University of Alberta

Logic Programming with stable models for constraint satisfaction

by

Srinivas Padmanabhuni ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta

Spring 2000

0-612-60011-4

University of Alberta

Library Release Form

Name of Author: Srinivas Padmanabhuni

Title of Thesis: Logic Programming with stable models for constraint satisfaction

Degree: Doctor of Philosophy

Year this Degree Granted: 2000

[illegible]

Srinivas Padmanabhuni

B-5/B, R. E. College
Durgapur, WB
India, 713209

Date: January. 28 2000

No day in which you learn something

is a complete loss.

by David Eddings

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Logic Programming with stable models for constraint satisfaction** submitted by Srinivas Padmanabhuni in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

[illegible]

J. H. You

Peter van Beek
for

M. Maher

[illegible]

R. Goebel

Peter van Beek

L. Y. Yuan

[illegible]

P. van Beek

[illegible]

J. F. Forbes

Date: January 28 2000

To my parents

Prof. P. V. Rama Rao

and

Smt. P. Raghupatamma

Abstract

Logic Programming with the stable model semantics, called stable logic programming (SLP), has been suggested as a new paradigm for solving a number of computationally hard problems in artificial intelligence, including constraint satisfaction problems (CSPs), planning problems and scheduling problems. The expressiveness of SLP as a general knowledge representation language is in sharp contrast with the conventional problem solvers that work only for special domains. The promise of this new paradigm has recently been demonstrated by efficient implementations of SLP systems, among which the **smodels** system developed by Niemela et al. is the most competitive. Niemela demonstrated that **smodels** successfully competes with special-purpose solvers for a class of planning problems. But to date, very few studies have been carried out for **smodels** in context of CSPs which form the basis of some of the most successful practical industrial-scale systems in artificial intelligence. Further, the promise of SLP can be strengthened by the development of strong and more powerful pruning techniques for stable model computation. One is unlikely to succeed in this direction without an understanding of the techniques employed by the efficient existing implementations, and how they relate to the pruning techniques we understand for other domains. Though the efficiency of **smodels** is apparently attributed to some of well-known techniques in solving CSPs (and SAT problems) so far no comprehensive studies of such implementations has been performed.

In this thesis, we study the important techniques incorporated in the implementation of **smodels**. We show that the three main techniques used in **smodels**, namely, constraint propagation, **lookahead**, and **backjumping**, are mappings from well-

known efficient techniques in CSPs. It turns out that **smodels** with **lookahead** can compete successfully with the best CSP algorithms.

An interesting yet challenging question in constraint satisfaction is what if a CSP is inconsistent, i.e., it does not have a solution in which all the given constraints are satisfied. These problems are called over-constrained problems. Our study extends to these problems. Research in the direction of representation and solving of over-constrained problems necessitated a clear understanding of their semantics and complexity. However the inadequate treatment of the semantic notion of solution in over-constrained problems in the literature prompted us to first explore their semantics. In the latter part of the thesis we studied the semantical problems with the notion of solution in over-constrained problems. We find that the existing notions of solutions in over-constrained problems suffer from the following semantic problems:

1. ad-hoc semantics;
2. higher computational complexity;
3. semantics not preserved in translation from non-binary to binary representations; and
4. techniques used in solving finite CSPs cannot be used directly.

Acknowledgements

I would like to first thank my dear wife Neelima for her consistent support and patience. I would also like to thank my parents and my friends for all their encouragement and endurance.

Thanks to my supervisor Prof. You for his advice not only on technical aspects but also on effective writing. Thanks to my supervisory committee members for their guidance. Thanks to Prof. Goebel for the constant encouragement and technical feedback he provided.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

Logic Programming with the stable model semantics, called stable logic programming (SLP), has been suggested as a new paradigm for solving a number of computationally hard problems in artificial intelligence, including constraint satisfaction problems (CSPs), planning problems and scheduling problems. The expressiveness of SLP as a general knowledge representation language is in sharp contrast with the conventional problem solvers that work only for special domains. The promise of this new paradigm has recently been demonstrated by efficient implementations of SLP systems, among which the **smodels** [54, 41] system developed by Niemela et al. is the most competitive. Niemela demonstrated that **smodels** successfully competes with special-purpose solvers for a class of planning problems.

Now we discuss the major motivations for studying the relationship between the techniques employed in **smodels** and finite CSP techniques.

1. CSPs have found strong practical and industrial applications due to the development of a *core* set of efficient techniques. It is natural to see if these techniques can be generalized to implementations of logic programs yielding efficient implementations.

2. The promise of SLP can be strengthened by the development of strong and more powerful pruning techniques for stable model computation. One is unlikely to succeed in this direction without an understanding of the techniques employed

by the efficient existing implementations, and how they relate to the pruning techniques we understand for other domains.

3. In the literature, many instances are known where a general purpose solver is shown to be as powerful as highly domain-specific solver. E.g. a domain-independent planner like SATPLAN [30] has been shown to be as efficient as some domain dependent planners. On account of the ease of representation of problems in these general purpose solvers, it is preferable to use these general purpose solvers in the face of comparable efficiency. In case of **smodels**, if it can be shown to be comparable to some efficient CSP solving techniques, it can succeed as a general purpose solver.

4. Comprehensive performance studies of existing implementations of logic programming is required so as to gather insights into the computational nature of stable models. One possible manner in which this can be achieved is by performance analysis of these systems on special classes of problems like CSPs, planning problems etc.

5. Though the efficiency of **smodels** is apparently attributed to some of well-known techniques in solving CSPs so far no comprehensive studies of such techniques has been performed.

In this thesis, we study the important techniques incorporated in the implementation of **smodels**. We show that the three main techniques used in **smodels**, namely, constraint propagation, **lookahead**, and **backjumping**, are mappings from well-known efficient techniques in CSPs. Our investigation of these techniques was two-fold: (i) to measure the relative effectiveness of each technique in terms of contribution to the efficiency of **smodels**, and (ii) to conduct a preliminary comparison of the efficiency of these techniques in **smodels** with the corresponding techniques in finite CSPs. This study was performed in the context of logic programs modeling finite

CSPs. Our investigation reveals that **lookahead** dominates the other two techniques in **smodels**, and that **smodels** employing **lookahead** can be as efficient as some of the most efficient techniques in finite CSPs. It thereby corroborates our contentions that CSP techniques can be generalized to provide efficient implementations of more general knowledge representation schemes and that general purpose solvers can be as efficient as domain specific problem solvers.

An interesting yet challenging question in constraint satisfaction is what if a CSP is inconsistent, i.e., it does not have a solution in which all the given constraints are satisfied. These problems are called over-constrained problems. Our study extends to these problems. Research in the direction of representation and solving of over-constrained problems necessitated a clear understanding of their semantics and complexity. However the inadequate treatment of the semantic notion of solution in over-constrained problems in the literature prompted us to first explore their semantics. Since time only permits us to investigate the computational and semantic difficulties with the standard notions of solution in the over-constrained context, our goal is only partially achieved. We studied the semantical problems with the notion of solution in over-constrained problems. We find that the existing notions of solutions in over-constrained problems suffer from the following semantic problems:

1. ad-hoc semantics;
2. higher computational complexity;
3. semantics not preserved in translation from non-binary to binary representations; and
4. techniques used in solving finite CSPs cannot be used directly.

Overall, the stress of this thesis is two-fold: (i) To push forward an efficient logic programming system with stable models as an expressive and efficient constraint programming paradigm; and (ii) To study the semantics of over-constrained problems.

1.1 Constraints

In the broadest of all definitions, a *constraint* refers to a relation that must be satisfied. In the domain of artificial intelligence, various definitions and formats of constraints have been used and proposed. In the recent past, the idea of programming with constraints has found a broad sense of acceptance in terms of practical problem solving. The loose definition of constraint, as given above, can be interpreted in a wide range of ways and a wide variety of formats, depending upon the application. In the seminal paper [33] Mackworth writes:

A constraint can be taken to mean a relation over a Cartesian product of sets, a Boolean predicate, a fuzzy relation, a continuous figure of merit analogous to energy, an algebraic equation, an inequality, a Horn clause in Prolog, and various other arbitrarily complex symbolic relationships.

From the above statement, it is clear that constraints have a wide applicability in terms of application domains, and consequently a diverse range of representations depending upon the domain of applicability. In many problem solving situations where problems have been modeled using constraints, it has been found that it is impossible to find a solution satisfying all the constraints. In such situations, the problems have been termed as *over-constrained problems* [39].

Over-constrained problems suffer from additional computational complexity as compared to finite CSPs due to the need to accommodate *preference* information. In over-constrained problems, a variety of extra-logical measures are used to specify the *preference* information. Some of the common measures used in over-constrained systems are weights, priorities, and hierarchies. In all such measures there is the notion of a *preferred* solution in contrast to other possible solutions. Due to the diversity of extra-logical measures in identifying *preferred* solutions, no clear semantics of a solution has been presented in the literature of over-constrained systems.

1.2 Logic programming with stable models

Ground logic programs with *default negation* [24, 64, 23] consist of clauses of the form

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

where $a_1, \ldots, a_n, b_1, \ldots, b_m$, and h are propositional or instantiated ground first-order atoms.

Ground horn logic programs (HLP) are a special sub-class of the above class of logic programs which represent a collection of clauses of the form

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

where $a_1, \ldots, a_n$, and h are propositional or instantiated ground first-order atoms. HLPs do not have any default negation *not*. HLPs are monotonic (addition of new knowledge does not decrease the set of conclusions) and they admit a unique intended model (the least model) which can be computed in polynomial-time [60]. E.g. if the Horn logic program P consists of the set of clauses $\{a \leftarrow; b \leftarrow a\}$ the unique intended model of P is $\{a, b\}$.

In contrast, logic programs with *default negation* (*not*) behave non-monotonically, i.e. the addition of new knowledge can decrease the set of conclusions. Diverse semantics [24, 64, 23] have been proposed to account for the meaning of logic programs with default negation. The most important among them being the *well-founded semantics* [23], the *stable model semantics* [24], and the *regular model semantics* [64].

The stable model semantics of logic programs with default negation is defined in terms of the unique model of the HLP generated by applying a transform to the original logic program. The transformed HLP (*reduct*) P_A of a logic program P with respect to a set of atoms A is the program obtained by first deleting each clause in P that has a not-atom *not* x in its body such that $x \in A$ and then deleting all not-atoms in the remaining clauses. The stable model semantics [24] for a general logic program P is then defined as follows: A set of atoms A is a *stable model* of P if and only if A is the same as the unique model of the HLP P_A. Consider $P = \{a \leftarrow not\ b; b \leftarrow not\ a\}$. Take the set of atoms $A = \{a\}$. The *reduct* P_A is $\{a \leftarrow\}$, whose unique model is $\{a\}$. Thus $\{a\}$ is a stable model of the program P.

The decision problem of finding whether a ground logic program has a stable model is known to be computationally hard (NP-complete) [36]. Logic programming with stable model semantics is however known to be a powerful knowledge representation paradigm with relations to many non-monotonic reasoning frameworks including autoepistemic logic [65], and default logic [49]. Conventionally inefficiency arguments have been used against non-monotonic formalisms as the main reason for the relatively fewer implementations of practical systems. Recently though some researchers have developed efficient implementations of non-monotonic systems [7, 42, 45], thereby enabling the massive amount of research that has gone into non-monotonic reasoning to be put to practical use.

1.3 Thesis layout

Chapter 2 reviews some of the concepts in constraint satisfaction, over-constrained problems, and non-monotonic logic programming based on stable models. Commonly used finite constraint satisfaction techniques are presented in detail. Some of the extra-logical measures used in identifying *preferred* solutions in over-constrained problems are also explained. LP_{SM} (a non-monotonic logic programming language based on stable models), and **smodels** (the underlying stable model computation engine of LP_{SM}) [41, 42, 54] are introduced. Towards the end a firm mapping between some of the existing techniques in solving the SAT problem and **smodels** techniques have been established with a view to providing an idea for the motivational basis for the implementation of **smodels** techniques.

Chapter 3 examines how LP_{SM} and **smodels** can be used to model and solve constraints. It is shown how the language of LP_{SM} shares the declarative nature as other constraint programming languages like Oz, Eclipse,OPL etc. Some unique representational features of LP_{SM} was also shown which highlights the advantages of using LP_{SM}. Finite CSPs are first modeled in LP_{SM} and then directly in **smodels**, and their relative efficiency measured.

Chapter 4 explores the question: Can CSP techniques be generalized to provide an efficient implementation of logic programs? This is studied by showing how the techniques used in implementation of **smodels** map to some of the most efficient techniques used in finite CSPs. A relative measurement of the effectiveness of **smodels** techniques is done by experiments on logic programs representing finite CSPs. In addition, the techniques in **smodels** are compared experimentally with the corresponding techniques in CSP. Further the relationship between the techniques in **smodels** and their corresponding equivalent techniques in finite CSPs is explored along the following three directions: (i) by showing sufficient conditions under which the technique coincides in semantics in CSP and **smodels**, (ii) by exploring the possibility of improvement of the **smodels** technique based on the special structure of the logic program, and (iii) by experimental comparison of the average performance of the two corresponding techniques on random CSPs. The technique of **lookahead** turns out to be the dominant of all the **smodels** techniques. In fact, **smodels** with **lookahead** turns out to be competitive to some of the best techniques in the finite CSP literature.

Chapter 5 provides a critique of over-constrained semantics and solution methods. It begins with a computational complexity-theoretic treatment of the various notions of priority in over-constrained systems. In the same chapter, it is shown that semantics of maximal constraint satisfaction problem is not preserved in translation from non-binary format to binary format of constraint representation. In the end, various intelligent backtracking based proof procedures for finite CSPs are modified to work for some of these prioritized over-constrained systems, and some related theoretical results elucidated. The material presented in this chapter is independent of the earlier chapters and hence can be read separately.

Chapter 6 concludes with a summary of the results in the thesis and a study of the areas where there is a scope for further research.

Chapter 2

Background

In this chapter, some of the basic concepts in the area of non-monotonic reasoning and constraint programming are presented. From constraint programming, concepts in the area of finite constraint satisfaction are reviewed. The most common finite constraint satisfaction techniques are explained with suitable illustrative examples. Next, common existing semantic notions of *preferred* solution in over-constrained systems are explained. Finally the non-monotonic logic programming language LP_{SM} [41] (based on stable model semantics [24]) is reviewed. In particular, the **smodels** proof procedure for deducing stable models of a ground program is explained in detail. In the end, some related algorithms in literature which form the basis of the **smodels** procedure are reviewed.

2.1 Constraint satisfaction

In this section, some basic concepts of constraint satisfaction are reviewed. In particular, the framework of finite constraint satisfaction is explained, and the common finite constraint satisfaction algorithms and techniques are illustrated.

A finite *constraint satisfaction problem* (CSP) involves a set $X = \{x_1, x_2, \ldots, x_n\}$ of n variables, and a set C of constraints, where each variable $x_i \in X$ takes a value from its finite domain d_i and each constraint $c_i \in C$, defined on a subset X_c of the set of variables X, is a relation expressed as a subset R_c of the cartesian product

$\Pi_{x_i \in X_c}[d_i]$. Each constraint c_i specifies the admissible combinations of the values of variables involved in the constraint. An assignment A of values to variables in $Y \subset X$ satisfies a constraint c such that $X_c \subset Y$ iff the tuple formed by X_c in A belongs to R_c, the relation associated with c. An assignment of values to a subset Y of the variables is consistent iff it satisfies all the constraints c such that $X_c \subset Y$. A *solution* is a consistent assignment of values to all the CSP variables. A CSP is consistent iff it has at least one solution.

There are a variety of algorithms to compute a solution of a given CSP. The two common techniques present in any such algorithm are:

1. Backtracking

2. Constraint Propogation

Many of them are variable manipulation algorithms, i.e., they either change the domains of the variables or assign values to the variables.

Chronological backtracking and its variants, are based on the idea of consistently assigning values to variables and expanding a partial solution till a dead-end is reached and then backtracking in case of such dead-ends. The intelligent versions of backtracking are endowed with more ways of pruning the search space than the chronological backtracking algorithm.

The pure constraint propagation algorithms on the other hand, are based on the idea of the altering of domains, based on a certain consistency criterion. The idea is to manipulate the domains of the variables, so as to bring the global state of the CSP to a consistent one.

But most commonly, the constraint propagation is interleaved with backtracking to get different algorithms based on the level of propagation used in the algorithm alongside backtracking. In certain cases, a partial assignment is fully propagated through the CSP, and in certain cases it is propagated in a limited sense. A relative comparison between the overhead of computation involved in the application of propagation, and the pruning achieved by propagation is used as a guide to determine the best algorithm to be used.

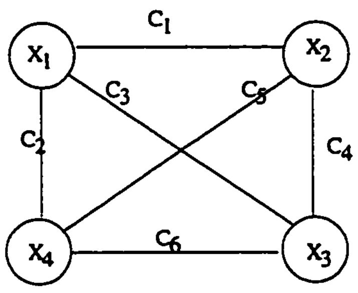
In the next section, the chronological backtracking algorithm and its intelligent variants are reviewed.

2.1.1 Chronological backtracking

The most naive of all the backtracking algorithms is the chronological backtracking (BT) algorithm. In chronological backtracking, the variables are assigned in a fixed order, which is not changed during the process of backtracking. The crux of chronological backtracking is as follows: A partial assignment of values is consistently expanded by instantiating a new variable x_h with a value v_h. This new value v_h assigned to x_h, is checked against all previously instantiated variables ($\{x_i \mid i < h\}$), to see iff it is consistent against each such variable. This value assignment is said to be consistent if it is consistent against all the previous variables. If it is consistent, then the next variable in the order is instantiated with a new value from its domain. But if that particular value v_h of x_h fails in its consistency check against at least one previous variable, the next value from the domain of x_h is assigned to x_h. In the case of all domain values of x_h being exhausted the algorithm backtracks to the previous variable, and instantiates it with the next available value from its domain. If a consistent assignment of a value is made to x_n, the last variable, then a *solution* is said to be found. An example of the search tree traversed by chronological backtracking for a finite CSP is shown in Figure 2.1.

2.1.2 Improving chronological backtracking

In general, chronological backtracking is not efficient, the reason being the unnecessary consistency checks performed in the blind process of backtracking to the most recently instantiated variable. But the cause of conflict with the current variable may have been an earlier variable in case of which any further instantiation of this



Constraint Graph of the CSP

Constraints in the CSP:

$C_1(X_1,X_2) = \{(b,c),(a,a),(a,b),(a,c)\}$

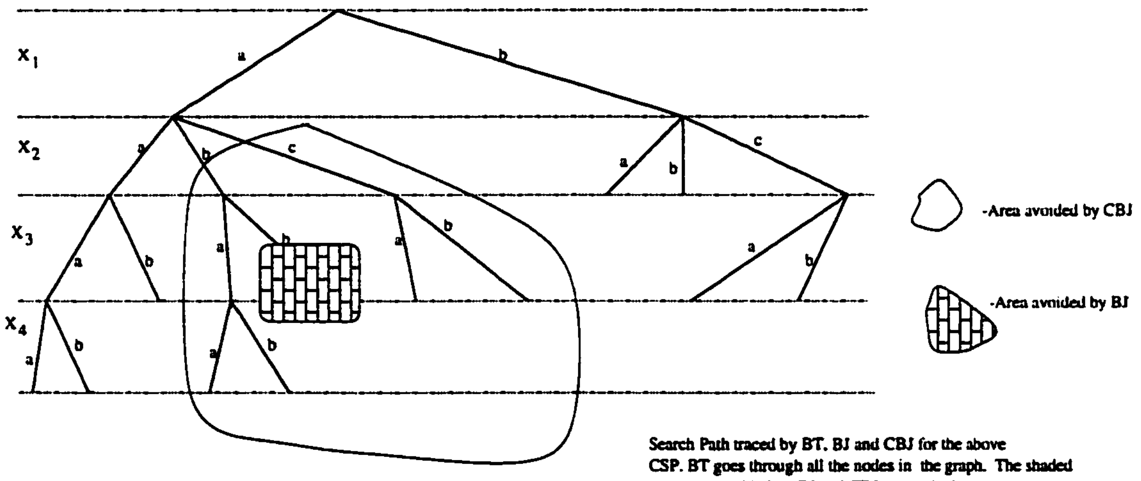
$C_2(X_1,X_4) = \{(a,b)\}$

$C_3(X_1,X_3) = \{(a,a)\}$

$C_4(X_2,X_3) = \{(b,a),(a,a)\}$

$C_5(X_2,X_4) = \{(b,a),(a,b)\}$

$C_6(X_4,X_3) = \{(a,a)\}$



Search Path traced by BT, BJ and CBJ for the above CSP. BT goes through all the nodes in the graph. The shaded areas are avoided by BJ and CBJ respectively.

Figure 2.1: A CSP example with execution traces of BT, BJ and CBJ

most recent variable will prove futile. This motivated the development of algorithms involving some amount of bookkeeping and avoiding unnecessary backtracking.

As a result of bookkeeping the algorithm becomes more informed and unnecessary backtracking is eliminated. The important improvements of chronological backtracking are listed below:

1. Backjumping (BJ) [22]

2. Conflict-directed backjumping (CBJ) [46]

Backjumping

Backjumping(BJ) is similar to *chronological backtracking* but it behaves more efficiently when no consistent instantiation can be found for the current variable x_i at dead-end. Instead of backtracking to the chronologically most recent variable, BJ jumps to the deepest past variable x_h that was checked against the current variable. This is more efficient because any further instantiation of any variable between x_h and x_i is futile.

In [11] BJ is shown to outperform BT universally, the improvement being significant when the constraint graph is sparse.

Conflict-directed backjumping

This technique is based on the same idea as *backjumping*. However, instead of the simplistic notion used in *backjumping* a more complicated bookkeeping method is used to give a more efficient method. In CBJ every variable has a record of its own conflict set, i.e. the set of past variables which failed consistency checks with its current instantiation. Every time a consistency check fails for a variable x_i with a past variable x_h, x_h is added to conflict set of x_i. When there are no more values of x_i to be tested, the algorithm backtracks to the deepest variable x_h in the conflict set of x_i, and the conflict set of x_i with the exception of x_h is added to the conflict set of

x_h, thus adding extra efficiency. This effectively eliminates unnecessary consistency checks of variables after x_h which are sure to fail.

To understand the difference between the effectiveness of these variations of backtracking, consider the CSP whose graph is shown in Figure 2.1. The CSP consists of four variables X_1,X_2,X_3 and X_4 with domains $d_1 = \{a, b\}$, $d_2 = \{a, b, c\}$, $d_3 = \{a, b\}$, and $d_4 = \{a, b\}$ respectively. It has six constraints $C_1(X_1, X_2) = \{(b, c), (a, a), (a, b), (a, c)\}$, $C_2(X_1, X_4) = \{(a, b)\}$, $C_3(X_1, X_3) = \{(a, a)\}$, $C_4(X_2, X_3) = \{(b, a), (a, a)\}$, $C_5(X_2, X_4) = \{(b, a), (a, b)\}$ and $C_6(X_4, X_3) = \{(a, a)\}$ as shown in the upper part of Figure 2.1. The solution space (search tree) explored by chronological backtracking is shown for the CSP represented by the graph. In the figure the portions of the solution space which are avoided by the BJ and CBJ algorithms have also been indicated. It is evident that a larger chunk of the search tree is truncated by CBJ as opposed to BJ.

Consider BJ. At the point $\{X_1 = a,\ X_2 = b,\ X_3 = a\}$ which is consistent, X_4 is tried for a. But it is inconsistent with $X_1 = a$. Trying the next value for X_4, namely b, we find that it is consistent with $X_1 = a$ but inconsistent with $X_2 = b$. By now all values of X_4 are exhausted and we need to backtrack. The deepest variable in conflict with X_4 is X_2, and hence any backtrack to X_3 is pointless. Thus BJ backjumps to X_2 to the point $\{X_1 = a, X_2 = c\}$ and the search proceeds further.

Consider CBJ. Here we consider the nodes $\{X_1 = a,\ X_2 = a,\ X_3 = a,\ X_4 = a\}$ and $\{X_1 = a,\ X_2 = a,\ X_3 = a,\ X_4 = b\}$. At this level the conflict set of variable X_4 is $\{X_1, X_3\}$ and thus backjump to X_3(the deepest point in the conflict set of X_4) takes place and X_1 is added to the conflict list of X_3 which is initially empty. After all domain values of X_3 are exhausted, it jumps to the deepest in the conflict set of X_3, which is X_1. Thus it bypasses all further values of X_2.

Hence the number of nodes pruned by CBJ is much higher than BJ which in turn is much higher than BT.

2.1.3 Constraint propagation algorithms

In contrast to chronological backtracking and its intelligent variants, the constraint propagation algorithms achieve a limited degree of consistency either statically or dynamically to simplify the CSP. Unless otherwise mentioned these algorithms here shall be explained for binary versions of constraint satisfaction problems.

The fundamental idea in consistency based algorithms is based on the idea of k-consistency, for a fixed k. The definition follows

Definition 1 *[34]: A constraint satisfaction problem (CSP) P is k-consistent, iff given any instantiation of any k-1 variables satisfying all the direct constraints between them, it is possible to find an instantiation of the kth variable such that the k values taken together satisfy all the constraints among the k variables.*

In other words, k-consistency states that any partial solution for $k-1$ variables can be consistently extended to a partial solution containing an additional variable. A 1-consistent CSP is said to be *node consistent*, a 2-consistent CSP is said to be *arc-consistent*, and a 3-consistent CSP is said to be *path consistent*. The idea of enforcing consistency on a CSP is to remove local inconsistencies so that the task of finding a global solution becomes easier. In general, it is common practice to use levels of consistency till 2. So most constraint systems applying constraint propagation algorithms limit themselves to enforcing node- and arc-consistency. Even the enforcement of arc-consistency is done in varying degrees. In some algorithms, full arc-consistency is attained at the start and then backtracking is applied. In some of the extreme algorithms the full arc-consistency is maintained at the start, as well as at every stage of the backtracking [17]. But the cost of achieving arc-consistency can be prohibitive in comparison to the cost saving achieved by pruning.

As a compromise, a limited version of arc-consistency, enforced dynamically alongside backtracking, is used commonly in CSP algorithms. One common technique which falls into this category is the forward checking algorithm.

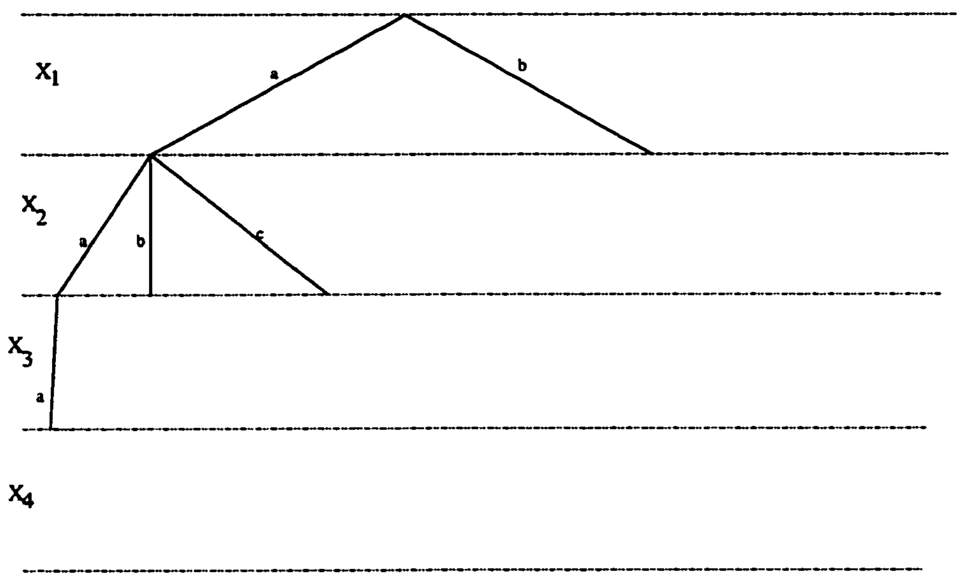


Figure 2.2: Search tree traced by forward checking for the CSP in Figure 2.1

Forward checking

Forward checking (FC) [25] is a limited constraint propagation algorithm where partial consistency is enforced in the intermediate stages of the backtracking algorithm.

In forward checking, consistency check is done between the current variable and the future variables. Once a current variable is instantiated, the partial solution is checked for consistency against all future variables, and the values which are inconsistent with the current partial value assignment are removed from the domain of each future variable. If a future variable has no value remaining, backtrack occurs. The current variable is then instantiated to its next available value from its domain, and all the values removed in the previous assignment are undone. Further, the domains of the future variables are brought back to the state before the last consistency step was enforced. If the last variable can be instantiated in this manner a solution has been found.

Forward checking embodies a compromise between naive backtracking and maintaining full arc-consistency [17].

Example

Let us illustrate the running of forward checking on the example shown in Figure 2.1. The search tree visited by forward checking is shown in Figure 2.2. It is immediately evident that the number of nodes traversed by FC is the least of all the algorithms BT, BJ, CBJ and FC. After starting at $\{X_1 = a\}$, this value is propagated against X_2 ,X_3 and X_4. At this stage the value b is removed from the domain of X_3, and the value a is removed from the domain of X_4, and none removed from domain of X_2. Then the assignment is expanded with $X_2 = a$. At this stage, we again propagate this value to X_3 and X_4. There is no reduction of domains of X_3 and X_4. The assignment is then expanded to assign a to X_3. At this stage on propagation to X_4, the domain of X_4 is annihilated (i.e. with all values in the domain removed), causing backtracking. The backtrack to the next value b of X_2 occurs. The value of b is then added back to X_4, as a part of the undoing process. This value of b for X_2 is then propagated to X_3, and X_4. The value of b is removed from the domain of X_4 causing the annihilation of the domain of X_4. This being a dead-end, backtracking occurs to value c of X_2. Upon propagating this value to the variable X_3, its domain is annihilated. Thus backtrack now returns to $X_1 = b$, and all propagation effects of $X_1 = a$ are undone. All domains are thus back to their original form. The assignment $\{X_1 = b\}$ is then propagated to the variables X_2, X_3 and X_4. The values a and b are removed from the domain of X_2, and the entire domain of X_3 is annihilated. So backtrack occurs and end of problem is reached without any solution.

2.2 Over-constrained systems

Some constraint satisfaction problems do not admit a solution satisfying all the constraints in the problem. Such problems are termed as *over-constrained problems*. Since not all constraints can be satisfied, commonly some measures are used to specify the relative importance of each constraint in generating a *preferred* solution. The relative importance of each constraint can be specified in terms of various mechanisms. The mechanisms used in literature of over-constrained systems include weights attached to constraints, priority information as partial order on constraints, and hierarchies.

In this section we present common notions of *preferred* solution to over-constrained problems discussed in the literature.

2.2.1 Maximal weighted constraint satisfaction

Maximally weighted constraint satisfaction problem (max-weighted CSP) refers to the problem setting in which each constraint is assigned a weight according to its importance. A solution is an assignment of values to variables, such that the total of the weight of the constraints satisfied by the assignment is maximum among all assignments.

Formally, a max-weighted CSP P involves a set $X = \{x_1, x_2, \ldots, x_n\}$ of n variables, a set C of constraints $\{c_1, c_2, \ldots, c_m\}$, where each variable $x_i \in X$ takes a value from its domain d_i, and each constraint $c_i \in C$ is defined as a relation on a subset X_c of the set of variables X, and associated with each constraint $c_i \in C$, is a numerical value w_i, w_i being a real number.

An assignment A of values to variables in X, *satisfies* a constraint c iff the tuple formed by X_c in A belongs to R_c, the relation associated with c. Let $w(A) = \sum\{w_i \mid A\ satisfies\ c_i\}$. Let W_P represent the set of all possible assignments A, assigning values to all variables in X. The *solution* set S of the problem is defined as

$$S = \{A' \in W_P \mid w(A') = max\{w(A) \mid A \in W_P\}\}$$

Consider a CSP P with three constraints c_1, c_2, and c_3 defined on three variables x_1,x_2, and x_3 with domains $\{0,1\}$, $\{1,2\}$, and $\{1\}$ respectively. Assume that the weights associated with c_1, c_2, and c_3 are 1, 2, and 4 respectively. Now let $c_1(x_1,x_2) = \{(0,2),(1,1)\}$, $c_2(x_2,x_3) = \{(2,1)\}$, and $c_3(x_1,x_2,x_3) = \{(0,1,1)\}$. P

has only one solution, namely $\{x_1 = 0, x_2 = 1, x_3 = 1\}$. This assignment satisfies only one constraint c_3 with weight 4. Any other assignment of variables yields a lower combined weight of satisfied constraints than this assignment.

2.2.2 Maximal constraint satisfaction problem

A special case of the max-weighted CSP is what is called the maximal constraint satisfaction problem (max-CSP), where the weights of all constraints are considered to be 1. The search for a solution turns out to be a search for an assignment satisfying maximal number of constraints.

Formally, a max-CSP P involves a set $X = \{x_1, x_2, \ldots, x_n\}$ of n variables, a set C of constraints $\{c_1, c_2, \ldots, c_m\}$, where each variable $x_i \in X$ takes a value from its domain d_i, and each constraint $c_i \in C$ is defined as a relation on a subset X_c of the set of variables X. If A is an assignment, let $sat(A) = \{c_i \mid A\ satisfies\ c_i\}$. Then let $w(A) = \mid sat(A) \mid$, where $\mid D \mid$ stands for the cardinality of a set D. Let W_P represent the set of all possible assignments A, assigning values to all variables in X. Then the *solution* set S of the problem is defined as

$$S = \{A' \in W_P \mid w(A') = max\{w(A) \mid A \in W_P\}\}$$

Consider a CSP P with three constraints c_1, c_2, and c_3 defined on three variables x_1,x_2, and x_3 with domains $\{0,1\}$, $\{1,2\}$, and $\{1\}$ respectively. Let $c_1(x_1, x_2) = \{(0,2),(1,1)\}$, $c_2(x_2, x_3) = \{(2,1)\}$, and $c_3(x_1, x_2, x_3) = \{(0,1,1)\}$. P has only one solution, namely $\{x_1 = 0, x_2 = 2, x_3 = 1\}$. This assignment satisfies two constraints: c_1 and c_2. Any other assignment of variables satisfies either a single constraint or no constraints.

2.2.3 Hierarchical constraint satisfaction problems

Another important school of over-constrained systems allowing for specification of preferences in constraints is the hierarchical constraint logic programming (HCLP) paradigm [63]. HCLP employs constraint hierarchies [4] in constraint logic Programming [28].

In constraint hierarchies, an arbitrary number of levels of preference is allowed, each successive level being more weakly preferred than the previous one. The set of *required* constraints represents the lowest level in the hierarchy.

A solution to a constraint hierarchy is an assignment to the free variables in the hierarchy, such that the set of required constraints in the hierarchy are satisfied and the remaining constraints are satisfied in the best possible manner. In addition to the basic criteria of satisfaction of the set of required constraints, the solution must be better than any other assignment which satisfies the hierarchy too. The notion of *better* is based on the comparison of how well two assignments of values satisfy the hierarchy of non-required constraints. The comparison measures are referred to as *comparators.* The conditions for a *comparator* are that it should be an irreflexive and transitive relation. Further it is desired that any comparator *respect* the hierarchy. By *respecting* the hierarchy, it is meant that if there is a solution to the set of *required* constraints such that it satisfies all non-required constraints through level k, then all solutions returned by the comparator must satisfy all the constraints through level k.

Comparators in HCLP

Comparators in HCLP [63] refer to methods to compare the valuations satisfying a constraint hierarchy. These methods are used for comparing constraint hierarchy solutions, and are hence based on combining the results of satisfaction of constraints at one level with the results at the next level. The magnitude of how well a solution satisfies constraints at a given level in the hierarchy is measured by an error function. The error function is a measure of how well a solution satisfies a constraint. The error function can be defined in a number of ways for a given hierarchy. For a given hierarchy, we can define a *trivial* error function which defines to 0 if the constraint is not satisfied or 1 if it is. Otherwise we can use a *metric* error function which defines the error in terms of the difference between the expected output and the output of a constraint. At a particular level, the error of each constraint is aggregated to obtain a combined measure of the total error at the current level with the current solution. So for any comparator, we can have the predicate better or the metric better version of the comparator.

Ideally the *metric* function should also be converted to *predicate* function so that we can use a uniform scale for comparison for comparators. But in many real life cases, metric error function is capable of yielding precise and better results.

But a notion which needs to be described before the concepts of comparison of valuations is described is the concept of combining function. The combining function g when applied to real valued vectors returns a value which can be compared using two relations $<>_g$ and $<_g$. The requirement for $<_g$ is that it must be irreflexive, antisymmetric and transitive. The relation $<>_g$ must be reflexive and symmetric.

The combining function G is a generalization of g and is applied to error sequences and returns a sequence of values that can be compared using $<>_g$ and $<_g$. This type of sequence is called the combined error sequence.

The combined error sequences are compared using the lexicographic ordering of the sequences. Based on this ordering, the solution set S of a constraint hierarchy is defined as the subset of S_0 , the set of solutions to set of required constraints (ignoring the soft constraints), consisting of all the valuations for which no *better* (based on lexicographic ordering) valuations in S_0 exist.

In HCLP [63] there can be a variety of comparators for combining the results across multiple levels depending upon which the type of the result may vary. The comparators can be of three types - local, regional and global. For a local comparator each constraint is considered individually. So for a particular level, there will occur a constraint for which one solution is better than the others and for all the other levels below this level this solution behaves same as other solutions worse than it. In a global comparator, the error sequence at a given level is aggregated using a aggregation function. In regional comparator, the constraint at each level is considered individually like in the local case, but it has the additional advantage that the comparators incomparable at a lower stricter level, can still be compared at a higher level. So the general behavior of each comparator depends upon the hierarchy and also on the constraints constituting each hierarchy.

Common comparators

Some of the more common comparators used in HCLP are presented in this section. These different comparators are obtained by using different relations $<>_g$ and g. To list a few they are:

Weighted-sum better The constraint errors at any given level are aggregated taking the weighted sum of the errors at that level.

Worst-case better The constraint errors at any given level are aggregated taking the weighted maximum of the errors at that level.

Least squares better The constraint errors at any given level are aggregated taking the weighted sum of the squares of the errors at that level.

Locally Better The solution must perform at least as well as any other solution at all levels below k and at level k must do strictly better than other solutions for at least one constraint and at least as well for all other constraints.

Regionally Better The solution is exactly as a local comparator except that the $<>_g$ is not the strict $=$, as is the case in locally better, but $<>_g$ also succeeds for cases where two values are not comparable.

In combination with the error function chosen, the comparators can be either of predicate type or metric type. e.g. Locally-metric-better or Locally-Predicate-Better.

2.2.4 Partially ordered constraint hierarchical systems

In this section a generalization of the scheme of HCLP shall be discussed. It can be shown that in lot of real life applications of constraints, where the underlying problem is over-constrained, a strict constraint hierarchy as used in HCLP does not work.

The problem with constraint hierarchies is in the imposition of levels which induces unnecessary priority relationships between constraints. The constraints which are at different levels have a priority relationship which is many to many. Any constraint in a higher priority level is more preferred than any constraint in the lower priority level. This concept does not fit very well with some realistic applications where constraint hierarchies have been used.

To avoid this problem of over-specification, a generalization of the priority relationship between the constraints is necessary. Such a generalization can be in the form of a partial order on the set of constraints. This is a superset of the constraint hierarchy relation on the set of constraints.

Partial order on constraints typically assumes the constraints to be ordered on a reflexive, transitive and anti-symmetric relation on the set of constraints. This relation, termed as the priority relation, forms the governing rule for deciding the order of application of constraints. An optimal solution can also be defined in terms of the partial order on the set of constraints involved in the relation.

Partially ordered hierarchies have been studied in [5]. A totally ordered hierarchy H is said to be consistent with a partially ordered hierarchy G, if H has the same set of constraints as G, and if there is a mapping m from the strengths of G to those in H such that if $s_1 < s_2$ in G, then $m(s_1) < m(s_2)$. and there is a one to one relationship between any constraint $s_i c_i$ in G and $m(s_i) c_i$ in H.

Under such definition the set of solutions of the partially ordered constraint hierarchy G, is given by the union of the solutions of all totally ordered hierarchies H consistent with G.

Thus in this semantics, the partial order information is heavily dependent upon the comparator used in the obtaining of solutions of the approximating totally ordered constraint hierarchies consistent with the partially ordered hierarchy.

Key fact to note is that such kind of approximation by a totally ordered constraint hierarchy is possible only because the priority relationship between pairs of constraints in this case is inclusive in nature. The satisfaction of a higher priority constraint does not affect the satisfaction of a lower ranked constraint.

2.2.5 Preference information in existing over-constrained frameworks

The over-constrained systems studied in this section use a range of methods to deal with the problem of identifying the *preferred* solution. In all the frameworks dealing with the notion of *preferred* solution, an important factor which affects the semantics of the solution is the nature of preference relation between constraints.

In the simplest case of max-CSP, every constraint is considered to be of equal importance. A solution is then defined by an assignment satisfying a maximal number of constraints. Thus there is practically no concept of any preference relation between constraints.

Next, in case of max-weighted CSP each constraint is given a unique weight depending upon its importance. But the introduction of weight does not precisely capture the nature of semantics of preference between two constraints. A large number of constraints of lower preference measure (weight) could outweigh or equal a constraint with a large weight. Thus albeit the preference information encoded in max-weighted CSP is more fine-grained than preference-less max-CSP, max-weighted CSP still does not capture the notion of preference between constraints in a satisfactory manner.

