The Dissertation Committee for Esra Erdem
certifies that this is the approved version of the following dissertation:

# Theory and Applications of Answer Set Programming

Committee:

Vladimir Lifschitz, Supervisor

Chitta Baral

J. Strother Moore

Tandy Warnow

Martin D. F. Wong

# Theory and Applications of Answer Set Programming

by

**Esra Erdem, B.S., M.S.**

**Dissertation**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Doctor of Philosophy**

**The University of Texas at Austin**

August 2002

This dissertation is dedicated to my parents Huriye and Orhan Erdem, and my sisters, Yelda and Seda Erdem, for their continuous love and support.

# Acknowledgments

I am indebted to Vladimir Lifschitz for providing me with the opportunity of working with him. This dissertation would not have been possible without his invaluable advice and continuing encouragement.

I have benefited from many delightful discussions with Michael Gelfond, Nicola Leone, Norman McCain, Illka Niemelä and Hudson Turner. I am thankful to all of them.

It has been a pleasure to work with Martin Wong on the wire routing problems, and Luay Nakhleh and Donald Ringe on the phylogeny reconstruction problems. I am grateful to them, and also Tandy Warnow, for patiently discussing these interesting problems with me, and for their encouragement.

I am thankful to Chitta Baral, J Strother Moore, Tandy Warnow, Martin Wong for agreeing to be on my committee, and for all their suggestions.

I have enjoyed many useful discussions with other friends, teachers and collegaues. Thanks to all of them, including Eyal Amir, Yuliya Babovich, Jonathan Campbell, Selim Erdoğan, Wolfgang Faber, Paolo Ferraris, Enrico Giunchiglia, Joohyung Lee, Sheila McIlraith, Bernard Moret, Monica

Nogueira, Gerald Pfeifer, Emilio Remolina, Jun Sawada, Tran Son, Tommi Syrjänen, Armando Tacchella, Li-San Wang, Stacia Wyman, and many members of Texas Action Group.

The Scientific and Technical Research Council of Turkey (TÜBİTAK) has provided me with a NATO Science fellowship, which partially supported this work, and I am thankful for this.

I am grateful to many friends, including Emery Berger, Gülşen Demiröz and Sinan Erdoğan, for their support and sweet encouragement.

Finally, I would like to thank my parents Huriye and Orhan Erdem, and my sisters, Yelda and Seda Erdem, for their continuous love and support.

Esra Erdem

*The University of Texas at Austin*

*August 2002*

# Theory and Applications of Answer Set Programming

Publication No. ______________

Esra Erdem, Ph.D.
The University of Texas at Austin, 2002

Supervisor: Vladimir Lifschitz

Answer set programming (ASP) is a new form of declarative logic programming. ASP interprets a logic program as a constraint on sets of literals, just as a propositional formula can be viewed as a constraint on assignments of truth values to atoms. The concept of an answer set was originally proposed as a semantics of negation as failure in Prolog. Instead of traditional Prolog systems, ASP uses answer set solvers. The input of Prolog consists of a logic program and a query, and Prolog computes answer substitutions; the input of an answer set solver is a logic program, and the solver computes the program's answer sets. The idea of ASP is to represent a given computational problem as a logic program whose answer sets correspond to solutions, and to use an answer set solver to find an answer set.

We have investigated the application of ASP to several combinatorial search problems, including planning, wire routing, and phylogeny reconstruction. Planning is the problem of finding a sequence of actions that leads to a given goal. Wire routing is the problem of determining the physical locations of all wires interconnecting the circuit components on a chip. Phylogeny reconstruction is the problem of constructing and labeling an evolutionary tree for a set of taxa (taxonomic units), which describes the evolution of the taxa in that set from their most recent common ancestor. In our work on phylogeny reconstruction, we have generated several conjectures about the evolutionary history of Indo-European languages.

The work on the use of ASP for planning has led us to the investigation of some theoretical questions related to answer sets. One is the problem of equivalent transformations of logic programs: under what conditions can we replace a program by an equivalent program that can be processed by an answer set solver more efficiently? Another problem is related to completion—a process that can translate a logic program into a set of formulas of classical logic. In some cases, the interpretations satisfying the completion of a program are also the answer sets for that program. In such cases, we can use propositional solvers—systems that compute a model of a given set of clauses—to find the program's answer sets. For some problems, propositional solvers are more efficient than answer set solvers. Therefore, we have investigated under what conditions we can use propositional solvers to find the program's answer sets.

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

Answer set programming is a new form of declarative programming [Marek and Truszczyński, 1999], [Niemelä, 1999], [Lifschitz, 2002]. It is based on the "answer set semantics" [Gelfond and Lifschitz, 1991] (called "stable model semantics" in [Gelfond and Lifschitz, 1988]). It differs both from traditional logic programming and from constraint logic programming [Jaffar *et al.*, 1983] in that it represents solutions to a problem by "answer sets" rather than "answer substitutions." Instead of PROLOG, it uses *answer set solvers*—software systems capable of computing answer sets, such as CCALC [McCain and Turner, 1998], DCS [East and Truszczyński, 2000], DERES [Cholewiński *et al.*, 1996], DLV [Eiter *et al.*, 1997], and SMODELS [Niemelä and Simons, 1996], [Simons *et al.*, 2002]. The idea of answer set programming is to represent a given computational problem as a logic program whose answer sets correspond to solutions, and to use an answer set solver to find an answer set for this program.

Answer set programming is a constraint programming paradigm in the sense that it interprets a program as a constraint on sets of literals, just as a set of propositional formulas can be viewed as a constraint on an assignment of truth values to propositional symbols.

We have investigated the possibility of using answer set programming for solving combinatorial search problems of several kinds, including the following:

- Planning. In a planning problem, we want to find a plan—a sequence of actions that leads to a given goal.

- Wire routing. This is the problem of determining the physical locations of all wires interconnecting the circuit components on a chip. Since the wires cannot intersect with each other, they are competing for limited spaces, thus making routing a difficult combinatorial optimization problem.

- Phylogeny reconstruction. This is the problem of constructing and labeling an evolutionary tree for a set of taxa, which describes the evolution of the taxa in that set from their most recent common ancestor.

We are interested in fully adequate declarative representations of problems in these areas, their relations to logic programming, and algorithms for solving these problems. Our work on these applications led to the investigation of some theoretical problems related to answer sets:

- Equivalent transformations. Under what conditions can we replace a program by another program that has the same answer sets but can be processed by an answer set solver more efficiently?

- Completion. Clark [1978] defined a "completion" procedure that translates a logic program into a set of formulas of classical logic. In some cases, the interpretations satisfying the completion of a program are also the answer sets for that program [Fages, 1994]. We have investigated under what conditions this is the case so that we can use *propositional solvers*—systems that can compute a model of a propositional theory given as a set of clauses, such as CHAFF [Moskewicz *et al.*, 2001], SATO [Zhang, 1997] or RELSAT [Bayardo and Schrag, 1997], instead of answer set solvers, to find the program's answer sets.

In the following, we will describe the earlier work in logic programming which set the stage for answer set programming, the answer set semantics, answer set solvers, and computing answer sets for a program using propositional solvers (Chapter 2). Next, we summarize our initial experiments on generating plans using answer set solvers and using propositional solvers, and describe in what ways these experiments motivated our theoretical work (Chapter 3).

The theoretical work completed so far is presented in the next three chapters. We state the theorem showing that the transformations of logic programs we use in our experiments preserve the answer sets for a program (Chapter 4). We also justify our use of propositional solvers instead of answer set solvers by showing that the completion semantics and the answer set semantics are equivalent for the programs we experiment with (Chapter 5). In particular, we show that, under some conditions, these two semantics are equivalent for programs containing the recursive definition of the transitive closure of one of its predicates (Chapter 6). Proofs are presented in Chapter 9.

Then we present applications of answer set programming to the wire routing problems (Chapter 7) and to the phylogeny reconstruction problems (Chapter 8).

# Chapter 2

# Origins of Answer Set Programming

Answer set programming [Marek and Truszczyński, 1999], [Niemelä, 1999], [Lifschitz, 2002] is a new form of declarative programming. It differs from traditional logic programming in that it represents solutions to a problem by answer sets rather than answer substitutions. Instead of Prolog, it uses *answer set solvers*—software systems capable of computing answer sets.

In the following, we will describe the earlier work in logic programming which set the stage for answer set programming, and then will describe the semantics that answer set programming is based on. (For more information about the development of logic programming, see the surveys [Shepherdson, 1988], [Apt and Bol, 1994], [Baral and Gelfond, 1994], [Dix, 1995], [Brewka and Dix, 2001], [Dix *et al.*, 2001].) After that, we will describe the systems we can use to find the answer sets for a program. In particular, we will describe the answer set solvers DLV [Eiter *et al.*, 1997] and SMODELS [Niemelä and Simons, 1996], and discuss the possibility of using a propositional solver, such as CHAFF [Moskewicz *et al.*, 2001], SATO [Zhang, 1997] or RELSAT [Bayardo and Schrag, 1997] instead of an answer set solver, to find the answer sets for a program. Then, we will describe the Causal Calculator (CCALC) [McCain and Turner, 1998], a system which turns a logic program into a propositional theory and uses a propositional solver to find the answer sets for that program.

## 2.1 Setting the Stage

### 2.1.1 Logic-Based Languages

The use of logic-based languages for representing declarative knowledge was proposed by McCarthy [1959]. When a body of knowledge is presented in logic, that knowledge can be used via automated theorem proving.

The work in automated theorem proving, in particular the introduction of the *resolution principle* by Robinson [1965] and its refinements, i.e., *linear resolution* [Loveland, 1970] and *SL resolution* [Kowalski and Kuehner, 1971], to prove theorems in first-order logic led to the proposal of the concept of logic programming by Kowalski [1974], and to the first implementation of the programming language PROLOG by Colmerauer and his students [Roussel, 1975].[1] Both Kowalski and Colmerauer noted that a subset of first-order logic

[1] See [Colmerauer and Roussel, 1996] for the history of PROLOG.

was adequate for a programming language. In this subset of first-order logic, logic programs consist of rules of the form

$$A_0 \leftarrow A_1, \ldots, A_n \tag{2.1}$$

for some $n \geq 0$. The head $A_0$ of rule (2.1) is an atom; the body $A_1, \ldots, A_n$ is a list of atoms. Such programs are called *positive programs*. Van Emden and Kowalski [1976] showed that the meaning of positive programs can be described in terms of fixpoints of an operator $T$ applied to sets of atoms. Hill [1974] refined SL resolution for positive programs and, Apt and van Emden [1982] renamed this proof procedure as *SLD resolution*. Later implementations of PROLOG were based on SLD resolution. SLD resolution is sound and complete for positive programs.

A negation of an atom is never a logical consequence of a positive program. Getting negative information from a positive program can be justified, however, by adopting the "closed world assumption" (CWA) [Reiter, 1980]. Reiter's CWA is based on the idea that a program contains all the positive information about the objects in its domain and that any ground atom[2] that is not implied by the program is false.

Using CWA can be viewed as a simple form of nonmonotonic reasoning. Other forms of nonmonotonic reasoning emerged around the same time out of attempts to capture the essential aspects of commonsense reasoning. One of the main motivations came from reasoning about actions. McCarthy and

[2] A *ground atom* is an atom without variables.

Hayes [1969] proposed the *situation calculus* as a means to formalize change in logic. It soon turned out that the problem was not so much to represent what changes but to represent what does not change when an action occurs. This problem is called the "frame problem" [McCarthy and Hayes, 1969]. McCarthy proposed to handle the frame problem by a default rule:

> If a property $P$ holds at situation $S$ then $P$ normally also holds in the situation obtained by performing an action $A$ in $S$.

This rule is called the "commonsense law of inertia". Many nonmonotonic logics have been introduced to deal with the frame problem, such as *circumscription* [McCarthy, 1980], *default logic* [Reiter, 1980], and *autoepistemic logic* [Moore, 1985]. In logic programming, default rules are represented using a nonmonotonic negation, proposed by Clark [1978] and called *negation as failure*.

### 2.1.2 Negation as Failure

Clark's theory of negation as failure is based on program completion. He considered rules whose bodies may consist not only of atoms, as (2.1), but also of atoms preceded by negation as failure.

Clark's program completion is based on the idea that, in a program, the bodies of rules with the head *Head* provide not only a sufficient condition for *Head*, but also a necessary condition.

The completion semantics is defined for finite programs only. (See Section 3.4 for the definition of completion.) In [Apt *et al.*, 1988], the authors introduced a counterpart of the completion semantics, called "supportedness", for possibly infinite programs, and proved an important theorem saying that, for a finite program $\Pi$ and any set $X$ of atoms, $X$ satisfies the completion of a program iff $X$ is supported by $\Pi$. (See Section 5.1 for the definition of supportedness.)

In [Clark, 1978], a query evaluation procedure, called *SLDNF resolution*, was also introduced. In this procedure, the query *not* $Q$ succeeds in a program $\Pi$ if the query $Q$ with respect to $\Pi$ fails on every evaluation based on SLD resolution. SLDNF resolution is used in most of the recent implementations of PROLOG. It is sound with respect to program completion but not complete.

Unfortunately, the completion semantics, even applied to a positive program is sometimes unintuitive and not equivalent to the usual meaning of the programs as a set of formulas of first-order logic. For instance, adding the rule $p \leftarrow p$ to a program can change its meaning under the completion semantics, although this rule is tautological. Examples like this prompted Apt, Blair and Walker [1988] to look for a different explanation of the meaning of negation as failure.

### 2.1.3 Stratified Programs

In 1988, Apt, Blair and Walker studied a class of programs restricted syntactically in a way. These programs are called the "stratified" programs. They introduced a new semantics for stratified programs via an *iterated fixpoint* *construction*. Van Gelder [1988] independently developed the concept of stratification, and introduced the *tight tree semantics* for such programs.

Lifschitz [1988] studied stratified programs using the concept of *prioritized circumscription* [McCarthy, 1986]. Przymusinski [1988] extended the notion of stratification to programs that allow disjunction in heads of rules, and defined "perfect models" for such programs.

Although the work on stratified programs led to a useful semantics for this class of programs, there are programs of interest that are not stratified. For instance, consider the program, similar to (5) of [Gelfond and Lifschitz, 1988],

$$\begin{aligned} &move(a,b) \leftarrow \\ &move(b,c) \leftarrow \\ &winning(x) \leftarrow move(x,y), not\ winning(y). \end{aligned} \tag{2.2}$$

This program describes a two-player game where a position $x$ is winning for a player if there is a legal move from $x$ to $y$ and $y$ is not winning for the other player. This program is not stratified. How can we give a meaning to such programs and avoid the defects of the completion semantics mentioned above? This question led to several semantics including the "well-founded semantics" and the "answer set semantics". The latter became the basis for answer set programming that will be studied in this dissertation.

### 2.1.4 The Well-Founded Semantics

In 1988, Van Gelder, Ross, and Schlipf defined the *well-founded semantics* for programs with negation as failure. Well-founded semantics defines a three-valued interpretation of a program [Van Gelder *et al.*, 1991]: each atom is assigned one of the values *true*, *false*, *unknown*. As in [Lifschitz, 1996], we say that an atom is *well-founded* (resp. *unfounded*) relative to a program if it is mapped to *true* (resp. *false*) in the three-valued interpretation of that program. For instance, for program (2.2), the set of well-founded atoms is $\{move(a,b), move(b,c), winning(b)\}$, and the rest of the atoms are unfounded. In some cases, some atoms may be neither well-founded nor unfounded. For instance, consider the program

$$\begin{aligned} p &\leftarrow not\ q \\ q &\leftarrow not\ p. \end{aligned} \tag{2.3}$$

Here none of the atoms is well-founded or unfounded: both $p$ and $q$ get the truth value *unknown*.

SLDNF resolution is sound with respect to the well-founded semantics, but it is not complete. This led to the development of some logic programming systems, such as XSB,[3] based on the well-founded semantics. The proof procedure used in this system is *SLG resolution* [Chen *et al.*, 1993], which is SLDNF resolution where the "cycles" as in $p \leftarrow p$ are detected.

[3] `http://xsb.sourceforge.net/` .

### 2.1.5 The Answer Set Semantics

In 1988, Gelfond and Lifschitz defined the *answer semantics* (also called the stable model semantics) for programs with negation as failure. They obtain the *reduct* of a program relative to a set $X$ of atoms, denoted $\Pi^X$, by deleting the rules in $\Pi$ that contain the expression *not* $A$ in their bodies where $A \in X$, and by removing the remaining *not* $A$ in $\Pi$. For instance, the reduct of the program

$$\begin{aligned} &p \leftarrow not\ q \\ &q \leftarrow not\ p \\ &r \leftarrow p \\ &r \leftarrow q \end{aligned} \tag{2.4}$$

relative to $\{p, r\}$ is

$$\begin{aligned} &p \leftarrow \\ &r \leftarrow p \\ &r \leftarrow q. \end{aligned} \tag{2.5}$$

A set $X$ of atoms is an *answer set* for $\Pi$ if it is the set of consequences of the reduct $\Pi^X$. For instance, the set of consequences of the reduct (2.5) is $\{p, r\}$. Then, the set $\{p, r\}$ is an answer set for (2.4). Similarly, the set $\{q, r\}$ is an answer set for (2.4).

The definition of an answer set was based on the study of the relation between logic programming and autoepistemic logic in [Gelfond, 1987]. A similar idea was proposed independently in [Bidoit and Froidevaux, 1987].

The well-founded semantics is weaker than the answer set semantics in the sense that any answer set for a program $\Pi$ contains all the atoms that are well-founded relative to $\Pi$. However, an atom may belong to all answer sets but may not be well-founded. For instance, none of the atoms is well-founded or unfounded relative to (2.4), although the atom $r$ belongs to both answer sets $\{p, r\}$, $\{q, r\}$.

The answer set semantics belongs to a higher level of computational complexity than the well-founded semantics. Thus the answer set semantics is more expressive. For instance, the propositional satisfiability can be expressed in terms of the answer set semantics but not in terms of the well-founded semantics. Note that both the completion semantics and the answer set semantics are in the same level of the complexity hierarchy. The relationship between these two semantics was first investigated by Fages [1994]. Fages showed that, for the programs that are now called "tight", these two semantics are equivalent. Fages' theorem, its generalizations, and applications to answer set programming are discussed in Section 3.4.2 and in Chapter 5.

As with the well-founded semantics, SLDNF resolution is sound with respect to the answer set semantics, but it is not complete. This led to the development of some systems, called answer set solvers, such as DLV and SMODELS, based on the answer set semantics. These systems are the main computational tools used in answer set programming. They will be described in Section 2.3.

## 2.2 Extending the Syntax of Logic Programs

The syntax of logic programs has been extended in several ways resulting in more expressive programs, and the answer set semantics has been extended to these more general programs. There are four main extensions that played an important role in answer set programming.

**Classical negation.** Sometimes the use of negation as failure leads to undesirable results that can be eliminated by substituting classical negation for it. Consider the following example by John McCarthy as appears in [Gelfond and Lifschitz, 1991]. A school bus may cross railroad tracks under the condition that there is no approaching train. We may try to express this by the rule

$$cross \leftarrow not\ train$$

but this rule is too strong: according to this rule, even if we do not have information about *train*, the bus crosses the tracks. Just because the information about an approaching train may not be available, for instance the driver's vision may be blocked, we do not want the bus to cross the tracks. We need a stronger negation, such as classical negation, in this case. Consider the rule

$$cross \leftarrow \neg train$$

instead. Here, the bus crosses the tracks only if $\neg train$ is known.

Examples like this led to the introduction of classical negation into logic programming. Gelfond and Lifschitz [1990, 1991] introduced the classical negation symbol $\neg$ into the syntax of logic programs, and extended the answer set semantics to handle such programs. A similar idea was independently developed by Pearce and Wagner [1990].

Answer sets for a program with classical negation consist of literals (propositional atoms possibly preceded by classical negation).

Computationally, adding classical negation to the language of logic programs is not an essential extension. In [Gelfond and Lifschitz, 1991], the authors suggest that, instead of introducing classical negation explicitly, an atom preceded by classical negation, such as $\neg p$, can be replaced by a new atom $np$. Consider a program $\Pi$ with classical negation, and the program $\Pi'$ obtained from $\Pi$ by replacing every expression $\neg A$ by the expression $nA$. It is shown in [Gelfond and Lifschitz, 1991] that, for a consistent set $X \subset Lit$ and the set $X'$ obtained from $X$ by replacing every literal $\neg A$ by $nA$, $X$ is an answer set for $\Pi$ iff $X'$ is an answer set for $\Pi'$.

**Disjunction in the heads of rules.** Gelfond and Lifschitz defined the answer set semantics for the programs that may contain disjunction ; in the heads of the rules.[4] Consider, for instance, the following problem from [Gelfond and Lifschitz, 1991]. Jack is employed by Stanford University or by SRI International; an employed individual has an adequate income. This problem

[4]In [Gelfond and Lifschitz, 1990] and [Gelfond and Lifschitz, 1991], the authors use the symbol | to denote disjunction.

was formalized by the program

$$employed(jack, stanford); employed(jack, sri) \leftarrow$$

$$adequate\_income(x) \leftarrow employed(x)$$

whose answer sets are

$$\{employed(jack, stanford), adequate\_income(jack)\}$$

and

$$\{employed(jack, sri), adequate\_income(jack)\}.$$

Here the disjunction ; is semantically different from the disjunction $\vee$ of the classical logic. Consider, for instance, the program

$$p; \neg p \leftarrow \tag{2.6}$$

that has two answer sets $\{p\}$ and $\{\neg p\}$. Adding this rule to a program may change the set of answer sets, unlike adding the tautology $p \vee \neg p$ to a set of formulas.

In particular, the number of literals in the head of a rule can be 0. A rule with the empty head is called a *constraint*.

**Negation as failure in the heads of rules.** In 1992, Lifschitz and Woo extended the answer set semantics to programs that may contain negation as failure in the heads of rules. For instance, the answer sets for the program

$$p; not\ p \leftarrow \tag{2.7}$$

are $\{p\}$ and the empty set. Later on, Inoue and Sakama [1994] related such programs to "abductive logic programs" in the sense of [Kakas *et al.*, 1992].

**Nested expressions.** In [Lifschitz *et al.*, 1999], the answer set semantics is defined for programs with nested expressions—formulas formed from literals using negation as failure, conjunction (,) and disjunction (;) that can be nested arbitrarily, and are allowed both in heads and bodies of rules. For instance,

$$r \leftarrow$$
$$(p; not\ p), (q; not\ q) \leftarrow r; \neg s$$

is a program with nested expressions that has the answer sets $\{r\}$, $\{r, p\}$, $\{r, q\}$, and $\{r, p, q\}$.

## 2.3 Answer Set Solvers

In the last five years, several systems have been developed, which, among other things, can be used to compute the answer sets of a logic program, such as CCALC [McCain and Turner, 1998],[5] DCS [East and Truszczyński, 2000],[6] DERES [Cholewiński *et al.*, 1996],[7] DLV [Eiter *et al.*, 1997],[8] and SMODELS [Niemelä and Simons, 1996].[9]

The systems DLV and SMODELS are based on the answer semantics. The other systems can be used to compute the answer sets for some logic programs as well due to the relationships between the answer set semantics and the semantics they are based on. For instance, CCALC can compute the models of

[5] `http://www.cs.utexas.edu/users/tag/cc/`.
[6] `http://www.cs.uky.edu/dcs/` .
[7] `http://www.cs.engr.uky.edu/ai/deres.html` .
[8] `http://www.dbai.tuwien.ac.at/proj/dlv/`.
[9] `http://www.tcs.hut.fi/Software/smodels/`.

the completion of a program using a propositional solver, and under some conditions these models are also the answer sets for the program (Section 2.1.5). DERES is an implementation of default logic, and, due to Proposition 3 of [Gelfond and Lifschitz, 1991], there is a 1-1 correspondence between the answer sets for a program and the "extensions" for the corresponding default theory. In the following, we will briefly describe the answer set solvers DLV and SMODELS, and then discuss computing answer sets using a propositional solver.

### 2.3.1 DLV

DLV [Eiter *et al.*, 1997] is a datalog system (i.e., a deductive database system without function symbols) that supports disjunction and classical negation.

Given such a logic program, DLV finds the answers sets for the program with an algorithm outlined in [Eiter *et al.*, 1998] and explained in [Citrigno *et al.*, 1997] and [Eiter *et al.*, 1997].

### 2.3.2 SMODELS

SMODELS [Niemelä and Simons, 1996], [Simons *et al.*, 2002] is an implementation of the answer set semantics for logic programs that may contain classical negation and function symbols, but no disjunctive rules. This limitation is mitigated to some extent by two circumstances.

First, the input language of SMODELS allows us to express any "exclusive disjunctive rule". For instance, the combination

$$\begin{aligned} p; q &\leftarrow \\ &\leftarrow p, q \end{aligned} \tag{2.8}$$

can be presented to SMODELS as an "exclusive disjunctive rule":

$$p \mid q \; .$$

Second, SMODELS allows us to represent disjunctions of the form

$$L; \mathit{not}\ L$$

appearing in heads of rules. For instance, we represent program (2.7) in the language of SMODELS as

$$\{ \ p \ \}.$$

This rule is called a "choice rule": $p$ can be chosen to be included or not included in an answer set.

A nice feature of SMODELS is that it supports "weight constraints" [Simons, 1999], [Niemelä *et al.*, 1999], [Niemelä and Simons, 2000]. That is, we can assign a "weight" to each literal, and put some constraints on the sum of the weights of the literals that belong to the answer sets. If the weights of all literals have the default value 1, then the constraint is called a "cardinality constraint".

SMODELS computes the answer sets for a given program according to the algorithm explained in [Niemelä and Simons, 1996], [Simons, 1997], and [Simons, 2000].

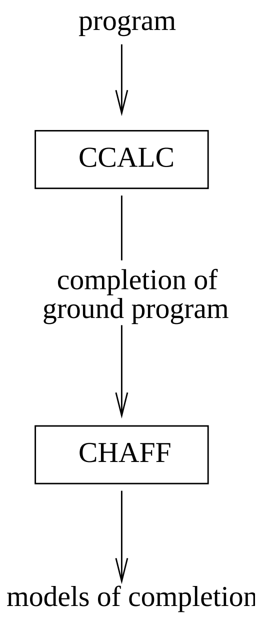


Figure 2.1: Computing answer sets for a tight program.

### 2.3.3 Computing Answer Sets using Propositional Solvers

Propositional solvers, such as CHAFF [Moskewicz *et al.*, 2001], SATO [Zhang, 1997] and RELSAT [Bayardo and Schrag, 1997],[10] are systems that can compute a model of a propositional theory given as a set of clauses. Many of these systems are based on the Davis-Putnam procedure [Davis and Putnam, 1960].

Propositional solvers can be used, in particular, to compute the models of the completion of a logic program. As mentioned in Section 2.1.5, François Fages [1994] showed that the models of the completion of a tight program are also the answer sets for that program. Consequently, we can use propo-

[10] `http://aida.intellektik.informatik.tu-darmstadt.de/~hoos/SATLIB/solvers.html` .

sitional solvers, instead of answer set solvers, to find the answer sets for a tight program. (See Sections 3.4.1 and 3.4.2 for the definition of completion and for the definition of tightness respectively.) The necessary preprocessing step—turning a logic program into its completion and converting the completion into clausal form—can be performed by the Causal Calculator, or CCALC (Figure 2.1). This is a system written by Norm McCain as part of his dissertation defended at the University of Texas at Austin in 1997. Now it is maintained by Texas Action Group at Austin.[11] We will investigate the use of propositional solvers as answer set programming tools in Chapters 5 and 6.

## 2.4 Answer Set Programming

The idea of answer set programming [Marek and Truszczyński, 1999], [Niemelä, 1999], [Lifschitz, 1999] is to represent a computational problem as a logic program whose answer sets correspond to the solutions of the problem, and to find the answer sets for that program using an answer set solver. Answer set programming has been used to solve combinatorial search problems in various fields, such as graph theory, propositional satisfiability checking, planning[12] [Dimopoulos *et al.*, 1997, Lifschitz, 1999], diagnosis [Eiter *et al.*, 1999], [Gelfond and Galloway, 2001], model checking [Liu *et al.*, 1998], [Heljanko and Niemelä, 2001], reachability analysis [Heljanko, 1999], product configuration

[11] `http://www.cs.utexas.edu/users/tag/` .

[12] The idea of relating planning problems to answer sets was first proposed in [Subrahmanian and Zaniolo, 1995].

[Soininen and Niemelä, 1998], dynamic constraint satisfaction [Soininen *et al.*, 1999], logical cryptanalysis [Hietalahti *et al.*, 2000], network inhibition analysis [Aura *et al.*, 2000], workflow specifications [Trajcevski *et al.*, 2000], [Köksal *et al.*, 2001], learning [Sakama, 2001], reasoning about policies [Son and Lobo, 2001], and circuit design [Balduccini *et al.*, 2000], [Balduccini *et al.*, 2001]. We used answer set programming to solve wire routing problems [Erdem *et al.*, 2000] and phylogeny reconstruction problems [Erdem *et al.*, 2002]. This will be discussed in Chapters 7 and 8.

# Chapter 3

# Answer Sets and Planning

In this chapter, we summarize experiments with the systems DLV, SMODELS, and CCALC applied to some planning problems.

In the following, we first introduce the syntax of the logic programs used in these experiments, and define the answer set semantics for such programs. Then we discuss the problem of plan generation in general and describe the first of the planning problems we experimented with—the "suitcase problem". Then, we show how to present the problem to DLV, SMODELS and CCALC, and describe the results of our experiments. Next, we define the blocks world domain, and several larger planning problems that we experimented with using DLV, SMODELS and CCALC, and present the information on computation times.[1] After that, we compare our approach to planning with related work,

[1]For details of our experiments with the use of systems CCALC, DLV and SMODELS to solve planning problems in the blocks world, see [Erdem, 1999] available at `http://www.cs.utexas.edu/users/ esra/experiments/experiments.html` .

and discuss some enhancements of the blocks world planning program.

## 3.1 Programs

The syntax of logic programs studied in this proposal is defined as follows. We begin with a set of propositional symbols, called *atoms*. A *literal* is an expression of the form $A$ or $\neg A$, where $A$ is an atom. A *rule element* is an expression of the form $L$ or *not* $L$, where $L$ is a literal. A *rule* is an ordered pair

$$Head \leftarrow Body \tag{3.1}$$

where *Head* is a finite set of literals, and *Body* is a finite set of rule elements. If

$$Head = \{L_1, \dots, L_k\}$$

and

$$Body = \{L_{k+1}, \dots, L_m, not\ L_{m+1}, \dots, not\ L_n\}$$

($0 \leq k \leq m \leq n$) then we will write (3.1) as

$$L_1; \dots; L_k \leftarrow L_{k+1}, \dots, L_m, not\ L_{m+1}, \dots, not\ L_n. \tag{3.2}$$

If the body is empty, we will sometimes drop $\leftarrow$; a rule with the empty body and one literal in the head is called a *fact*. If the head is empty, we will sometimes denote it by $\bot$; a rule with the empty head is called a *constraint*. A *program* is a set of rules. A program is called *nondisjunctive* if, in every rule, $k \leq 1$. We denote the set of literals in the language of a program $\Pi$ by $lit(\Pi)$.

We say that a consistent set $X$ of literals is *closed under* $\Pi$ if, for every rule (3.2) in $\Pi$,

$$\{L_1, \ldots, L_k\} \cap X \neq \emptyset \tag{3.3}$$

whenever

$$\{L_{k+1}, \ldots, L_m\} \subseteq X \tag{3.4}$$

and

$$\{L_{m+1}, \ldots, L_n\} \cap X = \emptyset. \tag{3.5}$$

This definition of closure corresponds to the definition of closure introduced in [Lifschitz, 1996] for programs without negation as failure.

Let $\Pi$ be a program without negation as failure. Then we say that $X$ is an answer set for $\Pi$ iff $X$ is a minimal set closed under $\Pi$. For instance, the answer sets for

$$p; q \tag{3.6}$$

are $\{p\}$ and $\{q\}$.

Now consider a program $\Pi$ that may contain negation as failure. The *reduct* of $\Pi$ relative to a consistent set $X$ of literals, as defined in [Gelfond and Lifschitz, 1991] and [Lifschitz, 1996], is obtained from $\Pi$

- by deleting each rule (3.2) that does not satisfy (3.5) and

- by replacing each remaining rule (3.2) by

$$L_1; \dots ; L_k \leftarrow L_{k+1}, \dots , L_m. \tag{3.7}$$

This program will be denoted by $\Pi^X$. For instance, consider the program

$$\begin{array}{l} p; q \\ \neg r \leftarrow \mathit{not}\ p. \end{array} \tag{3.8}$$

The reduct of this program relative to $\{p\}$ is (3.6).

We say that $X$ is an answer set for a program $\Pi$ iff $X$ is an answer set for $\Pi^X$. Consider, for instance, program (3.8) and its reduct (3.6) relative to $\{p\}$. Since $\{p\}$ is an answer set for (3.6), this set is an answer set for program (3.8) as well. It is easy to check that $\{q, \neg r\}$ is an answer set for program (3.8) too.

The definitions of an answer set given above are different from the ones in [Gelfond and Lifschitz, 1991] and [Lifschitz, 1996] in that we consider consistent answer sets only.

## 3.2 Representing Actions and Planning

Computational problems related to action and change are an important application area of answer set programming. Answer set programming can be used, for instance, to solve temporal projection problems and planning problems. In a temporal projection problem, the task is, given an initial state and a sequence of actions to be executed, to predict the outcome of these actions.

For instance, if a monkey moves a box to a new location and then climbs the box, we can predict that both the box and the monkey would be at that location and the monkey would be on the box. In a planning problem, the task is, given a goal, to find a plan—a sequence of actions that leads to the given goal. For instance, think of a monkey faced with the problem of getting a bunch of bananas hanging from the ceiling just beyond his reach. There is a box in the room, so that the monkey can solve the problem by pushing the box to an empty space under the bananas, climbing on top of the box, and then reaching the bananas. This is a plan.

To represent actions and change, we think of the world as being in one of many *states* (or *situations* [McCarthy and Hayes, 1969]). To be able to describe the states of the world, McCarthy and Hayes introduced *fluents*—relations or functions whose values depend on time or situation. For instance, the location of an object is a fluent that may change over time. The world changes its state due to some causal dependencies. Some of these causal dependencies involve events, such as *actions* performed by some agent(s).

To describe a domain in the world, we need to introduce fluents and actions, and then represent the causal dependencies that lead to change in this domain. For that, we need to describe the changes caused by performing actions. However, here, the challenge is to describe also what does not change, i.e., to solve the frame problem [McCarthy and Hayes, 1969].

Consider, for instance, a robot that has to move boxes to some locations. If the robot walks to a location then the position of the robot changes.

Meanwhile, the boxes do not change their locations. Representing this last fact is an instance of the frame problem. Without solving the frame problem, we would not be able to draw useful conclusions about the world.

A causal dependency does not always involve an action. Related to this is the "ramification problem" discovered by Finger [1986]. The problem is to represent the changes that are implied by the execution of actions. Another problem is the "qualification problem" discovered by McCarthy [1980]. This problem is about representing a change that prevents an action from being executed.

