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

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

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

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

Satisfying:
TC(Vertex, Vertex) // TC will be the transitive closure of Edge.

$\forall$ s $\leq$ NSpecies c : (A(s,c,MAX) $\Leftrightarrow$ Vector(s,c)) (1)

$\forall$ v1 v2: (Edge(v1,v2)$\supset$($\exists$ c1 : ($\neg$Vector(v1,c1)$\wedge$Vector(v2, c1) (2)
$\wedge$ ($\forall$c2 : (($\neg$Vector(v1,c2) $\wedge$ Vector(v2, c2)) $\supset$ (c1=c2))))))

$\forall$ u v c : ((TC(u, v) $\wedge$ Vector(u, c)) $\supset$ Vector(v,c)) (3)

$\forall$ u>MIN : $\exists$ v : (Edge(v,u) $\wedge$ ($\forall$v2 : (Edge(v2, u) $\supset$ (v2 = v)))) (4)

$\forall$ u v>u : $\neg$(TC(u, v) $\wedge$ TC(v, u)) (5)

$\forall$ u v : (TC(u,v) $\Leftrightarrow$ ((u = v) $\vee$ Edge(u,v) $\vee$ ($\exists$x : (TC(u, x) $\wedge$ Edge(x,v))))) (6)

Figure 4.4: MX-SAT axiomatization of binary CCS phylogeny re-construction

4.2.2 Non-Cardinality Axiomatization

To determine if we obtain a speed-up over pure SAT solving by using MXC with cardinality constraints, we produced several axiomatizations without cardinality constraints, which MXG grounds to SAT. Figure 4.4 shows the best-performing of these. The axioms state:

- The input species vector i must label vertex i (Axiom 1 - as before);
- Each edge has exactly one character changing from 0 to 1 (Axiom 2);
- On a directed path the set of 1-characters is monotone increasing (Axiom 3), so there are no reverse mutations;
- The graph is a tree (Axioms 4 and 5), since every node but the root has in-degree one and there are no cycles in the transitive closure of Edge;
- TC is the transitive closure of Edge (Axiom 6 provides the lower bound on TC; Axiom 5 the upper bound).

We report results based on the SAT solver minisat, arguably the best all-around SAT solver available.

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

Find : Edge(Vertex, Vertex)

Satisfying :

M(Vertex, Char)
P(Species, Vertex)
TC(Vertex, Vertex)

$\forall$ v : CARD(1; c; M(v,c)) (1)
$\forall$ s : CARD(1; v; P(s,v)) (2)
$\forall$ v : UB(1; u; Edge(u, v)) (3)
$\forall$ u v : (TC(u,v) $\Leftrightarrow$ ((u = v) $\vee$ Edge(u,v) $\vee$ ($\exists$x : (TC(u, x) $\wedge$ Edge(x,v))))) (4)
$\forall$ u v>u : $\neg$(TC(u,v) $\wedge$ TC(v,u)) (5)
$\forall$ s c: (A(s,c,MAX) $\Leftrightarrow$ ($\exists$ u v : (TC(u, v) $\wedge$ M(u,c) $\wedge$ P(s, v)))) (6)
$\forall$ s v c : $\neg$(P(s,v) $\wedge$ M(v,c) $\wedge$ A(s,c, MIN)) (7)
$\forall$ v v1>v c : $\neg$(TC(v, v1) $\wedge$ M(v1, c) $\wedge$ M(v, c)) (8)

Figure 4.5: MX-ASP axiomatization of binary CCS phylogeny re-construction

Translation of ASP to MX

We also used an MX axiomatization based on the best ASP encoding from [30] (denoted "A+" there). It differs from the previous two in that:

1. We have a type Species, distinct from Vertex.
2. Rather than identify the n species with the first n vertices, we construct a function P (represented as a binary relation) from species to vertices.
3. We construct a function M from vertices to characters. The mutation on edge (u,v) is the character c such that M(v,c) holds. In contrast, in our previous axiomatizations the mutation on edge (u,v) is implicit in the difference between the vectors labeling u and v.
4. The (root) vector $\vec{0}$ is left implicit, so a solution is a forest. A tree is obtained by adding an edge from $\vec{0}$ to the root of each forest component.

Figure 4.5 gives the axioms, which state:

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

Axioms 7 and 8 are redundant, but improve performance.

4.3 Evaluating Progress in Performance

Evaluating performance of solvers for NP-hard problems has many pitfalls, especially when there is no base-line provided by well-established benchmarks and solvers. Our goal is to have a clear measure of *progress* in performance. Direct comparison of run-times does not work here, because run-times for the methods we test vary by many orders of magnitude. All instances for which our best methods require a non-trivial amount of time go unsolved by our poorer methods in any reasonable amount of time (see Figure 4.9). A better measure is the number of instances that can be solved within reasonable time. For this, a collection of related instances of graduated difficulty is needed, but in practice this is often hard to arrange. For example, [30] obtained three real data sets: Two were too easy and the third was too hard. Randomly generated instances are easily graduated, but their use requires care (see, e.g., [39]), and may be irrelevant to practice.