The concept of preference between constraints is better captured in HCLP which employs constraint hierarchies [4]. In HCLP, the constraints are assigned unique levels according to the importance of the constraint. A set of *required* constraints form the most important and the lowermost level l_0 in the hierarchy. As we go from l_1 (the most important non-required level of constraints) to l_n (the least important non-required level of constraints) the importance of a constraint decreases. Though the precise definition of a solution to a hierarchy depends upon the choice of comparator, the preference information between constraints is captured in HCLP in a better manner than max-weighted CSP. This is due to the lexicographic ordering employed in the derivation of a solution. In a lexicographic ordering, any single constraint at a lower level l_i in the hierarchy is more important than any number of constraints at a level l_j higher than l_i. Thus the anomaly of the max-weighted CSP is corrected in HCLP. In spite of this general nature of preference relation between constraints captured by HCLP, in some cases the preference relation imposed becomes restrictive. This restrictiveness is due to the fact that the hierarchy imposes a many to many preference relation between constraints at a lower level and constraints at a higher level. This might be too restrictive in cases where we need to specify a one-to-one preference relation between any two constraints. Such kind of preference information requires a finer representation of priority.

The above-mentioned restrictive nature of the HCLP framework is overcome by partially ordered constraint hierarchies [5]. In these hierarchies, it is possible to specify more fine-grained priority information. The priority information is specified as a binary, transitive, and anti-symmetric relation on the set of constraints. Since it is possible to specify the relation between constraints directly, we can specify a fine-grained priority relation between pairs of constraints which does not suffer from the restrictiveness of HCLP. Among all, partially ordered constraint hierarchies represent the most general kind of priority information that can be accommodated in over-constrained frameworks.

2.3 Function-free normal logic programs and stable models

In the recent past, languages based on stable models for logic programs have been developed for expressing and solving problems in artificial intelligence. The notable among these languages are the LP_{SM} language developed by Niemela in [41], and the stable logic programming (SLP) paradigm developed by Marek and Truszczynski in [37]. In [41], LP_{SM} is shown to be an effective language for modeling constraint satisfaction problems, planning problems, and other computational problems in artificial intelligence. On similar lines, in [37] the SLP paradigm was used to model various constraint satisfaction problems. Before we proceed with a detailed discussion of the languages, we introduce the *stable model semantics* for logic programs with ground or propositional terms.

2.3.1 Stable model semantics

Logic programs with ground or propositional terms consist of rules of the form:

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

where $a_1, \ldots, a_n, b_1, \ldots, b_m$, and h are propositional atoms. The expression *not b* is called a not-atom.

Before proceeding with the definition of stable models, we need to define the concept of *reduct*. The *reduct* P_A of a general logic program P with respect to a set of atoms A is the program obtained by applying the following two steps to P:

1. by deleting each rule in P that has a not-atom *not* x in its body such that $x \in A$ and

2. by deleting all not-atoms in the remaining rules.

The *stable model semantics* [24] for a logic program with ground terms P is then defined as follows: A set of atoms A is a *stable model* of P if and only if A is the unique minimal model of the reduct P_A. The unique minimal model is obtained as the deductive closure of the program P_A.

Consider $P = \{a \leftarrow not\ b; b \leftarrow not\ a\}$. Take the set of atoms $A = \{a\}$. The reduct P_A is $\{a \leftarrow\}$. The deductive closure of P_A is $\{a\}$. Since it is the same as A, therefore $\{a\}$ is a stable model of P. But if we consider $A = \{a, b\}$, the reduct P_A is the empty program $\{\}$. The unique minimal model is thus $\{\}$. This is not the same as A. Hence $\{a, b\}$ is not a stable model of P.

2.3.2 Function-free normal logic programs

Function-free normal logic programs (FFNLPs) with variables refer to sets of clauses of the form

$$\begin{aligned} h(Y_1, Y_2, \ldots) \leftarrow\ & a_1(X_{11}, X_{12}, \ldots), a_2(X_{21}, X_{22}, \ldots), \ldots, a_n(X_{n1}, X_{n2}, \ldots), \\ & not\ b_1(Z_{11}, Z_{12}, \ldots), \ldots, not\ b_m(Z_{m1}, Z_{m2}, \ldots) \end{aligned}$$

where $h, a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m$ represent predicates and $Y_1, Y_2, X_{11}, Z_{11}, Z_{m1} \ldots$ etc. are variables.

The absence of functions in FFNLPs controls the explosion of the size of the ground version of these programs. The restriction of FFNLPs to finite programs yields a computationally interesting and expressive language. The stable models of a finite FFNLP are the stable models of the ground translation of the FFNLP. Both the important paradigms of SLP and LP_{SM} depend upon languages which are subsets of finite FFNLPs.

A finite FFNLP can capture integrity constraints. A ground integrity constraint refers to a rule of the form

$$f \leftarrow B_1, \ldots, B_m, not\ C_1, \ldots, not\ C_m, not\ f$$

The semantics of the integrity constraint above dictates that any set of atoms S satisfies the above integrity constraint iff it is not the case that $\{B_1, \ldots, B_m\} \subseteq S$

and $\{C_1, \ldots, C_n\} \cap S = \phi$. A non-ground integrity constraint is represented by the set of ground instances of the rule.

The idea of using finite FFNLPs for programming constraints is based on the idea that each clause is interpreted as a constraint on the solution set of the problem. The underlying principle of use of FFNLPs in modeling constraints is shown in the following example:

Consider the following finite non-ground FFNLP for the $k-$coloring of a graph. It assumes two predicates $vertex(X)$ and $arc(X, Y)$ to represent the graph. A graph is defined by a set of ground instances of *vertex* and *arc* predicates. Further the predicate $col(C)$ denotes the set of available colors. The FFNLP program consisting of the following rules captures the semantics of the $k-$coloring of the graph. (assume == to be defined extensionally as a predicate ==(X,Y) consisting of all pairs X, Y such that X is same as Y).

$$\begin{aligned} color(V, C) &\leftarrow vertex(V), col(C), not\ othercolor(V, C) \\ othercolor(V, C) &\leftarrow vertex(V), col(C), col(D), not\ C == D, color(V, D) \\ f &\leftarrow arc(V, U), col(C), color(V, C), color(U, C), not\ f \end{aligned}$$

The first clause states that if a vertex is assigned a color, it cannot be assigned another color. The integrity constraint states that if there is an arc between two nodes, they cannot be assigned the same color. The program returns a stable model iff there is a $k-$coloring of the graph. From a stable model, all facts of the form $color(v, c)$ indicate the respective color assigned to each node.

It is clear that a variety of constraints can be captured in the framework of finite FFNLPs.

2.4 LP_{SM} language and the smodels proof procedure

2.4.1 The language of LP_{SM}

In this section we present the details of one subset of finite FFNLPs called LP_{SM} proposed in [41]. In LP_{SM}, a sub-class of FFNLPs is used as the underlying language. The sub-class is defined by the property of *domain-restrictedness* [41]. *Domain restrictedness* is a special case of the general property of *range restrictedness* which refers to the property that any variable which appears in a rule also appears in some positive body literal in the same rule. The property of *domain-restrictedness* is defined as follows:

Definition 2 (Domain-restrictedness) *[41] Let P be a logic program and D be a set of predicates. Then P is domain restricted with respect to D, if for each rule it holds that every variable appearing in the rule appears also in a positive body literal for which the predicate is from D.*

A *domain restricted* program can be further restricted by restricting the set of predicates to those with a fixed set of ground instances.

Definition 3 *[41] Let P be a logic program that is domain restricted with respect to D, and let $\hat{D}$ be a set of ground instances of predicates in D. The program $P^{\hat{D}}$ is defined as the set of ground instances of the rules in P such that each positive body literal with a predicate from D belongs to $\hat{D}$.*

Further, the notion of completeness is defined for a set of ground instances $\hat{D}$.

Definition 4 *[41] Let P be a logic program which is domain restricted with respect to D and $\hat{D}$ a set of ground instances of predicates in D. Then $\hat{D}$ is complete for P iff for each ground instance $\hat{d}$ of a predicate $d \in D$ it hold that (i) if $\hat{d}$ is in some stable model of P, then $\hat{d} \in \hat{D}$ and (ii) if $\hat{d}$ is in some stable model of $P^{\hat{D}}$, then $\hat{d} \in \hat{D}$.*

Certain classes of domain definitions are amenable to efficient computation of complete ground instances. LP_{SM} uses a class of domain predicates which have stratified and non-recursive domain definitions in the program depending only on other domain predicates. This class of domain definitions permits an efficient way to generate complete set of ground instances. Thus effectively LP_{SM} uses function-free domain restricted programs with stratified non-recursive domains. The following theorem [41] states the result about the preservation of semantics in translation of a non-ground program to its ground equivalent.

Theorem 2.1 *[41] Let P be a domain restricted program with respect to D and $\hat{D}$ a subset of the ground instances of the predicates in D with respect to the Herbrand universe of P such that $\hat{D}$ is complete for P. Then P and $P^{\hat{D}}$ have the same stable models.*

The use of a restricted class of programs permits additional programming features to be integrated into the logic program rules. The additional features added in LP_{SM} in addition to the usual non-ground logic program rules allowed by FFNLPs are the *built-in* functions. The addition of *built-in* functions is possible because of the absence of the process of *floundering*. *Floundering* refers to the situation when a procedure receives uninitialized values as parameters. The use of *domain-restricted* programs avoids the phenomenon of *floundering*. Hence LP_{SM} allows built-in functions to be used in its language in the infix form. Some built-in functions and predicates allowed in LP_{SM} are *abs*, $\neq$, $+$.

Effectively, in the most general form LP_{SM} contains the following two types of rules:

Program rules In the most general form a program rule is of the form

$$\begin{aligned} h(X_1, X_2, \ldots) \leftarrow\; & d_1(X_1), d_2(X_2), \ldots, d_m(X_m), \\ & p_1(X_{i1}), p_2(X_{i2}), \ldots, expr_1 \; mathop \; expr_2, \ldots, \\ & not\; q_1(X_{k1}), \ldots. \end{aligned}$$

where $d_1, \ldots, d_m$ denote domain predicates, $p_1, p_2, q_1, \ldots$ etc. are non-domain predicates, $expr_1 \ldots$ etc. denote mathematical expressions over variables defined by the domain predicates, and *mathop* denotes a mathematical operator (including $\neq, >$ etc).

Integrity constraints An integrity constraint is a program rule without any head.

Consider the encoding of the k−coloring of a graph in LP_{SM} [41]. Keeping all parameters and definitions same as the case in FFNLPs, the corresponding encoding in LP_{SM} is

$$\begin{aligned}
color(V,C) &\leftarrow vertex(V), col(C), not\ othercolor(V,C)\\
othercolor(V,C) &\leftarrow vertex(V), col(C), col(D), C \neq D, color(V,D)\\
&\leftarrow arc(V,U), col(C), color(V,C), color(U,C)
\end{aligned}$$

Note that the only additional power here is in the inclusion of $\neq$ within the language of LP_{SM}. But an operational difference is to be mentioned with respect to the implementation of integrity constraints in LP_{SM}. In LP_{SM} any integrity constraint of the form

$$\leftarrow B_1, \ldots, B_m, not\ C_1, \ldots, not\ C_m$$

is enforced in LP_{SM} by converting the rule to a clause

$$false \leftarrow B_1, \ldots, B_m, not\ C_1, \ldots, not\ C_m$$

and then enforcing the constraint that any returned stable model should not have the atom $false$ in it.

An LP_{SM} program is solved in a two-stage process [41]. The two stages are:

lparse - This stage converts a LP_{SM} program to a ground program such that soundness and completeness is not lost.

smodels - This stage computes the stable models of the converted ground program.

Our stress in the next section is to explore the details of the **smodels** stage of the LP_{SM} solver.

2.4.2 Description of smodels system

In [41], LP_{SM} is presented as a language for constraint satisfaction, where the rules of a program are seen as constraints on the stable models. LP_{SM} uses **smodels** [54], an efficient procedure for computing the stable models of ground logic programs. In this section an overview of the **smodels** system is presented. The material in this section is taken from [54].

The **smodels** system uses two main efficiency techniques:

Backjumping An improvement over chronological backtracking.

Lookahead A pruning method based on an intelligent way of choosing the next literal to be instantiated.

Further two deductive closures **expand** and **conflict** are given linear-time implementations that provide a linear-space implementation method for the decision procedure.

2.4.3 The decision procedure

The decision procedure for the stable model semantics is presented in this section. Logic programs with ground terms consist of rules of the form:

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

where $a_1, \ldots, a_n, b_1, \ldots, b_m$, and h are propositional atoms. The expression $\mathit{not}\ b$ is called a not-atom. For an atom x let $\mathit{opp}(x) = \mathit{not}\ x$, and for a not-atom $\mathit{not}\ x$

let *opp*(*not* x)$= x$. For a set of atoms A, $not(A) = \{not\ a \mid a \in A\}$. For a set B containing both atoms and not-atoms, the set of atoms in B is denoted by B^+ and the set of not-atoms by B^-. Consider $B = \{a, b, not\ c, not\ a\}$. Then $B^+ = \{a, b\}$ and $B^- = \{not\ a, not\ c\}$. Atoms and not-atoms are also called literals. Further, for a set of literals B, let $Atoms(B) = \{x \mid x \in B \text{ or } not\ x \in B\}$. Similarly, for a program P, let $Atoms(P)$ denote the set of all atoms x such that either x or *not* x appears in P. For a set of atoms Δ, if the set of all atoms in the language is H, then $\overline{\Delta} = H - \Delta$. For a set of rules P let the set of negative antecedents in P be denoted by $Neg(P)$, i.e., $Neg(P)$ contains every atom x such that *not* x appears in P.

Definition 5 (Agreement) *[54] Let A be a set of atoms and let B be a set of literals. A set of atoms A* **agrees** *with B if*

$$B^+ \subseteq A \text{ and } B^- \cap not(A) = \phi$$

Definition 6 (Coverage) *[54] Let A be a set of atoms and B be a set of literals. B* **covers** *A if $A \subseteq Atoms(B)$.*

The deductive closure of a set of rules P and a set of literals B is denoted by $Dcl(P, B)$, where $Dcl(P, B)$ is the smallest set of atoms that contains B^+ and is the fix-point of $R(P, B)$ where

$$R(P,B) = \{h \leftarrow a_1, \ldots, a_n \mid h \leftarrow a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m \in P$$
$$\text{and } not\ b_i \in B^-, \text{ for i = 1,...,m } \}$$

is seen as a set of inference rules. Let $P = \{a \leftarrow not\ b; d \leftarrow not\ c\}$, and $B = \{not\ b, d\}$. Then $R(P, B) = \{a \leftarrow\}$. Hence $Dcl(P, B) = \{d, a\}$.

From the definition of a stable model given in previous section, it is clear that a stable model Δ can as well be defined by the following equation:

$$\Delta = Dcl(P, not(Neg(P) - \Delta)).$$

```
function smodels($P, B$)
$B'$ := expand($P, B$)
if conflict($P, B'$) returns true then
    return false
else if $Neg(P)$ is covered by $B'$ then
    return test($Dcl(P, B')$)
else
    Take some $x \in Neg(P)$ not covered by $B'$
    if smodels($P$, $B' \cup \{x\}$) returns true then
        return true
    else
        return smodels($P$, $B' \cup \{not\ x\}$)
    endif
endif
```

Figure 2.3: The **smodels** procedure

Basic decision procedure

The decision procedure for the stable model semantics is presented in Figure 2.3. It computes a stable model Δ of a program P agreeing with a set of literals B such that the function **test**(Δ) returns true, or it returns false if no such stable model exists. The function **test**(Δ) is used to test if the set of atoms Δ is a stable model of the program P.

The decision procedure **smodels**(P, B) finds stable models by nondeterministically exploring the search space consisting of all subsets of $Neg(P)$. The procedure assumes the existence of two functions, **expand** and **conflict**. We will give the precise definition of these functions as used in **smodels** shortly. In general, they should satisfy some basic conditions. For any B' = **expand**(P, B), the following properties are satisfied:

E1 $B \subseteq B'$, and

E2 every stable model Δ of P that agrees with B agrees with B'.

The function **conflict**(P, B) returns true or false and satisfies the following two conditions:

C1 if B **covers** $Neg(P)$ and the deductive closure $Dcl(P,\ B')$ does not agree with B, then **conflict**(P, B) returns true, and

C2 if **conflict**(P, B) returns true, then there is no stable model Δ of P such that Δ agrees with B.

The algorithm of **smodels** explores the search space of the possible subsets of S where $S = Neg(P) \cup not(Neg(P))$. Beginning from the starting set B, the algorithm interleaves the recursive calls to **smodels** and calls to the **expand** function. The idea is to avoid the costly calls to **smodels**, by propagating the truth values in B to the program P, thereby adding as many new literals to B as possible before generating a new call to **smodels**. Depending upon the way the procedure **smodels** is used, the starting value of B may be either the empty set $\{\}$, or a predetermined set of literals. If we need to find any stable model of the original program, we start with $B = \{\}$. If on the other hand, we want only stable models in which certain literals are necessarily true, we run the **smodels** procedure with a starting value of B as the set of such literals. E.g. suppose we want to find out if there is a stable model that contains a but not b, we invoke **smodels**$(P, \{a, not\ b\})$. If any stable model agreeing with this B exists, **smodels** outputs it.

In the intermediate stages of the algorithm, at each call to **smodels** a new literal (either an atom or a not-atom) l (where $l \in S$, S being $Neg(P) \cup not(Neg(P))$), is added to the existing B. Then the search proceeds for a stable model agreeing with this new set B.

The function of **expand** is to prune the search space by avoiding calls to **smodels**. At each call to **smodels**, the function **expand** is called to add as many literals to B before the next call to **smodels**.

The function **conflict**, on the other hand, is given the task of enforcing the stable model semantics. If the function **conflict**(P, B) returns true at any point, it indicates that the current set B cannot be expanded to generate any stable model.

The decision procedure **smodels** above is shown to be sound and complete in [54].

Expand and conflict

To prune the search space as much as possible, the **expand** function should return as large a set as possible and the **conflict** function should return true on as many sets as possible. Good candidates for the two functions **expand** and **conflict** are presented here.

Since **expand**(P, B) is going to enlarge the set B, (E1) is satisfied. If a stable model Δ of P agrees with the set of literals B, an approximation of the set of rules in P which actually take part in derivation of Δ, can be constructed by introducing a reduct of P . Let P_B, the reduct of P with respect to B, be the set of rules

$$\{h \leftarrow l_1, \ldots, l_n \in P \mid \; opp(l_i) \notin B\}$$

where $l_1, \ldots, l_n$ are literals and we know that for an atom x, $opp(not\ x) = x$ and $opp(x) = not\ x$. The following additional observations allow the propagation of B to B' without sacrificing the semantics of the stable models agreeing with B.

Inference rule 1(IR1) If the body $l_1, \ldots, l_n$ of the rule $h \leftarrow l_1, \ldots, l_n$ is a subset of B, then the head h belongs to every stable model agreeing with B.

Inference rule 2 (IR2) If P_B contains no rule with head h, then h is not an element of any stable model agreeing with B.

Inference rule 3 (IR3) If $h \in B$ is the head of only one rule $h \leftarrow l_1, \ldots, l_n$ in P_B, then $l_1, \ldots, l_n \in \Delta \cup not(\overline{\Delta})$ for every stable model Δ agreeing with B.

Inference rule 4 (IR4) If $not\ h \in B$ and h is the head of the rule $h \leftarrow l_1, \ldots, l_n$ in P_B for which $l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n \in B$, then every stable model that agrees with B agrees with $B \cup \{opp(l_i)\}$.

The above criteria help in identifying the additional literals which can be added to B without any problem in semantics. The literals which can be added in **expand** are based on the following two lemmas.

The closure $Cl(P, B)$ is defined as a minimal set closed under inference rules 1 to 4 above, that contains B. The following lemma is a direct result of the above observations.

Lemma 1 *[54] If a stable model Δ of P agrees with B, then Δ also agrees with Cl(P, B).*

More information can be deduced about the stable models that agree with B. In [54], it is shown that

$$\begin{aligned} \Delta &= Dcl(P, not(\overline{\Delta})) \\ &\subseteq Dcl(P_B, not(Neg(P))) \end{aligned}$$

Lemma 2 *[54] If a stable model Δ of P agrees with B, then Δ agrees with $B \cup \{ not\ x \mid x \in Atoms(P)$ and $x \notin Dcl(P_B, not(Neg(P)))\}$*

Based on the previous two lemmas, we have a strong criterion for the function **expand**(P, B). These two lemmas characterise the **expand** function outlined in Figure 2.4. The definition in Figure 2.4 satisfies both the criteria E1 and E2.

The function **conflict**(P, B) is made to return true when

$$not(B^+) \cap B^- \neq \phi$$

This function satisfies the criteria C1 and C2 as required for a definition of the **conflict** function. The functions **expand** and **conflict** are given in Figure 2.4.

```
function expand($P, B$)
repeat
        $B' := B$
        $B := Cl(P, B)$
        $B := B \cup \{not\ x \mid x \in Atoms(P)$ and $x \notin Dcl(P_B, not(Neg(P)))\}$
until $B' = B$
return $B$

function conflict($P, B$)
Precondition: $B =$ expand($P, B$)
if $not(B^+) \cap B^- \neq \phi$ then
        return true
else
        return false
endif
```

Figure 2.4: The procedures **expand** and **conflict**

Example

Let us consider an example program P to illustrate how the **smodels** procedure works in conjunction with **expand** and **conflict**. Let $P = \{a \leftarrow not\ a; a \leftarrow c, d; e \leftarrow not\ d; d \leftarrow not\ e; c \leftarrow\}$ Let the clauses in P be named R_1, R_2, R_3, R_4 and R_5 respectively to be able to refer to them at ease. The steps invoked in the generation of the first stable model of this program P are outlined in Figure 2.5.

2.4.4 Backtracking improvements: lookahead and backjumping

The decision procedure **smodels** employs chronological backtracking search. But this search method is very costly as it blindly explores all dead ends of the search space without exploiting any properties of the problem domain to lessen the amount of work.

Two main cost saving techniques are employed in **smodels**, namely the **lookahead** and backjumping. While **lookahead** makes an intelligent choice for the next literal, backjumping jumps over irrelevant variables when a conflict is encountered.

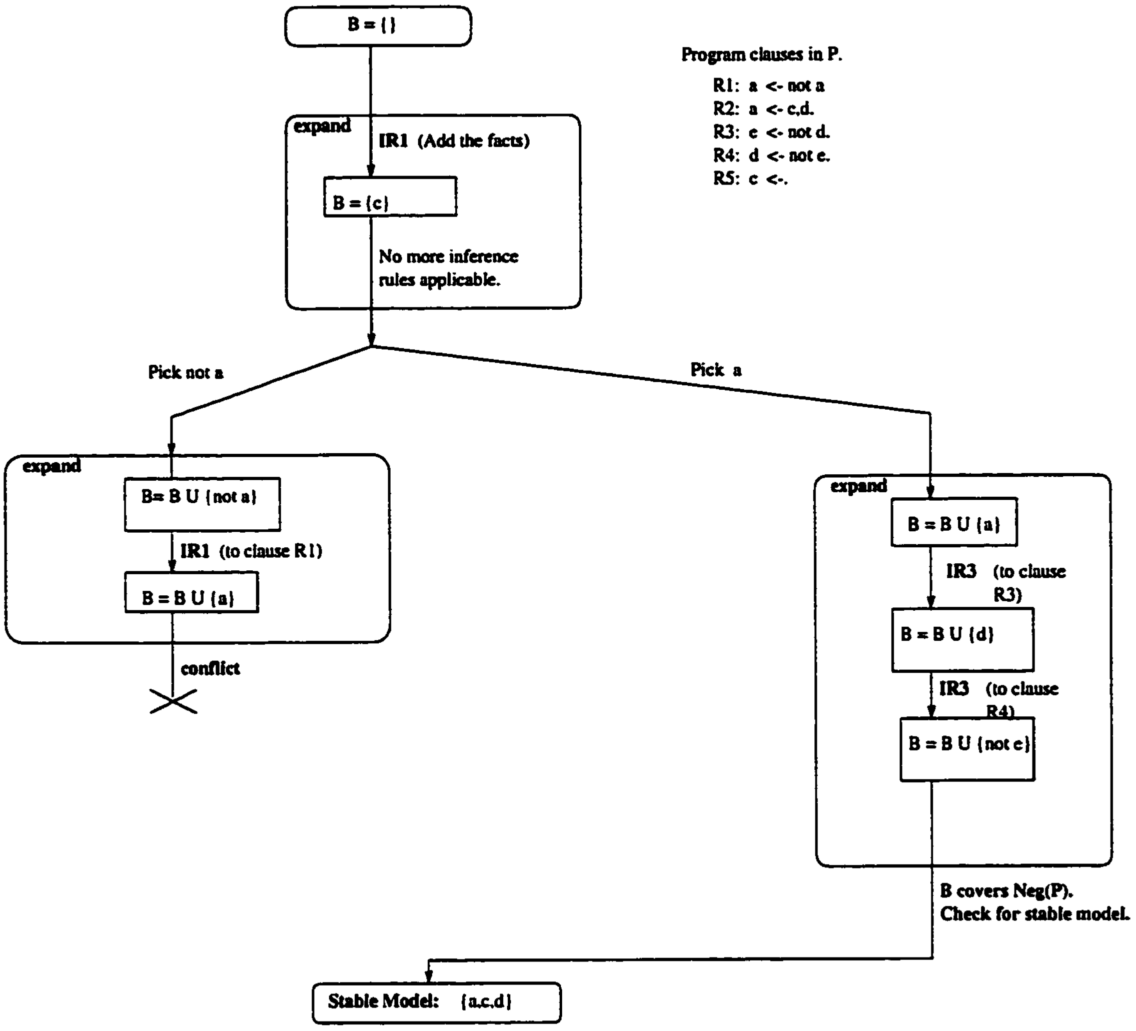


Figure 2.5: Execution trace of **smodels** on a sample program

```
function lookahead($P, B$)
$A := Neg(P)$ - $Atoms(B)$
$A := A \cup not(A)$
While $A \neq \phi$ do
    Take any literal $l \in A$
    $B'$ :=expand($P, B \cup \{l\}$)
    if conflict($P$,$B'$) returns true then
        return $opp(l)$
    else
        $A := A$ - $B'$
    endif
endwhile
return any atom in $Neg(P)$ not covered by $B$
```

Figure 2.6: The **lookahead** procedure

Adding lookahead

Lookahead is based on the idea of testing every possible choice before committing to one. In the case of **smodels**, this means that we can avoid choosing literals that directly leads to a conflict. Namely, if the stable model Δ agrees with the set of literals B, but Δ does not agree with $B \cup \{l\}$, we know that Δ agrees with $B \cup \{opp(l)\}$. Thus, **lookahead** immediately gives us a stronger pruning method: if **conflict**(P, B') returns true for B' =**expand**($P, B \cup \{l\}$), add $opp(l)$ to B. If we know for sure that choice of a literal l is leading to a conflict immediately, then we can save the search space traversed by **smodels** by considering only the path of addition of $opp(l)$, thereby saving a lot of unnecessary traversal. Lookahead has been shown to be one of the most effective cost-saving measures used in the **smodels** procedure.

The **lookahead** procedure is given in Figure 2.6. This principle is similar to the principle of dynamic variable ordering used in finite constraint satisfaction, where first fail principle is a guiding heuristic in the determination of the next variable to be chosen.

Adding backjumping

Although many conflicts are discovered using **lookahead**, there are still situations when the decision procedure exhaustively searches through assignments that are not relevant to a set of conflicts. A simple example is furnished by the union of two programs that do not share atoms. It is assumed that the first program has several stable models and that the second one has none. If we always begin by choosing atoms from the first program, then we will search through all stable models of the first program before we discover that the joint program has no stable models.

The notion of a path between two atoms in a program is introduced. A logic program generates an undirected graph whose nodes are the atoms appearing in the program and whose edges connect every pair of atoms appearing in the same rule. In effect, if

$$a_1 \leftarrow a_2 \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n$$

is a rule in the program, then every pair (a_i, a_j), for $1 \leq$ i, j $\leq$ n, is an edge in the graph. A path between two atoms in the program is then a path in the corresponding graph. Empty paths are accepted and a path between two literals is defined as the path between the atoms the literals cover.

The main technical result which allows backjumping is presented below.

Theorem 2.2 *[54] Let P be a program, B be a set of literals, and l_1 and l_2 be two literals such that there is no path from l_1 to l_2 in P_B and such that* **conflict***(P, B′) returns true for $B' =$* **expand***(P, $B \cup \{l_1, l_2\}$) but* **conflict***(P,* **expand***(P, $B \cup \{l_1\}$)) returns false. Then,* **conflict***(P,* **expand***(P, $B \cup \{l_2\}$)) returns true.*

This theorem explains the situation when a conflict is encountered on addition of two literals l_1 and l_2 to a set B, where there is no path between l_1 and l_2 in P_B. It states that if l_1 does not cause a conflict on addition to B, then l_2 definitely leads

```
function smodels($P, B$, level)
$B'$ := expand($P, B$)
if conflict($P$, $B'$) returns true
then
        return false
else if $Neg(P)$ is covered by $B'$ then
        top := level
        return test( $Dcl(P, B')$)
else
        conflict := $l$ := lookahead($P$, $B'$)
        if smodels($P$, $B' \cup \{l\}$, level + 1) returns true then
                return true
        else if (level < top or there is a path from $l$ to conflict in $P_{B'}$) ,then
                if level < top then
                    top := level
                endif
                return smodels($P$, $B' \cup \{opp(l)\}$, level + 1)
        else
                return false
        endif
endif
```

Figure 2.7: **smodels** with **backjumping** and **lookahead**

to a conflict upon addition to B. This theorem is the basis for the incorporation of backjumping in the proof procedure of **smodels**.

The decision procedure incorporating backjumping from conflicts and backtracking from stable models is presented in Figure 2.7. The algorithm assumes the existence of two global variables: *conflict* and *top*. The variable *conflict* holds the last choice leading up to a conflict, while the variable *top* keeps the level, or depth of recursion, above which backtracking is chronological.

The key savings occurs here when there is no path between a past variable l and a future variable $conflict$ where the conflict has actually taken place. In such a case, the possible cause of conflict is due to a value added between l and $conflict$. This ensures that the conflict remains intact if we try the **smodels** procedure with $opp(l)$ too. Hence there is no point in pursuing the path of addition of $opp(l)$, as we are sure to find the same conflict again. In that sense, it is possible to backjump to the previous choice point in the procedure instead of wasting searching the path along $opp(l)$. Hence the savings obtained by addition of *backjumping* into the **smodels** procedure.

2.5 Relation of smodels to earlier systems

The **smodels** algorithm derives its roots from the well-known Davis-Putnam (DP) procedure [9, 10] for solving the SAT problem. The SAT problem [8] is defined as follows:

Let V be a set of boolean variables $\{v_1, v_2, \ldots, v_n\}$. A literal v is of the form v_i or $\overline{v_i}$ where v_i is a boolean variable. A SAT problem P consists of a set C of clauses C_i each clause being a collection of literals. A truth assignment t is a function t: $V \rightarrow \{T, F\}$, where T and F represent the boolean constants *true* and *false* respectively. A variable v_i is satisfied by t if $t(v_i) = T$. A literal $\overline{v_i}$ is satisfied if $t(v_i) = F$. On the other hand a variable v_i is contradicted by t if $t(v_i) = F$, and a literal $\overline{v_i}$ is contradicted if $t(v_i) = T$. A clause C_i, which is a collection (disjunction) of literals is satisfied by t iff at least one of the literals in C_i is satisfied by t. A clause C_i is contradicted by t iff all the literals in C_i are contradicted by t. A set of clauses C is satisfied by t iff all the clauses in C are satisfied by t. A set of clauses C is contradicted by t iff at least one of the clauses in C is contradicted by t. The SAT decision problem is stated as follows [20, 8]:

Definition 7 (SAT) *Given a set V of boolean variables (each variable v_i is present either as a literal v_i or a literal $\overline{v_i}$), and a collection of clauses C over V, where each clause represents a disjunction of literals, is there a satisfying truth assignment t for C?*

In [8] SAT was shown to be NP-complete. Most implementations of SAT return a satisfying assignment for a SAT problem if any exists, otherwise return false.

```
function DP($C, t$)
if $C$ is satisfied by $t$ then
        return $(T, t)$
else if $C$ is contradicted by $t$ then
        return $(F, t)$
else
        Pick a variable $v$ such that $v$ does not have a value in $t$
        $t = t \cup \{v = T\}$
        $(I, t') =$ DP($C, t$)
        if $(I == T)$ then
                return $(T, t')$
        else
                return DP($C, t \cup \{v = F\}$)
        endif
endif
```

Figure 2.8: Pure Davis-Putnam procedure

2.5.1 Davis-Putnam procedure

DP is a simple and practical procedure for solving the SAT problem. If the set of clauses C in a SAT problem is satisfiable the DP procedure [9] returns a truth assignment t satisfying C, otherwise it terminates without returning any assignment. Let us consider a SAT problem P consisting of a set C of clauses defined over a set V of boolean variables. The basic Davis-Putnam procedure is shown in Fig. 2.8.

The pure DP procedure non-deterministically explores the space of the possible truth assignments over the set of variables X in a SAT problem. The procedure DP is initially called with an empty truth assignment $t = \phi$. In the end the returned t is a truth assignment satisfying C if C is satisfiable.

The procedure of **smodels** bears a strong similarity to the solution method followed in DP. The main points of similarity between the basic **smodels** procedure (Fig.2.3) and the pure DP procedure (Fig. 2.8) are:

- Terminates with a satisfying assignment in both cases if the problem (clause-set or program) has a solution. If there is no solution, the method terminates in both cases indicating the unsatisfiability.

- At any point where branching occurs, in each case a new literal l is chosen non-deterministically. There are exactly two branches corresponding to each such choice, one branch for assignment of T to l (adding l to B) and the other branch for assignment of F to l (adding *opp*(l) to B).

2.5.2 Improvements to basic Davis-Putnam algorithm

The basic procedure of DP assumes no intermediate processing at any stage. But it has been shown in literature that low-order polynomial processing at each stage is a useful mechanism for reducing the number of nodes in the search space of DP. The basic idea behind any such pre-processing algorithm is to process a set of clauses C based on the latest truth value assignment and return a new set of clauses C' which is simpler than C to solve. This type of pre-processing is commonly termed as *propagation*. A generic version of the DP algorithm with *propagation* is given in Fig. 2.9.

The basic **smodels** algorithm presented in Fig. 2.3 bears a closer resemblance to this version of DP with *propagate* as shown in Fig. 2.9. In addition to the two main similarities to the pure DP algorithm mentioned before, the call to **expand** in **smodels** after every non-deterministic choice is similar to the call to a *propagate* function after the addition of a new variable assignment in the DP procedure. The **expand** function is applied in **smodels** with the intent of simplifying the problem before another recursive call, as is also the case in application of the *propagate* function in DP.

Unit resolution

The most common propagation algorithm presented in DP [9, 10] is the *unit resolution* method. In *unit resolution* the effects of simple facts are propagated and used to subsume clauses. If the set of clauses C in SAT contains a unit clause (that is, a

```
function DP($C, t$)
if $C$ is satisfied by $t$ then
        return $(T, t)$
else if $C$ is contradicted by $t$ then
        return $(F, t)$
else
        Pick a variable $v$ such that $v$ does not have a value in $t$
        $t = t \cup \{v = T\}$
        $C'$ = propagate($C, t$)
        $(I, t') =$ DP($C', t$)
        if $(I == T)$ then
                return $(T, t')$
        else
                $t = t \cup \{v = F\}$
                $C'$ = propagate($C, t$)
                return DP($C', t$)
        endif
endif
```

Figure 2.9: Davis-Putnam procedure with propagation

clause consisting of a single literal v or $\overline{v}$), the clause can only be made true by one of the two possible assignments of truth value to the variable v. Hence this assignment can be made without branching and the resulting formula can be simplified based on this assignment. This procedure is termed as *unit resolution*. Since *unit resolution* only decreases the number of clauses in a formula, it terminates leaving a much simpler set of clauses C'.

Consider the set of clauses $C = \{(x), (\overline{x}, y), (y, z)\}$. We know x to be true and propagating x to the second clause we obtain that y is true, which we then find that it subsumes the last clause. In this case the theory is solved by the propagation alone.

Relation between expand and Unit resolution

In the basic **smodels** procedure, a call is made to **expand** after every choice of a new literal l in order to add as many new literals as possible to B as possible. This is similar to the call to a propagation algorithm in DP before the choice of a new literal. The technique of unit resolution is closely related to the technique of **expand** in **smodels** both being propagation algorithms. Here we show how the inference rules of **expand** can be interpreted in terms of application of *unit resolution* to logic program rules.

A ground logic program rule $h \leftarrow l_1, \ldots, l_n$ can be interpreted as a SAT clause $C_i=(h, opp(l_1), opp(l_2), \ldots, opp(l_n))$. Likewise a set of literals B can be interpreted by a truth assignment t which assigns the values to the variables in B such that all the literals in B are true. *Unit resolution* can then be used to explain the specific inference rules used in the definition of $Cl(P, B)$ as part of **expand**.

Consider IR1. If $l_1, \ldots, l_n$ are all true in B, then the corresponding truth assignment t satisfies all literals $l_1, \ldots, l_n$. Since t contradicts all other literals $opp(l_1), opp(l_2), \ldots, opp(l_n)$ in C_i, the clause C_i reduces to (h). Since this is a unit clause, by *unit resolution*, t must be extended so that h is assigned T. This is equivalent to addition of h to exisiting B, as dictated by rule IR1.

Consider IR4. Since B contains the literals *not* $h, l_1, \ldots, l_{i-1}, l_{1+1}, \ldots, l_n$, the truth assignment corresponding to B contradicts the literals $h, opp(l_1), \ldots, opp(l_{i-1}), opp(l_{i+1}), \ldots, opp(l_n)$. Thus the clause C_i reduces to $(opp(l_i))$. This is a unit clause, and by *unit resolution* t must be extended so that $opp(l_i)$ is true. This is equivalent to the addition of $opp(l_i)$ to the exisiting set B.

Failed literals

A more advanced technique employed in DP is to test for *failed literals* [13]. If setting a variable v to true and propagating results in a contradiction then v must be false in any model. Hence we can just add $\overline{v}$ to the truth assignment, and propagate the results of this. The effect of this is like a local propagation step applied to the current point in the search space. The examination of potential sources of *conflict* requires a linear pass through the set of variables, coupled with the application of propagation at each step. This is still a low-order polynomial pre-processing step, hence can be successfully used to reduce the search space of the DP algorithm.

The concept of **lookahead** function in **smodels** derives itself from the above technique of *failed literals* used in DP. In **lookahead** of **smodels**, when a literal l when added to B leads to a **conflict**, $opp(l)$ is added to B and the resulting B is propagated. This is exactly same as the above technique of *failed literals* employed in DP for solving SAT.

2.6 Summary

In this chapter, we explored some of the common finite constraint satisfaction problem solving techniques. Later some of the existing semantic notions of *solution* in over-constrained systems are studied in detail. Finite function-free normal logic programs were then studied with stress on stable models. The language LP_{SM} is introduced with stress on **smodels**, the procedural engine of LP_{SM} responsible for generation of stable models. In the end we presented the techniques for solving the SAT problem in literature from which **smodels** derives its algorithms.

Chapter 3

Constraint programming in LP_{SM} and smodels

In this chapter, we shall address two main issues involved in the specification and solving of constraints using logic programming with stable models (SLP). In the earlier part, we shall evaluate SLP as a modeling paradigm for specifying and solving constraints. In the latter part, we shall evaluate different representations and languages based on SLP for solving the finite CSP problem.

In the first part of this chapter we shall review some existing modeling languages for specifying constraints, by comparing the relative ease of modeling one specific constraint satisfaction problem in each of these languages. Later, we show that the stable logic programming paradigm (SLP) shares the declarative nature of these existing modeling languages. Further we show some additional advantages of SLP over other modeling languages which makes it an attractive candidate for specifying and solving dynamic CSPs.

In Section 3.3 we explore the modeling of finite CSPs in LP_{SM} as suggested in [41] by Niemela. Later in Section 3.4 we solve finite CSPs by applying **smodels** directly to ground logic programs representing finite CSPs. This direct method of solving finite CSPs is then compared theoretically and experimentally with chronological backtracking of CSPs and possible improvements are suggested.

3.1 Constraint modeling languages

In constraint programming, bulk of research in the past few years has concentrated on development of efficient solving techniques. In the recent past the thrust has shifted to the modeling aspects of constraint programming. The conventional model of finite CSPs is restricted in its modeling power. To overcome the representational limitations of the finite CSP paradigm, enriched models of constraint programming have been proposed to model constraints with more general semantics. Among these formalisms are constraint logic programming (CLP) [27], concurrent constraint programming (CCP) [53], abductive constraint logic programming (ACLP) [29] and logic programming with stable models (SLP) [41, 37].

In the recent past a multitude of languages have been developed for addressing the modeling issues in respresenting constraint satisfaction problems. These languages are based on the programming paradigms like CLP, CCP, SLP etc. Some of the languages which have been used in the modeling of constraints are Oz [53], Eclipse [38], OPL [26], and LP_{SM}[37]. In this chapter we shall first address the issue of modeling constraints in some of these languages and then try to identify some of the unique modeling features of LP_{SM} as compared to other langauges.

The main attraction of the use of these modeling languages is their declarative nature which supports the natural statement of a problem reducing the gap between the problem specification and the program. Further, a program is typically abstracted away from the low level implementation details of the solver making the model independent of the way it is implemented. In addition, this allows the separation of a model from the instance data, which enables the application of the same model to multiple problem instances. These attractive features of the declarative languages prompted researchers to invent constraint languages with powerful implementations and declarative semantics. This is a feature shared by all the languages which will be explored in this chapter. This makes them an attractive prospect for specifying and solving CSPs if the implementations of these languages are powerful. To illustrate the attractiveness of these features we shall show how to model a commonly used benchmark CSP in each of these modeling languages and show the ease of modeling constraint satisfaction problems in these languages. The *zebra problem* is a classic constraint satisfaction problem which is often used as a benchmark in the constraint programming literature.

The statement of the *zebra problem* is given below:

> Five men with different nationalities live in the first five houses of a street. They practise five distinct professions, and each of them has a favorite drink, and all of them has a favorite animal and a favorite drink, all of them different. The five houses are painted in different colors.
>
> The Englishman lives in a red house.
> The Spaniard owns a dog.
> The Japanese is a painter.
> The Italian drinks tea.
> The Norwegian lives in the first house on the left.
> The owner of the green house drinks coffee.
> The green house is on the right of the white one.
> The sculptor breeds snails.
> The diplomat lives in the yellow house.
> Milk is drunk in the middle house.
> The Norwegian's house is next to the blue one.
> The violinist drinks fruit juice.
> The fox is in a house next to that of the doctor.
> The horse is in a house next to that of the diplomat.
>
> Who owns a zebra and who drinks water?