Consider the problem described above. If the robot is at some location then it is not anywhere else. This causal dependency does not involve any action, but implies an "indirect" effect (or "ramification") of walking in that if the robot walks from location $l1$ to location $l2$ then it is not at location $l1$ anymore. On the other hand, the same causal dependency prevents the robot from walking to two different locations at the same time. This serves as an "implicit" precondition to performing the action of walking. Therefore, representing this causal dependency should address both the ramification problem and the qualification problem.

A solution to the frame problem was proposed by Schubert [1990] using the "explanation closure axioms", which, for every fluent, give a complete explanation of how the fluent changes. This monotonic solution is not applicable in the presence of ramifications. This is why the frame problem has been studied in many nonmonotonic logic formalisms. (See [Shanahan, 1993] for formalizations of the frame problem using the situation calculus, circumscription, event calculus, default logic, and logic programming; [Geffner, 1990] and [McCain and Turner, 1997] for formalizations of the frame problem using nonmonotonic causal logics, and [Kautz *et al.*, 1996] for formalizations of the frame problem in the context of satisfiability planning.)

There are other challenging problems with representing actions and change, such as representing actions that can be concurrently executed, representing actions that have nondeterministic effects, representing actions that have delayed effects, representing "sensing" actions that change our knowledge of the world (rather than the state of the world), etc.

## 3.3 The Suitcase Problem

### 3.3.1 Suitcase Domain

The suitcase domain is introduced in [Lin, 1995]. Consider a suitcase with two latches $L1$ or $L2$; when these two latches are up then the suitcase automatically opens. There are three propositional fluents: $up(l)$, where $l$ is $L1$ or $L2$, and $open$; $up(l)$ holds iff latch $l$ is up, $open$ holds iff the suitcase is open. There is an action of toggling a latch $l$ denoted by $toggle(l)$. If a latch is down (resp. up) then it becomes up (resp. down) after toggling it.

This domain is interesting because it involves a simple instance of the ramification problem: opening the suitcase may be an indirect effect of toggling a switch.

Our method of encoding planning problems by logic programs is similar to the one used in [Lifschitz, 1999]. Lifschitz starts with a planning domain described as a "transition system"—a directed graph whose nodes are characterized by the values of fluents and whose edges correspond to the actions that are executed. A path starting from an initial state and reaching a goal state is a "history of the world" that corresponds to a plan for the given planning problem. This transition system can be described by a logic program. We call the logic program describing a transition system a "history program"—a program whose answer sets represent possible histories of the world over a fixed time interval.

The transition system describing this domain has 7 possible states:

$$\{up(L1), up(L2), open\} \qquad (S_1)$$

$$\{up(L1), \neg up(L2), open\} \qquad (S_2)$$

$$\{up(L1), \neg up(L2), \neg open\} \qquad (S_3)$$

$$\{\neg up(L1), up(L2), open\} \qquad (S_4)$$

$$\{\neg up(L1), up(L2), \neg open\} \qquad (S_5)$$

$$\{\neg up(L1), \neg up(L2), open\} \qquad (S_6)$$

$$\{\neg up(L1), \neg up(L2), \neg open\} \qquad (S_7)$$

Note that $\{up(L1), up(L2), \neg open\}$ is not a possible state. At each possible state there are 4 applicable actions:

| | | |
|---|---|---|
| $\{toggle(L1), toggle(L2)\}$ | $(A_1)$ | "toggle $L1$ and $L2$ concurrently" |
| $\{\neg toggle(L1), \neg toggle(L2)\}$ | $(A_2)$ | "do nothing" |
| $\{toggle(L1), \neg toggle(L2)\}$ | $(A_3)$ | "toggle $L1$" |
| $\{\neg toggle(L1), toggle(L2)\}$ | $(A_4)$ | "toggle $L2$" |

so that each state has 4 outgoing edges. For instance, the edges outgoing from $S_1$ are

$$\langle S_1, A_1, S_6\rangle, \langle S_1, A_2, S_1\rangle, \langle S_1, A_3, S_4\rangle, \langle S_1, A_4, S_2\rangle.$$

The transition system that corresponds to the suitcase domain has 28 edges.

The first planning problem we experimented with can be described as follows:

**Initial State:** The latches $L1$ and $L2$ are down, and the suitcase is closed.

**Goal State:** The latches $L1$ and $L2$ are down, and the suitcase is open.

This problem has a solution of length 2, corresponding to the following path in the transition system:

$$\langle S_7, A_1, S_1, A_1, S_6\rangle. \tag{3.9}$$

### 3.3.2 The Suitcase Problem as a Logic Program

We solve the suitcase problem described in Section 3.3.1 by formalizing it as a logic program and then using an answer set solver. Let $T_{max}$ be a positive integer—a fixed length of paths in the transition system. We describe the suitcase domain as the logic program consisting of the rules

$$open(t) \leftarrow up(L1, t), up(L2, t) \tag{3.10}$$

where $t \in \{0, .., T_{max}\}$, and the rules

$$\begin{aligned}
&up(l, t+1) \leftarrow toggle(l, t), \neg up(l, t)\\
&\neg up(l, t+1) \leftarrow toggle(l, t), up(l, t)\\
&up(l, t+1) \leftarrow not\ \neg up(l, t+1), up(l, t)\\
&\neg up(l, t+1) \leftarrow not\ up(l, t+1), \neg up(l, t)\\
&open(t+1) \leftarrow not\ \neg open(t+1), open(t)\\
&\neg open(t+1) \leftarrow not\ open(t+1), \neg open(t)\\
&up(l, 0); \neg up(l, 0)\\
&open(0); \neg open(0)\\
&toggle(l, t); \neg toggle(l, t)
\end{aligned} \tag{3.11}$$

where $l \in \{L1, L2\}$ and $t \in \{0, .., T_{max} - 1\}$.

In the program above, rule (3.10) says that if both latches $L1$ and $L2$ are open then the suitcase gets open. The first two rules of (3.11) describe the effects of toggling: if a latch is down (resp. up) then it becomes up (resp. down) after toggling it. The following four rules implement the commonsense law of inertia presented in Section 2.1.1. (This is how the frame problem, discussed in Sections 2.1.1 and 3.2, is solved.) For instance, the rule

$$up(l, t+1) \leftarrow not\ \neg up(l, t+1), up(l, t)$$

says: if a latch is up at time $t$ and there is no evidence that it will not be up at time $t+1$ then indeed it will be up. The disjunctive rules say that the initial values of all fluents, and the occurrences and non-occurrences of actions are exogenous: they may have any value.

The answer sets for program (3.10) $\cup$ (3.11) with $T_{max} = 2$ are in a 1-1 correspondence with the paths of length 2 in the transition system described in Section 3.3.1. Specifically, each answer set is complete,[2] and its "time slices" correspond to the states and atoms forming such a path. For instance, the answer set that corresponds to path (3.9) is

$$\begin{aligned}&\{\neg up(L1,0), \neg up(L2,0), \neg open(0), toggle(L1,0), toggle(L2,0),\\ &\ up(L1,1), up(L2,1), open(1), toggle(L1,1), toggle(L2,1),\\ &\ \neg up(L1,2), \neg up(L2,2), open(2)\}.\end{aligned}$$

The suitcase problem can be described as the problem of finding an answer set (for program (3.10) $\cup$ (3.11) with $T_{max} = 2$) that contains

$$\neg up(L1,0), \neg up(L2,0), \neg open(0), \neg up(L1,2), \neg up(L2,2), open(2).$$

### 3.3.3 The Suitcase Problem presented to DLV

We want to present program (3.10) $\cup$ (3.11) to DLV. A program has to satisfy three conditions to be presentable to DLV.

The first condition is mentioned in Section 2.3.1: the program should not contain any function symbols.

The second condition is that the program should be "range-restricted". A logic program is *range-restricted* if, for each rule of that program, every

[2] A set of literals is said to be *complete* if, for every atom $A$, it contains $A$ or $\neg A$.

variable occurring in the rule also occurs in one of the literals that appears in the body of the rule not preceded by negation as failure. For instance, the program

$$p(x) \leftarrow r(x), not\ q(x) \tag{3.12}$$

is range-restricted, whereas

$$p(x) \leftarrow not\ q(x)$$

is not range-restricted.

The third condition is as follows. An expression of the form $P$ or $\neg P$, where $P$ is a predicate symbol, is *extensional* in a program if every rule that contains a literal beginning with this expression in the head is a fact, and *intensional* if no such rule is a fact. In an input program presented to DLV, every expression of the form $P$ or $\neg P$, where $P$ is a predicate symbol, is required to be extensional or intensional. For instance, the program

$$\begin{array}{l} p(a) \\ p(b) \leftarrow p(a) \end{array}$$

violates this condition.

Lin's suitcase domain can be presented to DLV as in Figure 3.1; and the suitcase problem is described in a separate file as in Figure 3.2.[3] This input satisfies the three conditions mentioned above. Then DLV can be invoked by typing

[3] Alternatively, the domain and the problem for Lin's Suitcase can be introduced in the same file.

```
dlv suitcase_domain suitcase_problem -n=1 -N=2
```

where `suitcase_domain` and `suitcase_problem` are the names of the files presented in Figures 3.1 and 3.2 respectively. The option `-n=1` specifies that only one answer set is to be computed. The option `-N=2` determines the value of `#maxint` (see below) to be 2.

The domain description in Figure 3.1 is an adaptation of program (3.10) $\cup$ (3.11) to the syntactic conventions of DLV. As in Prolog, we use `:-` instead of $\leftarrow$ and include a period at the end of every rule. Classical negation is denoted by '`-`' instead of '$\neg$'. One more difference is that Figure 3.1 contains the auxiliary predicates `latch`, `time`, and `next`. This difference is not essential: we can easily show that there exists a 1-1 correspondence between the answer sets for program (3.10)$\cup$(3.11) and the answer sets for the program in Figure 3.1, using the "splitting set theorem" [Lifschitz and Turner, 1994].

In the problem description (Figure 3.2), the initial conditions and the goal are presented as a "query", i.e., an expression of the form

$$b_1, \ldots, b_n, not\ b_{n+1}, \ldots, not\ b_m? \tag{3.13}$$

where $1 \leq n \leq m$ and each $b_i$ $(1 \leq i \leq m)$ begins with an intensional expression. An answer set *satisfies* query (3.13) if it contains $b_1, \ldots, b_n$, but not $b_{n+1}, \ldots, b_m$. Given a domain description and a query that describes a problem, DLV finds the answer sets for the domain description that satisfy the query.

```
% suitcase is open if both of the latches are open
open(T) :- up(l1,T), up(l2,T).

% effects of toggling
up(L,T1) :- latch(L), next(T,T1), toggle(L,T), -up(L,T).
-up(L,T1) :- latch(L), next(T,T1), toggle(L,T), up(L,T).

% inertia
up(L,T1) :- latch(L), next(T,T1), up(L,T), not -up(L,T1).
-up(L,T1) :- latch(L), next(T,T1), -up(L,T), not up(L,T1).

open(T1) :- next(T,T1), open(T), not -open(T1).
-open(T1) :- next(T,T1), -open(T), not open(T1).

% initial conditions are exogenous
up(L,0); -up(L,0) :- latch(L).
open(0); -open(0).

% actions are exogenous
toggle(L,T); -toggle(L,T) :- latch(L), time(T), T < #maxint.

% auxiliary predicates

latch(l1).
latch(l2).

time(T) :- #int(T).

next(X,Y) :- #succ(X,Y).
```

Figure 3.1: File suitcase_domain for DLV.

```
% initial conditions and the goal
-up(l1,0), -up(l2,0), -open(0), open(#maxint),
-up(l1,#maxint), -up(l2,#maxint) ?
```

Figure 3.2: File `suitcase_problem` for DLV.

A query is an alternative way of representing a set of constraints: including query (3.13) in the input has the same effect as adding to the program the constraints

$$
\begin{array}{l}
\bot \leftarrow not\ b_1 \\
\vdots \\
\bot \leftarrow not\ b_n \\
\leftarrow b_{n+1} \\
\bot \vdots \\
\bot \leftarrow b_m
\end{array}
$$

Files in Figures 3.1 and 3.2 use the DLV built-in functions `#maxint`, `#int` and `#succ`. In DLV, `#int(T)` expresses that `T` is an integer between 0 and `#maxint`, where the value of `#maxint` is specified in the command line. The atom `#succ(T,T1)` expresses that both `T` and `T1` are between 0 and `#maxint`, and `T1` is `T + 1`.

The answer set computed by DLV corresponds to solution (3.9).

### 3.3.4 The Suitcase Problem presented to SMODELS

As mentioned in Section 2.3.2, a program given as input to SMODELS should not contain disjunctive rules, but exclusive disjunctions and choice rules are allowed. As in DLV, SMODELS also requires some syntactic restrictions on the input programs.

A logic program presented to SMODELS is required to be "strongly" range-restricted in the following sense. A predicate symbol $P$ is a *domain predicate* in a program $\Pi$, if neither $P$ nor the predicates $P$ depends on use recursion in their definitions. A logic program $\Pi$ is *strongly range-restricted* if, for each rule of $\Pi$, every variable occurring in the rule also occurs in a literal that appears in the body of the rule not preceded by negation as failure, which includes a domain predicate. For instance, rule (3.12) combined with the definition

$$r(a)$$

of predicate $r$ is strongly range restricted. But this property will be lost if we extend the program by adding the rule

$$r(f(x)) \leftarrow r(x).$$

Lin's suitcase domain can be presented to SMODELS as in Figure 3.3; and the suitcase problem is described in a separate file as in Figure 3.4. Then SMODELS can be invoked by typing

```
lparse -c lasttime=2 --true-negation suitcase_domain suitcase_problem
| smodels
```

where `suitcase_domain` is the name of the file in Figure 3.3, and `suitcase_problem` is the name of the file in Figure 3.4. The option `-c lasttime=2` specifies that the value of the constant `lasttime` is 2, and the option `--true-negation` specifies that the classical negation symbol `-` is used. By the command `lparse` the input files are grounded—each rule with variables is replaced by the set of its ground instances. After that, the answer sets for the ground program are computed by the command `smodels`.

Let us discuss the differences between Figure 3.3 and Figure 3.1. Nothing in the input of SMODELS corresponds to `-toggle` in Figures 3.3 and 3.4 because the only rule in Figure 3.1 where `-toggle` appears in the head

```
toggle(L,T); -toggle(L,T) :- latch(L), time(T), T < #maxint.
```

is replaced by the choice rule

```
{toggle(L,T):latch(L)} :- time(T), lt(T,lasttime).
```

Alternatively, we could have expressed that actions are exogenous by the exclusive disjunctive rule

```
toggle(L,T) | -toggle(L,T) :- latch(L), time(T), lt(T,lasttime).
```

Since `-toggle` does not occur in any of the rules in Figure 3.3, using the choice rule above is computationally more efficient.

```
% suitcase is open if both of the latches are open
open(T) :- time(T), up(l1,T), up(l2,T).

% effects of toggling
up(L,T1) :- latch(L), next(T,T1), toggle(L,T), -up(L,T).
-up(L,T1) :- latch(L), next(T,T1), toggle(L,T), up(L,T).

% inertia
up(L,T1) :- latch(L), next(T,T1), up(L,T), not -up(L,T1).
-up(L,T1) :- latch(L), next(T,T1), -up(L,T), not up(L,T1).

open(T1) :- next(T,T1), open(T), not -open(T1).
-open(T1) :- next(T,T1), -open(T), not open(T1).

% initial conditions are exogenous
up(L,0) | -up(L,0) :- latch(L).
open(0) | -open(0).

% actions are exogenous
{toggle(L,T):latch(L)} :- time(T), lt(T,lasttime).

% auxiliary predicates
latch(l1).
latch(l2).

time(0..lasttime).

next(T,T+1) :- time(T), lt(T,lasttime).
```

Figure 3.3: File suitcase_domain for SMODELS.

```
% initial conditions and the goal
compute 1 {-up(l1,0), -up(l2,0), -open(0),  open(lasttime),
-up(l1,lasttime), -up(l2,lasttime)}.
```

Figure 3.4: File suitcase_problem for SMODELS.

The disjunctive rule

```
up(L,0); -up(L,0) :- latch(L)
```

is replaced by the exclusive disjunctive rule

```
up(L,0) | -up(L,0) :- latch(L).
```

which ensures that either `up(L,0)` or `-up(L,0)` is in an answer set provided that `latch(L)` is already in that answer set. The disjunctive rule, which specifies that `open(0)` is exogenous, is modified in a similar way.

As the rule

```
open(T) :- up(l1,T), up(l2,T)
```

is not strongly range-restricted, it is replaced by

```
open(T) :- time(T), up(l1,T), up(l2,T) .
```

In the problem description, the initial conditions and the goal are included in a `compute` statement. In a `compute` statement, we can specify how many answer sets we want SMODELS to find, and which atoms the computed answer sets should or should not contain. For instance, the `compute` statement in Figure 3.4 tells SMODELS to find one answer set that contains

```
-up(l1,0), -up(l2,0), -open(0),
open(lasttime), -up(l1,lasttime), -up(l2,lasttime) .
```

If `all` were written instead of 1 in the `compute` statement of Figure 3.4, all answer sets that contain the above atoms would have been found. Note that in a `compute` statement the list in braces is a way to represent a set of constraints. In our case including this list has the same effect as adding to the program the constraints

```
:- not -up(l1,0)
:- not -up(l2,0)
:- not -open(0)
:- not open(lasttime)
:- not -up(l1,lasttime)
:- not -up(l2,lasttime) .
```

SMODELS does not have built-in functions similar to `#maxint`, `#int`, and `#succ` of DLV. However, it has built-in function `+` and operator `lt` ("less than") used in

```
next(T,T+1) :- time(T), lt(T,lasttime).
```

In addition, it has an elegant way of representing a range of integers, as in

```
time(0..lasttime).
```

The programs in Figures 3.3 and 3.1 are not quite equivalent to each other, because of the difference in the use of `-toggle` mentioned above. But there is a simple 1-1 correspondence between the answer sets of both programs.

## 3.4 Completion

In the next section, we investigate the use of propositional solvers to solve the suitcase problem. As mentioned in Section 2.3.3, the idea is to run CCALC to form the completion of the program and then invoke a propositional solver to find a model of the completion.

In this section, we discuss the process of completing a program and its relation to answer sets.

### 3.4.1 Definition of Completion

As mentioned in Section 2.1.2, program completion is introduced by Clark [1978]. For programs without variables, it is defined as follows.

Consider a finite nondisjunctive program $\Pi$ without classical negation. The rules of $\Pi$ have the form

$$Head \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n \tag{3.14}$$

$(n \geq m \geq 0)$ where each $A_i$ is a propositional atom, and *Head* is an atom or the symbol $\bot$. If $A$ is an atom or the symbol $\bot$, by $Comp(\Pi, A)$ we denote the formula

$$A \equiv (Body'_1 \vee \ldots \vee Body'_k)$$

where the disjunction extends over all rules

$$Head \leftarrow Body_i$$

in $\Pi$ where *Head* is $A$, and each $Body'_i$ $(1 \leq i \leq k)$ is obtained from $Body_i$ by replacing each comma by $\wedge$, and by replacing *not* by $\neg$. The *completion* of $\Pi$ is the set of formulas $Comp(\Pi, A)$ for all $A$. For instance, the completion of program (2.3) (page 11) is

$$\begin{aligned} p &\equiv \neg q, \\ q &\equiv \neg p. \end{aligned} \tag{3.15}$$

### 3.4.2 Fages' Theorem

To relate the definition of completion to answer sets, observe first that an answer set for a program without classical negation never contains negative literals—it is a set of atoms. Second, any set $X$ of atoms can be viewed as a truth assignment: the atoms that belong to $X$ are true, and the others are false. For instance, take the answer set $\{p\}$ for program (2.3). It represents the truth assignment that makes $p$ true, and makes $q$ false. Note that this truth assignment satisfies the completion (3.15) of the program. The (truth assignment corresponding to the) second answer set $\{q\}$ satisfies (3.15) as well. In this case, the models of the program's completion are identical to its answer sets.

The general situation is more complicated. Consider a finite nondisjunctive program $\Pi$ without classical negation, and a set $X$ of atoms. If $X$ is an answer set for $\Pi$ then $X$, viewed as a truth assignment, satisfies the completion of $\Pi$. The converse, generally, is not true. For instance, the completion of

$$p \leftarrow p \tag{3.16}$$

is $p \equiv p$. This formula has two models $\emptyset$, $\{p\}$; the first is an answer set for (3.16), but the second is not. Fages' theorem tells us that, for a tight program, all models of the completion are the program's answer sets.

Tightness (called "positive-order-consistency" by Fages) is defined for nondisjunctive programs. Recall that, in a nondisjunctive program, every rule is an expression of the form

$$\mathit{Head} \leftarrow L_1, \dots, L_m, \mathit{not}\ L_{m+1}, \dots, \mathit{not}\ L_n \tag{3.17}$$

($n \geq m \geq 0$) where each $L_i$ is a literal, and *Head* is a literal or the symbol $\bot$ (Section 3.1).

A nondisjunctive program $\Pi$ is *tight* if there is a function $\lambda$ from $\mathit{lit}(\Pi)$ to ordinals such that, for every rule (3.17) in $\Pi$, if *Head* is not $\bot$ then

$$\lambda(L_1), \dots, \lambda(L_m) < \lambda(\mathit{Head}).$$

For instance, program (2.4) (page 12) is tight: take $\lambda(p) = \lambda(q) = 0$ and $\lambda(r) = 1$. Program (3.16) is not tight.

Consider the program (3.10) $\cup$ (3.11). This program is tight, which can be proved using the following function $\lambda$:

$$\lambda(L) = \begin{cases} 1, & \text{if } L \text{ has the form } \mathit{open}(t), \\ 0, & \text{otherwise.} \end{cases}$$

where $l \in \{L1, L2\}$ and $t \in \{0, .., T_{max}\}$.

Fages' theorem [Fages, 1994] can be stated as follows: *For any finite nondisjunctive program* $\Pi$ *without classical negation, if* $\Pi$ *is tight then the models of completion of* $\Pi$ *are identical to the answer sets for* $\Pi$.

The proposition below gives a simple characterization of tightness that does not refer to ordinals (and is actually close to Fages' original formulation). For any program $\Pi$, we say about literals $L, L' \in lit(\Pi)$ that $L$ is a *parent* of $L'$ relative to $\Pi$ if there is a rule (3.17) in $\Pi$ such that

- $L \in \{L_1, \ldots, L_m\}$, and
- $L' = Head$.

For instance, the parents of $r$ relative to (2.4) are $p$ and $q$.

**Proposition 1** *A program* $\Pi$ *is tight iff there is no infinite sequence* $L_0, L_1, \ldots$ *of elements of* $lit(\Pi)$ *such that for every* $i$*,* $L_{i+1}$ *is a parent of* $L_i$ *relative to* $\Pi$.

In other words, $\Pi$ is tight iff the parent relation relative to $\Pi$ is "well-founded". A binary relation $R$ is *well-founded* if there is no infinite sequence

$$x_0, x_1, \ldots \tag{3.18}$$

of elements of its domain such that, for all $i$, $x_{i+1} R x_i$. For instance, program (3.16) is not tight; take all $x_i$ in sequence (3.18) to be $p$.

Proposition 1 is a special case of the following general fact of set theory:

**Lemma 1** *A binary relation $R$ is well-founded iff there exists a function $\lambda$ from the domain of $R$ to ordinals such that, for all $x$ and $y$, $xRy$ implies $\lambda(x) < \lambda(y)$.*

The proof of Lemma 1 [Erdem and Lifschitz, 2001b] is presented in Section 9.1 (page 163). Tightness and Fages' Theorem will be discussed further in Chapter 5.

### 3.4.3 Literal Completion

The procedure implemented in CCALC is *literal completion* [McCain and Turner, 1997]. This is a syntactic transformation similar to Clark's completion discussed above. Unlike Clark's completion, literal completion is applicable to programs with classical negation. When applied to a tight program, it produces a formula that characterizes the complete answer sets for that program (see Footnote ($^2$) on page 33).

Program (3.10) $\cup$ (3.11) is tight, and its answer sets are complete. Consequently, the models of the literal completion of this program are also its answer sets. This justifies the use of CCALC to find the answer sets for this program.

## 3.5 The Suitcase Problem presented to CCALC

Recall that completion is defined for nondisjunctive programs only. When a program contains disjunctive rules, we cannot apply the process of completion directly. But if all disjunctive rules of the program are of the form (2.6) (page 16) then we can apply completion after we transform the program into a nondisjunctive program by replacing every disjunctive rule (2.6) with nondisjunctive rules

$$\begin{array}{c} p \leftarrow not\ \neg p \\ \neg p \leftarrow not\ p. \end{array} \tag{3.19}$$

This transformation preserves the answer sets for the program we start with due to our theorem presented in Section 4.1.

For instance, program (3.10) $\cup$ (3.11) contains disjunctive rules

$$\begin{array}{l} up(l, 0); \neg up(l, 0) \\ open(0); \neg open(0) \\ toggle(l, t); \neg toggle(l, t). \end{array}$$

Each of these rules can be replaced by a pair of nondisjunctive rules as shown above.

Lin's suitcase domain can be presented to CCALC as in Figures 3.5 and 3.6; and the suitcase problem can be described by a logic program as in Figure 3.7.

After compiling CCALC in Sicstus Prolog by

```
compile('ccalc').
```

we "load" the files presented in Figures 3.5–3.7 by typing

```
:- sorts
       latch; time >> step.

:- variables
       L :: latch;
       T :: step;
       T1 :: time.

:- constants
       0..lasttime-1 :: step;
       lasttime :: time;
       l1, l2 :: latch;
       up(latch,time),
       open(time),
       toggle(latch,step) :: atomicFormula.

:- macros
       next(#1,#2) -> #2 is (#1) + 1.

% suitcase is open if both of the latches are open
open(T1) :- up(l1,T1), up(l2,T1).

% effects of toggling
up(L,T1) :- next(T,T1), toggle(L,T), -up(L,T).
-up(L,T1) :- next(T,T1), toggle(L,T), up(L,T).

% inertia
up(L,T1) :- next(T,T1), up(L,T), not -up(L,T1).
-up(L,T1) :- next(T,T1), -up(L,T), not up(L,T1).

open(T1) :- next(T,T1), open(T), not -open(T1).
-open(T1) :- next(T,T1), -open(T), not open(T1).
```

Figure 3.5: File suitcase_domain.b for CCALC, part 1.

```
% initial conditions are exogenous
up(L,0) :- not -up(L,0).
-up(L,0) :- not up(L,0).
open(0) :- not -open(0).
-open(0) :- not open(0).

% actions are exogenous
toggle(L,T) :- not -toggle(L,T).
-toggle(L,T) :- not toggle(L,T).
```

Figure 3.6: File suitcase_domain.b for CCALC, part 2.

```
:- macros lasttime -> 2.

:- include 'suitcase_domain.b'.

% initial conditions and the goal
:- not -up(l1,0).
:- not -up(l2,0).
:- not -open(0).
:- not open(lasttime).
:- not -up(l1,lasttime).
:- not -up(l2,lasttime).
```

Figure 3.7: File suitcase_problem.b for CCALC.

```
loadf('suitcase_problem.b').
```

This command turns the logic program presented in Figures 3.5–3.7 into a propositional theory by the process of literal completion (Section 3.4.3). Solutions to the suitcase problem correspond to satisfying interpretations for that propositional theory, and such a solution can be found by typing

```
sat.
```

By this command, CCALC calls a propositional solver (in this case it calls the default solver SATO) to find a satisfying interpretation of the propositional theory it obtained from its input. Using a propositional solver to generate a plan is known as "satisfiability planning" [Kautz and Selman, 1992].

The union of the programs presented in these figures is different from the program presented to DLV, presented in Figure 3.1, in following ways. The auxiliary predicates `latch` and `time` are turned into "sorts". The sort `step` is declared to be a "subsort" of the sort `time`, i.e., every step is a time. The variable `L` ranges over the latches `l1` and `l2`; the variable `T` ranges over the time instants `0,...,lasttime-1`; `T1` ranges over the time instants `0..lasttime`. Here, `lasttime` is declared as a macro in the problem description in Figure 3.7. The use of sorted variables allows us to do without the predicates `latch` and `time`. The auxiliary predicate `next(T,T1)` is declared as a macro, which is expanded into the atomic formula `T1 = T+1`. The disjunctive rules of the program in Figure 3.1 that define the exogenous atoms are replaced by nondisjunctive rules in Figure 3.6 as explained above.

## 3.6 Blocks World Problems

The suitcase problem was interesting in that it shows how answer set programming solves the ramification problem. However, it is computationally trivial. Therefore, we consider larger planning problems in the blocks world domain.

In the blocks world domain, we have blocks $B_1, B_2, \ldots, B_n$ on a table arranged in several towers. The fluent $on(b, l)$ expresses that block $b$ is at location $l$, where $l$ can be a block or the table. There is an action of moving a block $b$ onto a location $l$ denoted by $move(b, l)$. In this domain, the actions are not allowed to occur concurrently. Here is a small blocks world problem, called the Sussman anomaly [Sussman, 1990]: given

```
Initial State:                  Goal State:


                                   B2
  B2                               B1
  B0     B1                        B0
------------                   -------------
```

find a series of actions to reach the goal state from the initial state.

We can describe the blocks world domain by the following logic program. In the rules below,

- $b, b', b''$ range over a finite set of block constants,
- $l, l'$ range over the set of location constants that consists of the block constants and the constant *table*,
- $t$ ranges over the symbols representing an initial segment of integers $0, \ldots, T_{max}$,

except that in atoms of the form $move(b, l, t)$ we require $t < T_{max}$. The program consists of the following rules.

Both the initial values and the occurrences of actions are exogenous:

$$\begin{aligned} &on(b, l, 0); \neg on(b, l, 0) \\ &move(b, l, t); \neg move(b, l, t). \end{aligned} \tag{3.20}$$

The following rule describes the effect of moving a block:

$$on(b, l, t+1) \leftarrow move(b, l, t). \tag{3.21}$$

The commonsense law of inertia is postulated in the form:

$$on(b, l, t+1) \leftarrow on(b, l, t), not\ \neg on(b, l, t+1). \tag{3.22}$$

Wherever a block is, it is not anywhere else:

$$\neg on(b, l', t) \leftarrow on(b, l, t) \qquad (l \neq l'). \tag{3.23}$$

No two blocks can be on the same block at the same time:

$$\bot \leftarrow on(b'', b, t), on(b', b, t) \qquad (b' \neq b''). \tag{3.24}$$

A block can be moved only when it's clear:

$$\bot \leftarrow move(b,l,t), on(b',b,t). \tag{3.25}$$

Actions cannot be executed concurrently:

$$\bot \leftarrow move(b,l,t), move(b',l',t) \qquad (b \neq b' \text{ or } l \neq l'). \tag{3.26}$$

This formalization of the blocks world domain is presented to DLV in essentially the same way as the suitcase domain (3.10) ∪ (3.11) is presented to DLV in Section 3.3.3.

In our experiments with SMODELS, we expressed the first rule of (3.20) by an exclusive disjunctive rule, and the second rule of (3.20) by a choice rule as in Figure 3.3. The no-concurrency constraint (3.26) is replaced by

$$:-2\{\texttt{move(B,L,T)} : \texttt{block(B)} : \texttt{location(L)}\}, \texttt{time(T)}. \tag{3.27}$$

The expression of the form $2\{\dots\}$ in the body of this rule is a cardinality constraint (see Section 2.3.2). It expresses that the cardinality of the set $\{\dots\}$ is at least 2. Consequently, rule (3.27) eliminates the answer sets that contain 2 or more ground instances of $move(b,l,t)$ for some time $t$. We turned constraint (3.24) into a constraint similar to (3.27).

How can we use propositional solvers to find answer sets for the blocks world program? As observed in Section 3.5, the disjunctive rules defining exogenous atoms can be replaced by nondisjunctive rules. For instance, we can express that the initial values of the fluents and the occurrences and the non-occurrences of the actions are exogenous (3.20) by the following nondisjunctive rules:

$$
\begin{aligned}
&on(b,l,0) \leftarrow not\ \neg on(b,l,0)\\
&\neg on(b,l,0) \leftarrow not\ on(b,l,0)\\
&move(b,l,t) \leftarrow not\ \neg move(b,l,t)\\
&\neg move(b,l,t) \leftarrow not\ move(b,l,t).
\end{aligned}
\tag{3.28}
$$

In the presence of rule (3.23), the second rule above turns out to be redundant.

Consider program (3.21)–(3.26) $\cup$ (3.28). We would like to find this program's answer sets using CCALC. To justify the applicability of this method, we would need to check that the program is tight (Section 3.4.2), which is not difficult. This allowed us to experiment with CCALC on the blocks world problems. We presented the blocks world domain to CCALC in essentially in the same way as we presented the suitcase domain to CCALC in Section 3.5.

DLV, unlike CCALC and SMODELS, use the general optimization technique described in [Faber *et al.*, 1999] to produce a smaller ground program. For instance, with this technique, no-concurrency constraint (3.26) are replaced by

$$
\begin{aligned}
&move_1(b,t) \leftarrow move(b,l,t)\\
&\leftarrow move_1(b,t), move_1(b',t) \qquad (b \neq b')\\
&move_2(l,t) \leftarrow move(b,l,t)\\
&\leftarrow move_2(l,t), move_2(l',t) \qquad (l \neq l').
\end{aligned}
\tag{3.29}
$$

Since the other rules of the program make it impossible to move a block to two different places at the same time, the rules involving $move_2$ are actually redundant, and the no-concurrency constraint can be expressed by the rules in the first and the second lines. This representation of no-concurrency in the blocks world is used in [Niemelä, 1999] (see Section 5.2).

There is a 1-1 correspondence between the answer sets for a program $\Pi$ with constraints (3.26) and the answer sets for $\Pi$ with rules (3.29). This is proven in [Erdem and Lifschitz, 1999] and will be discussed in Section 4.2. This allowed us to experiment with CCALC using both original programs (containing (3.26)) and optimized programs (containing the first two of rules (3.29)).

Using rules (3.29) instead of constraints (3.26) has a significant effect on the size of the program after grounding. For the original program, this size grows as $n^4$ with the number $n$ of blocks; for the modified program, it grows as $n^3$.

To describe, for instance, the Sussman anomaly we consider the blocks $B_0$, $B_1$ and $B_2$, and we specify $T_{max}$ to be 3. The problem can be described as the problem of finding an answer set that contains

$$on(B_1, Table, 0), on(B_2, B_0, 0), on(B_0, Table, 0),$$

$$on(B_0, Table, 3), on(B_1, B_0, 3), on(B_2, B_1, 3).$$

Besides the Sussman anomaly, we experimented with the blocks world problems presented in Figure 3.8.

**P1**

```
Initial State:                Goal State:

                                  B3
                                  B2
             B3                   B1
 B0    B1    B2                   B0
-----------------             -------------
```

**P2**

```
Initial State:                Goal State:

                                  B4
                                  B3
         B0                       B2
   B1    B3                       B1
   B4    B2                       B0
  -------------               -------------
```

**P3**

```
Initial State:                Goal State:

 B2          B7                   B7          B5
 B3    B4    B6                   B3    B2    B0
 B0    B1    B5                   B4    B6    B1
------------------              -------------------
```

**P4**

```
Initial State:                Goal State:

                                              B1
       B10   B8                               B2
 B2    B9    B7                   B0    B7    B10
 B1    B4    B6                   B4    B8    B6
 B0    B3    B5                   B9    B3    B5
------------------              --------------------
```

Figure 3.8: The blocks world problems we experimented with.