Instances

Here, we produce a set of suitably graduated instances from the one challenging real data set we have for our problem. This is possible because, if we view a set of n species vectors of length m as an $n \times m$ matrix M, any sub-matrix of M is a valid set of data, as real (or not) as the full matrix. For illustration: {eye-color,hair-color} is as valid a set of characters as {eye-color,hair-color,handedness}. (Not every such matrix is scientifically interesting, of

course.) Our initial instance is a 36×63 matrix obtained from the experimentally obtained haplotype data of [29] (for Poecilia reticulata - guppies) as described in [30]. Following [30], we produced a set of instances from upper-left sub-matrices of size $k \times l$, for k, l multiples of 3. Thus, we view performance as a function of two natural instance parameters: number of species and number of characters. Unfortunately, the resulting instances were not nicely graduated, as most moderate-size instances were very easy for our methods. The problem was that most sub-matrices had all-zero columns and duplicate rows, which we solved as follows. Keeping the zero vector as species 1, we put all other species in reverse lexicographic order. Thus, the first row was all zero, but the second row had many ones. The set of instances produced from this initial matrix by the scheme described above satisfied our main criteria: the instances are smoothly graded in difficulty for our solvers, and they do not contain significant numbers of trivial or duplicate rows or columns.

Performance Measure

As an objective measure of progress, we require the solver to establish the optimal phylogeny size within a fixed time bound. As with any optimization problem, one may trade solution quality for solving time. In phylogenetic inference, users often do not care about optimality *per se*, because the cost metrics do not exactly correspond to their subjective notion of quality. But if optimality is not a precise measure of quality, surely being within some distance of optimal is not either, so relaxing the optimality requirement does not improve our measure. The requirement to solve to optimality would seem to better measure whether we are making progress in dealing with whatever it is that makes up the combinatorially hard aspect of our instances. For measuring progress toward being able to practically solve larger and harder instances than currently possible, we believe that establishing optimal solutions within a reasonable time cut-off is as good a measure as any we know of.

Evaluation

MXG does not have a built-in optimization facility, so for each optimization instance we solve a sequence of search instances. The first asks for a perfect phylogeny (with the same number of mutations as characters). Successive instances allow one more mutation. We run the solver on the sequence, stopping when a solution is found – which must be optimal – or when the cumulative run time reaches two hours. Sequential search is faster than

# of extra vertices	Total Time (seconds)
0-8	0.01
9	1.45
10	1.86
11	2.70
12	3.14
13	2.57
14	5.67
15	8
16	9.20
17	9.51
18	20.81
19	46.63
20	52.04
21	139.24
22 (optimal)	42.65
23	32.13
24	57.14

Table 4.1: Run times of MX-Depth+(PP) for 24 species with 51 characters

binary search because instances with too few mutations are typically easier than those with too many. Time for sequential search is dominated, almost without exception, by the two instances needed to establish optimality: the one producing an optimum solution and the one with one fewer mutations (Table 4.1 shows run times for our MX-Depth+(PP) axiomatization described in Section 4.4). Binary search is often dominated by the instances just beyond optimal, which sequential search never visits. This pattern of hardness also supports our argument in favor of using optimality in our measure of performance.

Tests were run on the same computers from Section 3.2.3 The software versions were:

- MXG 0.171 with MXC 0.5 for the SAT solver. (denoted MXG+MXC) Available from "http://www.cs.sfu.ca/research/groups/mxp";
- MXG 0.171 with minisat_v2s for the SAT solver (denoted MXG+minisat) MXG is available from "http://www.cs.sfu.ca/research/groups/mxp", and minisat is available from "http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/";
- clasp 1.0.4, a "native" ASP solver with clause learning.

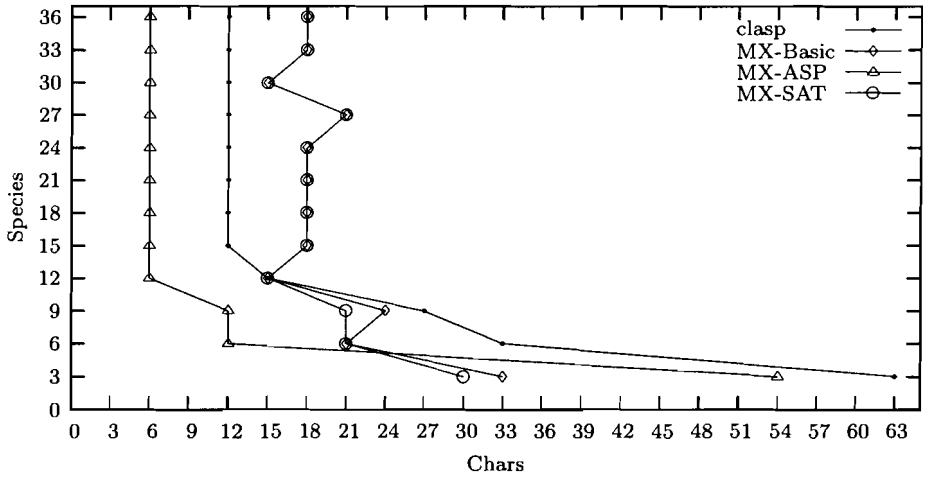

Figure 4.6: Frontier comparison of three MX axiomatizations and an ASP solution.

