ANSWER SET PROGRAMMING WITH CLAUSE LEARNING

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Jeffrey Alan Ward, B.A., B.S., M.S., M.S.

* * * * *

The Ohio State University

2004

ABSTRACT

In this dissertation we show how *conflict clause learning*, a technique that has been very useful in improving the efficiency of Boolean logic satisfiability search, can be adapted to speed up dramatically the search for models of answer set programs.

Answer set programming is a knowledge representation paradigm related to the areas of logic programming and nonmonotonic reasoning. Many of the applications of answer set programming come from the areas of artificial intelligence–related diagnosis and planning. The problem of finding an answer set for a normal or extended logic program is NP-hard. Current complete answer set solvers are patterned after the *Davis-Putnam-Loveland-Logemann* ($DPLL$) algorithm for solving Boolean satisfiability (SAT) problems, but are adapted to the nonmonotonic semantics of answer set programming.

Recent SAT solvers include improvements to the DPLL algorithm. Conflict clause learning has been particularly effective in this regard. A conflict clause represents a backtracking solver's analysis of why a conflict occurred. This analysis can be used to further prune the search space and to direct the search heuristic. The use of such clauses has improved significantly the efficiency of satisfiability solvers over the past few years, especially on structured problems arising from applications.

In this dissertation we describe how we have adapted conflict clause techniques for use in the answer set solver *Smodels*. We experimentally compare the performance of the resulting program, *Smodels*$_{cc}$, to that of the original Smodels program. Our tests show dramatic speedups for Smodels$_{cc}$ on a wide range of problems.

We also compare the performance of Smodels$_{cc}$ with that of two other recent answer set solvers, *ASSAT* and *Cmodels-2*. ASSAT and Cmodels-2 directly call Boolean satisfiability solvers in order to search for answer sets. On so-called "non-tight" problems, Smodels$_{cc}$ showed substantially better performance than these solvers. The performance advantage that Smodels$_{cc}$ enjoys on non-tight problems is due to the *unfounded set* test that Smodels$_{cc}$ inherits from the Smodels solver, and the fact that this test is executed frequently throughout the program's search for answer sets.

This is dedicated to the memory of my father, Robert, and to my mother, Carleen.

ACKNOWLEDGMENTS

My most sincere thanks to both of my co-advisors, John Schlipf and Tim Long. John basically taught me what I know about answer set programming. He was also very generous with his time, engaging me in many brainstorming sessions, patiently critiquing my ideas, and providing many detailed suggestions that improved my writing.

Tim has been an enormous encouragement and aid to me along this journey. He engaged me in many useful discussions of this work, and showed a very keen sense of what would constitute a good research direction. He also went well above the call of duty in navigating bureaucracy on my behalf.

Thanks to my other dissertation committee members, Neelam Soundarajan and Eric Fosler-Lussier, whose suggestions greatly improved the exposition of this dissertation.

Thanks to John Franco for introducing me to the literature on conflict clause learning in the realm of Boolean satisfiability search.

Thanks to Ilkka Niemelä, Patrick Simons, and Tommi Syrjänen for the Smodels and Lparse programs. Moreover, thanks for making Smodels open source, so that I could base the code for my program, Smodels$_{cc}$, on it.

Thanks also to two of the developers of Cmodels-2: Yuliya Lierler and Marco Maratea. Yuliya answered many of my questions about Cmodels-2. Marco provided useful bug reports on early versions of Smodels$_{cc}$, and suggested trying the bounded model checking benchmark problems.

VITA

February 11, 1961	Born - Ft. Thomas, Kentucky
1984	B.A. Philosophy, B.S. Mathematics, Northern Kentucky University
1987	M.S. Mathmatics, The Ohio State University
1988	M.S. Computer and Information Science, The Ohio State University
1985-1993	Graduate Teaching Associate, Mathematics and CIS Departments, The Ohio State University
1994-1996	Assistant Professor, Department of Mathematics and Computer Science, College of Mount Saint Joseph
1996-1999	Computer Game Programmer, PyroTechnix, Inc., Cincinnati, Ohio
1999-2002	Graduate Research Assistant, Department of Electrical and Computer Engineering and Computer Science, University of Cincinnati

PUBLICATIONS

Research Publications

Wolfgang W. Kuechlin and Jeffrey A. Ward. “Experiments with Virtual C-Threads”. *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing,* Ft. Worth, Texas, December 1991.

John Franco, Michal Kouril, John Schlipf, Jeffrey Ward, Sean Weaver, Michael Dransfield, and Mark Vanfleet. “SBSAT: A State-based, BDD-based Satisfiability Algorithm”. *Theory and Applications of Satisfiability Testing: 6th International Conference (SAT 2003)*, Santa Margherita Ligure, Italy, May 2003.

Jeffrey Ward and John S. Schlipf. “Answer Set Programming with Clause Learning”, *Proceedings of Logic Programming and Nonmonotonic Reasoning 7 (LPNMR-7)*, Ft. Lauderdale, Florida, January 2004.

FIELDS OF STUDY

Major Field: Computer Science

Studies in:

Logic and Logic Programming	Prof. John S. Schlipf
	Prof. Wolfgang W. Kuechlin
	Prof. Timothy J. Carlson
Theory of Computation	Prof. Timothy J. Long
Computer Algebra	Prof. George C. Collins

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Answer Set Programming

Answer set programming (ASP) is a knowledge representation paradigm related to the areas of logic programming and nonmonotonic reasoning. In this approach, a problem is represented as a logic program whose models under the *answer set semantics* constitute solutions (*answer sets*). The logic program representation is typically presented as input to an answer set search engine, which searches for one or more valid models. The answer set semantics was defined by Gelfond and Lifschitz [23] as a generalization of their definition in [22] of the *stable model semantics.* The answer set/stable model semantics is currently the leading declarative semantics for logic programs.

Many of the current practical applications of answer set programming are in the areas of artificial intelligence–related diagnosis and planning [34, 21, 2, 52, 50, 11]. Due to the expressive power of logic programs under the answer set semantics, applications have also been considered in other areas, such as graph algorithms [50] and bounded model checking [28].

Answer set programming may be compared to the widely used knowledge representation approaches that represent problems as classical, Boolean logic formulas. The problem of finding a model for such a formula is essentially the well-known satisfiability testing problem (SAT), which involves reasoning in a monotonic logic. There are parallels between the way that reasoning is currently done in answer set programming and in SAT. Firstly, logic programs that are to be tested for answer sets are normally "grounded" by a preprocessing stage so that they, like SAT problems, are represented in a propositional form before being presented to a search engine. Secondly, a solution to an answer set program, like a solution to a SAT problem, consists of a model that sets some of the atomic propositions in the problem to true, and the remaining atomic propositions to false. Thirdly, complete answer set search engines, like complete SAT search engines, are generally based on a depth-first, backtracking search patterned after the *Davis-Putnam-Loveland-Logemann* (*DPLL*) algorithm [10, 9].

However, some contrasts can be made between answer set programming and SAT that motivate considering answer set programming as a useful alternative. Some problems seem to be representable asymptotically more concisely as logic programs under the answer set semantics than they can be represented as SAT formulas. One important advantage of this is greater convenience for the user in representing the problem. Previous work has suggested that this can also result in a faster search for solutions on some problems. (See, for instance, experiments by Simons [54] on solving Hamiltonian cycle problems with SAT and ASP solvers.) Although the SAT decision problem can easily be reduced in linear time to the problem of the existence of answer sets, there is no known linear time reduction of answer set programming problems to SAT. Furthermore, all ASP to SAT reductions either (1) introduce a significant number of new atoms, which could substantially increase the size of the search space, or (2) create a representation that, in the worst case, is exponentially larger than the original answer set programming representation.

An important key to the ability of answer set languages to express certain problems succinctly is the principle of *negation as failure*, which is common to logic programming formalisms. This principle states that, if an atomic proposition cannot be "proven" from the rules given in the program/database, then the proposition will be assumed to be *false*. The embodiment of this principle in the answer set semantics adds considerable expressive power and convenience to the paradigm, but also results in answer set solvers having a considerably more complex set of inference rules than the set of inference rules used in SAT solvers.

The problem of determining whether a *normal* logic program has an answer set is NP-complete [43]. Thus, it is not surprising that the task of computing answer sets is often computationally expensive in practice. In order to expand the usefulness of the answer set programming paradigm, it will be very useful to improve the efficiency of current answer set search algorithms. This is a topic of considerable current interest in the answer set programming community [17, 41, 54, 24] and is the subject of this dissertation.

1.2 Clause Learning

Perhaps the most important development over the past several years towards making SAT solvers more efficient in solving practical problems has been the use of *conflict clauses* [3, 44]. A conflict clause represents a backtracking solver's analysis of why a conflict occurred during the search. This analysis can be used to further prune the search space and to direct the search heuristic. Conflict clauses have been especially useful in improving the efficiency of SAT solvers on structured problems arising from applications. The majority of current, well-known, complete SAT solvers, such as GRASP [44], rel_sat [3], SATO [63], Chaff [47], BerkMin [27], and SIMO [26], concentrate their optimizations and heuristics around efficiently and effectively generating and using conflict clauses.

1.3 Contributions

For this dissertation we incorporate conflict clause learning into one of the most widely used answer set solvers, Smodels [51]. The resulting program, Smodels$_{cc}$[62], is able to use conflict clauses to prune the search space and to direct the search heuristic. The fundamental new problem that we had to address in order to accomplish this concerned finding an algorithm to diagnose the causes of conflicts in Smodels' search. The main challenge in this regard involved analyzing conflicts that resulted from negative inferences derived from Smodels' detection of *unfounded sets*.[1] The detection of unfounded sets is central to Smodels' enforcement of the answer set semantics' version of negation as failure. Negative inferences derived from unfounded sets result in a number of complications when performing a conflict diagnosis. We outline these complications in Section 4.3.2. However, our experimental results show that frequently testing for unfounded sets is critical to obtaining good performance on certain important problems.

We conduct experimental tests comparing the performance of Smodels$_{cc}$ to that of the original Smodels program. We also compare the performance of Smodels$_{cc}$ to

[1]The notion of an unfounded set was defined by van Gelder, Ross, and Schlipf in [20]. We discuss it in Section 2.3.4.

that of two other recent high-performance answer set solvers, ASSAT and Cmodels-2. These two solvers instead take the approach of directly calling SAT solvers to perform the majority of the work involved in an answer set search.

The results of these experiments support two conclusions:

1. *Adding conflict clause learning can greatly improve the runtime performance of Smodels on a wide range of problems.* This was evident on problems arising from applications (hardware verification and bounded model checking), and on randomly generated problems that had a non-uniform data distribution (e.g., graph coloring where the distribution of edges was "clumpy"). Conflict clauses also sped up, by orders of magnitude, the performance of Smodels on Hamiltonian cycle problems, regardless of whether the distribution of edges was uniform or non-uniform.

2. *On so-called "non-tight" logic programs, the approach of ASSAT and Cmodels-2, which separates the test for unfounded sets from the main search process, is less efficient than the approach used in Smodels$_{cc}$.* On "tight" problems[2], ASSAT and Cmodels-2 provided somewhat better performance than Smodels$_{cc}$. This was expected since they call recent, highly optimized SAT solvers to perform the search, and tight problems are easily reduced to SAT.

   However, in our tests involving non-tight problems, such as the Hamiltonian cycle problems mentioned above, Smodels$_{cc}$ performed much better than ASSAT and Cmodels-2. On these problems, Smodels$_{cc}$ is able to provide stronger pruning of the search space than ASSAT and Cmodels-2 because of the test for

[2]See Definition 25 in Section A.2 for the distinction between tight and non-tight problems.

unfounded sets that Smodels$_{cc}$ inherits from the original Smodels solver, and the fact that this test is executed frequently throughout the program's search.

CHAPTER 2

BACKGROUND: ANSWER SET PROGRAMMING

This chapter provides background information on the area of Answer Set Programming. Nothing in this chapter is new to this dissertation. Citations are provided for major definitions and results.

2.1 Declarative Logic Programming

A logic program is a set of rules of the form

$$\varphi \leftarrow \psi_1, \ldots, \psi_n$$

where $\varphi, \psi_1, \ldots, \psi_n$ are formulas in some logic. In the above general form, φ is referred to as the *head* of the rule and $\psi_1, \ldots, \psi_n$ is the rule's *body.* Intuitively, such a rule says that if all of the formulas in the body of the rule are true, then the formula at the head of the rule must also be true. If $n = 0$ then we normally omit writing the $\leftarrow$ symbol. In such a case the rule simply states unconditionally that φ is true.

The basic task of a declarative logic programming system is to answer questions about which facts are implied by the logic program at hand, or to find a set of facts that would constitute a model for the logic program. There are different ways of interpreting these questions formally, which gives rise to different logic programming *semantics.*

2.1.1 Ground Instantiations

The formulas that appear in a logic program may include predicates that take one or more arguments, and some of these arguments may be variables. The various declarative logic programming semantics generally treat a rule with variables as representing the set of ground instantiations of that rule. Thus the meaning of a logic program with variables is reduced to the meaning of a logic program without variables.

Example 1 *Assume that a, b, and c are treated as constants and that X and Y are treated as variables.*[3] *Consider the logic program P:*

$$\begin{array}{l} arc(a,b) \\ arc(b,c) \\ nonterminal(X) \leftarrow arc(X,Y) \end{array}$$

The ground instantiation of this program would be:

$$\begin{array}{l} arc(a,b) \\ arc(b,c) \\ nonterminal(a) \leftarrow arc(a,a) \\ nonterminal(a) \leftarrow arc(a,b) \\ nonterminal(a) \leftarrow arc(a,c) \\ nonterminal(b) \leftarrow arc(b,a) \\ nonterminal(b) \leftarrow arc(b,b) \\ nonterminal(b) \leftarrow arc(b,c) \\ nonterminal(c) \leftarrow arc(c,a) \\ nonterminal(c) \leftarrow arc(c,b) \\ nonterminal(c) \leftarrow arc(c,c) \end{array}$$

A practical problem with creating a ground instantiation is that its size may be exponentially large in terms of the size of the original logic program. Most intelligent

[3]It is, in fact, a common convention in logic programming that arguments that begin with lowercase letters are constants, and arguments that begin with uppercase letters are variables. We will adopt this convention throughout this dissertation.

grounding programs try to produce relatively small ground instantiations, while still preserving the meaning of the original program. For instance, under the various semantics that we will consider in this dissertation, it would be permissible to restrict the ground instantiation above to only the rules:

$arc(a, b)$
$arc(b, c)$
$nonterminal(a) \leftarrow arc(a, b)$
$nonterminal(b) \leftarrow arc(b, c)$

This is because it is clear that the only instantiations of the *arc* predicate that can be established based on the given program are $arc(a, b)$ and $arc(b, c)$.

The methods used by the various grounding programs to cut down the size of the ground instantiation may vary. However, the grounding process is not a central concern of this dissertation. For our experiments in Chapter 6 we used the Lparse program[58] to create ground instantiations of our logic programs.

The upshot is that, in this dissertation, we will restrict our attention to finite, propositional logic programs. The kinds of decision problems that one would usually ask about a logic program, such as whether it has a model or whether it implies a particular statement, are generally decidable when the program is finite and propositional.[4]

Since the arguments to the predicates mentioned in a ground instantiated program will be constants, we can view each atomic proposition in the program as essentially a

[4]In order to ensure that the ground instantiation is finite, answer set grounders make some restrictions on how function symbols and variable symbols may be used in a program. These restrictions may vary from system to system, but are not regarded as part of the core definition of the answer set semantics. The interested reader may find details on Lparse's restrictions in [58] and [59]. In *non-answer set* logic programming languages with less tight syntactic restrictions, such as Prolog, ground instantiations may be infinite, and typical decision problems, e.g., the Halting problem, are often undecidable.

0-ary predicate. Thus, the logic program above, after grounding, might be expressed to a solver as follows:

$$\begin{array}{lcl} p & & \\ q & & \\ r & \leftarrow & p \\ s & \leftarrow & q \end{array}$$

2.2 Answer Set Semantics for Normal Logic Programs

In the following discussion, we will assume that *Atoms* is some fixed set of atomic propositions (0-ary predicate symbols).

Definition 1 (Horn program) *A* Horn rule *is a logic programming rule of the form*

$$h \leftarrow a_1, \ldots, a_n$$

where $h, a_1, \ldots, a_n \in Atoms$. *A* Horn program *is a collection of Horn rules.*

We will denote the set of atoms mentioned in P by $Atoms(P)$. Also, we will assume that the logic programs considered in this dissertation have a finite number of rules.

Definition 2 (Model of a Horn program) *A* model *of a Horn program* P *is a subset* $M \subseteq Atoms$ *such that for every rule* $h \leftarrow a_1, \ldots, a_n \in P$, $a_1, \ldots, a_n \in M$ *implies* $h \in M$.

Definition 3 (Minimum model) *A* minimum model *of a program* P *is a model of* P *that is a subset of all other models of* P.

If a program has a minimum model, then it has only one minimum model. An important fact is that every Horn program does have a minimum model. Furthermore, the minimum model of a Horn program can be computed by the following procedure:

```
MinimumModel(HornProgram P)
{
    $M \leftarrow \emptyset$
    while $\exists$ rule $R = h \leftarrow a_1, \ldots, a_n \in P$
              such that $a_1, \ldots, a_n \in M$
              and $h \notin M$
    do
          $M \leftarrow M \cup \{h\}$
    return $M$
}
```

Note, then, that the minimum model of a Horn program P is therefore the *deductive closure* of P, if we view the rules of P as inference rules. The minimum model is generally taken to be the canonical model of a Horn program. Using suitable data structures, the runtime of the above procedure is linear in the size of P. (This was shown by Dowling and Gallier in [12], and the resulting version of the procedure is known as the *Dowling-Gallier algorithm*). Thus, reasoning with Horn programs is easy from a computational complexity viewpoint.

An important concern in logic programming research is how to deal with the presence of logical negation operators in programs. The preeminent negation operator in logic programming is the *not* operator, also known as the *negation as failure* operator.

Definition 4 (Normal program) *A* normal rule *is a rule of the form:*

$$h \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m$$

where $h, a_1, \ldots, a_n, b_1, \ldots, b_m \in Atoms$. A normal logic program *is a collection of normal rules.*

Atom h is the *head* of the above rule, while $a_1, \ldots, a_n, \textit{not}\ b_1, \ldots, \textit{not}\ b_m$ form the rule's *body*. We refer to $a_1, \ldots, a_n$ as the body's *positive subgoals*, and $\textit{not}\ b_1, \ldots, \textit{not}\ b_m$ as its *negative subgoals*. The logic programming semantics that we consider will view the body of a rule as a *set* of subgoals: the order in which the subgoals appear in the body does not matter.

The next two definitions give the answer set semantics for normal logic programs, and therefore formally define the meaning of the *not* operator in the answer set programming paradigm.

Definition 5 (Reduct) *Given a normal logic program P and a set $S \subseteq \textit{Atoms}(P)$, the* reduct *of P by S, denoted P^S, is the logic program obtained from P by deleting*

1. *each rule containing a negative literal not b_i in its body where $b_i \in S$, and*

2. *all negative subgoals from the bodies of the remaining rules.*

Note that the reduct P^S is a Horn program, and therefore has a (unique) minimum model. The mapping that takes P and S and produces P^S is called the *Gelfond–Lifschitz transform*. This mapping is the key to the following definition.

Definition 6 (Answer set) *(Gelfond and Lifschitz [22]) If P is a normal logic program, then a set $S \subseteq \textit{Atoms}(P)$ is an* answer set *for P if the minimum model of P^S is equal to S.*

A normal logic program may have zero, one, or more than one answer sets. The problem of determining whether a grounded normal logic program has any answer sets is NP-complete [43].

Example 2 *The logic program*

a ← b, not c
b ← not c
c ← not b

has two answer sets: $S_1 = \{a, b\}$ *and* $S_2 = \{c\}$.

Example 3 *The logic program*

b ← not c
c ← b

has no answer sets.

Note that if a normal logic program does not contain the *not* operator, then it is a Horn program and it has exactly one answer set, namely the minimum model.

As suggested earlier, the *not* operator in logic programs is used to express a form of negation based on the principle of *negation as failure*. In other words, an expression of the form *not* a is satisfied if there is no evidence for the truth of a, i.e. it is not possible to prove a. An answer set S for a normal logic program P constitutes a two-valued logical interpretation of the atoms mentioned in P. Those atoms that are considered to be *true* are those that are elements of S. Atoms not in S are considered to be *false*. The atoms considered *false* in an answer set S are exactly those atoms for which there is no proof (specifically, no proof from P^S). The atoms that are considered *true* in S are exactly those atoms that have proofs (from P^S). The fact that P^S is a Horn program (does not mention *not*) means that the notion of whether an atom has a proof from P^S is well-defined and straightforward.

Classical negation

Answer sets for normal logic programs are also referred to as *stable models*, which was the term used in the paper where these notions were first defined ([22]). The term "answer set" was used in [23], where the semantics was extended to programs incorporating the $\neg$ operator.

The $\neg$ operator is often referred to as the *classical negation* operator, whereas the *not* operator is referred to as the *negation as failure* operator. Informally, $\neg a$ means that a is *false*, whereas *not* a means that a does not have a proof. Syntactically, the main difference is that the $\neg$ operator is allowed in the head of a rule, whereas the *not* operator is not allowed in the head. Thus, a rule such as

$$\neg a \leftarrow b,\ c$$

which says that "if b and c are true, then a is false" is allowed in an *extended* logic program [23]. However, a rule such as

$$not\ a \leftarrow b,\ c$$

which says that "if b and c are true, then a does not have a proof" is not allowed.

The interested reader may refer to [23] for the semantics and uses of the $\neg$ operator in answer set programming. Typically, one would include classical negation if one wanted to work with three- or four-valued logical models rather than two-valued ("*true/false*") models. For instance, if $\neg$ is used, then a and $\neg a$ are both treated as literals that may belong to an answer set M. If $a \in M$, then a is regarded as *true* in the model. If $\neg a \in M$, then a is regarded as *false*. If neither a nor $\neg a$ belong to M, then the truth value of a is considered *unknown*. If both a and $\neg a$ belong to M, then a represents a contradiction.

However, in [23], it was shown how programs involving both the *not* and $\neg$ operators could easily be reduced to normal programs (i.e., to programs that use *not* as the only negation operator). In fact, the Lparse program automatically performs this reduction on all programs involving $\neg$. Hence, we will restrict our consideration of negation operators to the *not* operator only.

Integrity constraints

Sometimes we would like to express in our logic program that a set of conditions is impossible. We can do this by writing an *integrity constraint*, which is simply a rule with an empty head.

Example 4 *The rule* $R =$

$$\leftarrow a, not\ b$$

states that no answer set of the program can have both $a =$ true *and* $b =$ false.

Formally, the rule R above is dealt with by treating it as shorthand for the rule $R' =$

$$f \leftarrow a, not\ b, not\ f$$

where f is a new atom introduced into the program specifically for use, as above, in integrity constraints. The reader can check that the resulting program, which includes R', cannot have an answer set with both $a = true$ and $b = false$.

2.3 Relationship to Classical Boolean Satisfiability (SAT)

One of the useful features (alluded to in Section 2.1.1) of answer set systems is that they allow the user to write predicates that can take variables as arguments. As we will see, this greatly helps to make it convenient to express problems in an answer set language. Answer set systems also generally provide a richer syntax than merely normal logic programs. For example, the Lparse and Smodels systems provide the "extended rules" that we will consider in Section 2.4. These are some of the strengths of answer set programming over classical Boolean logic as a knowledge representation paradigm.

However, for the remainder of this section we will restrict our attention to a comparison of grounded normal logic programs interpreted under the answer set semantics versus classical Boolean logic expressions (written, say, in conjunctive normal form) interpreted under the usual semantics in that domain. It turns out that even with these restrictions, the answer set programming paradigm has a significant advantage over classical Boolean logic when it comes to expressing problems concisely and conveniently. This advantage comes from the way in which the answer set semantics embodies the principle of negation as failure. In practice, one of the payoffs of negation as failure in the answer set semantics is that it allows us to write inductive definitions of predicates. We will see an instance of this in Section 2.3.3, where we express the Hamiltonian cycle problem as an answer set problem. Expressing inductively defined predicates tends to be much more difficult in classical Boolean logic.

Before we look at that example, we will consider the problem of reducing classical Boolean logic problems to answer set programming problems (Section 2.3.1). We label the respective decision problems as SAT and ASP. We recall that both SAT and ASP are NP-complete. Later, in Section 2.3.5, we will consider the problem of reducing ASP to SAT. We will see that it seems to be much easier to devise a concise, practical reduction of SAT to ASP than to do the reverse. We will see how this relates to the negation as failure principle. This motivates considering the answer set paradigm as an alternative to classical Boolean logic on certain problems. It is also relevant to understanding the differences between our approach with Smodels$_{cc}$ (discussed in Chapter 4), and the approach of answer set solvers that call SAT solvers directly (Chapter 5).

2.3.1 Reduction of SAT to ASP

Reducing SAT to ASP is rather simple, as we will see below.

Definition 7 (Literal) *A* literal *is an expression of the form a or $\neg a$, where a is an atom (proposition letter).*

In the above definition, a is a *positive* literal, and $\neg a$ is an *negative* literal.

Definition 8 (Clause) *A* clause *is a disjunction of literals.*

Definition 9 (Conjunctive Normal Form) *A Boolean statement is in* conjunctive normal form *(CNF) if it is expressed as the conjunction of a set of clauses.*

Example 5 *The Boolean expression*

$$(a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg b \vee c)$$

is in conjunctive normal form.

Suppose S is a formula written in conjunctive normal form. We construct a corresponding normal logic program P as follows:

1. For each atom a appearing in S, include atoms a and a' in P, along with the rules

$$\begin{array}{rcl} a & \leftarrow & not\ a' \\ a' & \leftarrow & not\ a \end{array}$$

2. For each clause $C = a_1 \vee \ldots \vee a_m \vee \neg b_1 \vee \ldots \vee \neg b_n$ in S, include in P the integrity constraint

$$\leftarrow not\ a_1, \ldots, not\ a_m, b_1, \ldots, b_n$$

Then M has a satisfying assignment if and only if P has an answer set.

Two favorable properties are evident in the above reduction. First, the size of the resulting logic program is linear in the size of the original CNF formula, with a rather small expansion factor. Secondly, although the reduction doubles the number of atoms that were in the original problem, the new atoms do not actually add to the size of the search space that needs to be considered. That is, for any of the commonly used answer set solvers, if a truth value is assigned to atom a at any time in the search, the solver can immediately infer the opposite truth value for a' from the rules given above. Likewise, if the solver assigns a truth value for a', it can immediately infer the opposite truth value for a. As a result, the new atoms that are introduced by this reduction do not increase the number of choices that would need to be made in the search for a solution to a problem instance.

2.3.2 Completion Semantics

Before considering reductions of ASP to SAT, we will look at a semantics for normal logic programs, called the *completion semantics*, that is *defined* in terms of classical Boolean logic. Understanding the relationship of the answer set semantics to the completion semantics is helpful in understanding the reductions of ASP to SAT, and is particularly relevant to understanding the approach of the solvers discussed in Chapter 5.

Definition 10 (Program Completion) *Let P be a normal logic program. For atom $h \in Atoms(P)$ let φ_h be the propositional formula*

$$h \leftrightarrow \bigvee\{a_1 \wedge \ldots \wedge a_n \wedge \neg b_1 \wedge \ldots \wedge b_m \mid h \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m \in P\}$$

Then the completion *of P, denoted $Comp(P)$, is the formula $\bigwedge\{\varphi_h | h \in Atoms(P)\}$.*

An interpretation M is said to be a model of P under the *completion semantics* if it satisfies $Comp(P)$. This semantics is due to Clark [7]. It is easy to check that every model of P under the answer set semantics is a model of P under the completion semantics. However, the converse is false:

Example 6

$$\begin{array}{ccc} a & \leftarrow & b \\ b & \leftarrow & a \end{array}$$

The completion of the preceding program is the Boolean logic statement

$$(a \leftrightarrow b) \wedge (b \leftrightarrow a).$$

Thus the program has two models under the completion semantics: $M_1 = \{\}$ and $M_2 = \{a, b\}$. However, M_1 is the only answer set of P.

Both the completion semantics and the answer semantics embody the principle of negation as failure, which states that an atom will be considered false if and only if it cannot be proven true. However, the answer set semantics interpretes this principle more strictly. That is, the answer set semantics places a tighter restriction on what kind of proof is permitted to justify the assertion of an atomic proposition.

To see this, suppose that we have a normal logic program P and we wish to check whether a particular interpretation I is a model of P under the answer set and completion semantics, respectively. Let $I^+ = \{a \in \mathit{Atoms}(P) : a \text{ is } \mathit{true} \text{ in I } \}$. Let $I^- = \{\neg a : a \in \mathit{Atoms}(P),\ a \text{ is } \mathit{false} \text{ in I } \}$. Then from Definition 6 we can see that I is an answer set of P if and only if I^+ is exactly the deductive closure of $I^- \cup P$.

On the other hand, I is a model of the completion of P if and only if I^+ is exactly the deductive closure of $I^+ \cup I^- \cup P$. Thus, under the completion semantics, the elements of I^+ can be used as assumptions to prove the elements of I^+. This means that atoms that are asserted *true* in completion semantics models may have circular justifications.

Such circular proofs are not allowed as justifications under the answer set semantics. Example 6 gave an instance of this distinction. We give another illustrating example in Section 2.3.3.

2.3.3 Expressing the Hamiltonian Cycle Problem

Suppose that we wish to express in a normal logic program the problem of finding a Hamiltonian cycle in a directed graph. As is often the case in logic programming, we will express the problem in two parts. The first part of our program, called the *intentional database* (IDB), consists of a set of rules (typically involving variables) that express the logic of the problem. The second part of the program, the *extensional database* (EDB), will consist of the set of facts (expressed as rules with empty bodies) that determine a particular instance of the problem. In the case of the Hamiltonian cycle problem, the IDB will consist of a set of rules that specify that a Hamiltonian cycle is a set of edges which form a path that starts from an initial node, visits every node exactly once, and returns to the initial node.

The following is an IDB similar to one given by Niemelä [49] for the Hamiltonian cycle problem:

```
% Select edges for the cycle
$hc(X,Y) \leftarrow not\ hc'(X,Y), edge(X,Y)$
$hc'(X,Y) \leftarrow not\ hc(X,Y), edge(X,Y)$

% Each vertex has at most one incoming edge in a cycle
$\leftarrow hc(X_1,Y), hc(X_2,Y), edge(X_1,Y), edge(X_2,Y), vertex(Y), X_1 \neq X_2$

% Each vertex has at most one outgoing edge in a cycle
$\leftarrow hc(X,Y_1), hc(X,Y_2), edge(X,Y_1), edge(X,Y_2), vertex(X), Y_1 \neq Y_2$

% Every vertex must be reachable from the initial vertex
% through the chosen hc edges.
$\leftarrow vertex(X), not\ r(X)$
$r(Y) \leftarrow hc(X,Y), edge(X,Y), initialvertex(X)$
$r(Y) \leftarrow hc(X,Y), edge(X,Y), r(X)$
```

Of special significance in this reduction is the reachability predicate, r. $r(Y)$ means that Y is reachable from the initial vertex. Note that r is defined inductively in this program. The rules state how to prove that the r predicate is *true*. It is not necessary to state when the r predicate is false: By the principle of negation as failure, it is assumed that if $r(Y)$ cannot be proven true based on the above rules, then $r(Y)$ is false. The problem of expressing the reachability relation is what seems to make the Hamiltonian cycle problem difficult to express concisely in classical Boolean logic. (See Simons [54] for some experiments contrasting solving the Hamiltonian cycle problem with an answer set solver versus solving it with SAT solvers. He was able to obtain much smaller reductions, and much better runtimes with the answer set approach.)

Now, suppose that the particular instance at hand is the directed graph shown in Figure 2.1. Then a corresponding EDB would be:

$initialvertex(1)$
$vertex(1)$
$vertex(2)$
$vertex(3)$
$vertex(4)$
$vertex(5)$
$vertex(6)$
$vertex(7)$
$vertex(8)$
$edge(1, 2)$
$edge(2, 4)$
$edge(3, 1)$
$edge(4, 3)$
$edge(4, 6)$
$edge(5, 3)$
$edge(5, 6)$
$edge(6, 8)$
$edge(7, 5)$
$edge(8, 7)$

Our program P will be the grounding of the union of the above IDB and EDB. From the perspective of feasibly solving HC problems, a very nice feature of the above reduction is that the size of the grounded program P will be linear in the size of the graph being represented.