## 3.7 Experimental Evaluation of DLV, SMODELS and CCALC

The table in Figure 3.9 summarizes the duration of the computations (in CPU seconds) for DLV, SMODELS and CCALC applied to 5 blocks world problems.[4] The first column of this table shows the problems used in these experiments. The problem labeled P0 is the Sussman anomaly problem. The problems labeled P1–P4 are described in Figure 3.8. The number of blocks, for each problem, is presented in the next column. The third column shows, for each problem, the length of the shortest plan. (We can verify that a plan computed by an answer set solver is the shortest possible by making the value of $T_{max}$ smaller and verifying that the resulting program has no answer sets.) The computation times of DLV are presented in the next column. Then, we show, for each problem, the timing for LPARSE and the timing for SMODELS. As for CCALC, we introduce the computation times with the original program (3.21)–(3.26) $\cup$ (3.28), and with the optimized program obtained by replacing (3.26) with the first two of rules (3.29). In both cases, for each problem, we present the timings for CCALC and for CHAFF.

If we look at the time spent at each phase of the computation, we can see that CCALC spends most of the elapsed time for transforming its input into

[4]We used CCALC 1.9 with CHAFF as implemented by Matthew Moskewicz; SMODELS 2.26 with LPARSE 1.0.4; DLV released October 2000. The experiments are performed on an Ultra 5/10 with 120 MB main memory and a 360 MHz SUNW, UltraSPARC-IIi CPU.

| Pblms | number of blocks | length of plan | DLV | LPARSE + SMODELS | CCALC + CHAFF | |
|---|---|---|---|---|---|---|
| | | | | | original | optimized |
| P0 | 3 | 3 | 0.09 | 0.08 + 0.06 | 0.38 + 0.02 | 0.25 + 0.02 |
| P1 | 4 | 4 | 0.16 | 0.12 + 0.13 | 1.08 + 0.04 | 0.63 + 0.02 |
| P2 | 5 | 6 | 0.52 | 0.23 + 0.28 | 3.22 + 0.13 | 1.5 + 0.05 |
| P3 | 8 | 8 | 81.52 | 0.84 + 7.75 | 22.43 + 0.91 | 6.49 + 0.35 |
| P4 | 11 | 9 | 329.81 | 2.16 + 19.57 | 76.16 + 3.76 | 17.04 + 1.35 |

Figure 3.9: Planning with blocks world problems using DLV, SMODELS, and CCALC.

an equivalent classical propositional theory while SMODELS spends most of the elapsed time for finding an answer set. For instance, for P4, CCALC spends 76.16 seconds for grounding, transforming the causal theory into propositional theory, and conversion to the clausal form, and 3.76 seconds in model-finding. If we measure the time spent in grounding and model-finding separately for SMODELS, we find out that SMODELS spends 2.16 seconds in grounding, and 19.57 seconds in model-finding for P4. SMODELS spends less time in grounding. One reason for this is that SMODELS makes use of the given query to produce a sufficient subset of the ground instances of the given program. Another reason is that it eliminates all of the domain predicates appearing in the ground program so that a smaller ground program is used to find the answer sets. This makes SMODELS more efficient in computing the answer sets after grounding. Also, it is important to keep in mind that grounding and conversion to clauses in CCALC are implemented in Prolog, and CHAFF, LPARSE, SMODELS and DLV are implemented in C++.

We do not have enough information about the time spent during each phase of computation in DLV.

As mentioned in the previous section, replacing no-concurrency constraints (3.26) by the first two rules of (3.26) reduces the size of the blocks world program after grounding. For instance, for P4, with the original program, CCALC produces a theory with 2508 atoms and 180378 clauses; with the optimized program, it produces a theory with 2607 atoms and 39006 clauses. As seen from the table above, this translation reduces the transformation time as well.

## 3.8 Comparisons with Related Work

In our experiments, we specified the planning problems with logic programs. As we emphasized earlier, our approach to logic programming is answer set programming. In the next subsection, we will compare our experiments with other experiments that apply answer set programming to planning. In the last subsection, we will compare two approaches, answer set programming and satisfiability checking, used to solve the planning problems.

### 3.8.1 Encodings of the Planning Problems

The earliest use of answer set programming for planning is described by Dimopoulos, Nebel, and Koehler [1997]. The encodings of planning problems used in [Lifschitz, 1999] and in this chapter are different from theirs in three ways.

First, in [Dimopoulos *et al.*, 1997], the authors start with planning problems described in STRIPS representation [Fikes and Nilsson, 1971]; we are interested in planning problems described by transition systems as in [Lifschitz, 1999].

Second, to encode the planning problems in a more compact and efficient way, they use linear encodings (introduced by Kautz and Selman [1992]), i.e., splitting the action predicates into a number of binary predicates. Our encodings are more natural and straightforward but less efficient.

The third difference is as follows. In some of the problems, the actions are allowed to be executed concurrently, as in the suitcase problem. In some other problems, such as the blocks world problems above, the parallel execution of actions is not allowed. In such problems, it may be easier to find a plan with concurrently executed actions and then "serialize" it. This is what Dimopoulos, Nebel, and Koehler do for planning in the blocks world. We do not use any post-processing methods to make planning more efficient. This method makes planning more efficient, but generally, sequential plans constructed in this way are longer than the shortest sequential plan.

### 3.8.2 Answer Set Planning vs. Satisfiability Planning

In our approach to planning, a plan corresponds to an answer set for the logic program representation of the planning problem. In satisfiability planning [Kautz and Selman, 1992], a plan corresponds to a model satisfying a set of propositional formulas describing the planning problem.

Using CCALC for planning can be viewed both as a form of answer set programming—the input of CCALC is a logic program—and, at the same time, as a form of satisfiability planning—an answer set is computed by converting the program into a set of propositional formulas.

## 3.9 Eliminating Circular Configurations of Blocks

In Section 3.6, we described the blocks world domain by rules (3.20)–(3.26). After constraints describing an initial configuration of blocks are added to these rules, we get a program whose answer sets are in a 1-1 correspondence with the histories of the blocks world that start with this initial configuration.

Unfortunately, this does not hold anymore if we allow concurrent execution of actions. If we drop no-concurrency constraint (3.26) then, for instance, moving block $a$ onto block $b$ and moving block $b$ onto block $a$ at the same time may be possible, and this would lead to a physically impossible, circular configuration of blocks.

To overcome this difficulty, we need to make sure that every block is supported by the table at every time instant. For that, we introduce the atom $supported(b,t)$: "block $b$ is supported by the table at time $t$". Then, we describe that every block is *supported* by the table:

$$\begin{array}{ll} supported(b,t) \leftarrow on(b, Table, t), & \\ supported(b,t) \leftarrow on(b,b',t), supported(b',t) \qquad (b \neq b'), & \qquad (3.30) \\ \bot \leftarrow not\ supported(b,t). & \end{array}$$

Adding rules (3.30) to the blocks world programs presented to DLV and SMODELS would handle the problem above.

Unfortunately, this method does not work for CCALC: program (3.21)–(3.26) $\cup$ (3.28) $\cup$ (3.30) is not tight, and the completion of this program has models that correspond to circular configurations of blocks. A way to overcome this difficulty is discussed in Chapter 6 of this proposal. The idea is to use the transitive closure *above* of the fluent *on*, instead of the predicate *supported*. "Bad" models of the program's completion can be eliminated by replacing rules (3.30) with the rules

$$\begin{aligned} above(b,l,t) &\leftarrow on(b,l,t) \\ above(b,l,t) &\leftarrow on(b,b',t), above(b',l,t) \end{aligned} \tag{3.31}$$

$$\bot \leftarrow above(b,b,t) \tag{3.32}$$

$$\bot \leftarrow not\ \ above(b, Table, t). \tag{3.33}$$

Atom $above(b,l,t)$ expresses that block $b$ is above location $l$ at time $t$. These atoms are used to express constraint (3.33) that requires every block to be "supported by the table".[5]

In [Erdem and Lifschitz, 2001b], we showed that, for the program modified in this way, the models of completion are also its answer sets. This is discussed in Chapter 6.

With the modified program, the computation times for DLV improve: for P4, it takes 117.42 seconds. The timings for SMODELS do not change much:

[5]Rules (3.31) and (3.32) were suggested to us by Norman McCain and Hudson Turner on June 11, 1999; similar rules are discussed in [Lifschitz, 1999, Section 8].

for P4, LPARSE takes 2.44 seconds and SMODELS takes 19.49 seconds. As for CCALC, both the transformation time and the model-finding time increase slightly: for P4, CCALC takes 83.08 seconds, and CHAFF takes 5.27 seconds.

## 3.10 Next Three Chapters

In this chapter, we discussed our experiments with some planning problems using the systems DLV, SMODELS, and CCALC. These experiments motivated us to do the theoretical work presented in the next three chapters. Here are the four main observations that led to this theoretical work .

**Defining exogenous atoms.** In Sections 3.3.3 and 3.6, we described the exogenous atoms, i.e., the atoms representing the initial values of fluents and the occurrences of actions, by disjunctive rules to (see for instance, rules (3.20)). To present this definition to CCALC, we had to transform these disjunctive rules into nondisjunctive rules (see for instance, rules (3.28)). Our Theorem 1 justifies this transformation: it shows that this transformation preserves the answer sets for a program.

**No-concurrency constraints.** In Section 3.6, we expressed that no two actions are allowed to occur at the same time by constraint (3.26). Including these constraints has a significant effect on the size of a program after grounding. For a blocks world problem with $n$ blocks, this size grows as $n^4$. To make the grounding more efficient we replaced (3.26) by rules (3.29). With this transformation the size of a program after grounding grows as $n^3$. Our Theorem 2 justifies this transformation: it shows that replacing (3.26) by rules (3.29) may extend the answer sets by new literals, but the parts of the answer sets that belong to the language of the original program remain the same.

**Transitive closure.** In Section 3.6, we observed that the completion of the blocks world program (3.21)–(3.26) with (3.28) and (3.30), after eliminating classical negation, has models that do not correspond to valid configurations of blocks. We also observed that we can overcome this difficulty by defining the transitive closure *above* of the fluent *on* by rules (3.31), and adding constraints (3.32) and (3.33). Our Theorem 4 shows that, the definition of *above* with rules (3.31) is correct relative to answer set semantics. Our Theorem 5 shows that, with this modification, the models of completion of this program are identical to its answer sets.

**Generalization of Fages' Theorem.** Usually, to show that the models of completion of a program are also its answer sets, we use Fages' theorem, as mentioned in Section 3.4.2. We noticed that Fages' theorem is not applicable to the blocks world program where we define the transitive closure *above* of the fluent *on* because it is not tight. For that we defined "tightness on a set of literals", and generalized Fages' theorem accordingly (see Theorem 3). Our Theorem 3 is used to prove Theorem 5.

# Chapter 4

# Transformations of Logic Programs related to Planning

In this chapter we state two properties of logic programs under the answer set semantics [Gelfond and Lifschitz, 1991] that may be useful in connection with applications of logic programming to planning.

According to the first of the two theorems, replacing a disjunctive rule of the form (2.6) (see page 16) by nondisjunctive rules (3.19) (see page 48) is an equivalent transformation, as far as consistent answer sets are concerned. Under some conditions, this fact follows from [Ben-Eliyahu and Dechter, 1994]; our Theorem 1 is more general.

The second theorem introduced in this chapter has to do with expressing uniqueness assumptions in logic programming. In a language containing a binary predicate constant $p$ and no function constants, the constraints

$$\leftarrow p(c,d), p(c',d') \qquad (c \neq c' \text{ or } d \neq d') \tag{4.1}$$

($c, d, c', d'$ range over the object constants) eliminate the answer sets that contain more than one atom of the form $p(c,d)$. Essentially the same result can be achieved using the rules

$$\begin{aligned}
&p_1(c) \leftarrow p(c,d),\\
&p_2(d) \leftarrow p(c,d),\\
&\leftarrow p_1(c), p_1(c') \qquad (c \neq c'),\\
&\leftarrow p_2(d), p_2(d') \qquad (d \neq d'),
\end{aligned}$$

where $p_1, p_2$ are auxiliary predicates. Theorem 2 is a generalization of this fact. Like Theorem 1, it is related to applications of answer set solvers to planning. When the goal is to find a plan in which actions are executed sequentially, the logic programming representation of the problem has to contain a no-concurrency constraint similar to (4.1). The equivalent transformation provided by Theorem 2 may allow us to state this constraint in a way that provides some computational advantages, as discussed in Section 3.7.

## 4.1 Theorem 1

**Theorem 1** *For any program $\Pi$ and any atom $p$, the programs*

$$\begin{aligned}
&\Pi\\
&p; \neg p \leftarrow
\end{aligned} \tag{4.2}$$

*and*

$$
\begin{array}{l}
\Pi \\
p \leftarrow not\ \neg p \\
\neg p \leftarrow not\ p
\end{array}
\tag{4.3}
$$

*have the same answer sets.*

Both the theorem and its proof can be extended to programs with negation as failure allowed in the heads of rules [Lifschitz, 1996]. The proof of Theorem 1, given in [Erdem and Lifschitz, 1999], is presented in Section 9.3 (page 164).

## 4.2 Theorem 2

In the statement of Theorem 2, $\Pi$ is a program, $C_1, \ldots, C_n$ $(n > 0)$ are sets, and $p$ is a function such that for all $c_1 \in C_1, \ldots, c_n \in C_n$ its values $p(c_1, \ldots, c_n)$ are pairwise distinct atoms in the language of $\Pi$. The expressions $p_i(c)$, where $1 \leq i \leq n$ and $c \in C_i$, are assumed to be pairwise distinct atoms that do not belong to the language of $\Pi$.

**Theorem 2** *If $X$ is an answer set for the program $\Pi_1$ obtained from $\Pi$ by adding the rules*

$$p_i(c_i) \leftarrow p(c_1, \ldots, c_n) \tag{4.4}$$

$$\leftarrow p_i(c), p_i(c') \tag{4.5}$$

*($1 \leq i \leq n,\ c_1 \in C_1, \dots, c_n \in C_n,\ c, c' \in C_i$) then*

$$X \cap \mathit{lit}(\Pi) \tag{4.6}$$

*is an answer set for the program $\Pi_2$ obtained from $\Pi$ by adding the rules*

$$\begin{array}{l} \leftarrow p(c_1, \dots, c_n), p(c'_1, \dots, c'_n) \\ \qquad\qquad (c_1, c'_1 \in C_1,\ \dots, c_n, c'_n \in C_n, \langle c_1, \dots, c_n \rangle \neq \langle c'_1, \dots, c'_n \rangle). \end{array} \tag{4.7}$$

*Moreover, every answer set for $\Pi_2$ can be represented in form (4.6) for some answer set $X$ for $\Pi_1$.*

The theorem asserts, in other words, that replacing constraints (4.7) with rules (4.4) and (4.5) may extend the answer sets by new literals $p_i(c)$, $\neg p_i(c)$, but the parts of the answer sets that belong to the language of $\Pi$ remain the same.

To apply Theorem 2 to the blocks world example (Section 3.7), we use it to replace the no-concurrency constraints by the rules involving $\mathit{move}_1$ and $\mathit{move}_2$ consecutively for $T = 0$, $T = 1$, etc., each time with

- $n = 2$,
- $C_1$ equal to the set of block constants,
- $C_2$ equal to the set of location constants,
- $p(c_1, c_2)$ equal to $\mathit{move}(c_1, c_2, T)$.

The proof of Theorem 2 is based on two facts. One is the splitting set theorem [Lifschitz and Turner, 1994].

The other is a property of constraints that easily follows from the definition of an answer set: the effect of adding a set of constraints to a program is to eliminate the answer sets that are not closed under these constraints.

The proof of Theorem 2, given in [Erdem and Lifschitz, 1999], is presented in Section 9.4 (page 167).

## 4.3 Related Work

Gelfond *et al.* [1991] compare a disjunctive logic program $\Pi$ with the nondisjunctive program $\Pi'$ obtained by replacing each rule of form (3.2) by $k$ rules:

$$
\begin{array}{l}
L_1 \leftarrow L_{k+1}, \dots, L_m, not\ L_{m+1}, \dots, not\ L_n, not\ L_2, \dots, not\ L_k \\
\vdots \\
L_k \leftarrow L_{k+1}, \dots, L_m, not\ L_{m+1}, \dots, not\ L_n, not\ L_1, \dots, not\ L_{k-1}
\end{array}
$$

Here each $L_i$ is a literal, i.e., an atom possibly preceded by classical negation, and $0 \leq k \leq m \leq n$. They show that each answer set for $\Pi'$ is also an answer set for $\Pi$. Ben-Eliyahu and Dechter [1994] show that $\Pi$ is equivalent to $\Pi'$ if $\Pi$ is "head-cycle free". Their proof is based on translating $\Pi$ and $\Pi'$ into propositional logic; for the two programs, the results happen to be the same.

Our Theorem 1 is different from that of [Ben-Eliyahu and Dechter, 1994] in two ways. First, it allows us to replace a single disjunctive rule (2.6)—and consequently any finite set of rules of this form—by nondisjunctive rules. This is not the same as completely eliminating all disjunctive rules from a program. Second, the disjunctive programs we consider are not required to be "head-cycle free".

Consider, for instance, the program

$$p; q \leftarrow \tag{4.8}$$

$$p; \neg p \leftarrow \tag{4.9}$$

$$p \leftarrow q \tag{4.10}$$

$$q \leftarrow p. \tag{4.11}$$

By Theorem 1, we can replace rule (4.9) by nondisjunctive rules (3.19) and leave rule (4.8) as it is. The theorem from [Ben-Eliyahu and Dechter, 1994] would not allow us to justify this replacement. It is not applicable to program (4.8)–(4.11) because this program is not "head-cycle free": its dependency graph has a cycle that goes through the atoms $p$ and $q$ that belong to the head of the same disjunctive rule.

On the other hand, our theorem is not applicable to head-cycle free disjunctive programs that do not contain any rule of the form (4.2).

After transforming programs (4.2) and (4.3) into programs without classical negation [Gelfond and Lifschitz, 1991], as explained in Section 2.2, Theorem 1 can be derived from Theorem 2 of [Lifschitz *et al.*, 2001]. According to Theorem 2 of [Lifschitz *et al.*, 2001], we need to show that the formulas

$$p \vee p'$$

$$\neg(p, p')$$

are equivalent to the formulas

$$\neg p' \supset p$$

$$\neg p \supset p'$$

$$\neg(p, p')$$

in the "logic of here-and-there", which is a monotonic logic that is intermediate between classical logic and intuitionistic logic [Gödel, 1932].

Theorem 2 is similar to "action splitting" in satisfiability planning [Kautz and Selman, 1996]. In "action splitting", auxiliary atoms similar to $p_i(c)$ from our Theorem 2 are sometimes used to eliminate action symbols altogether.

# Chapter 5

# A Generalization of Fages' Theorem

This chapter is about the relationship between the completion semantics [Clark, 1978] and the answer set (stable model) semantics [Gelfond and Lifschitz, 1991] for logic programs with negation as failure. The study of this relationship is important in connection with the emergence of answer set programming. Whenever the two semantics are equivalent, answer sets can be computed by a propositional solver, and the use of answer set solvers such as SMODELS and DLV is unnecessary.

Consider a finite propositional (or grounded) program $\Pi$ without classical negation, and a set $X$ of atoms. As mentioned in Section 3.4, if $X$ is an answer set for $\Pi$ then $X$, viewed as a truth assignment, satisfies the completion of $\Pi$. The converse, generally, is not true, as illustrated by program

(3.16) introduced in Section 3.4 (see page 45). As mentioned in Section 3.4.2, François Fages [1994] defined tightness, and proved that, for a tight program, every model of completion is an answer set. Thus, for tight programs, the completion semantics and the answer set semantics are equivalent.

Our generalization of Fages' theorem allows us to draw similar conclusions for some programs that are not tight. Here is one such program:

$$\begin{aligned} &p \leftarrow not\ q, \\ &q \leftarrow not\ p, \\ &p \leftarrow p, r. \end{aligned} \tag{5.1}$$

It is not tight. Nevertheless, each of the two models $\{p\}$, $\{q\}$ of its completion

$$\begin{aligned} &p \equiv \neg q \vee (p \wedge r), \\ &q \equiv \neg p, \\ &r \equiv \bot \end{aligned}$$

is an answer set for (5.1).

The idea of this generalization is to make the function $\lambda$ from the definition of tightness (Section 3.4.2) partial. Instead of tight programs, we will consider programs that are "tight on a set of literals."

In the following, answer sets are first related to a model-theoretic counterpart of completion introduced in [Apt *et al.*, 1988], called supportedness. This allows us to make the theorem applicable to programs with both negation as failure and classical negation, and to programs with infinitely many rules. Then, a corollary about completion is derived, and it is applied to a logic programming representation of the blocks world due to Ilkka Niemelä. Next, we show how the satisfiability solver SATO [Zhang, 1997] can be used to find answer sets for that representation, and compare the performance of SMODELS and SATO on several benchmarks. After that, we comment on the limitations of this method.

## 5.1 Generalized Fages' Theorem

For any program $\Pi$ and any consistent set $X$ of literals, we say that $X$ is *supported by* $\Pi$ if, for every $L \in X$, there is a rule (3.2) in $\Pi$ such that (3.4), (3.5), and

$$\{L_1, \ldots, L_k\} \cap X = L. \tag{5.2}$$

For instance, the set $\{p\}$ is supported by program (2.3) (see page 11) whereas the set $\{p, q\}$ is not. Note that, for any program $\Pi$, the empty set is supported by $\Pi$.

The definition of supportedness above corresponds to the definition of supportedness introduced in [Apt *et al.*, 1988] for nondisjunctive programs without classical negation, and corresponds to the definition of supportedness introduced in [Lifschitz and Turner, 1999] for nondisjunctive programs possibly with classical negation. Our definition of supportedness conjoined with our definition of closure (Section 3.1) corresponds to the definition of supportedness introduced in [Baral and Gelfond, 1994].

In this chapter, we consider nondisjunctive programs, that is, programs consisting of rules of the form (3.17) (page 45).

Instead of "level mappings" used in Section 3.4.2, we consider here *partial level mappings*—partial functions from literals to ordinals. A program $\Pi$ is *tight* on a set $X$ of literals if there exists a function $\lambda$ with the domain $X$ such that, for every rule (3.17) in $\Pi$, if $Head, L_1, \ldots, L_m \in X$ then

$$\lambda(L_1), \ldots, \lambda(L_m) < \lambda(Head).$$

For instance, program (5.1) is tight on $\{p, q\}$; on the other hand, it is not tight on $\{p, q, r\}$.

The proposition below gives a simple characterization of tightness on a set of literals that does not refer to ordinals. For any program $\Pi$ and any set $X$ of literals, we say about literals $L, L' \in X$ that $L$ is a *parent* of $L'$ relative to $\Pi$ and $X$ if there is a rule (3.17) in $\Pi$ such that

- $L_1, \ldots, L_m \in X$,
- $L \in \{L_1, \ldots, L_m\}$, and
- $L' = Head$.

For instance, the parents of $p$ relative to (5.1) and $\{p, q, r\}$ are $p$ and $r$; on the other hand, $p$ has no parents relative to (5.1) and $\{p, q\}$.

**Proposition 2** *A program $\Pi$ is tight on a set $X$ of literals iff there is no infinite sequence $L_0, L_1, \ldots$ of elements of $X$ such that for every $i$, $L_{i+1}$ is a parent of $L_i$ relative to $\Pi$ and $X$.*

In other words, $\Pi$ is tight on a set $X$ iff the parent relation relative to $\Pi$ and $X$ is well-founded. Proposition 2 is a special case of Lemma 1 (take the domain of $R$ to be $X$); Proposition 1 is a special case of Proposition 2 (take $X$ to be $lit(\Pi)$).

Our generalization of Fages' theorem is stated as follows:

**Theorem 3** *For any program $\Pi$ and any consistent set $X$ of literals such that $\Pi$ is tight on $X$, $X$ is an answer set for $\Pi$ iff $X$ is closed under and supported by $\Pi$.*

The proof of Theorem 3, given in [Babovich *et al.*, 2000], is presented in Section 9.6 (page 170).

In the special case when $\Pi$ is a finite program without classical negation, a set of atoms satisfies the completion of $\Pi$ iff it is closed under and supported by $\Pi$. We conclude:

**Corollary 1** *For any finite program $\Pi$ without classical negation and any set $X$ of atoms such that $\Pi$ is tight on $X$, $X$ is an answer set for $\Pi$ iff $X$ satisfies the completion of $\Pi$.*

For instance, program (5.1) is tight on the model $\{p\}$ of its completion: take $\lambda(p) = 0$. By Corollary 1, it follows that $\{p\}$ is an answer set for (5.1). In a similar way, the theorem shows that $\{q\}$ is an answer set also.

By $pos(\Pi)$ we denote the set of all literals that occur without negation as failure at least once in the body of a rule of $\Pi$.

**Corollary 2** *For any program* $\Pi$ *and any consistent set* $X$ *of literals disjoint from* $pos(\Pi)$*,* $X$ *is an answer set for* $\Pi$ *iff* $X$ *is closed under and supported by* $\Pi$*.*

**Corollary 3** *For any finite program* $\Pi$ *without classical negation and any set* $X$ *of atoms disjoint from* $pos(\Pi)$*,* $X$ *is an answer set for* $\Pi$ *iff* $X$ *satisfies the completion of* $\Pi$*.*

To derive Corollary 2 from Theorem 3, and Corollary 3 from Corollary 1, take $\lambda(L) = 0$ for every $L \in X$.

Consider, for instance, the program

$$\begin{aligned} &p \leftarrow not\ q, \\ &q \leftarrow not\ p, \\ &r \leftarrow r, \\ &p \leftarrow r. \end{aligned} \tag{5.3}$$

The completion of (5.3) is

$$\begin{aligned} &p \equiv \neg q \vee r, \\ &q \equiv \neg p, \\ &r \equiv r. \end{aligned}$$

The models of these formulas are $\{p\}$, $\{q\}$ and $\{p, r\}$. The only literal occurring in the bodies of the rules of (5.3) without negation as failure is $r$. In accordance with Corollary 3, the models of the completion that do not contain $r$—sets $\{p\}$ and $\{q\}$—are answer sets for (5.3).

## 5.2 Planning in the Blocks World

As a more interesting example, consider a logic programming encoding of the blocks world due to Ilkka Niemelä. This program is similar to the program presented in [Dimopoulos *et al.*, 1997]. The main part of the encoding consists of the following schematic rules:

```
goal :- time(T), goal(T).
:- not goal.

goal(T2) :- nextstate(T2,T1), goal(T1).

moveop(X,Y,T):-
   time(T), block(X), object(Y), X != Y,
   on_something(X,T), available(Y,T),
   not covered(X,T), not covered(Y,T),
   not blocked_move(X,Y,T).

on(X,Y,T2) :-
   block(X), object(Y), nextstate(T2,T1),
   moveop(X,Y,T1).

on_something(X,T) :-
   block(X), object(Z), time(T), on(X,Z,T).
```

```
available(table,T) :- time(T).

available(X,T) :-
   block(X), time(T), on_something(X,T).

covered(X,T) :-
   block(Z), block(X), time(T), on(Z,X,T).

on(X,Y,T2) :-
   nextstate(T2,T1), block(X), object(Y),
   on(X,Y,T1), not moving(X,T1).

moving(X,T) :- time(T), block(X), object(Y),
   moveop(X,Y,T).

blocked_move(X,Y,T):-
   block(X), object(Y), time(T), goal(T).

blocked_move(X,Y,T) :-
   time(T), block(X), object(Y),
   not moveop(X,Y,T).
```

```
blocked_move(X,Y,T) :-
   block(X), object(Y), object(Z), time(T),
   moveop(X,Z,T), Y != Z.

blocked_move(X,Y,T) :-
   block(X), object(Y), time(T), moving(Y,T).

blocked_move(X,Y,T) :-
   block(X), block(Y), block(Z), time(T),
   moveop(Z,Y,T), X != Z.

:- block(X), time(T), moveop(X,table,T),
   on(X,table,T).

:- nextstate(T2,T1), block(X), object(Y),
   moveop(X,Y,T1), moveop(X,table,T2).

nextstate(Y,X) :- time(X), time(Y),
   Y = X + 1.

object(table).
object(X) :- block(X).
```

To solve a planning problem, we combine the domain description with

(i) a set of facts defining `time/1` as an initial segment of nonnegative integers, for instance

```
time(0).   time(1).   time(2).
```

(ii) a set of facts defining `block/1`, such as

```
block(a).  block(b).  block(c).
```

(iii) a set of facts encoding the initial state, such as

```
on(a,b,0).  on(b,table,0).
```

(iv) a rule that encodes the goal, such as

```
goal(T) :- time(T), on(a,b,T), on(b,c,T).
```

The union is given as input to the "intelligent grounding" program LPARSE, and the result of grounding is passed on to SMODELS [Niemelä, 1999, Section 7], as seen in Section 3.3.4. The answer sets for the program correspond to valid plans.

Concurrently executed actions are allowed in this formalization as long as their effects are not in conflict, so that the plans described by this program are serializable (see Section 3.8.1).

The schematic rules above contain the variables `T`, `T1`, `T2`, `X`, `Y`, `Z` that range over the object constants occurring in the program, that is, over the nonnegative integers that occur in the definition of `time/1`, the names of blocks `a`, `b`,... that occur in the definition of `block/1`, and the object constant `table`.

The expressions in the bodies of the schematic rules that contain `=` and `!=` restrict the constants that are substituted for the variables in the process of grounding. For instance, we understand the schematic rule

```
nextstate(Y,X) :- time(X), time(Y), Y = X + 1.
```

as an abbreviation for the set of all ground instances of

```
nextstate(Y,X) :- time(X), time(Y).
```

in which `X` and `Y` are instantiated by a pair of consecutive integers. The schematic rule

```
blocked_move(X,Y,T) :-
   block(X), object(Y), object(Z), time(T),
   moveop(X,Z,T), Y != Z.
```

stands for the set of all ground instances of

```
blocked_move(X,Y,T) :-
   block(X), object(Y), object(Z), time(T),
   moveop(X,Z,T).
```

in which `Y` and `Z` are instantiated by different object constants.

According to this understanding of variables and "built-in predicates," Niemelä's schematic program, including rules (i)–(iv), is an abbreviation for a finite nondisjunctive program $BW$.

In the proposition below we assume that schematic rule (iv) has the form

```
goal(T) :- time(T), ...
```

where the dots stand for a list of schematic atoms with the predicate symbol `on` and the last argument `T`.

**Proposition 3** *Program $BW$ is tight on each of the models of its completion.*

The proof of Proposition 3, given in [Babovich *et al.*, 2000], is presented in Section 9.7 (page 172).

According to Corollary 1 from Section 5.1, we can conclude that the answer sets for program $BW$ can be equivalently characterized as the models of the completion of $BW$.

## 5.3 Answer Set Programming with CCALC and SATO

As discussed in Section 3.4.2, the equivalence of the completion semantics to the answer set semantics for program $BW$ shows that it is not necessary to use an answer set solver, such as SMODELS, to compute answer sets for $BW$; a propositional solver can be used instead.

We have conducted a series of experiments aimed at comparing the run times of SATO, when its input is generated from $BW$ by CCALC, with the run times of SMODELS, when its input is generated from $BW$ by LPARSE.

Since CCALC uses literal completion to compute the propositional theory corresponding to its input, it requires that the input program has complete answer sets (Section 3.4.3, page 47). Program BW does not have complete answers since it does not contain classical negation. For that, we add to BW the rule

$$\neg A \leftarrow not\ A$$

for every atom in BW. These rules define the closed world assumption.

Because the built-in arithmetic of CCALC is somewhat different from that of LPARSE, we had to modify $BW$ slightly. Our CCALC input file is presented in a similar way we present the suitcase domain. It uses variables of sorts `object`, `block` and `time` instead of the unary predicates with these names. The rules of $BW$ that contain those predicates in their bodies are modified accordingly. For instance, rule

```
on_something(X,T) :-
    block(X), object(Z), time(T), on(X,Z,T).
```

turns into

```
on_something(B1,T) :- on(B1,O2,T).
```

The macro expansion facility of CCALC expands

```
   nextstate(T2,T1)
```

into the expression

| Problem | Number of blocks | Number of steps | Run time of SMODELS | Run time of SATO |
|---|---|---|---|---|
| large.c | 15 | 7 | 5.22 | 1.90 |
| | | 8 | 22.58 | 2.42 |
| large.d | 17 | 8 | 10.37 | 10.17 |
| | | 9 | 44.61 | 4.80 |
| large.e | 19 | 9 | 15.46 | 13.26 |
| | | 10 | 75.77 | 12.53 |

Figure 5.1: Planning with $BW$: SATO vs. SMODELS

```
T2 is T1 + 1
```

that contains Prolog's built-in `is`.

Figure 5.1 shows the run times of SMODELS and SATO measured using the Unix `time` command, on the benchmarks from [Niemelä, 1999, Section 9, Table 3]. (See Footnote ([4]) on page 58 for the versions of these programs and the features of the machine.) For each problem, one of the two entries corresponds to the largest number of steps for which the problem is not solvable, and the other to the smallest number of steps for which a solution exists. The numbers do not include the grounding and completion times.

## 5.4 Further Generalization of Fages' Theorem

### 5.4.1 Weight Constraints and Nested Expressions

The generalization of Fages' theorem discussed above is not applicable to SMODELS programs with weight constraints (Section 2.3.2). According to [Ferraris and Lifschitz, 2001], weight constraints can be eliminated in favor of nested expressions (Section 2.2) in the sense of [Lifschitz *et al.*, 1999]. A generalization of the completion semantics to programs that contain such expressions was proposed by Lloyd and Topor [1984]. We have investigated how Fages' theorem on the relationship between completion and answer sets can be further extended to such more general programs.

Consider a simple example. Program

$$\begin{aligned} &p \leftarrow not\ not\ p, \\ &p \leftarrow p, q \end{aligned} \tag{5.4}$$

contains nested occurrences of negation as failure in the body of the first rule.[1] It belongs to the syntactic class for which our theorem guarantees the equivalence of the answer set semantics to the completion semantics. This program has two answer sets $\emptyset$, $\{p\}$; they are identical to the models of the completion of (5.4):

$$\begin{aligned} &p \equiv \neg\neg p \vee (p \wedge q) \\ &q \equiv \bot. \end{aligned}$$

We have extended, in [Erdem and Lifschitz, 2001a] and [Erdem and Lifschitz, 2002], the definition of tightness (Section 5.1) to programs with nested expressions to facilitate the verification of the equivalence between the answer set semantics and the completion semantics for programs like this.

[1]The double negation in the first rule of (5.4) is redundant from the point of view of the completion semantics, but it does affect the program's answer sets. On the other hand, the second rule is redundant from the point of view of the answer set semantics, but, generally, dropping a rule like this can change a program's completion in an essential way.

In the syntax of SMODELS, the first of rules (5.4) can be written as a choice rule

$$\{p\}\ .$$

The relationship between cardinality constraints and nested expressions is systematically studied in [Ferraris and Lifschitz, 2001].