Available from "http://www.dbai.tuwien.ac.at/proj/dlv/";

- paup4b10-opt-linux-a, a widely-used phylogeny software package.

Executables for PAUP were downloaded from the solver web site, while MXG, MXC, clasp, and minisat_v2s were compiled with gcc version 3.3.4.

Figure 4.6 shows the "frontier" for MXG with the axiomatizations of Section 4.2, and for the ASP solver clasp using the A+ axiomatization of [30]. We plot a curve for each solving method. A point at (x, y) denotes that x is the largest number of characters for which the method succeeded in solving instances with y species within the two-hour cut-off. Instances left of or below a curve were solved; those above and to the right were not. The basic MX axiomatization is best, except with very few species. MXG performs relatively poorly with few species because it must construct the whole vector for each vertex, and thus with many characters has quite a bit of work to do, while the ASP axiomatizations do not. The no-cardinality axiomatization and minisat performed essentially the same as our basic MX axioms, except for being slightly weaker with few species. Our translation of the ASP axioms to MX performed poorly.

We conclude that our basic MX axiomatization, which is already substantially better than the best solution in [30], is a good starting point for further work.

4.4 Enhancing Performance

In this section, we report on two ways we refined our basic axiomatization that dramatically improved performance. Adding redundant axioms is a standard method in SAT and CSP encodings, and [30] reported significant speedups with this method. Symmetry-breaking axioms eliminate some - but not all - solutions among a set of symmetric solutions.

Axioms 13 of Figure 4.7 states that no extra vertex is a leaf. It is neither symmetry-breaking nor redundant, but has a similar flavor in that it removes only solutions that are not very interesting, and improves performance.

Computing Vertex Depth

In a binary CCS tree, each vector labeling a vertex at depth k has exactly k 1's. Thus, if K is the maximum number of 1's in any species vector, the tree has height at most K. We can add axioms requiring labels of vertices to respect this property. These are axioms seven through ten and thirteen of Figure 4.7, which state:

- Each vertex must be assigned a unique depth (Axiom 8), which must be one greater than that of its parent (Axiom 9).
- The depth of each vertex is the number of 1's in its label (Axiom 10).
- Only the root has depth 0 (Axiom 11);

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

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

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

Satisfying:
// Axioms 1-6 are the Basic MX Axioms of Figure 4.3

VtxDepth(Nodes, Depth)
SpcDepth(Nodes, Chars, Depth)

$$
\begin{aligned}
&\{\ \mathrm{SpcDepth(u,c,d)} \leftarrow \mathrm{c{=}MIN} \wedge \mathrm{d{=}MIN} \wedge \mathrm{s} = \mathrm{MIN} \wedge \mathrm{A(u,c,s)} && (7)\\
&\quad \mathrm{SpcDepth(u,c,d)} \leftarrow \mathrm{c{=}MIN} \wedge \mathrm{SUCC(MIN,\ d)} \wedge \mathrm{s} = \mathrm{MAX} \wedge \mathrm{A(u,c,s)}\\
&\quad \mathrm{SpcDepth(u,c,d)} \leftarrow \mathrm{SpcDepth(u,c1,\ d)} \wedge \mathrm{SUCC(c1,\ c)} \wedge \mathrm{A(u,c,s)} \wedge \mathrm{s{=}MIN}\\
&\quad \mathrm{SpcDepth(u,c,d)} \leftarrow \mathrm{SpcDepth(u,c1,\ d1)} \wedge \mathrm{SUCC(c1,\ c)} \wedge \mathrm{SUCC(d1,\ d)}\\
&\qquad\qquad \wedge \mathrm{A(u,c,s)} \wedge \mathrm{s{=}MAX}\\
&\}\\
&\forall\ \mathrm{u} : \mathrm{CARD(1;\ d;\ VtxDepth(u,\ d))} && (8)\\
&\forall\ \mathrm{u\ v\ d1\ d2} : ((\mathrm{Edge(u,v)} \wedge \mathrm{VtxDepth(u,\ d1)} \wedge \mathrm{VtxDepth(v,d2)}) \supset\\
&\qquad\qquad \mathrm{SUCC(d1,\ d2)}) && (9)\\
&\forall\ \mathrm{u} \leq \mathrm{NSpecies\ d:}\ (\mathrm{VtxDepth(u,d)} \Leftrightarrow \mathrm{SpcDepth(u,MAX,d)}) && (10)\\
&\forall\ \mathrm{u} > \mathrm{MIN} : \neg\ \mathrm{VtxDepth(u,MIN)} && (11)\\
&\forall\ \mathrm{u} > \mathrm{NSpecies\ v} > \mathrm{u\ d1\ d2} : ((\mathrm{VtxDepth(u,d1)} \wedge \mathrm{VtxDepth(v,d2)}) \supset \mathrm{d2} \geq \mathrm{d1}) && (12)\\
&\forall\ \mathrm{u} > \mathrm{NSpecies} : \exists\ \mathrm{v} : \mathrm{Edge(u,v)} && (13)
\end{aligned}
$$

Figure 4.7: MX-Depth (Axioms 1-10) and MX-Depth+ (Axioms 1-13) axiomatizations.