House Number	1	2	3	4	5
Nationality	Norwegian	Italian	English	Spaniard	Japanese
Color	Yellow	Blue	Red	White	Green
Drink	Water	Tea	Milk	Juice	Coffee
Profession	Diplomat	Doctor	Sculptor	Violinist	Painter
Pet	Fox	Horse	Snails	Dog	Zebra

Table 3.1: Solution to the *zebra problem*

The solution to the problem is given in Table 3.1.

3.1.1 Expressing CSPs in OPL

A recent entry to the set of programming languages for specifying constraint satisfaction problems is OPL [26]. OPL combines the procedures involved in constraint satisfaction and linear optimisation to yield a powerful language where it is possible to specify both problems involving optimisation as well as constraint problems.

To illustrate the modeling power of OPL, let us express the *zebra problem* in OPL. The specification of the *zebra problem* is given in OPL as below in Fig. 3.1:

The encoding of the *zebra problem* in OPL as illustrated in Fig. 3.1 highlights the fundamental modeling power of OPL. The variables with discrete domains here are nationality, pets, color, profession, and drink. Each of them takes pairwise different values from the individual domains. This fact is represented in the program by use of the global constraint *alldifferent* which is in-built into the language of OPL. The global constraint *alldifferent* finds use in a variety of problem-solving applications of constraint programming. Further, the individual constraints in the *zebra problem* can be specified in an easy manner in OPL as illustrated in Fig. 3.1. The *solve* instruction instructs to find a suitable assignment to each of the five arrays so that the constraints in the problem are satisfied. The proximity of the specification of constraint and the constraint semantics is obvious by the way the individual constraints have been encoded in OPL. Take for example, the constraint that "the spaniard owns a dog". The corresponding statement in OPL is written as "nationality[spaniard]=pet[dog]".

```
/* The collection of domain values*/
enum Nations english,spaniard,japanese,italian,norwegian;
enum Colors red,green,white,yellow,blue;
enum Professions painter,sculptor,diplomat,violinist,doctor;
enum Pets dog,snails,fox,horse,zebra;
enum Drinks tea,coffee,milk,juice,water;
/* There are five houses in all */
range house 1..5;
/* Declaration for each discrete variable */
var house nationality[Nations];
var house color[Colors];
var house pet[Pets];
var house profession[Professions];
var house drink[Drinks];

solve {
/* Each discrete variable has pairwise different value
assignment*/

alldifferent(nationality);
alldifferent(color);
alldifferent(pet);
alldifferent(profession);
alldifferent(drink);

/* Problem-specific constraints of the zebra problem*/
drink[milk]=3;
nationality[norwegian]=1;
nationality[english] = color[red];
nationality[spaniard]=pet[dog];
nationality[japanese] = profession[painter];
nationality[italian] = drink[tea];
color[green] = drink[coffee];
color[green] = color[white] + 1 ;
profession[sculptor] = pet[snails];
profession[diplomat] = color[yellow];
nationality[norwegian] = color[blue] + 1 ∨
nationality[norwegian] = color[blue] - 1 ;
profession[violinist] = drink[fruit];
abs(profession[doctor] - pet[fox] ) = 1;
abs(profession[diplomat] - pet[horse]) = 1 ;

}
```

Figure 3.1: Expressing the *zebra problem* in OPL

This shows the declarative nature of constraint programming in OPL.

In OPL, the declarative treatment of constraints makes it easy to specify problems irrespective of the underlying constraint satisfaction method. This feature makes OPL as attractive as the other languages presented in this section.

3.1.2 Eclipse for modeling CSPs

Eclipse [38] is a language based on the constraint logic programming (CLP) paradigm. It offers a language for specifying and solving combinatorial problems including constraint satisfaction problems. The langauge offers two levels of flexibility in specification of problems in solving CSPs. The first level of fexibility corresponds to the abovementioned declarative nature of these languages. The specification of the "conceptual model" in an Eclipse program is precisely meant to capture the problem statement of a CSP. This satisfies the desired quality of natural problem specification of a problem. Let us capture the often mentioned *zebra problem* in the langauge Eclipse.

It is clearly evident that the conceptual model of Eclipse offers the required level of abstraction for specifying constraint satisfaction problems with the clear distinction between the implementation details and the problem statement. The implementation of the *zebra problem* is provided in Fig. 3.2. The solution is to define 5x5 integer variables for each mentioned item, to number the houses from one to five and to represent the fact that e.g. Italian drinks tea by equating Italian = Tea. The value of both variables represents then the number of their house.

3.1.3 Constraint programming in Oz

Oz [53] is a concurrent langauge which enables functional, object-oriented, and constraint programming. One of the main advantages shared by Oz which favours its use in constraint programming is its declarative nature, which makes it a easy language for specifying constraints. To illustrate the point let us implement the *zebra problem*

```
:- lib(fd).

zebra([zebra(Zebra), water(Water)]) :-
Sol = [Nat, Color, Profession, Pet, Drink],
Nat = [English, Spaniard, Japanese, Italian, Norwegian],
Color = [Red, Green, White, Yellow, Blue],
Profession = [Painter, Sculptor, Diplomat, Violinist, Doctor],
Pet = [Dog, Snails, Fox, Horse, Zebra],
Drink = [Tea, Coffee, Milk, Juice, Water],

Nat :: 1..5,
Color :: 1..5,
Profession :: 1..5,
Pet :: 1..5,
Drink :: 1..5,
alldifferent(Nat),
alldifferent(Color),
alldifferent(Profession),
alldifferent(Pet),
alldifferent(Drink),

English = Red,
Spaniard = Dog,
Japanese = Painter,
Italian = Tea,
Norwegian = 1,
Green = Coffee,
Green #= White + 1,
Sculptor = Snails,
Diplomat = Yellow,
Milk = 3,
Dist1 #= Norwegian - Blue, Dist1 :: [-1, 1],
Violinist = Juice,
Dist2 #= Fox - Doctor, Dist2 :: [-1, 1],
Dist3 #= Horse - Diplomat, Dist3 :: [-1, 1],

flatten(Sol, List),
labeling(List).
```

Figure 3.2: Capturing the *zebra problem* in Eclipse

in Oz. The implementation of the *zebra problem* is provided in Fig. 3.3. The houses are numbered from 1 to 5, where 1 is the first and 5 is last house in the street. There are 25 different properties (i. e. hosting an Englishman, being the green house, hosting a painter, hosting a dog, or hosting someone who drinks juice), and each of these properties must hold for exactly one house. The properties are partitioned into five groups of five members each, where the properties within one group must hold for different houses. The model has one variable for each of these properties, where the variable stands for the number of the house for which this property holds.

The solution in Fig 3.3 constrains the root variable Nb to a record that maps every property to a house number between 1 and 5.

The script introduces two defined constraints. The defined constraint {Partition Group} says that the properties in the list Group must hold for pairwise distinct houses. The defined constraint {Adjacent X Y} says that the properties X and Y must hold for houses that are next to each other. The statement {FD.distance X Y = 1} creates a propagator for $| X - Y |=1$.

3.2 SLP for constraint modeling and programming

We concern ourselves with the frameworks of logic programming based on stable models [37, 41]. In the recent past, languages based on stable models for logic programs have been developed for expressing and solving problems in artificial intelligence. The notable among these languages are the LP_{SM} language developed by Niemela in [41], and the stable logic programming (SLP) paradigm developed by Marek and Truszczynski in [37]. In [41], LP_{SM} is shown to be an effective language for modeling constraint satisfaction problems, planning problems, and other computational problems in artificial intelligence. On similar lines, in [37] the SLP paradigm was used to model various constraint satisfaction problems.

Fundamental to use of logic programming languages in modeling constraints are the following main ideas:

```
proc {Zebra Nb}
        Groups = [ [english spanish japanese italian norwegian]
            [green red yellow blue white]
            [painter diplomat violinist doctor sculptor]
            [dog zebra fox snails horse]
            [juice water tea coffee milk] ]
        Properties = {FoldR Groups Append nil}
        proc {Partition Group}
            {FD.distinct {Map Group fun {$ P} Nb.P end}}
        end
        proc {Adjacent X Y}
            {FD.distance X Y '=:' 1}
        end
        in
            /* Nb maps all properties to house numbers*/
            {FD.record number Properties 1..5 Nb}
            {ForAll Groups Partition}
            Nb.english = Nb.red
            Nb.spanish = Nb.dog
            Nb.japanese = Nb.painter
            Nb.italian = Nb.tea
            Nb.norwegian = 1
            Nb.green = Nb.coffee
            Nb.green >: Nb.white
            Nb.sculptor = Nb.snails
            Nb.diplomat = Nb.yellow
            Nb.milk = 3
              {Adjacent Nb.norwegian Nb.blue}
            Nb.violinist = Nb.juice
              {Adjacent Nb.fox Nb.doctor}
              {Adjacent Nb.horse Nb.diplomat}
            Nb.zebra = Nb.white
            {FD.distribute ff Nb}
        end
```

Figure 3.3: Capturing the *zebra problem* in Oz

1. The presence of an efficient implementation of stable models like **smodels** [54] at the back-end,

2. The interpretation of logic program rules as constraints on the solution set,

3. The use of restricted first order representations of logic programs so that the computation is not thrown out of bounds and yet the representation has the capacity to model a large class of constraint problems, and

4. The correspondence between stable models of the logic programs and solutions to the constraint problem.

5. The paradigm differs from the conventional view of logic programming followed in other extensions like CLP etc. The conventional view of logic programming is based on the idea that the user specifies a target goal clause, and an explanation of the target goal is desired as the output. In that sense, in CLP and other logic programming we are not interested in generating the models of the whole program. This has an effect on the solving procedures for CLP and similar systems. CLP and similar paradigms follow top-down goal-directed procedures for generating explanations of a goal. In contrast, the computation based on stable models here is directed at computation of complete models for the program. There is no concept of a goal directed computation of stable models here.

As studied in the previous chapter, the only additional features of LP_{SM} which were not present in the usual function-free normal logic programs (FFNLPs) were the *mathematical* operators. But in the extreme case, it is possible to represent the mathematical operators in LP_{SM} on variables with finite domains by a finite extensional set of all satisfying tuples. In that sense the power of LP_{SM} is just in the compact representation of such functions. Semantically it is possible to generate a FFNLP corresponding to a LP_{SM} program. Hence without loss of generality, we stress on use of LP_{SM} in this chapter even though the discussion is typical of any

FFNLP. So the references to LP_{SM} in this chapter should be viewed as representative of the class of FFNLPs.

3.2.1 Declarative nature of stable logic programming

In [41] LP_{SM} was used to model the finite CSP specification and other specific constraint satisfaction problems ($n-queens$, *schur* etc). We follow the methodology given in [41], to illustrate the decalarative nature of LP_{SM} to develop a non-trivial example of constraint programming. In that sense SLP offers langauges with similar declarative nature as the languages explained in the previous section, which enables use of SLP in easy modeling of constraint programming situations. Recently LP_{SM} has been extended with more additional features like weighted rules, choice rules etc. in [43]. However, our discussion in this section is limited to the version in [41].

Let us study how to model the *zebra problem* in LP_{SM}. The formulation of the *zebra problem* shall help us understand the modeling of general constraint programming problems in LP_{SM}. To model this in LP_{SM} assume a set of five domain predicates p, n, c, pe, and d (representing the profession, nationality, color, pet, and drink respectively) each having the same domain $d = \{1, 2, 3, 4, 5\}$. The domain of the predicate p represents the ordered set $\{painter, sculptor, diplomat, violinist, doctor\}$, each element represented by its index. Likewise, n represents the ordered set $\{english$ $, spaniard, japanese, italian, norwegian\}$, c represents the ordered set $\{red, green,$ $white, yellow, blue\}$, pe represents the ordered set $\{dog, snails, fox, horse, zebra\}$, and d represents the ordered set $\{tea, coffee, milk, juice, water\}$. A domain predicate *house* represents the corresponding house number and has the domain $d = \{1, 2, 3, 4, 5\}$. With this set-up, the LP_{SM} program in Figure 3.4 represents the *zebra problem.*

In addition to the domain predicates, the predicates *color*, *nationality*, *drink*, *pet*, and *profession* are used to specify the constraints of the problem and to return the solution. An atom of the form $color(X, Y)$ in a stable model indicates the assignment

```
% Domain Predicates

n(1..5). c(1..5). pe(1..5). p(1..5). d(1..5). house(1..5).
% Mutual Exclusion Rules

nationality(X,Y) ← house(X), n(Y), not negnationality(X,Y).
negnationality(X,Y) ← n(Y),house(X), n(Y1), nationality(X,Y1), Y1 ≠ Y.
color(X,Y) ← house(X), c(Y), not negcolor(X,Y).
negcolor(X,Y) ← c(Y),house(X), c(Y1), color(X,Y1), Y1 ≠ Y.
profession(X,Y) ← house(X), p(Y), not negprofession(X,Y).
negprofession(X,Y) ← p(Y), house(X), p(Y1), profession(X,Y1), Y1 ≠ Y.
pet(X,Y) ← house(X), pe(Y), not negpet(X,Y).
negpet(X,Y) ← pe(Y), house(X), pe(Y1), pet(X,Y1), Y1 ≠ Y.
drink(X,Y) ← house(X), d(Y), not negdrink(X,Y).
negdrink(X,Y) ← d(Y),house(X), d(Y1), drink(X,Y1), Y1 ≠ Y.
% All predicates have pairwise different values.
← house(X), n(Y), house(Z), nationality(X,Y), nationality(Z,Y), X ≠ Z.
← house(X), c(Y), house(Z), color(X,Y), color(Z,Y), X ≠ Z.
← house(X), p(Y), house(Z), profession(X,Y), profession(Z,Y), X ≠ Z.
← house(X), pe(Y), house(Z), pet(X,Y), pet(Z,Y), X ≠ Z.
← house(X), d(Y), house(Z), drink(X,Y), drink(Z,Y), X ≠ Z.
% Problem Specific Constraints

drink(3,3).
nationality(1,5).
← house(X),c(Y),color(X,Y),nationality(X,1),Y ≠ 1.
← house(X),pe(Y),pet(X,Y),nationality(X,2),Y ≠ 1.
← house(X),p(Y),profession(X,Y),nationality(X,3),Y ≠ 1.
← house(X),d(Y),drink(X,Y),nationality(X,4),Y ≠ 1.
← c(Y),house(X1),drink(X1,2),color(X1,Y),Y ≠ 2.
← p(X),house(X1),profession(X1,X),pet(X1,2),X ≠ 2.
← d(X),house(X1),drink(X1,X),profession(X1,4),X ≠ 4.
← c(X),house(X1),color(X1,X),profession(X1,3),X ≠ 4.
← color(X,2),house(X),house(Y),color(Y,3),X ≠ Y + 1 .
← house(X),house(X1),nationality(X,5),color(X1,5),abs(X-X1) ≠ 1.
← house(X),house(X1),pet(X,3),profession(X1,5),abs(X-X1) ≠ 1.
← house(X),house(X1),profession(X,3),pet(X1,4),abs(X-X1) ≠ 1.
```

Figure 3.4: Capturing the *zebra problem* in LP_{SM}

of the color number Y to house number X in the solution, e.g. *color*$(1,4)$ indicates that the first house is yellow in color. Likewise for the other predicates. There are three sets of rules in the LP_{SM} program:

1. Mutual exclusion rules stating that each house can have only one color, one profession, one pet, one drink, and one nationality.

2. Rules stating that the houses have different colors, different drinks, different pets, different professions, and different nationalities.

3. Problem specific constraints which are translated to the corresponding rules in the program.

The detailed discussion of each type of constraint and their representation in LP_{SM} is explained in the sections to follow. The program returns a stable model iff an assignment exists such that all the constraints of the problem are satisfied. The returned stable model contains instances of predicates *color*$(X,Y$, *drink*(X,Y), *nationality*(X,Y), *pet*(X,Y), and *profession*(X,Y). There is only stable model of the program. So the problem has only one solution. The lone stable model contains the atoms $\{$*color*$(1,4)$, *color*$(2,5)$, *color*$(3,1)$, *color*$(4,3)$, *color*$(5,2)\}$ indicating that the the houses 1,2,3,4, and 5 are assigned the colors *yellow*, *blue*, *red*, *white*, and *green* respectively.

3.2.2 Advantages of Stable Logic Programming

In this section till now we studied the declarative nature of SLP. The declarative nature makes it a good choice for modeling constraints because of the lesser distance between a program and the meaning of a constraint. This is particularly useful in situations where it is constantly required to change the specifications of the constraints. The re-formulation of a constraint program in such situations becomes easy if the underlying modeling laguage is declarative in nature. Hence in a language based on SLP like LP_{SM}, it is easy to change the problem specifications of a constraint problem with ease in case of change of problem definition. In addition there are certain constraint programming situations in which the paradigm of SLP proves to be a bright candidate for specifying and solving constraint problems.

Dynamic constraint satisfaction problems

Dynamic constraint satisfaction problems (DCSPs) [56, 57] are a generlization of the class of CSPs. In DCSPs, the set of variables in a solution changes dynamically on the basis of intermediate assignments of values to variables. This dynamic nature of problem space in DCSPs makes them a generalization of the general finite CSP paradigm. The finite CSPs can be considered a special case of the more general DCSPs because they are merely static version of DCSPs. The discussion in this section is taken mainly from [56, 57], where LP_{SM} has been shown to be a useful langauge for solving and specifying product configuration problems.

Formally, an instance P of a DCSP [40] is of the form $< V, D, V_I, C_C, C_A >$, where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of variables and $D=\{d_1, d_2, \ldots, d_n\}$ is the set of domains of each of the variables in V. The set $V_I \subseteq V$ is the set of *initial* constraints, C_C is the set of *compatibility* constraints, and C_A is the set of *activity* constraints. The *compatibility* constraints specify the set of allowed combinations of values for a set of variables similar to the case in a finite CSP. They are usually specified by a subset of the cartesian product of the domains of the variables. On the other hand, an *activity* constraint is of the type $c \rightarrow v$, where c is a constraint like the *compatibility* constraints, and v is a variable. Here not all variables need to have a value (be active) in all solutions but the activity of variables is controlled by the *activity* constraints. An *activity* constraint $c \rightarrow v$ states that the variable v needs to be active (have a value) in a solution where constraint c is satisfied.

An assignment of values A in a DCSP P need not assign a value to all the variables in V. Any assignment A assigns values to a subset of V, and all variables assigned values by A are said to be *active* in A. An assignment A satisfies P iff it satisfies all the compatibility constraints and activity constraints in P. A satisfies a constraint $c \in C_C$ iff all the variables in c are *active* in A and the tuple of values assigned to the active variables in c by A is allowed by c. This is exactly same as a constraint in a finite CSP case. On the other hand, an assignment A satisfies an *activity* constraint $c \rightarrow v$ iff whenever the compatibility constraint c is satisfied by A then the variable v is *active* in A. An assignment A is a *solution* to the DCSP P iff (a). All the variables $v \in V_I$ are *active* in A, (b). A satisfies all the compatibility and activity constraints in P, and (c). A is subset minimal.

Clearly the main difference between the DCSP paradigm and the CSP paradigm is the presence of *activity* constraints. These *activity* constraints are seem to be essential in various domains including product configuration and are hard to handle in standard CSP setting. DCSP can capture the elements of product configuration in a concise and succinct manner as shown in [57].

Stable logic programming languages like LP_{SM} can straigthforwardly capture the DCSP problems because of the unique language features in LP_{SM}. The dynamic nature of solution space where a new variable is dynamically activated based on a partial solution in the intermediate stage in the solution process, is easily captured by means of *default* and *conditional* rules in LP_{SM}. To illustrate how this type of knowledge can be captured in LP_{SM}, consider the following example of product configuration in LP_{SM}.

Product configuration is the process of finding a specification of a product in an industry as a collection of predefined components. The model of the configuration describes the various components in the configuration, the rules on how the components interact, and the requirements of the product individuals in terms of conditions of components. The output is a configuration which is a description of the process yielding the individual product specifying the components used in the process. Let us consider a small manufacturing unit where steel toys of various shapes are made. Two kinds of machines (*lathes* and *milling machines*) represented by variables *lathe*

```
lathe(l1) ← not lathe(l2), not lathe(l3), lathe-needed.
lathe(l2) ← not lathe(l1), not lathe(l3), lathe-needed.
lathe(l3) ← not lathe(l2), not lathe(l1), lathe-needed.

millmachine(m1) ← not millmachine(m2), mill-needed.
millmachine(m2) ← not millmachine(m1), mill-needed.

shape(cylinder) ← not shape(sphere).
shape(sphere) ← not shape(cylinder).

lathe-needed ← shape(cylinder).
mill-needed ← shape(sphere).

sat(c) ← lathe(l1), millmachine(m2).
sat(c) ← lathe(l1),millmachine(m3).

← not sat(c).
```

Figure 3.5: Product configuration example

and *millmachine* respectively, are used for producing different shapes. Consider a sample unit which produces two shapes *cylinder* and *sphere*. Further let the *activity* constraints state that *cylinder* shape requires only *lathe* machines while *sphere* requires only *milling machines*. Let there be three *lathe* machines $(l1, l2, l3)$ and two *milling machines* $(m1, m2)$. Further the size restrictions on the unit give the following *compatibility* constraint $c(lathe, millmachine) = \{(l1, m2), (l2, m1)\}$. This can be captured in LP_{SM} by the program shown in Fig. 3.5. The variables $lathe - needed$ and $mill - needed$ are used to identify the situations when the appropriate machines are required. Here it is evident that all solutions need not have values for both the variables *lathe* and *millmachine*. In case of manufacturing a *sphere* shaped object there is no value for *lathe* in the solution. Further the *compatibility* constraint c is needed to be satisfied by every solution, a fact enforced by an integrity constraint in LP_{SM}.

Dynamic situations involving constraints seem to be captured easily in LP_{SM}. An encoding of the familiar *hamiltonian cycle* problem in LP_{SM} appears in [41]. This is a problem where you seem to need to combine constraints dynamically (choosing edges leaving and entering nodes to form chains and making sure that there is a unique chain visiting all nodes exactly once and returning to its starting point). Hence LP_{SM} seems to be a promising representation langauage for constraint programming situations involving dynamic constraints.

3.3 Capturing finite constraint satisfaction by LP_{SM}

In [41] it is shown that finite CSPs can be captured in LP_{SM}, the clausal programming language based on the **smodels** system for computation of stable models. In this section, we shall present the translation method presented in [41] for representing a finite CSP in LP_{SM}. Let the translation algorithm presented be termed as *transl*.

3.3.1 Translation algorithm

The translation algorithm *transl* proposed by Niemela in [41] is described in Figure 3.6.

The two-stage process of solving a finite CSP P thus involves using *transl* to generate an equivalent LP_{SM} program P' corresponding to P and later solving P' by using the combination of the **lparse** and **smodels** components of LP_{SM} [41]. **lparse** first parses P' to a ground program which is then fed to **smodels** to generate a stable model of P' if one exists.

There are four sets of rules (and facts) in the output of *transl* corresponding to a finite CSP P:

Domain facts These are the facts of the form $d_i(val_i) \leftarrow$ added to denote that val_i is in the domain d_i of the variable v_i in P.

Unique name axiom rules These rules define two predicates $v_i(X)$ and $ov_i(X)$ corresponding to each variable v_i. Effectively $v_i(X)$ represents the assignment of value X to variable v_i in P, and $ov_i(X)$ is true whenever v_i is assigned a value

Input: A finite CSP P involving a set $X = \{v_1, v_2, \ldots, v_n\}$ of n variables, and a set C of constraints, where each variable $v_i \in X$ takes a value from its finite domain d_i and each constraint $c_i \in C$, is defined on a subset X_c of the set of variables X, is a relation expressed as a subset R_c of the cartesian product $\prod_{v_i \in X_c}[d_i)]$.
Output: A LP_{SM} program P' such that the stable models generated by the ground program translation of P' correspond to the solutions of the CSP P.
Procedure: In P' for each domain value t in P a constant t is adopted, and for each domain d_i in P a one-place predicate d_i is used. Moreover for each constraint c in P a constant c is adopted. For each variable v_i in P two one-place predicates v_i and ov_i are used. Two special purpose one-place predicates *sat* and *constraint* are used in the translation too. Given these, P' is the set of clauses formed by the following steps.

- For each domain d_i in P, a set of facts $d_i(val_1) \leftarrow, \ldots, d_i(val_m) \leftarrow$, is added to P' where $val_1, \ldots, val_m$ are the possible values in domain d_i.

- For each variable v_i with the domain d_i in P, to allow a unique value from d_i, we add

  $v_i(X) \leftarrow d_i(X), not\ ov_i(X)$

  $ov_i(X) \leftarrow d_i(X), d_i(Y), v_i(Y), X \neq Y$

  where the predicate $ov_i(X)$ expresses the fact that the variable v_i has some other value than X.

- For each constraint c defining a set of allowed value combinations for a set of variables $v_1, \ldots, v_j$ a fact $constraint(c) \leftarrow$ is added.

- For each allowed value combination in c of variables $v_1, \ldots, v_j$ (on which c is defined) of the form $\{v_1 = val_1, \ldots, v_j = val_j\}$ a rule

  $sat(c) \leftarrow v_1(val_1), \ldots, v_j(val_j)$

  is added.

- Finally an integrity constraint rule $\leftarrow constraint(C), not\ sat(C)$, is added to indicate that all constraints must be satisfied.

Figure 3.6: *transl* - an algorithm for translation of a finite CSP to LP_{SM}

other than X. The unique name rules assert that at any time any variable v_i can be assigned only one value.

Constraint satisfaction rules These rules represent the tuples in each constraint. Any such rule corresponding to a constraint c_i has $sat(c_i)$ at the head.

Constraint facts Facts of the form $constraint(c_i) \leftarrow$ corresponding to each constraint c_i in P are added.

In addition to the above groups of rules and facts, the final integrity constraint states that for any constraint c, $sat(c)$ has to be true in any stable model, thereby indicating that all constraints must be satisfied.

Example for solving a finite CSP by LP_{SM}

We shall illustrate the algorithm *trans*1 by an example with detailed analysis of all the steps involved in the solving of a CSP. Consider a finite CSP P with four variables v1, v2, v3, and v4 each with domains d1={1}, d2={1,2,3}, d3={1,2}, d4={1,2} respectively. P contains five constraints c1, c2, c3, c4, and c5 defined as follows: c1(v1,v2)={(1,2), (1,1)}, c2(v2,v3) = {(2,1),(3,2)}, c3(v3,v4) = {(1,1),(2,2)}, c4(v2,v4) = {(2,1)} and c5(v1,v4) = {(1,1)}. The output P' of *trans*1 is given in Figure 3.7. The LP_{SM} program in Figure 3.7 when fed to **lparse** returns the ground program shown in Figure 3.8.

The output of the parser **lparse** shown in Figure 3.8, expands the non-ground rules corresponding to the unique names axioms in the original program. For each ground term $v_i(val_j)$, the unique name axiom asserts that it is true as long as there is no other value assigned to v_i. This is asserted by making sure that the predicate $ov_i(val_j)$ is not true. $ov_i(val_j)$ returns true iff v_i takes on a value other than val_j, e.g. in the above example, $ov2(1)$ returns true whenever the predicates $v2(3)$ or $v2(1)$ return true. This means that $v2(1)$ will return true iff $ov2(1)$ is never true. The remaining rules simply relate to the axioms corresponding to each constraint, namely

```
d1(1).
d2(1). d2(2). d2(3).
d3(1). d3(2).
d4(1). d4(2).

v1(X) ← d1(X), not ov1(X).
ov1(X) ← d1(X), d1(Y), v1(Y), X != Y.
v2(X) ← d2(X), not ov2(X).
ov2(X) ← d2(X), d2(Y), v2(Y), X != Y.
v3(X) ← d3(X), not ov3(X).
ov3(X) ← d3(X), d3(Y), v3(Y), X != Y.
v4(X) ← d4(X), not ov4(X).
ov4(X) ← d4(X), d4(Y), v4(Y), X != Y.

sat(c1) ← v1(1), v2(2).
sat(c1) ← v1(1), v2(1).
sat(c2) ← v2(2), v3(1).
sat(c2) ← v2(3), v3(2).
sat(c4) ← v2(2), v4(1).
sat(c5) ← v1(1), v4(1).
sat(c3) ← v3(1), v4(1).
sat(c3) ← v3(2), v4(2).

constraint(c1).
constraint(c2).
constraint(c3).
constraint(c4).
constraint(c5).

← constraint(C), not sat(C).
```

Figure 3.7: Example output of *transl*

what combination of value assignments to the variables v_i's makes each constraint c_j true. The last set of rules state that all constraints c_j should be satisfied in any stable model of the program. This is ensured by making sure that the literal *not* $\&false$ is present in any stable model of the program.

Execution of the above example

Let us examine the execution of the procedure **smodels** when applied to the above program shown in Figure 3.8. We shall be referring to the *inference rules* 1 to 4 used in the definition of **expand** function of the **smodels** procedure explained in Chapter 2. That will help explain the different steps involved in the derivation of the stable model in this example. Let us denote the inference rules as IR1, IR2, IR3 and IR4 respectively for convenience to denote each of the inference rules in the definition of **expand**. The steps involved in the application of **smodels** are outlined in Figure 3.9.

A close look at the procedure of the previous section reveals that a finite CSP is solved effectively in a three stage process. The first stage generates a LP_{SM} program using the $trans1$ algorithm shown in Figure 3.6. Next, the output of $trans1$ is passed to the parser **lparse** which parses the non-ground program to a ground program. In the third stage, the ground program output by **lparse** is passed to **smodels** to generate a solution.

Use of a high level representation like LP_{SM} in $trans1$ is responsible for the requirement of an additional stage of compilation into a ground logic program before **smodels** can be applied. It would be interesting from an efficiency point of view to see if a direct encoding of finite CSPs in a lower level representation can enhance the efficiency of solving CSPs. In the next section we explore the possibility of directly encoding a finite CSP as a ground logic program. The ground program representing the CSP is then solved by directly applying **smodels** to it. Hence we are able to bypass the stage of translation from the non-ground program to the ground program using the **lparse** parser. We also perform experimental analysis to compare the ef-

```
sat(c1) ← v1(1), v2(2).
sat(c1) ← v1(1), v2(1).
sat(c2) ← v2(2), v3(1).
sat(c2) ← v2(3), v3(2).
sat(c4) ← v2(2), v4(1).
sat(c5) ← v1(1), v4(1).
sat(c3) ← v3(1), v4(1).
sat(c3) ← v3(2), v4(2).
constraint(c5). constraint(c4). constraint(c3).
constraint(c2). constraint(c1).
&false ← &false.
&false ← not sat(c5).
&false ← not sat(c4).
&false ← not sat(c3).
&false ← not sat(c2).
&false ← not sat(c1).
v4(1) ← not ov4(1).
v4(2) ← not ov4(2).
v3(1) ← not ov3(1).
v3(2) ← not ov3(2).
v1(1) ← not ov1(1).
v2(1) ← not ov2(1).
v2(2) ← not ov2(2).
v2(3) ← not ov2(3).
ov4(2) ← v4(1).
ov4(1) ← v4(2).
d4(1). d4(2).
ov3(2) ← v3(1).
ov3(1) ← v3(2).
d3(1). d3(2).
ov2(2) ← v2(1).
ov2(3) ← v2(1).
ov2(1) ← v2(2).
ov2(3) ← v2(2).
ov2(1) ← v2(3).
ov2(2) ← v2(3).
d2(1). d2(2). d2(3). d1(1).
compute 1 { not &false }
```

Figure 3.8: Output of **lparse** applied to the program in Figure 3.7

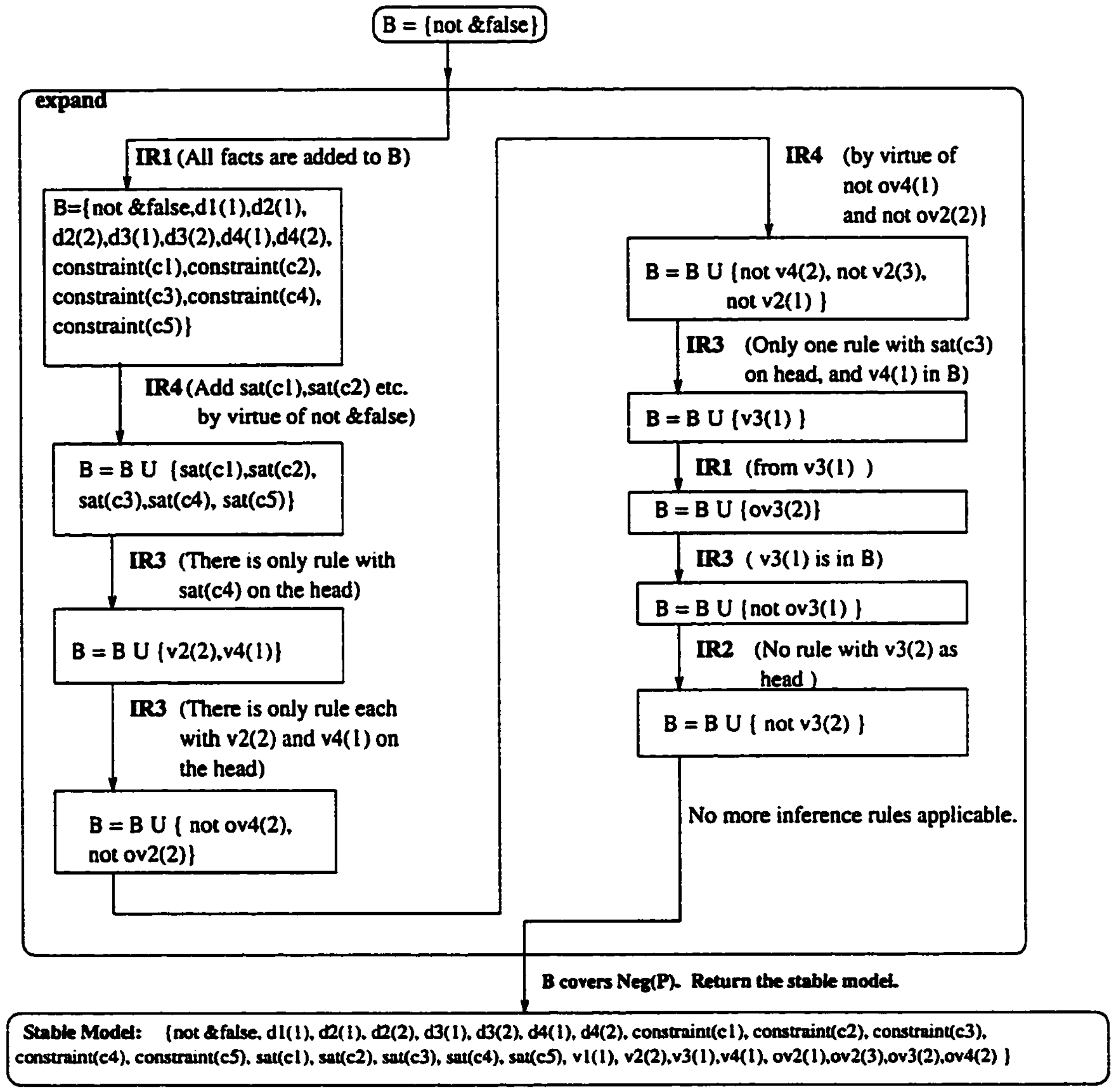

B = {not &false}
expand
IR1 (All facts are added to B)
B={not &false,d1(1),d2(1), d2(2),d3(1),d3(2),d4(1),d4(2), constraint(c1),constraint(c2), constraint(c3),constraint(c4), constraint(c5)}
IR4 (Add sat(c1),sat(c2) etc. by virtue of not &false)
B = B U {sat(c1),sat(c2), sat(c3),sat(c4), sat(c5)}
IR3 (There is only rule with sat(c4) on the head)
B = B U {v2(2),v4(1)}
IR3 (There is only rule each with v2(2) and v4(1) on the head)
B = B U { not ov4(2), not ov2(2)}
IR4 (by virtue of not ov4(1) and not ov2(2)}
B = B U {not v4(2), not v2(3), not v2(1) }
IR3 (Only one rule with sat(c3) on head, and v4(1) in B)
B = B U {v3(1) }
IR1 (from v3(1))
B = B U {ov3(2)}
IR3 (v3(1) is in B)
B = B U {not ov3(1) }
IR2 (No rule with v3(2) as head)
B = B U { not v3(2) }
No more inference rules applicable.
B covers Neg(P). Return the stable model.
Stable Model: {not &false, d1(1), d2(1), d2(2), d3(1), d3(2), d4(1), d4(2), constraint(c1), constraint(c2), constraint(c3), constraint(c4), constraint(c5), sat(c1), sat(c2), sat(c3), sat(c4), sat(c5), v1(1), v2(2),v3(1),v4(1), ov2(1),ov2(3),ov3(2),ov4(2) }

Figure 3.9: Execution of the ground program in Figure 3.8 for the example in Figure 3.7

fectiveness of the **smodels** procedure on the ground programs in each method. We find that experimental results on random finite CSPs show the higher efficiency of the directly encoded ground programs.

3.4 Solving finite CSPs by smodels directly

The translation method for translating a finite CSP P to a ground non-monotonic logic program Π_P is given in Figure 3.10. This program Π_P corresponding to P is then solved by invoking **smodels** directly.

The algorithm *trans*2 outlined in Figure 3.10 generates a ground logic program whose stable models correspond to solutions of a original CSP. We show that *trans*2 is both sound and complete.

Theorem 3.1 *trans2 is sound and complete.*

Proof. Since *trans*2 differs from *trans*1 only in the enforcement of the unique name axiom (any variable can be assigned only one value) and *trans*1 is sound and complete, *trans*2 is both sound and complete. □

The working of *trans*2 is best illustrated by an example.

Consider the same example as in Section 3.3.1. The CSP P has four variables v1,v2,v3 and v4 each with domains d1={1}, d2={1,2,3}, d3={1,2}, d4={1,2} respectively. P contains five constraints c1,c2,c3,c4 and c5 defined as follows: c1(v1,v2)={(1,2), (1,1)}, c2(v2,v3) = {(2,1),(3,2)}, c3(v3,v4) = {(1,1),(2,2)}, c4(v2,v4) = {(2,1)} and c5(v1,v4) = {(1,1)}. P is translated into the ground logic program shown in Figure 3.11.

The program in Figure 3.11 is then sent to **smodels** for derivation of a stable model. The different steps in the generation of a stable model corresponding to the solution is given in Figure 3.12.

Input: A finite CSP P involving a set $X = \{v_1, v_2, \ldots, v_n\}$ of n variables, and a set C of constraints, where each variable $v_i \in X$ takes a value from its finite domain d_i and each constraint $c_i \in C$, is defined on a subset X_c of the set of variables X, is a relation expressed as a subset R_c of the cartesian product $\prod_{v_i \in X_c}[d_i)]$.
Output: A ground logic program Π_P such that the stable models of Π_P correspond to the solutions of the CSP P.
Procedure: In Π_P for each domain value t in P a constant t is adopted Moreover for each constraint c in P a constant c is adopted. For each variable v_i in P a one-place predicate v_i is used. One special purpose one-place predicate *sat* is also used in the translation. Given these, Π_P is the set of clauses formed by the following steps.

- Let the variable v_i in P have the domain $d_i = \{val_1, val_2, val_3, \ldots, val_m\}$ consisting of m possible values $val_1, val_2, \ldots, val_m$. Corresponding to a variable v_i, define a set of m clauses as follows:

  $v_i(val_1) \leftarrow not\ v_i(val_2), not\ v_i(val_3), \ldots, not\ v_i(val_m)$
  $v_i(val_2) \leftarrow not\ v_i(val_1), not\ v_i(val_3), \ldots, not\ v_i(val_m)$
  ...
  $v_i(val_m) \leftarrow not\ v_i(val_1), not\ v_i(val_2), \ldots, not\ v_i(val_{m-1})$
  Define sets of clauses corresponding to each such variable v_i.

  If the domain of a variable v_i contains only one value val_i, add a fact $v_i(val_i) \leftarrow$.

- For each allowed value combination in c of variables $v_1, \ldots, v_j$ (on which c is defined) of the form $\{v_1 = val_1, \ldots, v_j = val_j\}$ a rule

  $sat(c) \leftarrow v_1(val_1), \ldots, v_j(val_j)$

  is added.

- Finally corresponding to each constraint c in P add a rule $sat(c) \leftarrow not\ sat(c)$. This is added to indicate that all constraints need to be satisfied.