There is only one answer set to the above program, and it is given in Table 2.1. This answer set corresponds to choosing the set of edges highlighted in Figure 2.2, which is the only set of edges from this graph that yields a Hamiltonian cycle.

Table 2.2 gives an interpretation which is a model of the program under the completion semantics. The corresponding set of edges is highlighted in Figure 2.3.

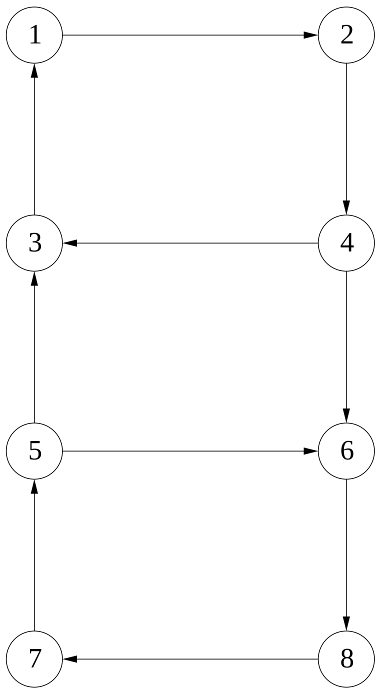

Figure 2.1: A Hamiltonian cycle problem instance

$initialvertex(1)$			
$vertex(1)$	$edge(1,2)$	$hc(1,2)$	$r(1)$
$vertex(2)$	$edge(2,4)$	$hc(2,4)$	$r(2)$
$vertex(3)$	$edge(4,6)$	$hc(4,6)$	$r(3)$
$vertex(4)$	$edge(6,8)$	$hc(6,8)$	$r(4)$
$vertex(5)$	$edge(8,7)$	$hc(8,7)$	$r(5)$
$vertex(6)$	$edge(7,5)$	$hc(7,5)$	$r(6)$
$vertex(7)$	$edge(5,3)$	$hc(5,3)$	$r(7)$
$vertex(8)$	$edge(3,1)$	$hc(3,1)$	$r(8)$
	$edge(4,3)$	$hc'(4,3)$	
	$edge(5,6)$	$hc'(5,6)$	

Table 2.1: Answer set for Hamiltonian cycle problem instance

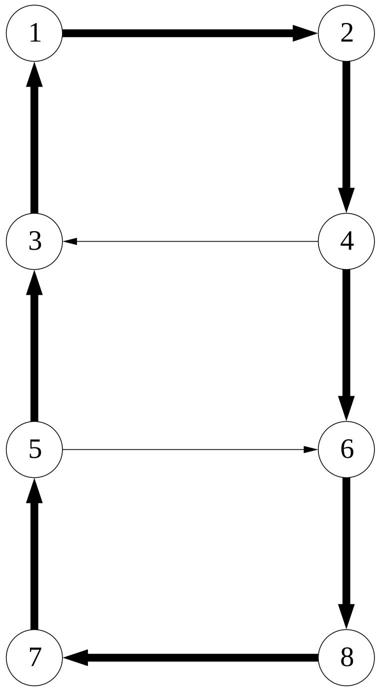

Figure 2.2: The solution to the HC problem instance

$initialvertex(1)$			
$vertex(1)$	$edge(1,2)$	$hc(1,2)$	$r(1)$
$vertex(2)$	$edge(2,4)$	$hc(2,4)$	$r(2)$
$vertex(3)$	$edge(4,6)$	$\mathbf{hc'(4,6)}$	$r(3)$
$vertex(4)$	$edge(6,8)$	$hc(6,8)$	$r(4)$
$vertex(5)$	$edge(8,7)$	$hc(8,7)$	$r(5)$
$vertex(6)$	$edge(7,5)$	$hc(7,5)$	$r(6)$
$vertex(7)$	$edge(5,3)$	$\mathbf{hc'(5,3)}$	$r(7)$
$vertex(8)$	$edge(3,1)$	$hc(3,1)$	$r(8)$
	$edge(4,3)$	$\mathbf{hc(4,3)}$	
	$edge(5,6)$	$\mathbf{hc(5,6)}$	

Table 2.2: Completion semantics model for Hamiltonian cycle problem instance, with differences from Table 2.1 highlighted.

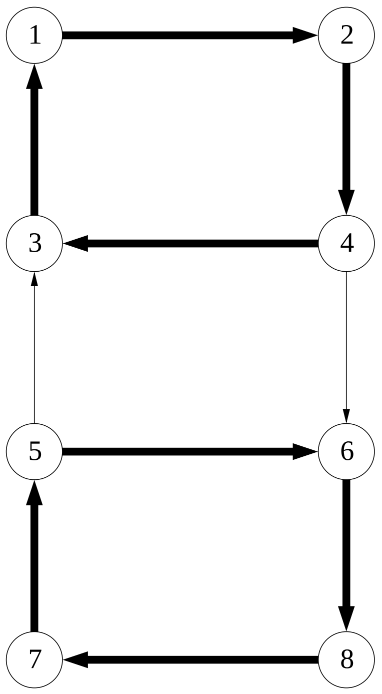

Figure 2.3: A "solution" under the completion semantics

Label the answer set given in Table 2.1 M, and the completion semantics model given in Table 2.2 N. We will again take special note of the reachability predicate r. $r(X)$ had to be *true* for every vertex X in each model, because of the integrity constraint $\leftarrow vertex(X), not\ r(X)$. It is not hard to check that each $r(X)$ is in the deductive closure of $M^- \cup P$, for $X = 1, \ldots, 8$.

However, as we turn our attention to N, we observe that $r(5), \ldots, r(8)$ are not provable from $N^- \cup P$. The only way to justify $r(5), \ldots, r(8)$ from N is by using elements of N^+ as assumptions. Specifically, $r(5), \ldots, r(8)$ can be proven only by a circular sequence of deductions.

We will return to this example in the next section.

2.3.4 Unfounded Sets

Definition 11 (Partial Interpretation) *Let P be a logic program. A* (partial) interpretation *(on P) is a set of literals mentioning only atoms from $Atoms(P)$.*

A (partial) interpretation I on P is considered *total* if every element of $Atoms(P)$ occurs in some element of I.

For example, if $Atoms(P) = \{a, b, c\}$, then $I_1 = \{a, \neg c\}$ is a partial interpretation on P and $I_2 = \{a, b, \neg c\}$ is a total interpretation. Informally, we consider a partial interpretation to be a set of assertions about which atoms are *true*, and which atoms are *false* in a model of P. For instance, based on the above, we may write $I_1(a) = \mathit{true}$, $I_1(c) = \mathit{false}$, and $I_1(b) = \mathit{unknown}$.[5] Also, if, for some atom d, both d and $\neg d$ are members of an interpretation I, then I is *inconsistent* and we may write both $I(d) = \mathit{true}$ and $I(d) = \mathit{false}$.[6] (Partial) interpretations are sometimes referred to as (partial) *truth assignments*.

We denote the positive (resp., negative) elements of I by I^+ (resp., I^-). In the example above, $I_2^+ = \{a, b\}$ and $I_2^- = \{\neg c\}$. We always have $I = I^+ \cup I^-$. And I is consistent if $Atoms(I^+) \cap Atoms(I^-) = \emptyset$.

We say that an interpretation J *extends* an interpretation I if $I \subseteq J$.

Suppose P is a logic program and $S \subseteq Atoms(P)$. Then the total interpretation on P corresponding to S is $M^+ \cup M^-$ where $M^+ = S$ and $M^- = \{\neg a : a \in Atoms(P) \backslash S\}$.

Likewise, a total interpretation M corresponds to the set of atoms M^+. If we state that an interpretation M is an answer set of a normal program P then we mean

[5] In the literature, $I_1(b) = \mathit{unknown}$ is often written $I_1(b) = \bot$.

[6] This situation is sometimes expressed by the equation $I(d) = \top$.

that (1) M is a total interpretation on P, and (2) M^+ is an answer set of P under Definition 6.

The following definition is due to Van Gelder, Ross, and Schlipf [20]:

Definition 12 (Unfounded Set) *Let P be a normal logic program and I a partial interpretation on P. Then $H \subseteq Atoms(P)$ is said to be* unfounded *with respect to P and I if for every rule R of P with head $h \in H$ we have at least one of the following conditions:*

1. *there is a negative subgoal 'not d' in the body of R such that $I(d) = \text{true}$,*
2. *there is a positive subgoal 'c' in the body of R such that $I(c) = \text{false}$, or*
3. *there is a positive subgoal 'c' in the body of R such that $c \in H$.*

Example 7 *Let P be the logic program from Section 2.3.3 that expresses the Hamiltonian cycle problem instance given in Figure 2.1. Let $I = \{\neg hc(4,6)\}$. Then $H = \{r(5), r(6), r(7), r(8)\}$ is unfounded with respect to P and I.*

Observe that if H is unfounded with respect to P and I, and interpretation J extends I, then H is unfounded with respect to P and J.

The following proposition is an immediate corollary to Theorem 6.1 in [20]. However, we prove the result directly here.

Proposition 1 (Van Gelder, Ross, and Schlipf [20]) *Let P be a normal logic program and I a partial interpretation on P. Let $H \subseteq Atoms(P)$ be unfounded with respect to P and I. Let M be a consistent, total interpretation on P that extends I, such that M^+ is an answer set of P. Then $M^+ \cap H = \emptyset$ (i.e., M interprets every element of H as* false*).*

Proof: Since M^+ is an answer set of P, we have $M^+ = \mathit{DeductiveClosure}(P^{M^+})$. We will show that $H \cap \mathit{DeductiveClosure}(P^{M^+}) = \emptyset$ by inducting on the length of a derivation $\mathcal{D} = [b_1, \ldots, b_n]$ from P^{M^+}. So we take as our induction hypothesis that $b_i \notin H$ for $1 \leq i < n$. For the sake of showing a contradiction, suppose that $b_n \in H$. Let

$$R^{M^+} = \quad b_n \leftarrow c_1, \ldots, c_m$$

be the rule from P^{M^+} used to derive b_n in $\mathcal{D}$. Then in P there is a corresponding rule

$$R = \quad b_n \leftarrow c_1, \ldots, c_m, \mathit{not}\ d_1, \ldots, \mathit{not}\ d_q$$

where each $d_i \notin M^+$. Since H is unfounded with respect to P and I, we have three possible cases for R:

1. $d_i \in I \subseteq M$, for some $1 \leq i \leq q$. This is not possible since it contradicts our assumption about R.

2. $\neg c_i \in I \subseteq M$, for some $1 \leq i \leq m$. Since R^M was used to deduce b_n in $\mathcal{D}$, $c_i = b_j$ for some $1 \leq j < n$. Thus, c_i is in the deductive closure of P^{M^+}, and is therefore in M^+. This contradicts M being consistent. Thus this case is impossible.

3. $c_i \in H$, for some $1 \leq i \leq m$. Again, since R^M was used to deduce b_n in $\mathcal{D}$, $c_i = b_j$ for some $1 \leq j < n$. But our induction hypothesis was that no such b_j was an element of H. Hence, this case is also impossible.

We conclude that $b_n \notin H$, which completes the inductive proof. $\square$

Example 8 *Let P, I, and H be as in Example 7. Let N be the set of atoms given in Table 2.2, which constituted a model of P under the completion semantics. If we* *consider the total interpretation M on P such that $M^+ = N$, we see that M extends I, but $M^+ \cap H \neq \emptyset$. Hence, by Proposition 1, $N = M^+$ is not an answer set of P.*

In summary, if I is a partial interpretation of P, the set of atoms U is unfounded with respect to I, and we hope to extend I to an answer set of P, then we will need to ensure that every element of U is interpreted as *false* in the extension. Detecting unfounded sets early in the search will be one of the key issues as we consider different approaches to searching for answer sets.

2.3.5 Reducing ASP to SAT

We saw in Section 2.3.1 that it is straightforward to reduce the Boolean satisfiability problem to the problem of finding an answer set for a normal logic program. We cited some nice practical properties of the reduction given there. Specifically, the reduction did not increase the size of the representation much, and the new representation did not include any new atoms that would require extra choice points in a search for a model. The task of reducing answer set programming problems to SAT problems is less straightforward, and has been studied in various papers. We divide such reductions into two classes: (1) those reductions that operate in polynomial time, but which significantly increase the number of atoms in the problem representation, and (2) those that do not increase the number of atoms, but which exponentially increase the representation size. Because the known reductions fall into either of these two types, it presently does not seem to be feasible, in general, to solve an ASP problem by first reducing the problem to SAT and then calling a SAT solver.

We present examples of ASP-to-SAT reductions in Appendix A.

2.4 Extended Rules

Certain ideas are difficult to express concisely in normal logic programs. An example would be a constraint that says that from a set of n atoms, $\{a_1, \ldots, a_n\}$, at least k atoms must be *true*. Or, suppose that the edges in a graph have various weights, and we wish to express that a Hamiltonian cycle which is chosen through the graph is to have at most a certain total weight.

In order to make expressing such constraints feasible, Simons, Niemelä, and Soininen [55] introduced *weight constraint rules*, which are implemented in Lparse and Smodels. We summarize their definitions below.

Definition 13 (Weight Constraint) *A* weight constraint *is an expression of the form*

$$l \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, not\ b_1 = w_{b_1}, \ldots, not\ b_m = w_{b_m}\} \leq u$$

where each a_i and b_j is an atom; and l, u, and each w_x is a real number.[7]

In the above definition, l and u are called the *lower* and *upper* bounds of the constraint, respectively. Each w term in the above expression is a *weight*. Intuitively, the constraint says that W, the sum of the weights of the satisfied a_i and *not* b_j expressions, satisfies the inequality $l \leq W \leq u$. Either of the bounds may be omitted from the constraint. A missing lower bound in the weight constraint is taken to indicate a lower bound of $-\infty$. Similarly, the default upper bound is $+\infty$.

Definition 14 (Weight Constraint Rule) *A* weight constraint rule *is an expression of the form*

$$C_0 \leftarrow C_1, \ldots, C_n$$

[7]In Smodels, l, u, and the w_x terms are all restricted to integers.

A *weight constraint rule program* is a program consisting of weight constraint rules. Appendix B, Section B.1 gives the formal definition of an answer set for a weight constraint rule program.

Useful shorthand notation

The notation given below (all of which comes from [55]) provides some useful shorthand for some commonly utilized weight constraint rule constructs. Of particular relevance to the implementation of the Smodels program, the grounding program Lparse translates any weight constraint rule program into a program consisting only of the types of rules that we describe below in this subsection. Thus, handling these rule types is sufficient for Smodels to deal with arbitrary weight constraint rule programs. A *cardinality constraint* is an expression of the form

$$l\ \{a_1, \ldots, a_n, \mathit{not}\ b_1, \ldots, b_m\}\ u$$

which is shorthand for the following weight constraint with all weights equal to one:

$$l \leq \{a_1 = 1, \ldots, a_n = 1, \mathit{not}\ b_1 = 1, \ldots, \mathit{not}\ b_m = 1\} \leq u.$$

A *weight rule* is a rule of the form

$$h \leftarrow l \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \mathit{not}\ b_1 = w_{b_1}, \ldots, \mathit{not}\ b_m = w_{b_m}\}$$

where l and all of the weights are non-negative. This is a shorthand for the rule

$$1 \leq \{h = 1\} \leftarrow l \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \mathit{not}\ b_1 = w_{b_1}, \ldots, \mathit{not}\ b_m = w_{b_m}\}.$$

A *choice rule* is a rule of the form

$$\{h_1, \ldots, h_k\} \leftarrow a_1, \ldots, a_n, \mathit{not}\ b_1, \ldots, \mathit{not}\ b_m$$

and is shorthand for

$$0 \leq \{h_1 = 1, \ldots, h_k = 1\} \leftarrow n{+}m \leq \{a_1 = 1, \ldots, a_n = 1, \mathit{not}\ b_1 = 1, \ldots, \mathit{not}\ b_m = 1\}.$$

Informally, a rule of this form states that if $a_1, \ldots, a_n$ are *true* and $b_1, \ldots, b_m$ are *false*, then any of $h_1, \ldots, h_k$ *may* be *true*.

A *cardinality rule*[8] is a rule of the form

$$h \leftarrow k\ \{a_1, \ldots, a_n, \mathit{not}\ b_1, \ldots, b_m\}$$

and corresponds to

$$h \leftarrow k \leq \{a_1 = 1, \ldots, a_n = 1, \mathit{not}\ b_1 = 1, \ldots, \mathit{not}\ b_m = 1\}.$$

Thus a cardinality rule is shorthand for a weight rule with weights all equal to one.

A *normal rule* is a rule of the form

$$h \leftarrow a_1, \ldots, a_n, \mathit{not}\ b_1, \ldots, \mathit{not}\ b_m$$

and corresponds to

$$h \leftarrow n + m \leq \{a_1 = 1, \ldots, a_n = 1, \mathit{not}\ b_1 = 1, \ldots, \mathit{not}\ b_m = 1\}.$$

This formal definition of a normal rule in terms of weight rules results in the same semantics for normal rules as that given in Definition 6.

An *integrity constraint* is a rule of the form

$$\leftarrow a_1, \ldots, a_n, \mathit{not}\ b_1, \ldots, \mathit{not}\ b_m$$

and corresponds to

$$f \leftarrow n + m + 1 \leq \{a_1 = 1, \ldots, a_n = 1, \mathit{not}\ b_1 = 1, \ldots, \mathit{not}\ b_m = 1, \mathit{not}\ f = 1\}$$

[8]In the Lparse user's manual, and in the Smodels source code, cardinality rules are referred to as "constraint rules".

where f is a new atom used only in integrity constraints.

In their paper, Simons, Niemelä, and Soininen show how to translate arbitrary weight constraint rule programs rules into programs using only weight rules and choice rules. (We outline their translation in Appendix B, Section B.2.) Lparse uses a similar approach whereby it translates arbitrary weight constraint rule programs into programs consisting only of weight rules, choice rules, cardinality rules, and normal rules. Separate data structures and classes are used by Smodels to deal with each of these four rule types.

Conditional literals

Conditional literals are relevant only to logic programs with variables. They provide a convenient way to express to the grounding program (such as Lparse) how the variables mentioned in a constraint may be grounded.

A conditional literal is an expression of the form $p : d$ where p is a predicate or a predicate preceded by the *not* operator. d is the *conditional* part of the literal, and also must be a predicate. The : operator in the literal may be read as "such that". The grounding program creates from the $p : d$ expression groundings of p such that d is true.

For example, recall the extensional database for the Hamiltonian cycle problem instance in Section 2.3.3:

$$
\begin{array}{l}
initialvertex(1)\\
vertex(1)\\
vertex(2)\\
vertex(3)\\
vertex(4)\\
vertex(5)\\
vertex(6)\\
vertex(7)
\end{array}
$$

$vertex(8)$
$edge(1,2)$
$edge(2,4)$
$edge(3,1)$
$edge(4,3)$
$edge(4,6)$
$edge(5,3)$
$edge(5,6)$
$edge(6,8)$
$edge(8,7)$
$edge(8,7)$

Given that the EDB uses the predicates "*vertex*" and "*edge*" as shown above, then the statement

$$1\,\{hc(X,Y)\,:\,edge(X,Y)\}\,1 \leftarrow vertex(Y)$$

expresses that every vertex Y must have exactly one incoming edge satisfying the "*hc*" predicate.

The conditional predicate d is used to determine which groundings of p are allowed. Therefore, for any ground instance of d' of d, it should be well-defined whether d' is *true* or *false*. That is, it should not be the case that d' is *true* in some models of the program and *false* in others. Also, it should be computationally *easy* to determine whether d' is *true* or *false*, since this needs to be determined by the grounding program before the ground instantiation is given to the answer set solver.

In [55], Simons, Niemelä, and Soininen enforce this restriction by stating that d must a *domain predicate*. Domain predicates are such that, if we restrict a logic program to only those rules which define domain predicates, then the resulting program will have a unique, easily computed answer set.[9]

[9]The definition of what qualifies as a domain predicate for Lparse has changed over time. The interested reader may refer to the Lparse manual [58] and to a paper by Syrjänen [59]. In this

Minimize statements

A *minimize statement* is a statement of the form

$$minimize\{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, not\ b_1 = w_{b_1}, \ldots, not\ b_m = w_{b_m}\}.$$

Such a statement asks the solver to find an answer set that minimizes the total weight of the satisfied subgoals listed inside the braces. If more than one minimize statement is given in a logic program, then the collection of minimize statements induce a lexicographic ordering on the answer sets of the program. Under this ordering, minimize statements that occur earlier in the program are taken to be more significant.

The maximize statement

$$maximize\{a = w_a, not\ b = w_b\}$$

is shorthand for $minimize\{a = -w_a, not\ b = -w_b\}$.

2.5 Disjunctive Logic Programming

A *disjunctive* rule differs from a normal rule in that it may contain a disjunction of atoms in its head. Thus a disjunctive rule is a rule of the form:

$$h_1 \vee \ldots \vee h_k \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m$$

where $h_1, \ldots, h_k, a_1, \ldots, a_n, b_1, \ldots, b_m \in Atoms$. A *disjunctive logic program* is a collection of disjunctive rules. The answer set semantics for disjunctive programs was defined by Gelfond and Lifschitz in [23]. The relevant definitions parallel the

dissertation, we will use as conditional predicates only predicates that are defined by the program's extensional database (EDB). Since all rules in the EDB have empty bodies, such predicates certainly qualify as domain predicates. Note also that the grounding phase is not the part of the solution finding process where our contribution lies.

definitions by which the same authors defined the answer set semantics for normal programs. We provide their definitions below.

A disjunctive program is *positive* if none of the rules involve the *not* operator. A set of atoms M is a *model* of a positive rule

$$h_1 \vee \ldots \vee h_k \leftarrow a_1, \ldots, a_n$$

if M is a model of the classical logic formula

$$(a_1 \wedge \ldots \wedge a_n) \rightarrow (h_1 \vee \ldots \vee h_k).$$

If P is a disjunctive program that consists of positive rules only, then M is an *answer set* of P if M is a minimal model of P. "Minimal model" means that M is a model of every rule of P, and no strict subset of M is a model of every rule of P.

The (Gelfond-Lifschitz) reduct, P^M, of a disjunctive program is constructed from P and M just as it is in the case of a normal program:

1. Remove every rule that has a subgoal of the form *not* b where $b \in M$, and

2. Remove every negative subgoal from the resulting program.

Note that, for an arbitrary disjunctive program P, P^M is positive. Hence, whether a set is an answer set of P^M is defined above.

Definition 15 (Answer Set of a Disjunctive Program) *Let P be an arbitrary disjunctive logic program. Then a set M is an* answer set *of P if it is an answer set of P^M.*

The question of whether a disjunctive program has an answer set is Σ_2^P-complete [15].

Disjunctive logic programming systems generally compute answer sets through a two phase guess-and-check process that works as follows:

1. (Guess Phase) Search for a set of atoms M that is a model of its own reduct P^M.

2. (Check Phase) Determine whether there is a set $M' \subsetneq M$ such that M' is also a model of P^M. M is an answer set of P iff no such M' exists.

The leading solver for disjunctive logic programming for the past several years has been DLV [13]. A recent system, GnT (“Guess ’n’ Test”) [30], performs the guess and check phases above by creating normal program instances from the original disjunctive problem. GnT makes successive calls to the Smodels solver to solve these normal instances until an answer set has been determined for the original problem.

CHAPTER 3

BACKGROUND: BASIC SEARCH ALGORITHMS FOR SAT AND ASP

This chapter provides background information on the Davis-Putnam-Loveland-Logemann (DPLL) algorithm, which searches for satisfying assignments to Boolean formulas. It also describes in some detail the answer set search program Smodels, whose high-level structure is based on DPLL. Here again, nothing in this chapter is new to this dissertation. However, it sets the stage for Chapter 4, where we describe how others have used conflict clause learning to improve the efficiency of DPLL, and also where we describe how we have adapted conflict clause learning to Smodels to create the program Smodels$_{cc}$.

3.1 DPLL SAT algorithm

Current algorithms that search for satisfying assignments to classical Boolean logic formulas can be placed into either of two broad categories: complete methods or incomplete methods. Complete methods are those methods that, in principle, are guaranteed to return a satisfying assignment if the problem has one, or a message that states that the problem is *unsatisfiable* if no satisfying assignment exists. The incomplete methods search for a solution to the given problem instance but are not guaranteed to return a result if the instance is unsatisfiable. So an incomplete method may in principle run forever if no satisfying assignment exists.

Incomplete methods for SAT usually employ some kind of local, hill-climbing search. GSAT [53] was a pioneering solver in this category.

In this dissertation we will only deal with complete methods for solving SAT and ASP problems. These complete methods are usually based on the Davis-Putnam-Loveland-Logemann algorithm [10],[9]. The algorithm assumes that the Boolean formula to be solved is expressed in conjunctive normal form, or CNF (Definition 9).

The DPLL algorithm may be described as a recursive procedure, as in Table 3.1. The procedure has three subroutines: BCP, Reduce, and Heuristic. The BCP routine performs *Boolean Constraint Propagation*, also known as *Unit Propagation.* It checks whether any of the clauses in the formula are *unit clauses.* A unit clause is one that consists of only a single literal. If such a clause exists, then the BCP routine adds the literal to the current partial assignment and simplifies the formula. This inference rule is known as the *unit clause rule* or the *unit literal rule.* BCP repeatedly applies the unit clause rule until either a conflict with the partial assignment is detected, or until no unit clauses remain in the formula.

The *Reduce* subroutine is used by both the DPLL procedure and the BCP procedure to simplify a CNF formula by a single new assignment.

It is assumed in the pseudocode of Table 3.1 that the negation of a negative literal is a positive literal. For instance, if $literal = \neg a$, then $\neg literal = a$.

The remaining subroutine used by DPLL is *Heuristic.* The call Heuristic(φ') selects an atom a appearing in φ' and returns either the positive or the negative literal mentioning a. This corresponds to the algorithm making a guess as to whether

```
DPLL($\varphi$, $I$)
// $\varphi$ is a Boolean CNF formula; $I$ a partial interpretation.
// Precondition: $Atoms(\varphi) \cap Atoms(I) = \emptyset$.
// This routine searches for an assignment that extends $I$ and satisfies $\varphi$.
// If such an assignment is found, it outputs the assignment and returns $true$.
// Otherwise, it returns $false$.
{
    ($\varphi'$, $I'$, $conflict$) $\leftarrow$ BCP($\varphi$, $I$)
    if ($conflict = true$)
        return $false$
    if ($\varphi'$ contains no clauses)
        output $I'$   // $I'$ is a satisfying assignment
        return $true$
    $literal \leftarrow Heuristic(\varphi')$
    if (DPLL(Reduce($\varphi'$, $literal$), $I' \cup \{literal\}$) = $true$)
        return $true$
    else
        return DPLL(Reduce($\varphi'$, $\neg literal$), $I' \cup \{\neg literal\}$)
}

BCP($\varphi$, $I$)
// Boolean Constraint Propagation
// Applies the unit clause inference rule to $\varphi$
// until a conflict is found or until no unit clauses remain.
// Returns the resulting $\varphi$ and $I$, as well as a flag
// indicating whether a conflict was found.
{
    while ($\exists$ a unit clause $C = \{literal\}$ in $\varphi$) do
        if $\neg literal \in I$
            return ($\varphi$, $I$, $true$)
        $I \leftarrow I \cup \{literal\}$
        $\varphi \leftarrow$ Reduce($\varphi, literal$)
    return ($\varphi$, $I$, $false$)
}

Reduce($\varphi$, $literal$)
// Returns $\varphi$ simplified by the assumption that $literal = true$.
{
    Remove from $\varphi$ all clauses containing $literal$.
    Remove $\neg literal$ from any remaining clauses in which it occurs.
    Return the resulting $\varphi$.
}
```

Table 3.1: Davis-Putnam-Loveland-Logemann algorithm for SAT

a will be *true* or *false* in the satisfying assignment. We refer to a as a *choice atom*, and the assignment that is made to a as a *choice assignment*. DPLL then makes a recursive call to itself with a instantiated according to the choice assignment. If that recursive call fails, then DPLL *backtracks* and makes another recursive call, this time with the value of a reversed. The efficiency of the DPLL algorithm depends to a significant extent on the choices made by the heuristic routine, and considerable research has gone into developing effective heuristic strategies for DPLL.

The execution of the DPLL algorithm may be portrayed as a binary tree where each node in the tree represents a recursive call to DPLL, and each leaf node represents either a conflict or a satisfying assignment.

3.2 Smodels

The Smodels algorithm follows the general outline of the DPLL procedure. However, rather than representing its problem instance as a set of Boolean clauses, Smodels uses the representation that it receives from the grounding program Lparse. Smodels accepts a grounded[10], extended logic program. (We will assume for the moment that the program includes no minimize statements, but will address minimize statements in Section 3.2.5). Recall that Lparse translates extended logic programs into a set of normal rules, choice rules, cardinality rules, and weight rules. The counterpart to the BCP procedure in the DPLL routine is the *Expand* routine in Smodels. Pseudocode for the Smodels algorithm is given in Table 3.2.

[10]i.e. propositional

```
Smodels($P$, $I$)
// $P$ is an extended logic program.
// $I$ is a set of literals representing a partial interpretation.
// This routine searches for a total interpretation that extends $I$ to an answer set for $P$.
// If such an interpretation is found, Smodels outputs it and returns $true$.
// Otherwise, it returns $false$.
{
    $J \leftarrow$ Expand($P$, $I$)
    if (Conflict($J$))
        return $false$
    if (J covers $Atoms(P)$)
        output $J^+$   // $J^+$ is an answer set
        return $true$
    $literal, forced \leftarrow$ Heuristic($P$, $J$)
    if (Smodels($P$, $J \cup \{literal\}$) $= true$)
        return $true$
    if ($forced$)
        return $false$
    return Smodels($P$, $J \cup \{\neg literal\}$)
}

Expand($P$, $I$)
// $P$ is an extended logic program.
// $I$ is a set of literals representing a partial interpretation.
// This routine derives inferences that are true in
// any answer set of $P$ extending $I$,
// and adds the inferred literals to $I$.
// Returns the resulting $\varphi$ and $I$, as well as a flag
// indicating whether a conflict was found.
{
    repeat
        $I' \leftarrow I$
        $I \leftarrow AtLeast(P, I)$
        $I \leftarrow I \cup \{not\, x \mid x \in Unfounded(P, A)\}$
    until $I = I'$ or $Conflict(I)$
    return $I$
}

Conflict($I$)
{
    return $I^+ \cap I^- \neq \emptyset$
}
```

Table 3.2: Smodels algorithm

The two main subroutines of Expand are the *AtLeast* and *Unfounded*[11] routines, both of which are used to obtain inferences that will extend the current truth assignment. In Section 3.2.1 we describe how these functions operate on normal logic programs. Then, in Section 3.2.2 we look at how they apply to extended logic programs.

3.2.1 Smodels' Inference Rules for Normal Programs

Recall from Definition 4 that a normal rule is one of the form

$$h \leftarrow a_1, \ldots, a_k, not\ b_1, \ldots, not\ b_m$$

and that a normal logic program is a collection of normal rules.

AtLeast(P,I)

AtLeast uses inference rules that are valid under both the completion semantics and the answer set semantics. We summarize the four inference rules used by AtLeast below:

Modus Ponens: *If all of the subgoals in a rule*

$$a \leftarrow b_1, \ldots, b_k, not\ c_1, \ldots, not\ c_m$$

are true in the current truth assignment, infer a. For example, suppose that the program contains the rule $w \leftarrow x, y, not\ z$ and that the current truth assignment includes x, y and $\neg z$. Then infer w.

[11]In the Smodels source code, and in papers such as [54] and [55], the routine that computes unfounded sets is called the *AtMost* procedure.

All Rules Cancelled *If every rule with head a has its body cancelled by the current truth assignment,*[12] *infer $\neg a$.* For example, suppose that the only rules with w in their head are

$$\begin{array}{l} w \leftarrow x, \mathit{not}\ y \\ w \leftarrow \mathit{not}\ x, \mathit{not}\ z \end{array}$$

and that the current truth assignment contains $\neg x$ and z. Then infer $\neg w$.