Symmetry Breaking

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

Instance Pre-processing

We found that instances (including the largest) often satisfied easily-checked properties that could be used to simplify them with a pre-processing step, which greatly improved performance. We recursively applied the following rules:

- Delete any all-zero column: The character does not mutate, so we need no node for it.
- Delete any column having exactly one 1: If c is 1 only in s, we construct a tree without c, then add $c = 0$ to every vector on the tree, adding one new edge and vertex where s appears.
- Delete any column having exactly one 0: The 0 occurs in the root zero vector. We construct a solution without c, and insert one new node beneath the root in the solution, setting $c = 1$ everywhere except the root.
- Delete any duplicate species.
- Delete s_2 for any pair s_1, s_2 of species such that:
  - $s_1 \subset s_2$, i.e., every character that is 1 for s_1 is also 1 for s_2;
  - $|s2 - s1| = k$, i.e., s_2 has k more 1's than s_1;
  - $\forall s_3 \notin \{s_1, s_2\}, |s_2 \backslash s_3| \geq k$.

  Solve the instance without s_2, then add it to a path of length k below s_1.

Performance with Refined Axioms and Pre-processing

Figure 4.8 (Upper) is a frontier plot showing performance improvements obtained with enhanced axiomatizations. For comparison, we included the curve MX-Basic, and in addition two new curves:

- MX-Depth: MX-Basic axioms extended with Axioms 7–10 of Figure 4.7;
- MX-Depth+: MX-Depth further extended with Axioms 11–13 of Figure 4.7;

Figure 4.8 (Lower) shows the further improvement of our axiomatizations aided by pre-processing. In addition to curves of figure 4.8 (Upper), we added three more curves for the same axiomatization and pre-processed instances.

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

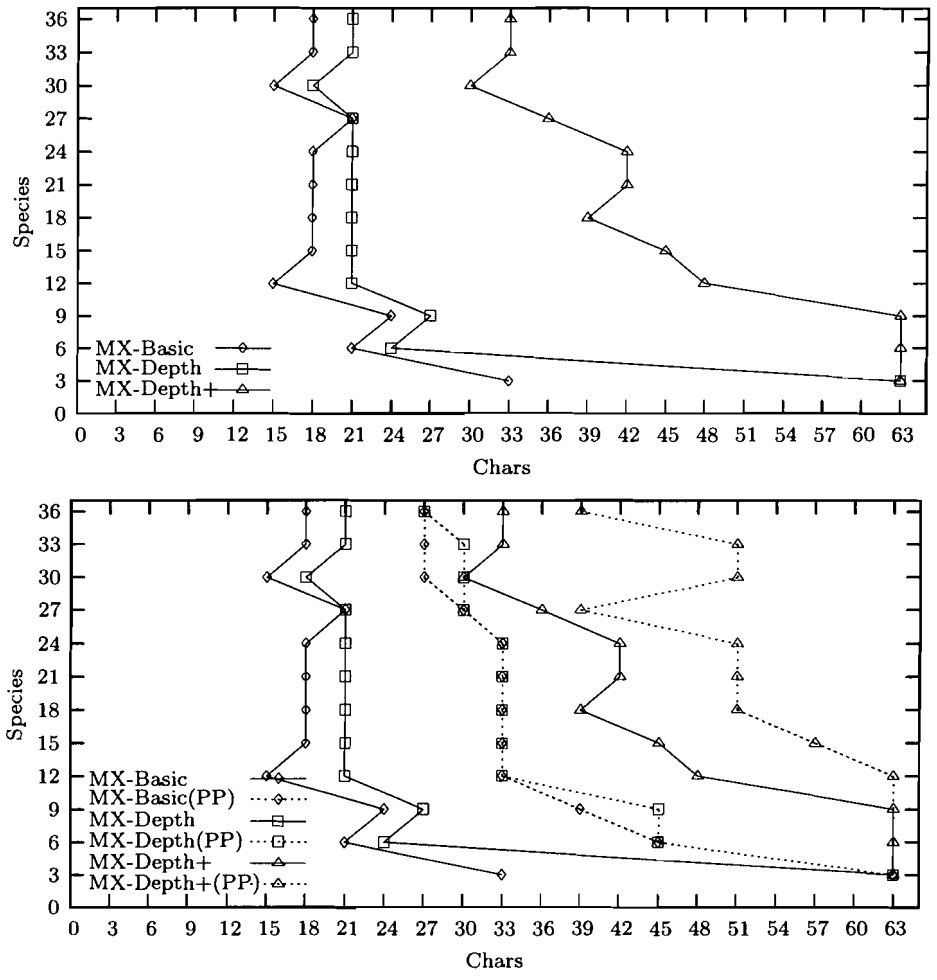

Figure 4.8: Frontier comparison of refined axiomatizations (Upper), and with pre-processing (Lower).