Figure 3.10: *trans*2 - Algorithm for translation of a finite CSP to a ground logic program

```
v1(1).
v2(1) ← not v2(2), not v2(3).
v2(3) ← not v2(1), not v2(2).
v2(2) ← not v2(1), not v2(3).
v3(1) ← not v3(2).
v3(2) ← not v3(1).
v4(1) ← not v4(2).
v4(2) ← not v4(1).
sat(c1) ← v1(1), v2(2).
sat(c1) ← v1(1), v2(1).
sat(c2) ← v2(2), v3(1).
sat(c2) ← v2(3), v3(2).
sat(c4) ← v2(2),v4(1).
sat(c5) ← v1(1), v4(1).
sat(c3) ← v3(1), v4(1).
sat(c3) ← v3(2), v4(2).
sat(c1) ← not sat(c1).
sat(c2) ← not sat(c2).
sat(c3) ← not sat(c3).
sat(c4) ← not sat(c4).
sat(c5) ← not sat(c5).
```

Figure 3.11: Example for *trans*2

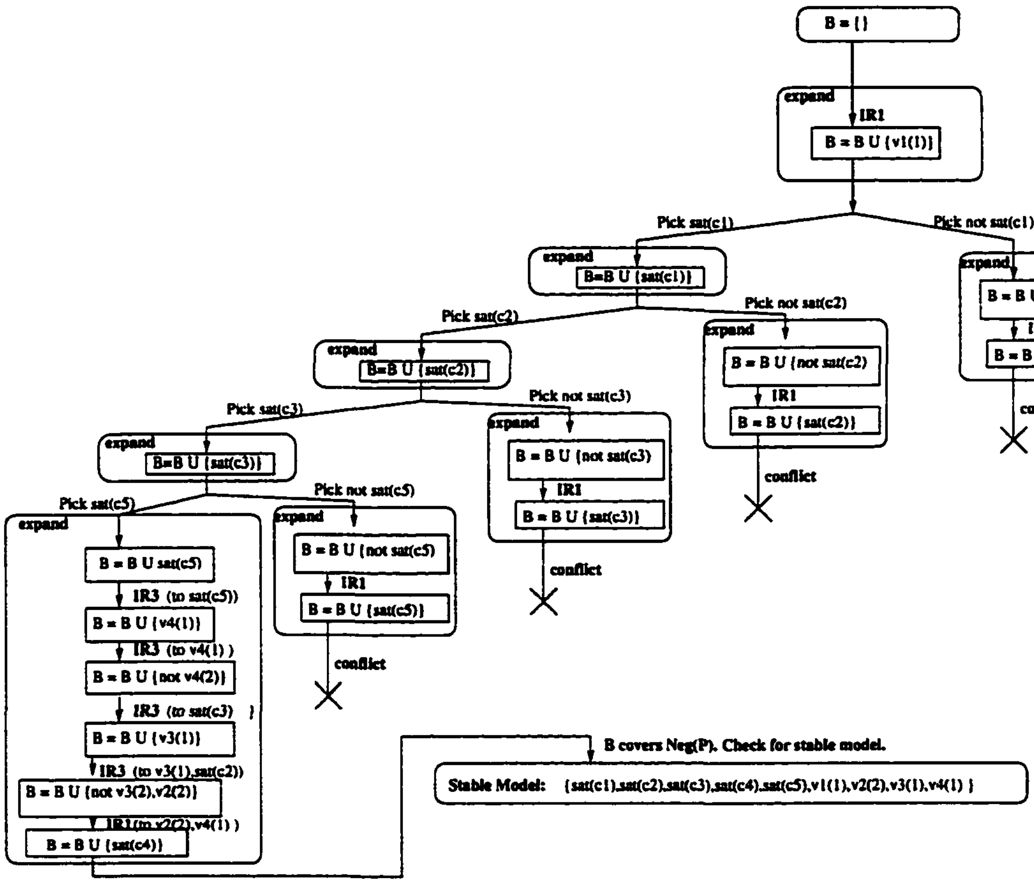


Figure 3.12: Execution of the ground program in Figure 3.11

3.4.1 Experimental comparison of *trans*1 and *trans*2

We performed experiments for a comparison of the relative efficiency of using *trans*1 and *trans*2 in solving finite CSPs. The experiments were performed on randomly generated constraint satisfaction problems. In all the cases only one solution was generated if one existed.

In the experimental set-up, ternary constraints were considered (constraints on 3 variables). The number of variables was fixed at 40 and each variable was considered to have a fixed domain size of 5. The number of constraints was fixed at 15. An ensemble of 20 random finite CSPs was generated for each value of k (number of tuples). The value of k varied from 3 to 125. The average time for each value of k was computed for both representations and plotted against k. In all cases, the time is measured for computing one stable model or reporting that no stable model exists. All experiments were carried on Pentium PII machines with 400 MHz clock, running RedHat Linux version 5.1. The graph in figure 3.13, shows the computing time plotted against k.

The experimental results show that the ground program obtained by applying *trans*2 to a finite CSP provided better results than the ground program generated by applying **lparse** to the LP_{SM} program generated by *trans*1. This can be explained by two main factors:

1. Let P_1 and P_2 represent the ground programs generated in *trans*1 and *trans*2 respectively for the same problem. Consider an assignment of a value val to a variable v_i whose domain is of size m_i. Corresponding to each literal $v_i(val)$ there is a single clause in P_2 while in P_1 corresponding to each $v_i(val)$, there are m_i clauses in P_1 (one clause $v_i(val) \leftarrow not\ ov_i(val)$ and other $m_i - 1$ clauses of the form $ov_i(val) \leftarrow v_i(val')$ where $val' \neq val$).

2. Excess effort is required in propagating the effect of a variable assignment in case of *trans*1. In *trans*2, a variable assignment $v_i = val$ is reflected by addi-

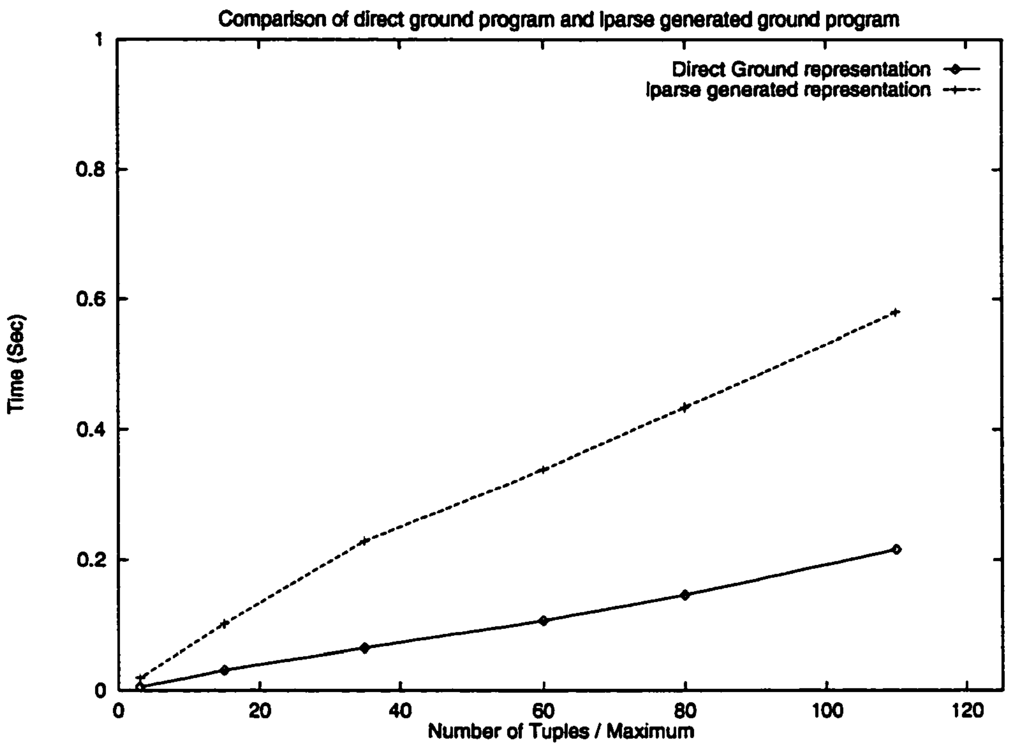


Figure 3.13: Direct ground representation versus **lparse** generated ground representation

tion of $v_i(val)$ to B. This assignment is immediately propagated in one step by application of inference rule IR3 of **expand** causing addition to B of all literals of the form *not* $v_i(val')$, where $val' \neq val$. But in *trans*1, any variable assignment $v_i = val$ is achieved by addition of a literal of the form *not* $ov_i(val)$ to B. This is propagated to add $v_i(val)$ to B in one step. The literals of the form *not* $v_i(val')$, where $val' \neq val$, are then added in a series of steps of application of inference rule IR3 of **expand**. Thus the propagation of an assignment takes a single step in *trans*2 while it takes multiple steps in *trans*1. This is a possible major contributor to the excess time required by *trans*1.

The results in this section concur with the expected results that use of a more specialized language is bound to have a positive effect on the efficiency of a system. On the other hand, use of a more general tool even though makes things expressive, might compromise on efficiency. However, an important point is that a general tool may also be fine-tuned to efficiently process special applications. In the context of finite CSPs, this means that once a methedology of representing constraints is fixed, the process of translating to ground programs is merely a "compilation process": the user writes constraints in some standard form, which are then parsed and translated to the corresponding ground program. The translations *trans*1 and *trans*2 can be viewed as these type of methodologies.

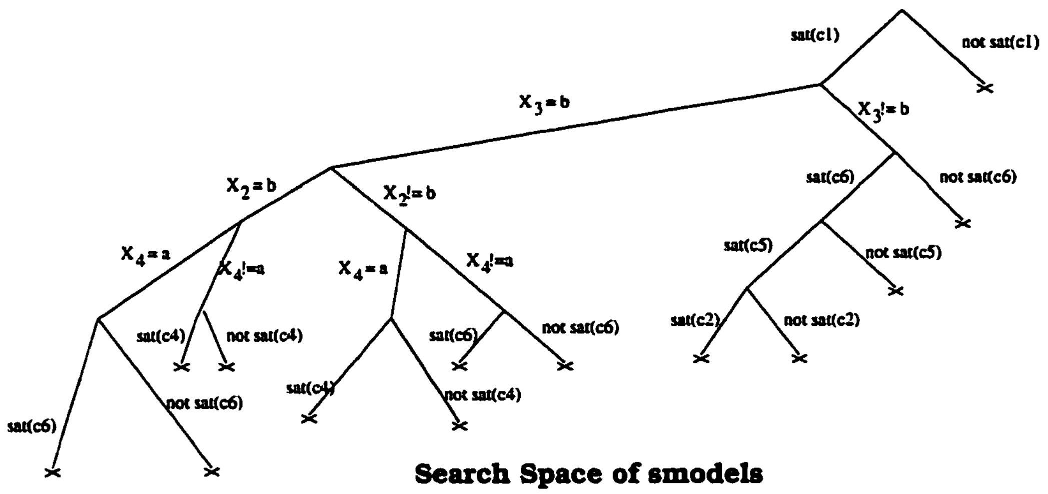
In the next section, we shall compare finite CSP methods based on backtracking and **smodels** based CSP solving method presented in the previous section. We shall be referring to the method of solving finite CSPs using *trans*2.

3.4.2 Comparison of search spaces of smodels and backtracking for binary CSPs

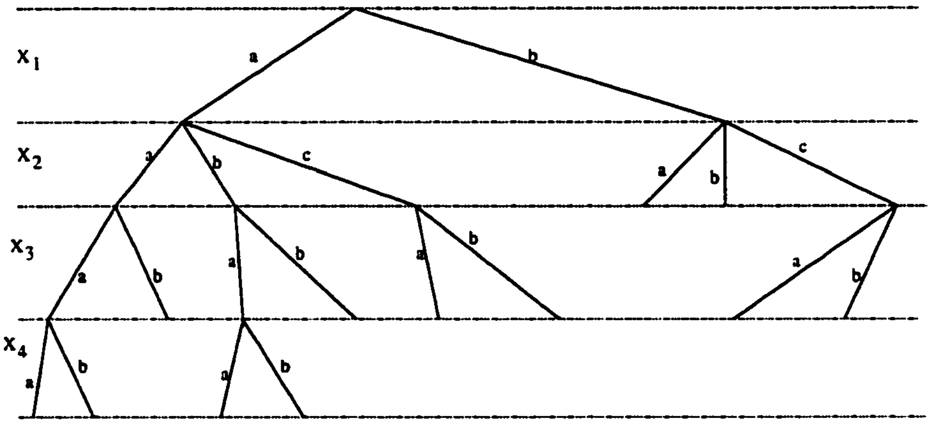
The **smodels** based solving method and BT differ in the way branching occurs. The ordering in **smodels** occurs dynamically whereas the order in fixed in BT. The difference between the two methods of solving the same CSP is illustrated by showing the running of BT and **smodels** on the CSP shown in Fig. 2.1. The search space of BT and **smodels** is shown in Fig. 3.14.

In the literature of finite CSPs common measures which are used to compare different methods are *execution time*, *number of consistency checks*, and *size of search space*. Our intent here is to provide a comparative analysis of the efficiency of solving finite CSPs using chronological backtracking (BT) and **smodels**. Thus we need to compare the relative efficiency of two different knowledge representation schemes. The concept of *consistency checks* does not have an identical corresponding measure in **smodels** based solving.

In [54], **smodels** was compared with other implementations of SAT problems on basis of *execution time*. The *execution time* is a commonly used parameter for measuring the relative efficiency of different implementations. The *execution times* of both methods when measured on same machines, gives a reasonable idea of the



Search Space of smodels



Search Space of Chronological Backtracking

Figure 3.14: Search spaces of smodels and BT for the example in Fig. 2.1

relative efficiency of the systems. However, in many cases some heterogeneous factors like the data structures used, the implementation langauge etc. make *execution time* a biased choice. So another relatively less biased metric of measurement presented along with *execution time* gives a balanced picture. Here both the methods here are essentially search algorithms.

In a typical search algorithm in artificial intelligence, a search node is defined by a choice point where one needs to choose between a fixed number of alternatives. This choice factor is responsible for the exponential nature of search space. Further if the computation involved at any node is polynomial in complexity, it fades out in contrast to the exponential number of choices in terms of contribution to the overall cost of solving the problem. Hence *size of search space* is a reasonable choice for comparing two heterogeneous methods provided the cost of computation at each node is polynomial in both cases. Thus we use the size of search space as the measure for comparing **smodels** and BT. Hence *size of search space* is used in combination with *execution times* as the method to compare two methods in our case.

We base our analysis here on finite binary CSPs. Further, we assume that number of variables is n and that each of the n variables $\{v_1, v_2, \ldots, v_n\}$ can assume any of the d values from its domain. In addition, let us assume the number of binary constraints to be m each with l tuples. Trivially it can be seen that m is of the order of magnitude $O(n^2)$ in the worst case. Further, we explain the concept of a node in the search space in both cases. In case of BT, a node refers to a point in the search space when a partial assignment is extended by assigning a value to a new variable. On the other hand, in **smodels** a node shall be referred to a point in search space when a new literal of the form l or *not* l is picked and **smodels** is called with either $B \cup \{l\}$ or $B \cup \{not\ l\}$. We further assume that B initially contains all atoms of the form $sat(c)$ where c is a constraint. This assumption is valid because any stable model of a program generated by *trans*2 contains all these atoms necessarily. Based on the assumptions, we now proceed to measure the relative efficiency of computation at each search node in the solving process.

The **smodels** system uses a linear-time method [60] at some nodes in the search space (when B covers $Neg(P)$) to check whether the model is a stable model of the program or not. Further all heuristics and deductive closures (Dcl) at any node are computable in time linear in the size of the program [54], and **expand** computes a deductive closure a linear number of times . Further there is only one call to **expand** at each node. The size of the program generated in the translation algorithm $trans2$ corresponding to a finite CSP is also polynomial in the total number of domain constants in the CSP. Hence overall the computation at a node is bounded by a polynomial in the size of a CSP.

Coming to the solving of finite CSPs by BT, we see that at any node in the search space we need to go through all the tuples in all the constraints in the problem in the worst case. This is polynomial in the size of the CSP.

So at any particular node in the search space of either method, the processing is polynomial in the total number of domain constants in the CSP. Given this background, we can now go ahead and compare the number of nodes in the search space of both methods of representing and solving finite CSPs.

Worst-case complexity bounds

Based on the concept of solution node as explained in the previous section, we show that the worst-case bound of the number of nodes in case of solving by **smodels** is double that of the worst-case bound of number of nodes in solving by backtracking based method. As explained already, the underlying assumption in case of a counting a node is the assumption of low order polynomial pre-processing at each node, and counting only the non-deterministic choice points as nodes in the solution space. In case of BT, a node refers to an extension of a partial solution by the assignment of a new value to a new variable, and checking of the consistency of this extended assignment with previous assignments forms the polynomial processing part at each node. In **smodels** based solving, the non-deterministic calls to **smodels** with a new literal is counted as a node and the polynomial processing consists of the processing in **expand**. Based on these assumptions, the following results can be explained.

Theorem 3.2 *The number of nodes in the worst-case for backtracking is bounded by* $\frac{d^{n+1}-d}{d-1}$.

Proof. In the search space of backtracking, let level 1 correspond to v_1, level 2 correspond to v_2 and so on. Hence the number of nodes at the first level is the same as the number of choices for v_1, namely d. Next, at level 2 there are d values of v_2 corresponding to each choice of v_1. Hence there are d^2 values at second level. Following this thread, at the leaf level there are d^n values. The total number of nodes in the search space is hence $\sum_{i=1}^{n} d^i$. The precise value of this is $\frac{d^{n+1}-d}{d-1}$. □

Theorem 3.3 *The number of nodes in the worst-case for* **smodels** *based solving is* $2*\frac{d^{n+1}-d}{d-1}$.

Proof. At any node, **smodels** chooses either l or *not* l where $l \in Neg(P)$. If P is the program generated by *trans*2, $Neg(P)$ contains all atoms of form $v_i(val)$ as well as atoms of the form $sat(c)$. Since **smodels** is assumed to be invoked with an initial B containing all literals of the form $sat(c)$, **smodels** only chooses literals of the form $v_i(val)$, where val is a value in the domain of v_i. Let the first level correspond to the choice of values for the variable v_1 whose domain contains d values $val_1, val_2, \ldots, val_d$. The program generated by *trans*2 ensures that when an atom of the form $v_1(val)$ is selected at a node by **smodels**, *not* $v_1(val')$ is added to B by **expand** (by inference rule IR3) for all values val' in the domain of v_1 such that $val' \neq val_1$. Hence any further choice of a literal of the form $v_1(val_2)$ or *not* $v_1(val_2)$ occurs only for the node of *not* $v_1(val_1)$. On similar lines, further choice of a literal of the form $v_1(val_3)$ or *not* $v_i(val_3)$ occurs only for the node of *not* $v_1(val_2)$. Hence corresponding to v_1, a total of $2d$ nodes are traversed corresponding to the d values of v_1.

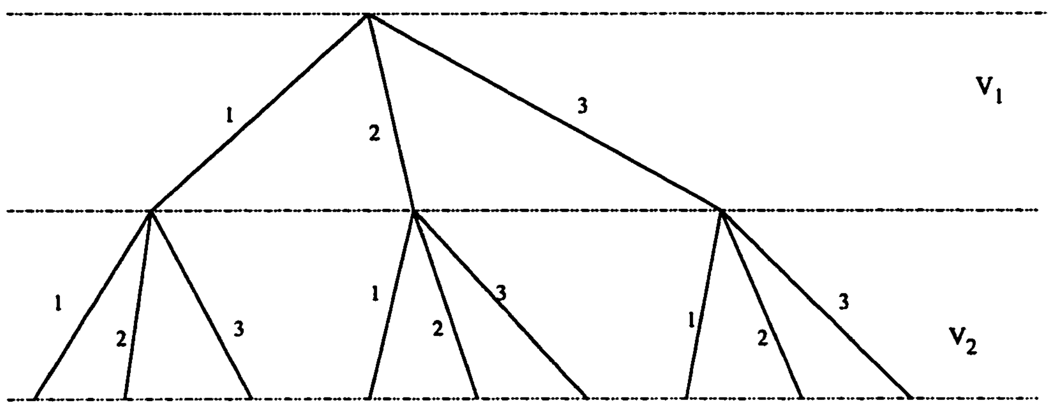


Figure 3.15: Search space of finite CSP in backtracking

At the second level however, there are value assignments corresponding to literals of the type $v_2(val)$ or *not* $v_2(val)$ only for nodes in the first level corresponding to positive atoms of the type $v_1(val_i)$. Thus the $2d$ possible values at second level are attached to only d nodes of the first level. Hence there are $2d^2$ nodes at the second level. Generalizing this we get the total number of nodes as $\sum_{i=1}^{n} 2 * d^i$, which is exactly $2 * \frac{d^{n+1}-d}{d-1}$. □

The discussions and proofs above will become clear if we see the example shown in Figures 3.15 and 3.16. We draw the search space for both for the case when $n = 2$ and $d = 3$. There are two variables v_1 and v_2 each with a domain size of 3. In Figure 3.15, we can see that the search space is made of a total of 12 nodes while in Figure 3.16 the search space is made of 24 nodes.

Experimental comparison of smodels and BT

The worst-case complexity analysis usually reflects only the extreme behavior of an algorithm, which might occur quite rarely in practice. To get an idea of the average case performance of BT and **smodels**, we performed experiments on randomly generated CSPs. We use the basic **smodels** procedure without either **lookahead** or **backjumping** as shown in Figure 2.3. Further we use the programs generated by *trans*2 for the comparison. We additionally enforce the condition that initially B

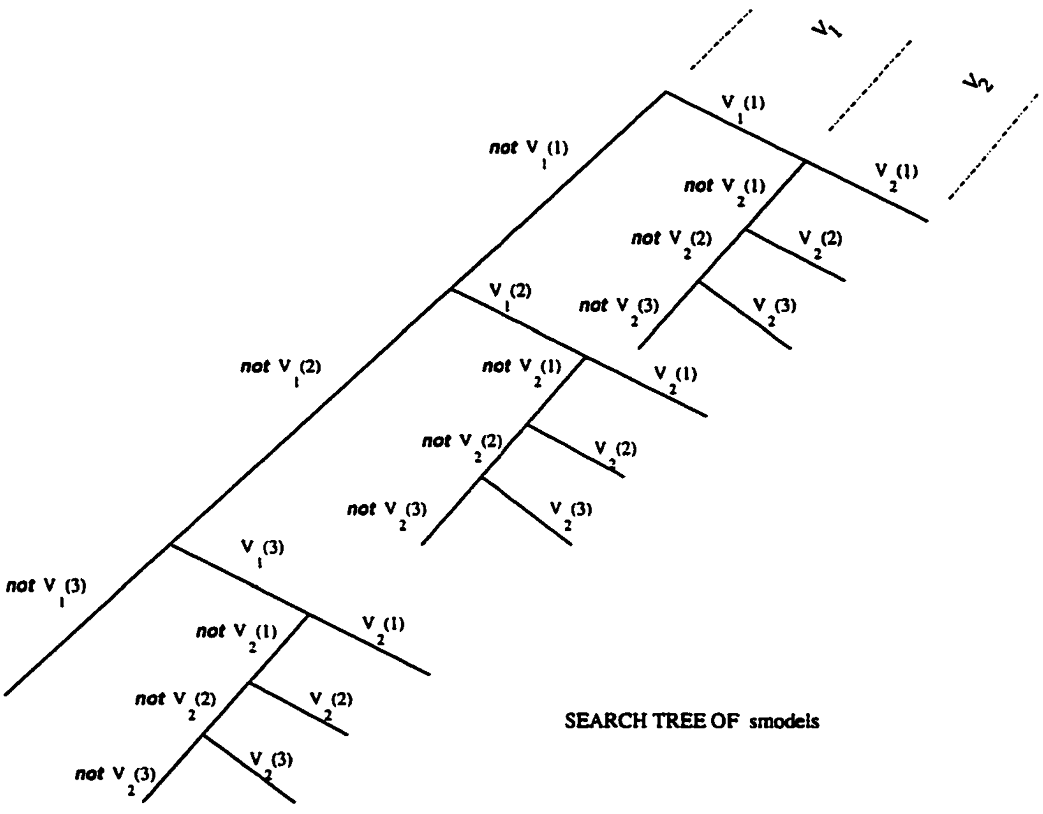


Figure 3.16: Search space of **smodels** based CSP solving

all literals of form $sat(c)$, where c is a constraint. Since the basic **smodels** procedure uses **expand**, which in turn performs constraint propagation, we also compared **smodels** and BT. A detailed study of the relationship between **expand** and FC is presented later. In all the cases the measurement was taken either to generate one solution if it existed or to report if none existed.

In the experimental set-up, binary constraints were considered. The number of variables was fixed at 20 and each variable was considered to have a fixed domain size of 5. Further all constraints were considered to contain 16 randomly chosen tuples (of the possible 25). Constraint density w, which is defined as the number of constraints in the CSP expressed as a fraction of the total possible number of constraints, is varied. An ensemble of 100 random finite CSPs was generated for each value of w. The value of w (in %) varied from 1 to 100. The average number of nodes for each value of w was taken by taking the *median* of the number of nodes for the 100 CSPs. The *median* is chosen because it gives a better measure of central tendency then mean because it is not affected by outliers in the data which is a source of major discrepancies in the data for arithmetic data. The graph in Figure 3.17 shows the relative comparison of the number of nodes in **smodels** and BT. The graph in Figure 3.18 shows the relative comparison of the execution times of **smodels** and BT.

The graph highlights some interesting results. The main points elucidated by the graph are:

1. Though the worst-case complexity of **smodels** is higher than that of BT, the average case performance of **smodels** is better than BT.

2. The phenomenon of ***phase transition*** [55, 62] in search methods refers to the occurrence of abrupt peaks in the search cost. It has been shown that these peaks in finite CSP methods occur in the region of transition from the region of soluble CSPs to the region of inconsistent CSPs. It has been also shown that problems in the phase transition region are the "hardest" problems to

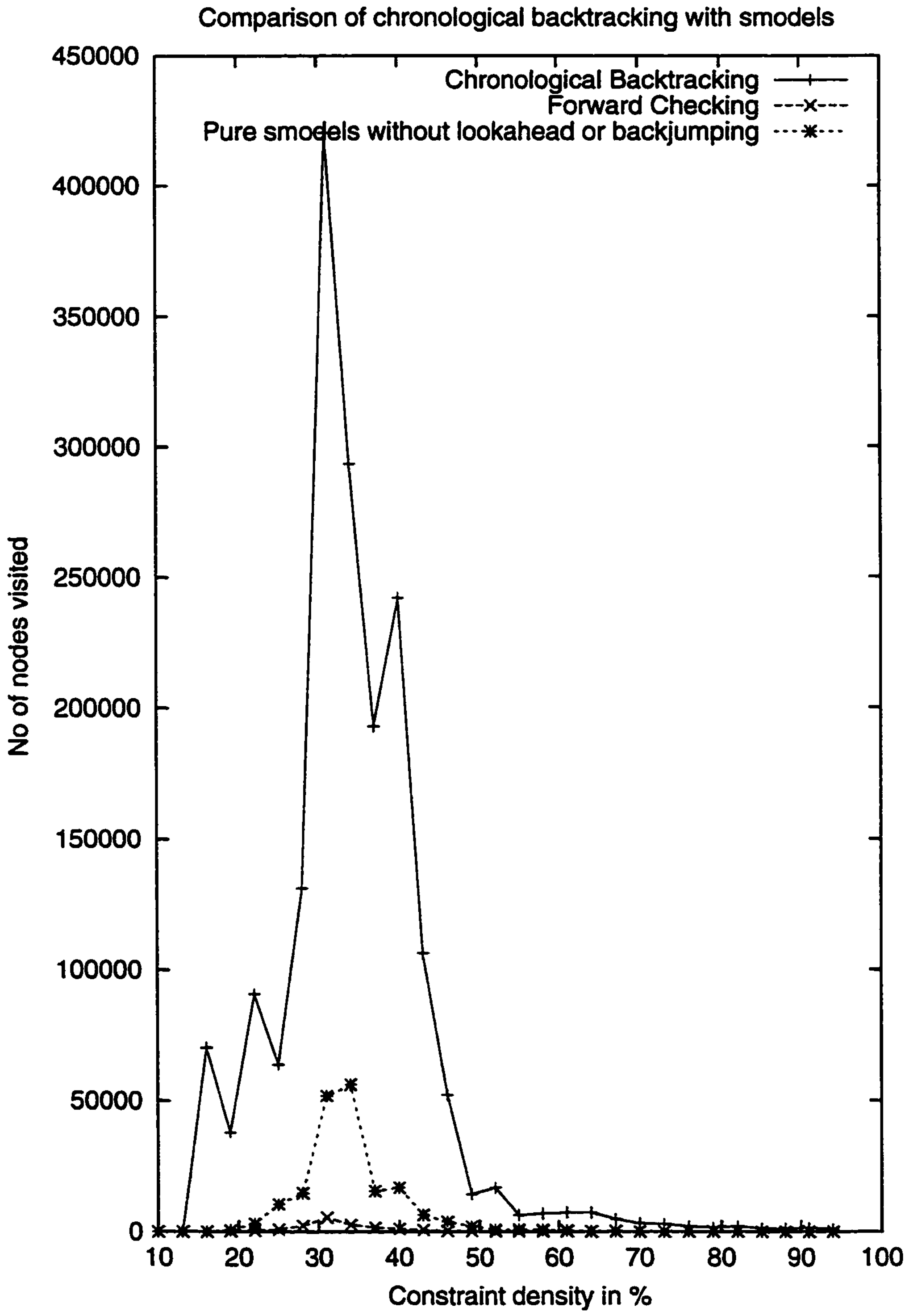


Figure 3.17: Comparison of median number of nodes of smodels and BT

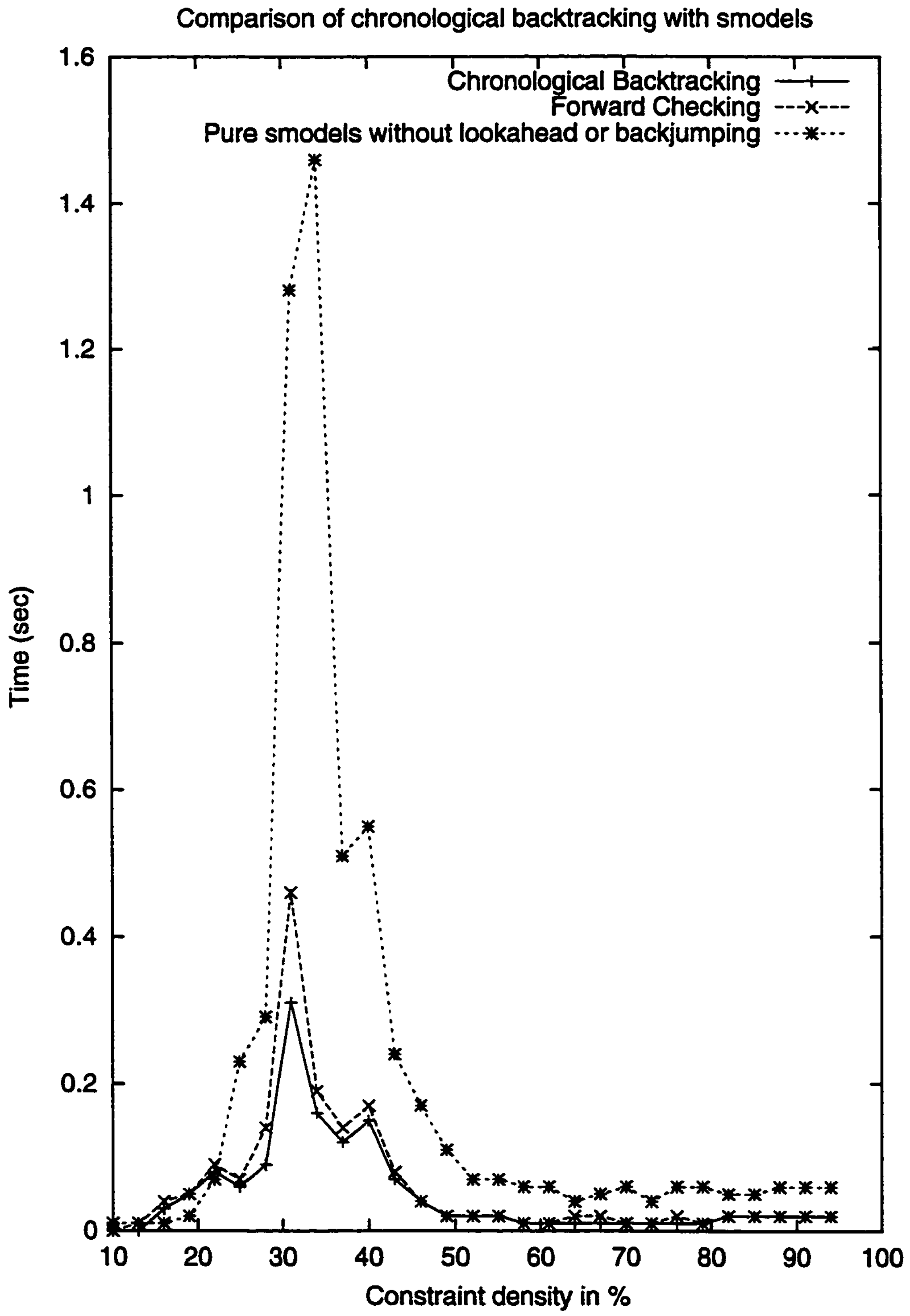


Figure 3.18: Comparison of median execution time of smodels and BT

solve. All backtracking based finite CSP solving methods have been shown to exhibit phase transition behavior. Here we observe that similar phase transition phenomenon is exhibited by **smodels** too.

3.4.3 Fine-tuning smodels for CSP generated logic programs

The worst-case analysis of **smodels** for solving CSPs has shown that the bound in terms of number of nodes traversed in the extreme case is double the worst-case bound of backtracking. A question arises if **smodels** can be modified taking into account the specific structure of the logic program generated by *trans*2 for CSPs. In this section, we present **csp-smodels**, a restricted version of the **smodels** procedure which works for logic programs generated by *trans*2 and has the same worst-case bound as BT. This exploits the specific structure of the logic program generated by *trans*2 on a CSP. Assume a program P' obtained by applying the algorithm *trans*2 to a finite CSP P.

The difference between *trans*2 and **csp-smodels** lies in the restriction that the procedure **csp-smodels** is called only with positive atoms of the type $v_i(val)$, and not with any negative literal of the form *not* $v_i(val)$. Further it is assumed that **csp-smodels** is invoked initially with

$$B = \{sat(c) \mid c \textit{ is a constraint in } P\}.$$

Moreover, **csp-expand**, a simplified version of **expand**, is used and has been shown to achieve the same result as **expand** for the class of logic programs generated by *trans*2. The procedures **csp-expand** and **csp-smodels** are presented in Figure 3.19.

Before we compare the worst-case bound of search spaces, we prove that the **csp-smodels** procedure is sound and complete. We need to show the equivalence of **expand** and **csp-expand** for P'. Recall that **expand**(P, B) is defined by the procedure in Figure 2.4. Recall further that the closure $Cl(P, B)$ is defined as a minimal set closed under inference rules 1 to 4 below, that contains B.

Inference rule 1(IR1) If the body $l_1, \ldots, l_n$ of the rule $h \leftarrow l_1, \ldots, l_n$ is a subset of B, then the head h belongs to every stable model agreeing with B.

Inference rule 2 (IR2) If P_B contains no rule with head h, then h is not an element of any stable model agreeing with B.

Inference rule 3 (IR3) If $h \in B$ is the head of only one rule $h \leftarrow l_1, \ldots, l_n$ in P_B, then $l_1, \ldots, l_n \in \Delta \cup not(\overline{\Delta})$ for every stable model Δ agreeing with B.

Inference rule 4 (IR4) If $not\ h \in B$ and h is the head of the rule $h \leftarrow l_1, \ldots, l_n$ in P_B for which $l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n \in B$, then every stable model that agrees with B agrees with $B \cup \{opp(l_i)\}$.

Theorem 3.4 **csp-expand**(P', B) = **expand**(P', B).

Proof. Effectively we need to show that the second clause in the definition of **expand** is irrelevant in case of **csp-expand**. This is shown by looking at the structure of the program P' generated by $trans2$. In $trans2$ corresponding to any assignment atom $v_i(val_i)$ (which stands for the assignment $v_i = val_i$ where val_i is in the domain of v_i), there is exactly 1 clause with $v_i(val_i)$ as the head. All other clauses for the variable v_i have $not\ v_i(val_i)$ in the body. Now considering $Neg(P')$, we find that it contains all the atoms of the form $v_l(val_j)$ for any such possible combination. Further it contains all atoms of the $sat(c_i)$ where c_i is a constraint. Now let us turn our attention to a specific B in an intermediate stage of the solving process. B contains a set of positive atoms of the form $v_i(val_i)$ for some variables v_i, and B contains all negative literals of the $notv_i(val_k)$ where $val_k \neq val_i$ and val_k is in the domain of v_i. Let us turn our attention to Dcl$(P'_B, not(Neg(P')))$. Since $Neg(P')$ contains all atoms in the program as explained above, all negated literals in P'_B can be removed indicating that now all the clauses are involved in the program P'_B are retained except the negated

```
function csp-smodels($P, B$)
$B'$ := csp-expand($P, B$)
if conflict($P, B'$) returns true then
    return false
else if for all $v_i \in \{v_1, \ldots, v_n\}$ there is an atom of the form $v_i(val_i)$ in $B'$ then
    return true
else
    Pick a variable $v_i$ such that there is no atom of the form $v_i(val_i)$ in $B'$
    Let $S_i = \{v_i(val_i) \mid v_i(val_i)$ is not covered by $B'\}$
    For all $x \in S_i$
    do
    if csp-smodels($P$, $B' \cup \{x\}$) returns true then
        return true
    endif
    $B' := B' \cup \{not\ x\}$
    enddo
    return false
endif
```

```
function csp-expand($P, B$)
    $B := Cl(P, B)$
return $B$
```

Figure 3.19: The **csp-smodels** and **csp-expand** procedures

atoms. This program contains all the facts of the form $sat(c_i) \leftarrow$, for all c_i. In addition for any variable v_j which does not have a positive atom of the form $v_j(val_j)$ in B, all clauses corresponding to variable assignments of v_j are converted to a set of facts $v_j(val_{j1}) \leftarrow; v_j(val_{j2}) \leftarrow; \ldots$ for all domain values of v_j. But corresponding to a variable v_i which has an assignment $v_i(val_i)$ in B, there is only one fact $v_i(val_i) \leftarrow$ in the program and there is no rule has as its head an atom of the form $v_i(val_k)$ where $k \neq i$. Hence for all v_i such that there is an assignment $v_i(val_i)$ is present in B, none of the atoms of the form $v_i(val_k)$ where $k \neq i$ are derivable from $\mathrm{Dcl}(P'_B, not(Neg(P')))$. Hence for all atoms x of this form $v_i(val_k)$ where $k \neq i$ and $v_i(val_i)$ is in B, *not* x can be added to **expand**(P, B). But all these literals of the form *not* x are added trivially by application of the inference rule IR3 to the program P'. Hence **csp-expand**(P, B) = **expand**(P, B). □

We now state and prove the main theorem.

Theorem 3.5 *The algorithm* **csp-smodels** *is sound and complete for solving a finite CSP.*

Proof. In **csp-smodels**, the algorithm uses **csp-expand** for expanding a set B consisting of a set of literals. We have seen that **csp-expand**$(P, B) = Cl(P, B)$. Hence **csp-expand** is correct.

The soundness is trivial because **csp-smodels** is a special restriction of the general **smodels** procedure which has been shown to be sound for the translated logic program obtained by *trans*2.

To prove completeness, we need to show that no solution of CSP is lost by restricting **smodels** to **csp-smodels**. Suppose there were a solution S to the original CSP which does not have any corresponding B which is output by **csp-smodels** when applied to the logic program P output by *trans*2. Clearly since S is a solution of the CSP, it is an assignment to all the variables which satisfies all the constraints in the CSP. Let $\{v_1 = val_1, v_2 = val_2, \ldots, v_n = val_n\}$ be the assignment in S. The set of atoms S' representing $\{v_1(val_1), v_2(val_2), \ldots, v_n(val_n)\}$ corresponds to a set of assignments which is a solution of the CSP. Hence each assignment is mutually consistent with the other assignment. Thus there is no possibility of a pair $v_k(val_k)$, *not* $v_k(val_k)$ to be ever generated. Consider a set B which contains all atoms in S' and in addition all atoms of the form $sat(c_i)$ where c_i is a constraint. Further let B contain all literals of the form *not* $v_k(val_j)$ where $val_j \neq val_k$. Further B does not have a pair $v_k(val_k)$, *not* $v_k(val_k)$ in it by virtue of the mutual consistency of the value assignments. Thus B will not exit from the **csp-smodels** procedure due to a **conflict**. Hence there is a node in the search space of **csp-smodels** which generates this B. Since such a B satisfies the exit condition of **csp-smodels**, this particular B is output by **csp-smodels**. This contradicts the assumption that no such B corresponding to S is output by **csp-smodels**. Hence **csp-smodels** is complete. □

Having proved the soundness and completeness of **csp-smodels**, we now go ahead and state a result concerning the search space of the algorithm.

Theorem 3.6 *The worst-case search space traversed by* **csp-smodels** *based CSP solver has the same number of nodes as the worst case of the backtracking algorithm.*

Proof. In the worst case of the **csp-smodels** procedure, all possibilities of the positive combinations of atoms of the form $v_j(val_k)$ should be tried. Let us assume d as the uniform domain size of each of the n variables in the problem. In the worst case, let v_1 be the first variable selected. The procedure **csp-smodels** can select any of the atoms corresponding to the d possible values. Suppose $v_1(val_1)$ is selected. By application of **csp-expand**, all negated literals of the form *not* $v_1(val_k)$ where $val_k \neq val_1$ are added to B before the next call to **csp-smodels**. Thus there are d possible nodes at the first level of the search space. In the worst case, the propagation of **csp-expand** might not be able to add to B any atom corresponding to any other variable v_j $(j \neq 1)$. In that case, for the second level for each of the d values of v_1 a set of d values are possible for the next variable, say v_2. Clearly at this layer, the number of nodes is d^2.

Going by this logic, we get the total number of nodes in the search space as $\sum_{i=1}^{n} d^i$, which is the same as the worst-case search space of backtracking in CSP. Hence the proof.□

It is established that **csp-smodels** is a restricted version of the **smodels** procedure which is fine-tuned for the restricted logic programs generated by a CSP, which has the same worst-case search space as the backtracking algorithms for CSP. we can identify the exact relationships between the techniques employed in **smodels** and the corresponding techniques in CSP.

3.5 Conclusions

The modeling power of SLP and languages based on SLP have been evaluated in context of constraint programming. In particular LP_{SM} has been compared with existing constraint programming languages in context of modeling power. We find that LP_{SM} shares the declarative nature as these other languages and also find that it has some definite advantages in constraint programming situations involving dynamic constraints as compared to other modeling languages.