The extended definition of tightness and the further generalization of Fages' theorem for programs with nested expressions can be found in [Erdem and Lifschitz, 2001a] and [Erdem and Lifschitz, 2002]. In the following we will present two examples where the further generalization of Fages' theorem allows us to use a propositional solver instead of an answer set solver. The first example is the New Year's Party Problem (Section 5.4.2) and the second example is the N-Queens Problem (Section 5.4.3).

### 5.4.2 Example: The New Year's Party Problem

The New Year's Party Problem was described by Vladimir Lifschitz on December 12, 2000 to Texas Action Group:[2]

> You are organizing a large New Year's Eve party. There will be $N$ tables in the room, with $M$ chairs around each table. You need to select a table for each of the guests, who are assigned numbers from 1 to $M \times N$, so that two conditions are satisfied. First, some guests like each other and want to sit together; accordingly, you

[2] `http://www.cs.utexas.edu/tag/` .

> are given a set $A$ of two-element subsets of $\{1, \dots, M \times N\}$, and, for every $\{i, j\}$ in $A$, guests $i$ and $j$ should be assigned the same table. Second, some guests dislike each other and want to sit at different tables; accordingly, you are given a set $B$ of two-element subsets of $\{1, \dots, M \times N\}$, and, for every $\{i, j\}$ in $B$, guests $i$ and $j$ should be assigned different tables. Your program should find such a seating arrangement or determine that it is impossible.

This problem can be described by the nested program below. In this program,

- $g, g'$ range over the integers in $\{1, \dots, M * N\}$ denoting guests,
- $t$ ranges over the integers in $\{1, \dots, N\}$ denoting tables, and
- $c, c'$ range over the integers in $\{1, \dots, M\}$ denoting chairs.

We start with an arbitrary arrangement.

$$assign\_table(g, t) \leftarrow not\ not\ assign\_table(g, t)$$
$$assign\_chair(g, c) \leftarrow not\ not\ assign\_chair(g, c)$$

Then we make sure that everyone sits in a chair at some table,

$$\leftarrow not\ (assign\_table(g, 1); \dots ; assign\_table(g, N))$$
$$\leftarrow not\ (assign\_chair(g, 1); \dots ; assign\_chair(g, M))$$

that no two guests sit in the same chair,

$$\begin{aligned}&\leftarrow assign\_table(g, t), assign\_table(g', t),\\ &\quad assign\_chair(g, c), assign\_chair(g', c) \qquad (g < g')\end{aligned}$$

that the guests who like each other sit at the same table,

$$\leftarrow not\ assign\_table(g,t), likes(g,g'), assign\_table(g',t)$$

and that the guests who do not like each other do not sit at the same table

$$\leftarrow assign\_table(g,t), dislikes(g,g'), assign\_table(g',t).$$

The further generalization of Fages' Theorem allows us to use a propositional solver to find the answer sets for the program above program.

### 5.4.3 Example: The N-Queens Problem

In this problem, the goal is to find a configuration of 8 queens on an $8 \times 8$ chessboard such that no queens can be taken by any other queen. In other words, no two queens may be on the same row, on the same column, or on the same diagonal.

The eight queens problem is presented to SMODELS as in Figure 5.2. The rule in the `GENERATE` section of the program instructs SMODELS to select atoms of the form $occupied(R,C)$ for including in an answer set in such a way that, for every column $C$, exactly one atom $occupied(R,C)$ be selected. The program has 92 answer sets, corresponding to the 92 possible arrangements of eight queens. Given this input file, SMODELS produces one of the solutions:

```
Stable Model: occupied(4,1) occupied(2,2) occupied(7,3)
occupied(5,4) occupied(1,5) occupied(8,6) occupied(6,7)
occupied(3,8)
```

```
% DOMAIN PREDICATES
row(1..8).
column(1..8).

% GENERATE
1{occupied(R,C):row(R)}1 :- column(C).

% TEST
:- occupied(R,C), occupied(R,C1),
   row(R), column(C), column(C1), C < C1.


:- occupied(R,C), occupied(R1,C1),
   row(R), column(C), row(R1), column(C1),
   C < C1, abs(R - R1) == abs(C - C1).
```

Figure 5.2: The eight queens problem presented to SMODELS

Consider the grounded version of the program in Figure 5.2, with the domain predicates *row* and *column* dropped from the rules:

$$1\{occupied(1, C), \ldots, occupied(8, C)\}1 \tag{5.5}$$

for all $C$ in $\{1, \ldots, 8\}$;

$$\leftarrow occupied(R, C), occupied(R, C1) \tag{5.6}$$

for all $R, C, C1$ in $\{1, \ldots, 8\}$ such that $C < C1$;

$$\leftarrow occupied(R, C), occupied(R1, C1) \tag{5.7}$$

for all $R, R1, C, C1$ in $\{1, \ldots, 8\}$ such that $C < C1$ and $|R - R1| = |C - C1|$.

Rewritten in terms of nested expressions, as described in [Ferraris and Lifschitz, 2001], rule (5.5) becomes

$$
\begin{array}{ll}
occupied(R, C) \leftarrow not\ not\ occupied(R, C) & \\
\leftarrow occupied(R, C), occupied(R1, C) \qquad (R < R1) & \qquad (5.8) \\
\leftarrow not\ occupied(R, C), \ldots, not\ occupied(R, C). &
\end{array}
$$

The first of these rules allows each of the atoms $occupied(R, C)$ to be included or not included in an answer set. The second rule prohibits the selections that include more than one atom $occupied(R, C)$ with the same value of $C$. The third rule prohibits the selections that include no such atoms for some value of $C$.

To sum up, the program from Figure 5.2 can be rewritten as the union of programs (5.8), (5.6) and (5.7).

We present the union of programs (5.8), (5.6) and (5.7) to CCALC as in Figure 5.3. CCALC produces the following output using SATO:

```
Satisfying Interpretation:  occupied(1,5)  occupied(2,8)
occupied(3,4) occupied(4,1)  occupied(5,3)  occupied(6,6)
occupied(7,2)  occupied(8,7)
```

According to the results of our experiments, excluding the preprocessing time, i.e., the time spent by LPARSE for grounding and the time spent by CCALC for grounding and transformation, SMODELS takes 0.06 seconds and SATO takes 0.01 seconds to find a solution to the eight queens problem. For 20 queens, SMODELS takes 200 seconds whereas SATO takes 0.08 seconds to find a

```
:- sorts
        row; column.

:- variables
        R,R1 :: row;
        C,C1 :: column.

:- constants
        1..8 :: row;
        1..8 :: column;
        occupied(row,column) :: cwAtomicFormula.

occupied(R,C) :- not (not occupied(R,C)).

:- occupied(R,C),occupied(R1,C), (R < R1).

:- (/\R: -occupied(R,C)).

:- occupied(R,C), occupied(R,C1), (C < C1).

:- occupied(R,C), occupied(R1,C1), (C < C1),
   abs(R-R1) =:= abs(C-C1).
```

Figure 5.3: The eight queens problem presented to CCALC

solution. Our results show that, starting from an efficient SMODELS program, it may be faster to find the answer sets for this program by transforming it to a propositional theory if possible.[3] The further generalization of Fages' theorem verifies the use of SATO to compute solutions to the n-queens problem.

[3]The constraint logic programming system CLP [van Hentenryck, 1989] is computationally even more efficient. For instance, for 20 queens, CLP takes 0.01 seconds to find a solution. According to [Pelov *et al.*, 2000], the queens problem can be also solved quickly using the abductive logic programming system SLDNFAC [Denecker and Van Nuffelen, 1999].

## 5.5 Discussion

Fages' theorem, and its generalizations presented in Sections 5.1 and 5.4, allow us to compute answer sets for some programs by completing them and then calling a propositional solver. We showed that this method can be applied, for instance, to the representation of the blocks world proposed in [Niemelä, 1999]. This example shows that propositional solvers may serve as useful computational tools in answer set programming.

There are cases, on the other hand, when the completion method is not applicable. Consider computing Hamiltonian cycles in a directed graph [Marek and Truszczyński, 1999]. We combine the rules

```
in(U,V) :- edge(U,V), not out(U,V).
out(U,V) :- edge(U,V), not in(U,V).

:- in(U,V), in(U,W), V != W.
:- in(U,W), in(V,W), U != V.

reachable(V) :- in(v0,V).
reachable(V) :- reachable(U), in(U,V).

:- vertex(U), not reachable(U).
```

with a set of facts defining the vertices and edges of the graph; `v0` is assumed to be one of the vertices. The answer sets for the resulting program correspond to the Hamiltonian cycles. Generally, the completion of the program has models different from its answer sets. Take, for instance, the graph consisting of two disjoint loops:

```
vertex(v0).  vertex(v1).  edge(v0,v0).
edge(v1,v1).
```

This graph has no Hamiltonian cycles, and, accordingly, the corresponding program has no answer sets. But the set

```
vertex(v0), vertex(v1), edge(v0,v0),
edge(v1,v1), in(v0,v0), in(v1,v1),
reachable(v0), reachable(v1)
```

is a model of the program's completion.

# Chapter 6

# Transitive Closure, Answer Sets, and Completion

In logic programming, the transitive closure $tc$ of a binary predicate $p$ is usually defined by the rules

$$tc(x, y) \leftarrow p(x, y),$$

$$tc(x, y) \leftarrow p(x, v), tc(v, y).$$

If we combine this definition *Def* with any set $\Pi$ of facts (that is, ground atoms) defining $p$, and consider the minimal model of the resulting program, the extent of $tc$ in this model will be the transitive closure of the extent of $p$. In this sense, *Def* is a correct characterization of the concept of transitive closure. We know, on the other hand, that the completion of $\Pi \cup Def$ in the sense of Clark [1978] may have models different from the minimal model. In these "spurious" models of completion, $tc$ is weaker than the transitive closure of $p$. The existence of such models is often considered a blemish on the completion semantics. The absence of "spurious" models can be assured, however, by requiring that facts $\Pi$ define relation $p$ to be acyclic.

In this chapter, we consider the more general situation when $\Pi$ is a logic program, not necessarily a set of facts. This program may define several predicates besides $p$. Even $tc$ is allowed to occur in $\Pi$, except that all occurrences of this predicate are supposed to be in the bodies of rules, so that all rules defining $tc$ in $\Pi \cup Def$ will belong to $Def$. The rules of $\Pi$ may include negation as failure, and, accordingly, we talk about answer sets [Gelfond and Lifschitz, 1990] instead of the minimal model mentioned in Section 2.1.1. Program $\Pi \cup Def$ may have many answer sets. Is it true that, in each of them, the extent of $tc$ is the transitive closure of the extent of $p$? Theorem 4 gives a positive answer to this question. Next, we would like to know under what conditions the completion of a program containing $Def$ has no "spurious" models. Such conditions are provided in Theorem 5.

The questions discussed in this chapter are important from the perspective of answer set programming. A logic program whose answer sets we want to find may define the transitive closure of one of its predicates—for instance, the transitive closure *above* of the relation *on* in the blocks world presented in Section 3.6. In such cases, we expect that the definition will "operate correctly" in the context of a set of rules with negation as failure, not merely in combination with a set of facts. If every model of the completion of such a program is an answer set then the program's answer sets can be found using a propositional solver.

In the following, first we present our theorem showing the correctness of the definition of transitive closure under the answer set semantics. Then, we investigate the correctness of the definition of transitive closure relative to the completion semantics. After that, this theory is illustrated by applying it to the formalization of the blocks world from Chapter 3.

## 6.1 Transitive Closure and Answer Sets

We consider nondisjunctive programs consisting of rules (3.17). The syntax of logic programs is propositional. Expressions containing variables, such as $\mathit{Def}$, can be treated as schematic: we select a non-empty set $C$ of symbols ("object constants") and view an expression with variables as shorthand for the set of all its ground instances obtained by substituting these symbols for variables. It is convenient, however, to be a little more general. In the theorem below, $p$ and $tc$ are assumed to be functions from $C \times C$ to the set of atoms such that all atoms $p(x, y)$ and $tc(x, y)$ are pairwise distinct.

**Theorem 4** *Let $\Pi$ be a program that does not contain atoms of the form $tc(x, y)$ in the heads of rules. If $X$ is an answer set for $\Pi \cup \mathit{Def}$ then*

$$\{\langle x, y\rangle : tc(x, y) \in X\} \tag{6.1}$$

*is the transitive closure of*

$$\{\langle x, y\rangle : p(x, y) \in X\}. \tag{6.2}$$

If atoms of the form $tc(x, y)$ do not occur in $\Pi$ at all then the answer sets for $\Pi \cup \mathit{Def}$ are actually in a 1-1 correspondence with the answer sets for $\Pi$.[1] The answer set for $\Pi \cup \mathit{Def}$ corresponding to an answer set $X$ for $\Pi$ is obtained from $X$ by adding a set of atoms of the form $tc(x, y)$. This is easy to prove using the splitting set theorem [Lifschitz and Turner, 1994].

The proof of Theorem 4, given in [Erdem and Lifschitz, 2001b], is presented in Section 9.8 (page 174).

## 6.2 Transitive Closure and Completion

For any program $\Pi$ and any set $X$ of literals, we say about literals $L, L' \in X$ that $L$ is an *ancestor* of $L'$ relative to $\Pi$ and $X$ if there exists a finite sequence of literals $L_1, \dots, L_n \in X$ $(n > 1)$ such that $L = L_1$, $L' = L_n$ and for every $i$ $(1 \leq i < n)$, $L_i$ is a parent of $L_{i+1}$ relative to $\Pi$ and $X$. In other words, the ancestor relation is the transitive closure of the parent relation.

**Theorem 5** *Let $\Pi$ be a program that does not contain atoms of the form $tc(x, y)$ in the heads of rules. For any set $X$ of literals, if*

*(i)* $\Pi$ *is tight on* $X$*,*

*(ii)* $\{\langle x, y\rangle : \ p(y, x) \in X\}$ *is well-founded, and*

*(iii) no atom of the form $tc(x, y)$ is an ancestor of an atom of the form $p(x, y)$ relative to $\Pi$ and $X$,*

[1] This observation is due to Hudson Turner (personal communication, October 3, 2000).

*then* $\Pi \cup Def$ *is tight on* $X$*. If, in addition,*

*(iv)* $X$ *is a consistent set closed under and supported by* $\Pi \cup Def$

*then* $X$ *is an answer set for* $\Pi \cup Def$*, and* $\{\langle x, y\rangle : tc(x, y) \in X\}$ *is the transitive closure of* $\{\langle x, y\rangle : p(x, y) \in X\}$.

The first part of the theorem tells us that, under some conditions, the tightness of a program is preserved when the definition of the transitive closure of a predicate is added. The second part, in application to finite programs without classical negation, tells us that, under some conditions, the answer sets for $\Pi \cup Def$ can be characterized as the models of this program's completion, so that, in any model of completion, the extent of $tc$ is the transitive closure of the extent of $p$.

Condition (ii) is similar to the acyclicity property mentioned in the introduction. In fact, if the underlying set $C$ of constants is finite then (ii) is obviously equivalent to the following condition: there is no finite sequence $x_1, \ldots, x_n \in C$ $(n > 1)$ such that

$$p(x_1, x_2), \ldots, p(x_{n-1}, x_n), p(x_n, x_1) \in X. \tag{6.3}$$

For an infinite $C$, well-foundedness implies acyclicity, but not the other way around.

Here is a useful syntactic sufficient condition for (ii):

**Proposition 4** *If* $\Pi$ *contains constraint*

$$\bot \leftarrow tc(x, x) \tag{6.4}$$

*and $C$ is finite then, for every set $X$ of literals closed under $\Pi \cup Def$, set $\{\langle x, y\rangle : \; p(y, x) \in X\}$ is well-founded.*

The proof of Proposition 4, given in [Erdem and Lifschitz, 2001b], is presented in Section 9.10 (page 177).

Without condition (ii), the assertion of the theorem would be incorrect. Program $\Pi$ that consists of one fact $p(1,1)$, with $C = \{1, 2\}$ and

$$X = \{p(1,1), tc(1,1), tc(1,2)\},$$

provides a counterexample.

Condition (iii) can be verified by checking, for instance, that $p$ does not depend positively on $tc$ in the dependency graph of $\Pi$. This condition is essential as well. Indeed, take $\Pi$ to be

$$p(x, y) \leftarrow tc(x, y).$$

With $C = \{1, 2\}$, set $X = \{p(2,1), tc(2,1)\}$ is closed under and supported by $\Pi \cup Def$, but is is not an answer set for $\Pi \cup Def$: the only answer set for this program is empty.

The proof of Theorem 5,, given in [Erdem and Lifschitz, 2001b], is presented in Section 9.9 (page 175).

## 6.3 Example: The Blocks World

As an example of the use of Theorem 5, consider the history program consisting of rules (3.28), (3.21)–(3.26) and (3.31)–(3.33) for the blocks world. Theorem 5 can be used to prove the following proposition:

**Proposition 5** *The program consisting of rules (3.28), (3.21)–(3.26) and (3.31)–(3.33) is tight on every set of literals that is closed under it.*

Since the program consisting of rules (3.28), (3.21)–(3.26) and (3.31)–(3.33) contains classical negation, the completion process is not applicable to it directly. But classical negation can be eliminated from the program by the transformation explained in Section 3.6. Proposition 5 tells us that, after this transformation, the program's answer sets can be computed by using a propositional solver to find models of the program's completion, as described in [Babovich *et al.*, 2000].

The idea of the proof is to check first that the program consisting of rules (3.28), (3.21)–(3.26) and (3.32)–(3.33) is tight, and then use Theorem 5 to conclude that tightness is preserved when we add the definition (3.31) of *above*. There are two complications, however, that need to be taken into account.

First, *on* and *above* are ternary predicates, not binary. To relate them to the concept of transitive closure, we can say that any binary "slice" of *above* obtained by fixing its last argument is the transitive closure of the corresponding "slice" of *on*. Accordingly, Theorem 5 will need to be applied $T_{max} + 1$ times, once for each slice.

Second, the first two arguments of *on* do not come from the same set $C$ of object constants, as required in the framework of Theorem 5: the set of block constants is a proper part of the set of location constants. We need to introduce a program similar to the program consisting of rules (3.28), (3.21)–(3.26) and (3.31)–(3.33) in which, syntactically, *Table* is allowed as the first argument of both *on* and *above*.

The proof of Proposition 5 that uses Theorem 5 is given in [Erdem and Lifschitz, 2001b], and also presented in Section 9.11 (page 177).

## 6.4 Discussion

To prove that the completion of a program has no models other than the program's answer sets, we can check that the program is tight. When the program contains the definition of the transitive closure of a predicate, it may be difficult to check its tightness directly. But our Theorem 5 can be sometimes used to show that the tightness of a program is not lost when such a definition is added to it.

Essential for the applicability of that theorem is the presence of the "irreflexivity" constraint from Proposition 4, such as constraint (3.32) from the blocks world example. There are cases, however, when such constraints cannot be included without distorting the meaning of the program, as, for instance, when the concept of transitive closure is used to talk about paths in an arbitrary graph (Section 5.5).

# Chapter 7

# Applications of Answer Set Programming to Wire Routing

Very large scale integrated circuits (VLSI), with millions of transistors and wires on a single silicon chip, are too complex to design without the aid of computers. Advances in integrated circuit technology will result in more complex chips in the near future—it is predicted that there will be over 1 billion transistors and wires on a single chip in about 10 years [Semiconductor Industry Association, 1997]. As a result, research and development in computer-aided design (CAD) software is very active in both industry and academia.

Routing is an important step in CAD for VLSI circuits [Lengauer, 1990]. It is the problem of determining the physical locations of all the wires interconnecting the circuit components (transistors, gates, functional units, etc.) on a chip. Since the wires cannot intersect with each other (otherwise resulting in short circuits), they are competing for limited spaces, thus making routing a difficult combinatorial optimization problem. In practice, the routing problem for the whole VLSI chip is decomposed into smaller routing problems [Lengauer, 1990]. The chip is partitioned into an array of rectangular regions. After determining the connections between adjacent regions, the routing of all the regions are carried out independently. But even for an individual region the problem is computationally difficult. VLSI routing has been shown to be NP-complete [Szymanski, 1985], and there are many heuristic routing algorithms in the literature [Lengauer, 1990].

We have introduced a new approach to VLSI routing using answer set programming. All existing routing systems are based on variations of the sequential maze routing approach using a shortest path algorithm connecting one wire at a time [Lee, 1961, Ohtsuki, 1986]. A major shortcoming of these algorithms is that they cannot guarantee finding a routing solution even when one exists. Our method differs from the existing ones in that it is complete: it always correctly determines whether a given routing problem is solvable, and it produces a routing solution whenever one exists.

Consider, for instance, the routing problem shown in Figure 7.1. The wiring space here is a rectangular grid. The goal is to connect 4 pairs of points ("pins")—the two points labeled $p0$, the two points labeled $p1$, and so on—without passing through the obstacles, shown in black. A solution—actually, the solution found by our method proposed in the following section—is given in Figure 7.2. If we try to solve this problem by finding first a shortest path

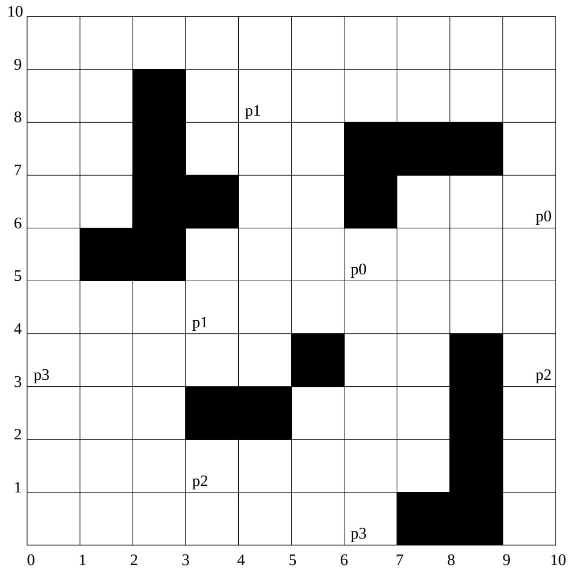


Figure 7.1: A routing problem with 4 wires.

between the points labeled $p0$, and then a shortest path between the points labeled $p1$ in the part of the grid that is still available, we will arrive at a partial solution like the one shown in Figure 7.3. This partial solution cannot be extended to a complete solution, however, because the points labeled $p2$ cannot be connected without intersecting the first of the two paths selected earlier.

In the new approach the idea is characterize VLSI routing as a graph problem. Consider, for instance, the routing problem shown in Figure 7.1. The goal is to find 4 paths that connect 4 pairs of points—the two points labeled $p0$, the two points labeled $p1$, and so on, and that do not pass through the obstacles, shown in black. We describe the problem as a logic program, and

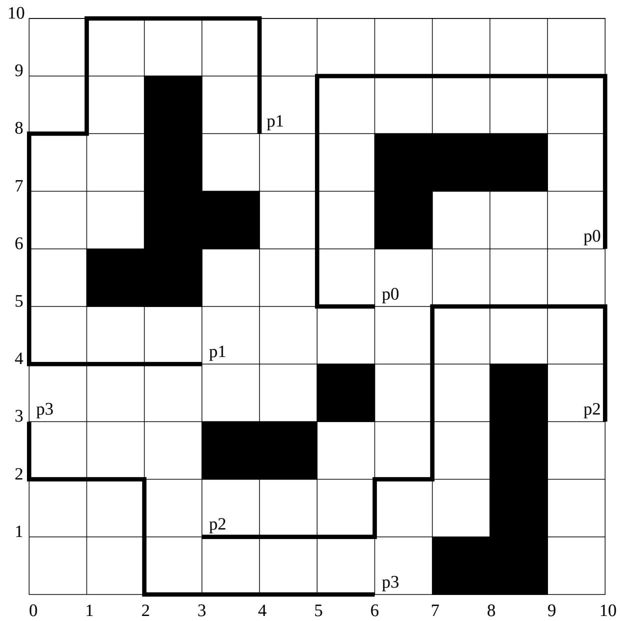


Figure 7.2: Another solution to the problem from Figure 7.1.

use SMODELS to find the answer sets for the program that corresponds to the graph problem.

In the next two sections we provide a more detailed description of this method as it applies to the problem above. Then we show that our approach can handle various kinds of additional routing constraints which ensure that a circuit meets its performance specification: constraints on the lengths of the wires, essential because signal delay through a wire is proportional to its length, and spacing constraints between the wires, related to the problem of avoiding signal interferences.

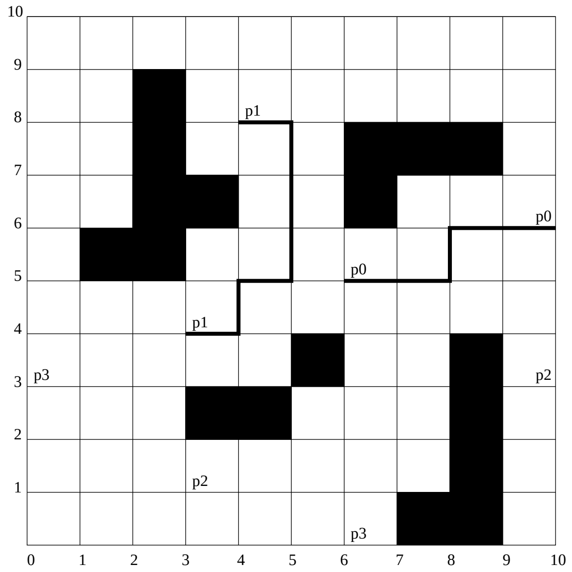


Figure 7.3: A partial solution to the problem from Figure 7.1. It cannot be extended to a complete solution.

## 7.1 Input and Output of SMODELS

The solutions of the wire routing problems with this approach are characterized by the truth values of the atoms `in_h(N,XC,YC)` ("the horizontal segment connecting the points `(XC,YC)` and `(XC+1,YC)` occurs in Path `N`") and `in_v(N,XC,YC)` (" the vertical segment connecting the points `(XC,YC)` and `(XC,YC+1)` occurs in Path `N`").

Consider, for instance, the problem shown in Figure 7.5. This problem is described to SMODELS by the file presented in Figure 7.4. The number of wires and the size of the grid are represented in that file by the constants `k`, `maxX` and `maxY`. These numeric values are defined in each particular rout-

```
% routing problem 0

const k = 1.
const maxX = 3.
const maxY = 3.

start(0,0,1).
start(1,0,2).
end(0,3,0).
end(1,2,3).

obstacle(1,3,1).

compute all {}.

minimize {in_h(N,X,Y): path(N): x_coordinate(X): y_coordinate(Y),
          in_v(N,X,Y): path(N): x_coordinate(X): y_coordinate(Y)}.

hide.
show in_h(N,X,Y).
show in_v(N,X,Y).
```

Figure 7.4: Input file for the problem from Figure 7.5

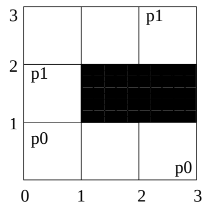


Figure 7.5: A routing problem with 2 wires.

ing problem. Then, for every path, the starting point and the ending point are specified. Next, we describe the shape of the obstacle in this example by `obstacle(1,3,1)`. Here, `obstacle(X1,X2,Y)` expresses that there is an obstacle occupying the points covered by the rectangle defined by the points `(X1,Y)`, `(X2,Y)`, `(X1,Y+1)`, and `(X2,Y+1)`. After that, we specify that, among all solutions, we want to find the one with the minimum number of horizontal and vertical segments. We do this using the special command **`minimize`** of SMODELS. With the expression **`minimize`** $\{\ldots\}$, SMODELS finds an answer set containing the least number of elements from the set $\{\ldots\}$. By the last three lines, we eliminate from SMODELS' output all the atoms except the ones that are either of the form `in_h(N,X,Y)` or of the form `in_v(N,X,Y)`.

To find a solution to this problem we need the file `routing.lp`, describing the effects and the executability of actions in the routing domain. Parts of file `routing.lp` are discussed in the next section.

Given these two files with the file presented in Figure 7.4, SMODELS finds the following graph:

```
Stable Model: in_h(0,0,0) in_h(0,1,0) in_h(1,0,3) in_h(0,2,0)
in_h(1,1,3) in_v(0,0,0) in_v(1,0,2) .
```

This is the solution shown in Figure 7.6.

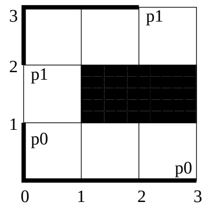


Figure 7.6: A routing problem with 2 wires.

## 7.2 The Routing Domain

In file `routing.lp`, first a set of atoms of the form `in_h(N,XC,YC)` are "generated" by the rule

```
{in_h(N,XC,YC)} :-
        XC < maxX, path(N),
        x_coordinate(XC),
        y_coordinate(YC),
        ends(N,X,Y).
```

where `ends(N,X,Y)` defines the end points of Path `N`. Similarly, a set of atoms of the form `in_v(N,XC,YC)` are "generated". The union of these sets describes a graph—a subgraph of the grid. Then this set is "tested" with some constraints. The constraints express that the graph is the union of several disjoint paths forming a solution to the routing problem.

To express that both the starting point and the ending point of a path should be in that path, that a point cannot be in two different paths, and that a path cannot contain a point blocked by an obstacle, we introduced the atom at(N,X,Y) ("the point (X,Y) is in Path N"). We express that the end points of Path N, specified in a problem description file, should be in Path N by the constraint:

```
:- not at(N,XC,YC),
   ends(N,XC,YC).
```

We need to make sure that the end points of Path N cannot be connected to two or more points, whereas each of the other points of Path N should be connected to exactly two points. Furthermore, we should make sure that there are no forks in the graph. For that, we defined the atom at(N,XC,YC,D) ("the unit segment that begins at the point (XC,YC) and goes in the direction D occurs in Path N"). We make sure that the end points of Path N cannot be connected to two or more points by the constraint

```
:- 2{at(N,XC,YC,D):direction(D)},
   path(N),
   x_coordinate(XC),
   y_coordinate(YC),
   ends(N,XC,YC).
```

Each of the other points of Path N cannot be connected to exactly one point

```
:- 1{at(N,XC,YC,D):direction(D)}1,
   path(N),
   x_coordinate(XC),
```

```
   y_coordinate(YC),
   not ends(N,XC,YC).
```

and cannot be connected to more than three points.

```
:- 3{at(N,XC,YC,D):direction(D)},
   path(N),
   x_coordinate(XC),
   y_coordinate(YC),
   not ends(N,XC,YC).
```

That is, each of these points should be connected to exactly two points.

To make computation more efficient, we define the atoms `r(N,X,Y)` ("the point `(X,Y)` is reachable from the starting point of Path `N`") and express that every node in Path `N` is reachable from the starting point of Path N by the constraint:

```
:- at(N,X,Y),
   path(N),
   x_coordinate(X),
   y_coordinate(Y),
   not r(N,X,Y).
```

Strictly speaking, this constraint is redundant in the presence of the `minimize` statement from Figure 7.5, because it cannot be violated in a graph of minimal size. But it gives some "direction" to SMODELS in search, which makes the computation time smaller.

We express that no two paths intersect by the following constraint.

```
:- path(N), path(N1),
   x_coordinate(XC),
   y_coordinate(YC),
   at(N,XC,YC),
   at(N1,XC,YC),
   N < N1.
```

After describing the shape of the obstacles on the grid by defining `obstacle(X1,X2,Y)`, we express that a path cannot contain a point blocked by an obstacle as follows:

```
:- path(N),
   x_coordinate(XC),
   y_coordinate(YC),
   at(N,XC,YC),
   obstacle(XC1,XC2,YC),
   le(XC1,XC), le(XC,XC2).


:- path(N),
   x_coordinate(XC),
   y_coordinate(YC),
   at(N,XC,YC+1),
   obstacle(XC1,XC2,YC),
```

```
le(XC1,XC), le(XC,XC2).
```

With the routing domain described above, for instance, SMODELS finds the solution presented in Figure 7.2 to the problem described in Figure 7.1 in less than two seconds.

## 7.3 Bus Routing

A bus is a set of wires, each connecting a source pin and a sink pin, where the source pins are all adjacent and the sink pins are all adjacent. In bus routing, given several pairs of points on a rectangular grid, we want to find a configuration of a bus such that all wires are of the same length: we want the signal delays through all wires to be equal. The need to express the equality of the lengths is the main special feature of bus routing problems. A bus routing problem, along with its solution found by SMODELS, is displayed in Figure 7.7.

To express the equality of the lengths of the buses, among all possible values of lengths of buses, we use the predicate `bus_length` to pick a particular value, and make sure that the length of every bus is equal to this value. This is done by adding to the problem description the following rules:

```
length(1..maxLength).


1{bus_length(L):length(L)}1.
```

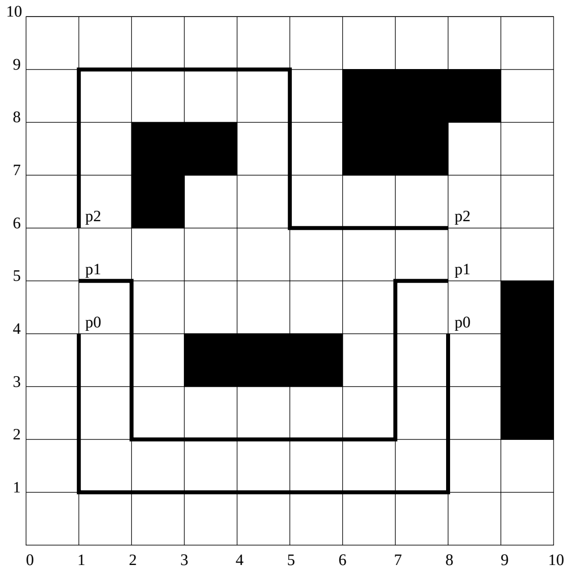


Figure 7.7: A bus routing problem. The wires are required to have the same length.

```
bus(N) :-
   L+1 {at(N,XC,YC):x_coordinate(XC):y_coordinate(YC)} L+1,
   bus_length(L), length(L), path(N).


:- not bus(N), path(N).
```

In some cases, a bus routing problem has no solution but becomes solvable if we relax the condition on the lengths of wires. For instance, with the configuration of obstacles shown in Figure 7.8, it is impossible to connect all pairs of pins by paths of the same length, but there is an "approximate solution" in which the lengths of wires do not differ by more than 2 (Figure 7.9).

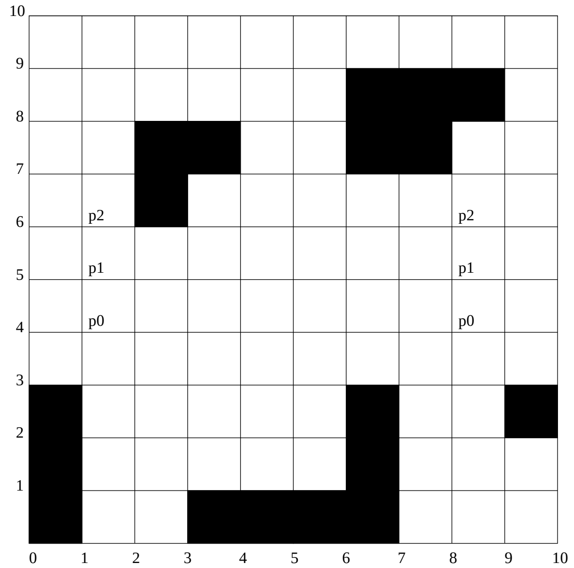


Figure 7.8: A bus routing problem that has no precise solution.