How Much Better: Frontier *vs.* Run-time

The frontier plots show that we have progressed in terms of our chosen measure, but do not show the (dramatic) corresponding changes in run-time. Figure 4.9 illustrates, showing run-times as a function of number of characters, with number of species fixed at 24. Analogous curves for fewer or more species are similar, except for very small numbers of species. The y (time) axis is log scale, so these run-times appear to be exponential in the number of characters. Notice that the curves have very different slopes, suggesting that the run-time

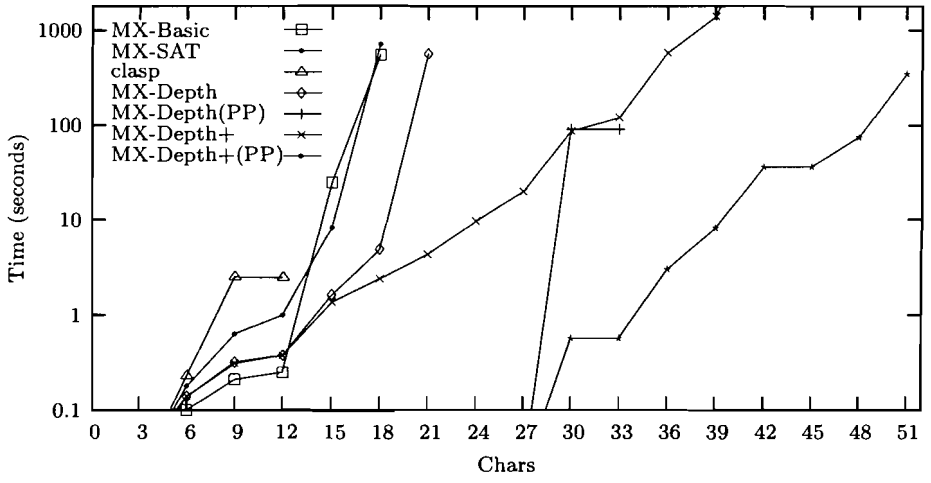

Figure 4.9: Run-times for instances with 24 species, as a function of number of characters.

curves for MX-Depth+ and MX-Depth+(PP) have much smaller exponents than the other solutions. We cannot really extrapolate the curves for the ASP or Basic MX solutions to compare run-times for large instances with the best methods, but unless the curves here are completely mis-leading the difference is certainly many orders of magnitude. Solving the hardest instances with those methods is completely infeasible in practice.

4.5 MXG *vs.* PAUP

The two most widely used phylogeny software packages, PHYLIP [22] and PAUP [47], both use two methods to carry out phylogenetic inference (for CCS and other models). One method is based on heuristic search, which cannot guarantee optimality, and one is based on branch-and-bound, which can. The branch-and-bound program for CCS in the PHYLIP package is called PENNY (after the second author of [28], where branch and bound was proposed for this task). In [30], the performance of PENNY was compared with the ASP-based solutions developed there. PENNY was unable to prove optimality of solutions for any instances with more than 18 species.

We compare the performance of our method against the branch and bound implementation in PAUP. (We might expect PAUP (which is not free) to be faster than PHYLIP (which is free), because it has had more development effort, and this seems to be the

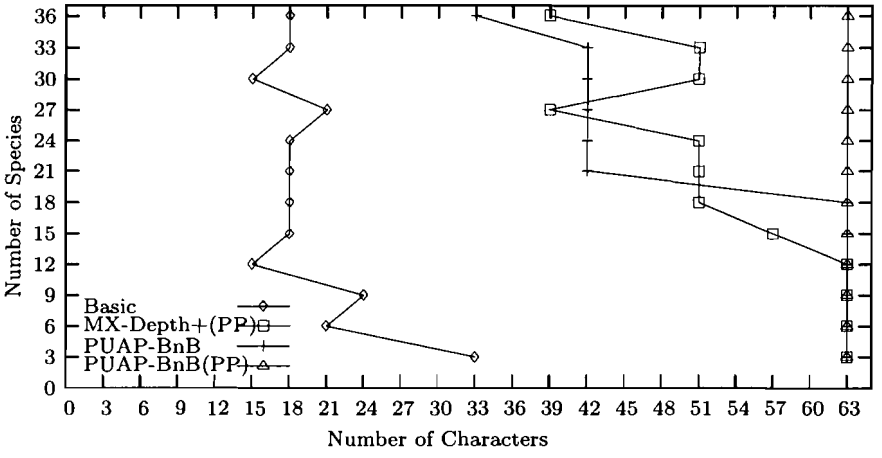

Figure 4.10: Frontier for MX Solutions and PAUP

case.) Figure 4.10 shows the frontier plots for the PAUP branch and bound implementation (PAUP-BnB), along with that for MX-Depth+(PP), and MX-Basic for comparison. Our MX-based solution is similar overall to PAUP, but comes closer to solving our largest – and presumably hardest – instances. For completeness, we also ran PAUP branch-and-bound on the instances produced by our pre-processing algorithm. Interestingly, PAUP performance improved, and with our pre-processing it solved all instances.

4.6 Discussion

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

magnitude, strongly suggest that our methods will be significant improvements by any other reasonable measure of performance.

MX *vs.* ASP

While our Model Expansion based solutions, using the grounder MXG with ground solver MXC, perform dramatically better than the ASP solution we evaluate, our results here do not justify a claim of superiority of MX methodology or solvers over those of ASP. The best ASP solvers are quite powerful, and alternate approaches to ASP axioms, combined with pre-processing of the sort we do, might yield effective ASP-based solutions. (A comparison of MXG with several ASP solvers appears in Chapter 3.)