We have explored the different issues in the modeling of finite CSPs in LP_{SM}, a subset of the class of function-free normal logic programs. Later the problem of encoding CSPs directly in **smodels** was studied. We find experimentally that applying **smodels** directly to the ground program representing a finite CSP is more efficient than the method proposed to model CSPs in LP_{SM} by Niemela in [41]. We perform theoretical and experimental comparison of the techniques of solving CSPs by **smodels** and chronological backtracking. Even though the worst-case search space bounds are worse for **smodels** based finite CSP solving method, it is found experimentally that the average case of **smodels** based method scores marginally. We then suggested improvements in the basic **smodels** procedure to improve the worst-case bounds for the class of logic programs generated by CSPs.

Overall in this chapter we have provided insights into the modeling of finite CSPs and other constraints by LP_{SM} and **smodels**.

Chapter 4

Mapping smodels techniques to CSP techniques

The paradigm of constraint programming has proved itself to be a practical one in artificial intelligence with widespread industrial scale applications. The applications of constraint programming have been in diverse areas like scheduling, configuration, user interfaces, design, diagnostic reasoning, and resource allocation. The main contributor to the success of constraint programming has been the development of efficient techniques for solving constraints. Massive research has gone into the devising of special CSP techniques which endow the constraint programming implementations the high degree of efficiency.

It is natural to expect that the inclusion of the efficient techniques from constraint programming into any problem-solving system is bound to yield an efficient implementation. In Chapter 2, we showed how the techniques in **smodels** mapped to some well known techniques in SAT solvers. It is however also well known that SAT and CSPs have a close relationship because SAT problem itself is a variant of the CSP problem specification. This chapter attempts to extend the abovementioned relationship between **smodels** techniques and SAT techniques by establishing a direct relation between the important techniques in **smodels** and some well-known efficient CSP techniques.

There are mainly three techniques employed in **smodels**, namely, propagation,

lookahead and **backjumping**. We show that they are mappings from the three corresponding techniques of ***constraint propagation***, dynamic variable ordering (DVO), and backjumping in finite CSPs. The precise relationship between the three corresponding pairs of techniques is explored in four steps : (i) by illustrating the effect of the particular technique in the solving process by an example in both CSP and **smodels**, (ii) by identifying sufficient conditions under which the technique in CSP exactly matches the corresponding technique in **smodels** and vice-versa, (iii) by pointing out possible improvements in the technique of **smodels** based on ideas from implementation of the same idea in the CSP paradigm, and (iv) by performing experiments on random constraint satisfaction problems and comparing the relative efficiency of the techniques in terms of search space. The implementations of various CSP techniques like forward checking, backjumping, backtracking etc were all taken from the public domain CSP library of C routines developed by van Beek et al. available at [58].

However some important points are to be taken into consideration while looking into the mapping process. The points are:

- The **smodels** proof procedure is programmed to perform constraint propagation in the default case. Only the techniques of **backjumping** and **lookahead** have the option of being excluded. In that case, the comparison of **lookahead** or **backjumping** with corresponding technique in CSP needs to ensure that constraint propagation is employed in the CSP alongside DVO or backjumping respectively.

- Another important difference is in the instantiation order of the variables which is fixed in many CSP algorithms. But in **smodels** this order is not fixed beforehand. But this does not prevent us from identifying the relationship between the corresponding pairs of techniques except for the **backjumping** case.

- In experimental comparisons of the corresponding pairs of techniques it is assumed that for **smodels**, *trans*2 (the direct method of solving in **smodels**) is used with the additional assumption that all atoms of the form *sat*(c) (where c is a constraint in the CSP) are in the initial B.

In the end we also perform a relative comparison of the three techniques in **smodels** experimentally in order to determine the singular contribution from each towards the efficiency of **smodels** . It turns out that **lookahead** dominates totally over the other two techniques by orders of magnitude. Moreover, **smodels** employing **lookahead** turns out to competitive to the best techniques in finite CSPs. Because the technique of **smodels** with **lookahead** turned out to be competitive to the best CSP techniques we performed a comparison of **smodels** including **lookahead** with the techniques of forward checking with DVO (FC-DVO), forward checking with full arc-consistency (FC-arc), forward checking with backjumping (FC-BJ), forward checking with both arc-consistency and DVO (FC-arc-DVO), and forward-checking with conflict-direct backjumping (FC-CBJ) on the hardest constraint satisfaction problems identified by *phase transitions*, briefly described in the previous chapter. The results show that **smodels** with lookahead competes with the best among these techniques.

4.1 Constraint propagation in CSPs vs. smodels

4.1.1 Constraint propagation in CSPs

In finite CSPs, the primary process of constraint propagation involves ensuring that the final form of the CSP, becomes k-consistent, for a predetermined value of k. Many variants of constraint propagation algorithms have been proposed in the CSP literature. These algorithms differ in the relative amounts of propagation and backtracking involved in the process of solving a CSP. The value of k which represents the level of consistency ($k-$consistency) of a CSP, determines the further amount of backtracking required. In one extreme, strong n-consistency for the CSP would completely eliminate the need for search. But since the consistency algorithms (propagation) algorithms are costly for higher values of k, it is desired to apply lower orders of consistency (lower value of k) before proceeding for backtracking. Commonly CSP propagation methods limit themselves to a value of $k = 2$, meaning they strive to achieve arc consistency before applying backtracking. In some rare cases, pure propagation can achieve a backtrack-free solution process for a finite CSP. In [14] Freuder presented a sufficient condition for a finite CSP to return a solution without any backtracking.

Various constraint propagation algorithms based on arc-consistency have been proposed in constraint programming literature. Any basic algorithm employing arc-consistency involves alternate assignment and propagation steps, where an assignment step incrementally expands a partial assignment like chronological backtracking (BT) and a propagation step propagates the partial assignment to make the resulting CSP arc-consistent to some degree. In forward checking (FC) [25], the propagation step is limited to enforcement of arc-consistency is limited to that between the current (latest instantiated) variable and all future variables. On the other hand, MAC (maintaining arc-consistency) [25, 17] enforces arc-consistency not only between the present variable and the future variables but also between all future variables.

In the example studied in Section 3.4, we shall show how application of MAC, which achieves a solution for this problem without any backtracking, by just an initial application of one arc-consistency step. The CSP P here is defined on four variables v_1, v_2, v_3 and v_4 with domains of $d_1 = \{1\}$, $d_2 = \{1,2,3\}$, $d_3 = \{1,2\}$, and $d_4 = \{1,2\}$ respectively. P contains five constraints $C_1(v_1,v_2) = \{(1,2),(1,1)\}$, $C_2(v_2,v_3) = \{(2,1),(3,2)\}$, $C_3(v_3,v_4) = \{(1,1),(2,2)\}$, $C_4(v_2,v_4) = \{(2,1)\}$ and $C_5(v_1,v_4) = \{(1,1)\}$. Consider starting with the arc between v_1 and v_2. Because there is only one value for v_1, the domain of v_1 remains intact at $\{1\}$. But there are only two values in domain of v_2 which have a corresponding value in v_1, namely 2 and 1. So 3 is removed from the domain of v_2. Next the arc between v_1 and v_4 is studied. Here there is no further change to domain of v_1. But there is only one value of v_4 which has a corresponding value in v_1, namely 1. So the value 2 is removed from the domain of v_4. Next the arc between v_2 and v_3 is explored. Since the domain of v_2 now contains only the values 2 and 1, we need to consider only these two values. But since the value 1 of v_2 has no corresponding value in v_3, we remove 1 from the domain of v_2. Further, since now we do not have any value of v_2 after this corresponding to the value 2 of v_3, this value is removed from domain of v_3. At this stage the domains of v_1, v_2, v_3 and v_4 are $\{1\}, \{2\}, \{1\}$ and $\{1\}$ respectively. No further change in P is possible. At this stage we observe that if any solution exists, it can be obtained directly without any backtracking and by simply checking if the only remaining tuple is a solution or not. Here it is a solution of the CSP. Hence the resulting solution is $\{v_1 = 1, v_2 = 2, v_3 = 1, v_4 = 1\}$.

Consider applying FC to the same example. Start with $\{v_1 = 1\}$. Propagating this value to v_2, v_3, v_4 we reduce the domains of v_2, v_3, and v_4 to $d_2 = \{1, 2\}$, $d_3 = \{1, 2\}$, and $d_4 = \{1\}$ respectively. Expanding the assignment by $\{v_2 = 1\}$, further propagation annihilates the domain d_3 of v_3 causing backtrack to $\{v_2 = 2\}$. Propagation reduces domains of v_3, and v_4 to $d_3 = \{1\}$, and $d_4 = \{1\}$ respectively. Expanding the assignment by $\{v_3 = 1\}$ does not change d_4 in the propagation step. The final expansion $\{v_4 = 1\}$ gives a solution to the CSP as $\{v_1 = 1, v_2 = 2, v_3 = 1, v_4 = 1\}$.

The above is an extreme example of an application of constraint propagation, which involved no backtracking.

In the next section, we shall recall how pure propagation component is able to derive a solution for the same example in case of the finite CSP representation by **smodels** .

4.1.2 Constraint propagation in smodels

It has been shown earlier that in **smodels** the function **expand** is used to propagate the effects of a partial assignment of values in B to add as many new literals to B as possible without having to call **smodels** again. Recall that the function **expand** is defined by a set of inference rules applied to a current partial solution B till no changes in B occur. The inference rules of $Cl(P, B)$, the principal component of **expand**(P, B), are defined here. Let P_B, the reduct of P with respect to B, be the set of rules

$$\{h \leftarrow l_1, \ldots, l_n \in P \mid opp(l_i) \notin B\}$$

where $l_1, \ldots, l_n$ are literals and we know that for an atom x, $opp(not\ x) = x$ and $opp(x) = not\ x$. The inference rules IR1, IR2, IR3 and IR4 for $Cl(P, B)$ are defined below:

Inference rule 1(IR1) If the body $l_1, \ldots, l_n$ of the rule $h \leftarrow l_1, \ldots, l_n$ is a subset of B, then the head h belongs to every stable model agreeing with B.

Inference rule 2 (IR2) If P_B contains no rule with head h, then h is not an element of any stable model agreeing with B.

Inference rule 3 (IR3) If $h \in B$ is the head of only one rule $h \leftarrow l_1, \ldots, l_n$ in P_B, then $l_1, \ldots, l_n \in \Delta \cup not(\overline{\Delta})$ for every stable model Δ agreeing with B.

Inference rule 4 (IR4) If $not\ h \in B$ and h is the head of the rule $h \leftarrow l_1, \ldots, l_n$ in P_B for which $l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n \in B$, then every stable model that agrees with B agrees with $B \cup \{opp(l_i)\}$.

For the example CSP P studied in Section 4.1.1, application of one step of arc-consistency was sufficient to solve the CSP. Here we solve the same example by application of **smodels** to the logic program corresponding to P generated by *trans*2. We make the simplifying assumption that **smodels** is invoked with an initial $B = \{sat(c_1), sat(c_2), sat(c_3), sat(c_4), sat(c_5)\}$. This assumption is justified because any stable model will necessarily contain all atoms of form $sat(c)$ where c is a constraint. We show the execution of this example in Figure 4.1. Here one call of *expand* is sufficient to return a solution without any further calls to **smodels** . This is an

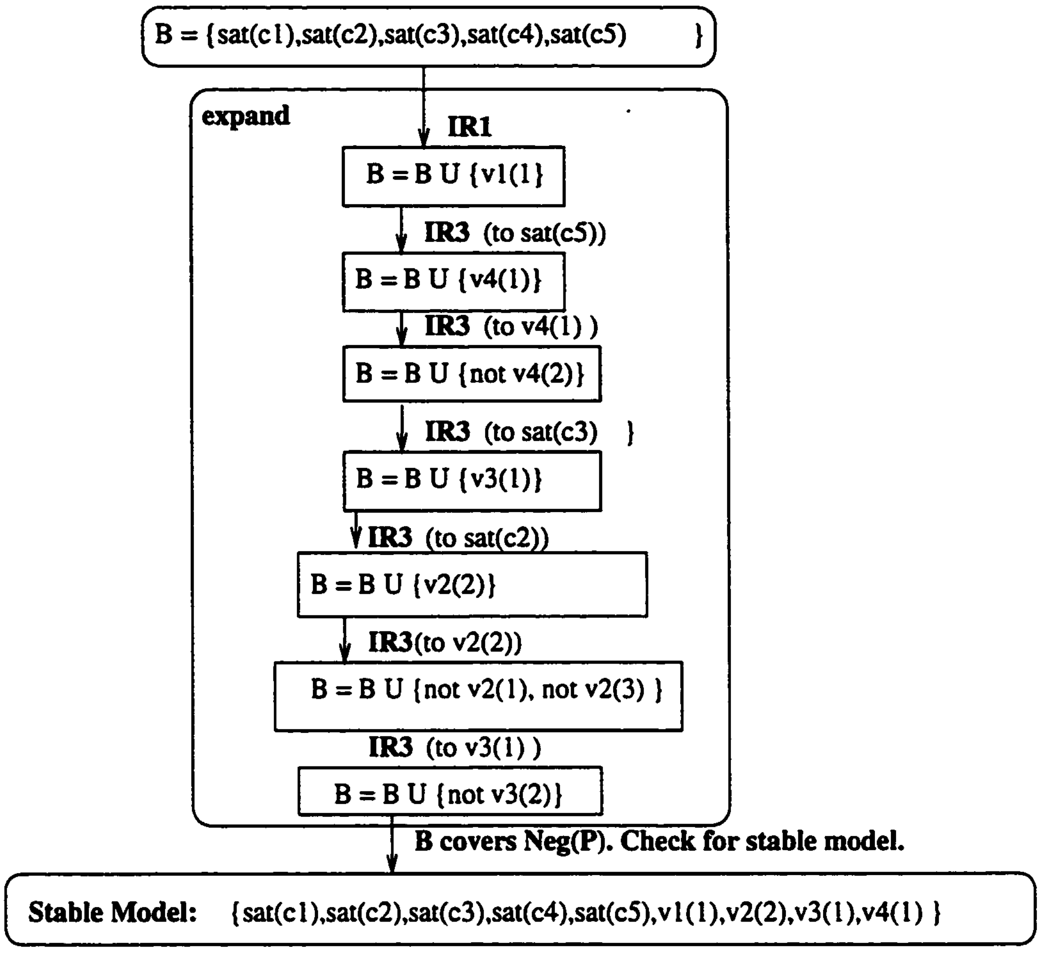


Figure 4.1: Execution of the CSP in Section 4.1.1 by **smodels**

extreme case but it suffices in our case to explain the pruning power of the propagation achieved by **expand** function. Thus this example is able to achieve a backtrack-free solving process by application of pure propagation in both **smodels** and CSP methods.

4.1.3 Relationship between expand and arc-consistency

A close look at the way the same example is solved by application of arc-consistency in CSP, and by **expand** in **smodels** suggests that the inference rules employed by **expand** are related to the inference rules enforcing arc-consistency. We shall try to see the relationship between these two sets of inference rules. Unless otherwise mentioned, we shall consider only binary CSPs in this section. Further, we assume the use of the translation algorithm *trans*2 to translate a CSP to the corresponding ground logic program. Additionally we assume that **smodels** is invoked with an initial B consisting of all atoms of the form $sat(c)$ where c is a constraint in the CSP.

It is known that **expand**(P, B) is derived by repeated applications of $Cl(P, B)$ which is in turn defined by the four inference rules IR1, IR2, IR3, and IR4. So, to compare the propagation in arc-consistency and **expand**, it suffices to study the relationship between inference rules used in $Cl(P, B)$ to the inference rules used in arc-consistency. The closure $Cl(P, B)$ is quite restrictive in its applicability to the specific form of logic programs generated by *trans*2. We present here two sufficient conditions under which $Cl(P, B)$ and arc-consistency coincide in effect.

Let B contain a set of positive atoms of the form $\{v_1(val_1), v_2(val_2), \ldots, v_j(val_j)\}$ corresponding to an assignment A of values $\{v_1 = val_1, v_2 = val_2, \ldots, v_j = val_j\}$ for a subset of the set of n variables $V = \{v_1, v_2, \ldots, v_n\}$. Clearly B also contains all negative literals of the form *not* $v_1(val_j)$ where $val_j \neq val_1$. Likewise for all variables which are assigned a value in A. Similarly B also contains all positive set of atoms of the form $sat(c_k)$ where c_k is a constraint. Assume we are trying to satisfy a constraint c_j such that c_j is defined over two variables v_l and v_t. Further assume that v_t is not yet assigned any value in A but v_l has been assigned the value val_l in A. Hence $v_l(val_l)$ is already in B. Now we attempt to satisfy $sat(c_j)$ in B (it is already in B by assumption) signifying an attempt to satisfy the constraint c_j. FC and other dynamic methods of achieving arc-consistency would make the arc between v_t and v_l consistent by removing all values from the domain of v_t which are not consistent with the value val_l of v_l. If there is more than one value val_t in the domain of v_t which is consistent with val_l of v_l, the inference rules of **expand** or arc-consistency cannot pick a specific value of v_t which satisfies c_j. No propagation to a value of v_t can be achieved by **expand** without a separate call to **smodels**. But if there is only such value val_t in the domain of v_t consistent with val_l of v_l, both arc-consistency and **expand** successfully assign val_t to v_t. With this setting we state the following sufficient condition:

Theorem 4.1 *If there is only value val_t in the domain of the variable v_t such that the pair $\{val_t, val_l\}$ is in the constraint c_j, then $Cl(P, B)$ makes the arc between v_t and v_l consistent.*

Proof. For any further expansion of B, v_t cannot get a value different from val_t. The proof is obtained by looking at the inference rules used in $Cl(P, B)$. When B is passed to **expand**, we first generate the program P_B which is the set of all clauses r such that none of the literals l_m in the body of r_j is such that opposite literal of l_m is in B (For a negative literal of the form *not* t the opposite is t and for an atom t the opposite is *not* t). By an earlier application of the inference rule IR1, because of the presence of $v_l(val_l)$ in B, B also contains all literals of the form *not* $v_l(val_{l'})$ where $val_{l'} \neq val_l$. Because of this, P_B has none of the clauses with $sat(c_j)$ at the head, which corresponds to a tuple in the constraint c_j having a value for v_l other than val_l. More precisely, there is no clause in P_B of the form $sat(c_j) \leftarrow v_l(val_{l'}), v_t(val_{t'})$ where $val_{l'}$ is in the domain of v_l, $val_{l'} \neq val_l$ and $val_{t'}$ is in the domain of v_t. Further, since constraint c_j has only one value val_t of v_t that occurs with val_l, P_B has only one clause with $sat(c_j)$ at the head. Applying inference rule IR3 we can then add $v_t(val_t)$ to B. Further, there is only one rule in P_B with $v_t(val_t)$ as head, namely the mutual exclusion rule corresponding to $v_t(val_t)$. By applying IR3 again we then add all literals of the form *not* $v_t(val_{t'})$ where $val_{t'}$ is in the domain of v_t and $val_{t'} \neq val_t$. Hence in any future expansion of this B any such $val_{t'}$ cannot be assigned to v_t. Hence the proof. □

Next, we state a much simpler condition which trivially achieves arc-consistency.

Theorem 4.2 *If a constraint c_j has only one tuple in it, $Cl(P, B)$ establishes arc-consistency between the variables forming c_j.*

Proof. Trivially proven by applying the inference rule IR3 of $Cl(P, B)$. □

We have shown that there are certain restricted conditions under which the propagation achieved by **expand** is the same as achieved by algorithms like FC which dynamically enforce arc-consistency.

4.1.4 Experimental comparison of FC and expand

In this section we explore the relative efficiency of applying pure constraint propagation in **smodels** and CSPs. We choose to execute **smodels** without any **lookahead** or **backjumping**. We compare it with FC, which dynamically enforces arc-consistency. We use the measure of *search space*, as described in a previous section. As explained earlier, the concept of a *node* in the context of *expand* refers to every non-deterministic choice of a new literal to be added to B when a new call to **smodels** is made. The processing of *expand* is polynomial, and is considered as the processing overhead at a node. In contrast, in case of FC, whenever a partial assignment is expanded with a assignment of a value to a variable. The forward checking component of the current assignment with future variables is polynomial and considered part of the processing at a node.

In the experimental set-up, binary constraints were considered. The number of variables was fixed at 20 and each variable was considered to have a fixed domain size of 5. Further all constraints were considered to contain 16 randomly chosen tuples (of the possible 25). An ensemble of 100 random finite CSPs was generated for each value of w (constraint density). The value of w (in %) varied from 1 to 100. The average number of nodes for each value of w was taken by taking the median of the number of nodes for each of the 100 CSPs. The graph in Figure 4.2 shows the relative comparison of the number of nodes in **smodels** and FC.

From the experimental results it is clear that the propagation achieved by **expand** is a quite limited in comparison to the propagation achieved by arc-consistency enforced in FC.

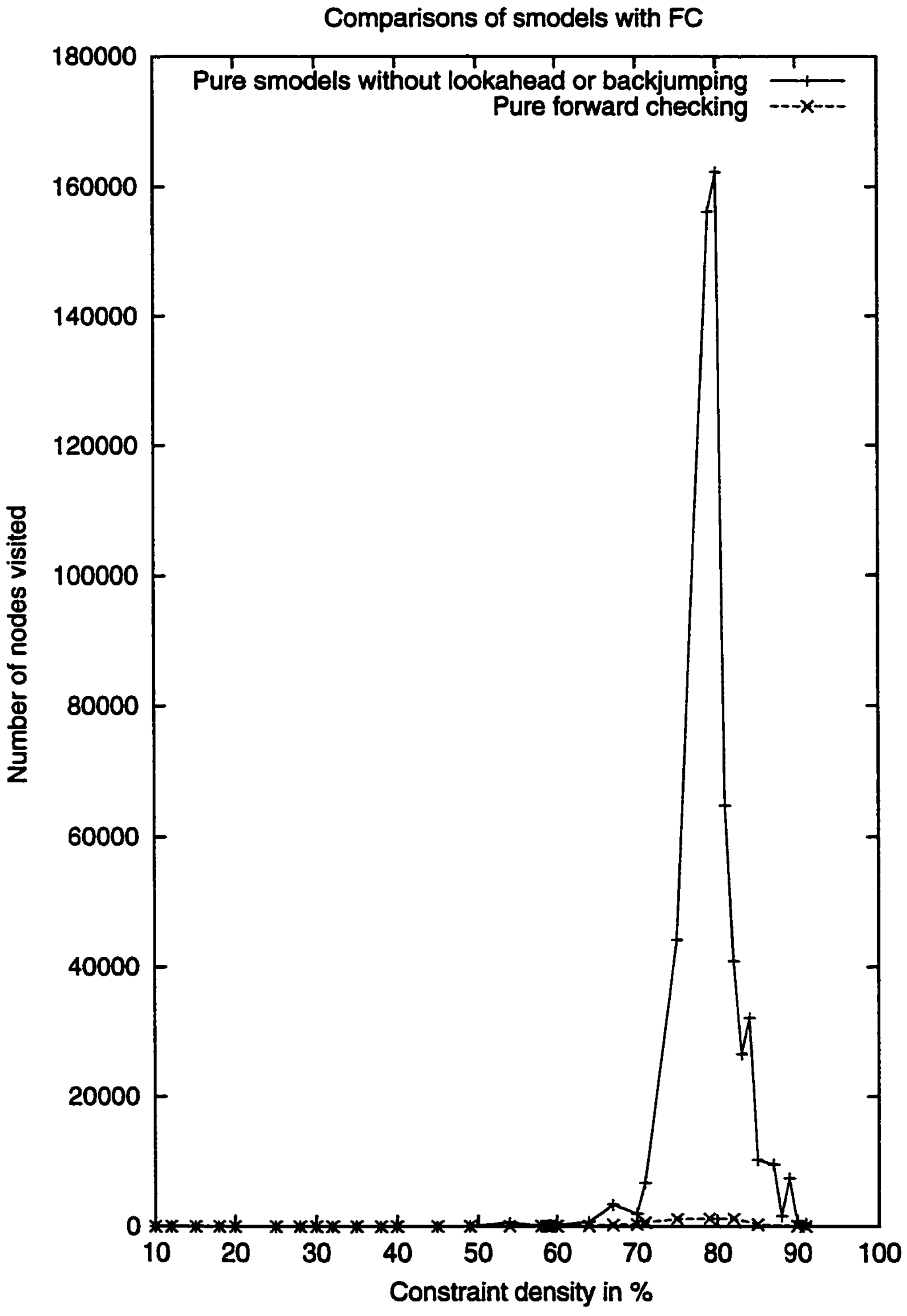


Figure 4.2: Comparison of average number of nodes of **smodels** and FC

4.1.5 Can expand be improved further?

A question which arises in such comparison of the two techniques is whether any of the techniques can be enriched by application of ideas from the other technique? Since **expand** of **smodels** has been shown to perform a less amount of propagation as compared to FC (an arc-consistency enforcing algorithm), we can incorporate inference rules of arc-consistency in **expand** to work for the class of logic programs generated by *trans*2 for finite CSPs. Consider a CSP P defined on three variables v_1, v_2, and v_3 with domains $d_1 = \{1, 2, 3, 4\}$, $d_2 = \{1, 2\}$, and $d_3 = \{1, 2\}$ respectively. Let P contain the constraints $c_1(v_1, v_2) = \{(1, 2), (2, 1)\}$, and $c_2(v_1, v_3) = \{(3, 1), (4, 2)\}$. On applying **expand** to the initial set $B = \{sat(c_1), sat(c_2)\}$, we do not add any further literal to B because none of the inference rules apply. But if arc-consistency is applied to the problem we can immediately eliminate all values from the domain of v_1, by removing $v_1 = 3$ and $v_1 = 4$ while considering constraint c_1 and removing $v_1 = 1, v_1 = 2$ while considering constraint c_2. The semantics of arc-consistency can thus be used to enrich the definition of $Cl(P, B)$. We can add the following inference rule (let us term it as IR5) to the existing inference rules of $Cl(P, B)$.

Inference Rule 5 (IR5) Let c_j is a constraint defined over the two variables v_t and v_l such that at least one of them does not have an assigned value in B. For any value val_t in the domain of v_t such that there is no value val_l in the domain of v_l such that there is a clause of the form $sat(c_j) \leftarrow v_t(val_t), v_l(val_l)$ in P_B, add *not* $v_t(val_t)$ to B. Likewise, for a value val_l in the domain of v_l such that there is no value val_t in the domain of v_t such that there is a clause of the form $sat(c_j) \leftarrow v_t(val_t), v_l(val_l)$ in P_B, then add *not* $v_l(val_l)$ to B.

The above inference rule is a direct enforcement of the arc-consistency in the translated logic program corresponding to the CSP. The CSP P discussed above can be now shown to lead to a **conflict** in the first call to **expand**. By IR5, add $\{not\ v1(1), not\ v1(2), not\ v1(3), not\ v1(4)\}$ to B. Then again by applying IR1, add

$v_1(1)$ to B causing a **conflict**. Hence **smodels** stops with one call to **expand**. The inference rule IR5 enriches the propagation achieved by $Cl(P, B)$ causing greater saving in search space in the application of **expand** to reduce the size of the search space traversed by **smodels**. This inference rule IR5, enforces full arc-consistency not only between variables in B and variables outside B, but also between variables outside B. In that sense it is a stronger version of arc-consistency as employed in MAC [17] than FC.

4.1.6 Conclusions on the constraint propagation in both systems

In this section, we saw the close semantic similarity between the propagation achieved by **expand** function in **smodels** and arc-consistency based constraint propagation techniques in CSPs. We first examined the role of constraint propagation in both systems by examples. We established sufficient conditions under which **expand** had exactly the same semantics as arc-consistency. We experimentally examined the relative amount of propagation performed by **expand** and FC. We found that the limited arc-consistency enforced by FC itself outperforms **expand**.

Further, we also saw how semantics from arc-consistency could be used to enrich the propagation in **expand** by adding a new inference rule which will work specifically for the logic programs representing binary CSPs.

4.2 Lookahead of smodels vs dynamic variable ordering of CSP

Lookahead, is a technique which employs a thorough examination of all future choices before committing to a choice. We study a related technique employed in CSPs, which semantically tries to achieve the same result as the **lookahead** technique in **smodels** . The corresponding technique in CSP is that of *dynamic variable ordering* (DVO).

4.2.1 Dynamic variable ordering in CSP

In a previous chapter, many classes of algorithms for solving constraint satisfaction problems were presented. These ranged from the simplest and costliest chronological backtracking through intelligent backtracking like **backjumping** to techniques involving combinations of *propagation* and backtracking like forward checking. In all these algorithms the order of instantiation of the variables was always fixed or predetermined. In other words the order of the instantiation of variables could not be changed during the process of obtaining a solution. In addition any constraint satisfaction requires the order in which the values are assigned to the variable on backtracking. The choice of the right order of variables (and values) can significantly improve the efficiency of constraint satisfaction [59].

The very notion of changing the order of variables to be instantiated during the process of solving, implies that if there is no change to the state of a CSP during a problem, then the dynamic variable ordering achieves nothing extra. In other words, if there is no change to domains of variables remaining to be instantiated during the solving process, the number of values corresponding to each remaining variable still remains the same as at the start of the solution. Hence the dynamic variable order would be the same as the fixed order of variables based on domain sizes selected at the start of the solution process. Hence a necessary requirement for the dynamic variable ordering to be effective is that there be a change in the state of the CSP, in terms of number of values for all remaining variables.

The above requirement is not satisfied by naive backtracking because the state of a CSP is not changed on the fly based on the partial solution at any stage. On the other hand, any constraint satisfaction algorithm which involves constraint propagation, ensures that the partial solution at any stage of the backtracking process, has an effect on the remaining variables yet to be instantiated. Thus any constraint satisfaction algorithm which involves the use of constraint propagation in conjunction with backtracking is a good choice for dynamic variable ordering. FC perfectly fits the case, as it consists of a certain amount of propagation interleaved with every incremental assignment of a value to a new variable. In FC the state of a CSP is changed after a new variable is instantiated, and the number of values corresponding to a variable is changed. This makes the application of any heuristic for dynamically re-ordering the variables worthwhile.

In the field of constraint satisfaction, several heuristics have been developed for selecting variable ordering. The most effective one is based on the *first-fail* idea, which described in simple words means that trying a variable which is likely to lead to a failure the fastest, is the best because of the enormous savings in the space possible by detecting this early failure.

The translation of the above semantics of the first-fail procedure is achieved in finite CSPs by assigning the variable with the fewest possible remaining values, next for instantiation. Thus at any stage during the process of solving the CSP, all the future variables are examined for all the remaining values and the variable with the least remaining values is chosen as the next variable to be instantiated. This leads to a different variable at a level in different branches of the search tree, unlike the case of the fixed variable ordering. This heuristic is effective because if the current partial solution does not lead to a complete solution, then the sooner we discover this the better it is. This heuristic reduces the average depth of branches in the search tree considerably by triggering early failure. We shall show the effect of dynamic variable ordering with the abovementioned heuristic, on forward checking, with an example below:

Consider a finite CSP P defined on four variables v_1, v_2, v_3 and v_4 with domains of $d_1 = \{1,2\}$, $d_2 = \{1,2,3\}$, $d_3 = \{1,2\}$, and $d_4 = \{1,2\}$ respectively. Let the problem P contain four constraints $C_1(v_1, v_2) = \{(1,2),(1,1)\}$, $C_2(v_2, v_3) = \{(2,1),(3,2)\}$, $C_3(v_3, v_4) = \{(1,1),(2,2)\}$, and $C_4(v_1, v_4) = \{(1,1),(2,2)\}$. At the outset, since the variable v_1 has the least number of values, namely 1, it is selected. Once $v_1 = 1$ is fixed, in the propagation phase of forward checking, it is checked against the future variables, v_2, v_3 and v_4. The variables v_3, v_2 are not connected to v_1. The variable v_4 which is connected to the variable v_1 is then tested and by propagation, the value of 2 is removed from the domain of v_4. At this stage among the remaining variables v_2, v_3 and v_4, the variable v_4 has the least number of remaining values, namely 1. By the heuristic, v_4 is instantiated to 1 next and the partial assignment $v_1 = 1, v_4 = 1$ is propagated among v_2 and v_3. Since only v_3 is connected to v_4, the propagation takes place on v_3 and the value of 2 is removed from the domain of v_3 as it is inconsistent with v_4. At this stage, among the two remaining variables v_2 and v_3, the one with least remaining values is v_3 with one value. The partial solution then expands to $v_3 = 1$. Propagating to v_4 the values of 3 and 1 are removed from the domain of v_2, leaving the only value 1, which is used to derive the final solution $\{v_1 = 1, v_4 = 1, v_3 = 1, v_2 = 2\}$. With the use of dynamic variable ordering in this case, we are able to avoid backtracking altogether.

4.2.2 Lookahead in smodels

In **smodels** with **lookahead**, the idea is to use a literal which is assured to give rise to an immediate conflict, thereby allowing us to filter a portion of the search space early on. At a point in **smodels** when no further literal can be added to B by **expand**, a new literal is chosen by **smodels** to be added to B. The function **lookahead** tries to expand with all literals and adds a literal l to B such that **expand**$(B \cup \{opp(l)\})$ leads to a conflict.

Consider the same CSP P as the one considered for demonstrating DVO. P is defined on four variables v_1, v_2, v_3 and v_4 with domains of $d_1 = \{1,2\}$, $d_2 = \{1,2,3\}$, $d_3 = \{1,2\}$, and $d_4 = \{1,2\}$ respectively. P contains four constraints $C_1(v_1, v_2) = \{(1,2),(1,1)\}$, $C_2(v_2, v_3) = \{(2,1),(3,2)\}$, $C_3(v_3, v_4) = \{(1,1),(2,2)\}$, and $C_4(v_1, v_4) = \{(1,1),(2,2)\}$. We assume working with the logic program output by *trans*2 on the CSP with the further restriction that initially **smodels** is invoked with $B = \{sat(c1), sat(c2), sat(c3), sat(c4)\}$ representing the four constraints. The initial call to **expand** does not add any new literal to B, as no inference rules are applicable. In the next step, **lookahead** checks with each literal l if **expand**($B \cup l$) leads to a conflict. On choosing $l = v1(2)$, **expand**($B \cup l$) causes conflict. Hence, **lookahead** enforces *not* $v1(2)$ to be added to B. Hence B is now $\{sat(c1), sat(c2), sat(c3), sat(c4), not\ v1(2)\}$. A call to **expand** adds $v1(1)$ to B. Next, **lookahead** finds a **conflict** with $l = v4(2)$ thereby adding *not* $v4(2)$ to B. In a call to **expand**, literals $v4(1), v3(1), not\ v3(2), v2(2), not\ v2(3)$, and *not* $v2(1)$ are added to B by repeated application of inference rule IR3. At this stage, B covers $Neg(P)$ and the stable model $\{sat(c1), sat(c2), sat(c3), sat(c4), v1(1), v2(2), v3(1), v4(1)\}$ corresponding to the assignment $\{v_1 = 1, v_2 = 2, v_3 = 1, v_4 = 1\}$.

4.2.3 Relationship between DVO and lookahead

The use of **lookahead** in **smodels** guarantees that the variable chosen next will fail first. On the same note, dynamic variable ordering in CSP ensures that the next variable to be chosen is the one with the least remaining values, so that it fails earliest. Hence the two notions of **lookahead** in **smodels** and *dynamic variable reordering* are used with the same intent. The rationale in either case is to cause an early failure in the expansion of a current partial solution to allow for a greater pruning of space.

We present some sufficient conditions when both the heuristics DVO and **lookahead** coincide for finite CSPs. DVO is applicable to variables with domains of any size. On the other hand, the **lookahead** in **smodels** operates on literals with only two values (l can be either l or $opp(l)$). To make the situations comparable, consider the setting in case of DVO applied to forward checking. Let us consider an intermediate stage where we propagate the latest assignment $v_i = val_i$ to all future variables. Of all the future variables, at this stage let v_j be a variable with two values $\{val_1, val_2\}$ remaining in its domain. Further assume that $v_j = val_2$ is consistent with $v_i = val_i$. Assuming that the domain of none of the future variables after v_i annihilates in the propagation step, the following sufficient condition shows the equivalence of the **lookahead** of **smodels** and DVO of CSP.

Theorem 4.3 *If the value assignment $v_j = val_1$ is inconsistent with $v_i = val_i$, both DVO and* **lookahead** *would select v_j as the next variable to be instantiated.*

Proof. The DVO heuristic will pick v_j as the next variable, because it has the least number of remaining values, namely 1. There is no variable such that its domain has zero values, for in that case the forward checking would stop at $v_i = val_i$ with no scope for any further assignment to any other variable. On the other hand, **lookahead** would select the variable v_j because assignment of one of the two possible values to v_j leads to a **conflict**. Hence the equivalence. □

4.2.4 Experimental comparison of lookahead and FC variants involving DVO

Which CSP algorithms to choose?

Let the forward checking procedure employing the heuristic of DVO be termed as FC–DVO. Similarly, let us term the algorithm where FC is employed with full arc-consistency as FC–arc. FC–arc–DVO refers to the FC–arc algorithm where DVO is employed instead of static variable order. Similarly FC–BJ refers to the FC algorithm with plain backjumping, and FC–CBJ refers to FC algorithm with conflict–directed backjumping. From the CSP literature, various experimental studies on random as well as benchmark problems [18, 19, 59, 47, 31] have shown that the above FC variants are among the most competetive among all CSP algorithms. In particular, previous experimental studies have shown that both CBJ and DVO invidually contribute maximum to the FC algorithm in terms of efficiency and reduction in search space. Based on these earlier results in the literature of constraint satisfaction, we decided to pursue the studies of **smodels** with **lookahead** with the abovementioned FC variants.

Experiments

Before we proceed further, let us clarify the notion of *node* in case of **smodels** with **lookahead**. In **smodels** with **lookahead**, on a call to **lookahead** a deterministic expansion of B occurs with *opp(l)* whenever $B \cup \{l\}$ leads to conflict. A series of calls to the **lookahead** procedure (number of calls is linear) is still polynomial in complexity. Thus all the processing that takes places between successive calls to a non-deterministic expansion of B, is considered as polynomial in nature. Because of this the number of nodes which are counted are the non-deterministic choices for expansion of B. Similar to FC, the node in FC–DVO and other FC variants refer to the expansion of a partial assignment by assigning a new value to a new variable.

The results in this section turned out to be quite exciting, and prompted us to perform a thorough experimental analysis of the two techniques. Only binary random finite CSPs were considered in the experimental set-up. The CSPs were all defined on $n = 20$ variables, each variable with a fixed domain size of $k = 5$. Random binary CSP instances were generated by varying two factors:

1. Constraint density (w) - It represents the number of constraints as a fraction of the number of possible constraints in the CSP.

2. Constraint tightness (d) - It represents the total number of tuples present in a constraint as a percentage of the total number of possible tuples.

Three sets of data were generated for experimental comparison. In all cases, measurements were taken to either generate one solution if it existed or to report that no solution was possible.

1. By keeping w (constraint density) fixed and varying the constraint tightness d.

2. By keeping d fixed and varying w (varying the the number of constraints).

3. By performing experiments on the hardest CSPs identified by the phenomenon of *phase transitions* [62, 61].

In all the cases, 100 random CSPs were generated for each value pair $< w, d >$ and the average (median) number of nodes computed for each $<w, d>$ combination.

The first set of experiments were performed by varying constraint tightness for a fixed constraint density w. In Figure 4.3, the number of nodes for FC-DVO (and other FC variants) and **smodels** is plotted for a fixed value of $w = 0.61$. In Figure 4.4, the execution times for FC-DVO and other FC variants and **smodels** is plotted for a fixed value of $w = 0.61$. The tightness d (in %) is varied from 10 to 90. The number of nodes is plotted against d. The results have been shown for a representative value of w and are expected to exhibit similar behavior for any value of w.

The second set of experiments were performed by varying the constraint density (varying the number of constraints) for a fixed tightness of constraints d. In Figure 4.5, the number of nodes for FC variants and **smodels** is plotted for a fixed value of $d = 64\%$ (16 tuples in each constraint). In Figure 4.6, the execution time for FC variants and **smodels** is plotted for a fixed value of $d = 64\%$ (16 tuples in each constraint). The number of nodes is plotted against w. The results have been shown for a representative value of d and are expected to exhibit similar behavior for any value of d.

Working with the hardest CSPs

In the literature of CSPs many experimental studies have proved the existence of phase transitions [62, 61, 48, 55]. *Phase transitions*, as explained in the previous chapter, refer to the phenomenon of sudden spikes in cost of search at specific points. This spike is observed due to the transition from the region of over-constrained CSPs (which do not admit a solution) to the region of soluble CSPs (which admit a solution). All these studies also point to the fact that the hardest CSPs fall in the region of phase transitions. The locations of the hardest problems have been shown to be independent of the search algorithm used in solving the CSP [55]. Experiments on random binary CSPs has confirmed this for various CSP algorithms like BT, and FC.