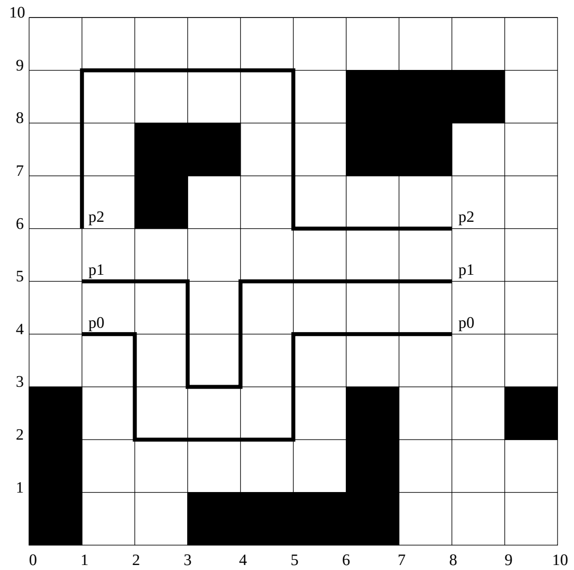


Figure 7.9: An approximate solution to the problem from Figure 7.8. The differences between the lengths of wires are limited by 2.

To find an approximate solution for the problem presented in Figure 7.8, the condition on the lengths of the buses can be relaxed by replacing the third rule above by

```
bus(N) :-
   L+1 {at(N,XC,YC):x_coordinate(XC):y_coordinate(YC)} L+1+relax,
   bus_length(L), length(L), path(N).
```

where `relax` is a constant declared in the problem description. When `relax` is declared to be 2, SMODELS finds the solution presented in Figure 7.9 in less than two seconds.

## 7.4 Restricting the Lengths of Wires

A wire routing problem may involve constraints on the lengths of some of the wires—that is to say, on signal delays through them. The approach to wire routing proposed in this chapter allows us to express such constraints by simple changes in the goal condition of the planning problem.

We can put restrictions on the lengths of the wires as follows. We can express that any wire cannot be longer than a specific value, say `maxLength`, by adding to the problem description the constraint

```
:- maxLength + 2 {at(N,X,Y):x_coordinate(X):y_coordinate(Y)},
   path(N).
```

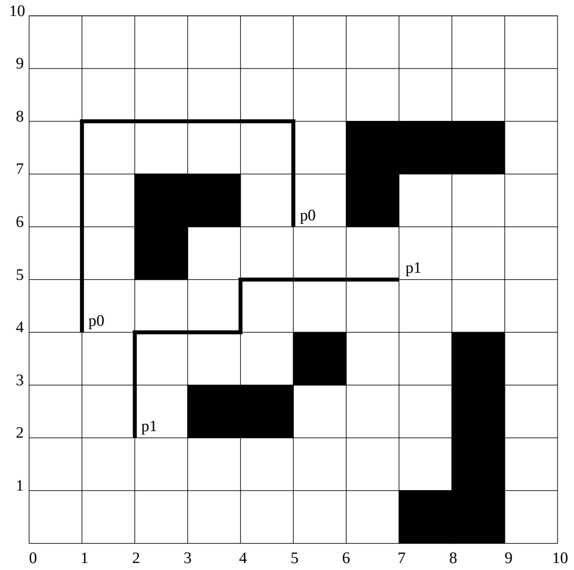


Figure 7.10: A solution to a routing problem with 2 wires.

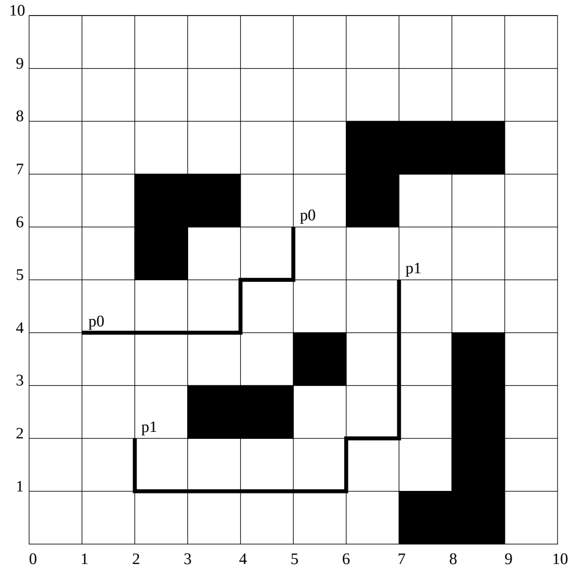


Figure 7.11: A solution to the problem from Figure 7.11 with the length of Wire 1 limited by 8.

After this change, a solution found by SMODELS is presented in Figure 7.10.

We can put restrictions on the length of a specific wire. For instance, for the problem described in Figure 7.11, we can express that Wire 0 cannot be longer than 7 (inclusive) by adding to the problem description the constraint

```
:- 8 {at(0,X,Y): x_coordinate(X): y_coordinate(Y)}.
```

The solution found by SMODELS after this change is shown in Figure 7.11.

We can express that the total length of all `k` wires cannot be greater than a specific value, say `maxTotalLength` by adding to the problem description the constraint

```
:- maxTotalLength + k + 1 {at(N,X,Y):path(N):x_coordinate(X):
   y_coordinate(Y)}.
```

## 7.5 Spacing Constraints

We say that two wires in a solution to a routing problem are *adjacent* if a segment of one of them and a segment of the other form two opposite sides of a unit square. In Figure 7.2, for instance, Wires 0 and 1 are adjacent, and Wires 2 and 3 are adjacent. In this section we consider the problem of finding a wire routing without adjacent wires. This is a simple spacing constraint, interesting in view of its relation to the problem of avoiding signal interferences.

We describe that no two wires can be adjacent by the following constraints:

```
:- in_h(N,XC,YC),
   in_h(N1,XC,YC+1),
   YC < maxY,
   path(N;N1), N != N1,
   x_coordinate(XC),
   y_coordinate(YC).

:- in_v(N,XC,YC),
   in_v(N1,XC+1,YC),
   XC < maxX,
   path(N;N1), N != N1,
   x_coordinate(XC),
   y_coordinate(YC).
```

More generally, we can describe that no two unit segments belonging to different paths can form the opposite sides of a rectangle of size `1` $\times$ `D` (`1` $\leq$ `D` $\leq$ **`dist`**) by the constraints:

```
:- in_h(N,XC,YC),
   in_h(N1,XC,YC+D),
   distance(D), le(YC,maxY-D),
   path(N;N1), N != N1,
```

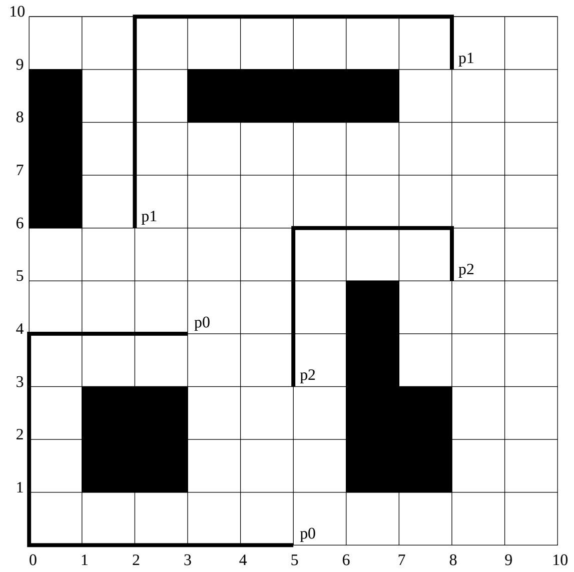


Figure 7.12: A solution to a routing problem without adjacent wires.

```
   x_coordinate(XC),
   y_coordinate(YC).


:- in_v(N,XC,YC),
   in_v(N1,XC+D,YC),
   distance(D), le(XC,maxX-D),
   path(N;N1), N != N1,
   x_coordinate(XC),
   y_coordinate(YC).
```

where `dist` is a constant declared in the problem description.

Figure 7.12 shows a solution to a routing problem, with adjacent wires prohibited, that was generated by SMODELS on the basis of such a formalization.

## 7.6 Planning Approach to Wire Routing

In the earlier sections, we characterized the wire routing problems as graph problems. Another approach is to consider the wire routing problems as planning problems [Erdem *et al.*, 2000]. In the planning approach, the idea is to think of each path as the trajectory of a robot moving along the grid lines, and to understand a routing problem as the problem of planning the actions of several robots. For instance, the problem presented in Figure 7.1 involves 4 robots. The initial position of Robot 0 is assumed to be (6,5), and its goal is to reach point (10,6) (or the other way around), and similarly for the other robots. The actions that a robot can perform are to move left, right, up or down to the closest grid point, or to do nothing.

With the planning approach, we can formalize the wire routing problems as logic programs in the same way we formalize planning problems in Chapter 3. Planning problems can be also formalized in a more compact way, using "action languages".

Action languages are formal models of parts of the natural language that are used to talk about the effects of actions [Gelfond and Lifschitz, 1998]. Action languages, in general, use two kinds of symbols: fluent symbols and action symbols. A set of propositions in an action language describes a transition system. One of many action languages is the action language $\mathcal{C}$ introduced in [Giunchiglia and Lifschitz, 1998]. The language $\mathcal{C}$ is based on the theory of causal explanation proposed in [McCain and Turner, 1997], which is closely related to the concept of an answer set [McCain, 1997, Chapter 6 ].

For instance, the transition system describing the suitcase domain presented in Section 3.3.1 can be formalized in the action language $\mathcal{C}$ as follows:

$$\begin{array}{l}
\textbf{caused } \mathit{open} \textbf{ if } \mathit{up}(L1) \ \wedge \mathit{up}(L2) \\
\mathit{toggle}(l) \textbf{ causes } \mathit{up}(l) \textbf{ if } \neg \mathit{up}(l) \\
\mathit{toggle}(l) \textbf{ causes } \neg \mathit{up}(l) \textbf{ if } \mathit{up}(l) \\
\textbf{inertial } \mathit{up}(l), \neg \mathit{up}(l), \mathit{open}, \neg \mathit{open}
\end{array}$$

where $l \in \{L1, L2\}$. Using the translation from $\mathcal{C}$ to logic programming described in [Lifschitz and Turner, 1999] and Theorem 1 of [Erdem and Lifschitz, 1999], discussed in Section 4.1, this action description can be turned into the union of logic programs (3.10) and (3.11), which is the first logic program presented in Section 3.3.2, on page 32 to describe the transition system for the suitcase domain.

In [Lifschitz and Turner, 1999], the authors also proved that the models of the completion of the program obtained from an action domain using their translation are identical to the answer sets for that program. This is important in that we can use propositional solvers to reason about action domains described in $\mathcal{C}$.

Besides the use of CCALC discussed in Chapters 3 and 5 above, this system can also be used to solve problems related to actions. In this case, its

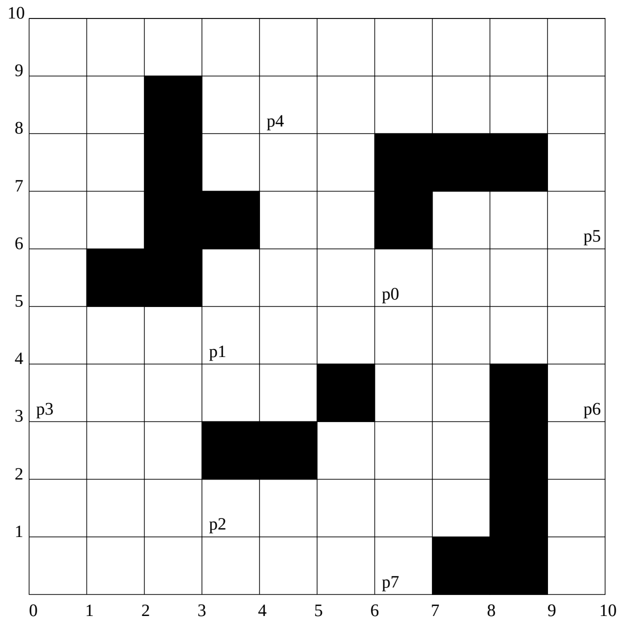


Figure 7.13: A routing problem.

input can be represented in $\mathcal{C}$. Starting with a problem described in $\mathcal{C}$, CCALC translates it into a propositional theory. Then, it uses a propositional solver to find a model of this theory. This model corresponds to an answer set for the program obtained from the action description we start with.

The possibility of using CCALC with an input described in the action language $\mathcal{C}$ allows us to encode the wire routing problems in the action language $\mathcal{C}$ and find solutions using CCALC as in [Erdem *et al.*, 2000].

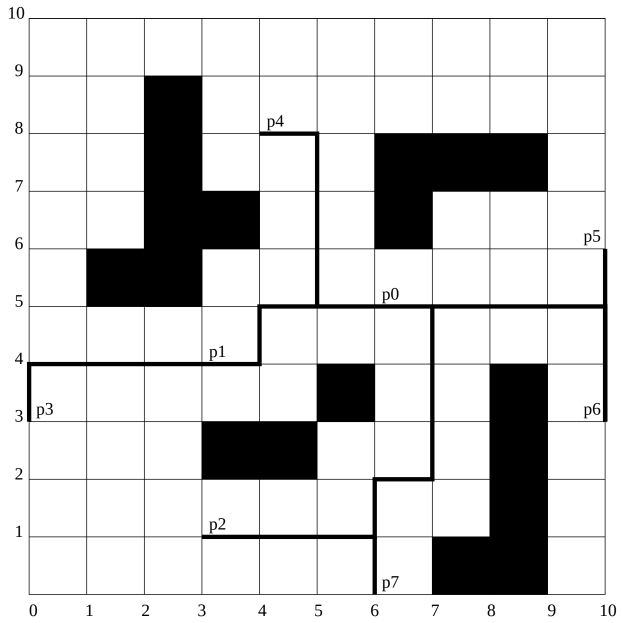


Figure 7.14: A solution to the problem presented in Figure 7.13.

## 7.7 Other Wire Routing Problems

In the routing problems we have addressed, the goal was to connect pairs of pins with wires, and the solutions consisted of paths that do not intersect with each other, and that do not intersect with obstacles. There are routing problems where the goal is to connect multiple pins with wires where the solutions consist of trees that do not intersect with obstacles. For instance, consider Figure 7.13. We want to connect the pins $p0, \ldots, p7$ with wires. A solution to this problem is presented in Figure 7.14 and consists of a "Rectilinear Steiner Tree" (RST) for nodes $p0, \ldots, p7$ [Hanan, 1966].[1] Automatic routing of wires

[1] The *Steiner tree problem* asks for a connected graph spanning a given set of points such that the total "length" of edges is minimum. A Steiner tree is different from a "minimum

has been the premier application of RSTs since Hanan's original paper [Hanan, 1965]. The routing problems with solutions as minimum RSTs are motivated by delay optimization. The problem presented in Figure 7.13 differs from other RST routing applications studied earlier in that there are obstacles on the grid. (See [Lengauer, 1990] for the work on RST routing applications). We can solve such a problem by removing the constraint expressing that no two paths intersect from the basic program described in Section 7.2.

We have studied the wire routing problems with the spacing constraint expressing that no adjacencies are allowed. In the future, we want to work on wire routing problems with more complex spacing constraints where adjacent wires are allowed but the total amount of adjacencies between each pair of wires should be bounded. For instance, consider the problem presented in Figure 7.12 with a solution where adjacencies are prohibited. If at most one unit of adjacency are allowed the a more economical solution exists, such as the one shown in Figure 7.15. This more general formulation captures the fact that small amount of adjacencies may not produce enough signal interferences to affect circuit performance.

Another future direction is to investigate how to extend both approaches

---

spanning tree" in that new auxiliary points (called *Steiner points*) can be introduced between the given points so that a connected graph spanning all the points will be "shorter" than otherwise possible. This problem was proposed by Georg Steiner as a generalization of a special case for three-points in the Euclidean plane, where the lengths of edges are the Euclidean distances, introduced by Fermat (1601–1665) [Jarník and Kössler, 1934]. A *Rectilinear Steiner Tree* for a given set $S$ of points in the plane is a Steiner tree for $S$ where edges are horizontal or vertical line segments and each point in $S$ is a leaf of this tree [Hanan, 1966]. Here the length of an edge is the number of segments contained in that edge. (See [Hwang *et al.*, 1992] for more information about Steiner trees.)

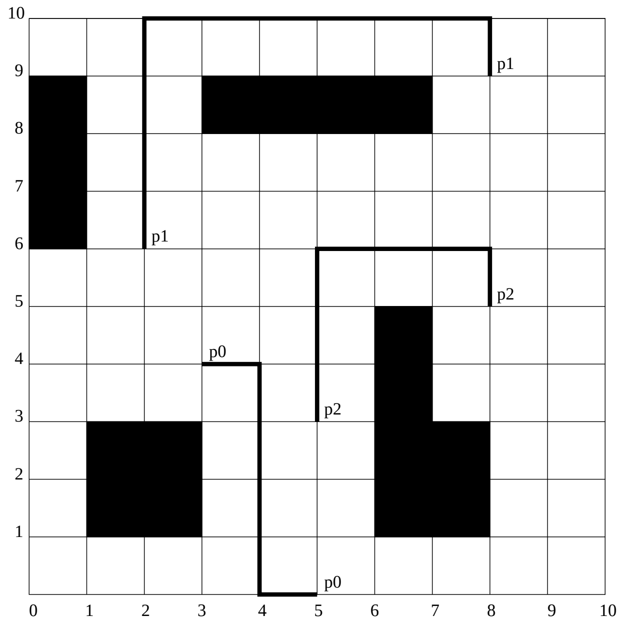


Figure 7.15: A solution to a routing problem where at most two units of adjacencies are allowed.

to solve the "global routing" problem [Lengauer, 1990], which is the problem of determining the connections between adjacent regions after a VLSI chip is decomposed into an array of smaller rectangular regions. The global routing problem resembles the routing problem we studied in this paper except that we would allow more than one wire to be placed on a grid edge.

## 7.8 Discussion

We have introduced two approaches to wire routing using answer set programming: the planning approach [Erdem *et al.*, 2000] and the graph approach

(Chapter 7). Both approaches always correctly determine whether a given problem is solvable, and they always produce a solution if it exists. Both approaches are attractive in that some enhancements of the basic problem—in which lengths of wires and distances between them come into play—can be easily represented by modifying goals or by adding auxiliary fluents. In this sense, both representation methods are similar to the work on elaboration tolerance [McCarthy, 1999] described in [Lifschitz, 2000].

On the negative side, the size of the grid used in our examples is much too small for serious applications. Investigating the applicability of these routing methods to larger problems is another topic for future work.

In the graph approach, to make our encoding more efficient, we can introduce two "circles" around the endpoints of a path we are looking for, and require that the path be contained in the union of these circles.[2] For that, we first declare the constant `radius`, and then we replace the rule "generating" a set of atoms of the form `in_h(N,XC,YC)` by

```
{in_h(N,XC,YC)} :-
        XC < maxX, path(N),
        x_coordinate(XC),
        y_coordinate(YC),
        ends(N,X,Y),
        radius - abs(XC-X) - abs(YC-Y) >= 0.
```

Similarly, we modify the rule "generating" a set of atoms of the form `in_v(N,XC,YC)`.

[2]This idea was suggested to us by Ilkka Niemelä at AAAI 2000.

This modification improves the computation time of SMODELS significantly. However, it can prevent SMODELS from finding a solution to a solvable problem if the value of `radius` is too small.

Another formalization of the routing domain for SMODELS is presented in [East and Truszczyński, 2001]. The encoding presented by East and Truszczyński is similar to our encoding for SMODELS in that they also put some restrictions on the area where the wires should be routed. It differs from our encoding in that paths are characterized by the points occurring in the paths. Since the paths are not defined in terms of segments, their formalization may need to be modified when other routing problems are considered. East and Truszczyński also present a similar encoding for the system DCS.

Another formalization of the routing problems presented to SMODELS is due to Tommi Syrjänen (personal communication, July 31, 2000). His encodings are similar to ours.

# Chapter 8

# Applications of Answer Set Programming to Phylogeny Reconstruction

A *phylogeny* is an evolutionary tree for a set of taxa, which describes the evolution of the taxa in that set from their most recent common ancestor. There are several reasons to construct phylogenies, as explained in [Hwang *et al.*, 1992]. The topology is of interest because it answers questions about basic classification of taxa. Sometimes one wants to know when divergence occurred and how "long" the edges of the tree are; this question is sometimes answered by explicitly deriving a description of the inferred (extinct) taxa.[1]

[1] These questions can be regarded as a special case of Steiner tree problems (see Footnote ([1])). The input taxa correspond to the leaves of a Steiner tree. The internal nodes are inferred ancestral taxa, and correspond to Steiner points. It is a special case because the internal nodes of phylogenies are of degree 3 whereas the Steiner points are of degree at

In particular, in biology, phylogeny reconstruction is crucial to a wide range of basic and applied biological problems such as the epidemiology of AIDS, the identification of viral agents, the analysis of protein structure and function, and the prediction of RNA structure; it is the basic tool to trace the evolutionary history. (See surveys [Felsenstein, 1982], [Felsenstein, 1988], [DasGupta and Wang, 1999], and the book [Li and Graur, 1991] for further discussion of biological motivation.)

In linguistics, phylogenies are reconstructed to trace the evolutionary history of a collection of natural languages. The leaves represent the extant languages, the internal vertices represent the ancestral languages, and the edges represent the "genetic" relations between the languages. For instance, when we say "French and Italian are both descendants of Latin" we refer to such a tree where French and Italian are denoted by leaves, and Latin is denoted by an internal vertex that is a common ancestor of these two leaves.

Reconstructing phylogenies for various language families is a major endeavor in historical linguistics, but is also of interest to archaeologists, human geneticists, and physical anthropologists. For instance, an accurate reconstruction of the evolutionary history of certain languages can help us answer questions about human migrations, the time that certain artifacts were developed, when ancient people began to use horses in agriculture [Mair, 1998], [Mallory, 1989], [Roberts *et al.*, 1990], [White and O'Connell, 1982].

The idea of formulating the phylogeny reconstruction problem formally

least 3.

and search for algorithmic solution started around 1950, when Hennig proposed a systematic way of finding a phylogeny for a set of species [Hennig, 1966]. Although there is no agreement on how to define a phylogeny mathematically, according to [Hwang *et al.*, 1992], there are various formalizations and algorithms to solve these problems. (See surveys [Felsenstein, 1982], [Felsenstein, 1988] for the phylogeny reconstruction methodologies in biology.)

In the following sections, we present applications of answer set programming to phylogeny reconstruction in linguistics (Section 8.1) and in biology (Section 8.2).

## 8.1 Phylogeny Reconstruction in Linguistics

Languages not only inherit characteristics from their ancestors but also sometimes borrow them from other languages. In such cases, "networks," rather than trees, are the appropriate model of evolution. Nakhleh *et al.* [2002] make this idea precise by defining "perfect phylogenetic networks." They start with a phylogeny built automatically from a dataset describing characteristics of Indo-European languages, and show how some perfect phylogenetic networks can be obtained from it by adding a small number of new edges.

The following sections address the same computational problem—computing a small set of additional edges that turn a given phylogeny into a perfect phylogenetic network. To solve this problem, we use answer set programming.

In the following, we will first describe the problem mathematically (Section 8.1.1) and then formalize it in the language of SMODELS (Section 8.1.2). Useful heuristics and optimization techniques will be discussed in Sections 8.1.3–8.1.5. After that, we will describe the dataset we used to find some explanations to the evolutionary history of the Indo-European languages and present the explanations computed by SMODELS (Section 8.1.6). Proofs of theorems, as given in [Erdem *et al.*, 2002], will be presented in Sections 9.12–9.17.

### 8.1.1 Problem Description

We describe the problem of computing perfect phylogenetic networks built on a given phylogeny as a graph problem. Therefore, we first introduce some definitions related to graphs.

Recall that a *directed graph (digraph)* is an ordered pair $(V, E)$ where $V$ is a set and $E$ is a binary relation on $V$. In a digraph $(V, E)$, the elements of $V$ are called *vertices*, and the elements $E$ are called the *edges* of the digraph.

In a digraph, we say that the edge $(u, v)$ is *incident from* $u$ and is *incident into* $v$. The *out-degree* of a vertex is the number of edges incident from it, and the *in-degree* of a vertex is the number of edges incident into it.

In a digraph $(V, E)$, a *path* from a vertex $u$ to a vertex $u'$ is a sequence $v_0, v_1, \ldots, v_k$ of vertices such that $u = v_0$ and $u' = v_k$, and $(v_{i-1}, v_i) \in E$ for $1 \leq i \leq k$. If there is a path from a vertex $u$ to a vertex $v$ then we say that $v$ is *reachable from* $u$. If $V'$ is a subset of $V$, and there exists a path from $u$ to $v$ whose vertices belong to $V'$ then we say that $v$ is *reachable from* $u$ *in* $V'$.

A *rooted tree* is a digraph with a vertex of in-degree 0, called the *root*, such that every vertex different from the root has in-degree 1 and is reachable from the root. In a rooted tree, a vertex of out-degree 0 is called a *leaf*.

A digraph $(V', E')$ is a *subgraph* of a digraph $(V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

A *phylogeny* is a triple of the form $(V, E, f)$ where $(V, E)$ is a finite rooted tree and $f$ is a function from $L \times I$ to $S$, where $L$ is the set of leaves of $(V, E)$, and $I$, $S$ are finite sets.

We are interested in the problem of turning a phylogeny into a perfect phylogenetic network by adding at most $k$ bidirectional edges. Formally, the problem is defined as follows:

**Input:** A phylogeny $(V, E, f)$, with $f : L \times I \to S$; a nonnegative integer $k$.

**Output:** a function $g : V \times I \to S$ and a symmetric irreflexive binary relation $N$ on $V$ such that

(i) $g|_{L \times I} = f$,

(ii) for all $i \in I$ and $s \in S$, if $V_{is} = \{u \in V : g(u, i) = s\}$ is not empty then $(V, E \cup N)$ has a subgraph with the set $V_{is}$ of vertices that is a rooted tree,

(iii) for every edge $(u, v) \in E$, $u$ is not reachable from $v$ in $(V, E \cup N)$,

(iv) the cardinality of $N$ is at most $2k$.

If $(V, E, N, g)$ satisfies conditions (i)–(iii) then we say that it is a *perfect (phylogenetic) network built on* $(V, E, f)$. The problem described above is

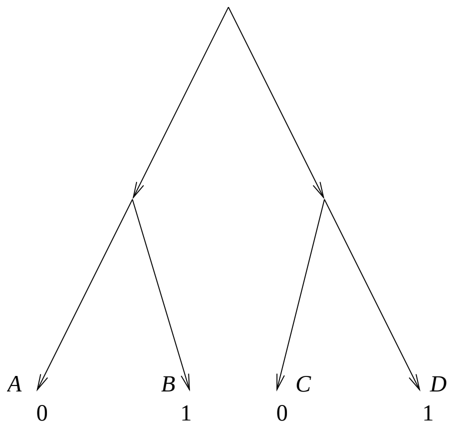


Figure 8.1: A phylogeny.

essentially the *Minimum Increment to Perfect Phylogenetic Network (MIPPN)* problem [Nakhleh *et al.*, 2002]. In place of (iv), MIPPN as defined in that paper includes a minimality condition.

Intuitively, the edges of the phylogeny $(V, E, f)$ show the "genetic" relations between languages; each leaf of this phylogeny corresponds to an extant language; the internal vertices represent ancestral languages. The languages are identified by a set of specific observable discrete characteristics, called "(qualitative) characters" (such as grammatical features, unusual sound changes, and cognate classes for different meanings). For every extant language, function $f$ maps every character to a "state"; we say that the leaves of the tree $(V, E)$ are "labeled" by $f$. Function $g$ extends $f$ to ancestral languages (condition (i)).

Languages can affect each other by transmitting some linguistic properties due to contact. These contacts are not represented on the given phylogeny. They correspond to the elements of $N$ in a perfect network $(V, E, N, g)$ built on $(V, E, f)$. A perfect network explains how every state of every character evolved from its original occurrence in some "root" language (condition (ii)). Languages cannot borrow characteristics from their descendants (condition (iii)). We are only interested in the perfect networks where the number of postulated borrowings is small (condition (iv)), because inheritance of characteristics of a language from its ancestors is far more probable than acquiring them through borrowing. In the case of the phylogeny of Indo-European languages discussed in Section 8.1.6 the fact that a small number of edges is sufficient was established in a preliminary analysis done by Tandy Warnow and Donald Ringe.

For instance, consider the phylogeny presented in Figure 8.1 that is reconstructed for 4 extant languages $A$, $B$, $C$, $D$. There is one character, i.e., $|I| = 1$, and there are two states ($S = \{0, 1\}$). The leaves of the phylogeny are labeled: $f(A, 1) = f(C, 1) = 0$ and $f(B, 1) = f(D, 1) = 1$. A perfect network built on this phylogeny is presented in Figure 8.2. The new bidirectional edge is added to make the vertices labeled 1 connected via a rooted tree, i.e., to satisfy condition (ii).

Another example, with two characters ($I = \{0, 1\}$) and two states ($S = \{0, 1\}$), is presented in Figure 8.3. The new edge is added to make the vertices labeled 0 at Character 1 connected via a rooted tree.

The phylogeny of Indo-European languages described in Section 8.1.6

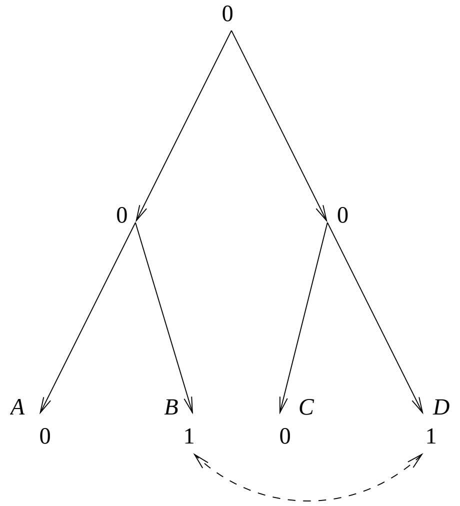


Figure 8.2: A perfect network built on the phylogeny of Figure 8.1 with $N = \{(B, D), (D, B)\}$.

| | | Characters | |
|---|---|---|---|
| | | 0 | 1 |
| Languages | $A$ | 0 | 0 |
| | $B$ | 0 | 1 |
| | $C$ | 1 | 0 |
| | $D$ | 1 | 1 |

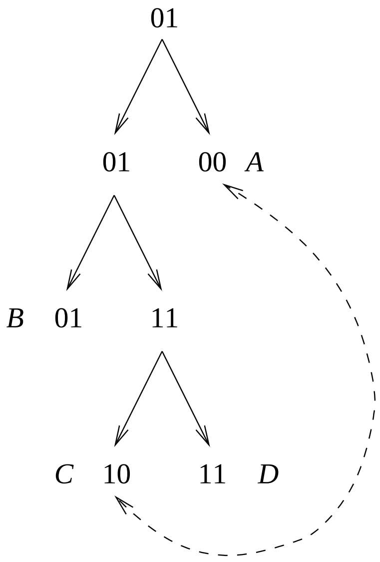


Figure 8.3: A perfect network.

```
% a phylogeny for the extant languages A, B, C, D:

% denote A by 1, B by 2, C by 3, D by 4,
%        the parent of A and B by 5,
%        the parent of C and D by 6,
%        the root by 0.

vertex(0..6).

edge(0,5). edge(0,6). edge(5,1).
edge(5,2). edge(6,3). edge(6,4).

state(0;1).

character(0).

f(1,0,0). f(2,0,1). f(3,0,0). f(4,0,1).
```

Figure 8.4: Input file describing the phylogeny of Figure 8.1.

below is a tree with 24 leaves, 370 characters, and 74 states.

### 8.1.2 Presenting the Problem to SMODELS

A phylogeny is defined by the domain predicates `vertex(X)`, `edge(X,Y)`, `state(S)`, `character(C)`, `f(X,C,S)` (expressing that the function $f$ maps the leaf `X` and the character `C` to the state `S`). For instance, the phylogeny of Figure 8.1 is described to SMODELS by the file presented in Figure 8.4.

The solutions of the problem are characterized by the atoms of the form `g(X,C,S)` (expressing that the function $g$ maps the vertex `X` and the character `C` to the state `S`) and `new(X,Y)` (expressing that the pairs `(X,Y)` and `(Y,X)` are elements of the set $N$; `X < Y`). For instance, the output of SMODELS describing the solution presented in Figure 8.2 (with $I = \{0\}$) is:

```
g(0,0,0) g(1,0,0) g(2,0,1) g(3,0,0) g(4,0,1) g(5,0,0) g(6,0,0)
new(2,4)
```

First we describe conditions on the labeling $g$ of the vertices. According to (i), $g$ coincides with $f$ where the latter is defined:

```
g(X,C,S) :- f(X,C,S).
```

Every internal vertex should be labeled by exactly one state for each character:

```
1 {g(X,C,S): state(S)} 1 :-
        vertex(X), not leaf(X), character(C).
```

Then, we add at most $k$ pairs of edges between the vertices of this phylogeny:

```
{new(X,Y): vertex(X;Y): X < Y} maxE.
```

(`maxE` represents the value of $k$).

However, due to (iii), we cannot add such a pair between two vertices if adding it will create a cycle that includes at least one edge of the given phylogeny. To express this condition, we first define the extended set of edges $E \cup N$ by the rules

```
an_edge(X,Y) :- edge(X,Y).
an_edge(X,Y) :- new(X,Y), vertex(X;Y).
an_edge(X,Y) :- new(Y,X), vertex(X;Y).
```

Then we define the binary predicate `directed_path` as the transitive closure of `an_edge`, and impose the constraint

```
:- directed_path(X,Y), edge(Y,X).                    (*)
```

Finally, we need to make sure that (ii) holds. For that we use the following proposition:

**Proposition 6** *For any finite digraph $(V, E)$, and any set $V' \subseteq V$, the following conditions are equivalent:*

*(a) there exists a subgraph of $(V, E)$ with the set $V'$ of vertices that is a rooted tree,*

*(b) there exists a vertex $v \in V'$ such that every vertex in $V'$ is reachable from $v$ in $V'$.*

The proof of Proposition 6 is presented in Section 9.12 (page 179).

Proposition 6 shows that condition (ii) in the statement of the MIPPN problem can be equivalently expressed as follows:

(ii′) for all $i \in I$ and $s \in S$, if $V_{is} = \{u \in V : g(u, i) = s\}$ is not empty then there exists a vertex $v$ in $V_{is}$ such that every vertex in $V_{is}$ is reachable from $v$ in $V_{is}$.

We consider pairs $(i, s)$ for which some vertex is labeled by state $s$ at character $i$, so that $V_{is}$ is not empty. After picking an element $v$ of $V_{is}$, and defining reachability of vertices in $V_{is}$ from $v$ by the predicate `reachable`, we express (ii′) by the constraint

```
:- g(X,C,S), not reachable(X,C,S), character(C),
   state(S), vertex(X).
```

The answer sets for the program described above correspond to the solutions of the MIPPN problem. We will call it "the basic program."

We can use SMODELS with the basic program to solve small instances of the MIPPN problem. Larger data sets, such as the one described in Section 8.1.6, require the use of some heuristics and optimization techniques. Some of these techniques are not complete, that is, do not allow us, generally, to find all solutions. We will discuss them in the following sections.