PAUP Heuristics

An easy criticism of the work presented here is that the heuristic methods implemented in PAUP and PHYLIP often perform very well, and we have not compared our methods with those. In fact, the heuristic search method of PAUP finds optimum solutions for all of our instances well within our time limit. PAUP, of course, does not know if they are optimal or not, and neither would a PAUP user. But, if optimality *per se* is not of much value to a biologist, why would they care?

One reply is that declarative solutions are potentially very useful. For example, user's don't worry about optimality because they are interested in criteria that are not captured by the cost function. With standard tools they are limited in how they can address these other preferences. Good declarative tools would allow them to add specific constraints, say that certain species should not be in the same sub-tree, and find solutions satisfying these (see also [20]). Another reason good declarative techniques could pay off is that problems of interest are often variants of a few core problems, and in some cases these are much more complex than the simple problem we studied here. An example is the Galled Tree Network Haplotyping Problem [27]. An instance is genotype data, which consists of vectors of conflated pairs of haplotypes. The task is to infer a set of haplotype vectors from the genotype data for which a parsimonious galled tree network exists. A galled tree network is a significantly more complex phylogeny than our binary CCS trees. The task of inferring small sets of haplotypes from genotype data, without worrying about phylogenies, is itself NP-hard (although SAT solvers do well at this [34], so MXG should also). Implementing a

special-purpose program for this problem would be some effort, and finding simple heuristics which work reliably on large instances of such a problems seems unlikely. However, if we had effective declarative solutions for haplotype inference and construction of galled-tree networks, it would be easy to combine them and have a good start toward a solution for the larger problem.

Declarative Pre-Processing

A point that may be argued against the progress we claim is that pre-processing of the instances before passing to the declarative solver was important, but this step is not declarative. Indeed, pre-processing is important in tackling many problems, seemingly a stumbling block for declarative methods. We point out the technique we used in our MX-Depth axiomatization (Figure 4.7), where we wrote an inductive definition to compute a set, and then used certain elements of that set in other axioms. MXG computes the defined set directly, while grounding, so the ground solver does not see this part of the axiomatization. Essentially any poly-time preprocessing can be carried out using this technique (not necessarily by the current version of MXG). With suitably refined languages, some users could accomplish such pre-processing more conveniently with declarative descriptions than with procedural code.

Chapter 5

Related Work

There are several approaches known as "constraint modeling" which provide declarative languages for describing search problems. Examples include ESRA [23], Essence [24] and Zinc [12]. Their specification can be viewed as MX specification for a suitable logic although these languages are not explicitly logic based. Implementations of these languages solve either by translation to a high level programming language (such as Constraint Logic Programming languages) or by grounding for some "CSP solver", a solver for some generalization of SAT to non-boolean domains.

Answer Set Programming [35, 43] is based on the language of logic programming, with the stable model semantics [25]. There are many ASP ground solvers including clasp [26], Cmodels [33], smodels [49] and DLV [32]. clasp, Cmodels and smodels take the ground programs produced by the Lparse [48] grounder. DLV has its own built-in grounder. The language of Lparse extends normal logic programs with weight constraints (a generalization of cardinality constraints), and arithmetic. Instances are provided in the form of a set of ground atoms, which formally are part of the logic program. Separation of problem and instance descriptions is considered important [35] but maintained only as a convention which is not always followed in practice.

Highly symmetric combinatorial problems constitute a great challenge in AI. For that, symmetry breaking and redundant constraint rules are the subject of studies by many researcher in [52, 46, 4, 5]. In [52], they show how to specify several variants of Latin Squares and their related structures in propositional logic. They examined various specification techniques such as redundant constraints and isomorphism elimination and were able to solve numerous previously open problems using these techniques. The Social Golfer problem has

many symmetrical solutions, since 1) players inside groups can be exchanged; 2) groups inside weeks can be exchanged; 3) weeks can be exchanged; 4) players can be re-numbered. Symmetries (1) to (3) can be removed statically by ordering golfers inside groups, ordering groups of every week, and ordering weeks according to their smallest player numbers respectively. Handling symmetry (4) needs more advanced techniques. In [4, 5] part of symmetry (4) was removed by setting some of the values statically, fixing some players for some weeks, and narrowing the domain for other choices.

Kavanagh et al [30] reported answer set programming (ASP) based solutions to binary CCS. Their best solution established optimal trees for instances for which the phylogeny software package PENNY [22] could not. They could not solve their largest instance (which is identical to our largest instance), or even moderate-sized sub-sets. ASP solutions to some other phylogeny problems, which are not directly comparable, are reported in [7, 20, 51]. In [7], the problem studied is "large compatibility", where the goal is to find the maximum number of characters, for a given set of species, for which there is a perfect phylogeny. In contrast, we use "large parsimony", where we find a (perhaps not perfect) phylogeny for the input species with the minimum number of evolutionary changes. The task studied in [20] is to construct a "perfect phylogenetic network", from given phylogenetic trees (the "species" there are natural languages). The authors of [51] studied the "Maximum Quartet Consistency" problem, and evaluated an ASP solution on synthetic data. They try to find the phylogeny to each subset of four *taxa*, Quartet, of a given set of taxon S, then they try to infer an overall un-rooted phylogeny for the whole set S by relying on the Quartets phylogeny.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Declarative programming languages describe what it is to be computed, not how to compute it. A declarative program specifies the relationship between an instance and its solutions. Mitchell and Ternovska proposed a new declarative programming framework, the Model Expansion Framework, as a formal basis for solving search problems [40]. MXG [42] is a FO(ID+Card)-MX solver for solving NP search problems. Specification of problem and description of instance structure are given separately to MXG. It applies a uniform, polytime reduction to SAT for any problem in NP specified by a FO(ID+Card) formula. Modern SAT solvers are very effective in solving many search problems, but employing them generally requires designing and implementing a reduction to SAT. MXG may be viewed as a high-level front-end for SAT solvers, allowing them to be much more easily exploited.