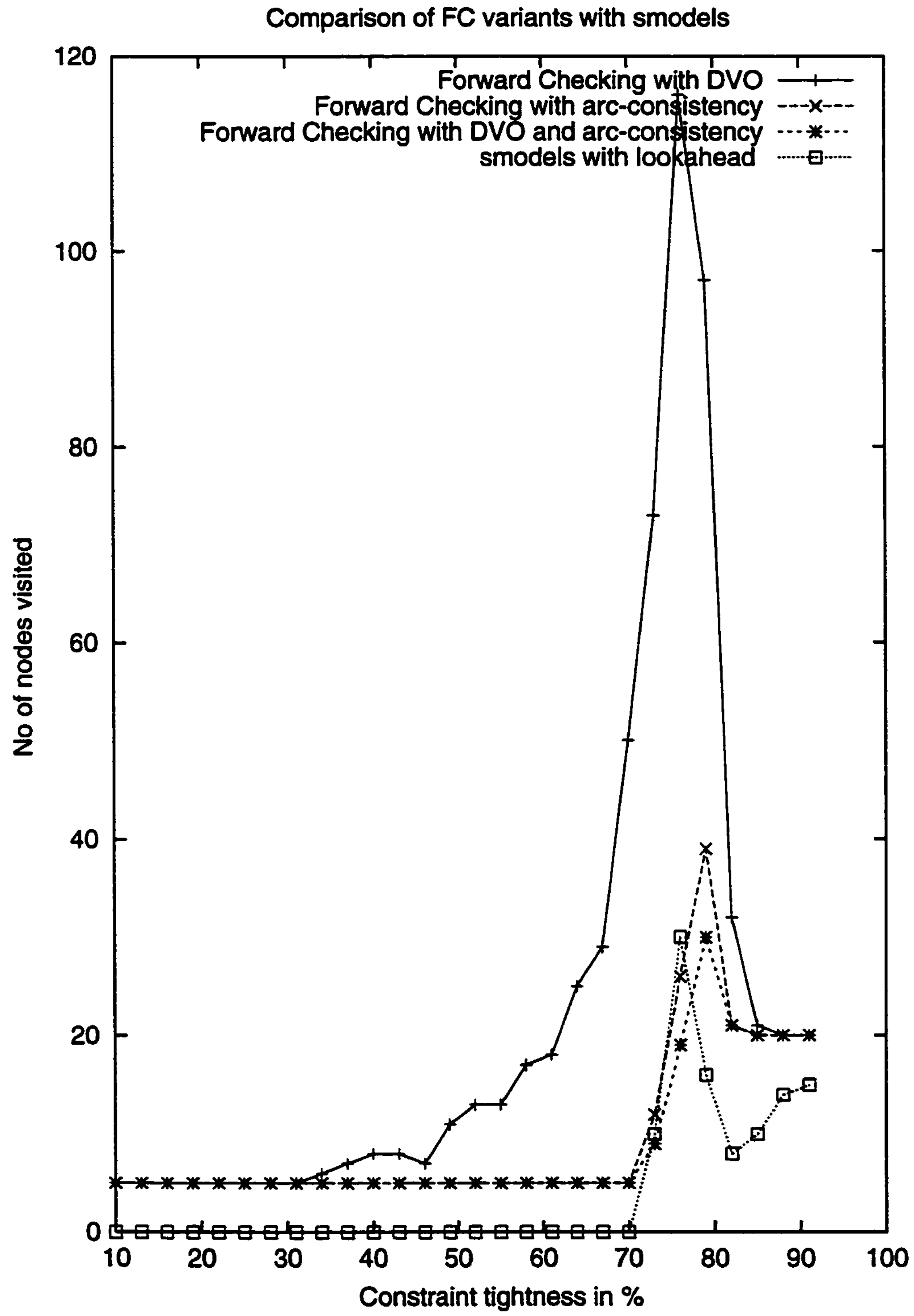


Figure 4.3: Comparison of median number of nodes of FC variants and **smodels** with **lookahead** for varying constraint tightness

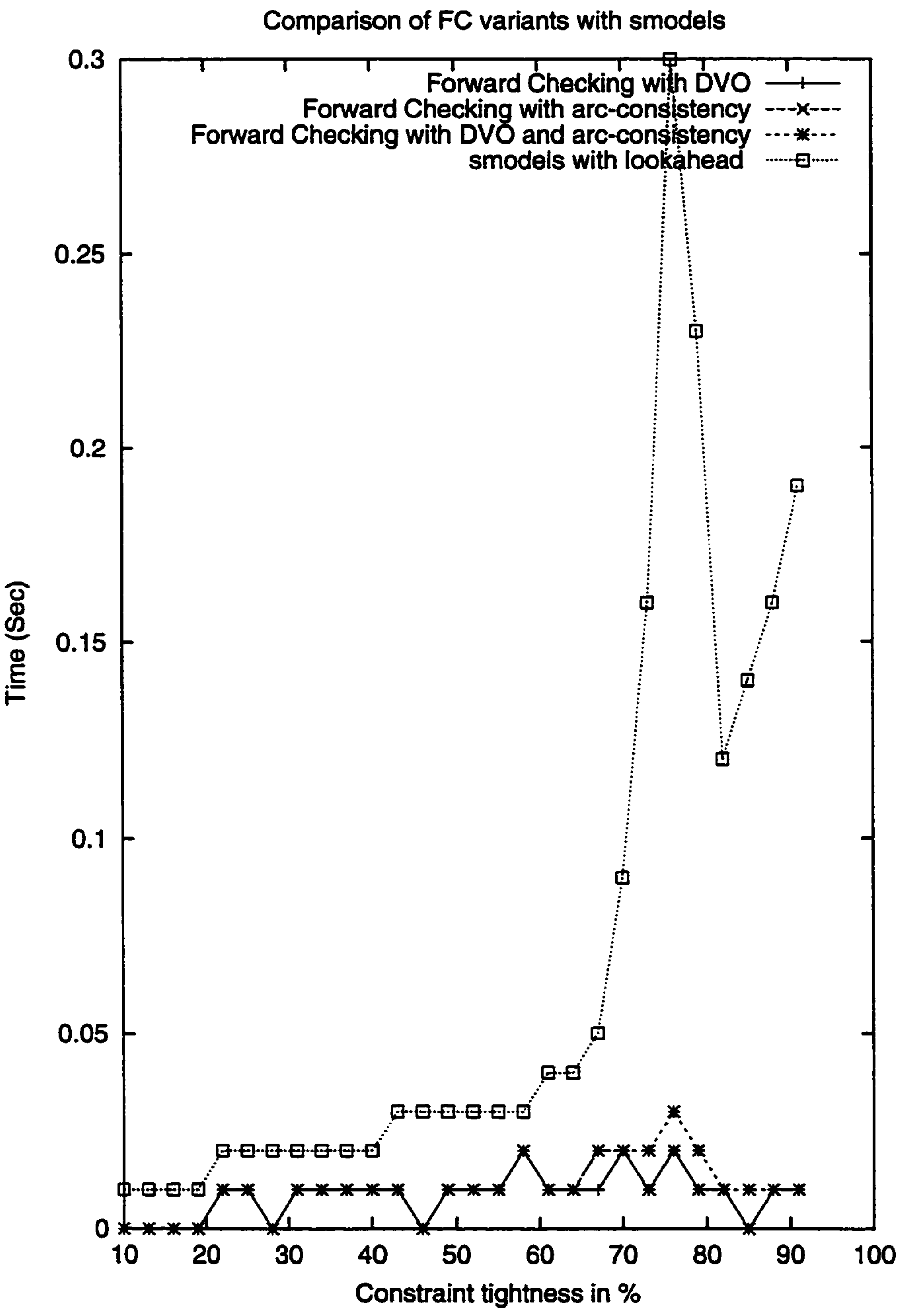


Figure 4.4: Comparison of median execution times of FC variants and **smodels** with **lookahead** for varying constraint tightness

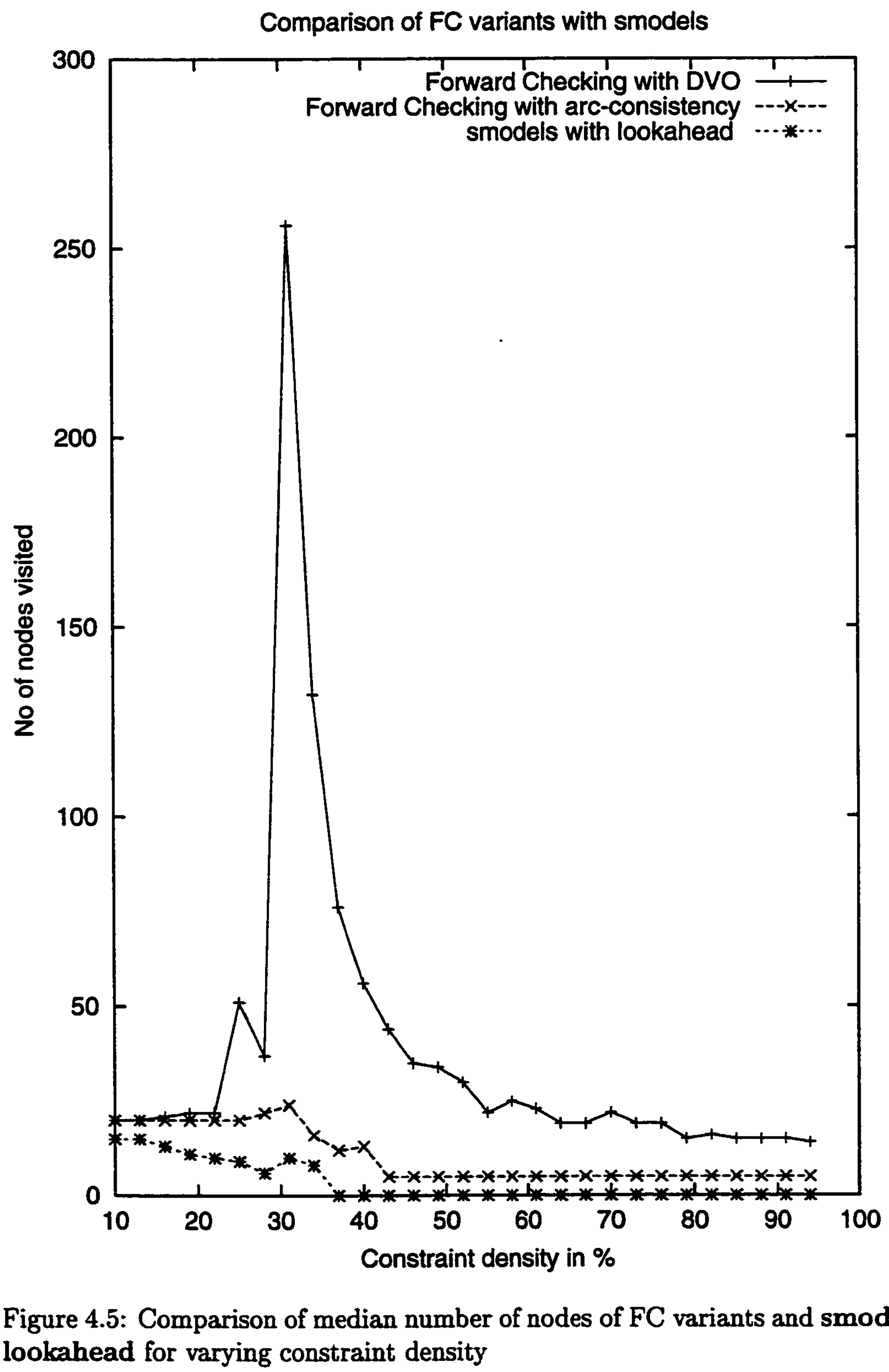


Figure 4.5: Comparison of median number of nodes of FC variants and **smodels** with **lookahead** for varying constraint density

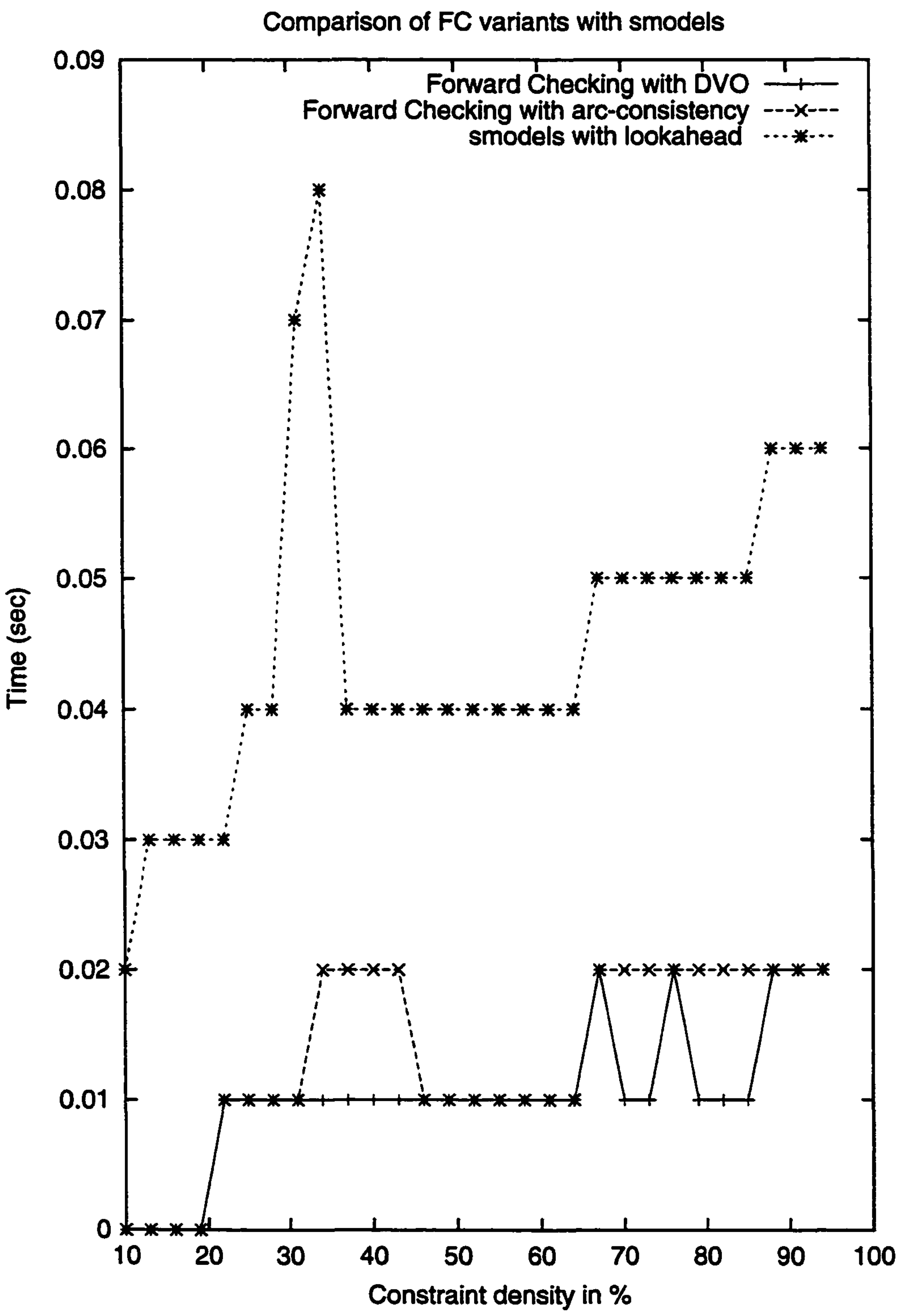


Figure 4.6: Comparison of median execution times of FC variants and **smodels with lookahead** for varying constraint density

w	d	FC-DVO	**smodels**	FC-BJ	FC-CBJ	FC	FC-Arc	Fc-Arc-DVO
0.20	0.43	42	0	853	323	899	5	5
0.40	0.65	78	6	1059	895	1353	10	8
0.60	0.75	109	25	1536	1302	1798	27	19
0.80	0.81	202	48	1820	1634	2193	63	38

Table 4.1: Comparison of nodes in CSP techniques with **smodels** employing **lookahead** for hardest CSPs

In [55], a theoretical approximation of the location of phase transition was provided which was corroborated by experimental results. To get results on the hardest CSP problems, we performed experiments on random CSPs with parameters defined by the theoretical location suggested in [55].

The location of the phase transition point [55] for a fixed w is given by

$$d = k^{-2/((n-1)w)}$$

where d denotes the tightness of the constraint (fraction of the number of possible tuples in a constraint), n is the number of variables, k is the uniform domain size of each variable, and w is the constraint density. Here n = 20 and k= 5. Hence d is given by the equation

$$d = 5^{-2/19w}$$

Experiments were performed on random CSPs with $<w, d>$ combination defined by the above equation. 100 samples were generated for each $<w, d>$ combination and the median number of nodes computed. Table 4.1 gives the number of nodes for some representative $<w, d>$ combinations. The techniques which have been used for the comparison are variants of FC algorithm.

The table shows that at all values of d and w, **smodels** with lookahead performs comparable to FC-Arc and FC-Arc-DVO, the most efficient of all the CSP techniques above. It beats all other techniques (FC,FC-DVO,FC-BJ,FC-CBJ) comfortably. The same set of experiments were measured for execution time, and the resulting data is shown in Fig. 4.2. It is immediately evident that the measure of solution nodes does

w	d	FC-DVO	**smodels**	FC-BJ	FC-CBJ	FC	FC-Arc	Fc-Arc-DVO
0.20	0.43	0.00	0.01	0.03	0.03	0.01	0.00	0.00
0.40	0.65	0.01	0.04	0.05	0.07	0.04	0.02	0.01
0.60	0.75	0.01	0.22	0.10	0.14	0.07	0.02	0.01
0.80	0.81	0.01	0.44	0.12	0.17	0.12	0.04	0.01

Table 4.2: Comparison of execution times of CSP techniques with **smodels** employing **lookahead** for hardest CSPs

w	d	FC-DVO	**smodels**	FC	FC-Arc	Fc-Arc-DVO
0.20	0.66	271	6	56,028	17	13
0.40	0.81	981	84	183,920	245	165
0.60	0.87	2103	304	473,026	398	265

Table 4.3: Comparison of number of nodes of CSP techniques with **smodels** employing **lookahead** for hardest CSPs for n=40

not quite present the exact picture as the execution time. The execution time of some techniques like FC-BJ and FC-CBJ are costlier than FC because of the overhead of bookkeeping.

Scaling the results

The experiments above were scaled for higher number of variables, and then the *execution times* and number of nodes both were compared for two sets of data: a). Varying w in $<w, d>$ for $n = 40$ and $d = 5$, and b). varying n where for each value of n, a fixed point ($w = 0.30$) was used to compute the relevant $<w, d>$ combination for a fixed $d = 5$.

For the first set of experiments, n was fixed at 40, and d was fixed at 5. For this set-up, both *execution time* and *number of nodes* were used for the measurement. The relevant phase transition point is computed for each value of w and the relevant measurements made. The number of nodes for this set-up for different w is shown in Table 4.3. The *execution times* for the same set-up is shown in Table 4.4. For each point $<w, d>$, the median value out of 20 sample runs were taken.

In the second set of experiments, for a fixed w, the relevant phase transition point d is computed for varying values of n and the median value computed for 20 sample

w	d	FC-DVO	**smodels**	FC	FC-Arc	Fc-Arc-DVO
0.20	0.66	0.04	0.24	1.02	0.08	0.06
0.40	0.81	0.09	3.55	4.27	0.25	0.22
0.60	0.87	0.13	14.5	9.85	0.53	0.29

Table 4.4: Comparison of *execution time* of CSP techniques with **smodels** employing **lookahead** for hardest CSPs for n=40

runs for each point. The plot of execution times for varying n is given in Fig. 4.8. The number of nodes for the same set-up is shown in Fig. 4.7.

4.2.5 Incorporation of DVO in smodels

The experimental results in the previous section show that DVO is able to prune search space better for many CSPs as compared to **smodels** with **lookahead**, especially for CSPs with higher density and tightness. Here we explore if DVO can be applied to **smodels** keeping in view the specific structure of the program generated by *trans*2 for finite CSPs. To apply the principle of DVO in **smodels**, let us consider how DVO can be used as the **lookahead** instead of the version of **lookahead** used in **smodels**. The modified **lookahead** should be looked upon in the context of **csp-smodels** which has been shown to be more suited (in terms of worst-case bounds) for the class of logic programs generated by *trans*2. The enforcement of full semantics of DVO in **lookahead** is used in the **dvo-smodels** procedure shown in Figure 4.9.

Let X be the set of all variables in a CSP whose translation by *trans*2 is the program P. At any intermediate point in the search space, B contains some atoms of the type $v_i(val_i)$ and some negated atoms of the form *not* $v_t(val_t)$. Atoms(B) contains all the atoms of the form $v_i(val_t)$ such that either $v_i(val_t)$ or *not* $v_i(val_t)$ appears in B. Let $Neg(P)$ represent all atoms of the type $v_t(val_t)$ in P. Let $Vars(B)$ be the set of variables v_j such there is at least one atom of the form $v_j(val_j)$ in B. Let $Nvar(B) = X - Vars(B)$. Further let $Pos(v_j, A)$ represent the set of atoms $v_j(val_k)$ for all val_k in the domain of v_j such that an atom of the form $v_j(val_k)$ appears in A. We assume the function **lookahead**(P, B) to return a set of atoms of

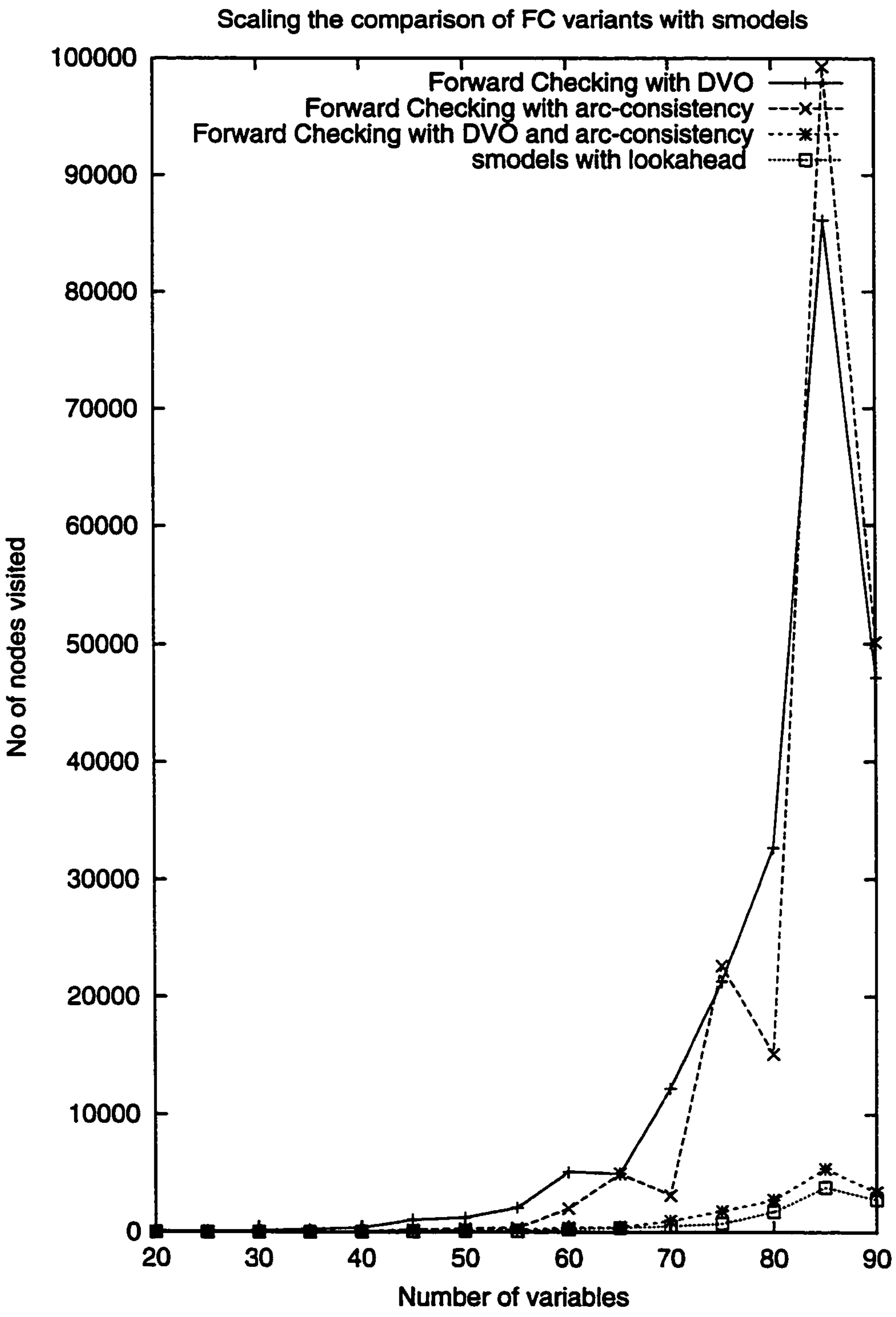


Figure 4.7: Comparison of median nodes of FC variants and smodels with lookahead for varying number of variables

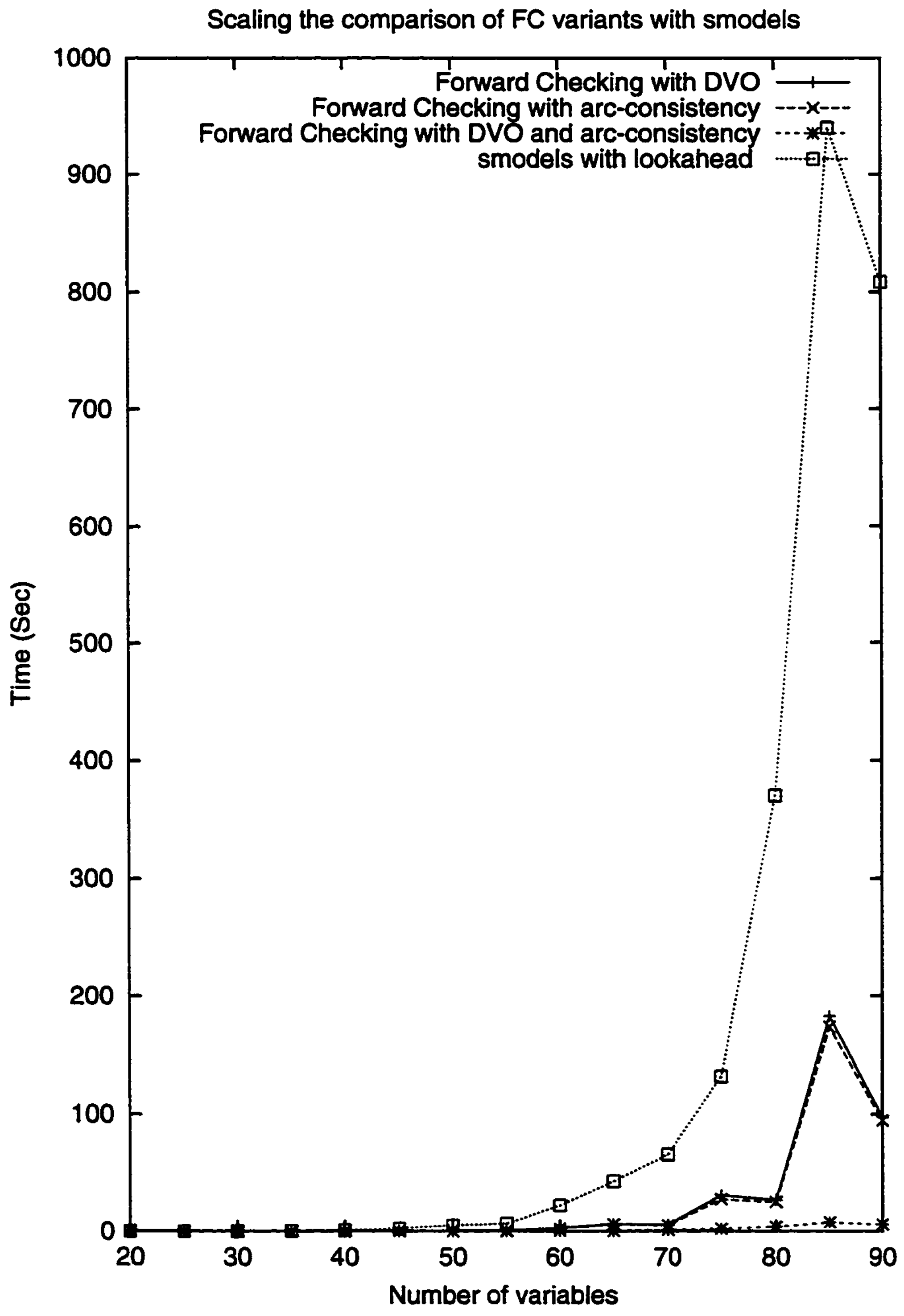


Figure 4.8: Comparison of median execution times of FC variants and **smodels** with **lookahead** for varying number of variables

the type $v_i(val_k)$ corresponding to a single variable v_i. The modified **dvo-lookahead** and the associated **dvo-smodels** procedure is presented in Figure 4.9. The **dvo-smodels** procedure modifies the **csp-smodels** procedure to take into the account the heuristic of *least-constrained* variable as the next one to be instantiated. The function **dvo-lookahead** returns the set of all possible values of the *least-constrained* variable among all the remaining variables. The **dvo-smodels** procedure then selects one of the possible values among these. We show that **dvo-smodels** is complete and sound.

Theorem 4.4 **dvo-smodels** *is sound and complete.*

Proof. The soundness follows from the observation that the procedure of **dvo-smodels** traverses through only consistent nodes, i.e. any node traversed by **dvo-smodels** does not have a value of B such that B contains an atom of the form $v_i(val_i)$ as well as a literal of the type *not* $v_i(val_i)$. Out of the consistent nodes, any solution node has a B such that B contains at least one atom of the form $v_i(val_i)$ for every variable v_i. So any such B corresponding to a solution node has a corresponding solution to the original CSP. Hence **dvo-smodels** is sound.

For completeness, we show that no solution is lost by the **dvo-smodels** procedure. Let $S = \{v_1 = val_1, v_2 = val_2, \ldots, v_n = val_n\}$ be a solution to the original CSP. Further let us assume that the set B_S (the set B corresponding to S) containing all the atoms $\{v_1(val_1), v_2(val_2), \ldots, v_n(val_n)\}$ is not output by **dvo-smodels**. Since S is a solution, B_S represents a consistent node. Now consider a partition of a set A, a set of disjoint subsets of A such the union of all the subsets is A. B_S can be partitioned in a number of ways. Each partition P contains a set of disjoint subsets $\{\{v_{i1}(val_{i1}), v_{i2}(val_{i2}), \ldots\}, \ldots, \{v_{ik}(val_{ik}), v_{in}(val_{in})\}\}$. There is at least one such partition P such that the first set of atoms in P correspond to the set of atoms instantiated in the first propagation step of **dvo-smodels**, followed by the second set of atoms in the partition in the second step in **dvo-smodels** and so on. Clearly this

```
function dvo-lookahead($P, B$)
counter := Infinity % A large number
val := $\phi$
$A := Neg(P)$ - $Atoms(B)$
for all $v_i$ in $Nvar(B)$
do
$t := Pos(v_i, A)$
tempcounter := | $t$ | % Size of $t$
if tempcounter $\geq$ counter then
    counter := tempcounter
    val := $t$
endif
enddo
return val

function dvo-smodels($P, B$)
$B'$ := csp-expand($P, B$)
if conflict($P, B'$) returns true then
    return false
else if for all $v_i \in \{v_1, \ldots, v_n\}$ there is an atom of the form $v_i(val_i)$ in $B'$
    then
    return true
else
    Let $S$ := dvo-lookahead($P, B$)
    For all $x \in S$
    do
    if dvo-smodels($P$, $B' \cup \{x\}$) returns true then
        return true
    endif
    $B' := B' \cup \{not\ x\}$
    enddo
    return false
endif
```

Figure 4.9: The **lookahead** procedure for DVO

partition corresponds to a particular sequence of steps taken by the **dvo-smodels** procedure. Since the union U of P contains an atom of the form $v_j(val_j)$ for all variables v_j, the set U along with all the other literals consistent with the atoms forms a set B which is output by **dvo-smodels**. Since B_S is a subset of such a B, B_S represents a solution node in the search tree of **dvo-smodels**. This contradicts the assumption of such a set B not being output by **dvo-smodels**. Hence **dvo-smodels** outputs all the solutions of the CSP and is complete. □

4.3 Backjumping in CSPs and backjumping in smodels

In **smodels**, **backjumping** is employed as one of the speed-up techniques for pruning the search space. Backjumping refers to a kind of intelligent backtracking, which is able to prune a larger search space than naive backtracking. In this section, we shall try to examine the relationship between the techniques of **backjumping** as employed in **smodels** and CSP. To make things clearer let us refer to the technique of **backjumping** as used in CSPs by the term *csp-backjumping.*

4.3.1 Backjumping in CSPs

The technique of *csp-backjumping* in finite CSP makes the process of backtracking intelligent by keeping track of the dependency relation between a current variable and a past variable. In contrast to chronological backtracking which backtracks to the immediately previous variable, *csp-backjumping* backtracks directly to a past variable which is the source of the conflicts at the current level. The backjump point is the deepest past variable in conflict with the variable at the present level.

The improvement in efficiency occurs due to the fact that the variables which are between the present level and the backjump point are bypassed.

Consider the CSP P defined on 6 variables v_1, v_2, v_3, v_4, v_5, and v_6 each with the same domain $d = \{0, 1, 2, 3, 4\}$. Let P contain 4 constraints $c_1(v_3, v_6) = \{(0, 1), (1, 3),$

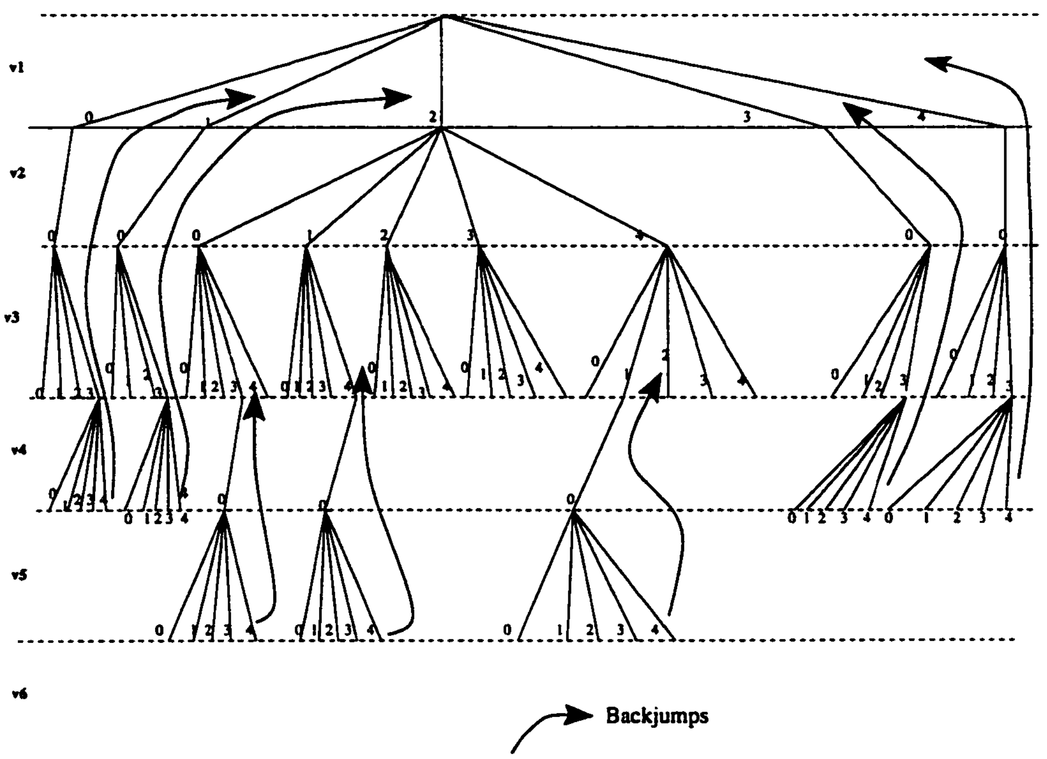


Figure 4.10: Search space of *csp-backjumping*

(4, 4)}, $c_2(v_1, v_4) = \{(2,0), (2,4)\}$, $c_3(v_2, v_3) = \{(1,4), (4,1), (0,3)\}$, and $c_4(v_5, v_3) = \{(4,0), (2,2)\}$. The search tree of *csp-backjumping* is shown in Figure 4.10. At the point $\{v_1 = 0, v_2 = 0, v_3 = 3\}$, all values of v_4 are in conflict with v_1. So any further value of v_2 or v_3 does not resolve the conflict between v_1 and v_4. Hence *csp-backjumping* directly directly jumps to the next value of v_1, namely 1. A total of 94 nodes are traversed by *csp-backjumping* in this example.

4.3.2 Backjumping in smodels

In the context of **smodels**, **backjumping** is not literally used to represent the same technique as used in CSP. In plain **smodels** without **backjumping**, on reaching a conflict at a present literal l, backtrack occurs to $opp(l')$ where l' is the literal chosen immediately prior to l. But if the literals l' and l are not connected by a path, then following the path of $opp(l')$ will again lead to a conflict. The technique of **backjumping in smodels** ensures that if there is no path between l' and l, then the path along *opp*(l) is avoided causing a backtrack to the literal l'' immediately preceding l'. Again if there is no path between l'' and l, the path of *opp*(l'') is avoided. This process continues till a literal l_n is reached such that l_n (let us call it the *backjump point*) is connected to l. Thus **backjumping** jumps over a series of literals immediately prior to a conflict-point l, none of which is connected to l.

Consider the same CSP P as used to illustrate plain *backjumping* in CSPs. The search space explored by **smodels** with **backjumping** is shown in Figure 4.11. At the conflict point $v2(3)$, the immediately preceding literal is $v4(1)$. But $v4(1)$ is not connected to $v2(3)$. Thus backjump takes place over $v4(1)$. This backjump jumps over literals $v4(2), v4(3), v1(0), v1(1), v1(3), v1(4)$ all of which are not connected to $v2(3)$ till it reaches $v6(0)$ which is connected to $v2(3)$. A total of 23 nodes are traversed by **smodels with backjumping.**

4.3.3 Relationship between backjumping in CSPs and smodels

Both CSPs and **smodels** use **backjumping** with the same intent, i.e. to make the process of backtracking more intelligent. However, there is one fundamental difference between the two notions of backjumping. In **smodels**, all intermediate variables l' between the jump point l_j and the conflict-point l are not connected to the conflict-point l. On the other hand, there is no such restriction on the intermediate variables in **csp-backjumping**. We state a condition (based on certain simplifying assumptions) under which **backjumping** achieves the same results in both CSPs and **smodels**.

Consider a point in the search by **smodels** where a conflict is reached on a literal $v_i(val)$. Let $v_j(val')$ represent the literal which is the deepest literal in the path from the root to $v_i(val)$ (in the search tree of **smodels**) such that there is a path between the variable v_j and v_i in the constraint graph of the CSP. Further assume that for each variable v_k, there is a maximum of one literal $v_k(val_k)$ in the path from the root to $v_i(val)$. Clearly this identifies a unique choice of a value for each variable

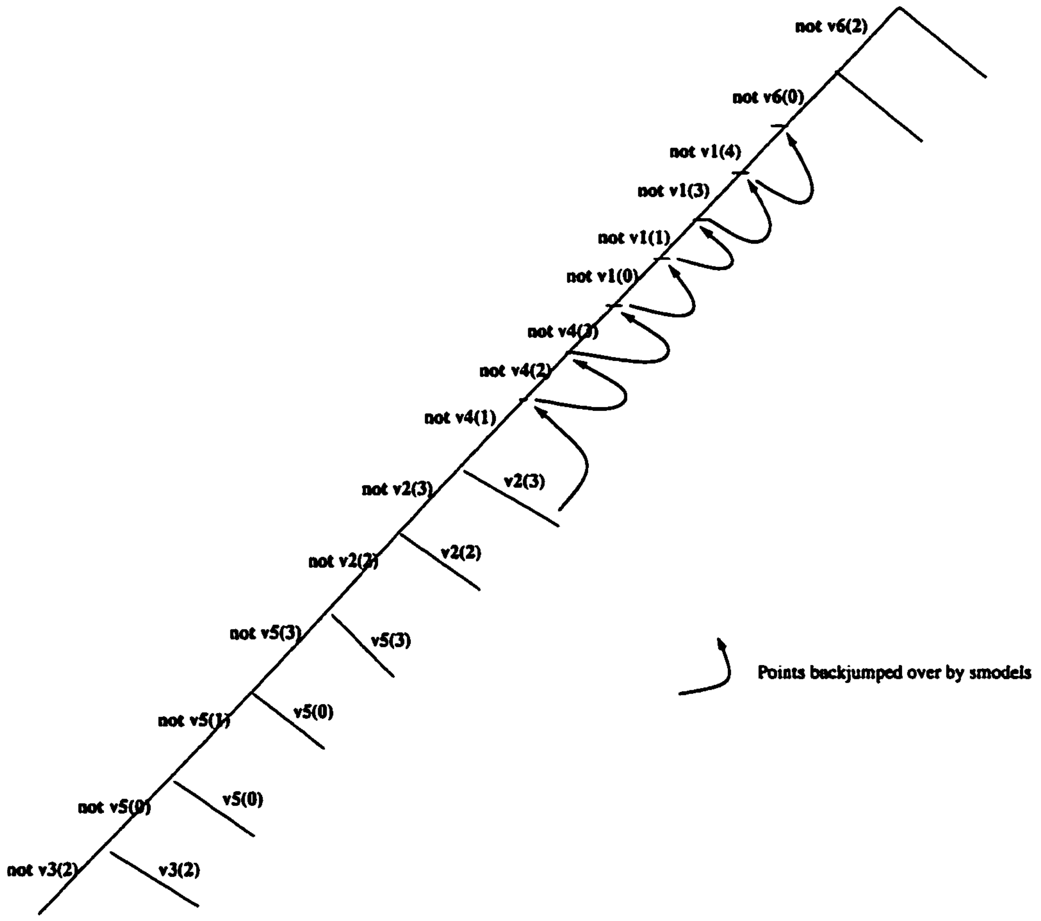


Figure 4.11: Search space of **smodels** with **backjumping**

involved in the path from the root to $v_i(val)$. Assume further that the CSP is solved by *csp-backjumping* such that the order of instantiation of the variables is the same as the one instantiated by **smodels**. If v_j represents the deepest variable which is in conflict with the current variable v_i, both situations are comparable. Then the variables responsible for conflict at $v_i(val)$ are either at or above the node $v_j(val')$ in both **smodels** and *csp-backjumping*. Based on this situation, the following condition precisely states when backjump occurs to $v_j(val')$ in both cases.