Backchain True: *If atom a is true in the current truth assignment, and if every rule with head a* except one *has at least one subgoal that is false in the current truth assignment, infer all the subgoals of that remaining rule to be true.* For example, suppose the only rules with a in their head are

$$\begin{array}{l} a \leftarrow b, c, \mathit{not}\ d \\ a \leftarrow e, f \\ a \leftarrow \mathit{not}\ g, h \end{array}$$

and that the current truth assignment contains $a, d, \neg e$. Then infer $\neg g, h$.

Backchain False: *If an atom a is false in the current truth assignment and some rule*

$$a \leftarrow b_1, \ldots, b_k, \mathit{not}\ c_1, \ldots, \mathit{not}\ c_m$$

has every subgoal except one true in the current truth assignment, infer the remaining subgoal to be false. For example, suppose the rule is $a \leftarrow b, c, \mathit{not}\ d$ and that $\neg a, b, c$ are in the truth assignment. Then infer d.

The call $\mathit{AtLeast}(P, I)$ takes the logic program P and current truth assignment I, and applies the above four inference rules until no further inferences can be made,

[12]A normal rule: $h \leftarrow a_1, \ldots, a_k, \mathit{not}\ b_1, \ldots, \mathit{not}\ b_m$ has its body cancelled by I iff $\neg a_i \in I$ for some $1 \leq i \leq k$ or $b_i \in I$ for some $1 \leq i \leq m$

or until a conflict is reached[13], whichever comes first. The resulting truth assignment is returned as the result of the call.

Unfounded(P,I)

Given a logic program P and a partial truth assignment I, it is immediate from the definition of an unfounded set (Definition 12) that the union of a collection of unfounded sets is itself unfounded. Thus, the *greatest unfounded set* [20], $GUS(P, I)$, with respect to P and I is the union of all of the sets that are unfounded with respect to P and I. The call *Unfounded*(P, I) returns the set *GUS(P,I)*.

A basic subroutine used by the *Unfounded* procedure is the $MinimumModel$ algorithm for computing the minimum model of a Horn program (Section 2.2). Recall that Dowling and Gallier presented a version of this algorithm that runs in linear time in the size of the Horn program.

One algorithm that could be used to compute *GUS(P,I)* is the following:

1. Let P' be the set of rules in P minus all rules that have their body cancelled by I.

2. Let P'' be the set of rules in P' but with all negative subgoals removed from all of the rule bodies. (Hence, P'' is a Horn program.)

3. Return $GUS(P, I) = Atoms(P) \setminus MinimumModel(P'')$.

However, the preceding algorithm is not as efficient as one would like for use in Smodels: the routine recomputes *MinimumModel*(P'') from scratch every time that the *Unfounded* routine is called. The *Unfounded* routine can be called multiple times from *Expand* each time that *Expand* is called by the *Smodels* algorithm.

[13]i.e., for some atom x, both x and $\neg x$ are in the truth assignment

Note that the GUS operator is monotonic in the sense that, if $I \subseteq I'$, then $GUS(P, I) \subseteq GUS(P, I')$. So if $GUS(P, I)$ has already been computed, and the set $I' \setminus I$ is not very large, one might hope to localize the computation of $GUS(P, I') \setminus GUS(P, I)$ to a relatively small fraction of the rules in P. Smodels accomplishes this through an optimization that uses *source pointers*.

For each atom $a \in Atoms(P)$, a source pointer $a.source$ is maintained which points to the first rule that caused a to be excluded from $GUS(P, I)$. During the execution of the *AtLeast* routine, if it is detected that a rule

$$R = \ a \leftarrow Body$$

has its body cancelled by a new guessed or inferred assignment, and $a.source = R$, then a is entered into a set U.

Then the call $Unfounded(P, I)$ works as follows:

1. Close set U under the following operator: If atom $h \notin U$ and the rule pointed to by $h.source$ has a positive subgoal $b \in U$ then $U \leftarrow U \cup \{h\}$.

2. Remove elements from U as follows: If $a \in U$ and rule

$$R = \ a \leftarrow Body$$

   is such that I does not cancel R's body and $PosSubgoals(R) \cap U = \emptyset$, then $U \leftarrow U \setminus \{a\}$. (Also, set $a.source = R$.)

3. Return $GUS(P, I) = U$.

Step 2 of the above algorithm essentially closes the set $\overline{U}$ using the Dowling-Gallier algorithm on P^I.

3.2.2 Smodels' Inference Rules for Extended Programs

As mentioned earlier, Lparse translates any extended logic program into a set of normal rules, choice rules, cardinality rules, and weight rules. For each of these rule types, Smodels implements the Modus Ponens, All Rules Cancelled, Backchain True, and Backchain False inference rules, as well as negation based on unfounded sets. In the preceding section, we explained how these inference rules are implemented for normal logic programming rules. In this section, we will explain how they are implemented for the choice, cardinality, and weight rules.

Choice Rules

Recall from Section 2.4 that a choice rule is a rule of the form

$$\{h_1, \ldots, h_k\} \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m.$$

Such a rule is equivalent to having k rules of the form

$$\{h_i\} \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m \tag{3.1}$$

where $1 \leq i \leq k$. The Backchain True, All Rules Cancelled, and Unfounded Set inference rules treat a rule of the form (3.1) exactly as though it were a normal rule with h_i at the head. The Modus Ponens and Backchain False inference rules, however, do not apply to choice rules.

Cardinality Rules

A cardinality rule is one of the form

$$h \leftarrow k\ \{a_1, \ldots, a_n, not\ b_1, \ldots, b_m\}. \tag{3.2}$$

With respect to implementing the Smodels inference rules, the main differences between a cardinality rule and a normal rule are the conditions under which the rule fires (i.e., when Modus Ponens applies) and the conditions under which the rule is cancelled. A normal rule

$$h \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m$$

fires when $n + m$ of its subgoals are satisfied, and is cancelled when any one of its subgoals is contradicted. A rule of form (3.2) fires when any k of its subgoals are satisfied, and is cancelled when any $n + m - k + 1$ of its subgoals are contradicted.

The threshold for firing affects when Modus Ponens and Backchain False are applied to the rule. For instance, suppose that h has been set to *false* and $k - 1$ of the subgoals in the rule body have been satisfied. Then the Backchain False inference rule is applied to ensure that the remaining subgoals of the rule are contradicted.

The threshold for rule cancellation affects the application of the All Rules Cancelled and Backchain True. For instance, if the head atom h from rule 3.2 has been set to *true*, all other rules with h in their head have been cancelled, and $n + m - k$ of the subgoals of this rule have been contradicted, then the remaining k subgoals may be inferred *true*.

Both thresholds are involved in implementing the inference rule for unfounded sets. The reason for this is evident from the algorithm given for the *Unfounded* procedure outlined in Section 3.2.1. The first phase of the procedure adds elements to the set U when supporting rules are (in a sense) “cancelled”. Then the second phase of the procedure removes elements from U when new supporting rules are found through a rule-firing process.

Weight Rules

Recall that a weight rule is a rule of the form

$$h \leftarrow l \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \textit{not } b_1 = w_{b_1}, \ldots, \textit{not } b_m = w_{b_m}\}. \tag{3.3}$$

As with cardinality rules, a weight rule has two important thresholds: the firing threshold, l, and the cancelling threshold $w = \sum_{i=1}^{n} w_{a_i} + \sum_{i=1}^{m} w_{b_i} - l$. The rule is cancelled only if the sum of the cancelled subgoals is greater than w.

The main difference in implementing weight rules, as opposed to cardinality rules, comes from fact that the weights mentioned in the body of the rule may differ from each other. This presents no extra complication regarding the Modus Ponens, All Rules Cancelled, and Unfounded Set inference rules, but does somewhat complicate the implementation of the two backchaining inference rules.

First, we will discuss how Smodels' Backchain True inference works for weight rules. Suppose that rule R is of the form shown in (3.3). Suppose that d is the combined weight of the cancelled subgoals in the body of R. Suppose also that x is an uninstantiated subgoal of maximal weight from the body of R. Then the Backchain True inference rule can be applied to R to infer that x is *true* if and only if (1) the atom h is *true*, (2) all of the rules other than R that have h in their heads have been cancelled, (3) $d \leq w$, and (4) $d + w_x > w$ (where w is the cancelling threshold defined above).

Once subgoal x has been inferred *true* by backchaining, Smodels checks whether there are any more uninstantiated subgoals in the body of R. If so, then x is reset to refer to the uninstantiated subgoal of maximal weight, as before. If we again have $d + w_x > w$ then the new x is also inferred to be *true* by the Backchain True inference rule. The rule is repeatedly applied to R until there is no subgoal x such that $d + w_x > w$.

Next, we consider how Smodels applies the Backchain False inference rule to rule R. Suppose that s is the combined weight of the *satisfied* subgoals in the body of R. Suppose that x is an uninstantiated subgoal of maximal weight from the body of R. The Backchain False inference rule can be applied to R to infer that x is *false* if and only if (1) h is *false*, (2) $s < l$, (3) $s + w_x \geq l$.

As with the Backchain True inference rule, the Backchain False inference rule for weight rules may make several inferences by iterating through the uninstantiated subgoals in the body of the rule. The iteration goes through the subgoals in decreasing order of their weights and sets each subgoal to *false* until there is no uninstantiated subgoal x remaining in the rule such that $s + w_x \geq l$.

3.2.3 Lookahead-Based Heuristic

Smodels' heuristic routine picks the choice atom a, which will be the next atom instantiated in the current truth assignment. It returns a if Smodels is to first instantiate a to *true*, and returns $\neg a$ if the initial value of a is to be *false*.

Perhaps the biggest drawback of Smodels' heuristic routine is that it is often rather expensive to compute. A significant advantage of the routine is that it sometimes provides new *forced* inferences, rather than just guessed inferences.

The heuristic attempts to choose a in such a way that instantiating a will yield a large number of inferences. Suppose that I is the current truth assignment being applied to logic program P. Then, for each uninstantiated atom $a \in Atoms(P)$ define

$$PosScore(a) \quad = \quad |Expand(P, I \cup \{a\})|$$

$$NegScore(a) \quad = \quad |Expand(P, I \cup \{\neg a\})|.$$

Smodels' heuristic returns the atom a such that $min(PosScore(a), NegScore(a))$ is maximized. Ties are broken by choosing a such that $max(PosScore(a), NegScore(a))$ is maximized. Ties that still remain are then broken randomly. The process of calling the *Expand* routine on $I \cup \{a\}$ and $I \cup \{\neg a\}$ is called a *lookahead*.

Calling the Expand routine on every uninstantiated atom at every choice point in the search may be very expensive. To partially alleviate this expense, Smodels does not perform a lookahead on an atom b if, during the execution of the same call to the heuristic, a lookahead was performed on some atom a such that b or $\neg b$ was an element of $Expand(P, I \cup a)$ or $Expand(P, I \cup \neg a)$. The reason for omitting the lookahead for b in this case is that if, for example, $b \in Expand(P, I \cup \{a\})$, then it is guaranteed that

$$Expand(P, I \cup \{b\}) \subseteq Expand(P, I \cup \{a\}).$$

Therefore, one may regard it as somewhat likely (though not certain) that a will score better than b on Smodels' heuristic.

Once the atom a with the best heuristic score has been selected, then a is returned as the choice literal if $PosScore(a) > NegScore(a)$. Otherwise, $\neg a$ is returned.

Obtaining forced inferences from lookaheads

If, for some literal x, $Expand(P, I \cup \{x\})$ is found to contain a conflict (i.e., both a and $\neg a$ for some atom a), it is valid to infer that x is *false* in any answer set that extends I. In such a situation the heuristic routine can immediately return $\neg x$ as a *forced* inference. Obtaining these forced inferences is potentially a very valuable benefit that is earned from the (potentially quite expensive) lookahead process.

Pseudocode for Smodels' heuristic routine in given in Table 3.3

3.2.4 Computing Multiple Answer Sets

The default behavior of Smodels is that it seeks to find and display a single answer set for the given logic program. However, a command-line option exists that allows the user to request that *all* answer sets for a given logic program be computed and displayed. The user can also request that Smodels compute and display up to n answer sets for the program, where n is a user-specified, positive integer.

Suppose the user has requested that Smodels compute multiple answer sets. Then, whenever Smodels computes a new answer set, it displays the answer set and checks whether the desired number of answer sets has been met. If not, then it backtracks from the most recent assignment, and continues its search.

3.2.5 Implementing Minimize Statements

In the case where the program P contains minimize statements, then Smodels must, at least implicitly, iterate through all of the answer sets of P and compare their scores relative to those statements. After all answer sets have been enumerated, the one with the best score is returned as the solution.

An optimization to this scheme that Smodels uses is to modify the Conflict routine so that it compares the score of the current partial truth assignment to the score of the best answer set found so far. (The score of a partial truth assignment is the total weight of the subgoals satisfied by the assignment. Thus it provides a lower bound on the score that can be achieved by any assignment that extends the current assignment.) If the score of the current partial assignment is already worse (i.e.

```
Heuristic($P$, $I$)
// Returns the next literal that is to be set to $true$
// by the search routine.
// Also, returns a Boolean value that indicates whether the literal
// represents a forced inference.
{
    $Avail \leftarrow Atoms(P) \setminus Atoms(I)$
    $bestMin \leftarrow bestMax \leftarrow 0$
    for $a \in Avail$
        $I' \leftarrow Expand(P, I \cup \{a\})$
        if $Conflict(I')$
            return $\neg a$, $true$
        $Avail \leftarrow Avail \setminus Atoms(I')$
        $posScore \leftarrow |I'|$
        $I' \leftarrow Expand(P, I \cup \{\neg a\})$
        if $Conflict(I')$
            return $a$, $true$
        $Avail \leftarrow Avail \setminus Atoms(I')$
        $negScore \leftarrow |I'|$
        $min \leftarrow min(posScore, negScore)$
        $max \leftarrow max(posScore, negScore)$
        if $(min > bestMin)$ or $(min = bestMin$ and $max > bestMax)$
            $bestMin \leftarrow min$
            $bestMax \leftarrow max$
            $bestAtom \leftarrow a$
            $bestPolarity \leftarrow (posScore > negScore)$
    if $(bestPolarity)$
        return $bestAtom$, $false$
    else
        return $\neg bestAtom$, $false$
}
```

Table 3.3: Smodels' lookahead-based heuristic

higher) than the score of the best known answer set, then Conflict returns *true*, indicating that there is no purpose in extending this assignment.

CHAPTER 4

CONFLICT CLAUSES FOR SAT AND ASP

In this chapter we will discuss how SAT solvers generate conflict clauses and how they use these clauses to improve search efficiency. In parallel, we will discuss how we have adapted these techniques into our Answer Set Programming solver, Smodels$_{cc}$ ("Smodels with conflict clauses").

The algorithmic techniques that are original to this dissertation are discussed in Sections 4.3.2-4.3.4 and 4.4.2, where we explain how Smodels$_{cc}$ diagnoses the cause of a conflict in an answer set search. (See also Appendices C and D, where we prove that the resulting Smodels$_{cc}$ search algorithm is correct and complete.)

4.1 Overview of Conflict Clauses

A conflict clause is a backtracking solver's diagnosis of why a conflict occurred. The solver can use this diagnosis to prune the search space and to direct the search heuristic.[14] Any solver that is based upon the Davis-Putnam-Loveland-Logemann algorithm can in principle utilize conflict clauses. This observation, combined with the success of conflict clause learning in improving the efficiency of DPLL-based SAT

[14]Conflict clauses are sometimes referred to as "lemmas" because they are intermediate results that may be reused by the solver as it develops a solution to the original problem.

solvers on industrial applications, was our motivation for adapting conflict clauses to ASP search.

In this section, we will motivate the general idea of conflict clause learning by an abstract example of DPLL-based search. This example will be equally applicable to SAT search or to ASP search.

Let us suppose that we have a DPLL-based solver that is searching for the solution to a problem. Suppose further that, before reaching the first conflict in its search, the solver has selected atoms $a_1, a_2, \ldots, a_{100}$, in this order, as its choice atoms, and that it has initially instantiated each of these atoms to the value *true*. Assume also that immediately after instantiating a_{100} to *true*, it reaches a conflict in the search (i.e., these 100 assumptions have caused the search engine to deduce that some other atom x is both *true* and *false*). This situation is portrayed in Figure 4.1.

After reaching this conflict, the basic DPLL algorithm would backtrack the assignment of $a_{100} = true$, set $a_{100} = false$, and continue the search. (See Figure 4.2.)

However, if our algorithm incorporates conflict clauses, it would pause immediately after the conflict is detected in Figure 4.1 in order to to analyze the "cause" of the conflict. In solvers using the same conflict clause strategy as Chaff and Smodels$_{cc}$, the diagnosis returned will identify a subset of the assignments made that would, in the context of the problem at hand, have been sufficient to result in the conflict that is being analyzed.

For instance, suppose in our example that the conflict analysis routine returns a diagnosis which states that the assignments $a_2 = true$, $a_3 = true$, and $a_{100} = true$, taken together, were sufficient to cause the conflict in Figure 4.1. (In Sections 4.3 through 4.5, we will explain how this diagnosis is obtained.) From this diagnosis,

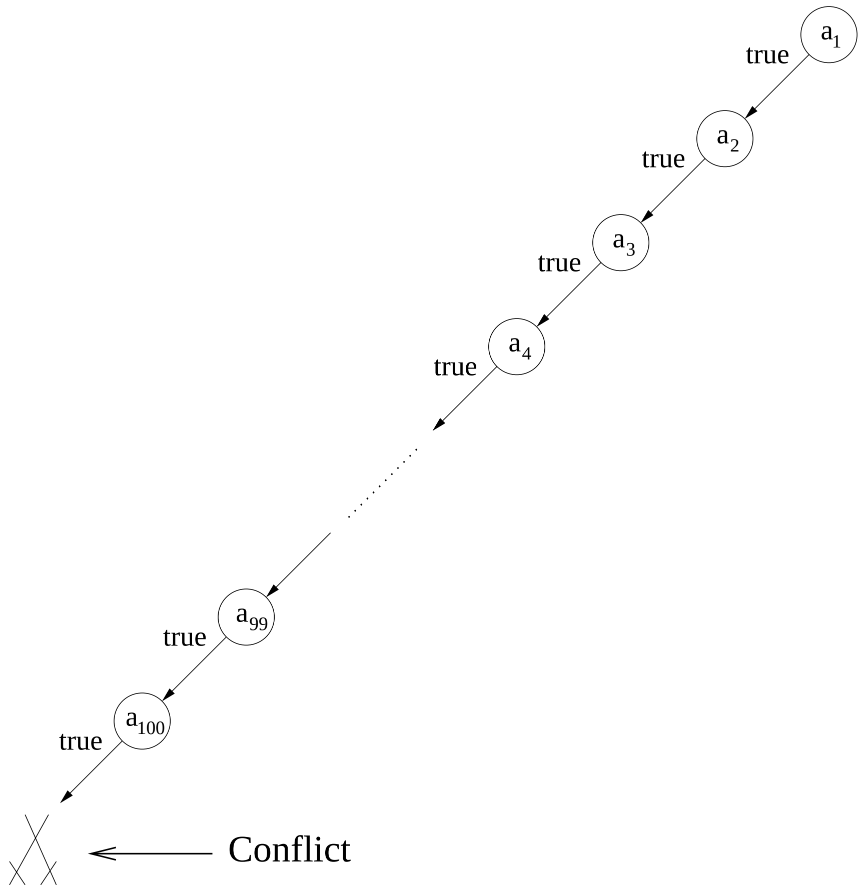

Figure 4.1: Reaching the first conflict in a DPLL-based search

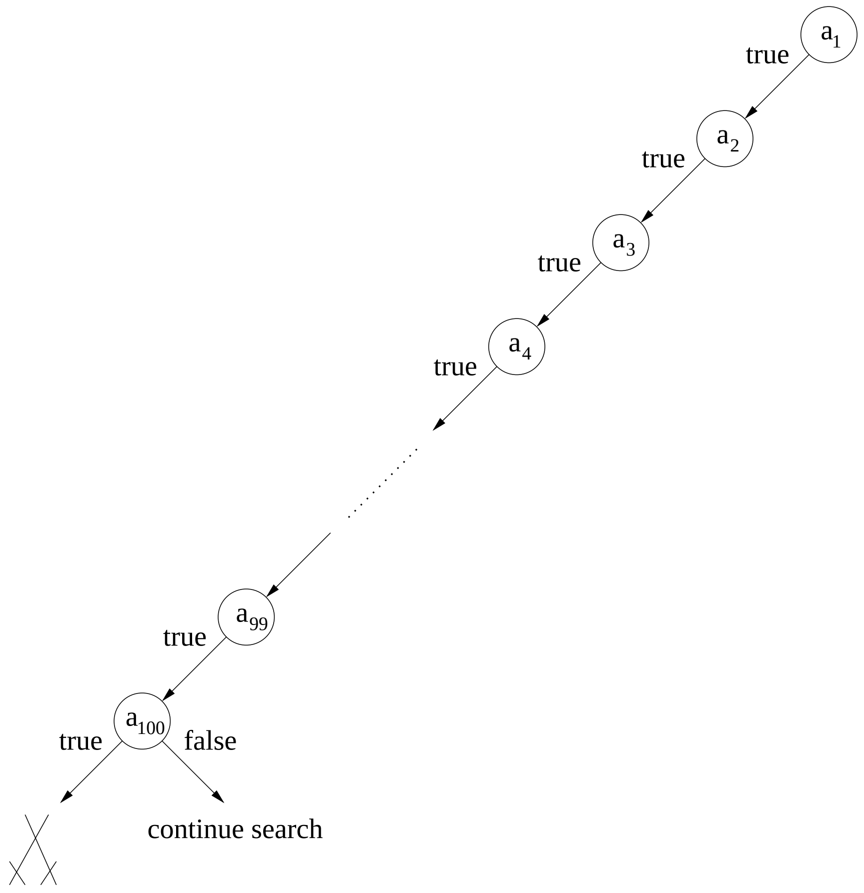

Figure 4.2: Backtracking from the first conflict

we may infer that no solution of the problem at hand can set a_2, a_3, and a_{100} simultaneously to *true*. This results in our solver inferring the classical logic clause: $\neg a_2 \vee \neg a_3 \vee \neg a_{100}$.

Once this conflict clause has been computed, it may be used by the solver in three ways to improve the search efficiency: (i) to direct backjumping, (ii) to serve as an added constraint, and (iii) to guide the search heuristic.

Backjumping, also known in the SAT literature as "nonchronological backtracking" [44], potentially allows the solver to remove several levels of assumptions from the search when backtracking from a single conflict. As we saw in Figure 4.2, the basic DPLL algorithm (which uses "chronological backtracking") removes only the assumption $a_{100} = \textit{true}$ from its stack, and sets $a_{100} = \textit{false}$ at the 99-th level of the search. (We define the *search level* of an inference to be the number of choice assignments that were in effect when the inference was made.) In contrast, a solver utilizing the conflict clause/diagnosis $\neg a_2 \vee \neg a_3 \vee \neg a_{100}$ can backjump past the assumptions of $a_{99} = \textit{true}$, ..., $a_4 = \textit{true}$ since none of these assumptions were required in order to produce the conflict. The solver then assigns $a_{100} = \textit{false}$ at level 3 of the search, as shown in Figure 4.3. (In the figure, the assignment $a_{100} =$ false does not appear in a bubble because it is a forced assignment in this circumstance.)

The second use of a conflict clause is to serve as an added ("learned") constraint. In our example, immediately after backjumping, the solver would store the clause $\neg a_2 \vee \neg a_3 \vee \neg a_{100}$ for later reference. Suppose, then, that later in the search the current partial assignment includes $a_2 = \textit{true}$ and $a_{100} = \textit{true}$. Then, based on the stored conflict clause, the solver could immediately infer that $a_3 = \textit{false}$, thereby further pruning the search space. Moskewicz *et al.* [47] provide the following intuition

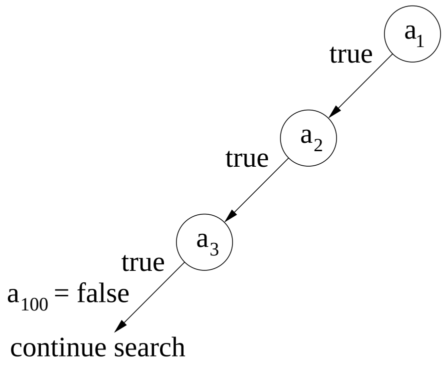

Figure 4.3: Situation after backjumping based on $\neg a_2 \vee \neg a_3 \vee \neg a_{100}$

for using conflict clauses in this way: If we interpret reaching a conflict in the search as the solver making a "mistake", then a conflict clause can be seen as the solver's attempt to diagnose the cause of the mistake, and to learn from it so that future instances of the same mistake are not repeated.

The third use of conflict clauses is to guide the search heuristic. Part of the role of the search heuristic in the DPLL algorithm is to determine, at each choice point in the search, which atom to instantiate next. In our example, suppose that, as the search proceeds, the atom a_3 appears in a relatively large number of conflict clauses. This suggests that a_3 is an important atom in pruning the search space. Therefore, a high weight can be assigned toward selecting a_3 as the next choice atom whenever that atom is uninstantiated.

An additional role of the search heuristic in DPLL is to determine whether to initially instantiate the choice atom to *true* or to *false*. In our example, suppose that a_3 has appeared negatively in more conflict clauses than it has appeared positively. Then our solver might prefer to instantiate a_3 to *false* at this point in the search, *false* being the value that would agree with the greater number of conflict clauses in which

a_3 appears. A possible intuition as to why this is a good way to choose a_3's initial value is as follows: The fact that a_3 appears negatively in most of its conflict clauses suggests that assigning a_3 a *positive* value has led to a large number of conflicts. This in turn suggests that it may be a good idea to initially instantiate a_3 to *false* in the search for a solution.

4.2 Relationship to Previous Work

Work on backtracking algorithms that diagnose the causes of their conflicts, and use the results of the diagnoses to direct backjumping and to prune the search space, date back in the artificial intelligence literature at least to the work of Stallman and Sussman in [57]. Successfully applying this idea to satisfiability search seems, however, to be a relatively recent development. GRASP [44] and rel_sat [3] seem to have been among the earliest SAT solvers to successfully incorporate these ideas, both doing so via conflict clause techniques as discussed in this dissertation. Two of the most effective recent SAT solvers for industrial applications, Chaff [47] and BerkMin [27], have refined some of the conflict clause techniques, and have also greatly influenced the design of Smodels$_{cc}$.

The program Smodels$_{cc}$, which we created to incorporate conflict clause learning into answer set search, is based on the algorithms and source code of the original Smodels program. The primary reasons for basing our program on Smodels were (i) its wide usage, (ii) its reputation for high performance compared to other answer set solvers, and (iii) the fact that it is an open source program. (The Smodels source code is under the GNU Public License, as is the Smodels$_{cc}$ code.) The main areas in which we modify or add to Smodels pertain to the following:

- Diagnosing the cause of each conflict in the search,
- Generating and storing the corresponding conflict clause,
- Changing the chronological backtracking of Smodels to nonchronological backtracking based on the most recently generated conflict clause,
- Obtaining inferences from the conflict clauses via Boolean constraint propagation,
- Modifying the search heuristic to one based on conflict clauses,
- Deleting older conflict clauses that are judged (based on some heuristic) to be less likely to be useful in the remaining search, and
- Periodically restarting the search.

Each of the items listed above is a feature that was incorporated into GRASP, rel_sat, Chaff, and BerkMin, although these four SAT solvers differ in the specifics of how they have incorporated these features. It is only the first feature, diagnosing the conflict, which is significantly different in Smodels$_{cc}$ from the SAT solvers. The process of adding conflict diagnosis to Smodels is considerably more complex than adding it to a DPLL-based SAT solver. This is because of the more varied and complex set of inference rules present in Smodels. (Smodels' *unfounded set* inference rule is particularly relevant in this regard, as we shall see.) Sections 4.3 through 4.5 below detail the process of performing a conflict diagnosis and generating the corresponding conflict clause, both in a SAT solver and in Smodels$_{cc}$.

The techniques that Smodels$_{cc}$ uses for implementing the remaining items in the above list are based rather closely on techniques used in GRASP, Chaff, and BerkMin, as we detail in Section 4.6: "Using Conflict Clauses in the Search".

4.3 Generating the Implication Graph

The Implication Graph (IG) is a directed graph that is constructed during the search process. Its role is to give an account of the chain of reasoning that led to each forced inference that was made during the search. In particular, when a conflict has occurred, the implication graph is used to analyze the chain of inferences leading to the pair of conflicting literals.

Each node in the implication graph represents a literal that is assigned *true* by the solver's current partial assignment. The edges coming into a node represent the set of assignments that caused that node's literal to be inferred by the search engine. Each literal that is represented in the implication graph is represented by only one node. (We will see example implication graphs in the next section.)

4.3.1 Generating the Implication Graph in a SAT Solver

As discussed in Section 3.1, the great majority of DPLL-based SAT solvers use only one rule of inference in their search: the unit clause rule. This rule states that when a clause has all of its literals contradicted except for one, then the remaining literal will be inferred to be true. (Recall also that the process of closing the current partial assignment under the unit clause rule is known as Boolean Constraint Propagation, or BCP.) Let *lit* be some literal that was inferred based on applying the unit literal rule to a clause C. Then the edges going into the IG node for *lit* will correspond to the assignments that falsified the other literals in C. As an example, suppose that

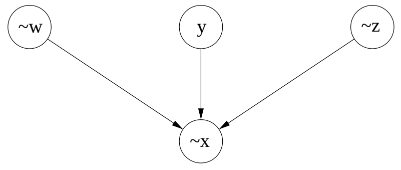

Figure 4.4: Implication graph for a single inference

the SAT problem contains the clause $w \lor \neg x \lor \neg y \lor z$, and that the current partial assignment includes $w = \mathit{false}$, $y = \mathit{true}$, and $z = \mathit{false}$. Then $x = \mathit{false}$ would be inferred by the unit clause rule, and the nodes and edges shown in Figure 4.4 would appear in the implication graph.

If a literal is satisfied in the current partial assignment by a choice assignment made by the search heuristic, then there will be no incoming edges for the corresponding node.

Figure 4.5 shows a more extensive SAT-based implication graph. In this example, we assume that a, $\neg b$, $\neg c$, d, $\neg e$, and f were all inferred or assumed at earlier levels in the search. s was chosen as the choice atom at the current search level and was initially instantiated to *true*. t was inferred from the clause $b \lor \neg s \lor t$. The solver continued the BCP process until z was inferred from the clause $\neg a \lor x \lor z$, and $\neg z$ was inferred from the clause $\neg w \lor \neg y \lor \neg z$.

The Conflict Node

After $\neg z$ has been inferred in this example, a conflict exists in the current partial assignment. This conflict is represented by the pair of conflicting nodes z and $\neg z$. At this point, the search engine would cease making new inferences and would prepare to perform its conflict diagnosis. In the GRASP solver, the node from the conflicting pair that was inferred later is labeled the *conflict node*, and a *conflict edge* is added

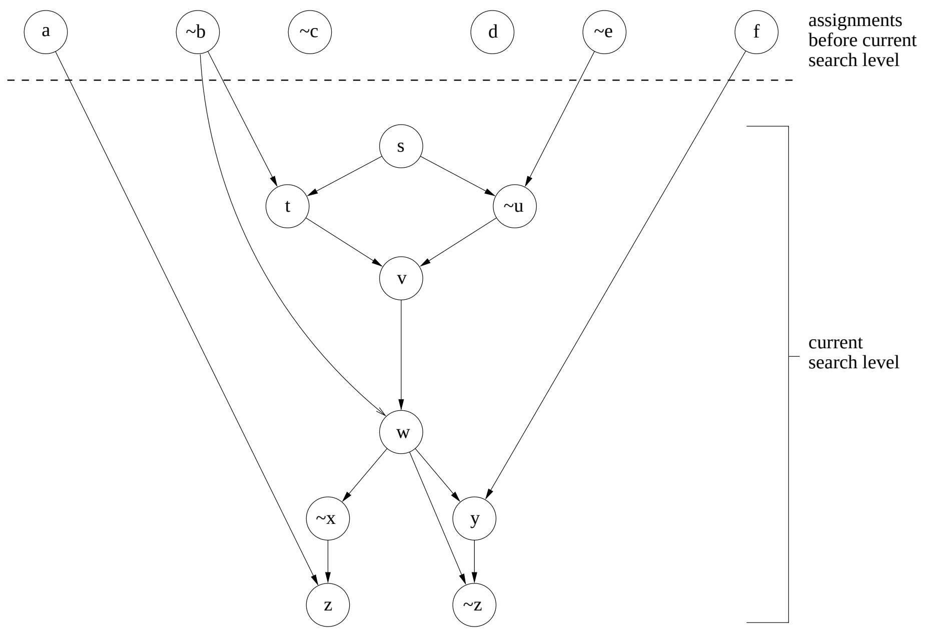

Figure 4.5: A SAT-based implication graph with conflict

that goes from the preceding element of the conflicting pair to the conflict node. In our example, suppose that z had been inferred before $\neg z$. Then $\neg z$ would be labeled by GRASP as the conflict node, and an edge would be included from node z to node $\neg z$. Figure 4.6 gives the completed implication graph for this example.