We explored whether MXG is effective, in terms of performance and convenience of modeling, by presenting specifications of a number of benchmark problems for MXG. In refining these specifications we showed how several built-in features of MXG can be used to improve the performance. We also presented and applied some performance improvement techniques, such as symmetry breaking and adding redundant axioms, that are used in other declarative approaches to solve NP search problems. To show feasibility of the MXG approach, we compared MXG with MidL [37], another FO(ID)-MX solver, and with three high performance ASP solvers; clasp [26], DLV [32] and smodels [49]. Our experiments on the problems studied show that MXG is competitive with these ASP solvers.

To demonstrate that MX-based tools, and particularly MXG, can be effective on more

realistic domains than has previously been shown, we studied a challenging problem of phylogenetic inference [38] on a real data set. We presented several specifications for this problem, applying a variety of techniques for obtaining good performance. We described a method that is faster, by many orders of magnitude, than the only other declarative solution [30] of which we are aware. Our best solution combines instance pre-processing, redundant axioms, and symmetry breaking axioms. We also showed that our instance pre-processing method improves the performance of PAUP[47] branch and bound.

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

6.2 Future Work

In Chapter 3 we provided techniques to improve the performance of our solver. These techniques can be used as a basis for building a tool that pre-processes problem specifications and re-writes axioms. It should recognize the cases where axioms can be rewritten into a form that results in better performance, and cases when it is beneficial to add extra constraints. For example, the use of bounded quantifiers and orderings as described in Section 3.1.6 can be potentially automated. Also a library of symmetries that are introduced with certain way of specifying problems can be constructed and be used for breaking symmetries as in [8]. Such a pre-processing tool can also be used to make a good choice of ground solver for a problem, by analyzing its specification.

Another direction for future work is the bio-informatics problems. Many phylogeny problems are variants of, or combinations of, a few basic problems. For example, in the *Galled Tree Network Haplotyping Problem* [27] an instance consists of a genotype data, which is of vectors of conflated pairs of haplotypes, and the task is to construct a parsimonious galled-tree network. This problem is combination of two problems: the *Pure Parsimony Haplotyping Problem* [31] and a generalized phylogenetic inference. In the *Pure Parsimony Haplotyping Problem*, the input is a genotype data and the task is to find the minimum set of haplotypes. The Galled Tree Network Haplotyping problem is a more complex phylogeny problem than our binary CCS. If we can provide effective declarative solutions for haplotype inference and construction of galled-tree networks, it would be easy to combine them and have a good start for a solution for the larger problem. More generally, we would like to

develop tools supporting modular solutions, where we have a library of solutions for different problems and we can combine these solutions to find a solution for a larger problem.

Bibliography

[1] ASP Solver Competition, http://asparagus.cs.uni-potsdam.de/contest/.

[2] Asparagus Repository, http://asparagus.cs.uni-potsdam.de/.

[3] http://mat.gsia.cmu.edu/COLOR/instances.

[4] Francisco Azevedo and Hau Nguyen Van. Extra constraints for social golfers problem. In *Proceedings of LPAR*, 2006.

[5] Francisco Azevedo and Hau Nguyen Van. Symmetry breaking and extra constraints for social golfers problem. In *Proceedings of International Symmetry Conference*, 2007.

[6] David R. Bregman and David G. Mitchell. The SAT solver MXC, version 0.5, 2007. Solver Description for the 2007 Sat Solver Competition.

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

[8] Marco Cadoli and Toni Mancini. Automated reformulation of specifications by safe delay of constraints. *Artificial Intelligence*, 170(8):779–801, 2006.

[9] Keith L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, 1977.

[10] Stephen A. Cook. The complexity of theorem proving procedures. In *Annual ACM Symp. on Theory of Computing*, pages 151–158, 1971.

[11] 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.

[12] Maria J. García de la Banda, Kim Marriott, Reza Rafeh, and Mark Wallace. The modelling language Zinc. In *CP*, pages 700–705, 2006.

[13] M. Denecker and E. Ternovska. A logic of non-monotone inductive definitions and its modularity properties. *Lecture Notes in Computer Science*, 2923:47–60, 2004.

[14] M. Denecker and E. Ternovska. A logic of non-monotone inductive denitions. *ACM Transactions on Computational Logic*, 2006.

[15] Marc Denecker. Extending classical logic with inductive definitions. *Lecture Notes in Computer Science*, 1861:703–717, 2000.

[16] J. Denes and A. Keedwell. *Latin squares and their applications.* Akademiai Kiado, Budapest, and English Universities Press, London, 1974.

[17] 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.