Theorem 4.5 *If the deepest variable v_j in conflict with the current variable v_i is also the deepest variable connected by a path to v_i in the constraint graph, then both* **smodels** *and csp-backjumping would jump to v_j.*

Proof. The result follows from the observation that the procedure of **csp-backjumping** would jump to v_j because it is the deepest variable in conflict with v_i. The procedure of **smodels** jumps to a literal l as it is the deepest literal in the search path not connected to $v_i(val)$ by a path in the program reduct P_B. But since v_j is the deepest variable in the original constraint graph connected to v_i, there is no path between any intermediate literal (between $v_i(val)$ and $v_j(val')$ and the literal $v_i(val)$. Hence **backjumping** in **smodels** falls through to $v_j(val')$ and thus the proof. □

A question now arises if either variety of **backjumping** can be incorporated in the other. First let us consider the possibility of incorporating *csp-backjumping* in **smodels**. **Csp-backjumping** relies on the chronological order of the instantiation of the variables in the CSP. Hence it is possible to conclude that the source of conflicts at the present level are all either at or above the jump point. But in **smodels** the conflict at any level is not necessarily due to the literals above the jump point, it may be due to future literals added after the conflict point. The checking of conflict does not rely on the order of instantiation of the variables in the CSP. Hence it is not possible to state that precisely the literals present above or at the jump point are responsible for the conflict. Thus *csp-backjumping* cannot be incorporated into

smodels directly.

On the other hand, if **backjumping** of **smodels** is to be incorporated in **csp-backjumping**, we only need to ensure that the intermediate literals between the jump point v_j (as determined by **smodels** style **backjumping**) and the conflict point v_ia re not connected to v_i in the constraint graph. But in any such situation, trivially *csp-backjumping* would backjump over all these intermediate variables because none of these would contribute to the conflict at v_i. The deepest variable in conflict with v_i would be either v_j or a point above v_j. Thus *csp-backjumping* subsumes the **backjumping** of **smodels**.

This observation leads us to believe that the technique of **backjumping** in **smodels** is a special case of the technique of backjumping in CSPs in terms of situations where it can be applied.

4.3.4 Experimental comparison of backjumping in smodels and CSPs

We compare the effectiveness of **backjumping** on random constraint satisfaction problems using the *csp-backjumping* method of CSPs and **smodels** with **backjumping**. The comparison was made for a fixed constraint density w=0.15, number of constraints n = 20, and fixed domain size k=5. The average number of nodes traversed by each method was taken as the average of 100 random CSPs generated for each value of constraint tightness d. The value of d was varied from 1 to 90 (in %). The graph is shown in Figure 4.12.

The technique of **smodels** with **backjumping** performs better than *csp-backjumping*. This can be attributed to the fact that **csp-backjumping** does not employ any constraint propagation in between successive choices. On the other hand **smodels** performs constraint propagation steps in between choice of literals. A natural extension then would be to compare it with a CSP technique employing both *csp-backjumping* and constraint propagation. One such technique is forward checking with backjump-

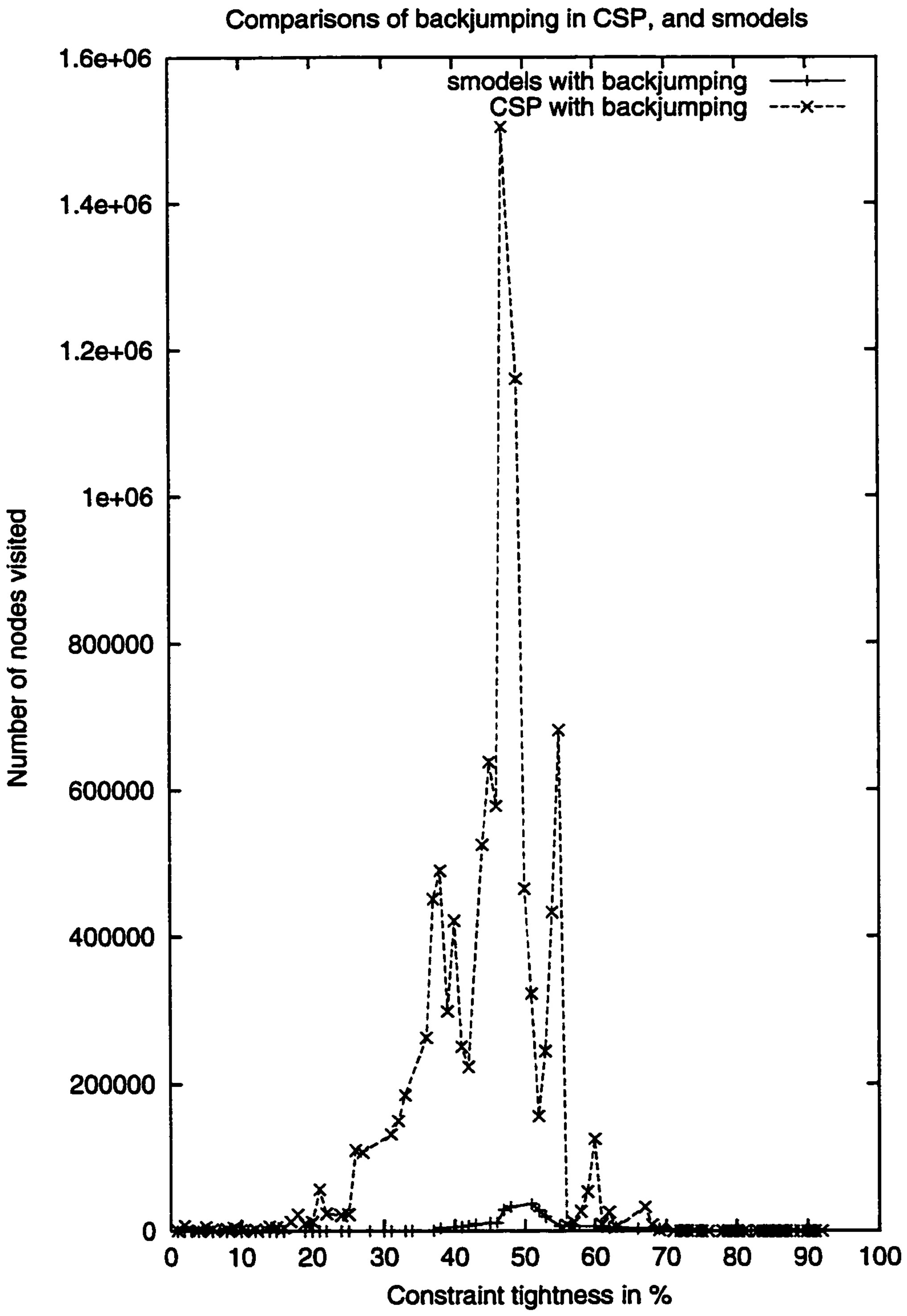


Figure 4.12: Comparison of backjumping in CSPs and smodels

ing (FC-BJ). FC-BJ employs forward checking alongside **csp-backjumping**. The results of experimental comparison of **smodels** with **backjumping** and FC-BJ are shown in Figure 4.13. We find that the both algorithms perform nearly the same. This is explainable as both involve interleaved propagation and **backjumping** steps.

4.4 Relative comparison of the techniques in smodels

As already seen, the three primary techniques responsible for the efficiency of the **smodels** system are :

1. Constraint propagation.
2. Lookahead.
3. Backjumping.

Constraint propagation is inherent in the definition of the **smodels** system. The **smodels** system interleaves the calls to **expand** function and the **smodels** function while finding a stable model. The role of **expand** function is to delay the choice of the next literal by **smodels** as much possible. By delaying this choice, **expand** could add a large number of literals to the set B before the next call to the **smodels** function. Thus **expand** function performs the operation of constraint propagation in the **smodels** system. On the other hand, the techniques of **lookahead** and **backjumping** are additional efficiency techniques incorporated into the **smodels** procedure. In this section we shall present the relative effectiveness to find the answer to the question: Which technique is responsible for the efficiency of **smodels** ?

4.4.1 Experimental comparison of backjumping and lookahead

In this section we shall present experimental results related to the relative effectiveness of the main techniques used in **smodels**, **backjumping** and **lookahead**. The

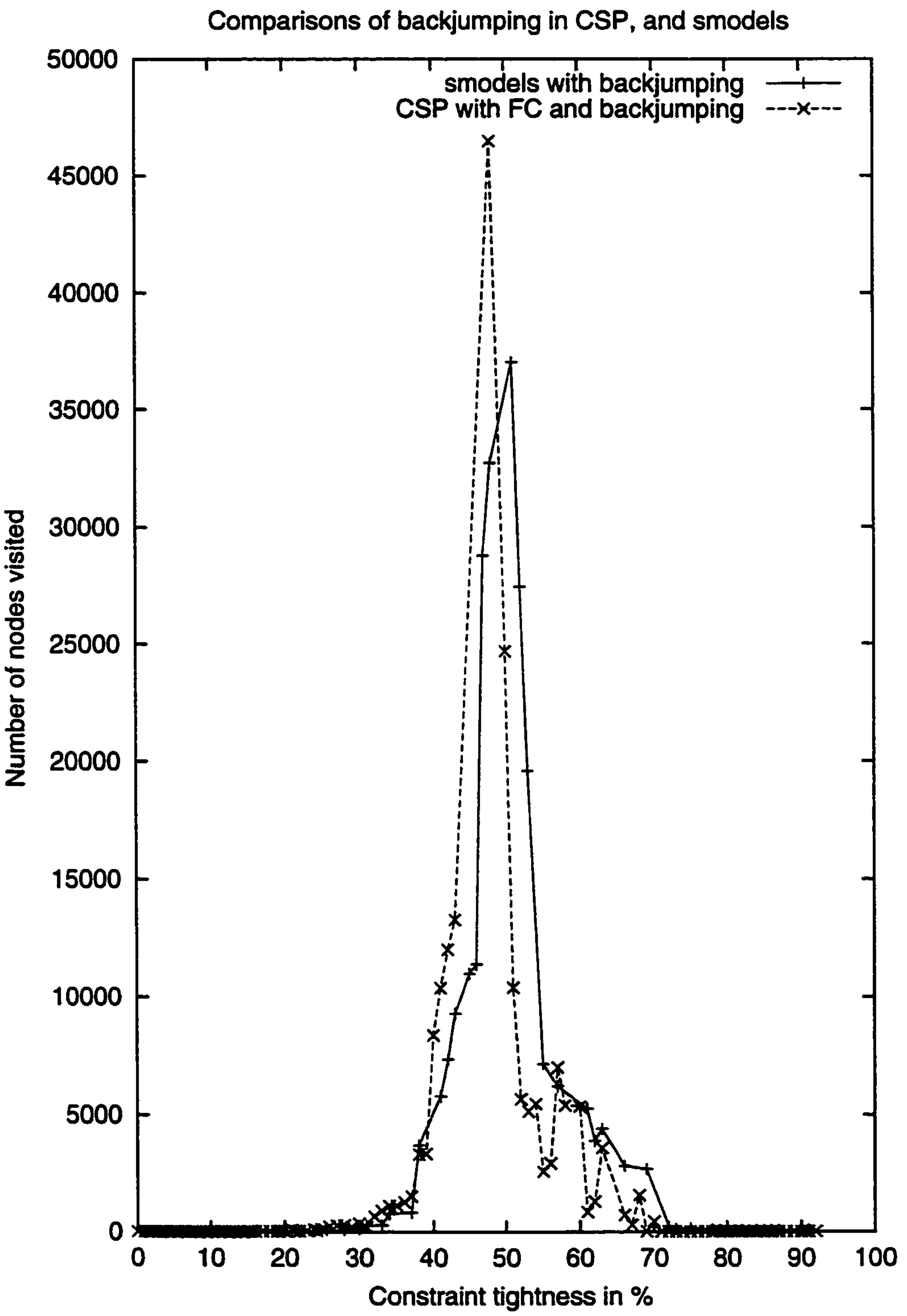


Figure 4.13: Comparison of FC-BJ in CSPs with smodels employing backjumping

experiments were conducted on two dimensions: *execution time* and *size of search space.*

In the first set of experiments, binary CSPs were considered on $n = 20$ variables, each with a domain size of $k= 5$. The experiments were conducted by varying constraint tightness d, keeping the constraint density w fixed at $w = 0.15$. The experiments were run on programs generated by $trans2$ with the assumption that initially B contained all atoms of $sat(c)$, where c is a constraint. In all the cases, either it was tested if no solution existed or one solution was returned if one existed. The number of nodes in search space was considered for four different cases:

1. With both **backjumping** and **lookahead**

2. With only **backjumping**

3. With only **lookahead**

4. With neither **backjumping** nor **lookahead**

The graphs comparing the number of nodes in the four cases is shown in Figure 4.14.

In the second set of experiments, the relative measurement is conducted by execution time. All these experiments were carried on Pentium PII machines with 400 MHz clock, running RedHat Linux version 5.1. The graphs showing the relative effectiveness of these techniques on random constraint satisfaction problems is shown in Figures 4.15 and 4.16. This set of experiments was performed on finite constraint satisfaction problems of arity $m = 3$, defined over a set of $n = 40$ variables, each of domain size $k = 5$, and each problem having 15 constraints. To scale the problems we use the number of tuples in the CSP on x-axis instead of the tightness. Both give the same graph except for the scaling factor. The time is again measured in terms of the average over a sample of 20 problems sampled over the ensemble for each value of tightness of the CSP. The relative effectiveness of the two techniques

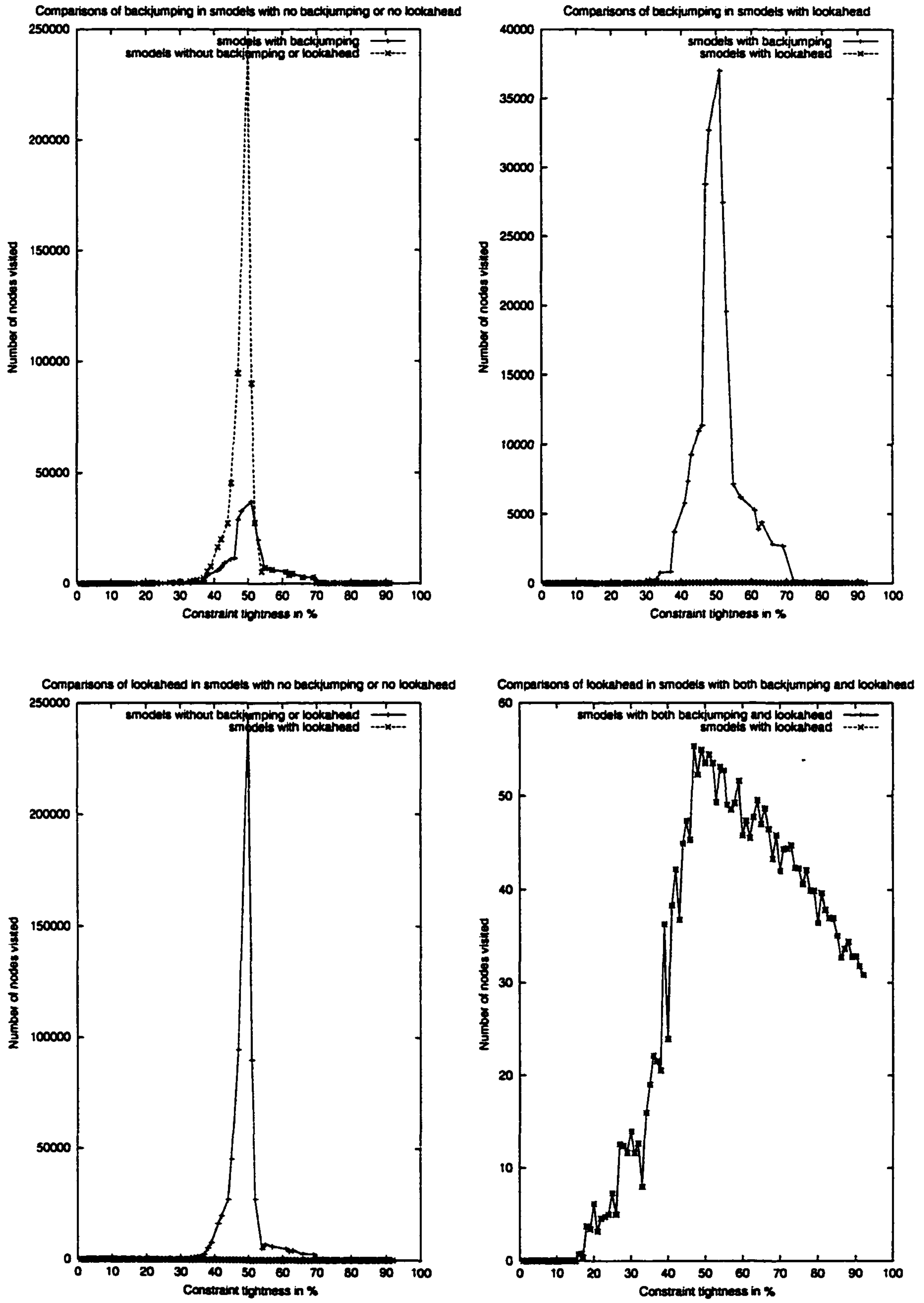


Figure 4.14: Relative comparison of the techniques in smodels for random CSPs

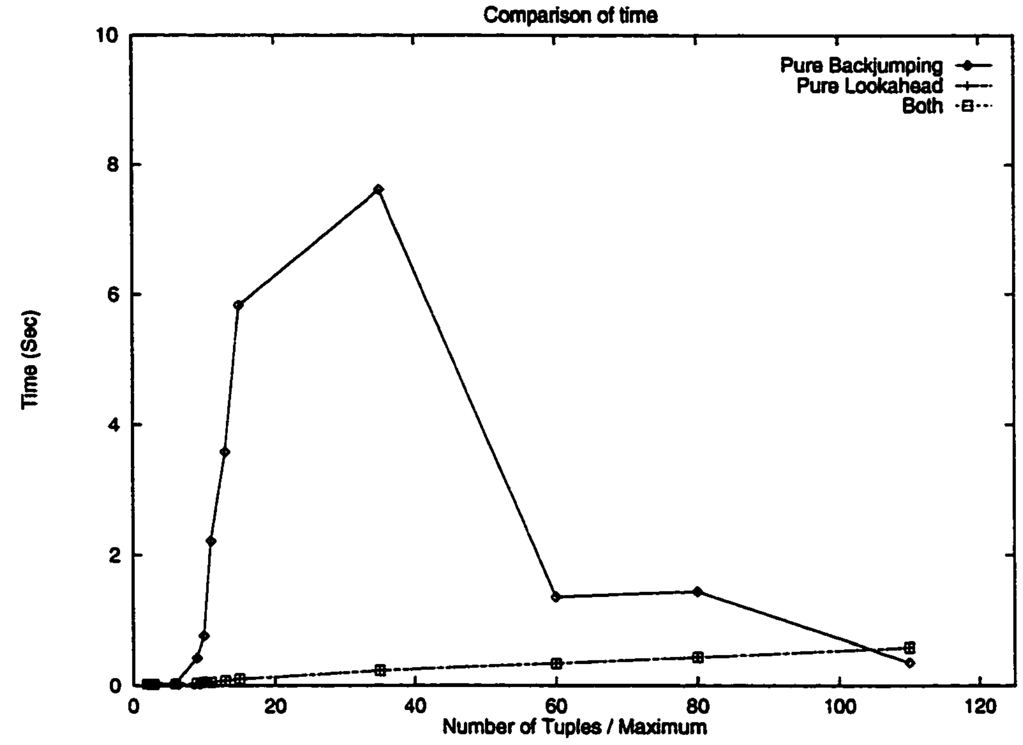


Figure 4.15: Relative effectiveness of both techniques for n=40, m = 3 and k=5

of **backjumping** and **lookahead**, is shown in the above setting. When no **lookahead** or **backjumping** was employed, the execution time in certain problem cases was crossing a quantum of 0.5 hr. In these cases we approximated them to infinity as compared to normal data values.

4.4.2 Discussion of experimental results

The graphs show that of **lookahead** and **backjumping**, the **lookahead** is a more effective technique as compared to **backjumping**. In fact the dominance of **lookahead** over **backjumping** is such that there is absolutely no additional savings in terms of pruning of search nodes by addition of **backjumping** to **lookahead** (shown by the fourth graph in Figure 4.14). The other fact to be noted is that the presence of at least one of the techniques (**backjumping/lookahead**), significantly enhances the efficiency of the system, though **backjumping** is not as effective as **lookahead**.

Another significant observation is the occurrence of the phenomenon of phase

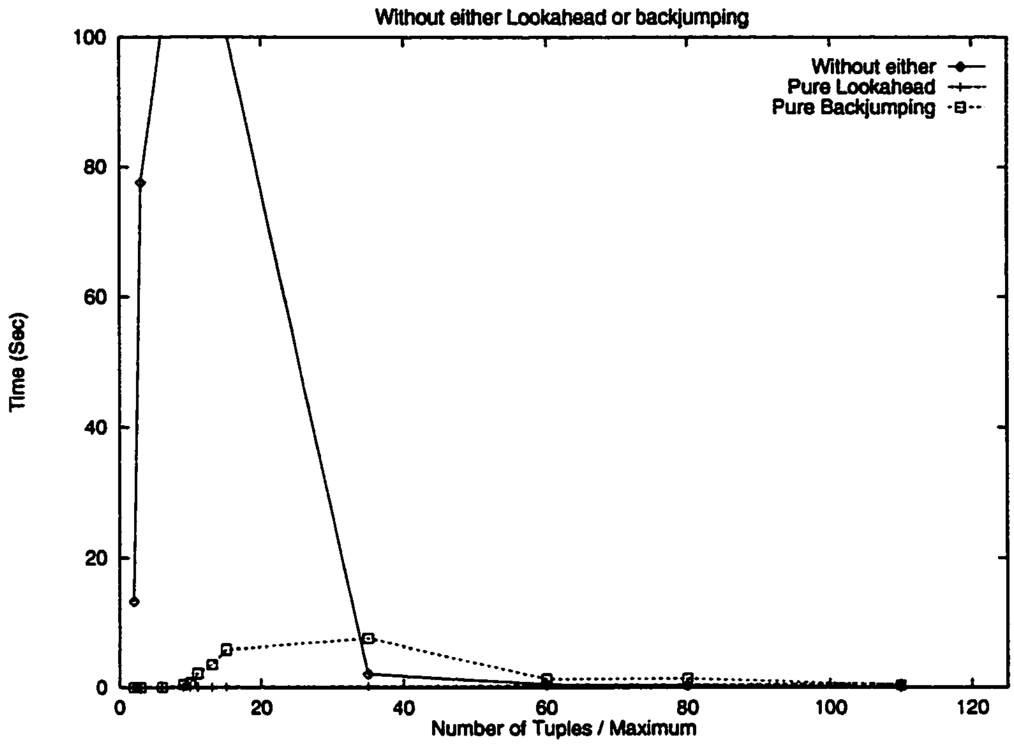


Figure 4.16: Comparison against no technique employed for n=40, m = 3 and k=5

transitions in all the techniques in **smodels**. This corroborates to the universality of the phenomenon of phase transitions which has been shown to be exhibited by all search algorithms including CSP techniques, and SAT procedures.

To summarize, the results from the experiments can be described below:

- The presence of at least one of the techniques **lookahead** or **backjump** has a considerable impact on the efficiency than when neither of them is present.
- When both **lookahead and backjumping** are used, **lookahead** dominates the search process, as evident from the graphs.
- In all the techniques, phase transitions has been observed.

4.5 Conclusions

Here we have studied the implementation techniques in **smodels**, and showed the mapping between the techniques used in **smodels** and CSP techniques. There are mainly three implementation techniques in **smodels** - constraint propagation, **lookahead** and **backjumping**. We showed how these techniques map from the common constraint satisfaction techniques of constraint propagation, dynamic variable ordering, and backjumping in CSPs. We also performed experimental comparison of the corresponding techniques. The technique of **lookahead** turned out to be competitive to the best finite CSP techniques even for the hardest constraint satisfaction problems.

In the end we performed a study of the relative effectiveness of the main techniques involved in **smodels** for the purpose of solving finite CSPs. We showed that **lookahead** outperforms **backjumping** in **smodels** .

Overall, a firm link between the areas of finite constraint satisfaction and **smodels** has been established with special stress on the correspondence of the implementation techniques in both. Further, **smodels** with **lookahead** promises to be an efficient constraint programming system.

Chapter 5

A Critique of Over-constrained Semantics and Solution Methods

5.1 Introduction

In Chapter 2, we discussed the semantics of the most common frameworks for over-constrained systems. In this chapter, we highlight the deficiencies of the semantic notions of solution in these over-constrained frameworks. We illustrate two main factors contributing to this critique: Higher computational complexity of these semantics, and non-preservation of semantics in translation from non-binary to binary representations. In the end of this chapter, we also provide a critical analysis of the solution methods for maximal constraint satisfaction problem (max-CSP) and related over-constrained problems.

In Section 5.2 of the chapter, we provide a complexity-theoretic analysis of various semantics of over-constrained systems. The results show that use of the common semantics of over-constrained problems is responsible for their falling into a higher computational complexity class than the pure CSPs.

Later in Section 5.3 we show that the most common methods of translation from non-binary to binary representations do not preserve the semantics of max-CSP. A bulk of research in the CSP community has been restricted to binary representations, based implicitly on the assumption that any non-binary CSP can be converted to a binary CSP still preserving the semantics. In this part of the chapter, we show that this assumption cannot be extended to max-CSP and other over-constrained problem semantics, thereby justifying the need for concentration of research on non-binary representations of over-constrained systems.

Finally in Section 5.4, we present extensions to the present techniques for solving max-CSP and related over-constrained problems. We critically analyze the solution techniques and provide a theoretical characterization of these techniques.

5.2 Complexity analysis of over-constrained systems

5.2.1 CSP and NP-completeness

NP-complete problems [20] represent a gamut of known computationally hard decision problems in computer science. A decision problem is said to be in P if it can be answered by a deterministic Turing machine in polynomial time. In contrast, any decision problem is said to be in NP, if a non-deterministic Turing machine returns a positive answer to a non-deterministic "guess" of a solution in polynomial time assuming a polynomially bounded size of the encoding of the "guess". Informally, NP refers to the class of decision problems which are solvable by polynomial time nondeterministic algorithms. Further, FNP is the class of all functions from strings to strings that can be computed by a polynomial-time non-deterministic Turing machine. On the other hand, the class co-NP refers to the class of problems P such that the complement P^c is in NP.

If a decision problem, say S, is in NP and any problem in NP can be reduced in a polynomial transformation to S, then the problem S is said to be NP-complete. Any NP-complete problem can be reduced to any other NP-complete problem in polynomial time. NP-complete problems refer to the class of hardest problems in NP, in the sense that if any one NP-complete problem can be solved in polynomial time, so can all the problems in NP be. Using this equivalence relation, we usually need to find the proof for one of the NP-complete problems, and then we can use a polynomial reduction from this proved one to any other NP-complete problem. A problem P whose complement P^c is NP-complete is termed as a co-NP-complete.

Historically, the satisfaction problem (SAT) was the first problem to be shown as NP-complete [20]. Later thousands of problems in computer science, engineering, mathematics and other areas have been proven to be NP-complete.

We recall the SAT problem definition below:

Let V be a set of boolean variables $\{v_1, v_2, \ldots, v_n\}$. A satisfying truth assignment t is a function t: $V \rightarrow \{T, F\}$. A literal v is satisfied if the variable v is assigned T. A literal $\overline{v}$ is satisfied if the variable v is assigned F. A clause C_i, which is a collection (disjunction) of literals is satisfied iff at least one of the literals in C_i is satisfied. So a set of clauses C is satisfied iff all the clauses in C are satisfied. The SAT decision problem is stated as follows [20, 8]:

Definition 8 (SAT) *Given a set V of boolean variables (each variable v_i is present either as a literal v_i or a literal $\overline{v_i}$), and a collection of clauses C over V, where each clause represents a disjunction of literals, is there a satisfying truth assignment for C?*

In [8] SAT was shown to be NP-complete.

Theorem 5.1 (Cook's theorem) *[8] SAT is NP-complete.*

The decision problem in finite constraint satisfaction problem (CSP) has been shown to be NP-complete [35]. Recall that a finite CSP involves a set $X = \{x_1, x_2, \ldots, x_n\}$ of n variables, and a set C of constraints, where each variable $x_i \in X$ takes a value from its finite domain d_i and each constraint $c_i \in C$, defined on a subset X_c of the set of variables X, is a relation expressed as an implicit function over the variables in X_c. Each constraint c_i implicitly specifies the admissible combinations of the values of variables involved in the constraint. An assignment A of values to variables in

$Y \subset X$ satisfies a constraint c such that $X_c \subset Y$ iff the tuple formed by X_c in A is admissible by the function associated with c. An assignment of values to a subset Y of the variables is consistent iff it satisfies all the constraints c such that $X_c \subset Y$. A *solution* is a consistent assignment of values to all the CSP variables. A CSP is consistent iff it has at least one solution. Formally, the decision problem corresponding to a finite CSP is defined as follows:

Definition 9 (Finite CSP) *Given a CSP defined over a set X of variables with finite domains, with a set C of constraints, is the CSP consistent?*

The finite CSP decision problem has been shown to be NP-complete by reduction from SAT [35].

Theorem 5.2 *Finite CSP is NP-complete.*

5.2.2 Polynomial hierarchy

In this section we shall be dealing with problems which are supposedly computationally harder than the class of NP-complete problems. The material in this section is from [20, 44]. The class of NP-complete problems simply require a non-deterministic polynomial time decision procedure to solve the NP-complete problem. The class of problems which shall be studied in this chapter are more complex than the NP-complete problems, because a simple non-deterministic polynomial time decision procedure cannot solve them.

P^{NP} class of problems

Consider a special machine of the following nature: We assume the availability of an oracle which answers the SAT decision problem in constant time. We also make use of another machine which calls the SAT oracle a polynomial number of times, and then returns an answer. Clearly this is a machine which gives an answer after a polynomial number of queries to the SAT oracle. Such class of problems which can be answered by a polynomial time Turing machine with a SAT oracle, is termed as P^{SAT}. Since SAT is NP-complete, any NP-complete problem can be used instead of SAT. P^{SAT} can be written as P^{NP}, because any NP-complete problem can be used in place of SAT. Extending the definition of this class, FP^{NP} is the class of all functions from strings to strings that can be computed by a polynomial-time Turing machine with a SAT oracle. Not all problems in P^{NP} can be in NP, because for some problems in P^{NP} a "guess" cannot be verified by a non-deterministic Turing machine in polynomial time. Moreover, since a polynomial number of calls to the SAT oracle are required, any problem in P^{NP} is harder then any problem in NP. This is reflected by the polynomial hierarchy explained below.

The hierarchy

The class P^{NP} represents the first in the series of classes in the polynomial hierarchy higher than NP. The hierarchy of increasingly complex classes is defined as below:

$$\Sigma_0 = \Pi_0 = \Delta_0 = \mathrm{P}$$

and for all $k > 0$:

$$\begin{aligned} \Delta_{k+1} &= \mathrm{P}^{\Sigma_k} \\ \Sigma_{k+1} &= \mathrm{N}P^{\Sigma_k} \\ \Pi_{k+1} &= co-\Sigma_{k+1}. \end{aligned}$$

Based on this hierarchy, the classes at the first level are Π_1 = co-NP, Σ_1 = NP and Δ_1 = P. The class P^{NP} falls in the second level in the hierarchy as $\Delta_2 = P^{NP}$. The polynomial hierarchy is based on the general assumption that P $\neq$ NP, as it would collapse if P = NP.

The class $P^{NP[logn]}$

Sometimes a problem does not require the full power of P^{NP} in order to return an answer to a query. These problems require a much weaker class to describe the problem. The difference comes in the number of queries made to the SAT oracle in order to get the problem output. The number of SAT queries in such problems is in the order of $O(log\ n)$ for an input n. $P^{NP[log\ n]}$ refers to this class of problems which are decided by a constant-time oracle machine which for an input of size x asks a total of $O(log\ x)$ queries. $FP^{NP[log\ n]}$ refers to the corresponding class of functions.

5.2.3 Max-SAT variants and their complexity classes

The maximal satisfaction problem (max-SAT) and its weighted variant max-weighted SAT, both have been shown to be harder than NP in [44, 32]. Before we give the complexity result of these problems, we give the definitions of the corresponding SAT problems.

Definition 10 (Max-weighted SAT) *Given a set of clauses (in conjunctive normal form) each with an integer weight, find a truth assignment that satisfies a set of clauses with the most total weight.*

Definition 11 (Max-SAT) *Given a set of clauses (in conjunctive normal form), find a truth assignment that satisfies a maximum number of clauses.*

Here we state a result from [32, 44] concerning the complexity of max-weighted SAT. a

Theorem 5.3 *[44, 32] Max-weighted SAT is FP^{NP}-complete.*

We can reduce max-weighted SAT to the problems in over-constrained systems and hence show that the problems in over-constrained systems are computationally harder than NP, and that they require a higher complexity class to be characterized.

Max-SAT, which is the unweighted version of the max-weighted SAT problem, does not require the full power of FP^{NP} in order to return a answer to a query. In fact the following result from [44] captures this weaker class of max-SAT precisely.

Theorem 5.4 *[44, 32] Max-SAT is $FP^{NP[logn]}$-complete.*

This lower class of max-SAT is due to the $O(log\ x)$ number of calls to a SAT oracle for an input x.

5.2.4 Complexity of various types of over-constrained problems

In this section, we study the complexity of two types of over-constrained problems - max-weighted CSP, and max-CSP.

Max-weighted CSP

Recall that a max-weighted CSP P involves a set $X = \{x_1, x_2, \ldots, x_n\}$ of n variables, a set C of constraints $\{c_1, c_2, \ldots, c_m\}$, where each variable $x_i \in X$ takes a value from its domain d_i, and each constraint $c_i \in C$ is defined as a relation on a subset X_c of the set of variables X, and associated with each constraint $c_i \in C$, is a numerical value w_i, w_i being a real number. An assignment A of values to variables in X, *satisfies* a constraint c iff the tuple formed by X_c in A is satisfied by the intensional function associated with c. Let $w(A) = \sum\{w_i \mid A\ satisfies\ c_i\}$. Let W_P represent the set of all possible assignments A, assigning values to all variables in X. Then the *solution* set S of the problem is defined as

$$S = \{A' \in W_P \mid w(A') = max\{w(A) \mid A \in W_P\}\}$$

Any member of such a solution set S is a solution to the max-weighted CSP. We state and prove the theorem about the complexity class of max-weighted CSP.

Theorem 5.5 *Max-weighted CSP is FP^{NP}-complete.*

Proof. We prove this in two steps:

1. We first reduce max-weighted SAT to max-weighted CSP. Consider a max-weighted SAT instance C. Let C be defined over a set of boolean variables X. The problem C consists of a set of clauses, each of which is a disjunction of literals. Corresponding to C, let us construct a max-weighted CSP C'. Let C' be defined over

the same set of variables X. Since any variable x in X is boolean, let x take any of the two values T or F, standing for *true* or *false* respectively. So C' consists of the same set of variables as in C, each variable with a fixed domain of $\{T, F\}$. We then construct the set of constraints in C'. Corresponding to each clause c in C of the form $(L_1 \vee L_2 \vee \dots L_k)$, let X_c represent the set of variables which appear in c. The only assignment A_c which falsifies c, is the one which assigns values to variables in X_c such that all literals in c are false. Corresponding to a clause c in C, we create a constraint c' in C' over the variables in X_c, which intensionally allows any assignment to variables in X_c other than A_c. The time required to construct constraint c' is linear in the number of literals in c. Further, whenever c is satisfied by any assignment in C, the same assignment also satisfies c'. Likewise, when an assignment does not satisfy c, it does not satisfy c' either. E.g. If $c = (X_1 \vee \overline{X_2})$ then the corresponding constraint c' can be defined an intensional relation over $\{X_1, X_2\}$ as the one allowing all tuples except the tuple $\{F, T\}$. Further, we assign the weight of c to c'. We translate all the clauses in original SAT problem C to their corresponding constraints in C'.

Based on this construction, we see that if a truth assignment violates a specific clause c, then the constraint c' corresponding to c is not satisfied. Hence based on this one-to-one correspondence, we can conclude that if a truth assignment A satisfies a maximal weighted subset of clauses in C, the same assignment in the CSP C' satisfies a maximal weighted subset consisting of the constraints corresponding to clauses satisfied by A. Similarly if some assignment A is not satisfying a maximal weighted subset of clauses, there is another assignment A' which satisfies a subset of clauses with a greater combined weight. Accordingly the assignment of values in the CSP corresponding to A does not satisfy a maximal weighted subset of constraints because of the assignment corresponding to A'.

So max-weighted SAT is polynomially reducible to max-weighted CSP.

2. For the second part of the proof we need to show that max-weighted CSP is in FP^{NP}. For a given choice of an integer N, we ask whether there is a sequence

of non-deterministic choices which lead to an assignment with a total weight greater than N. This question can be answered in NP. N_{max}, the maximum sum of weights of satisfied clauses, can be derived by asking the above question for various integers and converging by binary search. The variable assignment that achieves N_{max} is then obtained by assigning values to variables one-by-one. The number of questions (each in NP) is polynomial in size of an instance and hence the problem max-weighted CSP is in FP^{NP}.

Based on the above two steps, max-weighted CSP is FP^{NP}-complete.

Hence the proof. □

Max-CSP

Recall that max-CSP is a simplification of max-weighted CSP, where all constraints are of weight 1. Here we seek a solution satisfying the maximum number of constraints and all constraints are of equal weight, namely 1. A solution is defined as an assignment to the variables satisfying the maximum number of clauses.

Based on this observation it is obvious that max-CSP is the equivalent of max-SAT and shares the complexity class of max-SAT. We state the following result:

Theorem 5.6 *Max-CSP is $FP^{NP[\log n]}$-complete.*

Proof. The proof as in max-weighted CSP case proceeds in two steps:

1. We reduce any instance of max-SAT problem C polynomially to a max-CSP instance C' by following the same translation as used for the max-weighted CSP case except for the lack of weights. The translation once again preserves the one-to-one correspondence between an instance of a max-SAT problem and an instance of the translated max-CSP. If any truth assignment in max-SAT problem C does not satisfy a specific clause c, the same assignment in the max-CSP translation C' does not satisfy the constraint corresponding to the clause c. Hence any assignment A which is a maximal solution of max-SAT is also a maximal solution to the max-CSP, satisfying the set of constraints in C' corresponding to the clauses in C satisfied by A.

Conversely any assignment A which is not a maximal solution of C is not a maximal solution of C', because of the presence of an assignment which satisfies a greater number of clauses/constraints.

2. To show that max-CSP is in $FP^{NP[log\ n]}$, we know that max-CSP can be solved in $log_2(N)$ steps, N being the number of constraints and each step being in NP. For a given choice of an integer K, we ask whether there is a sequence of non-deterministic choices which lead to an assignment which satisfies a greater number of constraints than K. This question can be answered in NP. By a binary search starting from $K = N$(the number of constraints), the maximum number of satisfied constraints can be determined. Then by construction, the assignment satisfying the maximum number of constraints can be identified. The binary search process takes a total of $log_2(N)$ steps, N being the number of constraints.

Hence max-CSP is in $FP^{NP[logn]}$.

Based on the two steps, max-CSP is $FP^{NP[log\ n]}$-complete.

Hence the proof. □

The theorems about the complexity of the over-constrained problems, provide an intuition into the nature of the notion of priority considered in the case of over-constrained problems. Any form of priority structure which involves the comparison of a solution with another solution on a global scale, pushes the complexity to the second level in the polynomial hierarchy.

5.2.5 Discussion of complexity results

The results discussed in this section highlight the fundamental difference between over-constrained problems and CSPs. While CSPs strive to either find a solution if one exists or to stop of no solution exists, over-constrained problems try to overcome the incomplete information by identifying the *best* partial solution. To identify the optimal partial solution, it is often necessary to compare among different partial solutions.

In this section, the commonly used semantics of over-constrained problems have been shown to fall in the second level of the polynomial hierarchy. The reason for their falling into the second level of complexity lies in the fact that we are comparing between solutions to find the optimal solution thereby necessitating the generation of a multiple number of "guesses" for the comparison. A max-weighted CSP needs to go through a polynomial number of queries to an NP oracle, while max-CSP needs a log-polynomial number of queries to an NP oracle.

An impact of this also occurs on the algorithms necessary to solve the max-CSP and max-weighted CSP problems. It is shown in Section 5.4 that backtracking based algorithms used for CSPs cannot be used to solve max-CSP or max-weighted CSPs because of the abovementioned differences in complexity. Backtracking algorithms would end up with no solution to any over-constrained problem because there is no provision for bookkeeping of an optimal partial solution in these complete algorithms based on backtracking.

5.3 Non-binary versus binary representations of over-constrained problems

In the recent past there has been considerable interest in non-binary representations of CSPs as opposed to binary ones, for they are more expressive and general in nature. In the past research concentrated on efforts to optimize and improve the performance of binary constraint reasoners implicitly based upon the assumption that any non-binary CSP could be transformed into an equivalent binary CSP in polynomial time. Only in the recent past has there been some work [3] in comparing the relative effectiveness of working directly with non-binary CSP representations versus working with translated binary representations.

If we use binary representations, we need to measure the overhead of translation of non-binary constraints to binary constraints, and also verify the preservation of correctness of the problem in going from one representation to the other. The initial observations from the study in [3], suggests that the effectiveness of algorithms for the translated binary CSPs depends upon many factors like the number of tuples satisfied by the constraints etc. The study also highlights the need to modify the algorithms based on backtracking and its variants considerably to be applicable to the translations.

In finite CSPs, a solution satisfies all the constraints. In contrast, in max-CSP there is no solution satisfying all the constraints in the problem. In such a case, the notion of solution is relaxed and it is defined as an assignment of values to variables that satisfies a maximal number of constraints. Research till date in max-CSP algorithms has concentrated on binary representations [15].

In this section we will try to answer questions pertaining to preservation of semantics in translation from non-binary representations of max-CSP to binary max-CSP representations.

5.3.1 Two methods of conversion from non-binary to binary representations

There are mainly two methods for translating non-binary constraint satisfaction problems to binary constraint satisfaction problems - *dual graph* method [12] and *hidden variable* method [50, 3].

Dual Graph method

In this translation method, the constraints of the original problem become variables (called C-variables) in the new representation. The domain of each C-variable is exactly the set of tuples that satisfy the original constraint, each tuple representing a possible value for the C-variable. A binary constraint between two C-variables exists iff the original constraints share at least one variable. The binary constraints dictate explicitly that the shared variables between the two C-variables, carry the same values.