A relatively naive conflict clause

From the example graph that we have just considered, it is easy to see that the combination of assignments $a = \mathit{true}$, $b = \mathit{false}$, $e = \mathit{false}$, $f = \mathit{true}$, and $s = \mathit{true}$ led to the conflicting pair of assignments involving z. From this we could infer the conflict clause $\neg a \vee b \vee e \vee \neg f \vee \neg s$. This diagnosis can be generated by traversing the graph backwards from the conflict until nodes from earlier search levels, or else the choice node, are reached. In Sections 4.4.2 and 4.5 we will see how to obtain a shorter, and probably more useful, conflict clause from this graph.

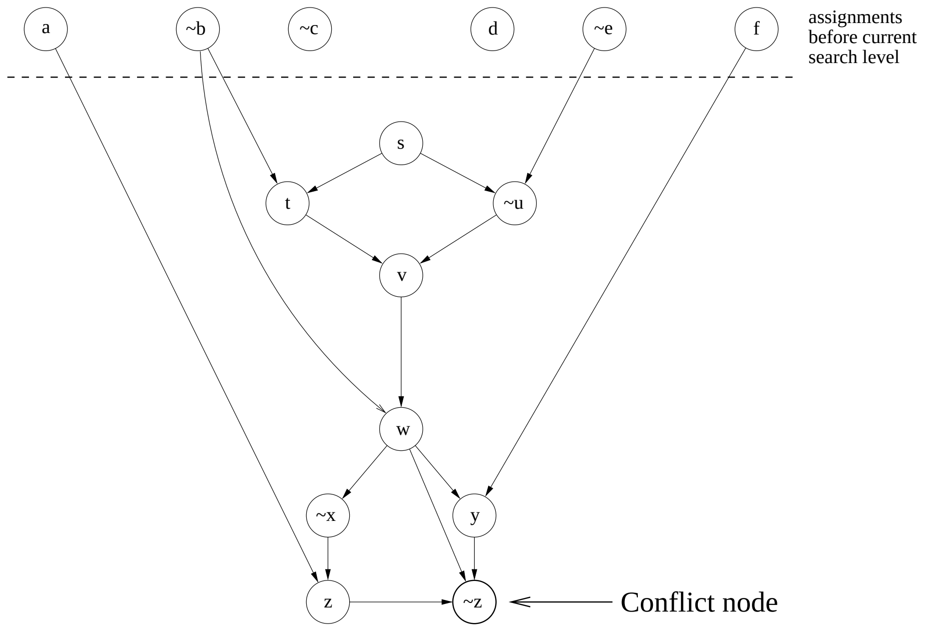

Figure 4.6: Adding a conflict edge

Representing the SAT-based implication graph

Because the unit clause rule is the only inference rule in effect in a DPLL-based SAT solver, constructing and representing the implication graph is quite easy: It is sufficient that each inferred literal maintains a reference to the clause that was used to infer it.

Properties of the SAT-based implication graph

In our example, the conflicting nodes z and $\neg z$ both occurred at the last level of the search. In fact, this will always be the case when a conflict is reached in a DPLL-based SAT search engine using the unit clause rule as its only inference rule. That is, it is impossible for such a solver to infer a literal at one level of the search and then to infer its negation at a later level. For instance, if z is set to *true* at some level of the search, then the literal $\neg z$ will be (effectively) removed from all clauses in which it appears before the search engine goes on to create a new choice point. Thus, $\neg z$ will not be present in any of the clauses at the later search levels.

A second property of the SAT-based implication graph is that it is always acyclic. This is because a literal is never inferred from the unit clause rule unless all of the other literals in the clause have already been contradicted by the current partial assignment.

Finally, a third property of a SAT-based implication graph is that it contains at most one pair of conflicting nodes. This is because the BCP process is interrupted as soon as the first conflicting pair of inferences is made.

In the next section we will see that, due to the unfounded set inference rule, each of these three properties can be violated in the implication graphs constructed by Smodels$_{cc}$.

4.3.2 Generating the Implication Graph in an ASP Solver

As mentioned above, current SAT solvers generally have only one data type for expressing their constraints (the clause) and usually apply only one rule of inference (the unit clause rule). Smodels, however, works with a number of different logic programming rule types, and also has a number of different rules of inference that it applies. In this section we will discuss how Smodels$_{cc}$ constructs its implication graph when searching for a model for a *normal* logic program (i.e., a program with normal rules only). In Section 4.3.3 we will construct an example implication graph from part of such an answer set search. In Section 4.3.4, we will discuss how Smodels$_{cc}$ constructs an implication graph when searching for a model of an extended logic program.

So, let us assume that the logic program P is normal. We recall from Section 2.2 that this means that every rule in P is of the form $a \leftarrow b_1, \ldots, b_k, not\ c_1, \ldots, not\ c_m$, where $a, b_1, \ldots, b_k, c_1, \ldots, c_m$ are all atoms. Section 3.2.1 provided a description of each of the five inference rules that Smodels applies to normal rules. Below, we review these inference rules, and describe how Smodels$_{cc}$ constructs the corresponding implication graph edges when each of these rules is applied.

Modus Ponens: *If all of the subgoals in a rule*

$$a \leftarrow b_1, \ldots, b_k, not\ c_1, \ldots, not\ c_m$$

are true in the current truth assignment, infer a. Corresponding to this inference, we add implication graph edges from each of the b_i's and $\neg c_j$'s to a.

All Rules Cancelled: *If every rule with head a has at least one subgoal negated in the current truth assignment, infer $\neg a$.* For each rule $a \leftarrow b_1, \ldots, b_k, not\ c_1, \ldots, not\ c_m$ with head a, determine a cancelling assignment, $\neg b_i$ or c_j that occurred before the current application of this inference rule. Add an edge from that assignment to $\neg a$ in the implication graph.[15]

Backchain True: *If atom a is true in the current truth assignment, and if every rule with head a* except one *has at least one subgoal that is false in the current truth assignment, infer all the subgoals of that remaining rule to be true.* Each subgoal inferred by the same application of this rule will have the same set of predecessors in the implication graph. One of the predecessors will be the literal a. The remaining

[15]For a given cancelled rule, there may be several cancelling assignments from which to choose. From the available cancelling assignments, Smodels$_{cc}$ always chooses the assignment that was made *first*. This is also the approach that it uses for selecting cancelling assignments for the Backchain True and Unfounded Set inference rules. However, for each of these rules, the fact that Smodels$_{cc}$ happens to choose the first cancelling assignment is not critical to the correctness of the algorithm.

predecessors will be the reasons that cancelled the remaining rules with the same head (one cancelling assignment per rule). For example, suppose the only rules with a in their head are $a \leftarrow b, c, \mathit{not}\ d$; $a \leftarrow e, f$; and $a \leftarrow \mathit{not}\ g, h$; and that the current truth assignment contains $a, d, \neg e$. Then $\neg g, h$ will be inferred. Add edges from each of $a, d, \neg e$ to each of $\neg g, h$ into the implication graph.

Backchain False: *If an atom a is false in the current truth assignment and some rule*

$$a \leftarrow b_1, \ldots, b_k, \mathit{not}\ c_1, \ldots, \mathit{not}\ c_m$$

has every subgoal except one true in the current truth assignment, infer the remaining subgoal to be false. The predecessors of the inferred literal will be $\neg a$ plus the assignments that set the other subgoals to *true*. For example, suppose the rule is $a \leftarrow b, c, \mathit{not}\ d$ and that $\neg a, b, c$ are in the truth assignment. Infer d, and add edges from each of $\neg a, b, c$ to d.

Unfounded Sets: *Determine U, the greatest unfounded set of the program with respect to the current truth assignment. Set each of the atoms in U to false.* See Section 3.2.1 for a description of how Smodels computes U.

The algorithm for determining the implication graph edges for unfounded sets is similar to the one for All Rules Cancelled. Let I be the current partial assignment, including $\neg a$ for every $a \in U$. For each rule R whose head atom a is in U, choose a literal $x \in I$ such that x cancels the body of R. Such a literal x must exist since U is an unfounded set (see Definition 12 in Section 2.3.4) and all of the atoms in U have just been inferred *false*. Include an edge from x to $\neg a$ in the implication graph.

As an example where the unfounded set inference rule is applied, suppose that the program P reads as follows:

$$\begin{array}{rcl}
b & \leftarrow & \mathit{not}\ b' \\
b' & \leftarrow & \mathit{not}\ b \\
c & \leftarrow & \mathit{not}\ c' \\
c' & \leftarrow & \mathit{not}\ c \\
d_1 & \leftarrow & b \\
d_1 & \leftarrow & d_2 \\
d_2 & \leftarrow & d_3 \\
d_3 & \leftarrow & d_1,\ c \\
d_4 & \leftarrow & d_3
\end{array}$$

Suppose that at the beginning of the search for an answer set, Smodels$_{cc}$ performs a choice assignment setting $b = \mathit{false}$. Then $U = \{d_1, d_2, d_3, d_4\}$ is an unfounded set. Also, b' will be inferred by *modus ponens.* The implication graph that corresponds to these inferences is shown in Figure 4.7. Note that in this situation, the implication graph contains a cycle among some of the literals inferred from the unfounded set test.

Complications Introduced by the Unfounded Sets Inference Rule

The first four of the inference rules mentioned above correspond to inferences that would be made by a SAT solver if it were run on Clark's completion of the logic program (see Section 2.3). In fact, this is exactly the approach taken by the Cmodels-1 solver, which we will discuss in Section 5.1. So, it is not surprising that, if we were to restrict Smodels$_{cc}$ to these four inference rules, then the implication graphs that it produces would have much in common with the implication graphs produced

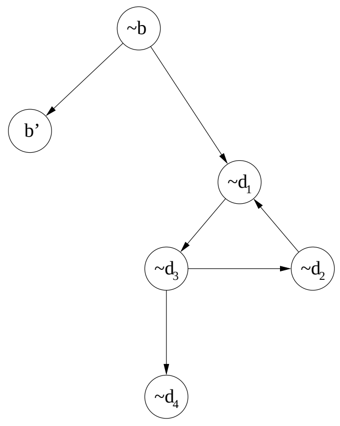

Figure 4.7: Implication graph involving unfounded set inferences

by SAT solvers such as GRASP, Chaff, and BerkMin. In particular, the IG would always be acyclic. Also, (after rather slight and inconsequential modifications to the original Smodels part of the code) it would be the case that whenever a conflict is detected there would be exactly one pair of conflicting nodes in the IG, and both of these nodes would appear at the current search level.

However, because Smodels$_{cc}$ does utilize the Unfounded Set inference rule from Smodels, it is possible (1) for the IG to have cycles, (2) for there to exist a conflicting pair of nodes x and $\neg x$ where x was inferred prior to the current search level, and (3) for there to be multiple pairs of conflicting literals appearing simultaneously.

Situation (1) has already been demonstrated by our example in Figure 4.7. The example that we give in Section 4.3.3 will present a case where situations (1)-(3) all occur simultaneously. In order to ensure correctness and completeness of the Smodels$_{cc}$ algorithm, it will be necessary to deal with these complications properly.

The Conflict Node

As is the case with the DPLL-based SAT solvers, a conflict is reached in Smodels$_{cc}$ when some atom has been been assigned both the values *true* and *false*. Smodels$_{cc}$, like GRASP, designates the node corresponding to the later of these two conflicting assignments as the conflict node, and adds a conflict edge from the other node in the conflicting pair to the conflict node.

We noted above that, because of the Unfounded Set inference rule, there may be multiple conflicting node pairs in the implication graph constructed by Smodels$_{cc}$. Which pair we select, and therefore which node we select as our conflict node, will affect which conflict clause we generate. This may, in turn, affect the performance and completeness of our search algorithm. For instance, it is theoretically possible for our algorithm to enter an infinite loop if we are not careful about which conflict node we select. To understand how the selection of the conflict node relates to performance and completeness, we will need to look at how the selection of an appropriate *Unique Implication Point*, or UIP, affects the algorithm. Therefore, we will postpone further discussion of this issue until Section 4.4.2.

4.3.3 An ASP Implication Graph Example

In this section we will construct an example implication graph corresponding to a conflict in an answer set search. Then, in Sections 4.4 and 4.5, we will use this example to demonstrate the steps involved in computing the corresponding conflict clause.

Suppose that Smodels$_{cc}$ is searching for an answer set to a normal logic program P. Assume that P contains the following rules mentioning atoms $a, b, c, d, e, f, u, v, w, x, y, z$:

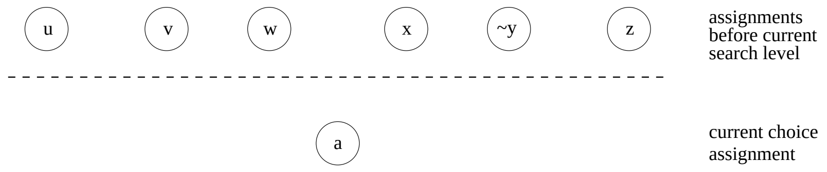
u
v
w
x
~y
z
assignments before current search level
a
current choice assignment

Figure 4.8: Implication graph after choosing a = true

$$\begin{array}{rcl}
b & \leftarrow & a,\ x \\
c & \leftarrow & w,\ x,\ \mathit{not}\ a \\
c & \leftarrow & h,\ \mathit{not}\ b \\
c & \leftarrow & y,\ z \\
d & \leftarrow & \mathit{not}\ c,\ \mathit{not}\ e \\
f & \leftarrow & h,\ \mathit{not}\ z \\
f & \leftarrow & u,\ \mathit{not}\ h \\
u & \leftarrow & v,\ w \\
v & \leftarrow & d \\
v & \leftarrow & \mathit{not}\ e \\
v & \leftarrow & f \\
w & \leftarrow & c,\ h \\
w & \leftarrow & \mathit{not}\ d
\end{array}$$

Assume that the above rules are the only rules in P with c, f, u, v, or w in their heads. Suppose also that the current partial assignment at this point in the search includes the literals u (i.e., $u = \mathit{true}$), v, w, x, $\neg y$ (i.e., $y = \mathit{false}$), and z. Assume that the current partial assignment has already been closed under Smodels$_{cc}$'s inference rules, and that the search heuristic has selected $a = \mathit{true}$ as the next choice assignment. This situation is portrayed in Figure 4.8.

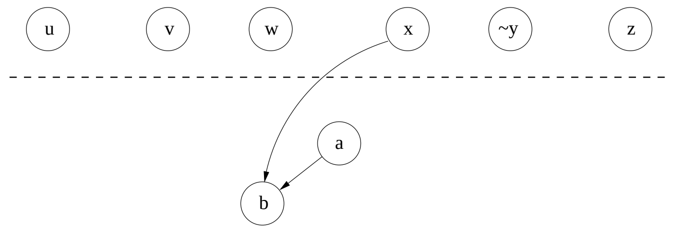

Figure 4.9: Implication graph after inferring b by Modus Ponens

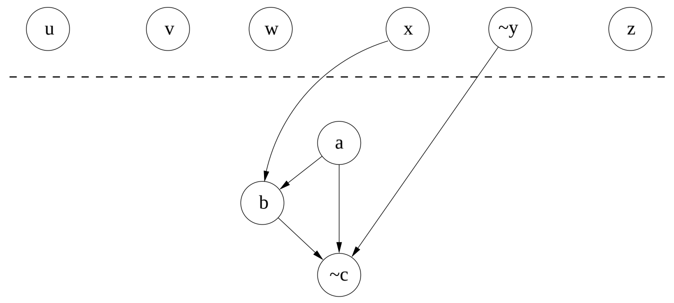

Figure 4.10: Implication graph after inferring $\neg c$ from All Rules Cancelled

Because P contains the rule $b \leftarrow a, x$, the search engine can infer b by the Modus Ponens rule. So it will add a node labeled b to the implication graph and add edges to b from each of a and x, producing the graph in Figure 4.9.

At this point, each rule with c in its head has at least one subgoal in its body cancelled. So the search engine may infer $\neg c$ based on All Rules Cancelled, and add edges as shown in Figure 4.10.

Since w in is the head of only two rules, and one of those rules is now cancelled, the body of the other rule must be satisfied. Thus the search engine can infer $\neg d$ from the Backchain True inference rule, based on the values of w and c (Figure 4.11).

Since c and d are *false*, then, based on the program rule $d \leftarrow \mathit{not}\ c, \mathit{not}\ e$ and the Backchain False inference rule, e may be inferred *true* (Figure 4.12).

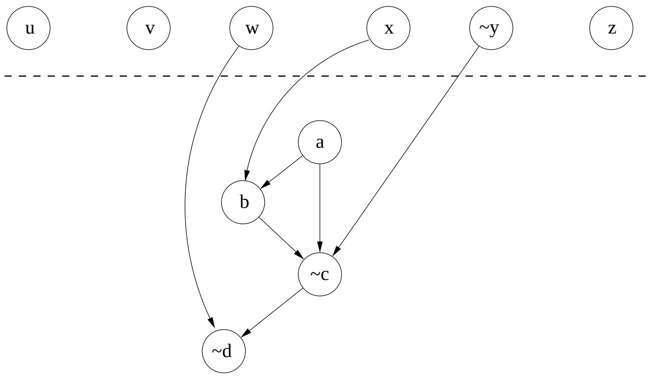

Figure 4.11: Implication graph after inferring $\neg d$ from Backchain True

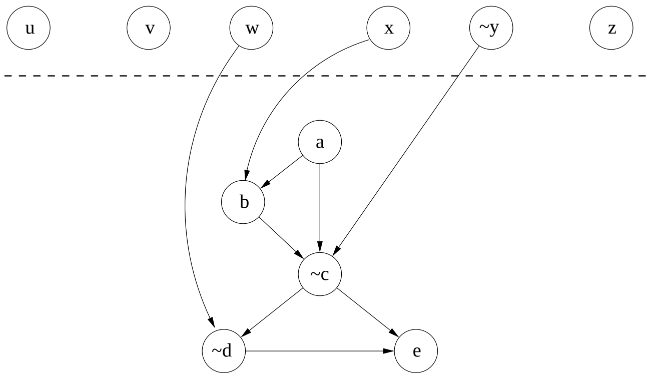

Figure 4.12: Implication graph after inferring e from Backchain False

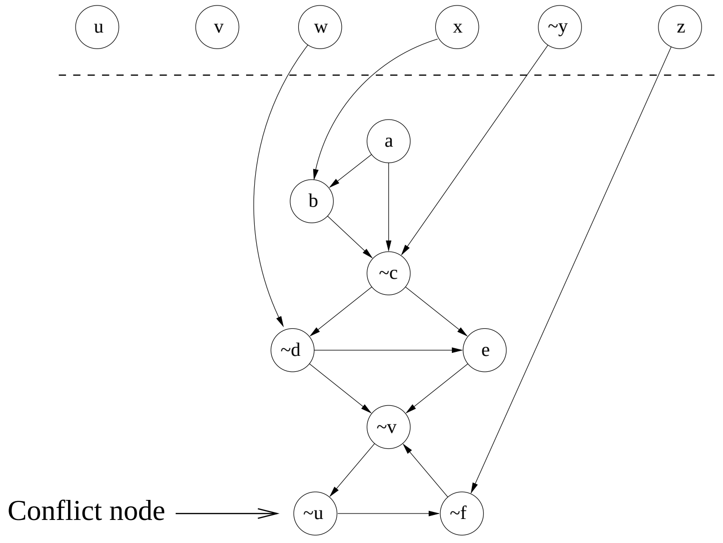

Figure 4.13: Implication graph after unfounded set detection

Next, we assumed that the following were the only rules in P with f, u, or v in their heads:

$$\begin{array}{lcl} f & \leftarrow & h,\ \mathit{not}\ z \\ f & \leftarrow & u,\ \mathit{not}\ h \\ u & \leftarrow & v,\ w \\ v & \leftarrow & d \\ v & \leftarrow & \mathit{not}\ e \\ v & \leftarrow & f \end{array}$$

Therefore, since $\neg d$, e, and z are in the current partial assignment, $\{f,\ u,\ v\}$ is an unfounded set. So we may infer $\neg f$, $\neg u$, $\neg v$ and add the implication graph edges shown in Figure 4.13: one edge for every rule with f, u, or v in its head.

At this point there are two pairs of conflicting nodes in the implication graph: $\{u, \neg u\}$ and $\{v, \neg v\}$. For this example, we will assume that Smodels$_{cc}$ first detects the conflict between u and $\neg u$, and tentatively selects it as the pair of conflicting nodes on which it will base its conflict diagnosis. As in GRASP, the element from this pair that was inferred later is labeled the Conflict Node, as shown in Figure 4.13. As we will see in Section 4.4.2, Smodels$_{cc}$ may need to adjust its conflict node selection once the diagnosis process begins. We postpone adding a conflict edge until this adjustment has been accomplished.

By inspection of this graph, it is easy to see that the combination of assignments $u = \mathit{true}$, $w = \mathit{true}$, $x = \mathit{true}$, $y = \mathit{false}$, $z = \mathit{true}$, and $a = \mathit{true}$ led to a conflict. From this we can infer the clause $\neg u \vee \neg w \vee \neg x \vee y \vee \neg z \vee \neg a$. In Sections 4.4.2 and 4.5 we will see how to obtain a shorter, and probably more useful, conflict clause from this graph.

4.3.4 Incorporating Extended Rules

As discussed in Section 3.2.2, Lparse translates extended logic programs into programs involving only (1) normal rules, (2) choice rules, (3) cardinality rules, and (4) weight rules. We refer to the latter three types of rules as the *basic extended rules.* Section 3.2.2 discussed how the Smodels inference rules work on the basic extended rules. In Section 4.3.2 we discussed how Smodels$_{cc}$ constructs the implication graph edges based on inferences made from normal rules. In the present section, we will do the same thing for the extended rules.

Choice Rules

Recall that a choice rule is one of the form:

$$\{a_1, \ldots, a_j\} \leftarrow b_1, \ldots, b_k, \mathit{not}\ c_1, \ldots, \mathit{not}\ c_m.$$

The Modus Ponens and Backchain False rules are not applicable to choice rules. The All Rules Cancelled, Backchain True, and Unfounded Sets rules work exactly the same as for normal logic programming rules, and the corresponding implication graph edges are constructed in the same way.

Cardinality Rules

A cardinality rule is one of the form:

$$R \quad = \quad a \leftarrow n\,[b_1, \ldots, b_k, not\ c_1, \ldots, not\ c_m]$$

where the lower bound n is a non-negative integer.

When a is inferred from such a rule by Modus Ponens, Smodels$_{cc}$ scans across the body of the rule in search of exactly n subgoals that are satisfied by the partial assignment that immediately preceded the new inference. Once these n subgoals have been found, an edge from each of the satisfying literals to the literal a is added to the implication graph.

The All Rules Cancelled and Unfounded Set inference rules work similarly for cardinality rules as for normal rules except as follows. A normal rule is cancelled whenever a single literal from its body is falsified (or when a single positive atom from its body is rendered unfounded). On the other hand, as discussed previously in Section 3.2.2, the above cardinality rule is cancelled only after $k+m-n+1$ subgoals from the body have been cancelled (including positive subgoals mentioning atoms in unfounded sets). Thus, if a has been inferred *false* by either the All Rules Cancelled or the Unfounded Set inference rule, then if R is a rule of the form shown above, with a at its head, then $k+m-n+1$ implication graph edges are added corresponding to the cancelling of this rule: one corresponding to each of the $k+m-n+1$ assignments that cancelled subgoals from the body of R.

The Backchain True inference rule is applied to R when all of the following three conditions are satisfied: (1) the atom a at the head of the rule is *true*, (2) all of the rules other than R that have a in their heads have been cancelled, and (3) $k + m - n$ subgoals from the body of the rule have been cancelled. When these three conditions are met, the remaining n subgoals from the body of the rule are all forced to be true. So to each of the inferred literals, we add the following implication graph edges: (1) an edge from the literal a, (2) for each cancelled rule R' with a in its head, an edge from each of the causes of the cancellation of R', and (3) an edge from each of the $k + m - n$ assignments cancelling subgoals from the body of R.

The Backchain False inference rule will be applied to R when (1) a is false, and (2) $n-1$ subgoals from the body of R have been satisfied. In this case, the uninstantiated subgoals from the body of R are set to *false*, and n incoming edges are added to each newly inferred literal: (1) one edge from $\neg a$, and (2) $n-1$ edges from the assignments satisfying subgoals in the body of R.

Weight Rules

A weight rule is a rule of the form:

$$R \quad = \quad a \leftarrow n\,[b_1 = w_{b_1}, \ldots, b_k = w_{b_k}, not\ c_1 = w_{c_1}, \ldots, not\ c_m = w_{c_m}]$$

where n, as well as each of the weights, is a non-negative integer.

Weight rules currently are not implemented in Smodels$_{cc}$, due to some problems that Lparse seems to have in translating rules into this form. However, we describe here an algorithm that could be used when weight rules are added to the Smodels$_{cc}$ implementation. As with normal rules, choice rules, and cardinality rules, the question is how to construct the implication graph edges corresponding to an application of an inference rule. The approach described below is quite similar to the approach given above for cardinality rules.

If a is about to be inferred from R by Modus Ponens, Smodels$_{cc}$ will scan the body of the rule in search of satisfied subgoals whose combined weight is at least n. Once these subgoals have been found, an edge is added from each of the corresponding satisfying assignments to the literal a.

The remaining relevant rules of inference usually also involve consideration of when a rule is *cancelled*. As discussed in the subsection on weight rules in Section 3.2.2, let $w = \sum_{i=1}^{k} w_{b_i} + \sum_{i=1}^{m} w_{c_i} - n$. Then w is the cancelling threshold for R: if the sum of the weights of the cancelled subgoals in the body of R exceed w, then R cannot fire.

The All Rules Cancelled and Unfounded Set inference rules work identically for weight rules as for cardinality rules except for the above adjustment to the definition of when a rule is cancelled.

Next, we review when the Backchain True inference rule is applied to a weight rule. Suppose that d is the combined weight of the cancelled subgoals from the body of R. Suppose also that x is an uninstantiated subgoal of maximal weight from the body of R. Then the Backchain True inference rule can be applied to R to infer that x is *true* if and only if (1) the atom a is *true*, (2) all of the rules other than R that have a in their heads have been cancelled, (3) $d \leq w$, and (4) $d + w_x > w$.

In the event that such an inference is made, search the body of R for a set S of cancelled subgoals whose combined weight is greater than $w - w_x$. Then, to the inferred literal x, add the following implication graph edges: (1) an edge from the literal a, (2) for each cancelled rule R' with a in its head, an edge from each of the causes of the cancellation of R', and (3) an edge from $\neg y$ for each element $y \in S$.

Next, we review how the Backchain False inference rule is applied to a weight rule. Suppose that d is the combined weight of the *satisfied* subgoals from the body of R. Suppose that x is an uninstantiated subgoal of maximal weight from the body of R. The Backchain False inference rule can be applied to R to infer $\neg x$ if and only if (1) a is false, (2) $d < n$, and (3) $d + w_x \geq n$.

In the event that such an inference is about to be made, search the body of R for a set S of satisfied subgoals whose combined weight is at least $n - w_x$. Then, to the inferred literal x, add the following implication graph edges: (1) an edge from the literal $\neg a$, and (2) an edge from y for each element $y \in S$.

4.4 Unique Implication Points

Once a conflict has been found and an implication graph has been constructed, the next step in producing a conflict clause is to identify *Unique Implication Points* (UIPs) in the implication graph. The notion of a UIP was defined by Marques-Silva and Sakallah in their work on the GRASP SAT solver [44].

Recall the idea of a *choice assignment* introduced in Section 3.1. A choice assignment is a guessed assignment that was returned by the search heuristic. If a node in an implication graph corresponds to a choice assignment, then we call that node a *choice node*. Choice nodes are distinguished by the fact that they do not have any incoming edges in the IG.

The only level of the search that would not have a choice node is level 0, which contains inferences that did not involve a choice assignment as a precondition. For each $n > 0$, there is exactly one choice node C_n at level n, and for every node N at level n in the IG there is a path from C_n to N.

Definition 16 (Unique Implication Point) *Let G be an implication graph containing a conflict node X at the current level of the search. Let C denote the choice node at the current search level. Then a node N at the current search level is a* Unique Implication Point *relative to X if $N \neq X$ and every path from C to X includes N.*

Note that the choice node C in Definition 16 is always a UIP. (A choice node may be part of a conflict pair, but it is never selected as a conflict node since the conflict node is always the element from a conflict pair that was inferred later, and a choice assignment would never be selected that conflicts with a current assignment.)

We will describe in Section 4.5 a precise algorithm for generating a conflict clause once the relevant UIP(s) have been found. For now, however, we can get an idea of the usefulness of UIPs by looking at the SAT-based implication graph example that we constructed in Section 4.3.1 (Figure 4.6). We display that implication graph again here in Figure 4.14.

In that example, s is the choice node at the current search level. We observed that no solution to the given SAT problem could simultaneously set $a = \textit{true}$, $b = \textit{false}$, $e = \textit{false}$, $f = \textit{true}$, and $s = \textit{true}$ since these assignments would lead to the conflicting inferences z and $\neg z$. This conflict diagnosis corresponds to making a backwards traversal in the implication graph from the conflict node $\neg z$. The traversal terminates at the choice node s and at nodes that occur at previous search levels. The corresponding conflict clause, $\neg a \vee b \vee e \vee \neg f \vee \neg s$, contains five literals.

However, there are two other UIPs in this particular graph besides the choice node s: nodes v and w.[16] Consider what would have happened if we had taken $w = \textit{true}$ as our choice assignment at the current level of the search. The inferences that occurred

[16]So unique implication points are not actually "unique".

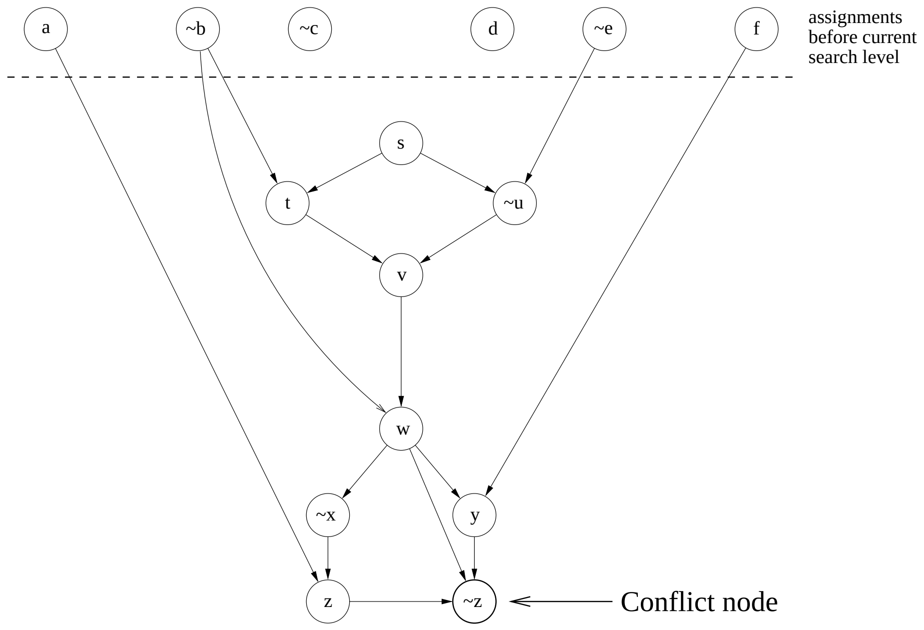

Figure 4.14: SAT implication graph example (Figure 4.6) revisited

below w in Figure 4.14 would still be in effect, and so we would have obtained the same conflict, z and $\neg z$. However, the conflict diagnosis that we would have obtained from our backwards traversal in this case would have involved only three assignments: $a = true$, $f = true$, and $w = true$. From this diagnosis we would have obtained the shorter conflict clause $C_2 = \neg a \vee \neg f \vee \neg w$.