### 8.1.3 Preprocessing

Sometimes the MIPPN problem for a given phylogeny can be simplified by making its set $I$ of characters smaller. If $(V, E, N, g)$ is a perfect network built on a phylogeny $(V, E, f)$ then, for any subset $J$ of $I$, $(V, E, N, g\,|_{L\times J})$ is obviously a perfect network built on $(V, E, f\,|_{L\times J})$. Proposition 7 below shows, for some special choice of $J$, that every perfect network built on $(V, E, f\,|_{L\times J})$ can be extended to a network built on $(V, E, f)$.

Consider a phylogeny $(V, E, f)$, with $f : L \times I \rightarrow S$. We say that a character $j \in I$ is *inessential* if there exists a perfect network $(V, E, \emptyset, g)$ built on $(V, E, f|_{L \times \{j\}})$. For instance, in the phylogeny of Figure 8.3, the first of the two characters is inessential. In the phylogeny of Indo-European languages described in Section 8.1.6, 352 characters out of 370 are inessential.

**Proposition 7** *Let $(V, E, f)$ be a phylogeny, with $f : L \times I \rightarrow S$, and let $I'$ be the set of its inessential characters. There exists a function $g' : V \times I' \rightarrow S$ such that, for every perfect network $(V, E, N, g)$ built on $(V, E, f|_{L \times (I \setminus I')})$, $(V, E, N, g \cup g')$ is a perfect network built on $(V, E, f)$.*

This theorem shows that the sets $N$ in solutions to the MIPPN problem for a phylogeny $(V, E, f)$ are identical to the sets $N$ in the solutions to the same problem for the smaller phylogeny $(V, E, f|_{L \times (I \setminus I')})$ (although the functions $g$ in these solutions are, generally, different).

The proof of Proposition 7 is presented in Section 9.13 (page 180).

Another way to make a given phylogeny smaller is to check whether it has an internal vertex $v$ such that all leaves descending from $v$ are labeled in the same way: for any such leaves $v_1, v_2$, and any character $i$, $f(v_1, i) = f(v_2, i)$. In Figure 8.5, for instance, this condition holds for the common parent of $A1$ and $A2$. If such a vertex $v$ is found, then we remove all its descendants from the tree, so that $v$ turns into a leaf. The labeling of $v$ in the reduced phylogeny $(V', E', f')$ is the same as the common labeling of the descendants of $v$ in the given phylogeny. For instance, this process turns Figure 8.5 into the phylogeny

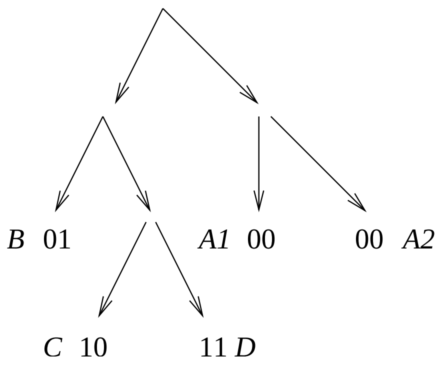


Figure 8.5: A phylogeny.

of Figure 8.3.

In our work on the phylogeny of Indo-European languages with 24 leaves, we repeated this process several times, and reduced the given tree to a tree with 15 leaves. This process improved the computational efficiency of SMODELS. Consider, for instance, the use of SMODELS to find a perfect network for just one character, Character 158, with at most one edge. Pruning reduced the computation time in this case from 1.5 seconds to 0.6 seconds.

Every solution to the MIPPN problem for the pruned phylogeny can be extended to a solution for the original phylogeny:

**Proposition 8** *Let* $(V, E, f)$ *be a phylogeny, with* $f : L \times I \to S$*, and* $(V', E', f')$ *be the phylogeny obtained from it as described above. Let* $(V', E', N, g')$ *be a perfect network built on* $(V', E', f')$*. Then there exists a function* $g$ *from* $V \times I$ *to* $S$ *with* $g|_{V' \times I} = g'$ *such that* $(V, E, N, g)$ *is a perfect network built on* $(V, E, f)$*.*

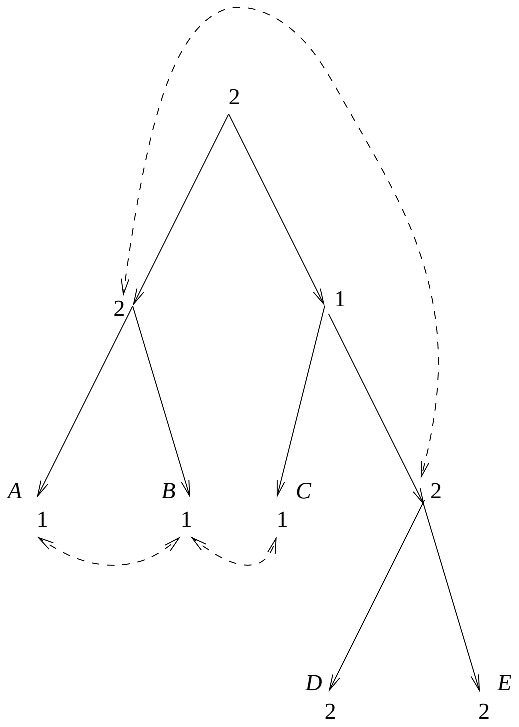


Figure 8.6: A perfect network with $|N| = 3$.

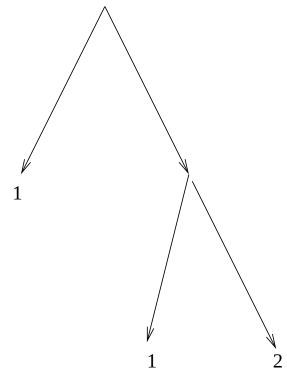


Figure 8.7: The phylogeny obtained from the phylogeny of Figure 8.6 by preprocessing.

However this preprocessing is not complete, i.e., there may be a solution to the MIPPN problem such that its set $N$ of new edges will not be found after pruning vertices. For instance, the perfect network built on the phylogeny of Figure 8.5 in which new edges connect $C$ with $A1$ and with $A2$ will not be generated after $A1$ and $A2$ are removed from the tree.

Also, after preprocessing we may not find a perfect network with $|N| = k$ even when such a perfect network exists before preprocessing. For instance, the perfect network built on the phylogeny of Figure 8.6 has 3 new edges. If we prune the phylogeny of Figure 8.6 as described above, we get the phylogeny of Figure 8.7. Note that there is no perfect network with 3 new edges built on this phylogeny.

On the other hand, it is not known whether after preprocessing we may find a perfect network with $|N| \leq k$ when such a perfect network exists before preprocessing.

The proof of Proposition 8 is presented in Section 9.14 (page 181).

### 8.1.4 Partial Perfect Networks and Essential States

The basic program (Section 8.1.2) can be improved using "partial" perfect phylogenetic networks—a generalization of the "total" version of this concept defined in Section 8.1.1.

Let $(V, E, f)$ be a phylogeny, with $f : L \times I \to S$. A *partial perfect (phylogenetic) network* built on this phylogeny is a quadruple of the form $(V, E, N, g)$, where $N$ is a symmetric irreflexive binary relation on $V$, and $g$ is

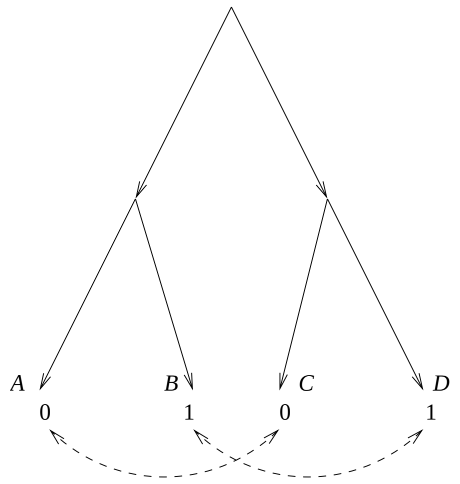


Figure 8.8: A partial perfect network built on the phylogeny of Figure 8.1 with $N = \{(A, C), (C, A), (B, D), (D, B)\},\ g = f$.

a partial mapping of $V \times I$ to $S$ such that the domain of $g$ contains $L \times I$, and conditions (i)–(iii) from Section 8.1.1 are satisfied. For instance, a partial perfect network built on the phylogeny of Figure 8.1 can be defined by

$$N = \{(A, C), (C, A), (B, D), (D, B)\},\ g = f$$

(Figure 8.8).

Every partial perfect network can be extended to a perfect network:

**Proposition 9** *Let $(V, E, f)$ be a phylogeny, with $f : L \times I \to S$. For any partial perfect network $(V, E, N, g)$ built on this phylogeny there exists an extension $g'$ of $g$ to $V \times I$ such that $(V, E, N, g')$ is a perfect network built on the same phylogeny.*

The proof of Proposition 9 is presented in Section 9.15 (page 183).

Proposition 10 below shows, on the other hand, that every perfect network can be obtained by extending a partial perfect network satisfying a certain condition, expressed in terms of "essential states." Let $(V, E, f)$ be a phylogeny, with $f : L \times I \to S$. We say that a state $s \in S$ is *essential* with respect to a character $j \in I$ if there exist two different leaves $l_1$ and $l_2$ in $L$ such that $f(l_1, j) = f(l_2, j) = s$. For instance, in the phylogeny of Figure 8.9, State 3 is essential, and States 1 and 2 are not. There is no need to use inessential states to label internal vertices:

**Proposition 10** *Let $(V, E, f)$ be a phylogeny, with $f : L \times I \to S$. For any perfect network $(V, E, N, g')$ built on this phylogeny there exists a partial mapping $g$ of $V \times I$ to $S$ such that*

- *(V,E,N,g) is a partial perfect network built on the same phylogeny,*
- *$g'$ is an extension of $g$ to $V \times I$, and*
- *$g(v, i)$ is essential with respect to $i$ whenever $v \notin L$.*

For instance, if $(V, E, N, g')$ is the perfect network of Figure 8.9, then the partial perfect network of Figure 8.10 satisfies the conditions of this theorem as $(V, E, N, g)$. In this partial perfect network, inessential states 1 and 2 are not used for labeling internal vertices.

The proof of Proposition 10 is presented in Section 9.16 (page 185).

Propositions 9 and 10 show that the problem of computing a perfect network built on a given phylogeny is closely related to the problem of computing a partial perfect network $(V, E, N, g)$ built on this phylogeny such that

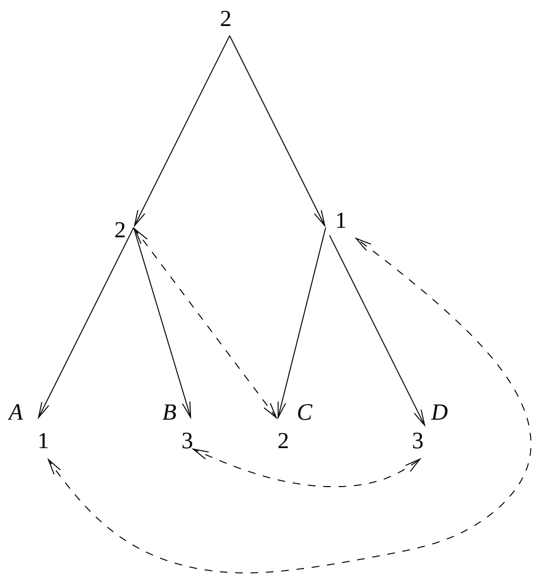


Figure 8.9: A perfect network.

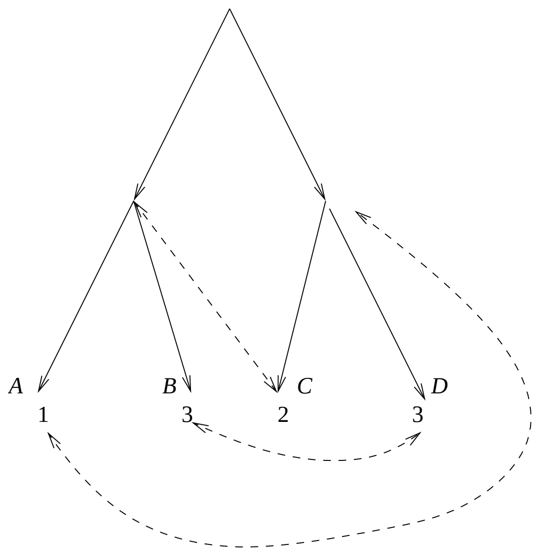


Figure 8.10: A partial perfect network obtained from the perfect network of Figure 8.9 via Proposition 10.

$g(v,i)$ is essential with respect to $i$ for every internal vertex $v$ and every character $i$. The sets $N$ in the solutions to this modification of the problem are identical to the sets $N$ in the solutions to the original problem (although the functions $g$ are, generally, different).

For instance, in the phylogeny of Indo-European languages described in Section 8.1.6, the range of $f$ for Character 39 contains 11 elements from $S$, and only 3 of them are essential. When we consider essential states and want to find a partial network built on the given phylogeny for this character, the computation time reduces from 16 seconds to 5 seconds.

To adapt the basic program (Section 8.1.2) to the modification of the problem described above, we replace

```
1 {g(X,C,S): state(S)} 1 :-
        vertex(X), not leaf(X), character(C).
```

with the rule

```
{g(X,C,S): essential_state(C,S)} 1 :-
        vertex(X), not leaf(X), character(C).
```

where `essential_state` is defined by the rule

```
essential_state(C,S) :- f(X,C,S), character(C),
        f(X1,C,S), X != X1, vertex(X;X1).
```

### 8.1.5 A Divide-and-Conquer Strategy

Unfortunately, even with all the preprocessing discussed above the program representing the Indo-European languages remain far too large for SMODELS. For some machines, even the grounding turned out to be impossible. For some other machines, grounding was successful but SMODELS could not find a solution after many days. This motivated us to divide the problem into smaller problems.

Recall that a perfect network built on a phylogeny $(V, E, f)$ is a quadruple $(V, E, N, g)$ satisfying conditions (i)–(iii) from Section 8.1.1. If $(V, E, N, g)$ satisfies the first two of these conditions, we will call it an *almost perfect network.*

The divide-and-conquer approach we use is based on the following fact: if, for each $j \in I$, $(V, E, N_j, g_j)$ is an almost perfect network built on $(V, E, f\,|_{V\times\{j\}})$ then $(V, E, \bigcup_j N_j, \bigcup_j g_j)$ is an almost perfect network built on $(V, E, f)$. In view of this fact, solutions to the MIPPN problem can be generated by finding such networks $(V, E, N_j, g_j)$ for all characters $j$ and checking that $(V, E, \bigcup_j N_j, \bigcup_j g_j)$ satisfies conditions (iii) and (iv).

Proposition 11 below shows that this process can generate every *minimal* solution—every network $(V, E, N, g)$ in which the set $N$ of new edges cannot be replaced by its proper subset without violating condition (ii).

**Proposition 11** *Let $(V, E, N, g)$ be a minimal almost perfect network built on $(V, E, f)$. For every $j \in I$, there exists a minimal almost perfect network*

$(V, E, N_j, g\,|_{V\times\{j\}})$ *built on* $(V, E, f\,|_{L\times\{j\}})$ *such that* $\bigcup_{j\in I} N_j = N$.

The proof of Proposition 11 is presented in Section 9.17 (page 187).

To compute almost perfect networks $(V, E, N_j, g_j)$ for each character $j$, we use the basic program without the definition of `directed_path` and without constraint `(*)`. We can find all such networks using the `compute all {}` statement of SMODELS. However, there may be more than one such network with the same set $N$ of new edges where the labelings $g$ of the vertices differ; we are only interested in solutions with different sets of new edges. For this reason, we wrote a script that calls SMODELS repeatedly to compute one value of $N$ at a time. To ensure that every next $N_j$ computed by SMODELS is different from the sets computed so far, we add appropriate constraints to the program at every iteration. For instance, if the first call to SMODELS produces `new(1,2), new(3,4)` then the constraint

```
:- new(1,2), new(3,4).
```

will be added to the program when SMODELS is called for the second time, so that SMODELS will now compute an answer set that does not contain {`new(1,2), new(3,4)`}. To compute all minimal almost perfect networks with at most $2k$ new edges, we start with `maxE=0` and increment `maxE` by 1 until it reaches $k$.

The verification of conditions (iii) and (iv) for the networks $(V, E, \bigcup_j N_j, \bigcup_j g_j)$ generated from the almost perfect networks $(V, E, N_j, g_j)$ is performed by SMODELS programs.

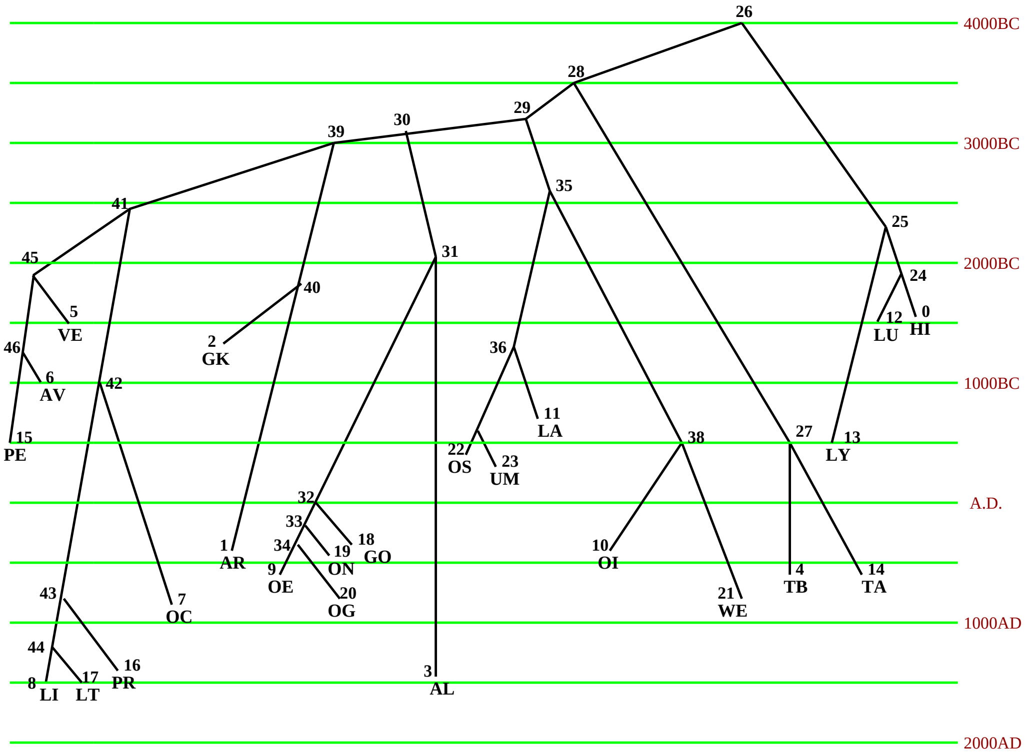


Figure 8.11: The phylogeny obtained from the Indo-European dataset.

This divide-and-conquer strategy can be extended to partial networks (Section 8.1.4) in a straightforward way.

### 8.1.6 The Evolutionary History of the Indo-European Languages

We have applied the computational methods described above to the phylogeny of Indo-European languages that was generated automatically [Ringe *et al.*, 2002] on the basis of a dataset assembled by Donald Ringe and Ann Taylor, who are specialists in Indo-European historical linguistics, with the advice of other specialist colleagues. Figure 8.11 shows the tree $(V, E)$ of this phylogeny. Its leaves correspond to the following 24 Indo-European languages: Hittite (HI), Luvian (LU), Lycian (LY), Tocharian A (TA), Tocharian B (TB), Vedic (VE), Avestan (AV), Old Persian (PE), Classical Armenian (AR), Ancient Greek (GK), Latin (LA), Oscan (OS), Umbrian (UM), Gothic (GO), Old Norse (ON), Old English (OE), Old High German (OG), Old Irish (OI), Welsh (WE), Old Church Slavonic (OC), Old Prussian (PR), Lithuanian (LI), Latvian (LT), and Albanian (AL). All these languages are "historic," that is, recorded, and the position of every leaf vertex against the time line in Figure 8.11 corresponds to the earliest period at which there is substantial attestation of the corresponding language.

The internal vertices of this tree represent "prehistoric" languages, or "protolanguages," which were reconstructed by comparison of their descendants. For instance, Vertex 38 is proto-Celtic, reconstructed by comparison of Old Irish and Welsh. The position of every internal vertex against the time line corresponds to the time period when the corresponding protolanguage split up into daughter languages, each spoken by a different speech community.

There are 370 characters in this phylogeny.[2] Out of 370 characters, 22 are phonological characters encoding regular sound changes that have occurred in the prehistory of various languages, 15 are morphological characters

[2]We disregard the 20 characters that take into account multiple character coding and parallel development.

encoding details of inflection (or, in one case, word formation), and 333 are lexical characters defined by meanings on a basic word list.

The SMODELS program used to generate perfect networks built on this phylogeny incorporated several domain-specific constraints. One of these constraints prohibits new edges incident with Vertex 24 and its descendants; this constraint is justified by the fact that the labelings of the leaves HI, LU and LY are "disjoint," in their essential parts, from the labelings of the other leaves. Other constraints prohibit contacts between specific pairs of languages which, we know, were spoken at different times. For instance, Old Prussian (Vertex 16) could not be in contact with proto-Celtic (Vertex 38).

Using such domain specific constraints with the heuristics and the optimization techniques discussed earlier improves the performance of SMODELS. For instance, consider Character 197. We want to find all minimal almost perfect networks built on the given phylogeny with at most 1 edge. When we do not use domain specific constraints, SMODELS computes 12 solutions in 100 seconds. When we add domain specific constraints, SMODELS computes 9 solutions in 77 seconds. Therefore, using domain specific information SMODELS computes fewer number of implausible solutions, and it does it faster.

Under these constraints, we have found that there are no solutions to the MIPPN problem with fewer than 5 new bidirectional edges. There is only one solution with 5 new edges:

- (32,38) (38,40) (32,43) (18,34) (7,44) .

According to this solution there are 5 borrowings: between proto-Germanic and proto-Celtic (32,38), between proto-Celtic and proto-Greco-Armenian (38,40), between and proto-Germanic and proto-Baltic (32,43), between Gothic and proto-West-Germanic (18,34), and between Old Church Slavonic and proto-East-Baltic (7,44). Each of these contacts is understood to occur at a time prior to the dates assigned to the two languages in the chronology of Figure 8.11. The approximate times of these contacts are shown in Figure 8.12.

We have also computed 52 solutions with 6 new edges; 8 of these solutions do not include any borrowings that would be historically implausible:

- (32,38) (31,36) (27,41) (32,43) (18,34) (7,44)
- (32,38) (31,36) (35,40) (32,43) (18,34) (7,44)
- (32,38) (31,36) (36,40) (32,43) (18,34) (7,44)
- (32,38) (31,36) (27,42) (32,43) (18,34) (7,44)
- (32,38) (31,36) (31,40) (32,43) (18,34) (7,44)
- (32,38) (31,36) (27,45) (32,43) (18,34) (7,44)
- (7,27) (32,38) (38,40) (32,42) (18,34) (7,44)
- (32,38) (38,40) (3,7) (32,42) (18,34) (7,44)

In addition to the phylogeny of Figure 8.11, we considered its modification in which additional internal vertices are introduced—one vertex in

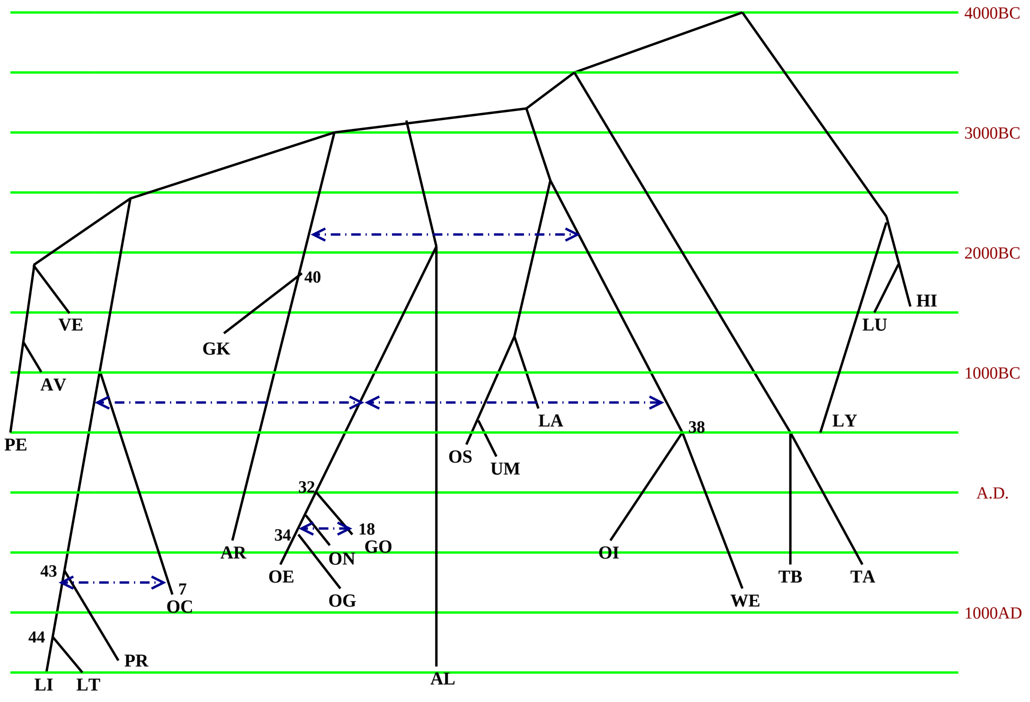


Figure 8.12: Contacts between Indo-European languages according to the 5-edge solution.

the middle of every edge. This extension reflects the possibility of ancestral languages not represented in the original phylogeny, and it was used as the starting point in the work reported in [Nakhleh *et al.*, 2002]. The calculations described in that paper are limited to the case when new edges are inserted between additional vertices only. Nakhleh *et al.* found 2 solutions with 5 new edges that satisfy this restriction and are historically plausible. To facilitate comparison with that work, we included the same restriction in our program.

For the modified problem, we have computed 11 solutions with 5 new edges; we found among them the 2 solutions from [Nakhleh *et al.*, 2002], and also one other historically plausible solution. This solution is very close to the 5-edge solution for the phylogeny of Figure 8.11.

To sum up, the collection of conjectures about the evolutionary history of Indo-European languages generated in our experiments is richer than what is found in [Nakhleh *et al.*, 2002] in two ways. First, we arrived at several historically plausible perfect networks built on the phylogeny of Figure 8.11 without additional vertices; this network was not studied in the earlier work at all. Second, we found one new historically plausible perfect network built on the extended phylogeny. The reason why this network could not be generated by Nakhleh *et al.* is that these authors started their computations with a major simplification step, in which every language group (such as Germanic languages or Baltic languages) is reduced to a single vertex, and 16 of the 18 essential characters are considered. Nakhleh *et al.* found 2 plausible solutions in 8 hours. Our program, with all the heuristics and optimization techniques, and domain specific information, on the pruned phylogeny of [Nakhleh *et al.*, 2002] did not terminate in 8 hours.

## 8.2 Phylogeny Reconstruction in Biology

In biology, phylogenies can be reconstructed from "genomes" [Sankoff and Blanchette, 1998]. The *genome* of a single-chromosome organism can be represented by circular configurations of numbers $1, \ldots, n$, with a sign $+$ or $-$

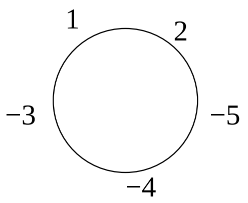


Figure 8.13: A genome.

assigned to each of them. For instance, Figure 8.13 shows a genome for $n = 5$. Numbers $\pm 1, \ldots, \pm n$ will be called *labels*. Intuitively, a label corresponds to a gene, and its sign corresponds to the orientation of the gene. By $(l_1, \ldots, l_n)$ we denote the genome formed by the labels $l_1, \ldots, l_n$ ordered clockwise. For instance, each of the expressions $(1, 2, -5, -4, -3), (2, -5, -4, -3, 1), \ldots$ denotes the genome in Figure 8.13.

About genomes $g, g'$ we say that $g'$ is an *inversion* of $g$ (or *can be obtained from $g$ by an inversion*) if, for some labels $l_1, \ldots, l_n$ and a number $m$ $(0 < m \leq n)$,

$$g = (l_1, \ldots, l_n),$$

$$g' = (-l_{m-1}, \ldots, -l_1, l_{m+1}, \ldots, l_n).$$

For instance, the genome in Figure 8.14 is an inversion of the genome in Figure 8.13.

The *(inversion) distance* between $g$ and $g'$ is the smallest number $k$ such that $g'$ can be obtained from $g$ by $k$ successive inversions.

The *median genome problem* is the problem of finding, for given genomes $g_1, g_2, g_3$, a genome $g$ such that the sum of the distances from $g$ to the genomes

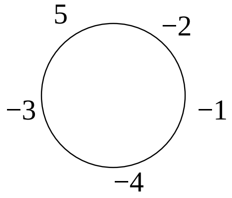


Figure 8.14: Another genome.

$g_i$ is minimal. This problem is proven to be NP-hard [Caprara, 1999]. Phylogenies can be constructed from a set of given genomes by solving the median genome problem for triplets of genomes iteratively in the sense of [Sankoff and Blanchette, 1998]. The problem of constructing a phylogeny is proven to be NP-hard [Caprara, 1999].

We view the median genome problem as a planning problem. Given the initial conditions of the three genomes, we want to reach a state, in at most $k$ steps, where all three genomes become equal and where the number of inversions is minimal.

We represent a genome as an order on the set of labels. An inversion is viewed as the concurrent execution of several "atomic inversions", each involving one or two of labels. For instance, the inversions leading from Figure 8.13 to Figure 8.14 consists of two atomic inversions replacing $1, -5$ by $5, -1$ and replacing 2 by $-2$.

We experimented on three groups of median genome problems. In the first group, each genome is a circular ordering of 5 genes, i.e., $n = 5$, where the maximum length of a plan is 2, i.e., $k = 2$. In the second group, each genome is a circular ordering of 10 genes where the maximum length of a plan is 2. The third group of problems are generated using the simulator implemented by Li-San Wang [Wang and Warnow, 2001]. Here each genome is a circular ordering of 120 genes; the maximum length of a plan is 2. After "condensing", each genome becomes a circular ordering of 11 or 12 genes.

SMODELS cannot solve any of the problems in these three groups in less than an hour whereas the system GRAPPA [Moret *et al.*, 2001] with the software REVMED [Siepel, 2002] can solve all of these problems in less than a second.

When we consider the decision problem corresponding to the median genome problem described above, SMODELS can find solutions to some of the problems in the first group in less than 5 seconds; but still there are some problems in the first group that SMODELS takes more than an hour to solve. SMODELS cannot find a solution to any of the problems of the second and the third groups in less than an hour either.

## 8.3 Discussion

The Minimum Increment to Perfect Phylogenetic Network problem discussed above is a combinatorial search problem well suited to the use of answer set programming. But the basic SMODELS program for the MIPPN problem will produce solutions reasonably fast only if the given phylogeny is small. Several ideas helped us adapt this program to a large phylogeny of Indo-European languages.

One idea is to make the given phylogeny smaller by removing its inessential characters and some of its vertices. This kind of preprocessing is somewhat similar to reducing every language group to a single vertex in [Nakhleh *et al.*, 2002]. The difference is that our preprocessing is domain-independent—it is defined for any phylogeny. Second, there is no need to use inessential states for labeling the internal vertices of the tree. Finally, a divide-and-conquer strategy allowed us to replace a single run of SMODELS by a series of invocations that solve subproblems of the given problem.

The input program included some domain-specific information about the impossibility of contacts between languages spoken at different times. These constraints helped us further reduce the computation time. Even so, computing the results reported in this paper involved thousands of calls to SMODELS and took more than a week of CPU time. In spite of the presence of several constraints of this kind, most perfect networks computed by our program turned out to be impossible or implausible for historical reasons. To weed out unacceptable solutions, we had to analyze each solution carefully on the basis of the conclusions of earlier research in historical linguistics.

On the other hand, our attempts to apply ASP to the median genome problem were mostly unsuccessful.

# Chapter 9

# Proofs

## 9.1 Proof of Lemma 1

**Lemma 1 (page 47)** *A binary relation $R$ is well-founded iff there exists a function $\lambda$ from the domain of $R$ to ordinals such that, for all $x$ and $y$, $xRy$ implies $\lambda(x) < \lambda(y)$.*

**Proof.** The "if" part follows from the well-foundedness of $<$ on sets of ordinals. To prove the "only if" part, consider the following transfinite sequence of subsets of the domain of $R$:

$$S_0 = \emptyset,$$
$$S_{\alpha+1} = \{x : \ \forall y(yRx \Rightarrow y \in S_\alpha)\},$$
$$S_\alpha = \textstyle\bigcup_{\beta<\alpha} S_\beta \quad \text{if } \alpha \text{ is a limit ordinal.}$$

For any $x \in \bigcup_\alpha S_\alpha$, define $\lambda(x)$ to be the smallest $\alpha$ such that $x \in S_\alpha$. From the well-foundedness of $R$ we can conclude that $\bigcup_\alpha S_\alpha$ is the whole domain of

$R$.

## 9.2 Proof of Proposition 1

**Proposition 1 (page 46)** *A program* $\Pi$ *is tight iff there is no infinite sequence* $L_0, L_1, \ldots$ *of elements of* $lit(\Pi)$ *such that for every* $i$*,* $L_{i+1}$ *is a parent of* $L_i$ *relative to* $\Pi$*.*

**Proof.** Proposition 1 is a special case of Lemma 1: take the domain of $R$ to be $lit(\Pi)$.

## 9.3 Proof of Theorem 1

**Theorem 1 (page 67)** *For any program* $\Pi$ *and any atom* $p$*, the programs*

$$\begin{array}{l} \Pi \\ p; \neg p \leftarrow \end{array} \tag{9.1}$$

*and*

$$\begin{array}{l} \Pi \\ p \leftarrow not\ \neg p \\ \neg p \leftarrow not\ p \end{array} \tag{9.2}$$

*have the same answer sets.*

We will use the following lemma to prove Theorem 1. In the proof of Theorem 1, we denote program (9.1) by $\Pi_1$ and program (9.2) by $\Pi_2$.

**Lemma 2** *For any consistent set $X$ of literals and any subset $Y$ of $X$, $Y$ is closed under $\Pi_1^X$ iff $Y$ is closed under $\Pi_2^X$.*

**Proof of Theorem 1.** Let $X$ be a consistent set of literals. We want to show that $X$ is an answer set for $\Pi_1^X$ iff $X$ is an answer set for $\Pi_2^X$. Recall that a consistent set $X$ of literals is an answer set for a disjunctive program without negation as failure iff it is a minimal set closed under this program.

Notice first that $X$ is closed under $\Pi_1^X$ iff $X$ is closed under $\Pi_2^X$, due to Lemma 2 for $Y = X$. It remains to check that

(a) $X$ is minimal among the sets closed under $\Pi_1^X$

iff

(b) $X$ is minimal among the sets closed under $\Pi_2^X$.

Assume (a), and consider any subset $Y$ of $X$ that is closed under $\Pi_2^X$. Due to Lemma 2, $Y$ is closed under $\Pi_1^X$ as well. By (a), it follows that $X = Y$. The proof in the other direction is similar.