[18] N. Een and N. Sorensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[19] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, 1997.

[20] 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.

[21] Ronlad Fagin. Generalized first-order spectra and polynomial-time recognizable sets. *Complexity of Computation*, 7:43–73, 1974.

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

[23] Pierre Flener, Justin Pearson, and Magnus Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In *CP*, page 971, 2003.

[24] Alan M. Frisch, Matthew Grum, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. The design of ESSENCE: A constraint language for specifying combinatorial problems. In *IJCAI*, pages 80–87, 2007.

[25] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[26] M. Gesber, B. Kaufmann, A.Neumann, and T. Schaub. clasp: A conflict-driven Answer Set solver. In *Proc. of Ninth Int'l Conference on Logic Programming and Nonmonotonic Reasoning(LPNMR'07)*, volume 4483 of *LNAI*, pages 260–265, 2007.

[27] Arvind Gupta, Ján Manuch, Xiaohong Zhao, and Ladislav Stacho. Characterization of the existence of Galled-tree networks. *J. Bioinformatics and Computational Biology*, 4(6):1309–1328, 2006.

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

[29] M. Hoffmann, N. Tripathi, S. R. Henz, A. K. Lindholm, D. Weigel, F. Breden, and C. Dreyer. Opsin gene duplication and diversification in the guppy, a model for sexual selection. *Proc. of the Royal Society of London Series B*, 274:33–42, 2007.

[30] Jonathan Kavanagh, David G. Mitchell, Eugenia Ternovska, Ján Manuch, Xiaohong Zhao, and Arvind Gupta. Constructing Camin-Sokal phylogenies via Answer Set programming. In *Proc. of LPAR*, pages 452–466, 2006.

[31] Giuseppe Lancia, Maria Cristina Pinotti, and Romeo Rizzi. Haplotyping populations by pure parsimony: Complexity of exact and approximation algorithms. *INFORMS Journal on Computing*, 16(4):348–359, 2004.

[32] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM TOCL*, 7(3):499–562, 2006.

[33] Yuliya Lierler and Marco Maratea. Cmodels-2: SAT-based Answer Set solver enhanced to non-tight programs. In *Logic Programming and Nonmonotonic Reasoning, 7th International Conference*, volume 2923 of *LNCS*, pages 346–350, 2004.

[34] I. Lynce and Marques Silva J. Efficient Haplotype inference with boolean satisfiability. *AAAI*, July 2006.

[35] W. Marek and M. Truszczy. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective, K.R. Apt, V.W. Marek, M. Truszczynski, D.S. Warren, Eds, Springer-Verlag*, pages 375–398, 1999.

[36] Maarten Mariën, Rudradeb Mitra, Marc Denecker, and Maurice Bruynooghe. Satisfiability checking for PC(ID). In *Proc. of LPAR*, pages 565–579, 2005.

[37] Maarten Mariën, Johan Wittocx, and Marc Denecker. The IDP framework for declartive problem solving. In *Search and Logic: Answer Set Programming and SAT*, pages 19–34, 2006.

[38] David G. Mitchell, Faraz Hach, and Raheleh Mohebali. Faster phylogenetic inference with MXG. In *LPAR*, volume 4790 of *Lecture Notes in Computer Science*, pages 423–437. Springer, 2007.

[39] David G. Mitchell and Hector J. Levesque. Some pitfalls for experimenters with random SAT. *Artificial Intelligence*, 81(1,2), Mar 1996. *Special Issue* – Frontiers in Problem Solving: Phase Transitions and Complexity.

[40] David G. Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In *Proc. of the 20th National Conf. on Artif. Intell. (AAAI)*, pages 430–435, 2005.

[41] David G. Mitchell, Eugenia Ternovska, Faraz Hach, and Raheleh Mohebali. Model Expansion as a framework for modelling and solving search problems. Technical Report TR 2006-24, School of Computing Science, Simon Fraser University, December 2006.

[42] Raheleh Mohebali, Faraz Hach, and David G. Mitchell. MXG: A model expansion grounder and solver. Accepted as an LPAR'07 short paper, 2007.

[43] I. Niemela. Logic programs with stable model semantics as a constraint programming paradigm. In I. Niemela and T. Schaub, editors, *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, pages 72–79, 1998.

[44] 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.

[45] Murray Patterson, Yongmei Liu, Eugenia Ternovska, and Arvind Gupta. Grounding for model expansion in k-guarded formulas with inductive definitions. In *IJCAI*, pages 161–166, 2007.

[46] Barbara M. Smith. Reducing symmetry in a combinatorial design problem. In *Proceedings of CPAIOR*, pages 351–359, 2001.

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

[48] Tommi Syrjänen. Lparse 1.0 user's manual.

[49] Tommi Syrjänen and Ilkka Niemelä. The Smodels system. In *6th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 434–438, 2001.

[50] G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 466–483. Springer, Berlin, Heidelberg, 1983.

[51] Gang Wu, Jia-Huai You, and Guohui Lin. Quartet-based Phylogeny reconstruction with Answer Set programming. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 4(1):139–152, 2007.

[52] Hantao Zhang. Specifying latin square problems in propositional logic. Automated Reasoning and Its Applications: Essays in Honor of Larray Wos, Chapter 6, MIT Press, 1997.