The utilization of UIPs to create the conflict clauses represents one of the key differences between the rel_sat and GRASP SAT solvers. In rel_sat, which was also one of the very earliest SAT solvers to successfully incorporate conflict clauses, exactly one conflict clause was produced for every conflict and that clause always mentioned the most recent choice atom. From the implication graph shown in Figure 4.5, rel_sat would have produced only the conflict clause $C_1 = \neg a \vee b \vee e \vee \neg f \vee \neg s$.

4.4.1 UIP-based Clause Generation Strategies

Marques-Silva and Sakallah comment in [44] that GRASP may be configured to use a variety of clause generation strategies. However, it is typically configured so that, from a single conflict, it generates as many clauses as there are UIPs in the implication graph. From the IG in Figure 4.14, GRASP would normally generate the clause $C_2 = \neg a \vee \neg f \vee \neg w$ mentioned earlier. The clause results from the relationship of the conflict nodes z and $\neg z$ to the closest UIP w. Typically, GRASP would also generate the clause $\neg v \vee b \vee w$, which relates the UIP v to the UIP w, and the clause $\neg s \vee b \vee e \vee v$, which relates UIP s to UIP v.

In [64], members of the Chaff project performed extensive experiments on a number of different strategies for generating conflict clauses in a SAT solver, including the strategies incorporated by rel_sat and GRASP. The strategy that they found to be most effective overall (in terms of runtimes) is what they refer to as the *1-UIP* strategy. This strategy generates only one conflict clause per conflict in the search: the conflict clause that corresponds to the UIP that is closest to the detected conflict. In the example considered above, the 1-UIP strategy would generate only the conflict clause $C_2 = \neg a \vee \neg f \vee \neg w$.

Beame, Kautz, and Sabharwal [4] report that GRASP generally has fewer choice points in its search tree than does Chaff. They suggest that this is due to the additional clauses that GRASP learns from each conflict. However, they also state that GRASP may run slower than Chaff because of the extra time consumed in learning these additional clauses, and because of the time involved in using these clauses in Boolean Constraint Propagation.

Based on the results of the study in [64], and Chaff's excellent reputation as a current, high-performance SAT solver, we chose to use the *1-UIP* strategy in Smodels$_{cc}$. Therefore, given that Smodels$_{cc}$ has detected a conflict in the search, it needs to be able to compute the corresponding UIP that is "closest" in the implication graph to the conflict.

Before we give an algorithm to compute the closest UIP, we give a precise definition of what it is and prove that the definition is reasonable even in graphs with cycles.

Definition 17 (Closest UIP) *Let G be an implication graph and X a conflict node at the current search level. Let C be the choice node at the current search level, and let U be the set of UIPs relative to X. Let P be any acyclic path from C to X. Then the* closest UIP *(CUIP) relative to X is the last element of U on this path.*

That at least one element of U will exist on the path P is clear because the choice node C is necessarily a UIP.

Proposition 2 (CUIP Well-defined) *The* CUIP relative to X *is well-defined by Definition 17. That is to say, regardless of which acyclic path P is chosen from C to X, the node that we obtain as the CUIP will be the same.*

Proof: From the definition of a UIP, it is immediate that every UIP is present on every path from the choice node to the conflict node. So, to prove the result, it suffices to show that, given any two acyclic paths P and P' from C to X, and U_1 and U_2 any two UIPs relative to X, then if U_1 occurs before U_2 on P, then U_1 also occurs before U_2 on P'.

Suppose not. Assume instead that U_1 precedes U_2 on path P, and U_2 precedes U_1 on path P'. By splicing the part of path P that goes from the choice node to U_1

with the part of path P' that goes from U_1 to X, we can then obtain a path that goes from the choice node to X but which excludes U_2. (The new path excludes U_2 because P and P' were assumed to be acyclic.) This contradicts the assumption that U_2 is a UIP. $\square$

Note in Definition 17 that the CUIP is defined *relative to the conflict node X*. From Proposition 2, every conflict node has a unique corresponding CUIP. However, we observe that distinct conflict nodes may have distinct CUIPs. For instance, in Figure 4.13 there are two distinct conflict nodes: $\neg v$ and $\neg u$. The CUIP corresponding to $\neg u$ is $\neg v$. The CUIP corresponding to $\neg v$ is $\neg c$.

4.4.2 Finding UIPs

Papers on GRASP [44] and Chaff [47] do not specify the algorithm(s) used to compute UIPs, other than to say that their programs can "identify UIPs in linear time with one traversal of the implication graph."[44] In Smodels$_{cc}$ our algorithm for finding UIPs needs to work in graphs that may contain cycles, unlike the situation with SAT solvers.

Tarjan [60] gives an algorithm that can be used for finding the closest UIP of each node that is reachable from the latest choice point.[17] So we could use Tarjan's algorithm here. However, since we need to compute the closest UIP only for the selected conflict node, it is simpler and more efficient to use the algorithm that we give below.[18]

[17]Tarjan uses the term *immediate dominator* rather than the term "closest UIP".

[18]Tarjan's paper solves a different problem because his motivating application domain, code optimization, was different from the one that this dissertation considers.

Indeed, we can solve this problem in linear time, even though our implication graphs may contain cycles. Specifically, our algorithm for finding UIPs, and identifying the closest UIP, has a runtime that is linear in the number of edges that are incident to the nodes at the current level of the search that in turn have paths to the conflict node.

The algorithm takes as its parameters an implication graph G with a specified choice node C and a specified conflict node X. It is assumed that all of the nodes in G are marked with their search levels. It is also assumed that C and X are at the current level of the search, and that there is a path from C to X consisting only of nodes from the current search level. The steps of the algorithm are as follows:

Step 1. Compute $Path_G$. $Path_G$ is an arbitrary acyclic path from C to X. We denote the length of $Path_G$ (i.e., the number of nodes in the path) by the expression $|Path_G|$. $Path_G$ can be computed by performing a depth first search.

In our example ASP-based implication graph, there are several possible paths from the choice node a to the conflict node $\neg u$. Suppose that the path that our depth first search returned is $Path_G = a, b, \neg c, \neg d, \neg v, \neg u$ (see Figure 4.15).

Step 2. Adjust conflict node selection. Determine the first node X' on $Path_G$ whose assignment conflicts with an assignment that was made earlier in the search. If X' differs from the initial conflict node X, then make X' the new conflict node, and truncate $Path_G$ so that it terminates at X'.

The reason that the issue of adjusting the conflict node comes up is that the Unfounded Set inference rule may introduce multiple conflict nodes simultaneously. (As noted earlier, this will never occur in a DPLL SAT solver.) To see why it is important to make the adjustment in Step 2, consider our ongoing example in Figure 4.15. If we

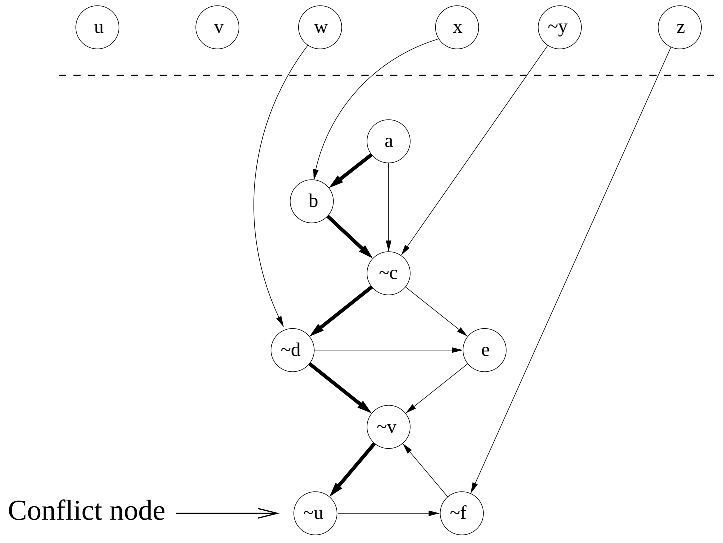

Figure 4.15: A path from the choice node to a conflict node

base our analysis on $\neg u$ as our conflict node, then (after adding a conflict edge from u to $\neg u$) we will have $\neg v$ as our closest UIP. From this UIP, our conflict diagnosis will be that we cannot simultaneously have $u = \mathit{true}$ and $v = \mathit{false}$. From this diagnosis, we would generate the conflict clause $\neg u \vee v$. This is a sound clause in the sense that any model of P must satisfy this clause. However, this conflict diagnosis is not as useful as we would like: Because this clause mentions v positively, it is actually satisfied by the partial assignment that led up to the conflict. Thus, by learning this clause, the solver has not learned anything that would have prevented the conflict which is being diagnosed. Because Smodels$_{cc}$ uses nonchronological backtracking, it could actually go into an infinite loop if it generated such conflict clauses: Potentially, it could backjump and then follow exactly the same search path as before.

However, by adjusting the conflict node (and $Path_G$) as described above, we remedy this problem: As a result of the adjustment, it is guaranteed that all of the literals in the conflict clause except one will disagree with the partial assignment that

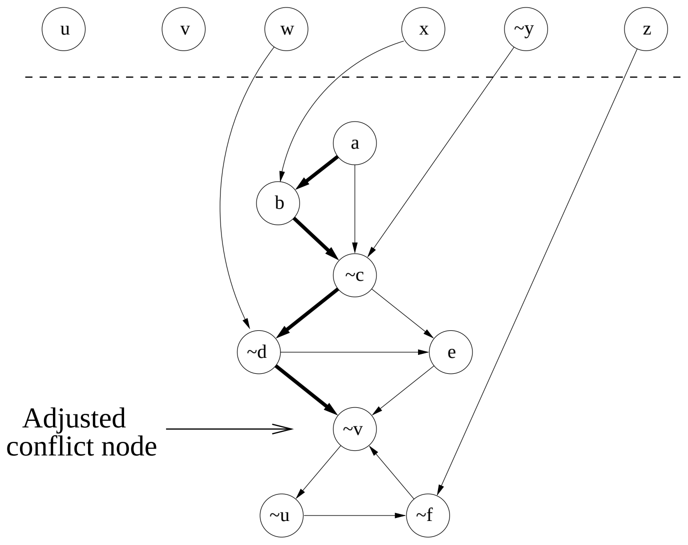

Figure 4.16: Adjusting the conflict node and corresponding path

was in effect immediately before the current search level was reached. The remaining literal in the new conflict clause will correspond to the complement of the closest UIP. Since the CUIP is guaranteed to be on $Path_G$, and is not a conflict node, we are assured that the corresponding atom was not instantiated before the current search level. Thus we know that, after backjumping, the algorithm cannot repeat the same search path (sequence of choice assignments and inferences) that led to the current conflict.

A formal proof that Smodels$_{cc}$ always terminates is given in Appendix D. The fact that the CUIP is not a conflict node is crucial to that proof.

In our example (Figure 4.15), we will make $\neg v$ the new conflict node, and shorten $Path_G$ to a, b, $\neg c$, $\neg d$, $\neg v$ (see Figure 4.16).

Step 3. Create an additional edge to the adjusted conflict node. As in GRASP, we add a *conflict edge* that goes from the complement of the conflict node to the conflict node. In our example, this edge goes from v to $\neg v$ (see Figure 4.17). In this particular

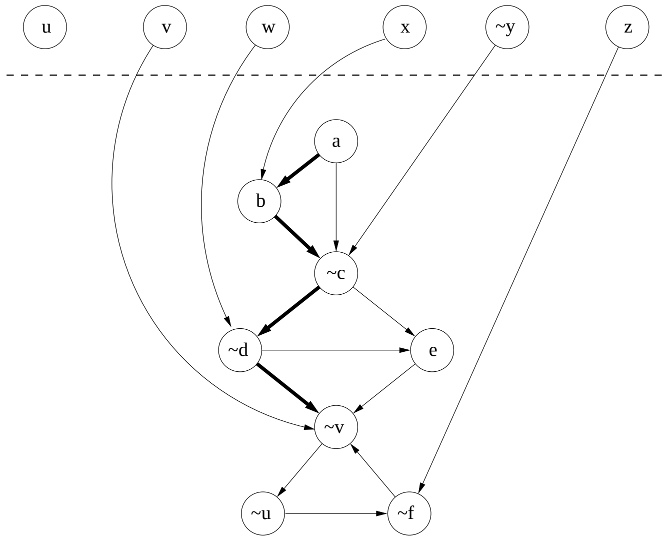

Figure 4.17: Adding a conflict edge to the adjusted conflict node

case, the conflict node represents the fact that the combination of assignments v, $\neg d$, and e result in a conflict.

We will now motivate Step 4 in our algorithm for finding the CUIP. We denote the final, adjusted conflict node by X'. Note that any UIP relative to X' must be a node preceding X' on $Path_G$. Further note that any node M preceding X' on $Path_G$ is *not* a UIP if, and only if, there are nodes N and N', also belonging to $Path_G$, such that N occurs earlier in $Path_G$ than M, N' occurs later in $Path_G$ than M, and there is a path R from N to N' such that no intermediate nodes in R appear in $Path_G$. In this case, we say that path R "skips" the node M. This brings us to our next definition.

Definition 18 (Skip-to-depth) *Suppose that G is an implication graph, with choice node C and conflict node X' both at the current search level. Suppose that $Path_G = [C = x_1, \ldots, x_n = X']$ is an acyclic path in G from C to X'. Then the* skip-to-depth *(relative to $Path_G$) of a node N at the current search level is the highest index i such that there exists a path R from N to x_i where at most the first node and the last node of R may belong to $Path_G$.*

We will denote the skip-to-depth of node N by $StD(N)$. If there is no path from N to any node in $Path_G$, then $StD(N)$ is undefined. In the following, we denote each x_i in $Path_G$ by $Path_G(i)$. We also denote the length of $Path_G$, n, by $|Path_G|$.

Step 4. Compute skip-to-depths. We compute the skip-to-depths by performing multiple traversals backwards along the graph. However, the total process will run in linear time because no edge in the graph is traversed more than once. Each traversal starts at a distinct node $Path_G(i)$ with $i = |Path_G|$ downto 2. The traversal propagates the value i as the skip-to-depth of each node that it reaches, provided that that node has not already been assigned a (greater) skip-to-depth. The two routines involved in computing skip-to-depths are given in Table 4.1. They take as implicit parameters the implication graph G, the path $Path_G$, and the conflict node X'.

Table 4.2 gives the skip-to-depths of the nodes in Figure 4.17, relative to the indicated $Path_G$.

Step 5. Scan $Path_G$ for UIPs. Now we can easily determine which nodes in $Path_G$ are UIPs: The i-th node in $Path_G$ ($i < |Path_G|$) is a UIP iff each of the preceding nodes in $Path_G$ has a skip-to-depth less than or equal to i. The UIP that we want is the CUIP, the UIP that has the highest path index and which is therefore closest to the conflict node. The only two nodes that are UIPs (relative to $\neg v$) in our example are nodes a and $\neg c$. $\neg c$ is the CUIP since it has the higher index in $Path_G$. Table 4.3 gives pseudocode for finding the CUIP.

```
ComputeSkipToDepths()
{
    for each node $N$ at the current level of the search do
        initialize $StD(N) \leftarrow$ 'Uncomputed'
    $StD(X') \leftarrow |Path_G|$
    for $i \leftarrow |Path_G|$ downto 2 do
        $N \leftarrow Path_G(i)$
        for $M$ a predecessor of $N$ do
            MarkSkipToDepthsFromNode($M$, $i$)
}

MarkSkipToDepthsFromNode(Node $M$, int $i$)
{
    if ($M$ was inferred at the current search level
                and $StD(M)$ = 'Uncomputed') then
        $StD(M) \leftarrow$ i
        if ($M$ is not a member of $Path_G$) then
            for $M'$ a predecessor of $M$ do
                MarkSkipToDepthsFromNode($M'$, $i$)
}
```

Table 4.1: Computing skip-to-depths

node	path index	skip-to-depth	Is UIP?
a	1	3	Y
b	2	3	N
$\neg c$	3	5	Y
$\neg d$	4	5	N
$\neg v$	5	5	N
e	-	5	N
$\neg f$	-	5	N
$\neg u$	-	5	N

Table 4.2: Skip-to-depths of nodes from example implication graph

```
ScanForCUIP()
{
    $ClosestUIP \leftarrow Path_G(1)$ // The first UIP in the path is
                                      // always the choice node
    $maxStD \leftarrow StD(Path_G(1))$
    for $i \leftarrow 2$ to $|Path_G| - 1$ do
        if $(maxStD \leq i)$ then
            $ClosestUIP \leftarrow Path_G(i)$
        $maxStD \leftarrow max(maxStD, StD(Path_G(i)))$
    return $ClosestUIP$
}
```

Table 4.3: Finding the closest UIP

4.5 Generating the Conflict Clause

Once the closest UIP has been found, Smodels$_{cc}$ constructs the conflict clause by using the same method as GRASP's.

Definition 19 (Predecessor) *We say that node u is a* predecessor *of node v in G if the edge $u \rightarrow v$ exists in G.*

Definition 20 (Choice Node) *A* choice node *is a node with no predecessors.*

Definition 21 (Dominate) *A set D* dominates *a node x in an implication graph if every path from a choice node to x includes an element of D.*

In the above definition, a path consisting of a single node is allowed. So, if D dominates x and x is itself a choice node, then we may conclude $x \in D$.

The algorithm for generating the conflict clause computes a set of nodes that dominate the chosen pair of conflicting nodes. (See Table 4.4.) This is accomplished by traversing the implication graph backwards from the conflict node. The terminal

```
ComputeConflictClause()
{
        // Precondition: $\forall$ node $N \in G$, $N.IsMarked = false$
        initialize $D \leftarrow \emptyset$
        ComputeDominatingSet($ConflictNode$,$D$)
        $ConflictClause \leftarrow \{\neg x : x \in D\}$
        Restore precondition
        return $ConflictClause$
}

ComputeDominatingSet($Node$, $D$)
// $D$ is a set of nodes.
// Adds to $D$ a set of nodes sufficient to ensure that $D$ dominates $N$.
// The only node from the current decision level that may be
// placed in $D$ is the closest UIP.
{
        if $Node.IsMarked$ then
            return
        $Node.IsMarked \leftarrow true$
        if $Node.DecisionLevel < CurrentDecisionLevel$ or $Node = ClosestUIP$ then
            $D \leftarrow D \cup \{Node\}$
        else
            for $Node'$ a predecessor of $Node$ do
                ComputeDominatingSet($Node'$, $D$)
}
```

Table 4.4: Computing a conflict clause

nodes of this traversal are the closest UIP plus any nodes that are reached that are at earlier search levels. These terminal nodes constitute a conflict diagnosis D. The complements of the literals in D then constitute the conflict clause.

For the implication graph given in Table 4.17, the dominating set D is $\{v, w, z, \neg c\}$. The corresponding conflict clause is $\neg v \vee \neg w \vee \neg z \vee c$.

A key observation used in the proof of correctness (Appendix C) of the Smodels$_{cc}$ algorithm is that the elements of D do in fact dominate the chosen pair of conflicting nodes. An important fact used in the proof of completeness (Appendix D) is that only one node from the current search level is contained in D: all nodes in D other than the closest UIP come from earlier search levels.

4.6 Using Conflict Clauses in the Search

Once a conflict clause has been generated, there are three ways in which it is used to increase the performance of the search engine: (1) to direct backjumping, (2) to serve as an additional constraint, and (3) to guide the search heuristic. We adapted techniques from the GRASP, Chaff, and BerkMin satisfiability solvers to accomplish these tasks in Smodels$_{cc}$, as we detail in this section.

4.6.1 Backjumping

As we saw from the example in Section 4.1 (Figure 4.3), backjumping (nonchronological backtracking) involves potentially removing several search levels from the search engine's stack when backtracking from a single conflict. In the example given there, the search level at which the conflict occurred was level 100, and the literals appearing in the conflict diagnosis were at levels 2, 3, and 100 of the search. One fact that was true about this diagnosis is also true about all of the diagnoses returned by the 1-UIP conflict clause generation scheme used by Chaff and by Smodels$_{cc}$: exactly one literal in the diagnosis comes from the highest search level (i.e., the level at which the conflict occurred). Under the 1-UIP scheme, this literal will be the one corresponding to the closest UIP. The remaining literals in the diagnosis will always come from earlier levels in the search. The search level to which the solver will backtrack will be the highest search level, other than the current search level, which is involved in the conflict diagnosis. After backjumping to that level, the search engine can then immediately infer the complement of the UIP literal, based on the conflict clause.

4.6.2 Serving as Additional Constraints

Once the conflict clause has been generated, the search engine can store it for future reference. One of the main uses of stored conflict clauses is that they can serve as constraints from which new inferences may be derived via the unit clause rule. For instance, suppose that we have stored the conflict clause $C = a \vee b \vee \neg c \vee d \vee \neg e$ and that later in the search the current partial assignment includes $a = \mathit{false}$, $b = \mathit{false}$, $c = \mathit{true}$, and $e = \mathit{true}$. Then C has been reduced to the unit literal d, and we may infer $d = \mathit{true}$. (Furthermore, edges from $\neg a$, $\neg b$, c, and e to d will be added to the Smodels$_{cc}$ implication graph.)

Optimized BCP

The process of obtaining inferences from the unit clause rule is also known as Boolean Constraint Propagation (BCP). The BCP process seems to be the most time-consuming aspect of using conflict clauses in Smodels$_{cc}$. In order to make this process as efficient as possible, Smodels$_{cc}$ incorporates the "Optimized BCP" technique that is used in Chaff and explained by Moskewicz *et al.* in [47]. We outline this technique in the remainder of this section.

An obvious approach to implementing the unit clause rule/BCP for a set of clauses would be to keep a count, for each clause C, of the number of literals in it that have not been contradicted. Once this count goes down to 1, then it would be time to set the remaining literal in the clause to *true*. However, this approach can be unnecessarily time consuming because it involves updating the count at least $N - 1$ times before an inference is obtained. Perhaps even worse, this technique requires that the value of the count is updated when the search engine backtracks.

A better situation would be if the engine only had to visit the clause when the count is updated from the value 2 to the value 1, because that is the only time at which we potentially obtain an inference from the clause. Chaff's Optimized BCP process attempts to approximate this ideal by choosing two *watched literals* from the clause. For instance, if the clause were $C = a \vee b \vee \neg c \vee d \vee \neg e$ as mentioned above, then the watched literals might be a and $\neg e$. The watched literals are chosen so that either (1) one of the watched literals is satisfied, or (2) neither of the watched literals is contradicted. This state of affairs is maintained as an invariant throughout the life of the clause.

Whenever one of the two watched literals (call it $Watched_1$) is contradicted, then the clause is visited. If the other watched literal ($Watched_2$) is already satisfied, then nothing needs to be done. If, however, the atom of the other watched literal is merely uninstantiated, then the clause is searched for another non-contradicted literal to become the new $Watched_1$. If the search fails to find a candidate to be the new $Watched_1$ then the clause is unit, and $Watched_2$ is forced to true by the unit clause rule.

An advantage of Chaff's Optimized BCP is that no clauses need to be visited or updated when the search engine uninstantiates atoms during backtracking.

In [42], Lynce and Marques-Silva evaluated the performance of ten different BCP schemes, including several of the most recent ones. They implemented each of the schemes in their SAT solver (JQUEST) in order to normalize the results, and tested on a wide range of SAT benchmarks. Their empirical results showed Chaff's optimized

BCP procedure, with its watched literals, to be among the fastest, perhaps the fastest, overall of all of the approaches that they tested. Based on these findings, we have adopted Chaff's approach as the BCP procedure in Smodels$_{cc}$.

4.6.3 Search Heuristics

Section 3.2.3 discussed the lookahead-based search heuristic of the original Smodels program. The current search heuristic of Smodels$_{cc}$ is modeled after that of the BerkMin SAT solver [27], which itself extends and refines ideas from Chaff [47]. In this section we will review the search heuristics of Chaff and BerkMin, and discuss how BerkMin's heuristic is adapted to Smodels$_{cc}$.

Chaff's Search Heuristic

The Chaff developers tried a number of previously published search heuristics in their solver. However, they reported that they obtained considerably better results on their test cases with a new heuristic they designed called the Variable State Independent Decaying Sum (VSIDS) heuristic [47]. In order to compute this heuristic, Chaff keeps a separate counter for each literal. That is, for each atom in the problem, a counter is kept for the positive literal mentioning that atom, and also for the negative literal. Prior to reading in the clauses that represent the problem, each of these counters is initialized to zero. When a new clause is stored (either a clause from the original problem representation or a conflict clause), the counter associated with each literal mentioned in the clause is incremented by one. At each choice point in the search, the literal with the highest count is chosen and that literal is forced to true. Ties are broken randomly. Periodically, the counters for each of the literals is divided by some small constant greater than one.

In order to speed up the computation of the heuristic, Chaff does not compare the counts every time that its search heuristic routine is called. Instead, the search heuristic periodically calls a sorting routine from the C++ Standard Template Library to sort the literals into an STL set, ordering the literals based on their counts. During subsequent calls to the search heuristic routine, uninstantiated literals are chosen from this set, with higher scoring literals being chosen first, until the next sorting operation is performed.

Difficult problems will generally result in a large number of conflicts during the search. Thus, on hard problems, Chaff's heuristic is driven primarily by the conflict clauses that have been generated. Periodically dividing the values of the counters aims the heuristic towards satisfying the more current conflict clauses, in particular.

BerkMin's Search Heuristic

In their paper on BerkMin [27], Goldberg and Novikov cite Chaff's aim of satisfying the most recent conflict clauses as in important step in the right direction. However, at least in their intended application domains (chiefly hardware verification), they advocate realizing this idea to an even greater degree. They argue that the search heuristic should be very *dynamic*. By this they mean that the weights assigned to the atoms in the problem should be able to change very quickly, because changing the value of even a single atom in the problem may greatly affect the search space that is being examined. An example that they give is that changing the value of a single wire in a processor from a 1 to a 0 may entirely change which part of the processor will be involved in a computation. They argue that it is important for the search heuristic to focus on the most recently created conflict clauses because these clauses will be most relevant to the part of the search space that is currently being explored.

BerkMin's search heuristic works as follows. The conflict clauses generated by BerkMin are arranged chronologically, based on the order in which they were created. When a choice point is reached, BerkMin selects the most recently generated conflict clause that is not satisfied by the current partial assignment, if such a clause exists. Call this clause C. (Shortly, we will discuss what BerkMin does when no such clause exists.) BerkMin selects as its choice atom, a, the atom in C that has the highest "activity count".

The activity count $ac(a)$ of an atom a is computed as follows. At the start of the search, $ac(a)$ is zero. Thereafter, whenever a is involved in causing a conflict, $ac(a)$ is incremented by one. "Causing a conflict" means either being mentioned in a conflict clause or else being mentioned in an implication graph node that is strictly between a conflict clause node[19] and a conflict node. For example, in the conflict depicted in Figure 4.14, where node w is the closest UIP, the atoms that would have their ac values incremented would be z, a, x, w, y, and f. Similar to Chaff, BerkMin periodically divides by a small constant greater than one the ac counts of all of the atoms in the problem. (This constant is also chosen to be greater than the constant that Chaff uses for the same purpose.)

Three of the more important differences of BerkMin from Chaff are that (1) atoms are chosen only from conflict clauses that are currently not satisfied, if such clauses exist; (2) priority is given even more aggressively to the most recent conflicts; and

[19] "Conflict clause node" means a node from the dominating set D in the ComputeConflictClause procedure. Recall that the literals in a conflict clause are the complements of the literals mentioned in D.

(3) the activity count of an atom is incremented whenever that atom is *involved* in generating a conflict, not only when it is mentioned in a conflict clause.

Once BerkMin has chosen the choice atom a, it must decide whether to instantiate a to *true* or to *false*. BerkMin determines this by checking whether a has appeared in more conflict clauses positively or negatively. If a has appeared in more conflict clauses as a positive literal, then a is instantiated to *true*. If it has appeared more often negatively, it is instantiated to *false*. In the case of a tie, the value is chosen randomly.

If all of the conflict clauses in the conflict clause store are satisfied by the current partial assignment, then BerkMin selects the next choice atom a so that the activity count $ac(a)$ is maximized. In this case, BerkMin also uses a different method to decide whether to instantiate a to *true* or to *false*. However, the technique that it uses to choose this value is based entirely on the set of binary clauses[20] that appear in the original CNF representation of the problem. Since the logic program presented to Smodels$_{cc}$ is not represented as a set of clauses, BerkMin's technique in this case does not seem to be relevant to Smodels$_{cc}$. (However, the interested reader may find the details of their approach in [27].)

Smodels$_{cc}$'s Search Heuristic

Smodels$_{cc}$ uses the same search heuristic strategy as BerkMin. The main difference from BerkMin is that, as noted above, in situations where all of the conflict clauses are currently satisfied, Smodels$_{cc}$ does not treat binary clauses from the original problem specification as special. Once Smodels$_{cc}$ has selected a choice atom a, the decision as

[20]A clause is said to be *binary* if it has exactly two literals.

to whether to instantiate a to *true* or to *false* is *always* based on which of the two literals, a or $\neg a$, has appeared in more conflict clauses.

[27] does not specify how BerkMin selects its choice atoms before its first search conflict. Before Smodels$_{cc}$ reaches its first conflict, it selects choice atoms randomly, and always assigns its choice atoms the initial value *false*. For future work, it might be useful to refine this aspect of Smodels$_{cc}$. Smodels$_{cc}$ does perform lookaheads before the first choice point, in order possibly to gain some initial inferences. It would be possible to initially score the atoms in problems based on the results of these lookaheads, perhaps following a scheme such as the one used by the original Smodels program (see Section 3.2.3).

4.6.4 Restarts

Restarting the search can be done in a solver whether or not conflict clauses are used, just as conflict clauses can be incorporated whether or not restarts are used. However, from a performance perspective restarts do seem to be very useful, particularly in solvers that use conflict clauses. Currently, it seems that the great majority of SAT solvers that incorporate conflict clauses (including GRASP, Chaff, and BerkMin) also incorporate restarts.

The idea of restarts is to periodically restart the DPLL search from the root node, but to retain in the subsequent search some of the results that were learned in the preceding iterations. In particular, in solvers incorporating conflict clauses, some of the conflict clauses resulting from the previous search iterations would be held over into the new search. Literals that were learned in the previous iterations would also be retained. In SAT solvers that incorporate conflict clauses, this would include assignments learned from any conflict clauses of size one, along with any assignments that were inferred from these assignments through the BCP process.

The fact that learned information is carried over to the new search generally forces the new search to be different from the previous search iterations. It also gives the search heuristics the opportunity to use the learned information to direct the early decisions in the new search tree.

Chaff and BerkMin each perform their first restart after a certain number of conflicts N have been reached. In the case of Chaff, a small constant, which we will call C, is used to increment the number of conflicts between restarts. For example, examination of the source code for the zChaff variant of Chaff (Version Z2001.2.17) revealed that it uses N = 40,000 and C = 100. This means that zChaff performs its first restart after the first 40,000 conflicts, its second restart after the next 40,100 conflicts, its third restart after the next 40,200 conflicts, etc. The "Spelt3" version of Chaff2 (which is roughly a contemporary of the above version of zChaff, but maintained by a different individual) uses N = 2,000 and C = 200, or else N = 10,000 and C = 1,000, depending on which of the accompanying configuration files is used. It was reported in [47] that using a positive value for C does in fact seem to provide some performance benefit.

According to [27], BerkMin does a restart after every 550 conflicts (i.e., $N = 550$, $C = 0$).

Smodels$_{cc}$ uses a restart frequency that is more or less intermediate between that of Chaff and and that of BerkMin. Specifically, the current version uses $N = 500$ and $C = 50$.