**Proof of Lemma 2.** Let $X$ be a consistent set of literals, and let $Y$ be a subset of $X$. Since $X$ is consistent, at least one of the literals $p$, $\neg p$ does not belong to $X$. Consider 3 cases.

*Case 1:* $p \notin X$ and $\neg p \notin X$. Then $\Pi_1^X$ is

$$\Pi^X$$
$$p; \neg p \leftarrow$$

and $\Pi_2^X$ is

$$\Pi^X$$

$$p \leftarrow$$

$$\neg p \leftarrow$$

Since $Y \subseteq X$, $p \notin Y$ and $\neg p \notin Y$. Consequently, $Y$ is closed neither under $\Pi_1^X$ nor under $\Pi_2^X$.

*Case 2:* $p \in X$ and $\neg p \notin X$. Then $\Pi_1^X$ is

$$\Pi^X$$

$$p; \neg p \leftarrow$$

and $\Pi_2^X$ is

$$\Pi^X$$

$$p \leftarrow$$

The "if part": assume that $Y$ is closed under $\Pi_2^X$. Then $Y$ is closed under $\Pi^X$. In addition, $p$ is in $Y$. Therefore, $Y$ is also closed under $\Pi_1^X$. The "only if" part: assume that $Y$ is closed under $\Pi_1^X$. Then $Y$ is closed under $\Pi^X$. As $\neg p$ is not in $X$ due to the case assumption, and $Y \subseteq X$, $\neg p$ is not in $Y$ either. Then $p$ is in $Y$. Therefore, $Y$ is closed under $\Pi_2^X$ also.

*Case 3:* $p \notin X$ and $\neg p \in X$. Similar to Case 2.

## 9.4 Proof of Theorem 2

**Theorem 2 (page 68)** *If $X$ is an answer set for the program $\Pi_1$ obtained from $\Pi$ by adding the rules*

$$p_i(c_i) \leftarrow p(c_1, \ldots, c_n) \tag{9.3}$$

$$\leftarrow p_i(c), p_i(c') \tag{9.4}$$

*($1 \leq i \leq n$, $c_1 \in C_1, \ldots, c_n \in C_n$, $c, c' \in C_i$) then*

$$X \cap \mathit{lit}(\Pi) \tag{9.5}$$

*is an answer set for the program $\Pi_2$ obtained from $\Pi$ by adding the rules*

$$\begin{array}{l} \leftarrow p(c_1, \ldots, c_n), p(c'_1, \ldots, c'_n) \\ \qquad (c_1, c'_1 \in C_1,\ \ldots, c_n, c'_n \in C_n, \langle c_1, \ldots, c_n \rangle \neq \langle c'_1, \ldots, c'_n \rangle). \end{array} \tag{9.6}$$

*Moreover, every answer set for $\Pi_2$ can be represented in form (9.5) for some answer set $X$ for $\Pi_1$.*

The proof of Theorem 2 is based on two facts. One is the splitting set theorem [Lifschitz and Turner, 1994]. The other is a property of constraints that easily follows from the definition of an answer set: the effect of adding a set of constraints to a program is to eliminate the answer sets that are not closed under these constraints.

Recall that in the statement of Theorem 2, $\Pi$ is a program, $C_1, \ldots, C_n$ ($n > 0$) are sets, and $p$ is a function such that for all $c_1 \in C_1, \ldots, c_n \in C_n$ its values $p(c_1, \ldots, c_n)$ are pairwise distinct atoms in the language of $\Pi$. The expressions $p_i(c)$, where $1 \leq i \leq n$ and $c \in C_i$, are assumed to be pairwise distinct atoms that do not belong to the language of $\Pi$.

For any subset $Y$ of $lit(\Pi)$, we define:

$$Y^* = \{p_i(c_i) : c_1 \in C_1, \ldots, c_n \in C_n,\ p(c_1, \ldots, c_n) \in Y\}.$$

Clearly

$$Y^* \cap lit(\Pi) = \emptyset.$$

**Lemma 3** *A consistent set $X$ of literals is an answer set for the union of* $\Pi$ *with rules (9.3) iff $X = Y \cup Y^*$ for some consistent answer set $Y$ for* $\Pi$.

**Proof of Theorem 2.** Let $X$ be an answer set for $\Pi_1$. Then $X$ is an answer set for the union of $\Pi$ with rules (9.3). As any answer set for a program with constraints without negation as failure, $X$ is consistent. By Lemma 3, $X$ can be represented as $Y \cup Y^*$, where $Y$ is an answer set for $\Pi$. It is clear that

$$X \cap lit(\Pi) = Y. \tag{9.7}$$

We will show that $Y$ is an answer set for $\Pi_2$. As $Y$ is an answer set for $\Pi$, it is sufficient to show that $Y$ does not violate any of constraints (9.6). Assume that it does. That means that $Y$ contains a pair of distinct atoms $p(c_1, \ldots, c_n)$ and $p(c'_1, \ldots, c'_n)$. Take an $i$ such that $c_i \neq c'_i$. Both $p_i(c_i)$ and $p_i(c'_i)$ are in $Y^*$, and consequently in $X$. It follows that $X$ violates (9.4), contrary to the assumption that it is an answer set for $\Pi_1$.

Now we will show that any consistent answer set $Y$ for $\Pi_2$ can be represented as $X \cap lit(\Pi)$ for some consistent answer set $X$ for $\Pi_1$. Take $X = Y \cup Y^*$. By (9.7), to complete the proof, we only need to show that $X$ is a consistent answer set for $\Pi_1$. By Lemma 3, $X$ is a consistent answer set for the union of $\Pi$ with rules (9.3). Assume that $X$ violates constraints (9.4). Then $X$ contains a pair of atoms $p_i(c)$, $p_i(c')$ with $c \neq c'$. Since $p_i(c) \in X = Y \cup Y^*$ and $Y \subseteq lit(\Pi)$, it follows that $p_i(c) \in Y^*$. This means that $c = c_i$ for some atom $p(c_1, \ldots, c_n)$ in $Y$. Similarly, $c' = c'_i$ for some atom $p(c'_1, \ldots, c'_n)$ in $Y$. It follows that $Y$ violates (9.6), contrary to the assumption that it is an answer set for $\Pi_2$.

**Proof of Lemma 3.** Take $U = lit(\Pi)$. This set splits the union of $\Pi$ with rules (9.3), and $\Pi$ is the bottom of this union relative to $U$. By the splitting set theorem, $X$ is an answer set for the union program iff $X$ can be represented as a union of an answer set $Y$ for $\Pi$ with an answer set for program $e_U(\Pi, Y)$. The latter consists of the rules

$$p_i(c_i) \leftarrow$$

for all $c_1, \ldots, c_n$ such that

$$p(c_1, \ldots, c_n) \in Y.$$

Consequently, its only answer set is $Y^*$.

## 9.5 Proof of Proposition 2

**Proposition 2 (page 76)** *A program $\Pi$ is tight on a set $X$ of literals iff there is no infinite sequence $L_0, L_1, \dots$ of elements of $X$ such that for every $i$, $L_{i+1}$ is a parent of $L_i$ relative to $\Pi$ and $X$.*

**Proof.** Proposition 2 is a special case of Lemma 1: take the domain of $R$ to be $X$.

## 9.6 Proof of Theorem 3

**Theorem 3 (page 77)** *For any program $\Pi$ and any consistent set $X$ of literals such that $\Pi$ is tight on $X$, $X$ is an answer set for $\Pi$ iff $X$ is closed under and supported by $\Pi$.*

We use the following lemmas to prove Theorem 3. Recall that we consider nondisjunctive programs.

**Lemma 4** *For any program $\Pi$ without negation as failure and any consistent set $X$ of literals such that $\Pi$ is tight on $X$, if $X$ is an answer set for $\Pi$ then $X$ is closed under and supported by $\Pi$.*

**Lemma 5** *For any program $\Pi$, and any consistent set $X$ of literals,*

*(i) $X$ is closed under $\Pi$ iff $X$ is closed under $\Pi^X$;*

*(ii) $X$ is supported by $\Pi$ iff $X$ is supported by $\Pi^X$.*

**Lemma 6** *For any program* $\Pi$ *without negation as failure and any consistent set* $X$ *of literals such that* $\Pi$ *is tight on* $X$*, if* $X$ *is closed under and supported by* $\Pi$*, then* $X$ *is an answer set for* $\Pi$.

**Proof of Theorem 3.**

Let $\Pi$ be a program and $X$ be a consistent set of literals. Assume that $\Pi$ is tight on $X$. We want to show that $X$ is an answer set for $\Pi$ iff $X$ is closed under and supported by $\Pi$. *Left-to-right:* assume that $X$ is an answer set for $\Pi$. By the definition of an answer set, $X$ is an answer set for $\Pi^X$. By Lemma 4, $X$ is closed under and supported by $\Pi^X$. Then, by Lemma 5, $X$ is closed under and supported by $\Pi$. *Right-to-left:* assume that $X$ is closed under and supported by $\Pi$. Then, by Lemma 5, $X$ is closed under and supported by $\Pi^X$. Since $\Pi$ is tight on $X$, by the definition of tightness and by the definition of reduct, so is $\Pi^X$. Hence, by Lemma 6, $X$ is an answer set for $\Pi^X$, and consequently an answer set for $\Pi$.

**Proof of Lemma 4.** Let $\Pi$ be a program without negation as failure and $X$ be an answer set for $\Pi$. By the definition of an answer set for programs without negation as failure, $X$ is closed under $\Pi$. To prove supportedness, take any literal $L$ in $X$. Since $X$ is minimal among the sets closed under $\Pi$, $X \setminus \{L\}$ is not closed under $\Pi$. This means that $\Pi$ contains a rule (3.17) with $n = m$ such that $L_1, \ldots, L_m \in X \setminus \{L\}$, but $Head \notin X \setminus \{L\}$. Then, $L_1, \ldots, L_m \in X$, and, since $X$ is closed under $\Pi$, it follows that $Head \in X$. Therefore, $Head = L$.

**Proof of Lemma 5.** Immediately follows from the definitions of closure, supportedness, and the definition of a reduct of a program.

**Proof of Lemma 6.** We need to show that $X$ is minimal among the sets closed under $\Pi$. Assume that it is not. Let $Y$ be a proper subset of $X$ that is closed under $\Pi$, and let $\lambda$ be a partial level mapping establishing that $\Pi$ is tight on $X$. Take a literal $L \in X \setminus Y$ such that $\lambda(L)$ is minimal. Since $X$ is supported by $\Pi$, there is a rule

$$L \leftarrow L_1, \ldots, L_m$$

in $\Pi$ such that $L_1, \ldots, L_m \in X$. By the choice of $\lambda$,

$$\lambda(L_1), \ldots, \lambda(L_m) < \lambda(L).$$

By the choice of $L$, we can conclude that $L_1, \ldots, L_m \in Y$. Consequently $Y$ is not closed under $\Pi$, contrary to the choice of $Y$.

## 9.7 Proof of Proposition 3

**Proposition 4 (page 84)** *Program $BW$ is tight on each of the models of its completion.*

We use the following lemma to prove Proposition 3

**Lemma 7** *For any atom of the form* `nextstate(Y,X)` *that belongs to a model of the completion of $BW$,* $\mathtt{Y} = \mathtt{X} + 1$.

**Proof of Proposition 3.** Let $X$ be an answer set for $BW$. By $T_{max}$ we denote the largest argument of `time/1` in its definition (i). Consider the partial level mapping $\lambda$ with domain $X$ defined as follows:

$$\lambda(\texttt{time(T)}) = 0,$$

$$\lambda(\texttt{block(X)}) = 0,$$

$$\lambda(\texttt{object(X)}) = 1,$$

$$\lambda(\texttt{nextstate(Y,X)}) = 1,$$

$$\lambda(\texttt{covered(X,T)}) = 4 \cdot \texttt{T} + 3,$$

$$\lambda(\texttt{on\_something(X,T)}) = 4 \cdot \texttt{T} + 3,$$

$$\lambda(\texttt{available(X,T)}) = 4 \cdot \texttt{T} + 4,$$

$$\lambda(\texttt{moveop(X,Y,T)}) = 4 \cdot \texttt{T} + 5,$$

$$\lambda(\texttt{on(X,Y,T)}) = 4 \cdot \texttt{T} + 2,$$

$$\lambda(\texttt{moving(X,T)}) = 4 \cdot \texttt{T} + 6,$$

$$\lambda(\texttt{goal(T)}) = 4 \cdot \texttt{T} + 3,$$

$$\lambda(\texttt{blocked\_move(X,Y,T)}) = 4 \cdot \texttt{T} + 7,$$

$$\lambda(\texttt{goal}) = 4 \cdot T_{max} + 4.$$

This level mapping satisfies the inequality from the definition of a tight program (page 75) for every rule of $BW$; Lemma 7 above allows us to verify this assertion for the rules containing `nextstate` in the body.

**Proof of Lemma 7.** The completion of Program $BW$ contains

$$nextstate(Y, X) \equiv false$$

for all $Y$, $X$ such that $Y \neq X + 1$. Therefore, atoms `nextstate(Y,X)` with $\texttt{Y} \neq \texttt{X} + 1$ do not occur in any model of the completion of Program $BW$.

## 9.8 Proof of Theorem 4

**Theorem 4 (page 98)** *Let* $\Pi$ *be a program that does not contain atoms of the form* $tc(x,y)$ *in the heads of rules. If* $X$ *is an answer set for* $\Pi \cup \mathit{Def}$ *then*

$$\{\langle x,y\rangle : tc(x,y) \in X\} \tag{9.8}$$

*is the transitive closure of*

$$\{\langle x,y\rangle : p(x,y) \in X\}. \tag{9.9}$$

**Proof.** We will first prove the special case of Theorem 4 when $\Pi$ doesn't contain negation as failure. Let $X$ be an answer set for $\Pi \cup \mathit{Def}$; denote set (9.9) by $R$, and its transitive closure by $R^\infty$. We need to prove that for all $x$ and $y$, $tc(x,y) \in X$ iff $\langle x,y\rangle \in R^\infty$.

*Left-to right.* Since there is no negation as failure in $\Pi$, $X$ can be characterized as the union $\bigcup_i X_i$ of the sequence of sets of literals defined as follows: $X_0 = \emptyset$; $X_{i+1}$ is the set of all literals $L$ such that $\Pi \cup \mathit{Def}$ contains a rule $L \leftarrow \mathit{Body}$ with $\mathit{Body} \subseteq X_i$. We will show by induction on $i$ that $tc(x,y) \in X_i$ implies $\langle x,y\rangle \in R^\infty$. If $i=0$, the assertion is trivial because $X_0 = \emptyset$. Assume that for all $x$ and $y$, $tc(x,y) \in X_i$ implies $\langle x,y\rangle \in R^\infty$, and take an atom $tc(x,y)$ from $X_{i+1}$. Take a rule $tc(x,y) \leftarrow \mathit{Body}$ in $\Pi \cup \mathit{Def}$ such that $\mathit{Body} \subseteq X_i$. Since $\Pi$ doesn't contain atoms of the form $tc(x,y)$ in the heads of rules, this rule belongs to $\mathit{Def}$. *Case 1:* $\mathit{Body} = \{p(x,y)\}$. Then $p(x,y) \in X_i \subseteq X$, so that $\langle x,y\rangle \in R \subseteq R^\infty$. *Case 2:* $\mathit{Body} = \{p(x,v), tc(v,y)\}$.

Then $p(x,v) \in X_i \subseteq X$, so that $\langle x,v\rangle \in R \subseteq R^\infty$; also, $tc(v,y) \in X_i$, so that, by the induction hypothesis, $\langle v,y\rangle \in R^\infty$. By the transitivity of $R^\infty$, it follows that $\langle x,y\rangle \in R^\infty$.

*Right-to-left.* Since $R^\infty = \bigcup_{j>0} R^j$, it is sufficient to prove that for all $j > 0$, $\langle x,y\rangle \in R^j$ implies $tc(x,y) \in X$. The proof is by induction on $j$. When $j = 1$, $\langle x,y\rangle \in R$, so that $p(x,y) \in X$; since $X$ is closed under $Def$, it follows that $tc(x,y) \in X$. Assume that for all $x$ and $y$, $\langle x,y\rangle \in R^j$ implies $tc(x,y) \in X$, and take a pair $\langle x,y\rangle$ from $R^{j+1}$. Take $v$ such that $\langle x,v\rangle \in R$ and $\langle v,y\rangle \in R^j$. Then $p(x,v) \in X$ and, by the induction hypothesis, $tc(v,y) \in X$. Since $X$ is closed under $Def$, it follows that $tc(x,y) \in X$.

We have proved the assertion of Theorem 4 for programs without negation as failure. Now let $\Pi$ be any program that does not contain atoms of the form $tc(x,y)$ in heads of rules, and let $X$ be an answer set for $\Pi \cup Def$. Clearly, the reduct $\Pi^X$ is a program without negation as failure that does not contain atoms of the form $tc(x,y)$ in the heads of rules, and $X$ is an answer set for $\Pi^X \cup Def$. By the special case of the theorem proved above, applied to $\Pi^X$, (9.8) is the transitive closure of (9.9).

## 9.9 Proof of Theorem 5

**Theorem 5 (page 99)** *Let $\Pi$ be a program that does not contain atoms of the form $tc(x,y)$ in the heads of rules. For any set $X$ of literals, if*

*(i)* $\Pi$ *is tight on* $X$,

*(ii)* $\{\langle x, y\rangle : \ p(y, x) \in X\}$ *is well-founded, and*

*(iii)* *no atom of the form* $tc(x, y)$ *is an ancestor of an atom of the form* $p(x, y)$ *relative to* $\Pi$ *and* $X$*,*

*then* $\Pi \cup \mathit{Def}$ *is tight on* $X$*. If, in addition,*

*(iv)* $X$ *is a consistent set closed under and supported by* $\Pi \cup \mathit{Def}$

*then* $X$ *is an answer set for* $\Pi \cup \mathit{Def}$*, and* $\{\langle x, y\rangle : tc(x, y) \in X\}$ *is the transitive closure of* $\{\langle x, y\rangle : p(x, y) \in X\}$.

**Proof.** Assume (i)–(iii). To prove the first assertion of Theorem 5,suppose that $\Pi \cup \mathit{Def}$ is not tight on $X$. By Proposition 2, there is an infinite sequence $L_0, L_1, \ldots \in X$ such that for every $i$, $L_{i+1}$ is a parent of $L_i$ relative to $\Pi \cup \mathit{Def}$ and $X$. Consider two cases.

*Case 1:* Sequence $L_0, L_1, \ldots$ contains only a finite number of terms of the form $tc(x, y)$. Let $L_n$ be the last of them. Then for every $i > n$, $L_{i+1}$ is a parent of $L_i$ relative to $\Pi$ and $X$. Proposition 2, applied to sequence $L_{n+1}, L_{n+2}, \ldots$, shows that $\Pi$ is not tight on $X$, contrary to (i).

*Case 2:* Sequence $L_0, L_1, \ldots$ contains infinitely many terms of the form $tc(x, y)$. By (iii), it follows that this sequence has no terms of the form $p(x, y)$. The examination of rules $\mathit{Def}$ shows that every $tc(x, y)$ in this sequence is immediately followed by a term of the form $tc(v, y)$ such that $p(x, v) \in X$. Consequently, sequence $L_0, L_1, \ldots$ consists of some initial segment followed by an infinite sequence of literals of the form

$$tc(v_0, y), tc(v_1, y), \ldots$$

such that, for every $i$, $p(v_i, v_{i+1}) \in X$. This is impossible by (ii).

The second assertion of Theorem 5 follows from the first, in view of Theorem 3, and by Theorem 4.

## 9.10 Proof of Proposition 4

**Proposition 4 (page 100)** *If* $\Pi$ *contains constraint*

$$\bot \leftarrow tc(x, x) \tag{9.10}$$

*and* $C$ *is finite then, for every set* $X$ *of literals closed under* $\Pi \cup \mathit{Def}$*, set* $\{\langle x, y\rangle : \ p(y, x) \in X\}$ *is well-founded.*

Let $\Pi$ be a program containing constraint (9.10), with finite $C$, and let $X$ be a set of literals closed under $\Pi \cup \mathit{Def}$. Assume that $\{\langle x, y\rangle : \ p(y, x) \in X\}$ is not well-founded. Take $x_1, \ldots, x_n \in C$ that satisfy (6.3). Since $X$ is closed under $\mathit{Def}$, $tc(x_1, x_1) \in X$. But this is impossible because $X$ is closed under (9.10).

## 9.11 Proof of Proposition 5

**Proposition 5 (page 102)** *The program consisting of rules (3.28), (3.21)–(3.26) and (3.31)–(3.33) is tight on every set of literals that is closed under* *it*.

Let $\Pi$ be the program that differs from (3.28), (3.21)–(3.26) and (3.31)–(3.33) in that

- its underlying set of atoms includes, additionally, expressions of the forms $on(table, l, t)$ and $above(table, l, t)$, and
- rules (3.31) and (3.32) are replaced by

$$\begin{aligned} above(l, l', t) &\leftarrow on(l, l', t), \\ above(l, l', t) &\leftarrow on(l, l'', t), above(l'', l', t) \end{aligned} \tag{9.11}$$

  and

$$\bot \leftarrow above(l, l, t). \tag{9.12}$$

Let $X$ be a set of literals that does not contain any of the newly introduced atoms or their negations and is closed under the original program consisting of (3.28), (3.21)–(3.26) and (3.31)–(3.33). We will prove that $\Pi$ is tight on $X$. It will follow then that the original program is tight on $X$ as well, because that program is a subset of $\Pi$.

For every $k = 0, \ldots, T + 1$, let $\Pi_k$ be the subset of the rules of $\Pi$ in which rules (9.11) are restricted to $t < k$. Since $\Pi_{k+1} = \Pi$, it is sufficient to prove that, for all $k$, $\Pi_k$ is tight on $X$. The proof is by induction on $k$. *Basis:* $k = 0$. The rules of $\Pi_0$ are (3.28), (3.21)–(3.26), (3.33) and (9.12). To see that this program is tight, define

$$\lambda(on(l,l',t)) = t+1,$$
$$\lambda(\neg on(l,l',t)) = t+2,$$
$$\lambda(move(b,l,t)) = \lambda(\neg move(b,l,t)) = 0,$$
$$\lambda(above(l,l',t)) = \lambda(\neg above(l,l',t)) = 0.$$

*Induction step:* Assume that $\Pi_k$ is tight on $X$. Let $C$ be the set of location constants, and let functions $p$ and $tc$ be defined by

$$p(l,l') = on(l,l',k+1),$$
$$tc(l,l') = above(l,l',k+1).$$

Then $\Pi_{k+1} = \Pi_k \cup Def$. Let us check that all conditions of Theorem 5 are satisfied. Condition (i) holds by the induction hypothesis. Since $X$ is closed under the original program consisting of (3.28), (3.21)–(3.26) and (3.31)–(3.33) and does not contain any of the newly introduced literals, it is closed under $\Pi_{k+1}$ as well; in view of the fact that $\Pi_{k+1}$ contains constraint (9.12), condition (ii) follows by Proposition 4. By inspection, (iii) holds also. By Theorem 5, it follows that $\Pi_{k+1}$ is tight on $X$.

## 9.12 Proof of Proposition 6

**Proposition 6 (page 141)** *For any finite digraph $(V,E)$, and any set $V' \subseteq V$, the following conditions are equivalent:*

*(a) there exists a subgraph of $(V,E)$ with the set $V'$ of vertices that is a rooted tree,*

(b) *there exists a vertex $v \in V'$ such that every vertex in $V'$ is reachable from $v$ in $V'$.*

**Proof.** Let $(V, E)$ be a finite digraph, and $V'$ be a subset of $V$. We want to show that conditions (a) and (b) are equivalent. Assume (a). Then, (b) trivially holds by the definition of a rooted tree. Assume (b). Take a subset $E'$ of $E$ minimal relative to set inclusion subject to the condition that every vertex in $V'$ is reachable from $v$ in the subgraph $(V', E')$. Then the in-degree of $v$ is 0, and the in-degrees of other vertices in $V'$ are 1. Therefore, the subgraph $(V', E')$ of $(V, E)$ is a rooted tree with the root $v$, that is, (a) holds.

## 9.13 Proof of Proposition 7

**Proposition 7 (page 143)** *Let $(V, E, f)$ be a phylogeny, with $f : L \times I \to S$, and let $I'$ be the set of its inessential characters. There exists a function $g' : V \times I' \to S$ such that, for every perfect network $(V, E, N, g)$ built on $(V, E, f|_{L\times(I\setminus I')})$, $(V, E, N, g \cup g')$ is a perfect network built on $(V, E, f)$.*

The following facts will be useful to prove Proposition 7:

**Fact 1** *If $(V, E, N, g)$ is an almost perfect network built on $(V, E, f)$ then, for every $j \in I$, $(V, E, N, g|_{V\times\{j\}})$ is an almost perfect network built on $(V, E, f|_{L\times\{j\}})$.*

**Fact 2** *If, for each $j \in I$, $(V, E, N_j, g_j)$ is an almost perfect network built on $(V, E, f|_{V\times\{j\}})$ then $(V, E, \bigcup_j N_j, \bigcup_j g_j)$ is an almost perfect network built on*

$(V, E, f)$.

**Proof of Proposition 7.** Let $(V, E, f)$ be a phylogeny, with $f : L \times I \to S$. Let $I'$ be the set of its inessential characters. Then, for every $j \in I'$, there exists a perfect network of the form $(V, E, \emptyset, g'_j)$ built on $(V, E, f\,|_{L\times\{j\}})$. Define $g' = \bigcup_j g'_j$. Let $(V, E, N, g)$ be a perfect network built on $(V, E, f\,|_{L\times(I\setminus I')})$. Then, by Fact 1, for every $j \in I \setminus I'$, $(V, E, N, g\,|_{V\times\{j\}})$ is an almost perfect network built on $(V, E, f\,|_{L\times\{j\}})$. Then, by Fact 2, $(V, E, N, g \cup g')$ is an almost perfect network built on $(V, E, f)$. Since $(V, E, N, g)$ is a perfect network built on $(V, E, f\,|_{L\times(I\setminus I')})$, for every edge $(u, v) \in E$, $u$ is not reachable from $v$ in $(V, E \cup N)$. It follows that, $(V, E, N, g \cup g')$ is a perfect network built on $(V, E, f)$.

## 9.14 Proof of Proposition 8

**Proposition 8 (page 144)** *Let $(V, E, f)$ be a phylogeny, with $f : L \times I \to S$, and $(V', E', f')$ be the phylogeny obtained from it as described in Section 8.1.3. Let $(V', E', N, g')$ be a perfect network built on $(V', E', f')$. Then there exists a function $g$ from $V \times I$ to $S$ with $g\,|_{V'\times I}= g'$ such that $(V, E, N, g)$ is a perfect network built on $(V, E, f)$.*

**Proof.** Let $(V, E, f)$ be a phylogeny with $f : L \times I \to S$, $v$ be an internal vertex of $(V, E)$ such that all leaves descending from $v$ are labeled in the same way. Let $(V', E', f')$ be the phylogeny, with $f' : L' \times I \to S$, obtained from

$(V, E, f)$ by removing all descendants of $v$ from the tree $(V, E)$, and labeling $v$ in the same way as its descendants were labeled in $(V, E, f)$. Let $(V', E', N, g')$ be a perfect network built on $(V', E', f')$. Define the function $g : V \times I \to S$ recursively as follows:

- $g(w, j) = g'(w, j)$ if $w \in V'$, and
- $g(w, j) = g(u, j)$ if $w \in V \setminus V'$ and $(u, w) \in E$.

We want to show that $(V, E, N, g)$ is a perfect network built on $(V, E, f)$, i.e., that conditions (i)–(iii) hold for $(V, E, N, g)$.

(i) Since every descendant of $v$ is labeled in the same way as $v$, by the definition of $g$, $g(w, j) = f(w, j)$ for all $w \in L$ and for all $j \in I$.

(ii) Take any $j \in I$ and any $s \in S$. Suppose that $V_{js} = \{u \in V : g(u, j) = s\}$ is not empty. Then, due to the definition of $g$, $V'_{js} = \{u \in V : g'(u, j) = s\}$ is not empty. Since $(V', E', N, g')$ is a perfect network built on $(V', E', f')$, $(V', E' \cup N)$ has a subgraph with the set $V'_{js}$ of vertices that is a rooted tree. Then, by the definition of $g$, $(V, E \cup N)$ has a subgraph with the set $V_{js}$ of vertices that is a rooted tree.

(iii) We know that no cycle in $(V', E' \cup N)$ contains an edge from $E'$. It follows that no cycle in $(V, E \cup N)$ contains an edge from $E$: this graph can be obtained from $(V', E' \cup N)$ by attaching a tree to vertex $v$, and this operation does not create new cycles.

## 9.15 Proof of Proposition 9

**Proposition 9 (page 147)** *Let $(V, E, f)$ be a phylogeny, with $f : L \times I \to S$. For any partial perfect network $(V, E, N, g)$ built on this phylogeny there exists an extension $g'$ of $g$ to $V \times I$ such that $(V, E, N, g')$ is a perfect network built on the same phylogeny.*

**Proof.** Let $(V, E, f)$ be a phylogeny, with $f : L \times I \to S$. Let $(V, E, N, g)$ be a partial perfect network built on $(V, E, f)$. We want to show that there exists a total function $g'$ from $V \times I$ to $S$ with $g'(v, j) = g(v, j)$ for every $(v, j)$ in $dom(g)$, such that $(V, E, N, g')$ is a perfect network built on $(V, E, f)$. We define such a function $g'$ as follows.

First we define a finite sequence of partial functions $g_i$ from $V \times I$ to $S$. We start with $g_0 = g$. For some $j \in I$ and some $s \in S$, we look for a vertex $r_{js} \in \{u \in V : g_m(u, j) = s\}$ from which all vertices in this set are reachable. If such a vertex exists, and it has a parent $v$ such that $(v, j) \notin dom(g_m)$ then we define $g_{m+1}$ as follows:

$$g_{m+1}(u, i) = \begin{cases} g_m(u, i), & \text{if } (u, i) \in dom(g_m) \\ g_m(r_{js}, j), & \text{if } u = v, i = j \end{cases}$$

If it is impossible to find such $r_{js}$ and $v$ for any $j \in I$ and $s \in S$ then $g_m$ is the last member of the sequence.

Since the tree $(V, E)$ is finite, this process terminates and leads to a finite sequence $g_0, \ldots, g_M$ of partial functions.

We define $g' : V \times I \to S$ recursively:

$$g'(v,j) = \begin{cases} g_M(v,j), \text{ if } (v,j) \in dom(g_M) \\ g'(u,j), \text{ if } (v,j) \notin dom(g_M) \text{ and } (u,v) \in E. \end{cases}$$

To justify the soundness of the definition of $g'$, we need to show that, for all $j \in I$, $(u_0, j) \in dom(g_M)$, where $u_0$ is the root of the tree $(V, E)$. To prove this fact, first we define, for vertices $w, w' \in V$, $w < w'$ to mean that there exists a path from $w$ to $w'$ in $(V, E \cup N)$ that contains at least one edge from $E$. Relation $<$ is transitive and irreflexive. It follows that any nonempty set of vertices has an element minimal relative to $<$. Consider an element $w_0$ of $\{w : (w,j) \in dom(g_M)\}$ that is minimal relative to $<$, and let $s = g_M(w_0, j)$. Then, every vertex in $\{w \in V : g_M(w,j) = s\}$ is reachable from $w_0$ in this set. Therefore, we can pick $w_0$ to be $r_{js}$. We claim that $w_0 = u_0$. Indeed, otherwise, $w_0$ has a parent $v$ in $(V, E)$, and, by the choice of $M$, $(v,j) \in dom(g_M)$, which contradicts the minimality of $w_0$.

What remains to show is that $(V, E, N, g')$ is a perfect network built on $(V, E, f)$, i.e., that conditions (i)–(iii) hold for $(V, E, N, g')$:

(i) Since $g'(v,j) = g_M(v,j) = g(v,j) = f(v,j)$ for all $v \in L$ and for all $j \in I$, $g'|_{L \times I} = f$.

(ii) Take any $j \in I$ and any $s \in S$. Since $(V, E, N, g)$ is a perfect network built on $(V, E, f)$, $(V, E \cup N)$ has a subgraph with the set $\{u \in V : g(u,j) = s\}$ of vertices that is a rooted tree $T$. Let $r_{js}$ be the root of this tree. Due to the definition of $g_M$, there exists a vertex $r'_{js} \in V$ such that

$g_M(r'_{js}, j) = s$ and $r_{js}$ is reachable from $r'_{js}$ in $\{u \in V : g_M(u,j) = s\}$. Then, there is a rooted tree $T'$ in $(V, E \cup N)$ with the root $r'_{js}$ and with the set $\{u \in V : g_M(u,j) = s\}$ of vertices such that $T$ is a subgraph of $T'$. Therefore, $(V, E \cup N)$ has a subgraph with the set $\{u \in V : g_M(u,j) = s\}$ of vertices that is a rooted tree. Then, due to definition of $g'$, for every descendant $w$ of $r'_{js}$ in $(V,E)$ with $(w,j) \notin dom(g_M)$, $g'(w,j) = s$. It follows that there is a rooted tree $T''$ in $(V, E \cup N)$ with the root $r'_{js}$ and with the $\{u \in V : g'(u,j) = s\}$ of vertices such that $T'$ is a subgraph of $T''$. Therefore, $(V, E \cup N)$ has a subgraph with the set $\{u \in V : g'(u,j) = s\}$ of vertices that is a rooted tree.

(iii) Since $(V, E, N, g)$ is a partial perfect network built on $(V, E, f)$, for every edge $(u,v) \in E$, $u$ is not reachable from $v$ in $(V, E \cup N)$.

## 9.16 Proof of Proposition 10

**Proposition 10 (page 148)** *Let $(V, E, f)$ be a phylogeny, with $f : L \times I \to S$. For any perfect network $(V, E, N, g')$ built on this phylogeny there exists a partial mapping $g$ of $V \times I$ to $S$ such that*

- *(V,E,N,g) is a partial perfect network built on the same phylogeny,*
- *$g'$ is an extension of $g$ to $V \times I$, and*
- *$g(v,i)$ is essential with respect to $i$ whenever $v \notin L$.*

**Proof.** Let $(V, E, f)$ be a phylogeny, with $f : L \times I \to S$. Let $(V, E, N, g')$ be a perfect network built on $(V, E, f)$. For every character $j$ in $I$, let $S_j$ denote the set of essential states with respect to $j$, and let $V_j$ denote the set $\{v \in V : g'(v, j) \in S_j\}$. Define $g$ to be a partial function from $V \times I$ to $S$ such that, for every $j \in I$,

- $g(v, j) = f(v, j)$ if $v \in L$, and
- $g(v, j) = g'(v, j)$ if $v \in V_j \setminus L$.

Note that $g'$ is an extension of $g$ to $V \times I$, and $g(v, i)$ is essential with respect to $i$ whenever $v \notin L$. We want to show that $(V, E, N, g)$ is a partial perfect network built on $(V, E, f)$, i.e., that conditions (i)–(iii) hold for $(V, E, N, g)$.

(i) Since $g(v, j) = f(v, j)$ for all $v \in L$ and for all $j \in I$, $g|_{L \times I} = f$.