4.6.5 Clause Deletion

It is important that the search engine is able to delete old conflict clauses from its store. This is so that the store of conflict clauses does not exceed the supply of available system memory and so that the time required to perform the BCP operation, which derives new inferences from the conflict clauses, does not take an excessive amount of time. The main criterion is to delete clauses that seem unlikely to be useful for pruning the search space during future BCP operations.

Clause Deletion in Chaff

Chaff actually schedules in advance when a clause is available for deletion. At the time that a clause is created it is determined at what future point in the backtracking search, if any, the clause should be marked as available for deletion. The criterion used is that a clause should be so marked when it has at least N literals uninstantiated, where N is typically between 100 and 200 [47].

Chaff periodically pauses its search to mark as deleted all of the clauses that are currently available for deletion. The conflict clause store is then garbage collected to free up the unused space, and the remaining clauses are recompacted. The recompaction serves both to eliminate memory fragmentation problems, and to create better data locality during subsequent BCP.

Clause Deletion in BerkMin

BerkMin also has a periodic deletion/garbage-collection/recompaction phase that it performs on its store of conflict clauses. BerkMin performs this phase always, and only, at the beginning of a restart of the search (see Section 4.6.4).

First, BerkMin removes all clauses that are satisfied by assignments “learned” from the previous search iterations.

The remaining clauses that are removed by BerkMin are determined by a relatively complex heuristic that is explained in Section 5 of [27]. Basically, BerkMin keeps its store of conflict clauses arranged in a queue, with older clauses sitting toward the queue’s front. When a restart is done, BerkMin removes from the front 1/16th of the queue those clauses with more than 8 literals. From the remaining 15/16ths of the queue, all clauses containing more than 42 literals are removed. However, BerkMin always retains the most recently generated clause and “a fraction” of the more active clauses, regardless of the number of literals that they contain. The “activity count” of a clause is the number of conflicts in which it has been involved through the BCP process. Early in the search, clauses from the front 1/16th of the queue with activity counts greater than 60 are regarded as “active”. Clauses from the remaining 15/16ths of the queue with activity counts greater than 7 are also regarded as active. The threshold for whether a clause from the front 1/16th of the queue is regarded as active is gradually increased throughout the search.

During each clause deletion phase, BerkMin tries to eliminate at least 1/16th of the conflict clauses from the store. Towards this end, if a clause deletion phase ever deletes fewer than 1/16th of the conflict clauses, then the threshold value of 8 literals, used to determine which clauses from the front 1/16th of the queue are small enough to save, may be decremented, though it is never decremented below a value of 4.

In general, BerkMin will retain a significantly smaller store of conflict clauses than Chaff. Indeed, one of the advantages of BerkMin over Chaff cited in [27] is that the more aggressive clause deletion scheme makes BerkMin less likely to exceed system memory capacity.

Clause Deletion in Smodels$_{cc}$

The clause deletion scheme used by Smodels$_{cc}$ is roughly as simple as the one incorporated in Chaff. However, it utilizes BerkMin's idea of arranging the conflict clause store into a queue from which larger clauses towards the head/front are deleted. In terms of how large it allows the clause store to grow, it is designed to be intermediate between Chaff and BerkMin.

Smodels$_{cc}$ does not have a periodic deletion/garbage-collection/recompaction process. Instead, whenever a new conflict clause is added, Smodels$_{cc}$ checks whether any clauses should be deleted. It does no recompaction after clauses are deleted. Currently, Smodels$_{cc}$ retains (1) all clauses created since the most recent restart, (2) all of the 5000 most recently generated clauses, and (3) all clauses with no more than 50 literals. Any clause that does not meet at least one of these three criteria is deleted.

This scheme is easy to implement efficiently: A single pointer is maintained into the chronologically ordered queue of clauses. The pointer moves from the head of the queue (where the older clauses are) towards the tail. Each clause is visited by the pointer exactly one time during the entire search: that is, at the first moment when the clause no longer satisfies conditions (1) and (2) above. Then, if the clause has more than 50 literals, it is deleted. Otherwise, it is retained for the rest of the search.

Smodels$_{cc}$'s clause deletion scheme does allow the possibility of exceeding system memory constraints. However, we would point out that this is less likely with this scheme than with Chaff's scheme. Indeed, in our experiments so far, we have never seen Smodels$_{cc}$ abort due to lack of memory. However, this does not rule out the possibility of incorporating a more sophisticated scheme in the future.

Adopting a periodic deletion/garbage-collection/recompaction process, as in Chaff or BerkMin, probably would be a useful change in order to improve data locality. Also, Smodels$_{cc}$ does not automatically delete clauses that are satisfied by learned assignments, as BerkMin does. This would also be a reasonable optimization to incorporate.

4.7 Computing Multiple Answer Sets

Because the original Smodels uses chronological backtracking, its approach to computing multiple answer sets is rather straightforward, as described in Section 3.2.4.

The nonchronological backtracking in Smodels$_{cc}$ presents a complication in this regard: when the search engine backjumps from a newly computed answer set, we need to ensure that it does not subsequently recompute the same answer set. We accomplish this by causing Smodels$_{cc}$ to create a new clause which eliminates that answer set from the search space.

So suppose that Smodels$_{cc}$ has computed a new answer set M. In order to ensure that M is never recomputed it would be sufficient to create and store the clause

$$C_M = \bigvee\{\neg x : x \text{ is a literal satisfied by } M\}.$$

C_M mentions every atom contained in the original search problem. It would be beneficial to create a smaller clause than this, since a smaller clause would tend to provide higher backjumping and would be more useful in the BCP process. However, we want to be sure that the new clause does not eliminate any valid answer sets that have not yet been computed.

Consider the set of literals

$$S = \{x : x \text{ was a choice assignment in effect when } M \text{ was found}\}.$$

Let $\varphi = \bigwedge S$. Also, let ψ be the conjunction of the literals that Smodels$_{cc}$ has inferred at level zero of the search. Due to the soundness of Smodels$_{cc}$'s conflict clauses and inference rules (see Appendix C), it is clear that the only answer set of P that satisfies $\varphi \wedge \psi$ is M. The inferences contained in ψ are never removed by Smodels$_{cc}$. Thus, adding the constraint $\neg\varphi$ removes from the search space no answer sets other than M. Note that $\neg\varphi$ may be expressed as a single clause (call it C_M^*) that subsumes C_M. Thus by adding C_M^* as a new constraint, we eliminate M, and only M, as a possible solution.

Unlike conflict clauses, which are sometimes deleted from the clause store (Section 4.6.5), clauses like C_M^*, which correspond to solutions, are never deleted.

4.7.1 Minimize statements

We do not yet have a fully working implementation of minimize statements in Smodels$_{cc}$. One approach that could be used to implement them is to follow our approach above for computing multiple answer sets. That is, when a new optimum is found, add a clause corresponding to the complements of all of the choice assignments currently in effect. This will force Smodels$_{cc}$ to never recompute that answer set. When a conflict is found due to a partial assignment's weight exceeding the current minimum, the same type of clause could be added to ensure that the current partial assignment is never repeated. Or, smaller clauses than this could be obtained by performing a more detailed analysis of the conflict. We leave this for possible future work.

CHAPTER 5

ANSWER SET SOLVERS THAT CALL SAT SOLVERS DIRECTLY

In this chapter we look at three recent answer set solvers, each of which calls a SAT solver to do most of the work involved in an answer set search. Our description of these solvers in this chapter sets the stage for Chapter 6, where we compare their performance to that of Smodels$_{cc}$.

Like Smodels$_{cc}$, these solvers incorporate recent SAT technology in order to improve the efficiency of answer set search. However, whereas Smodels$_{cc}$ adapts SAT techniques to an existing ASP solver, each of these three solvers translate answer set problems into a conjunctive normal form representation and use a SAT solver to perform the actual search.

In Section 2.3.5 we discussed various translations from ASP to SAT. We also remarked on difficulties with each kind of translation with respect to obtaining an efficient method for finding answer sets. Because of these difficulties, none of the following three solvers simply performs a direct "many-one" translation of ASP to SAT that preserves the answer set semantics in all cases. Instead, the Cmodels-1 solver is restricted to working on so-called *tight*[21] logic programs only. The ASSAT

[21]Definition 25 in Section A.2 states what it means for a logic program to be tight.

and Cmodels-2 solvers are able to work on tight and non-tight programs. ASSAT and Cmodels-2 start by calling a SAT search engine to find a model of the completion of the logic program. If the model returned by the SAT solver is not an answer set, then they iteratively add *loop formulas* to the completion to deal with unfounded sets, and call the SAT solver again. This process is repeated until the SAT solver returns a genuine answer set or until it is determined that no answer set exists.

5.1 Cmodels-1

For tight logic programs, the answer set semantics and the completion semantics agree exactly regarding what constitutes a valid model. The Cmodels-1 solver [37] is designed to find answer sets of tight programs. First it transforms and simplifies the given program and verifies that the result is tight. (In some cases, but not in general, the transformation and simplification process may transform a non-tight program to a tight one. See [37] for details.) It runs a SAT solver on a CNF version of the program completion, $Comp(P)$ (Definition 10). If the transformed program is not tight, then Cmodels-1 does not attempt to find an answer set.

Cmodels-1 does work with the choice, cardinality, and weight rules supported by Lparse and Smodels. The translation that deals with these extended rule types involves logic programs with "nested expressions" and is explained in [19]. This may involve introducing new atoms to deal with the cardinality and weight rules.

In this dissertation, we are primarily concerned with solvers that will work on both tight and non-tight programs, so we will not discuss Cmodels-1 further.

5.2 ASSAT

The ASSAT program by Lin and Zhao [40] extends the Cmodels-1 approach to non-tight programs by utilizing *loop formulas* (Definition 24 in Section A.2). Given a normal logic program P, ASSAT first creates the program completion, converts the representation to CNF, and calls a SAT solver on this formula. If the SAT solver finds that $Comp(P)$ has no classical Boolean logic models, then ASSAT reports that P has no answer sets. Otherwise, the SAT solver returns a classical model M for $Comp(P)$ and ASSAT checks whether M constitutes an answer set for P. It does this by checking whether there is an *unfounded set* (Definition 12) H with respect to P and M such that M assigns some of the atoms in H the value *true*. If there is no such unfounded set, then M is an answer set of P. Otherwise, some of these atoms in H are contained in a *loop* L in P, such that M does not satisfy L's *loop formula*.[22] So ASSAT translates L's loop formula to CNF, adds the resulting clauses to the current CNF formula (which includes $Comp(P)$), and calls the SAT solver on this new set of clauses. This process is repeated until the SAT solver returns an actual answer set for P or until it finds that the formula that it was asked to solve is unsatisfiable. The ASSAT algorithm is summarized in Table 5.1.

ASSAT does not support any of Lparse's extended rule types (choice, cardinality, or weight rules).

ASSAT supports plugging in a variety of different SAT solvers of different types. The default SAT solver for ASSAT is a version of Chaff, but it can also use GRASP,

[22]The definitions and important results concerning loops and loop formulas are given in Section A.2. Loop formulas serve the role of enforcing that all atoms in unfounded sets are assigned the value *false*. The reason that ASSAT does not simply add all possible loop formulas to $Comp(P)$ before the first call to the SAT solver is that there may be exponentially many loop formulas.

```
ASSAT(NormalProgram $P$)
{
    Construct $Comp(P)$
    Convert $Comp(P)$ to a set of clauses $C$
    loop
        $result \leftarrow$ SAT-solver($C$)
        if ($result =$ “Unsatisfiable”)
            return “No answer set”
        if ($result$ is an answer set of $P$)
            return $result$
        Let $\phi$ be a loop-formula not satisfied by $result$
        Let $C'$ represent $\phi$ as a set of clauses
        $C \leftarrow C \cup C'$
    end-loop
}
```

Table 5.1: ASSAT algorithm

SATO, *satz*, and Walksat. Chaff, GRASP, and *satz* use complete search methods[23] and utilize conflict clauses. *satz* is a complete solver that does not use conflict clauses, but which uses lookaheads similar to what was described for Smodels in Section 3.2.3. Walksat is an incomplete SAT solver, related to the GSAT solver, which uses a local, hill-climbing search [53].

5.3 Cmodels-2

Cmodels-2 [36],[24] uses the basic algorithm and loop formula definition provided by ASSAT for normal programs. Cmodels-2 is able to handle Lparse’s extended rules by using Cmodels-1’s translation, which produces a nested expression logic program from an extended program. It then incorporates the loop formulas defined by Lee and Lifschitz [33] for nested expression programs. Cmodels-2 supports using a version

[23]Recall that *complete* means that the algorithm, in theory, always returns a result about whether or not the input formula is satisfiable. Incomplete SAT methods are, in general, not able to prove that a formula is unsatisfiable.

of Chaff as one of its possible SAT search engines. It also supports using the SIMO solver, which was implemented by Marco Maratea, who is also one of the implementers of Cmodels-2.

When Cmodels-2 uses SIMO as its SAT solver, a closer integration is in place that substantially improves search efficiency on non-tight programs. The advantage that is enjoyed in this circumstance concerns SIMO's conflict clauses. When SIMO finds a satisfying assignment to the problem it has been given from Cmodels-2, the SIMO process is merely paused while Cmodels-2 checks whether the assignment actually constitutes an answer set. If the assignment is not an answer set, then Cmodels-2 adds loop formula clauses to the previous SAT problem that it passed to SIMO, but SIMO *retains the conflict clauses that it possessed when it was paused.* Thus, many of the conflict clauses that SIMO generated in solving the previous SAT instance are immediately available to use in solving the new instance.

In our experiments on the non-tight Hamiltonian cycle problem instances (Section 6.3), we found that if we used Chaff with both ASSAT and Cmodels-2, then the solution times of the two solvers were very close. However, using the SIMO solver, Cmodels-2 did dramatically better on these problems. This is in spite of the fact that Giunchiglia, Lierler, and Maratea report in [24] that SIMO is modeled after Chaff with respect to its algorithms and heuristics, but is "within a factor of 3 slower" because it lacks some of Chaff's low level optimizations.

5.4 Comparison with Smodels$_{cc}$

The main advantage of ASSAT and Cmodels over Smodels$_{cc}$ is their ability to incorporate a variety of recent, highly optimized SAT solvers with relatively little additional programming effort. Because of this, we expect Smodels$_{cc}$ to be somewhat slower than ASSAT and Cmodels on tight programs.

Implementing Smodels$_{cc}$ is relatively complex because of the variety of inference rules involved in Smodels. Also, Smodels$_{cc}$ implements its own version of most of the major processes involved in a DPLL-based SAT solver that uses clause learning, such as Boolean Constraint Propagation, conflict clause creation and management, and computation of heuristics based on conflict clauses. Thus, adding new SAT technologies to Smodels$_{cc}$ is likely to be rather laborious.

The main advantage of Smodels$_{cc}$ is on non-tight problems where a significant number of unfounded sets are produced during the search. Smodels$_{cc}$ inherits from Smodels a test for unfounded sets that is executed potentially at every node in the search tree. ASSAT and Cmodels-2 test for unfounded sets only after the SAT solver returns a prospective answer set, because SAT solvers do not have tests for unfounded sets built into their inference rules. Thus, the process used by ASSAT and Cmodels-2 may waste time by exploring a path that could have been pruned away if an unfounded set test were executed more frequently.

CHAPTER 6

EXPERIMENTAL RESULTS

This chapter summarizes experiments that we performed to compare the performance of Smodels (version 2.26), ASSAT (version 1.50), Cmodels-2 (version 1.04), and Smodels$_{cc}$ (version 1.07). Each test was performed on a 1.533 GHz Athlon XP processor with 1 GB of main memory, running Linux.

In each of the tables below (except for the DLX and bounded model checking benchmarks) the values reported represent the median and maximum number of seconds taken to solve each of eleven (11) randomly generated problem instances. We obtained these runtimes by recording the user time reported by the Linux "/usr/bin/time" command. The runtimes include the time to execute Lparse (version 1.0.13), the default grounder for Smodels, ASSAT, Cmodels-2, and Smodels$_{cc}$. It also includes the time required by the solver to read in the grounded input and to write the resulting output. Each solver process was aborted ("timed out") if it required more than 3600 seconds to complete.

Each of the solvers accepts certain command line options that can affect performance. Also, ASSAT and Cmodels-2 accept a variety of SAT solvers that can be "plugged in" to perform the core of their search. The results that we report for

ASSAT were obtained with the “-s 2” setting. This instructs ASSAT to do extra preprocessing work to simplify the problem, and seems to give marginally better results than the default or “-s 1” settings in our tests. The SAT solver used by ASSAT in our experiments was Chaff2 (version spelt3) [46], which is ASSAT's default solver.

For Cmodels-2, we varied the command line settings in order to obtain the best performance. For the tests involving tight logic programs (specifically, Boolean satisfiability and graph coloring) we used the default setting, which incorporates the zChaff (version 2003.7.1) SAT solver. This setting gave the best performance in these domains. For the tests on non-tight logic programs, we found that Cmodels-2 performed much better with the “-si” option. This setting caused Cmodels-2 to use the SIMO satisfiability solver (version 3.0) [26],[25]. (See Section 5.3 for an explanation of why this improves Cmodels-2's performance on non-tight programs.)

We note that Chaff and SIMO both generate conflict clauses during their search, and use these heavily in guiding their search heuristics. Thus, in these tests, ASSAT and Cmodels-2 are both (indirectly) incorporating conflict clauses in their search.

The Smodels setting that affects search performance is the “-nolookahead” command line option, which instructs Smodels not to use lookaheads. With “-nolookahead” in effect, Smodels makes essentially a random selection of the next atom to instantiate at each choice point. Furthermore, under that setting, it always instantiates the choice atom to *false*. Without the “-nolookahead” option in effect, Smodels uses its default, lookahead-based search heuristic as discussed in Section 3.2.3. The possible advantage of using the “-nolookahead” option is that the solver saves the time that would be used to perform the lookaheads, which can be very expensive. Thus, on rather easy problems with a large number of atoms, the "-nolookahead" option may give better performance.

Smodels$_{cc}$ accepts the same set of command line options as Smodels. When the "-nolookahead" option is in effect, Smodels$_{cc}$ uses a search heuristic based on that of the BerkMin SAT solver, as discussed in Section 4.6.3. Without the "-nolookahead" option in effect, Smodels$_{cc}$ performs lookaheads as Smodels does and uses the same lookahead-based heuristic as Smodels. However, in that case, Smodels$_{cc}$ still uses its conflict clauses for backjumping and for Boolean constraint propagation.

Whether or not the "-nolookahead" option is in effect, Smodels$_{cc}$ always uses lookahead at the root node of the first iteration of the search. Lookahead is repeated at that node until no inferences can be obtained (i.e., until no lookahead on any atom produces a conflict). When the original Smodels uses the "-nolookahead" option, it performs no lookaheads whatsoever.

In Appendix F we compare the results of using Smodels and Smodels$_{cc}$ with and without the "-nolookahead" option. In nearly every test Smodels performed better with lookaheads and Smodels$_{cc}$ performed better without them. Therefore, all of the results that we report in the present chapter have Smodels using lookaheads and Smodels$_{cc}$ not using them (except at Smodels$_{cc}$'s first root node, as mentioned above).

We ran experiments in four problem domains: Boolean satisfiability testing, graph 3-coloring, computing Hamiltonian cycles, and bounded model checking. In the case of satisfiability testing and graph coloring, the reductions that we use are tight. Therefore, we expect that ASSAT and Cmodels-2 will have an advantage over Smodels$_{cc}$ in these domains, as discussed in Section 5.4. However, on the tight problems, we are interested in seeing whether Smodels$_{cc}$ can perform better than Smodels. We are also interested in whether Smodels$_{cc}$'s performance will be reasonably close to that of ASSAT and Cmodels-2.

On the other hand, the Hamiltonian cycle and bounded model checking tests that we run generally incorporate non-tight programs. It is in these domains that we expect Smodels$_{cc}$ to have an advantage over ASSAT and Cmodels-2 because Smodels$_{cc}$, like Smodels, checks for unfounded sets at every node in its search tree.

Some of our tests involved extended logic programs, as we will note below. Since ASSAT does not run on such programs, it was not included in those tests.

6.1 Boolean Satisfiability

Our tests in this domain include randomly generated 3-SAT problems and circuit verification problems. In each case, the problem is originally provided as a CNF-SAT problem, which we have converted to an answer set program. The reduction from SAT to ASP that we used for these tests is the one given in Section 2.3.1.

Evidence that we gathered before running these tests indicated that we might expect the original Smodels, which employs lookaheads and which lacks conflict clauses, to perform better on the random 3-SAT tests than the other ASP solvers (which employ conflict clauses). But we also expected that Smodels might perform significantly worse than the others on the hardware verification benchmarks.

Part of the evidence suggesting that the original Smodels would be the best performer on the random 3-SAT problems came from Simons' doctoral dissertation [54]. There, he reported his tests of Smodels' performance versus that of the ASP solver DLV [13], and the SAT solvers tableau (or *ntab*) [8], SATO [63], and *satz* [35] on random 3-SAT problems. The order of these solvers, from best to worst, in terms of their performance on these tests was:

1. *satz*
2. tableau
3. Smodels
4. DLV
5. SATO

The relevant point here is that the first four solvers in the list used lookaheads, but not conflict clauses, in their search. The worst performer on the tests, SATO, used conflict clauses but not lookaheads.

Those results parallel the results in the SAT2002 competition at the Fifth International Symposium on Theory and Application of Satisfiability Testing, in Cincinnati.[24] The solver that won the contest on "random k-SAT" problems was OKSolver by Oliver Kullmann [31],[32], which utilized lookaheads but not conflict clauses.

However, on the "industrial benchmarks" portion of the SAT2002 contest, the top solvers were zChaff, limmat [6], BerkMin, 2clseq [1], and SIMO, all of which use conflict clauses but not lookaheads.

Our program for generating random 3-SAT problems did not allow duplicate or complementary literals in the same clause. Duplicate clauses were allowed (but were unlikely to occur given our problem parameters). We chose the clauses-to-variables ratios so that they were near the threshold where roughly 50% of randomly generated 3-SAT problems are satisfiable. Problems generated near this threshold tend to be much harder than randomly generated 3-SAT problems with other clause-to-variable ratios [45].

[24]A report on the contest is available at http://citeseer.ist.psu.edu/simon02sat.html.

Median and maximum seconds on 11 random 3-SAT problems

Data Set		Smodels		Smodels$_{cc}$		ASSAT		Cmodels-2		#AS
vars	clauses	med	max	med	max	med	max	med	max	
250	1025	16.5	112.2	36.5	934.5	3.3	552.2	2.5	746.0	10
250	1050	86.1	207.2	657.5	2221.5	334.3	2321.1	777.5	1761.2	6
250	1075	133.4	376.6	641.7	1 abort	801.1	2 abort	883.9	1 abort	3
250	1100	74.4	169.8	343.7	2197.6	329.9	1 abort	360.0	2216.0	1

Median and maximum seconds on 8 DLX benchmark problems

Data Set	Smodels		Smodels$_{cc}$		ASSAT		Cmodels-2		#AS
	med	max	med	max	med	max	med	max	
8 satisfiable	>3600	8 abort	9.9	22.6	2.6	4.4	2.9	10.8	8
8 unsatisfiable	>3600	7 abort	15.2	45.2	6.8	18.2	8.6	14.4	0

Table 6.1: Boolean satisfiability runtimes

In our tables, the column under the "#AS" heading indicates the number of instances that had answer sets.

In order to verify the effectiveness of conflict clauses on non-uniform, "real world" data, we tested the four answer set solvers on 16 of the "DLX" circuit verification benchmarks from Miroslav Velev [61].[25]

As anticipated, Smodels had the best performance of the group on the random 3-SAT problems. However, the solvers that utilized conflict clauses did dramatically better than Smodels on the the DLX benchmarks. The performance of Smodels$_{cc}$ was reasonably close to that of ASSAT and Cmodels-2 on the satisfiability problems as a whole.

[25]The benchmarks that we used were from the Superscalar Suite 1.0 (SSS.1.0), available at http://www.ece.cmu.edu/~mvelev. The eight satisfiable instances that we tested were dlx2_cc_bug01.cnf, ..., dlx2_cc_bug08.cnf. The eight unsatisfiable instances were dlx1_c.cnf, dlx2_aa.cnf, dlx2_ca.cnf, dlx2_cc.cnf, dlx2_cl.cnf, dlx2_cs.cnf, dlx2_la.cnf, and dlx2_sa.cnf.

6.2 Graph Coloring

Graph k-coloring was used as an ASP benchmark in [49] and [40]. Also, it commonly appears online at sites that contain ASP benchmark problems or ASP problem generators. As is the case with the Boolean satisfiability problems, the resulting logic programs are tight, so the unfounded set test is not required. Thus we expect ASSAT and Cmodels-2 to compare well with respect to Smodels$_{cc}$ on these tests.

We first generated 3-coloring problems on uniform, random graphs with 400 to 500 nodes each. "Uniform" in this context means that each possible edge had an equal likelihood of being included in the graph. Similar to our random 3-SAT experiments, we chose the number of edges for our graphs to be near the threshold where the graphs change from likely 3-colorable to likely non-3-colorable.

Next, we generated "clumpy" 3-coloring problems, where each graph consisted of 100 "clumps" of 100 nodes each (for a total of 10,000 nodes per graph). For the first set of clumpy graphs we set our density parameter $d = 150$, which means that we randomly placed 150 edges in each clump and 150 edges between clumps. This gave us graphs with a total of $100 \times 150 + 150 = 15{,}150$ edges each. We then increased d to 170 and 190, obtaining graphs with 17,170 and 19,190 edges, respectively.

The reduction was based on the following normal intentional database (IDB):[26]

```
col(X,r) :- vtx(X), not col(X,g), not col(X,b).
col(X,g) :- vtx(X), not col(X,r), not col(X,b).
col(X,b) :- vtx(X), not col(X,r), not col(X,g).
:- edge(X,Y), col(X,C), col(Y,C), iscolor(C).
iscolor(r).
iscolor(g).
iscolor(b).
```

[26]Note that in practice the expression ":-" is used to express the $\leftarrow$ symbol. In the following, we write our logic programs exactly as they appeared in our datafiles.

Median and max secs on 11 random 3-coloring problems, uniform distribution of edges

Data Set		Smodels		Smodels$_{cc}$		ASSAT		Cmodels-2		#AS
vertices	edges	med	max	med	max	med	max	med	max	
400	900	3.4	163.0	2.9	181.1	0.3	88.3	0.3	88.2	10
400	950	99.5	511.8	94.9	1463.4	58.3	837.7	24.6	837.6	2
400	1000	20.7	134.3	16.4	53.8	3.7	14.3	4.0	13.1	0
500	1100	0.8	4.2	1.7	3.9	0.2	2.4	0.3	2.2	11
500	1150	196.0	356.6	66.1	1 abort	44.4	2 abort	188.3	2 abort	10
500	1200	2753.9	5 abort	>3600	8 abort	>3600	8 abort	>3600	8 abort	0-5

Median and max secs on 11 random 3-coloring problems, clumpy distribution of edges

Data Set		Smodels		Smodels$_{cc}$		ASSAT		Cmodels-2		#AS
vertices	edges	med	max	med	max	med	max	med	max	
10000	15150	256.1	1 abort	21.4	45.2	8.0	9.7	8.4	10.1	10
10000	17170	201.6	3 abort	19.7	21.0	8.5	11.4	10.1	11.8	7
10000	19190	>3600	8 abort	8.0	222.6	3.2	12.9	4.2	11.7	2

Table 6.2: Coloring runtimes

We enforced the restriction in all of our extensional databases that no graph nodes had “self-edges”, and that there were no duplicate edges.

Our results for the four programs on coloring problems parallel our results on the SAT problems: The original Smodels program with lookahead had the best overall performance on the problems created with a uniform distribution (although the results are somewhat mixed). But the three programs that incorporated conflict clauses had substantially better performance on the problems with a non-uniform distribution of edges.

6.3 Hamiltonian Cycles

This seems to be the most popular domain for benchmarking ASP search engines. For instance, timings on Hamiltonian cycle problems have been reported in [54, 49, 16, 40, 39, 24]. Perhaps the principle reason for its popularity is that the more natural and concise reductions for Hamiltonian cycle problems produce non-tight programs where the answer set semantics differs from the completion semantics. Thus, some form of unfounded set test is needed in order to ensure that the search engine returns a valid answer set. Because classical Boolean logic lacks the concept of negation as failure embodied in the negation of unfounded sets, it seems to be difficult to find a concise, practical reduction of the Hamiltonian cycle problem to the Boolean satisfiability problem. Thus, this is a domain where ASP solvers would seem to have an advantage over SAT solvers. This seems to be evidenced in the empirical results reported by Simons [54]. He tested the performance of the SAT solvers SATO and *satz* on two different reductions of the Hamiltonian cycle problem to Boolean satisfiability. The SAT problem encodings were much larger (and more complex) than the ASP encoding. The time required to solve the problems was orders of magnitude higher for the SAT solvers than for Smodels, which Simons suggested was due to the encoding size issue.

We used directed graphs for our Hamiltonian cycle tests. We experimented with five different reductions. The first four reduced the problem instances to normal logic programs. The fifth reduction resulted in extended programs. Appendix E lists each of the five reductions and provides results that we obtained comparing the performance obtained with these reductions using Smodels, Smodels$_{cc}$, ASSAT, and Cmodels-2. The best reduction to normal logic programs is labeled *NNT3* in the appendix. The best reduction overall was the reduction labeled *Extended*, which uses extended rules. Since ASSAT does not work with extended rules, and because the results with *NNT3* were not much worse than those with *Extended*, we used *NNT3* for most of the tables below. Our last table with Hamiltonian Cycle results uses the *Extended* reduction, as noted below.

Uniform Random Graphs

One set of Hamiltonian cycle tests involved generating random directed graphs using a uniform distribution of edges. The results from these tests are given in Table 6.3. Smodels$_{cc}$ without lookahead had by far the best performance on these tests. In fact, using the *NNT3* reduction and this distribution of edges, we did not generate any problems that were hard for Smodels$_{cc}$ to solve, even though we created graphs that were quite large. (Our 6000 node problems resulted in ground instantiations from Lparse that were approximately 12MB in size. These groundings were too large to run through ASSAT.) Unsatisfiable instances taken from this distribution were generally solved by each of the solvers with no backtracks.

On these problems, the runtimes for Smodels$_{cc}$ are much better than the runtimes for the original Smodels program. This was somewhat of a surprise since the problems were generated using a uniform distribution, so we thought that Smodels$_{cc}$ might be the slower of the two solvers on these problems. An examination of some of the search statistics explained why the original Smodels was slower on this problem set: the time used by Smodels to perform its lookaheads is quite great on such large problem instances. It turns out that on the problems instances that did have Hamiltonian cycles, Smodels almost never backtracked. However, each 6000 node problem instance mentioned between 100,000 and 150,000 atoms in the grounded logic program. Furthermore, solving the problem could involve several thousand

Median and max secs on 11 Hamiltonian cycle problems,
uniform distribution of edges

Data Set		Smodels		Smodels$_{cc}$		ASSAT		Cmodels-2		#AS
vertices	edges	med	max	med	max	med	max	med	max	
1000	4000	1.0	1.1	1.0	1.0	0.9	1.1	1.0	1.1	0
1000	4500	1.2	132.0	1.2	4.7	1.1	971.3	1.3	308.0	4
1000	5000	172.7	201.0	3.5	4.2	52.6	1064.1	6.8	438.4	6
1000	5500	244.6	269.5	4.6	5.5	275.5	1440.6	141.1	175.7	10
6000	30000	8.4	3 abort	8.0	51.0	...	...	8.3	2 abort	3
6000	33000	>3600	9 abort	55.9	64.2	...	...	>3600	9 abort	9
6000	36000	>3600	7 abort	61.5	73.7	...	...	1963.4	3 abort	7
6000	39000	>3600	10 abort	72.7	97.0	...	...	>3600	7 abort	10

Table 6.3: Runtimes on uniform Hamiltonian cycle problems

choice points, even if no backtracks were involved. The time required to perform all of the resulting lookaheads resulted in Smodels exceeding the 3600 second time limit.

An obvious solution to try with the original Smodels program, then, is to run it with no lookaheads. However, as we see in Appendix F, this results in only somewhat better results on these problems. In this case, the Smodels search heuristic is running "blind" (i.e., choice atoms are selected at random, and always initially assigned the value *false*), and usually requires hundreds of thousands or millions of backtracks in order to solve the problem. With Smodels$_{cc}$ the cost of lookaheads is avoided (except for the lookaheads performed before the first choice point), and the program usually requires no more than a few hundred backtracks, even on the 6000 node problem instances.

Clumpy Random Graphs

We sought to produce some hard Hamiltonian cycle problems that were of reasonable size and had a less uniform (more "clumpy") distribution of edges. We designed these experiments so that each problem instance would have some, but relatively few, Hamiltonian cycles.

For this purpose we generated each random mn-node "clumpy" graph G as follows: Let n be the number of clumps in the graph and let m be the number of nodes in each clump. First generate an n-node graph A as the "master graph," specifying how the clumps are to be connected; each vertex of A will correspond to a clump in the final graph. Add random edges to A until A has a Hamiltonian cycle.

Now generate the clump C corresponding to a vertex v of A. Let $x = indegree(v)$ and $y = outdegree(v)$. C has m nodes; select x nodes to be "in-nodes" and y different nodes to be "out-nodes"; increase C to $x + y$ nodes if $x + y > m$. Add random edges to clump C until there is a Hamiltonian path from each in-node of C to each out-node of C. (Thus C will have at least xy Hamiltonian paths.)

Finally, for every edge (v_i, v_j) in the master graph A, insert an edge from an out-node of C_i to an in-node of C_j. Every in-node in every clump is to have exactly one incoming edge from another clump. Likewise, every out-node in every clump is to have exactly one outgoing edge to another clump. G will have at least one Hamiltonian cycle.

The results obtained on these problems are shown in Table 6.4. Here again, we used reduction *NNT3*, which involves normal logic programming rules only. Smodels$_{cc}$ had the strongest performance of all of the solvers.

Because Smodels$_{cc}$ and Cmodels-2 were clearly the two best performers on the clumpy graph Hamiltonian cycle tests, because both of these solvers support extended logic programs, and because the *Extended* reduction from Appendix E gave somewhat better results than the normal logic program reductions, we reran the above tests using

Median and max secs on 11 Hamiltonian cycle problems,
clumpy distribution of edges.
n = number of clumps; m = vertices per clump

Data Set		Smodels		Smodels$_{cc}$		ASSAT		Cmodels-2		#AS
n	m	med	max	med	max	med	max	med	max	
10	10	0.4	1 abort	0.4	0.9	0.8	1.4	0.5	0.9	11
12	12	620.9	4 abort	0.7	1.3	2.6	10.8	2.6	4.5	11
14	14	>3600	10 abort	1.8	10.4	108.8	194.1	11.4	83.4	11
16	16	>3600	9 abort	8.8	19.2	263.0	608.8	24.8	100.6	11
18	18	>3600	11 abort	21.3	156.0	>3600	6 abort	133.7	2063.5	11
20	20	>3600	11 abort	102.4	264.4	>3600	8 abort	206.0	1 abort	11

Table 6.4: Runtimes on clumpy Hamiltonian cycle problems

the *Extended* reduction. The results are shown in Table 6.5. Smodels$_{cc}$ maintained a significant performance advantage in these tests.

6.4 Bounded Model Checking

In [28], Heljanko and Niemelä discuss bounded model checking of asynchronous concurrent systems as an application area for answer set programming. They present translations of problems of bounded reachability, deadlock detection, and linear temporal logic model checking to answer set programming problems. They also report a number of experiments where they show the performance of the Smodels solver on these problems. They have provided for download[27] most of the logic programs used in their experiments (specifically, all of the problems from Table 1 of their paper). We tested Smodels, Smodels$_{cc}$, and Cmodels-2 on this problem set. The logic programs from this collection employed extended rules, so we were unable to include ASSAT

[27]http://www.tcs.hut.fi/~kepa/experiments/boundsmodels/

Median and max secs on 11 Hamiltonian cycle problems, clumpy distribution of edges, Extended reduction

Data Set		Smodels$_{cc}$		Cmodels-2		#AS
# of clumps	vertices / clump	med	max	med	max	
10	10	0.3	0.4	0.3	0.5	11
12	12	0.4	1.0	1.6	6.7	11
14	14	1.8	11.2	6.4	73.8	11
16	16	4.7	21.0	16.2	31.5	11
18	18	42.5	278.4	145.0	1 abort	11
20	20	83.8	364.0	190.5	3 abort	11

Table 6.5: Runtimes on clumpy Hamiltonian cycle problems, *Extended* reduction

in these tests. Most of the programs were non-tight, and Smodels$_{cc}$'s unfounded set test was often used to derive inferences during its search.

Tables 6.6 and 6.7 report the runtimes that we obtained on these problems. We have separated the results into two tables: one for problem instances generated under the "step" semantics, and one for problems generated under the "interleaving" semantics. (See the Heljanko and Niemelä paper for a description of the step and interleaving semantics.) Smodels$_{cc}$ had the best overall performance on these problems.

Bounded model checking runtimes in secs
(Deadlock checking under “step” semantics)

Problem	Smodels	Smodels$_{cc}$	Cmodels-2	AS?
dp-6.s-O2-b1	0.00	0.00	0.00	Yes
dp-8.s-O2-b1	0.01	0.00	0.01	Yes
dp-10.s-O2-b1	0.00	0.00	0.00	Yes
dp-12.s-O2-b1	0.00	0.00	0.00	Yes
key-2.s-O2-b25	489.58	6.18	73.45	No
mmgt-3.s-O2-b7	3.01	0.34	0.21	Yes
mmgt-4.s-O2-b8	163.71	2.38	42.99	Yes
q-1.s-O2-b9	0.06	0.08	0.10	Yes
dartes-1.s-O2-b32	0.69	0.96	6.01	Yes
elev-1.s-O2-b4	0.03	0.02	0.02	Yes
elev-2.s-O2-b6	0.14	0.11	0.06	Yes
elev-3.s-O2-b8	1.49	1.05	0.42	Yes
elev-4.s-O2-b10	44.67	16.61	5.82	Yes
hart-25.s-O2-b1	0.01	0.01	0.01	Yes
hart-50.s-O2-b1	0.01	0.01	0.01	Yes
hart-75.s-O2-b1	0.02	0.01	0.02	Yes
hart-100.s-O2-b1	0.02	0.03	0.02	Yes
Total	703.45	27.79	129.15	

Table 6.6: Bounded model checking runtimes - step semantics

Bounded model checking runtimes in secs
(Deadlock checking under "interleaving" semantics)

Problem	Smodels	Smodels$_{cc}$	Cmodels-2	AS?
dp-6.i-O2-b6	0.03	0.04	0.09	Yes
dp-8.i-O2-b8	0.09	0.18	4.07	Yes
dp-10.i-O2-b10	0.93	208.51	231.15	Yes
dp-12.i-O2-b12	150.77	3.71	>3600.00	Yes
key-2.i-O2-b26	16.11	9.13	89.65	No
mmgt-3.i-O2-b10	23.81	3.24	80.05	Yes
mmgt-4.i-O2-b11	518.54	685.95	>3600.00	No
q-1.i-O2-b17	731.58	116.74	2597.67	No
dartes-1.i-O2-b32	0.47	0.76	3.39	Yes
elev-1.i-O2-b9	0.20	0.39	0.60	Yes
elev-2.i-O2-b12	2.10	5.33	37.72	Yes
elev-3.i-O2-b15	90.83	27.50	705.44	Yes
elev-4.i-O2-b13	593.83	53.72	373.00	No
hart-25.i-O2-b5	0.23	0.08	0.05	No
hart-50.i-O2-b5	1.12	0.25	0.13	No
hart-75.i-O2-b5	3.51	0.54	0.24	No
hart-100.i-O2-b5	7.82	0.97	0.35	No
Total	2141.97	1117.04	>11323.60	

Table 6.7: Bounded model checking runtimes - interleaving semantics

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Summary

We have shown how to incorporate clause learning techniques directly into a solver whose rules of inference are specifically oriented towards reasoning under the answer set semantics. Of special significance in this regard is that our algorithm produces conflict clauses from reasoning that includes negative inferences drawn from the detection of unfounded sets. Issues that needed to be addressed in order to accomplish this included:

- How does one generate an *implication graph* from applications of Smodels-style inference rules, including search involving normal logic programming rules (addressed in Section 4.3.2) and extended rules (addressed in Section 4.3.4)?
- How does one analyze an implication graph involving cycles (Section 4.4.2)?
- Smodels' negation based on unfounded sets may introduce several conflicts simultaneously. Does it matter on which conflict we base our analysis? (The answer is yes, as discussed in Section 4.4.2. It is important for performance and completeness of the algorithm on non-tight problems.)

Correctness and completeness of the Smodels$_{cc}$ algorithm is proven in Appendices C and D.

7.2 Conclusions

1. Adding conflict clause learning to Smodels substantially speeds up Smodels on many problem domains, including both of the industrial benchmark domains we tried (hardware verification and bounded model checking). The most popular benchmarking domain for answer set solvers is the Hamiltonian cycle problem. On our tests in this domain, Smodels$_{cc}$ was orders of magnitude faster than the original Smodels solver.

2. The approach of Smodels$_{cc}$ is substantially faster on non-tight programs than the approach of ASSAT and Cmodels-2. Smodels$_{cc}$ adds conflict clause learning directly to Smodels. Therefore, it preserves Smodels' unfounded set test, which is executed potentially at every node in the branching search. ASSAT and Cmodels-2 iteratively call SAT solvers (which themselves usually incorporate conflict clause learning), and test for unfounded sets only after the SAT solvers return their results. The fact that the unfounded set test is executed more frequently in Smodels$_{cc}$ gives Smodels$_{cc}$ greater pruning of the search space on non-tight problems.

7.3 Future work

One possible direction for future work is to investigate how to incorporate clause learning techniques into disjunctive logic programming systems such as DLV [13] and GnT [30].

Another possibility is to investigate calling a SAT solver on the program completion, as done by ASSAT and Cmodels-2, but including an Smodels-type unfounded set check in the SAT solver's search. Such a system might provide many of the strengths of both the ASSAT/Cmodels-2 approach and the approach of Smodels$_{cc}$.

APPENDIX A

REDUCTIONS OF ASP TO SAT

In this appendix we present translations that reduce the problem of whether a normal logic program has an answer set to the problem of whether a Boolean formula in conjunctive normal form has a satisfying assignment.

As mentioned in Section 2.3.5, we divide these reductions into two classes: (1) those reductions that operate in polynomial time, but significantly increase the number of atoms in the problem representation, and (2) those that do not increase the number of atoms, but exponentially increase the representation size.

A.1 Reductions that Increase the Number of Atoms

The reductions in this category seem to follow an idea that goes back to a 1955 paper by Spector [56]. The following reduction is based on his idea.

Let P be a normal logic program, and let $N = |Atoms(P)|$.

Recall that $M \subseteq Atoms(P)$ is an answer set of P if and only if M is the deductive closure of P^M. The idea of this reduction is to regard M as being the union of several stages that are indexed from 0 to N. No atoms will belong to stage 0. An atom $h \in Atoms(P)$ will be considered part of stage $i > 0$ if and only if (1) h belongs to stage $i - 1$, or (2) there is a rule of the form $h \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m$, where

$\neg b_1, \ldots, \neg b_m \notin M$ and $a_1, \ldots, a_n$ all belong to stage $i-1$. Then h is in stage N if and only if it is in the deductive closure of P^M.

For each $h \in \mathit{Atoms}(P)$ we introduce new atoms $h^0, h^1, \ldots, h^N$ to indicate in which stages h belongs. For each rule $R = \quad h \leftarrow a_1, \ldots, a_n, \mathit{not}\ b_1, \ldots, \mathit{not}\ b_m$ and for each $1 \leq i \leq N$ we define the propositional formula ${\varphi^i}_R = \quad a_1^{i-1} \wedge \ldots \wedge a_n^{i-1} \wedge \neg b_1 \wedge \ldots \wedge \neg b_m$. Intuitively, ${\varphi^i}_R$ states whether rule R is ready to fire at stage i of the construction of the deductive closure of P^M. Also, for each $h \in \mathit{Atoms}(P)$, we define the following propositional formulas:

1. $\varphi_h^0 = \neg h^0$
2. For $1 \leq i \leq N$, $\varphi_h^i \ = (h^i \equiv (h^{i-1} \vee \bigvee \{{\varphi^i}_R :$ $R \in P$ is a rule with h in its head$))\}$

We then define $\varphi_h = ((h \equiv h^N) \wedge \bigwedge \{\varphi_h^i : 0 \leq i \leq N\})$. We also define $\varphi_P = \bigwedge \{\varphi_h : h \in \mathit{Atoms}(P)\}$.

We assert without proof that an interpretation M is an answer set of P if and only if M satisfies φ_P. From the point of view of solving problems efficiently, the drawbacks of the above translation are (1) the new representation is of size $\Theta(N \times |P|)$, and (2) the number of new atoms introduced is $\Theta(N^2)$.

Ben-Eliyahu and Dechter [5] gave a translation of "head-cycle free disjunctive logic programs" to SAT, extending the idea shown above. The class of logic programs that they considered includes normal logic programs as a special case.

Recently, Janhunen [29] presented another translation based on this idea, directed at translating normal programs only. Janhunen's translation involves a relatively intricate binary encoding of the stage numbers. Through this encoding, he was able to shrink the size of the resulting propositional logic formula to $\Theta(|P| \times log_2(|Atoms(P)|))$ and the number of new atoms that are introduced to $\Theta(|Atoms(P)| \times log_2(|Atoms(P)|))$, although rather large constants of proportionality seem to be involved in both the formula size and the number of new atoms. Janhunen's result is theoretically interesting. However, it remains to be seen whether this reduction will yield an efficient method of finding answer sets. Suppose, for instance, that we translate ASP search problems to SAT using Janhunen's translation, and call a SAT solver on the result. Because each stage number is represented in binary, it is not clear that partial interpretations that lead to unfounded sets will cause the SAT search engine to immediately infer that the atoms in the unfounded set are *false*. An important advantage of the Smodels solver, which is an advantage inherited by our solver Smodels$_{cc}$, is that a test for unfounded set is built into the search engine. SAT solvers do not have a unfounded set test built into them. If the representation does not force the SAT solver to make unfounded set-related inferences as soon as possible in the search, then a great deal of efficiency may be lost. (This is one of the main conclusions we draw from our experimental results in Sections 6.3 and 6.4.)

A.2 An Exponential Space Reduction that Does Not Increase the Number of Atoms

The two preeminent, current answer set solvers that translate answer set problems to a propositional logic form and use SAT solvers for the bulk of the search are ASSAT [40] and Cmodels-2 [24]. In Chapter 5 we discuss how these solvers operate, and in Chapter 6 compare their performance to that of Smodels and Smodels$_{cc}$. In the current section, we will present the translation from ASP to SAT on which ASSAT and Cmodels-2 are based. This translation was presented by Lin and Zhao [40] in their paper on ASSAT.

An advantage of this translation is that it does not introduce new atoms into the problem. The main drawback is that the number of clauses in the formula can be exponential in the size of the logic program. The reduction is based on the program completion (Definition 10) and so-called *loop formulas*.

Definition 22 (Positive Dependency Graph) *Let P be a normal logic program. The* positive dependency graph *of P is a directed graph whose nodes are the atoms mentioned in P. An edge exists from node p to node q in the graph iff there is a rule in P such that p is the head of R and q appears as a positive subgoal in R.*

A directed graph is said to be *strongly connected* if, between any ordered pair of nodes $<p, q>$, there is a path from p to q. Given a directed graph, a set of nodes S is a *strongly connected component* if for any ordered pair of nodes $<p, q>$, where $p, q \in S$, there is a path from p to q.

Definition 23 (Loop) *[40] A set L of atoms is called a* loop *of a logic program if L is a strongly connected component of the program's positive dependency graph.*

Given a rule R we will denote the set of positive subgoals of R by $PSG(R)$, and the head of R by $head(R)$. For each loop L in a logic program, Lin and Zhao associate two sets of rules with it:

$$\begin{array}{rcl} R^{+}(L) & = & \{R \in P \mid head(R) \in L,\ PSG(R) \cap L \neq \emptyset\} \\ R^{-}(L) & = & \{R \in P \mid head(R) \in L,\ PSG(R) \cap L = \emptyset\} \end{array}$$

Example 9 *Let P be the logic program from Section 2.3.3 that encodes the instance of the Hamiltonian cycle problem displayed in Figure 2.1. Then $L = \{r(5), r(6), r(7), r(8)\}$ is a loop of P. Furthermore,*

$$R^{+}(L) = \left\{ \begin{array}{l} r(6) \leftarrow hc(5,6),\ edge(5,6),\ r(5) \\ r(7) \leftarrow hc(6,7),\ edge(6,7),\ r(6) \\ r(8) \leftarrow hc(7,8),\ edge(5,6),\ r(7) \\ r(5) \leftarrow hc(8,5),\ edge(5,6),\ r(8) \end{array} \right\}$$

and

$$R^{-}(L) = \left\{\ r(6) \leftarrow hc(4,6),\ edge(4,6),\ r(4)\ \right\}$$

Lin and Zhao motivate their idea of loop formulas as follows: "For any given logic program P and any loop L in P, one can observe that $\emptyset$ is the only answer set of $R^{+}(L)$. Therefore an atom in the loop cannot be in any answer set unless it is derived using some other rules, i.e. those from R^{-}."

Definition 24 (Loop Formula) *[40] Let P be a logic program, and L a loop in it. Suppose that we enumerate the rules in $R^{-}(L)$ as follows:*

$$\begin{array}{c} p_1 \leftarrow Body_{11}, \ldots, p_1 \leftarrow Body_{1k_1}, \\ \vdots \\ p_n \leftarrow Body_{n1}, \ldots, p_n \leftarrow Body_{nk_n}, \end{array}$$

then the loop formula *associated with L is the following implication:*

$$\neg(Body_{11} \vee \cdots \vee Body_{1k_1} \vee \cdots \vee Body_{n1} \vee \cdots \vee Body_{nk_n}) \supset \bigwedge_{p \in L} \neg p.$$

In the above definition, the *not* operators used in negative subgoals are, naturally, translated to the classical negation operator $\neg$, and the commas in rule bodies are treated as the conjunction operator $\wedge$.

Example 10 *Consider the loop given in Example 9. The corresponding loop formula is:*

$$\neg(hc(4,6) \wedge edge(4,6) \wedge r(4)) \supset (\neg r(5) \wedge \neg r(6) \wedge \neg r(7) \wedge \neg r(8))$$

Note that the completion semantics model given in Table 2.2 violates this loop formula, but the answer set given in Table 2.1 satisfies it.

Proposition 3 (Lin and Zhao [40]) *Let P be a logic program, $Comp(P)$ its completion, and LF the set of loop formulas associated with the loops of P. Then an interpretation is an answer set of P iff it is a model of $Comp(P) \cup LF$.*

Definition 25 (Tight Program) *A normal logic program is said to be* tight *if its positive dependency graph contains no loops.*

Fages [18] showed in 1994 that if a program is tight, then its answer sets are exactly it models under the completion semantics. This, of course, follows as an immediate corollary from Proposition 3, which was stated by Lin and Zhao in 2002.

We can see that a logic program may have exponentially many loops by once again considering the IDB given in Section 2.3.3 for expressing the Hamiltonian cycle problem on directed graphs. If G is a complete directed graph on n vertices, then for any set S of vertices in V, $\{r(s) \mid s \in S\}$ is a loop of the corresponding grounded logic program. Hence, that program will have 2^n loops.

Because there may be so many loop formulas for a given problem, ASSAT and Cmodels-2 do not directly translate logic programs into the formula given in Proposition 3. We discuss their approach in Chapter 5, and also compare it to our approach with Smodels$_{cc}$. The main point in this appendix is that it seems to be difficult to translate the ASP problem directly to SAT in a way that will yield an efficient implementation for finding answer sets.

APPENDIX B

WEIGHT CONSTRAINT RULES: SEMANTICS AND TRANSLATIONS

With the exception of minimize statements, all of the rule types supported by Lparse and Smodels can be defined in terms of weight constraint rules. Simons, Niemelä, and Soininen introduced weight constraint rules in [55] and implemented them in Lparse and Smodels. In Section B.1 below, we list their formal definitions concerning the answer set semantics of weight constraint rule programs. Then, in Section B.2, we give their translation of weight constraint rule programs to programs consisting of weight rules and choice rules only.

Much of this appendix quotes directly from [55].

B.1 Formal Semantics

Recall the following definitions from Section 2.4:

A *weight constraint* is an expression of the form

$$l \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, not\ b_1 = w_{b_1}, \ldots, not\ b_m = w_{b_m}\} \leq u \tag{B.1}$$

A *weight constraint rule* is an expression of the form

$$C_0 \leftarrow C_1, \ldots, C_n$$

where each C_i is a weight constraint. A *weight constraint rule program* is a program consisting of weight constraint rules.

Let M be a total interpretation of $Atoms(P)$, where P is a weight constraint rule program. As usual, we identify M with the set of atoms that are *true* in M.

Definition 26 *A set of atoms M* satisfies *a weight constraint C of the form (B.1), denoted by $M \models C$, iff $l \leq w(C, M) \leq u$, where*

$$w(C, M) = \sum_{a_i \in M} w_{a_i} + \sum_{b_i \notin M} w_{b_i}$$

is the sum of the weights of the literals in C satisfied by M.

A rule $C_0 \leftarrow C_1, \ldots, C_n$ is satisfied by M ($M \models C_0 \leftarrow C_1, \ldots, C_n$) iff M satisfies C_0 whenever it satisfies each of $C_1, \ldots, C_n$. A program P is satisfied by M ($M \models P$) if each rule in P is satisfied by M.

In defining the answer set semantics for weight constraint rule programs, Simons, *et al.* assume that each weight constraint has only non-negative weights. They provide a method of translating weight constraints with negative weights to weight constraints with non-negative weights. We refer the reader to their paper for this translation.

Definition 27 (Reduct of a weight constraint) *The reduct C^M of a weight constraint C of the form (B.1) with respect to a set of atoms M is the constraint*

$$l' \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}\}$$

with the lower bound

$$l' = l - \sum_{b_i \notin M} w_{b_i}.$$

Thus the reduct of a weight constraint contains no reference to the *not* operator.

If C is a weight constraint, then we let $PSG(C)$ signify the positive subgoals in C.

Definition 28 (Reduct of a weight constraint rule program) *Let P be a weight constraint rule program and M a set of atoms. The* reduct P^M *of P with respect to M is defined by*

$$\begin{aligned} P^M = \{p \leftarrow C_1^M, \ldots, C_n^M \mid \; & C_0 \leftarrow C_1, \ldots, C_n \in P,\ p \in \ PSG(C_0) \cap M \\ & \textit{and } w(C_i, M) \leq u \textit{ for all } C_i \textit{ of the form (B.1)} \\ & \textit{where } i = 1, \ldots, n\} \end{aligned}$$

Definition 29 (Answer set of a weight constraint rule program) *[55] Let P be a weight constraint rule program with non-negative weights. Then $M \subseteq Atoms(P)$ is an* answer set *of P iff the following two conditions hold:*

(i) $M \models P$,

(ii) $M = DeductiveClosure(P^M)$.

B.2 Translation to Basic Rules

Simons, Niemelä, and Soininen refer to weight rules and choice rules as *basic rules.* The following is their technique for translating an arbitrary weight constraint rule to a set of basic rules.

Let $C_0 \leftarrow C_1, \ldots, C_r$ be an arbitrary weight constraint rule. For each $i = 0, \ldots, r$, a constraint C_i of the form

$$l \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, not\ b_1 = w_{b_1}, \ldots, not\ b_m = w_{b_m}\} \leq u$$

is mapped into two rules. The first of these two rules encodes whether the lower bound of C_i is satisfied. The later rule encodes whether the upper bound is *not* satisfied.

$$c_i^l \;\; \leftarrow \;\; l \leq \{a_1 = w_{a_1}, \ldots, not\ b_m = w_{b_m}\}$$

$$c_i^u \quad \leftarrow \quad u < \{a_1 = w_{a_1}, \ldots, \mathit{not}\ b_m = w_{b_m}\}$$

where c_i^l and c_i^u are new atoms.[28] Then the following choice rule and integrity constraint rules are added:

$$\begin{aligned}
\{a_1, \ldots, a_n\} \quad &\leftarrow \quad c_1^l, \mathit{not}\ c_1^u, \ldots, c_r^l, \mathit{not}\ c_r^u \\
&\leftarrow \quad \mathit{not}\ c_0^l, c_1^l, \mathit{not}\ c_1^u, \ldots, c_r^l, \mathit{not}\ c_r^u \\
&\leftarrow \quad c_0^u, c_1^l, \mathit{not}\ c_1^u, \ldots, c_r^l, \mathit{not}\ c_r^u
\end{aligned}$$

where $a_1, \ldots, a_n$ are the positive subgoals mentioned in the head constraint C_0.

[28]If, as in Smodels, only integer weights are allowed, then the latter rule can be expressed by increasing u by one and changing the "$<$" to "$\leq$". Otherwise, the syntax and semantics are extended to deal with the $<$ operator.

APPENDIX C

SMODELS$_{CC}$ PROOF OF CORRECTNESS

The inference rules used by Smodels$_{cc}$ are the inference rules of Smodels plus the unit clause rule applied to the conflict clauses produced by Smodels$_{cc}$.[29] Since each of these inference rules is known to be sound, the remaining issue is to ensure that the conflict clauses produced by Smodels$_{cc}$ are sound. This is what we show in the following proposition. We restrict our attention in this proof to normal logic programs.

Proposition 4 (Smodels$_{cc}$ correctness) *Let P be a normal logic program. Then every conflict clause $\mathcal{C}$ generated by Smodels$_{cc}$ during the search for an answer set for P is satisfied by every answer set of P.*

Before we proceed with the proof of Proposition 4 we will prove Lemma 5 below.

Definition 30 (Classical Model of a Normal Logic Program) *Let P be a normal logic program, and let I be a total interpretation of the atoms in P (i.e., $I : Atoms(P) \rightarrow \{\text{true}, \text{false}\}$). Then I is a* classical model *of M if, for every rule*

$$R = \quad h \leftarrow c_1, \ldots, c_m, not\ d_1, \ldots, not\ d_n$$

[29]Recall that the process of applying the unit clause rule is called Boolean Constraint Propagation (BCP).

in P, I satisfies the Boolean logic formula

$$(c_1 \wedge \ldots \wedge c_m \wedge \neg d_1 \wedge \ldots \wedge \neg d_n) \to h.$$

The following lemma is an immediate consequence of Theorem 1 in the paper by Gelfond and Lifschitz [22] that originally defined the stable model/answer set semantics.

Lemma 5 *Suppose M is an answer set of a normal logic program P. Then M viewed as a total interpretation is a classical model of P.*

Proof of Lemma 5: Let

$$R = \quad h \leftarrow c_1, \ldots, c_m, not\ d_1, \ldots, not\ d_n$$

be a rule in P. Suppose M satisfies

$$c_1 \wedge \ldots \wedge c_m \wedge \neg d_1 \wedge \ldots \wedge \neg d_n$$

Then, $c_1, \ldots, c_m \in M$ and $d_1, \ldots, d_n \notin M$. Since $d_1, \ldots, d_n \notin M$,

$$R^M = \quad h \leftarrow c_1, \ldots, c_m$$

is a rule in P^M. Since $c_1, \ldots, c_m \in M$, and M is an answer set of P, $c_1, \ldots, c_m \in$ $\mathit{DeductiveClosure}(P^M)$. Thus, $h \in \mathit{DeductiveClosure}(P^M) = M$, and M satisfies h. $\square$

Proof of Proposition 4: Induct on the sequence of conflict clauses generated by Smodels$_{cc}$ during the search. So assume that every answer set of P satisfies every conflict clause generated before the latest conflict clause. (Call this Induction Hypothesis 1.) Let G be the implication graph corresponding to the latest conflict in the search. Let x be the conflict node in G and let $\neg x$ be x's complement node. We will assume that the conflict edge has not yet been added to G. (So, if there is an edge from $\neg x$ to x, it is only because $\neg x$ was used in the inference rule application that derived x.)

Recall Definitions 19-21 from Section 4.5: A *predecessor* of a node v is a node with an edge to v. A *choice node* is a node with no predecessors. A set of nodes D *dominates* a node s if every path from any choice node of G to s includes an element of D.

As is usual, we allow paths consisting of a single node. So every node has a path to itself. Therefore, an immediate consequence of the above definitions is that if D dominates a choice node s_0, then necessarily $s_0 \in D$.

Another immediate consequence is that, if D dominates s, then either (i) $s \in D$, or (ii) D dominates every predecessor of s.

Recall also from Section 4.5 that Smodels$_{cc}$ generates its conflict clause by computing a set of nodes $D = \{x_1, \ldots, x_n\}$ that dominates the conflicting pair of nodes x and $\neg x$. To complete our inductive proof of Proposition 4, then, it suffices to prove the following claim:

Claim 1 *If D is a set of nodes in G that dominates node s, then every answer set of P that satisfies D satisfies s.*

That Claim 1 is sufficient to complete our proof of Proposition 4 follows from the fact that no answer set of P simultaneously satisfies x and $\neg x$. From Claim 1, this implies that no answer set of P simultaneously satisfies $x_1, \ldots, x_n$, since these nodes dominate x and $\neg x$. Hence, the conflict clause $\neg x_1 \vee \ldots \vee \neg x_n$ generated by Smodels$_{cc}$ is satisfied by any answer set of P.

Proof of Claim 1: An implication graph G corresponds to a derivation $\mathcal{D} = [\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_m]$ where each $\mathcal{S}_i$ is a set of nodes in G consisting of either

1. a singleton node $\{s\}$ with no predecessor in G (i.e., s is a choice node),
2. a singleton node $\{s\}$ derived from the unit clause rule,
3. a singleton node $\{s\}$ derived from the *modus ponens* inference rule,
4. a singleton node $\{s\}$ derived from the *backchain true* inference rule,
5. a singleton node $\{s\}$ derived from the *backchain false* inference rule,
6. a singleton node $\{s\}$ derived from the *all rules cancelled* inference rule,
7. a set of negative nodes $\{\neg a \ : \ a \in U\}$ inferred by the *unfounded set* inference rule, where U is the entire unfounded set detected by Smodels$_{cc}$ for that inference.

Note: A single application of the backchain true inference rule can also infer several literals, but we will consider each such literal as a separate $\mathcal{S}_i$. The reason that we treat the unfounded set inference rule differently in this regard is that we want it to be the case that each edge in G goes from a node in $\mathcal{S}_j$ to a node in $\mathcal{S}_k$ where $j \leq k$. Since some of the nodes inferred from a single application of the unfounded set rule may form a cycle, we consider inferences obtained from the entire unfounded set to be a single $\mathcal{S}_i$.

We prove Claim 1 by induction on the length m of the derivation $\mathcal{D}$. So, we take as Induction Hypothesis 2 that if D' dominates node $s' \in \bigcup_{i=1}^{m-1} \mathcal{S}_i$ then every answer set of P that satisfies D' also satisfies s'. Now suppose D dominates node $s \in \mathcal{S}_m$.

Also, suppose that M is an answer set of P that satisfies D. We will show that M satisfies s.

Case 1: $\mathcal{S}_m = \{s\}$, where s has no predecessors in G. Then, since D dominates s, we have $s \in D$. Thus M satisfies s.

Case 2: $\mathcal{S}_m = \{s\}$ where s was inferred via the unit clause rule from a previously generated conflict clause $\mathcal{C}'$. If $s \in D$ we are done. Otherwise, each predecessor of s in G is dominated by D. Each predecessor of s is a member of some $\mathcal{S}_i$, where $i < m$. Then, by Induction Hypothesis 2, each predecessor of s is true in M. By Induction Hypothesis 1, M satisfies $\mathcal{C}'$. Thus, by the soundness of the unit clause rule, M satisfies s.

Case 3: $\mathcal{S}_m = \{s\}$ where s was inferred via the modus ponens inference rule from a rule

$$s \leftarrow b_1, \ldots, b_j, not\ c_1, \ldots, not\ c_k$$

in P. If $s \in D$ we are done. Otherwise, each predecessor of s in G is dominated by D. By Induction Hypothesis 2, each predecessor of s is true in M. I.e., M satisfies $b_1, \ldots, b_j, \neg c_1, \ldots, \neg c_k$. Thus, by the soundness of the modus ponens inference rule, M satisfies s.