(ii) Take any $j \in I$ and any $s \in S$. Suppose that $V_{js} = \{u \in V : g(u, j) = s\}$ is not empty. If $s \in S_j$ then $V_{js}$ is equal to $V'_{js} = \{u \in V : g'(u, j) = s\}$. Since $(V, E, N, g')$ is a perfect network built on $(V, E, f)$, due to (ii), $(V, E \cup N)$ has a subgraph with the set $V'_{js}$ of vertices that is a rooted tree; therefore, $(V, E \cup N)$ has a subgraph with the set $V_{js}$ of vertices that is a rooted tree. Otherwise, i.e., $s \notin S_j$, $V_{js} = \{u \in V : g(u, j) = s\}$ is a singleton due to the definition of $S_j$. Then $(V, E \cup N)$ has a subgraph with the set $V_{js}$ of vertices that is a rooted tree.

(iii) Since $(V, E, N, g')$ is a perfect network built on $(V, E, f)$, for every edge $(u, v) \in E$, $u$ is not reachable from $v$ in $(V, E \cup N)$.

## 9.17 Proof of Proposition 11

**Proposition 11 (page 151)** *Let* $(V, E, N, g)$ *be a minimal almost perfect network built on* $(V, E, f)$. *For every* $j \in I$, *there exists a minimal almost perfect network* $(V, E, N_j, g\,|_{V\times\{j\}})$ *built on* $(V, E, f\,|_{L\times\{j\}})$ *such that* $\bigcup_{j\in I} N_j = N$.

**Proof.** Let $(V, E, N, g)$ be a minimal almost perfect network built on $(V, E, f)$. Then, by Fact 1, for every $j \in I$, $(V, E, N, g\,|_{V\times\{j\}})$ is an almost perfect network built on $(V, E, f\,|_{L\times\{j\}})$. Then, for every $j \in I$, there exists a minimal set $N_j \subseteq N$ such that $(V, E, N_j, g\,|_{V\times\{j\}})$ is an almost perfect network built on $(V, E, f\,|_{L\times\{j\}})$. Note that $\bigcup_j N_j \subseteq N$. Then, by Fact 2, $(V, E, \bigcup_j N_j, g)$ is an almost perfect network built on $(V, E, f)$. Since $N$ is minimal subject to condition (ii), $N \subseteq \bigcup_j N_j$. Therefore, $N = \bigcup_j N_j$.

# Chapter 10

# Concluding Remarks

This dissertation is about answer set programming (ASP)—a new form of declarative logic programming based on the answer set semantics. Instead of traditional Prolog systems, ASP uses answer set solvers such as CCALC, DLV and SMODELS. The input of Prolog consists of a logic program and a query, and Prolog computes answer substitutions; the input of an answer set solver is a logic program, and the solver computes the program's answer sets. The idea of ASP is to represent a given computational problem as a logic program whose answer sets correspond to solutions, and to use an answer set solver to find an answer set.

In this dissertation, we have investigated the possibility of using answer set programming for solving combinatorial search problems of several kinds, including the following:

**Planning (Chapter 3).** In a planning problem, we want to find a plan—a sequence of actions that leads to a given goal. We start with planning problems described by transition systems as in. We represent a planning problem as a logic program whose answer sets correspond to plans. We use the answer set solvers, such as CCALC, DLV and SMODELS to compute plans. In this way, we have experimented with several planning problems, such as the Suitcase problem and Blocks World problems. We have found out that, with our approach, ASP provides elegant solutions to challenging problems of representing actions and change, such as the frame problem, the ramification problem, the qualification problem, and representing concurrent execution of actions.

**Wire routing (Chapter 7).** This is the problem of determining the physical locations of all wires interconnecting the circuit components on a chip. Since the wires cannot intersect with each other, they are competing for limited spaces, thus making routing a difficult combinatorial optimization problem. We have introduced two new approaches to wire routing using answer set programming: the graph approach and the planning approach. All existing routing systems are based on variations of the sequential maze routing approach using a shortest path algorithm connecting one wire at a time. A major shortcoming of these algorithms is that they cannot guarantee finding a routing solution even when one exists. Our methods differ from the existing ones in that they are complete: they always correctly determine whether a given routing problem is solvable, and they produce a routing solution whenever one exists. We have used SMODELS with the graph approach and CCALC with the planning approach to find solutions to routing problems.

**Phylogeny reconstruction (Chapter 8).** This is the problem of constructing and labeling an evolutionary tree for a set of taxa, which describes the evolution of the taxa in that set from their most recent common ancestor. We have presented two applications of ASP to phylogeny reconstruction: one to solve the MIPPN problem in linguistics and the other to solve the median genome problem in biology. Using SMODELS, we have computed some solutions explaining the evolution of Indo-European languages. We have found out the median genome problem cannot be efficiently solved using SMODELS.

We are interested in fully adequate declarative representations of problems in these areas, their relations to logic programming, and algorithms for solving these problems. Our work on these applications led to the investigation of some theoretical problems related to answer sets:

**Equivalent transformations (Chapter 4).** We have presented some conditions under which we can replace a program by another program that has the same answer sets but can be processed by an answer set solver more efficiently.

- Defining exogenous atoms. We can describe the exogenous atoms to DLV and SMODELS by disjunctive rules similar to (2.6) (see for instance, rules (3.20)). To present this definition to CCALC, we have to transform these disjunctive rules into nondisjunctive rules (3.19) (see for instance, rules (3.28)). Our Theorem 1 justifies this transformation: it shows that this transformation preserves the answer sets for a program.

- No-concurrency constraints. When the goal is to find a plan in which actions are executed sequentially, the logic programming representation of the problem has to contain a no-concurrency constraint similar to (4.1) (see for instance, rules (3.26)). The equivalent transformation provided by Theorem 2 may allow us to state this constraint in a way that provides some computational advantages, as discussed in Section 3.7. For instance, according to the encoding of blocks world in Section 3.6, for a blocks world problem with $n$ blocks, the size of the program after grounding grows as $n^4$. After replacing (3.26) by rules (3.29), the size of a program after grounding grows as $n^3$. Our Theorem 2 justifies this transformation: it shows that replacing (3.26) by rules (3.29) may extend the answer sets by new literals, but the parts of the answer sets that belong to the language of the original program remain the same.

**Completion (Chapters 5 and 6).** Clark defined a "completion" procedure that translates a logic program into a set of formulas of classical logic. In some cases, the interpretations satisfying the completion of a program are also the answer sets for that program (Fages' Theorem). For some problems, propositional solvers, such as CHAFF, SATO or RELSAT, are more efficient than answer set solvers (see for instance, Figure 3.9 for a comparison of CHAFF to SMODELS, Figure 5.1 for a comparison of SATO to SMODELS). Therefore, we have investigated under what conditions this is the case so that we can use propositional solvers instead of answer set solvers, to find the program's answer sets.

- Generalization of Fages' Theorem. Usually, to show that the models of completion of a program are also its answer sets, we use Fages' theorem, as mentioned in Section 3.4.2. There are some programs where we cannot use Fages' theorem to show the equivalence of completion semantics to answer set semantics. Consider, for instance, the blocks world program due to Ilkka Niemelä, presented in Section 5.2. This program is not tight; however, the models of the completion of this program are identical to its answer sets. To make Fages' theorem applicable to such programs, we have defined "tightness on a set of literals", and generalized Fages' theorem accordingly (Theorem 3).

- Transitive closure. A logic program whose answer sets we want to find may define the transitive closure of one of its predicates—for instance, the transitive closure *above* of the relation *on* in the blocks world presented in Section 3.6. Our Theorem 4 shows that, the definition of the transitive closure of a predicate in the context of a set of rules with negation as failure, not merely in combination with a set of facts, is correct relative to answer set semantics. If every model of the completion of such a program is an answer set then the program's answer sets can be found using a propositional solver. When a program contains the definition of the transitive closure of a predicate, it may be difficult to check its tightness directly. Our Theorem 5 can be sometimes used to show that the tightness of a program is not lost when such a definition is added to it. Theorem 3 is used to prove Theorem 5.

- Further generalization of Fages' Theorem. There are some applications of ASP where the results summarized above are not sufficient. Consider, for instance, the program describing the New Year's Party problem and the programs describing the N-Queens problem. These programs contain cardinality constraints. The cardinality constraints can be equivalently transformed to nested expressions. We have extended the definition of tightness on a set of literals to programs with nested expressions to facilitate the verification of the equivalence between the answer set semantics and the completion semantics for programs like this.

According to the results of our experiments, the computational efficiency of ASP depends on several factors. How we present problems to answer set solvers is important: wire routing problems can be solved faster using SMODELS with the graph approach, or using CCALC with the planning approach. Another essential factor is the use of heuristics and optimization techniques: using a divide-and-conquer strategy reduced solving the MIPPN problem for Indo-European languages to smaller problems; representing no-concurrency constraints in Blocks World using the optimization technique stated in Theorem 2 reduced computation time; preprocessing and extracting essential states and characters allowed us to start with a smaller phylogeny to solve the MIPPN problem for Indo-European languages. Using domain specific information may also affect the computational efficiency of ASP: information about Indo-European languages allowed us to eliminate many solutions that do not make sense from the point of view of linguistics. Such information takes the form of additional constraints that helps SMODELS compute solutions faster. Depending on the problem, some systems may perform better than the others: for the N-Queens problems, the SAT-solver SATO outperforms the answer set solver SMODELS; for the blocks world problems, CCALC performs better than DLV and SMODELS.

Our experiments show that sometimes the computational efficiency of ASP is very poor even for small problems (like the median genome problems), sometimes it is good only for small problems (like the small wire routing problems), and sometimes it is good enough to solve serious problems (like the MIPPN problem).

# Bibliography

[Apt and Bol, 1994] Krzysztof Apt and Ronald Bol. Logic programming and negation: a survey. *Journal of Logic Programming*, 19,20:9–71, 1994.

[Apt and Emden, 1982] K. Apt and van M. Emden. Contributions to the theory of logic programming. *Journal of ACM*, 29(3):841–862, 1982.

[Apt *et al.*, 1988] Krzysztof Apt, Howard Blair, and Adrian Walker. Towards a theory of declarative knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, San Mateo, CA, 1988.

[Aura *et al.*, 2000] Tuomas Aura, Matt Bishop, and Dean Sniegowski. Analyzing single-server network inhibition. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 108–117, Cambridge, UK, june 2000. IEEE Computer Society Press.

[Babovich *et al.*, 2000] Yuliya Babovich, Esra Erdem, and Vladimir Lifschitz. Fages' theorem and answer set programming.[1] In *Proc. NMR-2000*, 2000.

[1] `http://arxiv.org/abs/cs.ai/0003042` .

[Balduccini *et al.*, 2000] M. Balduccini, M. Gelfond, and M. Nogueira. A-Prolog as a tool for declarative programming. In *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE'2000)*, 2000.

[Balduccini *et al.*, 2001] M. Balduccini, M. Gelfond, M. Nogueira, R. Watson, and M. Barry. An A-Prolog decision support system for the space shuttle. In *Working Notes of the AAAI Spring Symposium on Answer Set Programming*, 2001.

[Baral and Gelfond, 1994] Chitta Baral and Michael Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 19,20:73–148, 1994.

[Bayardo and Schrag, 1997] Roberto Bayardo and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proc. IJCAI-97*, pages 203–208, 1997.

[Ben-Eliyahu and Dechter, 1994] Rachel Ben-Eliyahu and Rina Dechter. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.

[Bidoit and Froidevaux, 1987] Nicole Bidoit and Christine Froidevaux. Minimalism subsumes default logic and circumscription. In *Proc. LICS-87*, pages 89–97, 1987.

[Brewka and Dix, 2001] Gerhard Brewka and Jürgen Dix. Knowledge representation with extended logic programs. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, 2nd Edition, Volume 6, Methodologies*, chapter 6. Reidel Publ., 2001. Shortened version also appeared in Dix, Pereira, Przymusinski (Eds.), *Logic Programming and Knowledge Representation*, Springer LNAI 1471, pages 1–55, 1998.

[Caprara, 1999] A. Caprara. Formulations and hardness of multiple sorting by reversals. In *Proc. 3rd Conf. Computational Molecular Biology RECOMB99*, pages 84–93. ACM Press, 1999.

[Chen *et al.*, 1993] Weidong Chen, Terrance Swift, and David Warren. Goal-directed evaluation of well-founded semantics for XSB. In Dale Miller, editor, *Proc. ILPS-93*, page 679, 1993.

[Cholewiński *et al.*, 1996] Pawel Cholewiński, Victor Marek, and Miroslaw Truszczyński. Default reasoning system DeReS. In *Principles of Knowledge Representation and Reasoning: Proc. of the Fifth Int'l Conf.*, pages 518–528, 1996.

[Citrigno *et al.*, 1997] Simona Citrigno, Thomas Eiter, Wolfgang Faber, Georg Gottlob, Christoph Koch, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The DLV system: Model generator and application frontends. In *Proceedings of Workshop on Logic Programming (WLP97)*, 1997.

[Clark, 1978] Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[Colmerauer and Roussel, 1996] Alain Colmerauer and Philippe Roussel. The birth of PROLOG. In *History of Programming Languages*. ACM Press/Addison-Wesley, 1996.

[DasGupta and Wang, 1999] Bhaskar DasGupta and Lusheng Wang. Biology computing. In *Wiley Encyclopedia of Electrical Engineering and Electronics*, pages 386–394. John Wiley and Sons, Inc., 1999.

[Davis and Putnam, 1960] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[Denecker and Van Nuffelen, 1999] Marc Denecker and Bert Van Nuffelen. Experiments for integration CLP and abduction.[2] In Krysztof R. Apt, Antonios C. Kakas, Eric Monfroy, and Francesca Rossi, editors, *Proceedings of the 1999 ERCIM/COMPULOG workshop on Constraints*, pages 1–15, 1999.

[Dimopoulos *et al.*, 1997] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding planning problems in non-monotonic logic programs. In Sam Steel and Rachid Alami, editors, *Proc. European Conf. on Planning 1997*, pages 169–181. Springer-Verlag, 1997.

[2] `http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=20013` .

[Dix *et al.*, 2001] Jürgen Dix, Ulrich Furbach, and Ilkka Niemelä. Nonmonotonic reasoning: Towards efficient calculi and implementations. In Andrei Voronkov and Alan Robinson, editors, *Handbook of Automated Reasoning.* Elsevier-Science-Press, 2001.

[Dix, 1995] Jürgen Dix. Semantics of logic programs: Their intuitions and formal properties. An overview. In Andre Fuhrmann and Hans Rott, editors, *Logic, Action and Information – Essays on Logic in Philosophy and Artificial Intelligence*, pages 241–327. DeGruyter, 1995.

[East and Truszczyński, 2000] Deborah East and Mirosław Truszczyński. dcs: An implementation of DATALOG with constraints. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*, 2000. Special session on System descriptions and demonstration.

[East and Truszczyński, 2001] Deborah East and Mirosław Truszczyński. More on wire routing with ASP. In *Working Notes of the AAAI Spring Symposium on Answer Set Programming*, 2001.

[Eiter *et al.*, 1997] Thomas Eiter, Nicola Leone, Christinel Mateis, Gerald Pfeifer, and Francesco Scarcello. A deductive system for non-monotonic reasoning. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 363–374. Springer-Verlag, 1997.

[Eiter *et al.*, 1998] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald

Pfeifer, and Francesco Scarcello. The KR system DLV: Progress report, comparisons and benchmarks. In Anthony Cohn, Lenhart Schubert, and Stuart Shapiro, editors, *Proc. Sixth Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 406–417, 1998.

[Eiter *et al.*, 1999] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. The diagnosis frontend of the DLV system. *The European Journal on Artificial Intelligence*, 12(1–2):99–111, 1999.

[Emden and Kowalski, 1976] Maarten van Emden and Robert Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[Erdem and Lifschitz, 1999] Esra Erdem and Vladimir Lifschitz. Transformations of logic programs related to causality and planning. In *Logic Programming and Non-monotonic Reasoning: Proc. Fifth Int'l Conf. (Lecture Notes in Artificial Intelligence 1730)*, pages 107–116, 1999.

[Erdem and Lifschitz, 2001a] Esra Erdem and Vladimir Lifschitz. Fages' theorem for programs with nested expressions. In *Proceedings of the Seventeenth International Conference on Logic Programming*, pages 242–254, 2001.

[Erdem and Lifschitz, 2001b] Esra Erdem and Vladimir Lifschitz. Transitive closure, answer sets, and predicate completion. In *Working Notes of the AAAI Spring Symposium on Answer Set Programming*, 2001.

[Erdem and Lifschitz, 2002] Esra Erdem and Vladimir Lifschitz. Tight logic programs. To appear in the Special Issue of the Theory and Practice of Logic Programming Journal on Answer Set Programming, 2002.

[Erdem *et al.*, 2000] Esra Erdem, Vladimir Lifschitz, and Martin Wong. Wire routing and satisfiability planning. In *Proc. CL-2000*, pages 822–836, 2000.

[Erdem *et al.*, 2002] Esra Erdem, Vladimir Lifschitz, Luay Nakhleh, and Donald Ringe. Reconstructing the evolutionary tree of natural languages using answer set programming. Submitted for publication, 2002.

[Erdem, 1999] Esra Erdem. Applications of logic programs to planning: computational experiments.[3] Unpublished draft, 1999.

[Faber *et al.*, 1999] Wolfgang Faber, Nicola Leone, Cristinel Mateis, and Gerald Pfeifer. Using database optimization techniques for nonmonotonic reasoning. In *Proceedings of the Seventh International Workshop on Deductive Databases and Logic Programming*, 1999.

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

[Felsenstein, 1982] J. Felsenstein. Numerical methods for inferring evolutionary trees. *The Quarterly Review of Biology*, 57:379–404, 1982.

[3] `http://www.cs.utexas.edu/users/esra/experiments/experiments.html` .

[Felsenstein, 1988] J. Felsenstein. Phylogenies from molecular sequences: Inference and reliability. *Annual Reviews in Genetics*, 22:521–565, 1988.

[Ferraris and Lifschitz, 2001] Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. Submitted for publication.[4], 2001.

[Fikes and Nilsson, 1971] Richard Fikes and Nils Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

[Finger, 1986] Jeffrey Finger. *Exploiting Constraints in Design Synthesis*. PhD thesis, Stanford University, 1986. PhD thesis.

[Geffner, 1990] Hector Geffner. Causal theories for nonmonotonic reasoning. In *Proc. AAAI-90*, pages 524–530. AAAI Press, 1990.

[Gelfond and Galloway, 2001] Michael Gelfond and Joel Galloway. Diagnosing dynamic systems in AProlog. In *Working Notes of the AAAI Spring Symposium on Answer Set Programming*, 2001.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Logic Programming: Proc. Fifth Int'l Conf. and Symp.*, pages 1070–1080, 1988.

[Gelfond and Lifschitz, 1990] Michael Gelfond and Vladimir Lifschitz. Logic

[4] `http://www.cs.utexas.edu/users/vl/mypapers/weight.ps` .

programs with classical negation. In David Warren and Peter Szeredi, editors, *Logic Programming: Proc. Seventh Int'l Conf.*, pages 579–597, 1990.

[Gelfond and Lifschitz, 1991] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[Gelfond and Lifschitz, 1998] Michael Gelfond and Vladimir Lifschitz. Action languages.[5] *Electronic Transactions on AI*, 3:195–210, 1998.

[Gelfond *et al.*, 1991] Michael Gelfond, Vladimir Lifschitz, Halina Przymusińska, and Mirosław Truszczyński. Disjunctive defaults. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Principles of Knowledge Representation and Reasoning: Proc. Second Int'l Conf.*, pages 230–237, 1991.

[Gelfond, 1987] Michael Gelfond. On stratified autoepistemic theories. In *Proc. AAAI-87*, pages 207–211, 1987.

[Giunchiglia and Lifschitz, 1998] Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: Preliminary report. In *Proc. AAAI-98*, pages 623–630. AAAI Press, 1998.

[Gödel, 1932] Kurt Gödel. Zum intuitionistischen Aussagenkalkül. *Anzeiger der Akademie der Wissenschaften in Wien*, pages 65–66, 1932. Reproduced in: Kurt Gödel, *Collected Works*, Vol. 1, OUP, 1986.

[5] `http://www.ep.liu.se/ea/cis/1998/016` .

[Hanan, 1965] M. Hanan. Net wiring for large scale integrated circuits. Technical report, IBM, 1965.

[Hanan, 1966] M. Hanan. On Steiner's problem with rectilinear distance. *SIAM Journal on Applied Mathematics*, 14:255–265, 1966.

[Heljanko and Niemelä, 2001] K. Heljanko and I. Niemelä. Answer set programming and bounded model checking. In *Working Notes of the AAAI Spring Symposium on Answer Set Programming*, 2001.

[Heljanko, 1999] K. Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe petri nets. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, pages 240–254, 1999.

[Hennig, 1966] W. Hennig. *Phylogenetic Systematics.* University of Illinois Press, 1966. Translated from *Grundzuege einer Theorie der phylogenetischen Systematik* (1950) by D. D. Davis and R. Zangerl.

[Hietalahti *et al.*, 2000] M. Hietalahti, F. Massacci, and I. Niemelä. DES: a challenge problem for nonmonotonic reasoning systems. In *Proceedings of the 8th International Workshop on Nonmonotonic Reasoning*, 2000.

[Hill, 1974] R. Hill. LUSH resolution and it completeness. Technical report, Department of Artificial Intelligence, University of Edinburgh, 1974.

[Hwang *et al.*, 1992] F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner Tree Problem*, volume 53. North-Holland, Amsterdam, Netherlands, 1992.

[Inoue and Sakama, 1994] Katsumi Inoue and Chiaki Sakama. On positive occurrences of negation as failure. In *Proc. Fourth Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 293–304, 1994.

[Jaffar *et al.*, 1983] Joxan Jaffar, Jean-Louis Lassez, and John Lloyd. Completeness of the negation as failure rule. In *Proc. IJCAI-83*, volume 1, pages 500–506, 1983.

[Jarník and Kössler, 1934] V. Jarník and M. Kössler. O minimálních grafech obsahujících n daných bodu. *Ĉasopis Pêstování Mat.*, 63:223–235, 1934.

[Kakas *et al.*, 1992] Antonis Kakas, Robert Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1992.

[Kautz and Selman, 1992] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proc. ECAI-92*, pages 359–363, 1992.

[Kautz and Selman, 1996] Henry Kautz and Bart Selman. Pushing the envelope: planning, propositional logic and stochastic search. In *Proc. AAAI-96*, pages 1194–1201, 1996.

[Kautz *et al.*, 1996] Henry Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. In *Principles of Knowledge Representation and Reasoning: Proc. of the Fifth Int'l Conf.*, pages 374–384, 1996.

[Köksal *et al.*, 2001] Pınar Köksal, Nihan Kesim Çiçekli, and İsmail Hakkı Toroslu. Specification of workflow process using the action description language C. In *Working Notes of the AAAI Spring Symposium on Answer Set Programming*, 2001.

[Kowalski and Kuehner, 1971] Robert A. Kowalski and Donald Kuehner. Linear resolution with selection function. *Journal of Artificial Intelligence*, 2:22–260, 1971.

[Kowalski, 1974] Robert A. Kowalski. Predicate logic as programming language. In Jack L. Rosenfeld, editor, *Proceedings of International Federation of Information Processing Conference*, pages 569–574, Stockholm, Sweden, 1974. North–Holland.

[Lee, 1961] C. Y. Lee. An algorithm for path connections and its application. *IRE Transactions on Electronic Computers*, EC–10:346–365, 1961.

[Lengauer, 1990] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Design.* John Wiley & Sons, 1990.

[Li and Graur, 1991] W. Li and D. Graur. *Fundamentals of Molecular Evolution.* Sinauer Associates, Inc., 1991.

[Lifschitz and Turner, 1994] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In Pascal Van Hentenryck, editor, *Proc. Eleventh Int'l Conf. on Logic Programming*, pages 23–37, 1994.

[Lifschitz and Turner, 1999] Vladimir Lifschitz and Hudson Turner. Representing transition systems by logic programs. In *Logic Programming and Non-monotonic Reasoning: Proc. Fifth Int'l Conf. (Lecture Notes in Artificial Intelligence 1730)*, pages 92–106, 1999.

[Lifschitz and Woo, 1992] Vladimir Lifschitz and Thomas Woo. Answer sets in general nonmonotonic reasoning (preliminary report). In Bernhard Nebel, Charles Rich, and William Swartout, editors, *Proc. Third Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 603–614, 1992.

[Lifschitz *et al.*, 1999] Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

[Lifschitz *et al.*, 2001] Vladimir Lifschitz, David Pearce, and Agustin Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2001. To appear.

[Lifschitz, 1988] Vladimir Lifschitz. On the declarative semantics of logic programs with negation. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 177–192. Morgan Kaufmann, San Mateo, CA, 1988.

[Lifschitz, 1996] Vladimir Lifschitz. Two components of an action language.

In *Working Papers of the Third Symposium on Logical Formalizations of Commonsense Reasoning*, 1996.

[Lifschitz, 1999] Vladimir Lifschitz. Answer set planning. In *Proc. ICLP-99*, pages 23–37, 1999.

[Lifschitz, 2000] Vladimir Lifschitz. Missionaries and cannibals in the causal calculator. In *Principles of Knowledge Representation and Reasoning: Proc. Seventh Int'l Conf.*, pages 85–96, 2000.

[Lifschitz, 2002] Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138:39–54, 2002.

[Lin, 1995] Fangzhen Lin. Embracing causality in specifying the indirect effects of actions. In *Proc. IJCAI-95*, pages 1985–1991, 1995.

[Liu *et al.*, 1998] X. Liu, C. Ramakrishnan, and S. Smolka. Fully local and efficient evaluation of alternating fixed points. In *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 5–19. SpringerVerlag, 1998.

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

[Loveland, 1970] D. W. Loveland. A linear format for resolution. In *Proceedings of the IRIA Symposium on Automatic Demonstration*, pages 147–162, New York, 1970. Springer-Verlag.

[Mair, 1998] V.H. Mair, editor. *The Bronze Age and Early Iron Age Peoples of Eastern Central Asia.* Institute for the Study of Man, Washington, 1998.

[Mallory, 1989] J.P. Mallory. *In Search of the Indo-Europeans.* Thames and Hudson, London, 1989.

[Marek and Truszczyński, 1999] Victor Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.

[McCain and Turner, 1997] Norman McCain and Hudson Turner. Causal theories of action and change. In *Proc. AAAI-97*, pages 460–465, 1997.

[McCain and Turner, 1998] Norman McCain and Hudson Turner. Satisfiability planning with causal theories. In Anthony Cohn, Lenhart Schubert, and Stuart Shapiro, editors, *Proc. Sixth Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 212–223, 1998.

[McCain, 1997] Norman McCain. *Causality in Commonsense Reasoning about Actions.*[6] PhD thesis, University of Texas at Austin, 1997.

[McCarthy and Hayes, 1969] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

[6] `ftp://ftp.cs.utexas.edu/pub/techreports/tr97-25.ps.Z` .

[McCarthy, 1959] John McCarthy. Programs with common sense. In *Proc. Teddington Conf. on the Mechanization of Thought Processes*, pages 75–91, London, 1959. Her Majesty's Stationery Office.

[McCarthy, 1980] John McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39,171–172, 1980. Reproduced in [McCarthy, 1990].

[McCarthy, 1986] John McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 26(3):89–116, 1986. Reproduced in [McCarthy, 1990].

[McCarthy, 1990] John McCarthy. *Formalizing Common Sense: Papers by John McCarthy.* Ablex, Norwood, NJ, 1990.

[McCarthy, 1999] John McCarthy. Elaboration tolerance.[7] In progress, 1999.

[Moore, 1985] Robert Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25(1):75–94, 1985.

[Moret *et al.*, 2001] B. Moret, S. Wyman, D.Bader, T. Warnow, and M. Yan. A new implementation and detailed study of breakpoint analysis. In *Proc. Sixth Pacific Symp. Biocomputing (PSB)*, pages 583–594, 2001.

[Moskewicz *et al.*, 2001] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient

[7] `http://www-formal.stanford.edu/jmc/elaboration.html` .

SAT solver. In *Proceedings of Design Automation Conference (DAC2001)*, 2001.

[Nakhleh *et al.*, 2002] L. Nakhleh, D. Ringe, and T. Warnow. Perfect phylogenetic networks: A new methodology for reconstructing the evolutionary history of natural languages. Submitted for publication, 2002.

[Niemelä and Simons, 1996] Ilkka Niemelä and Patrik Simons. Efficient implementation of the well-founded and stable model semantics. In *Proc. Joint Int'l Conf. and Symp. on Logic Programming*, pages 289–303, 1996.

[Niemelä and Simons, 2000] Ilkka Niemelä and Patrik Simons. Extending the Smodel system with cardinality and weight constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*. Kluwer, 2000.

[Niemelä *et al.*, 1999] Ilkka Niemelä, Patrik Simons, and Timo Soininen. Stable model semantics of weight constraint rules. In *Proceedings of the Fifth International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer-Verlag, 1999.

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

[Ohtsuki, 1986] T. Ohtsuki. Maze-running and line-search algorithms. In T. Ohtsuki, editor, *Layout Design and Verification*, chapter 3. Elsevier Science Publishers, 1986.

[Pearce and Wagner, 1990] David Pearce and Gerd Wagner. Reasoning with negative information I: Strong negation in logic programs. *Acta Philosophica Fennica*, 49, 1990.

[Pelov *et al.*, 2000] Nikolay Pelov, Emmanuel De Mot, and Marc Denecker. Logic programming approaches for representing and solving constraint satisfaction problems : a comparison.[8] In Parigot M. and Voronkov A., editors, *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*, volume 1955 of *Lecture Notes in Artificial Intelligence*, pages 225–239. Springer, 2000.

[Przymusinski, 1988] Teodor Przymusinski. On the declarative semantics of deductive databases and logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, San Mateo, CA, 1988.

[Reiter, 1980] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[Ringe *et al.*, 2002] D. Ringe, T. Warnow, and A. Taylor. Indo-European and computational cladistics. *Transactions of the Philological Society*, 100(1):59–129, 2002.

[Roberts *et al.*, 1990] R.G. Roberts, R. Jones, and M.A. Smith. Thermolu-

[8] `http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=32258` .

minescence dating of a 50,000-year-old human occupation site in Northern Australia. *Science*, 345:153–156, 1990.

[Robinson, 1965] Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[Roussel, 1975] Philippe Roussel. PROLOG, manuel de reference at d'utilisation. Technical report, Université d'Aix-Marseille II, 1975.

[Sakama, 2001] Chiaki Sakama. Learning by answer sets. In *Working Notes of the AAAI Spring Symposium on Answer Set Programming*, 2001.

[Sankoff and Blanchette, 1998] D. Sankoff and M. Blanchette. Multiple genome rearrangement and breakpoint phylogeny. *Journal of Computational Biology*, 5:555–570, 1998.

[Schubert, 1990] Lenhart Schubert. Monotonic solution of the frame problem in the situation calculus: an efficient method for worlds with fully specified actions. In H.E. Kyburg, R. Loui, and G. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer, 1990.

[Semiconductor Industry Association, 1997] Semiconductor Industry Association. The national roadmap for semiconductors, 1997.

[Shanahan, 1993] Murray Shanahan. Explanation in the situation calculus. In *Proc. IJCAI-93*, pages 160–165, 1993.

[Shepherdson, 1988] John Shepherdson. Negation in logic programming. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 19–88. Morgan Kaufmann, San Mateo, CA, 1988.

[Siepel, 2002] Adam Siepel. An algorithm to find all sorting reversals. In *Proc. Sixth Int'l Conf. on Comput. Mol. Biol. (RECOMB)*, 2002.

[Simons *et al.*, 2002] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, 2002.

[Simons, 1997] Patrik Simons. Towards constraint satisfaction through logic programs and the stable model semantics. Technical Report 47, Helsinki University of Technology, 1997.

[Simons, 1999] Patrik Simons. Extending the stable model semantics with more expressive rules. In *Logic Programming and Non-monotonic Reasoning: Proc. Fifth Int'l Conf. (Lecture Notes in Artificial Intelligence 1730)*, pages 305–316, 1999.

[Simons, 2000] Patrik Simons. Extending and implementing the stable model semantics. Technical Report 58, Helsinki University of Technology, 2000.

[Soininen and Niemelä, 1998] Timo Soininen and Ilkka Niemelä. Developing a declarative rule language for applications in product configuration. In Gopal Gupta, editor, *Proc. First Int'l Workshop on Practical Aspects of*

*Declarative Languages (Lecture Notes in Computer Science 1551)*, pages 305–319. Springer-Verlag, 1998.

[Soininen *et al.*, 1999] T. Soininen, E. Gelle, and I. Niemelä. A fixpoint definition of dynamic constraint satisfaction. In *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, pages 419–433, 1999.

[Son and Lobo, 2001] Tran Cao Son and Jorge Lobo. Reasoning about policies using logic programs. In *Working Notes of the AAAI Spring Symposium on Answer Set Programming*, 2001.

[Subrahmanian and Zaniolo, 1995] V.S. Subrahmanian and Carlo Zaniolo. Relating stable models and AI planning domains. In *Proc. ICLP-95*, 1995.

[Sussman, 1990] J. Gerald Sussman. The virtuous nature of bugs. *Readings in Planning*, pages 11–117, 1990.

[Szymanski, 1985] T. G. Szymanski. Dogleg channel routing is NP-complete. *IEEE Transactions on Computer-Aided Design*, 4(1):31–41, 1985.

[Trajcevski *et al.*, 2000] Goce Trajcevski, Chitta Baral, and Jorge Lobo. Formalizing (and reasoning about) the specifications of workflows. In *Proceedings of the Fifth IFCIS International conference on Cooperative Information Systems (CoopIS'2000)*, 2000.

[Van Gelder *et al.*, 1988] Allen Van Gelder, Kenneth A. Ross, and John S.

Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 221–230. ACM Press, 1988.

[Van Gelder *et al.*, 1991] Allen Van Gelder, Kenneth Ross, and John Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[Van Gelder, 1988] Allen Van Gelder. Negation as failure using tight derivations for general logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 149–176. Morgan Kaufmann, San Mateo, CA, 1988.

[van Hentenryck, 1989] Pascal van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[Wang and Warnow, 2001] Li-San Wang and Tandy Warnow. Estimating true evolutionary distances between genomes. In *Proceedings of the Thirty-Third Annual ACM Symposium on the Theory of Computing (STOC)*, pages 637–646, 2001.

[White and O'Connell, 1982] J.P. White and J.F. O'Connell. *A Prehistory of Australia, New Guinea, and Sahul.* Academic Press, New York, 1982.

[Zhang, 1997] Hantao Zhang. SATO: An efficient propositional prover. In *Proc. CADE-97*, 1997.

# Vita

Esra Erdem was born in Afyon, Turkey on December 5, 1974, the daughter of Huriye Erdem and Orhan Erdem. In 1992 she entered Bilkent University, Ankara, Turkey, where she received the degree of Bachelor of Science in Computer Engineering and Information Sciences (June 1996). In 1996 she entered the University of Texas at Austin for graduate studies. She received the degree of Master of Science in Computer Sciences in December 1998.

Permanent Address: Vişnelik Mahallesi
Savaş Sokak, 21/9
Eskişehir, TURKEY

This dissertation was typeset with LaTeX 2ε[9] by the author.

[9] LaTeX 2ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.