Case 4: $\mathcal{S}_m = \{s\}$ was inferred from the backchain true inference rule.

Case 5: $\mathcal{S}_m = \{s\}$ was inferred from the backchain false inference rule.

Case 6: $\mathcal{S}_m = \{s\}$ was inferred from the all rules cancelled inference rule.

The proofs in Cases 4-6, above, are similar to Case 3.

Case 7: $\mathcal{S}_m = \{\neg a : a \in U\}$ where U was an unfounded set detected by Smodels$_{cc}$.

Let $C = \{\text{atoms } a : \neg a \text{ is dominated by } D\}$. Since s is dominated by D and $s \in \mathcal{S}_m$, we have $s = \neg a$ for some $a \in U \cap C$.

Claim 2 *$U \cap C$ is unfounded with respect to P and M.*

Proof of Claim 2: Let

$$R = \quad h \leftarrow c_1, \ldots, c_j, not\ d_1, \ldots, not\ d_k$$

be any rule in P with head $h \in U \cap C$. We will show that R satisfies at least one of conditions 1 - 3 given in the definition of an unfounded set (Definition 12).

First, suppose $\neg h \in D$. Since M agrees with D, then $h \notin M$. Since M is an answer set of P, M is a classical model of P (Lemma 5). Therefore, either $c_i \notin M$ for some $1 \leq i \leq j$, or $d_i \in M$ for some $1 \leq i \leq k$. Thus, R satisfies either condition 2 or condition 1, respectively, in the statement of Definition 12.

Otherwise, we have $\neg h \notin D$. Since $h \in C$, then $\neg h$ is dominated by D. So every predecessor of $\neg h$ in the implication graph is dominated by D. By Smodels$_{cc}$'s construction of the implication graph for the unfounded set case, there is a subgoal t in the body of R such that t cancels R and t has an edge to $\neg h$. We break this situation down into three possible cases:

Case 7.1: $t = d_i$ for some $1 \leq i \leq k$. In this case, t is a positive literal, so $t \notin \mathcal{S}_m$. (Recall that we are in Case 7 in the inductive proof of Claim 1, and $\mathcal{S}_m$ contains only negative literals in that case.) So t occurs in $\mathcal{D}$ before $\mathcal{S}_m$. Since D dominates $t = d_i$, then by Induction Hypothesis 2, M satisfies d_i. Then R satisfies condition 1 in Definition 12.

Case 7.2: $t = \neg c_i$ for some $1 \leq i \leq j$, where $c_i \notin U \cap C$. Since D dominates $t = \neg c_i$, $c_i \in C$. Therefore, $c_i \notin U$. So t occurs in $\mathcal{D}$ before $\mathcal{S}_m$. Since D dominates $t = \neg c_i$, then by Induction Hypothesis 2, M satisfies $\neg c_i$. Thus $c_i \notin M$, and R satisfies condition 2 in Definition 12.

Case 7.3: $t = \neg c_i$ for some $1 \leq i \leq j$, where $c_i \in U \cap C$. Then R satisfies condition 3 in Definition 12.

Thus $U \cap C$ is unfounded with respect to P and M. This completes the proof of Claim 2.

Returning to Case 7 of our proof of Claim 1, by Proposition 1, every element of $U \cap C$ is *false* in M. Since $s = \neg a$ for some $a \in U \cap C$, s is *true* in M.

This completes our inductive proof of Claim 1, and therefore our proof of Proposition 4. $\square$

APPENDIX D

SMODELS$_{CC}$ PROOF OF COMPLETENESS

Correctness of Smodels$_{cc}$ was proven in Appendix C. Therefore, the Smodels$_{cc}$ algorithm is *complete* if, on any problem instance, given sufficient space resources, the algorithm will eventually terminate with a result. That result may, of course, either be positive ("answer set found") or negative ("no answer set for this program exists").

The original Smodels program searches for an answer set via an adaptation of the Davis-Putnam-Loveland-Logemann (DPLL) procedure, using chronological backtracking and no restarts. The algorithm's search may be portrayed as a binary tree that the algorithm traverses until either a solution is found or until every leaf node in the tree represents a conflict. Therefore, it is not hard to see that the original Smodels algorithm always terminates.

Smodels$_{cc}$ utilizes non-chronological backtracking and restarts. Either of these two techniques could cause a DPLL-based search algorithm to go into an infinite loop, depending on how these techniques are incorporated. However, Smodels$_{cc}$ will always terminate because of the following reasons:

- The interval between restarts in Smodels$_{cc}$ is progressively increased as the search proceeds. Specifically, based on the parameters discussed in Section 4.6.4, the algorithm will wait for $50n + 500$ conflicts to occur after the n-th restart before it will perform the $(n + 1)$-st restart.

- One conflict clause is generated for each conflict in the search.
- Smodels$_{cc}$ does not delete from its conflict clause cache any conflict clause that has been generated after the most recent restart. (This aspect of the algorithm was mentioned in Section 4.6.5.)
- For any logic program, there is a limited, finite set of conflict clauses that can possibly be generated. This is clear because each conflict clause corresponds one-to-one with a particular set of literals, and the number of literals available to be used in any clause is at most $2m$, where m is the number of atoms appearing in the grounded version of the program, which itself is finite.
- Smodels$_{cc}$ will not regenerate a conflict clause $\mathcal{C}$ if that clause is currently contained in the conflict clause cache.

The final item in the above list warrants a proof. We will perform a proof by contradiction. So suppose that clause $\mathcal{C}$ is already present in the conflict clause cache when Smodels$_{cc}$ reaches its next conflict, and that Smodels$_{cc}$ rederives $\mathcal{C}$ from this conflict. The new derivation of $\mathcal{C}$ will be based on a set of nodes D that dominate the chosen pair of conflicting nodes (as discussed in Section 4.5). Exactly one element of D (specifically, the closest UIP) comes from the current search level. All other nodes in D represent assignments that were made at earlier levels in the search. Suppose that y_c is the closest UIP. Since $\mathcal{C}$ was contained in the conflict clause cache when the current conflict was detected, we know that $\mathcal{C}$ was in the cache immediately before the current search level started. The only way that Smodels$_{cc}$ would derive $\mathcal{C}$ from the set D is if $\mathcal{C} = \bigvee\{\neg y : y \in D\}$. But in that case we have all of the literals of $\mathcal{C}$, except $\neg y_c$, rendered *false* by the partial assignment that was in effect immediately before the current search level started. Hence, $\neg y_c$ would have been inferred via Boolean Constraint Propagation on clause $\mathcal{C}$ *before* the current search level. Therefore, y_c is itself a conflict node. However, Smodels$_{cc}$ adjusts its choice of conflict node specifically to avoid the possibility of the corresponding CUIP being a conflict node. (See *"Step 2: Adjust conflict node selection"* in the UIP computation algorithm in Section 4.4.2.)

Thus, it is impossible for Smodels$_{cc}$ to rederive a clause that is currently in the conflict clause cache, proving the final item stated in the above list.

Now, suppose that the Smodels$_{cc}$ algorithm goes into a search that never terminates on some finite, grounded program P. Let N be the maximum number of distinct clauses that can be expressed using the atoms from P. Eventually, after some restart, Smodels$_{cc}$ will generate $N' > N$ distinct conflict clauses using only atoms from P. This is an obvious contradiction (pigeon-hole principle). Thus, Smodels$_{cc}$ will necessarily terminate. This finishes the proof of completeness.

APPENDIX E

COMPARISON OF HAMILTONIAN CYCLE REDUCTIONS

The two reductions that we use in Chapter 6 do not seem to appear in the literature. In this appendix we justify their use by showing that they produce better solver performance than the reductions that we have tested which came from previous work.

Our Hamiltonian cycle tests were done on directed graphs. All of our logic programs for expressing these problems specified a single vertex as the initial vertex and listed the vertices and edges in the graph. Thus the extensional database for one of these problems could read as follows:

```
initialvertex(1).
vertex(1).
vertex(2).
vertex(3).
vertex(4).
vertex(5).
edge(1,2).
edge(1,4).
edge(2,4).
edge(2,5).
edge(3,1).
edge(3,2).
edge(4,1).
edge(4,3).
edge(5,4).
```

The first reduction that we tested was one given by Niemelä in [49]. We label this reduction *NNT1* in our tables (for "Normal Non-Tight 1"). The IDB for this reduction is as follows:

```
% Reduction ``NNT1''
hc(X,Y) :- edge(X,Y), not otherroute(X,Y).
otherroute(X,Y) :- edge(X,Y), hc(X,Z), Y != Z.
otherroute(X,Y) :- edge(X,Y), hc(Z,Y), X != Z.
r(Y) :- hc(X,Y), r(X).
r(Y) :- hc(X,Y), initialvertex(X).
noncircuit :- vertex(X), not r(X).
f :- not f, noncircuit.
```

The statement `otherroute(X,Y)` here means that there is a route in the cycle from node X to node Y that does not involve the edge from X to Y. Thus, `otherroute(X,Y)` is *true* if and only if `hc(X,Y)` is *false*.

The second reduction is one presented by Lin and J. Zhao [39] for testing with ASSAT. This reduction is normal. It is also "tight on its models", which implies that there is no need for ASSAT to add loop formulas during the search. It is only necessary for ASSAT (or Cmodels) to call its SAT solver once on such a problem. We named this reduction *Tight* in our tables. The IDB is:

```
% Reduction ``Tight''
outgoing(V) :- edge(V,U), hc(V,U).
incoming(U) :- edge(V,U), hc(V,U).
:- vertex(V), not outgoing(V).
:- vertex(V), not incoming(V).
hc(V,U) :- edge(V,U), not otherroute(V,U).
otherroute(V,U) :- edge(V,U), edge(V,W), hc(V,W), U != W.
otherroute(V,U) :- edge(V,U), edge(W,U), hc(W,U), V != W.
reached(V,U) :- edge(V,U), hc(V,U), not initialvertex(V).
reached(V,U) :- edge(W,U), hc(W,U), vertex(V), not initialvertex(W), reached(V,W).
:- vertex(V), reached(V,V).
```

The third reduction that we tested (labeled *NNT2*) is similar to NNT1 but separates the rules allowing edge selection and the rules that constrain how many incoming and outgoing edges a single vertex can have.

```
% Reduction ''NNT2''
% Select edges for the cycle
hc(X,Y) :- not not_hc(X,Y), edge(X,Y).
not_hc(X,Y) :- not hc(X,Y), edge(X,Y).

% Each vertex has at most one incoming edge in a cycle
:- hc(X1,Y), hc(X2,Y), edge(X1,Y), edge(X2,Y), vertex(Y), X1 != X2.

% Each vertex has at most one outgoing edge in a cycle
:- hc(X,Y1), hc(X,Y2), edge(X,Y1), edge(X,Y2), vertex(X), Y1 != Y2.

% Every vertex must be reachable from the initial vertex
% through the chosen hc edges.
:- vertex(X), not r(X).
r(Y) :- hc(X,Y), edge(X,Y), initialvertex(X).
r(Y) :- hc(X,Y), edge(X,Y), r(X), not initialvertex(X).
```

The fourth reduction is the same as NNT2 but adds two lines to the IDB which state that, in order for a set of edges to constitute a Hamiltonian cycle, every node in the graph must have an outgoing edge included in that set. This fact is implicit in reductions NNT1 and NNT2. However, stating it explicitly helps the solvers to prune their search spaces. We note that these two lines were included in reduction Tight.

```
% Reduction ''NNT3''
% The first seven rules are copied from NNT2.
hc(X,Y) :- not not_hc(X,Y), edge(X,Y).
not_hc(X,Y) :- not hc(X,Y), edge(X,Y).
:- hc(X1,Y), hc(X2,Y), edge(X1,Y), edge(X2,Y), vertex(Y), X1 != X2.
:- hc(X,Y1), hc(X,Y2), edge(X,Y1), edge(X,Y2), vertex(X), Y1 != Y2.
:- vertex(X), not r(X).
r(Y) :- hc(X,Y), edge(X,Y), initialvertex(X).
r(Y) :- hc(X,Y), edge(X,Y), r(X), not initialvertex(X).

% Rules in NNT3 but not in NNT2:
% In the chosen cycle, every vertex has
% an outgoing edge.
```

```
outgoing(V) :- edge(V,U), hc(V,U).
:- vertex(V), not outgoing(V).
```

The final reduction that we tested in this problem domain uses extended rules in the IDB. This reduction is non-tight. It is also similar to NNT3 in that it explicitly states that each vertex must have an outgoing edge included in the cycle.

```
% Reduction ``Extended''
% Each vertex has exactly one incoming edge in a cycle.
1 {hc(X,Y):edge(X,Y)} 1 :- vertex(Y).

% Each vertex has exactly one outgoing edge in a cycle.
1 {hc(X,Y):edge(X,Y)} 1 :- vertex(X).

% Every vertex must be reachable from the initial vertex
% through the chosen hc edges.
:- vertex(X), not r(X).
r(Y) :- hc(X,Y), edge(X,Y), initialvertex(X).
r(Y) :- hc(X,Y), edge(X,Y), r(X), not initialvertex(X).
```

Table E.1 compares the runtimes that we obtained on eleven randomly generated Hamiltonian cycle problems. Each problem had 120 vertices and 470 edges. Exactly eight of these graphs contained Hamiltonian cycles. The best runtimes were obtained with the *Extended* reduction. The second best reduction in this regard was *NNT3*. We could not use ASSAT with the *Extended* reduction because ASSAT does not work with extended logic programs. It is somewhat mysterious that Cmodels-2 performed considerably worse than ASSAT on the NNT1 and NNT2 reductions.

Besides runtime performance, another issue is that different reductions may produce different sized groundings. Table E.2 compares the average size of the grounding produced by Lparse on the above mentioned Hamiltonian problem instances under each of the aforementioned reductions. Also reported is the average number of atoms mentioned in each grounding. Here again, the *Extended* reduction produced the best

Median and max secs on 11 uniform HC problems with 120 vertices, 470 edges

Reduction	Smodels lookahead		Smodels_{cc} no lookahead		ASSAT		Cmodels-2	
	med	max	med	max	med	max	med	max
NNT1	0.32	2 abort	1.23	2 abort	45.15	2 abort	>3600	8 abort
NNT2	0.37	2 abort	0.96	2 abort	35.69	2 abort	>3600	7 abort
Tight	168.83	220.32	23.22	48.07	2.74	3.12	3.86	4.40
NNT3	0.33	0.44	0.17	0.21	0.48	0.74	0.34	0.65
Extended	0.14	0.19	0.11	0.14	na	na	0.19	0.26

Table E.1: Runtime comparison of different Hamiltonian Cycle reductions

Groundings produced by Lparse 120 node, 470 edge HC problems

Reduction	Size (KB)	Atoms
NNT1	114	1653
NNT2	128	1652
Tight	1795	16172
NNT3	140	1772
Extended	63	1662

Table E.2: Grounding comparison for different HC reductions

results. It was followed closely by the other non-tight reductions. The *Tight* reduction had the worst results in this regard. Table E.3 shows the grounding sizes produced with the various reductions on a single randomly generated problem with 1000 nodes and 3500 edges. In this case, the *Tight* reduction resulted in a grounding that was too large to feed in to any of the solvers. However, with the NNT3 reduction the problem was easily solved by each of the solvers.

As a result of these tests, in Section 6.3 we use reduction *NNT3* in comparisons that involve ASSAT and *Extended* in comparisons that do not involve ASSAT.

Lparse groundings on a 1000 node, 3500 edge HC problem

Reduction	Size (KB)
NNT1	885
NNT2	986
Tight	122966
NNT3	1086
Extended	543

Table E.3: Grounding comparison for a larger HC problem

APPENDIX F

EXPERIMENTAL RESULTS WITH LOOKAHEAD OPTIONS IN SMODELS AND SMODELS$_{CC}$

In this appendix we compare the performance of Smodels, with and without lookaheads, to Smodels$_{cc}$, also with and without lookaheads. Each of these tables corresponds to a table given in Chapter 6. The problem sets were explained more fully in that chapter.

In nearly every one of these test domains, it turned out that the original Smodels program performed better with lookaheads enabled, while Smodels$_{cc}$ performed better with lookaheads disabled. The one exception to this was the Hamiltonian cycle problems with a uniform distribution of edges (Table F.3). In this domain, Smodels performed better without lookaheads.

Part of the reason for reporting these results is to show that, on those tests in Chapter 6 where Smodels$_{cc}$ showed better performance than the original Smodels, the performance increase was not due merely to the fact that Smodels$_{cc}$ was avoiding the expense of using lookaheads. That is, the use of conflict clauses was integral to obtaining those speedups.

All logic programs used in these tests were normal, except the bounded model checking problems.

Median and maximum seconds on 11 random 3-SAT problems

Data Set		Smodels lookahead		Smodels no lookahead		Smodels$_{cc}$ lookahead		Smodels$_{cc}$ no lookahead		#AS
vars	clauses	median	max	median	max	median	max	median	max	
250	1025	16.5	112.2	>3600	11 abort	88.0	1 abort	36.5	934.5	10
250	1050	86.1	207.2	>3600	11 abort	>3600	6 abort	657.5	2221.5	6
250	1075	133.4	376.6	576.4	1 abort	>3600	6 abort	641.7	1 abort	3
250	1100	74.4	169.8	432.8	2110.8	>3600	6 abort	343.7	2197.6	1

Median and maximum seconds on 8 DLX benchmark problems

Data Set	Smodels lookahead		Smodels no lookahead		Smodels$_{cc}$ lookahead		Smodels$_{cc}$ no lookahead		#AS
	median	max	median	max	median	max	median	max	
8 satisfiable	>3600	8 abort	>3600	8 abort	>3600	8 abort	9.9	22.6	8
8 unsatisfiable	>3600	7 abort	>3600	8 abort	2064.8	3330.5	15.2	45.2	0

Table F.1: Lookahead results on satisfiability problems

Median and max secs on 11 random 3-coloring problems, uniform distribution of edges

Data Set		Smodels lookahead		Smodels no lookahead		Smodels$_{cc}$ lookahead		Smodels$_{cc}$ no lookahead		#AS
vertices	edges	median	max	median	max	median	max	median	max	
400	900	3.4	163.0	>3600	11 abort	21.6	2743.0	2.9	181.1	10
400	950	99.5	511.8	>3600	11 abort	1122.0	3 abort	94.9	1463.4	2
400	1000	20.7	134.3	>3600	11 abort	144.1	493.0	16.4	53.8	0
500	1100	0.8	4.2	>3600	11 abort	4.0	42.6	1.7	3.9	11
500	1150	196.0	356.6	>3600	11 abort	1565.8	5 abort	66.1	1 abort	10
500	1200	2753.9	5 abort	>3600	11 abort	>3600	11 abort	>3600	8 abort	0-5

Median and max secs on 11 random 3-coloring problems, clumpy distribution of edges

Data Set		Smodels lookahead		Smodels no lookahead		Smodels$_{cc}$ lookahead		Smodels$_{cc}$ no lookahead		#AS
vertices	edges	median	max	median	max	median	max	median	max	
10000	15150	256.1	1 abort	560.2	5 abort	726.5	825.8	21.4	45.2	10
10000	17170	201.6	3 abort	>3600	6 abort	577.6	628.3	19.7	21.0	7
10000	19190	>3600	8 abort	>3600	9 abort	308.3	2978.4	8.0	222.6	2

Table F.2: Lookahead results on coloring problems

Median and max secs on 11 random Hamiltonian cycle problems, uniform distribution of edges

Data Set		Smodels lookahead		Smodels no lookahead		Smodels$_{cc}$ lookahead		Smodels$_{cc}$ no lookahead		#AS
vertices	edges	median	max	median	max	median	max	median	max	
1000	4000	1.0	1.1	0.9	0.9	1.0	1.0	1.0	1.0	0
1000	4500	1.2	132.0	1.1	828.5	1.2	289.4	1.2	4.7	4
1000	5000	172.7	201.0	1.6	29.1	384.6	439.6	3.5	4.2	6
1000	5500	244.6	269.5	2.7	30.6	577.3	591.8	4.6	5.5	10
6000	30000	8.4	3 abort	7.6	2 abort	8.0	3 abort	8.0	51.0	3
6000	33000	>3600	9 abort	525.8	5 abort	>3600	9 abort	55.9	64.2	9
6000	36000	>3600	7 abort	19.8	4 abort	>3600	7 abort	61.5	73.7	7
6000	39000	>3600	10 abort	1128.4	4 abort	>3600	10 abort	72.7	97.0	10

Table F.3: Lookahead results on uniform Hamiltonian cycle problems

Median and max secs on 11 Hamiltonian cycle problems, clumpy distribution of edges

Data Set		Smodels lookahead		Smodels no lookahead		Smodels$_{cc}$ lookahead		Smodels$_{cc}$ no lookahead		#AS
# of clumps	vertices / clump	median	max	median	max	median	max	median	max	
10	10	0.4	1 abort	4.8	1 abort	1.7	7.3	0.4	0.9	11
12	12	620.9	4 abort	315.9	5 abort	5.0	38.1	0.7	1.3	11
14	14	>3600	10 abort	>3600	8 abort	90.0	437.0	1.8	10.4	11
16	16	>3600	9 abort	>3600	11 abort	741.8	3 abort	8.8	19.2	11
18	18	>3600	11 abort	>3600	11 abort	>3600	7 abort	21.3	156.0	11
20	20	>3600	11 abort	>3600	11 abort	>3600	6 abort	102.4	264.4	11

Table F.4: Lookahead results on clumpy Hamiltonian cycle problems

Bounded model checking runtimes in secs
(Deadlock checking under "step" semantics)

Problem	Smodels lookahead	Smodels no lookahead	Smodels$_{cc}$ lookahead	Smodels$_{cc}$ no lookahead	AS?
dp-6.s-O2-b1	0.00	0.00	0.01	0.00	Yes
dp-8.s-O2-b1	0.01	0.01	0.01	0.00	Yes
dp-10.s-O2-b1	0.00	0.00	0.00	0.00	Yes
dp-12.s-O2-b1	0.00	0.01	0.00	0.00	Yes
key-2.s-O2-b25	489.58	>3600.00	20.18	6.18	No
mmgt-3.s-O2-b7	3.01	51.78	2.07	0.34	Yes
mmgt-4.s-O2-b8	163.71	>3600.00	14.92	2.38	Yes
q-1.s-O2-b9	0.06	90.26	0.12	0.08	Yes
dartes-1.s-O2-b32	0.69	0.53	0.99	0.96	Yes
elev-1.s-O2-b4	0.03	0.02	0.03	0.02	Yes
elev-2.s-O2-b6	0.14	0.19	0.18	0.11	Yes
elev-3.s-O2-b8	1.49	18.17	2.50	1.05	Yes
elev-4.s-O2-b10	44.67	649.42	38.88	16.61	Yes
hart-25.s-O2-b1	0.01	0.00	0.01	0.01	Yes
hart-50.s-O2-b1	0.01	0.01	0.01	0.01	Yes
hart-75.s-O2-b1	0.02	0.02	0.02	0.01	Yes
hart-100.s-O2-b1	0.02	0.02	0.02	0.03	Yes
Total	703.45	>8010.44	79.95	27.79	

Table F.5: Lookahead results on Bounded Model Checking - step semantics

Bounded model checking runtimes in secs
(Deadlock checking under "interleaving" semantics)

Problem	Smodels lookahead	Smodels no lookahead	Smodels$_{cc}$ lookahead	Smodels$_{cc}$ no lookahead	AS?
dp-6.i-O2-b6	0.03	0.05	0.07	0.04	Yes
dp-8.i-O2-b8	0.09	1.52	0.22	0.18	Yes
dp-10.i-O2-b10	0.93	191.13	7.67	208.51	Yes
dp-12.i-O2-b12	150.77	>3600.00	1336.33	3.71	Yes
key-2.i-O2-b26	16.11	>3600.00	20.52	9.13	No
mmgt-3.i-O2-b10	23.81	91.15	103.98	3.24	Yes
mmgt-4.i-O2-b11	518.54	>3600.00	>3600.00	685.95	No
q-1.i-O2-b17	731.58	>3600.00	1094.29	116.74	No
dartes-1.i-O2-b32	0.47	0.42	0.75	0.76	Yes
elev-1.i-O2-b9	0.20	0.76	0.64	0.39	Yes
elev-2.i-O2-b12	2.10	1416.73	10.81	5.33	Yes
elev-3.i-O2-b15	90.83	>3600.00	635.40	27.50	Yes
elev-4.i-O2-b13	593.83	>3600.00	275.33	53.72	No
hart-25.i-O2-b5	0.23	0.12	0.32	0.08	No
hart-50.i-O2-b5	1.12	1.46	2.15	0.25	No
hart-75.i-O2-b5	3.51	9.37	4.26	0.54	No
hart-100.i-O2-b5	7.82	10.91	8.02	0.97	No
Total	2141.97	>23323.61	>7100.76	1117.04	

Table F.6: Lookahead results on Bounded Model Checking - interleaving semantics

BIBLIOGRAPHY

[1] Fahiem Bacchus. Enhancing Davis Putnam with extended binary clause reasoning. In *Eighteenth national conference on Artificial intelligence*, pages 613–619. American Association for Artificial Intelligence, 2002.

[2] Marcello Balduccini, Michael Gelfond, Richard Watson, and Monica Nogueira. The USA-Advisor: A case study in answer set planning. In Eiter et al. [14], pages 439 – 442.

[3] Roberto J. Bayardo Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence, Rhode Island, 1997.

[4] Paul Beame, Henry Kautz, and Ashish Sabharwal. Understanding the power of clause learning. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI)*, Acapulco, Mexico, August 2003.

[5] Rachel Ben-Eliyahu and Rina Dechter. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1996.

[6] Armin Biere. Source code and notes on limmat are available at *http://www2.inf.ethz.ch/personal/biere/projects/limmat/*.

[7] Keith L. Clark. Negation as failure. In Herve Gallaire, Jack Minker, and Jean M. Nicolas, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[8] James M. Crawford and Larry D. Auton. Experimental results on the crossover point in random 3-SAT. *Artificial Intelligence*, 81(1-2):31–57, 1996.

[9] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[10] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[11] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of the Fourth European Conference on Planning*, pages 169 – 181. Springer–Verlag, 1997.

[12] William Dowling and Jean Gallier. Linear time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 3:267–284, 1984.

[13] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative problem-solving using the DLV system. In *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

[14] Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczynski, editors. *Proceedings of 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, Vienna, Austria, September 2001.

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

[16] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Experimenting with heuristics for answer set programming. In Nebel [48], pages 635 – 640.

[17] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Optimizing the computation of heuristics for answer set programming systems. In Eiter et al. [14], pages 295 – 308.

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

[19] Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming.* To appear.

[20] Allen Van Gelder, Kenneth Ross, and John S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[21] Michael Gelfond, Marcello Balduccini, and Joel Galloway. Diagnosing physical systems in A–Prolog. In Eiter et al. [14], pages 213 – 225.

[22] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*, pages 1070 – 1080, 1988.

[23] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and deductive databases. *New Generation Computing*, 9:365 – 385, 1991.

[24] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. SAT-based answer set programming. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI2004)*, pages 61–66, San Jose, California, July 2004.

[25] Enrico Giunchiglia, Marco Maratea, and Armando Tacchella. SIMO code is available with the Cmodels-2 distribution, or from *http://www.mrg.dist.unige.it/˜sim/simo/*.

[26] Enrico Giunchiglia, Marco Maratea, and Armando Tacchella. (In)Effectiveness of look-ahead techniques in a modern SAT solver. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*, 2003.

[27] Evgueni Goldberg and Yakov Novikov. BerkMin: a fast and robust Sat–solver. In *Design Automation and Test in Europe (DATE)*, pages 142–149, 2002.

[28] Keijo Heljanko and Ilkka Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming*, 3(4-5):519–550, 2003.

[29] Tomi Janhunen. A counter-based approach to translating normal logic programs into sets of clauses. In *Proceedings of ASP'03 Workshop*, pages 166–180, 2003.

[30] Tomi Janhunen, Ilkka Niemelä, Patrik Simons, and Jia-Huai You. Unfolding partiality and disjunctions in stable model semantics. In *Principles of Knowledge Representation and Reasoning: Proceedings of the 7th International Conference*, pages 411–419, Breckenridge, Colorado, 2000.

[31] Oliver Kullmann. Code and documentation for OKSolver is available from http://www.cs.swan.ac.uk/˜csoliver/OKsolver.html.

[32] Oliver Kullmann. On the use of autarkies for satisfiability decision. In Henry Kautz and Bart Selman, editors, *Electronic Notes in Discrete Mathematics*, volume 9. Elsevier Science Publishers, 2001.

[33] Joohyung Lee and Vladimir Lifschitz. Loop formulas for disjunctive logic programs. In Catuscia Palamidessi, editor, *ICLP*, volume 2916 of *Lecture Notes in Computer Science*, pages 451–465. Springer, 2003.

[34] Nicola Leone, Riccardo Rosati, and Francesco Scarcello. Enhancing answer set planning. In *Proceedings of the IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[35] Chu Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Principles and Practice of Constraint Programming - CP97, Linz, Austria*, pages 341–355, Linz, Austria, October/November 1997.

[36] Yuliya Lierler and Marco Maratea. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In Lifschitz and Niemelä [38], pages 346–350. System description.

[37] Yuliya (Babovich) Lierler. Documentation and code for Cmodels-1 is available from *http://www.cs.utexas.edu/users/tag/cmodels.html.*

[38] Vladimir Lifschitz and Ilkka Niemelä, editors. *Logic Programming and Non-monotonic Reasoning, 7th International Conference (LPNMR)*, Fort Lauderdale, Florida, January 2004. Springer.

[39] Fangzhen Lin and Jicheng Zhao. On tight logic programs and yet another translation from normal logic programs to propositional logic. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJ-CAI)*, Acapulco, Mexico, August 2003.

[40] Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-2002)*, pages 112–117. American Association for Artificial Intelligence, 2002.

[41] Thomas Linke. Graph theoretical characterization and computation of answer sets. In Nebel [48], pages 641 – 646.

[42] Inês Lynce and Joao Marques-Silva. Efficient data structures for backtrack search SAT solvers. In *5th International Symposium on the Theory and Applications of Satisfiability Testing (SAT'02)*, May 2002.

[43] Wiktor Marek and Miroslaw Truszczynski. Autoepistemic logic. *Journal of the ACM*, 38(3):588 – 619, 1991.

[44] Joao P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[45] David G. Mitchell, Bart Selman, and Hector J. Levesque. Hard and easy distributions for SAT problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–465, Menlo Park, California, 1992. AAAI Press.

[46] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff source code and documentation is available from *http://www.princeton.edu/˜chaff.*

[47] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Conference on Design Automation*, pages 530–535. ACM Press, 2001.

[48] Bernhard Nebel, editor. *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

[49] Ilkka Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999.

[50] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241 – 273, 1999.

[51] Ilkka Niemelä, Patrick Simons, and Tommi Syrjänen. Smodels: a system for answer set programming. In *Proceedings of the Eighth International Workshop on Non-Monotonic Reasoning*, 2000.

[52] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An A-Prolog decision support system for the space shuttle. In I. V. Ramakrishnan, editor, *PADL*, volume 1990 of *Lecture Notes in Computer Science*, pages 169 – 183. Springer–Verlag, 2001.

[53] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26, 1996.

[54] Patrick Simons. *Extending and Implementing the Stable Model Semantics.* PhD thesis, Helsinki University of Technology, 2000.

[55] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[56] Clifford Spector. Recursive well-orderings. *Journal of Symbolic Logic*, 20(2):151–163, 1955.

[57] Richard M. Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[58] Tommi Syrjänen. Lparse 1.0 user's manual. Available from *http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz*.

[59] Tommi Syrjänen. Omega-restricted logic programs. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, Vienna, Austria, September 2001. Springer-Verlag.

[60] Robert Tarjan. Finding dominators in directed graphs. *SIAM Journal of Computing*, 3(1):62–89, 1974.

[61] Miroslav Velev and Randy Bryant. Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic. In *Correct Hardware Design and Verification Methods (CHARME'99)*, 1999.

[62] Jeffrey Ward and John S. Schlipf. Answer set programming with clause learning. In Lifschitz and Niemelä [38], pages 302–313.

[63] Hantao Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE)*, pages 272–275, 1997.

[64] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, pages 279–285. IEEE Press, 2001.