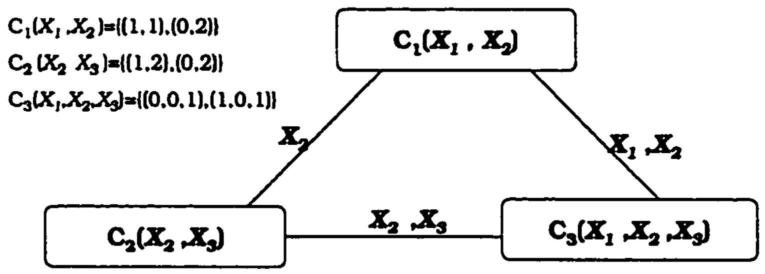


Figure 5.1: An example of the dual graph method

Consider the problem P with three constraints C_1, C_2 and C_3 on three variables, X_1, X_2 and X_3 respectively. Let C_1 be the constraint on $\{X_1, X_2\}$ consisting of the tuples $\{(0,1),(1,1)\}$, C_2 be the constraint on $\{X_2, X_3\}$ with the tuples $\{(1,2),(0,2)\}$ and C_3 be the constraint on $\{X_1, X_2, X_3\}$ with the tuples $\{(0,0,1),(1,0,1)\}$. The Dual graph for this constraint satisfaction problem P is shown in Figure 5.1. Each of the constraints C_1, C_2 and C_3 now becomes a C-variable. The domain of each C-variable is precisely the set of tuples in the corresponding constraint. The C-variable C_1 takes two values 1 and 2 representing the tuples $(0,1)$ and $(1,1)$ respectively. Likewise, C_2 and C_3 take two values representing the two tuples in each of the constraints. Coming to the constraints, let DG_1 represent the constraint between C_1 and C_2. $DG_1(C_1, C_2)$ is written as $C_1[X_2] = C_2[X_2]$, specifying that C_1 and C_2 should agree on X_2. $DG_2(C_1, C_3)$, the constraint between C_1 and C_3, dictates that C_1 and C_3 agree on X_1 and X_2, and $DG_3(C_2, C_3)$, the constraint between C_2 and C_3, dictates that C_2 and C_3 agree on X_2 and X_3.

Hidden variable method

In the hidden variable representation, the set of variables includes all of the variables of the original problem (with no changes to their domains) plus a new set of ***hidden*** or h-variables. For each constraint C_i in the original problem an h-variable H_i is added. The domain of H_i consists of a unique identifier for every tuple in the original C_i. The new representation contains only binary constraints, and these constraints are constructed by the following method: For every h-variable H_i we impose a binary constraint between H_i and each of the variables involved in C_i. Suppose H_i and X_k are thus constrained. Every value of H_i corresponds to a tuple of values for the variables involved in C_i, defining a unique value for X_k. The binary constraint between H_i and X_k consists of a unique value for X_k for every value of H_i.

Consider the same CSP as in Figure 5.1. The hidden variable representation is shown in Figure 5.2. Here the constraints C_1, C_2 and C_3 each spawn off a hidden variable each H_1, H_2 and H_3 respectively. Each of the hidden variables H_1, H_2 and H_3 each take two values $\{1,2\}$, $\{1,2\}$ and $\{1,2\}$ respectively because there are two tuples in each constraint. A value of 1 for H_1 stands for the first tuple of H_1 ($\{X_1 = 0, X_2 = 1\}$), and likewise for other values of hidden variables. Next a constraint is added between any hidden variable and the variable involved in the corresponding constraint. This constraint contains the pairs formed as follows: If a value v_j of X_j occurs in tuple t of hidden variable H_1, then a pair (v_j, t) is part of the constraint. E.g., here (0,1), (1,2) form the constraint between X_1 and H_3 as the first tuple ($H_3 = 1$) in C_3 has $X_1 = 0$ and the second tuple ($H_3 = 2$) has $X_1 = 1$.

5.3.2 Semantics of max-CSP under translation with both methods

In case of max-CSP, no solution satisfying all the constraints can be found. The semantics of the problem is specified in terms of a solution satisfying a maximal number of constraints in the problem. No studies have been undertaken to study

TRANSLATED CSP

Variables: $X_1, X_2, X_3, H_1, H_2, H_3$

Domains: $X_1(0,1), X_2(0,1,2), X_3(2), H_1(1,2), H_2(1,2), H_3(1,2)$.

Constraints:

$C_1(X_1, H_2) = \{(0,1),(1,2)\}$ $C_2(X_2, H_1) = \{(1,2),(2,1)\}$
$C_3(X_1, H_3) = \{(0,1),(1,2)\}$ $C_4(X_2, H_2) = \{(1,1),(0,2)\}$
$C_5(X_2, H_3) = \{(0,1),(0,2)\}$ $C_6(X_3, H_2) = \{(2,1),(2,2)\}$
$C_7(X_3, H_3) = \{(1,1),(1,2)\}$

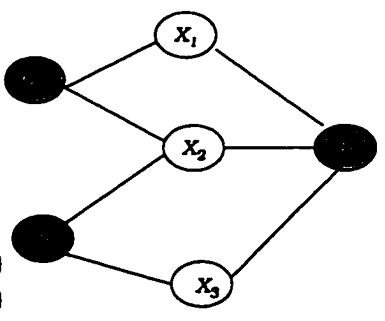


Figure 5.2: An example of the hidden variable method

the relative merits and demerits of translations between non-binary and binary forms of constraints in the case of max-CSP. In the following subsections we examine the soundness and completeness of the *dual graph* and the hidden variable methods of translation from non-binary to binary representations of max-CSP.

The questions which need to be answered are two fold:

Soundness Is a solution to the translated binary max-CSP a solution to the original non-binary max-CSP (in either of the two translation methods)?

Completeness Is a solution to max-CSP in the original non-binary representation still a solution in the translated binary max-CSP (with hidden variables or dual graph)?

Translation by the dual graph method

In translation from the non-binary representation of a max-CSP to the binary representation using the dual graph method, the role of constraints and variables is interchanged. In the translated binary max-CSP, the constraints in the original representation become C-variables. Constraints in the translated representation are introduced between C-variables iff the original constraints corresponding to the C-variables share a variable. The constraints are semantically enforced to make sure that any two C-variables, whose corresponding constraints in the original CSP share a variable, agree on the values of all the variables common to both constraints.

The semantics of the solution of a binary maximal constraint satisfaction problem dictates that it return an assignment of values that satisfies maximum number of constraints.

Now we show that the dual graph method is not sound in preservation of semantics of max-CSP.

Theorem 5.7 *The dual graph method of translation from a non-binary max-CSP to its binary equivalent is unsound.*

Proof. We show the unsoundness of translation by the counterexample in Figure 5.3. The original CSP has four constraints $C_1(X_1, X_2) = \{(1,1),(2,0)\}$, $C_2(X_1, X_6) = \{(1,8),(1,2)\}$, $C_3(X_5, X_6) = \{(3,2)\}$ and $C_4(X_2, X_5, X_3) = \{(0,9,2),(2,3,1)\}$.

In the dual graph, there are four C-variables C_1, C_2, C_3 and C_4. The domain of C_1 is $\{1,2\}$ representing the tuples $\{(1,1),(2,0)\}$ respectively. Similarly the domains of C_2, C_3, and C_4 are $\{1,2\}$,$\{1\}$, and $\{1,2\}$ respectively representing the corresponding tuples in the original constraints. The dual graph CSP consists of four constraints $DG_1(C_1, C_2)$, $DG_2(C_1, C_4)$, $DG_3(C_2, C_3)$, and $DG_4(C_3, C_4)$. $DG_1(C_1, C_2)$ states that $C_1[X_1] = C_2[X_1]$, DG_2 states that $C_1[X_2] = C_4[X_2]$, DG_3 states that $C_3[X_6] = C_2[X_6]$, and DG_4 states that $C_3[X_5] = C_4[X_5]$.

Now, consider the assignment A (including values assigned to C-variables), namely $\{C_1 = 1, C_2 = 2, C_3 = 1, C_4 = 2,\ X_1 = 1, X_2 = 0, X_3 = 1,\ X_5 = 3, X_6 = 2\}$. This assignment satisfies three constraints,namely DG_1, DG_3, and DG_4, in the translated dual CSP. This is the maximum satisfied by any assignment. Hence A is a solution

$C_1(X_1,X_2) = \{(1,1),(2,0)\}$

$C_2(X_1,X_6) = \{(1,8),(1,2)\}$

$C_3(X_5,X_6) = \{(3,2)\}$

$C_4(X_2,X_5,X_3) = \{(0,9,2),(2,3,1)\}$

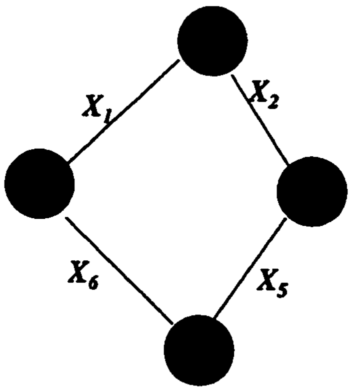


Figure 5.3: Dual graph method - counterexample for soundness

to the binary translated max-CSP. The implicit original assignment corresponding to A in the original CSP is $\{X_1 = 1, X_2 = 0, X_3 = 1, X_5 = 3, X_6 = 2\}$. This satisfies only two constraints in the original CSP, which has a solution $\{X_1 = 1, X_2 = 1, X_3 = 1, X_5 = 3, X_6 = 2\}$ satisfying three constraints. Hence the implicit original assignment corresponding to A is not a solution to the original max-CSP though A is a solution of the translated max-CSP. Hence we have a solution to the translated max-CSP whose original corresponding assignment is not a solution in the original max-CSP.

We have shown that the translation method using dual graph is not sound for max-CSP.

Hence the proof. □

We now show that the dual graph method of translation is not complete either.

Theorem 5.8 *The dual graph method of translation from a non-binary max-CSP to its binary equivalent is not complete.*

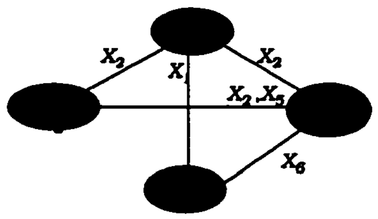
Example:

$C_1(X_1, X_2, X_3)=\{(1,2,0)\}$

$C_2(X_2, X_4, X_5)=\{(2,0,0)\}$

$C_3(X_2, X_5, X_6)=\{(2,1,3)\}$

$C_4(X_1, X_7, X_6)=\{(1,8,3),(1,7,2)\}$



Maximal CSP: **Solution satisfying MAXIMUM number of constraints**

$\{X_1 = 1, X_2 = 2, X_3 = 0, X_4 = 0, X_5 = 0, X_6 = 2, X_7 = 7\}$ satisfies 3 constraints,

$\{X_1 = 1, X_2 = 2, X_3 = 0, X_4 = 0, X_5 = 0, X_6 = 3, X_7 = 8\}$ satisfies 4 constraints.

Figure 5.4: Dual graph method - counterexample for completeness

Proof. We show that the dual graph method is not complete by the counterexample in Figure 5.4. The max-CSP in Figure 5.4 contains four constraints $C_1(X_1, X_2, X_3)=\{(1,2,0)\}$, $C_2(X_2, X_4, X_5) = \{(2,0,0)\}$, $C_3(X_2, X_5, X_6) = \{(2,1,3)\}$, and $C_4(X_1, X_7, X_6) = \{(1,8,3),(1,7,2)\}$.

In the dual graph, there are four C-variables C_1, C_2, C_3 and C_4. The domain of C_1 is $\{1\}$ representing the lone tuple $\{(1,2,0)\}$. Similarly the domains of C_2, C_3, and C_4 are $\{1\}$,$\{1\}$, and $\{1,2\}$ respectively representing the corresponding tuples in the original constraints. The dual graph CSP consists of five constraints DG_1, DG_2, DG_3, DG_4, and DG_5 respectively. $DG_1(C_1, C_2)$ states that $C_1[X_2] = C_2[X_2]$, $DG_2(C_1, C_3)$ states that $C_1[X_2] = C_3[X_2]$, $DG_3(C_1, C_4)$ states that $C_1[X_1] = C_4[X_1]$, $DG_4(C_2, C_3)$ states that $C_2[X_2, X_5] = C_3[X_2, X_5]$, and $DG_5(C_3, C_4)$ states that $C_3[X_6] = C_4[X_6]$.

In the original max-CSP, the assignment A, namely $\{X_1 = 1, X_2 = 2, X_3 = 0, X_4 = 0,$ $X_5 = 0, X_6 = 2, X_7 = 7\}$, is a solution because it satisfies maximum number of constraints, namely 3. But in the translated max-CSP, when the same assignment is supplemented with the values of the C-variables $C_1 = 1$ (stands for the first tuple of C_1 constraint), $C_2 = 1, C_3 = 1$, and $C_4 = 2$, the corresponding translated assignment of A satisfies only 3 constraints in the translated max-CSP, namely DG_1, DG_2, and DG_3. But there exists another assignment $\{X_1 = 1, X_2 = 2, X_3 = 0, X_4 = 0, X_5 = 0, X_6 = 3, X_7 = 8, C_1 = 1, C_2 = 1, C_3 = 1, C_4 = 1\}$ which satisfies 4 constraints DG_1, DG_2, DG_3, and DG_5. Hence the translation of A is not the solution to the translated binary max-CSP. So a solution in the original non-binary representation is no longer a solution in the translated binary representation. The dual graph method of translation for max-CSP is thus not complete.

Hence the proof. □

We have shown that the dual graph method of translation of non-binary max-CSP to binary max-CSP problem is neither sound nor complete.

Translation by hidden variable method

In this section, we shall explore if the semantics of max-CSP is preserved when an original non-binary max-CSP problem is translated to the binary representation using the hidden variable method.

We show by counterexamples that the translation method using hidden variables is neither sound nor complete. We first prove the unsoundness of the translation using hidden variables.

Theorem 5.9 *The translation of a non-binary max-CSP to its equivalent binary max-CSP by the hidden variable method is unsound.*

Proof. We shall prove the unsoundness by a counterexample. Consider the max-CSP as shown in Figure 5.5.

The max-CSP in Figure 5.5 has 4 constraints $C_1(X_1, X_2) = \{(1,1),(2,0)\}$, $C_4(X_2, X_5, X_3) = \{(0,9,2),(2,3,0)\}$, $C_3(X_5, X_6, X_3) = \{(3,2,1)\}$ and $C_2(X_1, X_6) = \{(1,8),(1,2)\}$.

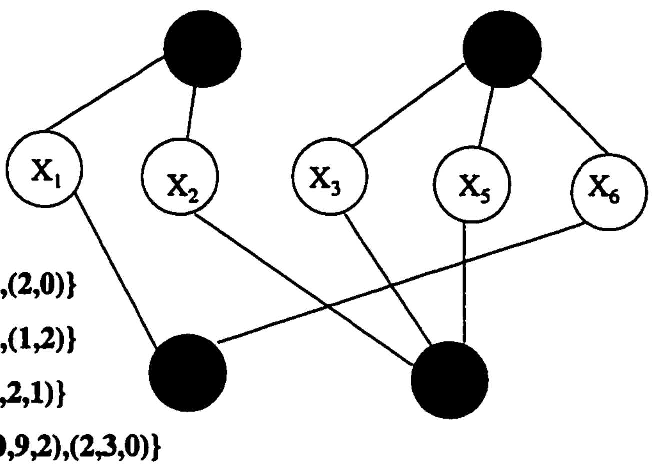

X_1
X_2
X_3
X_5
X_6
$C_1(X_1,X_2) = \{(1,1),(2,0)\}$
$C_2(X_1,X_6) = \{(1,8),(1,2)\}$
$C_3(X_5,X_6,X_3) = \{(3,2,1)\}$
$C_4(X_2,X_5,X_3) = \{(0,9,2),(2,3,0)\}$

Figure 5.5: Hidden variable method - counterexample for soundness

Consider the assignment A, namely $\{H_1 = 1, H_2 = 2, H_3 = 1, H_4 = 2, X_1 = 1, X_2 = 1, X_3 = 0, X_5 = 3, X_6 = 2\}$. A is a solution to the translated max-CSP with hidden variables, as it satisfies 8 constraints, the highest by any assignment. The original assignment corresponding to A in the original max-CSP is $\{X_1 = 1, X_2 = 1, X_3 = 0, X_5 = 3, X_6 = 2\}$. This satisfies two constraints while there exists an assignment $\{X_1 = 1, X_2 = 1, X_3 = 1, X_5 = 3, X_6 = 2\}$ which satisfies three constraints. Hence the original assignment corresponding to A is not a solution to the original max-CSP.

So we have a solution to the translated max-CSP whose corresponding assignment in the original max-CSP is not a solution.

Hence the hidden variable method of translation is unsound.

Hence the proof. □

We now show that the hidden variable method is not complete either.

Theorem 5.10 *The translation of a non-binary max-CSP to its equivalent binary*

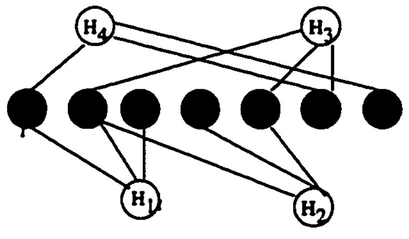
Example:

$C_1(X_1, X_2, X_3)=\{(1,2,0)\}$

$C_2(X_2, X_4, X_5)=\{(2,0,0)\}$

$C_3(X_2, X_5, X_6)=\{(2,1,3)\}$

$C_4(X_1, X_7, X_6)=\{(1,8,3),(1,7,2)\}$



Maximal CSP: **Solution satisfying MAXIMUM number of constraints**

$\{X_1 = 1, X_2 = 2, X_3 = 0, X_4 = 0, X_5 = 0, X_6 = 2, X_7 = 7\}$ satisfies 10 constraints.

$\{X_1 = 1, X_2 = 2, X_3 = 0, X_4 = 0, X_5 = 0, X_6 = 3, X_7 = 8\}$ satisfies 11 constraints.

Figure 5.6: Hidden variable method - counterexample for completeness

max-CSP by the hidden variable method is not complete.

Proof. Consider the same max-CSP as shown in Figure 5.4. The hidden variable translation of this max-CSP is shown in 5.6.

Here the original max-CSP admitted the variable assignment A, namely $\{X_1 = 1, X_2 = 2,\ X_3 = 0, X_4 = 0,\ X_5 = 0, X_6 = 2,\ X_7 = 7\}$ as a solution to the max-CSP, because it satisfies three constraints C_1, C_2 and C_4 in the original problem which is the maximum number of constraints possible to be satisfied as all four constraints C_1, C_2, C_3 and C_4 cannot be satisfied at the same time. But in the translated binary constraint representation with the hidden variables H_1, H_2, H_3 and H_4, the associated tuple $\{X_1 = 1, X_2 = 2, X_3 = 0, X_4 = 0,\ X_5 = 0,\ X_6 = 2, X_7 = 7,\ H_1 = 1, H_2 = 1, H_4 = 2, H_3 = 1\}$ satisfies only 10 constraints. But there exists a solution $\{X_1 = 1, X_2 = 2,\ X_3 = 0, X_4 = 0,\ X_5 = 0, X_6 = 3,\ X_7 = 8,\ H_1 = 1, H_2 = 1,\ H_4 = 1, H_3 = 1\}$ to the translated binary max-CSP which satisfies 11 constraints and hence is the solution to the translated problem. So a solution in the original max-CSP is no longer a solution in the translated max-CSP.

Hence the translation method using hidden variables is not complete. Hence the proof. □

So the hidden variable based method of translation from non-binary to binary representation of max-CSP is neither sound nor complete.

5.3.3 Discussion and conclusions

The results in this section lead us mainly to the following two conclusions:

1. In the absence of sound translation mechanisms for translation from non-binary over-constrained problems to binary over-constrained problems, the translation of non-binary over-constrained problems should be accompanied with alternative mechanisms to compute equivalent solutions.

2. The argument vis-a-vis conducting research extensively into binary representations of maximal and other over-constrained problems, based on the existence of a sound translation from non-binary to binary over-constrained problems falls apart. The research into binary over-constrained problems falls into a very narrow domain of applicability given the succinct form of representation in binary case.

In the following section, we study the solution methods for max-CSP and critically analyze them.

5.4 Extended analysis of intelligent branch and bound algorithms for max-CSP

In this section we shall critically examine the existing solving methods for max-CSP, and improve them.

In the first part of this section, the background material concerning application of branch and bound methods and its variants to the max-CSP is presented [16].

The material in the initial part of this section is cited from [16]. The methods of depth-first branch and bound (DFBB) and backjumping for max-CSP were discussed in [16]. We extend the methods to incorporate conflict-direct backjumping (CBJBB) and in the end critically analyze the different branch and bound variants theoretically.

5.4.1 Depth-first branch and bound

In naive backtracking for constraint satisfaction problems(CSPs), a partial assignment of values to the variables is consistently expanded till a dead-end is encountered. A dead-end refers to a situation when any value assigned to the latest variable does not lead to a solution. In such a case, backtracking occurs and the next possible value for the immediately previous variable is tried till a value assignment to all variables is reached which satisfies all the constraints in the CSP. Backtracking clearly indicates that we need to go through an exponential number of nodes in the worst case to solve the CSP. This prompts the possibility of extension of backtracking based algorithms for the problem of computing one solution to the max-CSP. It is clear from the definition of the max-CSP, that naive backtracking will ultimately end for a max-CSP without returning any solution because the CSP is known to be insoluble. Thus backtracking cannot be used directly for the maximal constraint satisfaction problem (max-CSP) [16].

In [16], it was shown that the natural analogue of the naive backtracking algorithm for max-CSP was the depth-first branch and bound (DFBB) algorithm. In DFBB for max-CSP a partial solution is expanded along the way. But unlike the backtracking algorithm, a partial solution is expanded even on encountering of an inconsistent partial solution. A counter N is kept for the current best solution during the solving process, and a counter C keeps track of the number of constraints violated by the current assignment of values. We may begin with a large N, close to infinity. Then on encountering the first assignment of values to all variables, this figure N is updated by C, the number of constraints updated by the current solution.

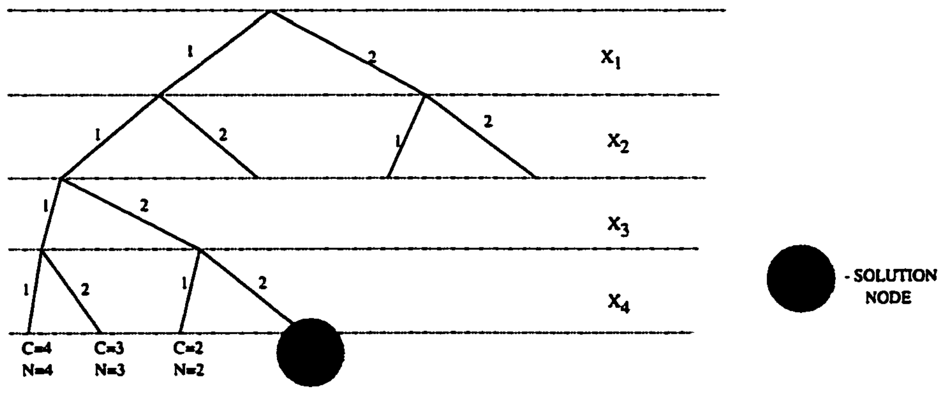
Then search backtracks and proceeds with a different value for the preceding variable and C is recomputed. On encountering a partial assignment of values during the search process which violates $\geq N$ constraints, search along that path is cut short and backtracking occurs to the next value of the preceding variable. This is because any further assignment along the same path cannot lead to a better solution. This is the bounding process of the branch and bound algorithms.

Consider the max-CSP shown in Figure 5.7. Here we use three variables C and N and S which represent the number of constraints violated by the current assignment of values, the number of constraints violated by the current best solution and the current solution respectively.

The algorithm begins with the value N = infinity. Then we proceed along $X_1 = 1, X_2 = 1$ till $X_3 = 1$ without any problem. But $X_3 = 1$ is inconsistent with $X_1 = 1$. In naive backtracking this would have resulted in a backtrack to $X_3 = 2$. But here we proceed further and try $X_4 = 1$. Here we have a possible solution which violates 4 constraints. Hence N is now updated to 4 and S (the current best solution) is $\{X_1 = 1, X_2 = 1, X_3 = 1, X_4 = 1\}$. Next we backtrack and reach $X_4 = 2$ and S now becomes $\{X_1 = 1, X_2 = 1, X_3 = 1, X_4 = 2\}$ as the number of inconsistencies in this solution is 3 which is less than 4, the current value of N, and N is updated to 3.

Further backtrack to $X_3 = 2, X_4 = 1$ gives an even better solution $S = \{X_1 = 1, X_2 = 1, X_3 = 2, X_4 = 1\}$, which gives $N = 2$. In the next branch, this result is further improved with $S =\{X_1 = 1, X_2 = 1, X_3 = 2, X_4 = 2\}$ giving $N = 1$. So this assignment violates just one constraint namely constraint C_3, the constraint between X_4 and X_1.

Further backtrack leads us to $X_1 = 1, X_2 = 2$ which violates 1 constraint and thus any further expansion of this assignment can't lead to any better solution than the current best solution S. So we bound the search here and backtrack to the next level $X_1 = 2$, and then proceed to $X_2 = 1$. Here again the number of constraints violated is greater than or equal to the current N, which is 1. So we bound it here and face a



CONSTRAINTS:

$C_1(X_1, X_2) = \{(1,1)\}$

$C_2(X_3, X_1) = \{(2,1)\}$

$C_3(X_4, X_1) = \{(2,2),(1,2)\}$

$C_4(X_4, X_2) = \{(2,2),(1,2),(2,1)\}$

Figure 5.7: Depth first branch and bound (BB) for max-CSP

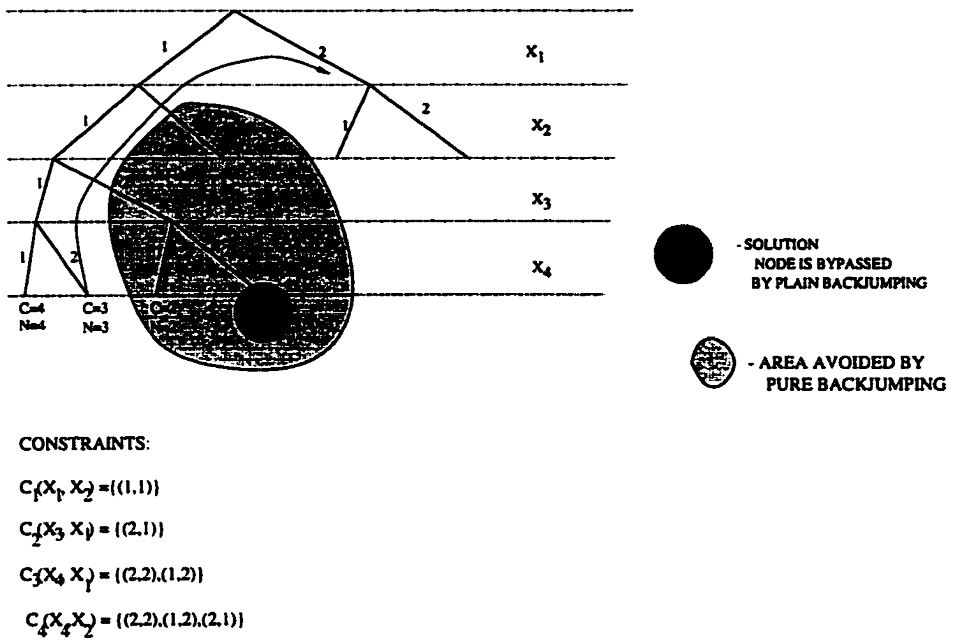
similar condition at $X_1 = 2, X_2 = 2$.

This concludes the search process and we get the solution S=$\{X_1 = 1, X_2 = 1, X_3 = 2, X_4 = 2\}$ which violates $N = 1$ constraints.

5.4.2 Backjumping

Depth-first branch and bound [16] has been formulated as the equivalent of the naive backtracking algorithm for max-CSP. In backjumping proposed by Gashnig [21], the naive backtracking method for CSPs was modified with a provision for some bookkeeping to achieve a significant reduction in the search space than the naive backtracking algorithm. Backjumping as applied to max-CSP shall be termed as BJBB and was presented in [16].

When backtracking occurs from a dead-end at a variable, the control passes to the chronologically previous variable. But it is possible that the variables responsible for the dead-end at that node are actually much higher up than the immediately previous



1
2
X_1
1
1
2
X_2
1
X_3
1
2
X_4
C=4
N=4
C=3
N=3
- SOLUTION NODE IS BYPASSED BY PLAIN BACKJUMPING
- AREA AVOIDED BY PURE BACKJUMPING
CONSTRAINTS:
$C_1(X_1, X_2) = \{(1,1)\}$
$C_2(X_3, X_1) = \{(2,1)\}$
$C_3(X_4, X_1) = \{(2,2),(1,2)\}$
$C_4(X_4, X_2) = \{(2,2),(1,2),(2,1)\}$

Figure 5.8: BJ for max-CSP

variable. Let X_j be the deepest variable responsible for creation of the dead-end at a node X_k, and let X_i be the variable immediately chronologically prior to X_k. In naive backtracking the control passes on to X_i after X_k. But since X_j is the deepest variable responsible for the dead-end at X_k, any exploration of the nodes between X_j and X_k, will not prevent the dead-end at X_k. So if instead of jumping to X_i, if we jump directly to X_j, skipping across all variables between X_j and X_i, we still preserve the solution and save a lot of search space too. Backjumping results in a significant reduction in search space which overshadows the slight overhead involved in the bookkeeping of the deepest variable in conflict with the current node X_k.

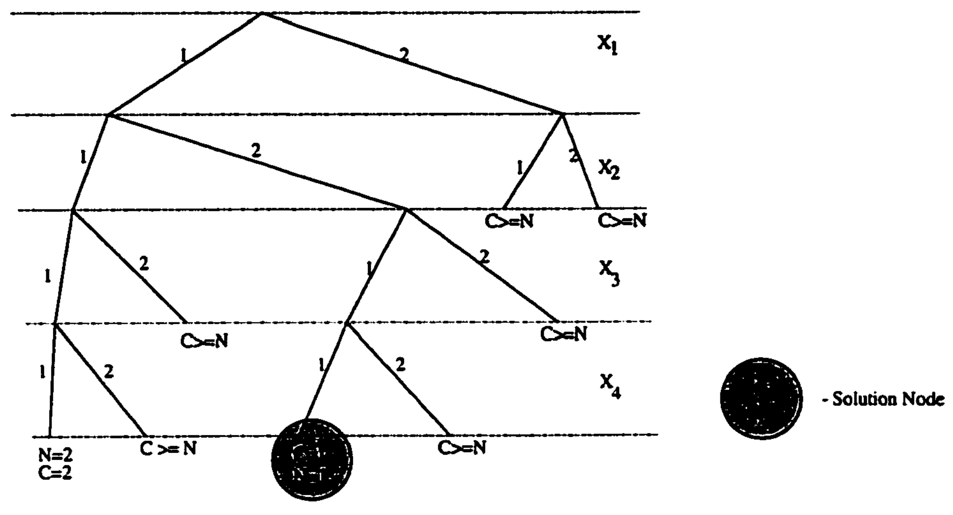
But this technique cannot be directly translated to the max-CSP case, and cannot be incorporated in depth-first branch and bound method in the same manner as is done in adding backjumping to backtracking in CSP [16]. Consider the example in Figure 5.8. This is the same as the example in Figure 5.7. Here we follow depth first branch and bound till we reach the node representing $\{X_1 = 1, X_2 = 1, X_3 = 1, X_4 = 2\}$. At this node we find that the node X_1 itself is in conflict with both values of X_4. So we can backjump from X_4 to X_1 directly instead of X_3 or X_2.

So we can backjump to $X_1 = 2$ directly. But in the process we avoid a significant portion of the search space, which is a considerable savings in the normal case. But from the figure it is also clear that the search space we are avoiding actually contains the maximal solution,namely, $\{X_1 = 1, X_2 = 1, X_3 = 2, X_4 = 2\}$. So backjumping algorithm for CSP when directly applied to the max-CSP leads to incorrect results. The phenomenon occurs because of the fact that $X_3 = 1$ was an inconsistent node in one branch while in the avoided part of search space $X_3 = 2$ was not an inconsistent node, and this lead to the possibility of a better solution along that path. Hence the correctness of backjumping is lost. The solution proposed in [16] to overcome this keeps track of the latest inconsistent node encountered in the current path, and backjump occurs to the later of the two variables, the backjump point or the deepest inconsistent variable encountered. This ensures that the correctness of the algorithm is not compromised even though the savings in search space due to backjumping may be lost in some situations. Consider the situation above. At $X_4 = 2$, we have to choose between the backjump point namely X_1, and the last inconsistent point, namely X_3. Since X_3 is deeper than X_1, backjump occurs to X_3 and we have no savings in space though we preserve the correctness of the algorithm.

5.4.3 Applicability of conflict-directed backjumping to max-CSP

It has been shown that in backjumping an extra variable is needed to keep track of the deepest inconsistent variable at any time, to decide the next backjump point. The backjump took place to the deeper of the two variables, the deepest variable in conflict with the current node or the deepest of the inconsistent nodes.

In this section we apply conflict-directed backjumping (CBJ) to max-CSP, the resulting algorithm termed as CBJBB. In contrast to backjumping, in conflict-directed backjumping, the backjump point is not just dependent upon the variable at the current level but also upon the variables at levels below. At each level of x_i an array



CONSTRAINTS:

$C_1(X_2, X_1) = \{(2,1),(1,2)\}$ $C_4(X_1,X_3) = \{(1,1)\}$

$C_2(X_2X_1) = \{(2,1)\}$

$C_3(X_3, X_4) = \{(1,1)\}$

Figure 5.9: Depth first BB for CBJ example

conflict-set, keeps track of all the past variables in conflict with the current level. Every time an inconsistency is encountered between x_i and some past variable, the past variable is added to the conflict set of x_i. In the event of all values of x_i being exhausted, the algorithm backjumps to the deepest variable x_h in the conflict-set of x_i. In the process of this backjump, the variables in the conflict-set of x_i (excluding x_h) are added to the conflict-set of x_h, because none of these valuations can lead to a successful solution. The bookkeeping in conflict-directed backjumping (CBJBB) is more complex than that in plain BJBB case.

We show that similar to BJBB, CBJBB algorithm cannot be applied as it is to the depth-first branch and bound algorithm. Consider the example in Figure 5.9. In Figure 5.9, the depth-first branch and bound traversal for the CSP is shown. The depth-first branch and bound algorithm traverses the search space and returns with the maximal solution $\{X_1 = 1, X_2 = 2, X_3 = 1, X_4 = 1\}$.

Now, consider the same example with conflict-directed backjumping(CBJBB) as shown in Figure 5.10. In CBJBB, at $X_4 = 1$ in the first pass at $\{X_1 = 1, X_2 = 1, X_3 = 1, X_4 = 1\}$ the conflict-set consists of the set $\{X_1\}$. Next at $\{X_1 = 1, X_2 = 1, X_3 = 1, X_4 = 2\}$, the conflict-set now becomes $\{X_1, X_3\}$ for X_4. So backtrack occurs to the deepest variable in the conflict-set namely $\{X_3\}$. At the same time the conflict-set of X_3 is changed to $\{X_1\}$, since that is the only remaining element in the conflict-set of X_1 other than X_3 itself. So the search proceeds along $\{X_1 = 1, X_2 = 1, X_3 = 2\}$. But at this point we reach a bound and now need to backtrack or backjump to the deepest variable in the conflict-set of X_3, which is X_1. The search will then proceed to $X_1 = 2$ onwards. So in this process, we have jumped across a major portion of search space. But this search space contains the actual maximal solution found by DFBB. So in applying CBJ to the branch and bound algorithm, the correctness of the algorithm is lost in CBJBB.

The phenomenon occurring here is similar to that occurring in the plain backjumping case as shown in [16]. The anomaly occurs because of the presence of an inconsistent node X_2 between the point to which conflict-directed backjumping occurs and the current level. The inconsistent node in the present branch leaves the prospect of a better solution in an alternative branch at the inconsistent node (X_2 here). Here the maximal solution $\{X_1 = 1, X_2 = 2, X_3 = 1, X_4 = 1\}$ occurs in one such alternative branch.

The solution suggested by [16], can be extended to the CBJBB case, but with additional bookkeeping. The idea is to backjump to the deepest of the two variables, the backjump point or the deepest inconsistent node prior to the current node. In BJBB case, a variable was kept to keep track of the deepest inconsistent node encountered.

But in CBJBB case, we need to keep track of the information of the deepest inconsistent node prior to every level, not just for the present level because backjumping requires only conflict information corresponding to the present level while in CBJ, the conflict information of the present level is passed on to the upper level to which backjumping occurs. So the information pertaining to the inconsistent nodes should

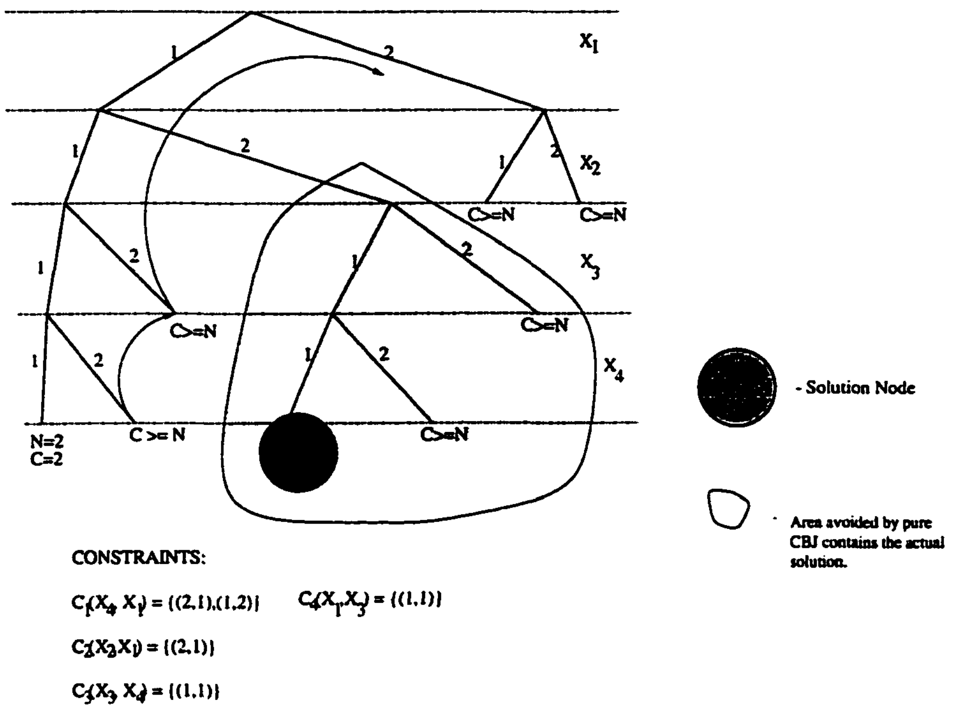


Figure 5.10: CBJ example

be kept corresponding to each level. Let inconsistent[i] correspond to the deepest inconsistent node prior to the level i. This information is maintained and passed on in the following manner. Initially all are initialized to 0. Then as the partial solution is expanded on in the CBJBB algorithm, all entries still remain 0, till an actual inconsistent node say X_j is encountered. Then as this partial solution is expanded to X_{j+1}, then inconsistent[j + 1] is updated to X_j. The same value X_j is passed on to all entries till the next such inconsistent node is encountered, say X_c. Even inconsistent[c] is made to X_j. For any further expansion of this value X_c will be used. This goes on till an actual backjump takes place. At a level k, let X_i be the deepest of the two variables inconsistent[k], and the deepest variable in conflict-set[k]. Now when backjumping takes place to X_i, two changes are done. Conflict-set[i] is updated to the union of conflict-set[i] and conflict-set[k] minus the variable X_i. Further expansion along the alternate value of the variable X_i then proceeds again as before by keeping tracking of the deepest node prior to any variable X_j in inconsistent[j].

This minor additional bookkeeping ensures the correctness of the CBJBB algorithm.

5.4.4 Analysis of branch and bound algorithms for the max-CSP

In this section we shall explore some theoretical results associated with the branch and bound algorithms for max-CSP. The results are analogous to ones discussed for plain constraint satisfaction problems in [31]. Here we observe that since in max-CSP, we cannot usually satisfy all the constraints at one time, we need to concentrate on the problem of satisfying a subset of constraints at a time.

The main idea of this section is to present a characterization of static conditions under which a particular backtrack algorithm visits a node in the search. In doing so, it is possible to sometimes obtain a rough idea of the behavior of the backtrack algorithm. A characterization in terms of dynamic conditions is not of practical use since the checking of conditions in such case itself will be a costly process. By dynamic characterization, we mean a characterization in terms of a variable which dynamically changes during the search process.

The main difficulty in extrapolating the results from [31], to max-CSP is that the mere presence of an inconsistency at a node does not stop the search at that point in branch and bound algorithms.

In depth-first branch and bound method, N (the number of allowed constraint violations) varies as we proceed along the way in the search process. Hence it is impossible to obtain any static condition dependent on N for measuring the effectiveness of any branch and bound algorithm. So we need to fix our value of N, the necessary bound on the number of constraints violated in the max-CSP, in order to get static quantified results pertaining to the performance of the individual branch and bound algorithms. So from now on in this section we concentrate on the problem of max-CSP, with a predetermined N, for the conditions to be applicable. In other words, in the modified formulation of the max-CSP, we would like to find the best possible solution to the CSP subject to the condition that no more than N (predetermined or fixed) constraints are violated.

Based on a measure of N, we can now elucidate sufficient and necessary conditions for the different variants of the branch and bound methods to work for this version of the max-CSP.

Depth first branch and bound

In plain depth first branch and bound (DFBB) method, the search proceeds by assigning a value for a variable and then expanding the variable sets till either all variables are exhausted or we reach a point where the limit of N constraint violations is reached.

Because of the presence of the predetermined bound N on the number of inconsistencies, we can elucidate a sufficient condition for the DFBB algorithm to visit a node.

Theorem 5.11 *If DFBB visits a node, its parent is a node with less than N inconsistencies.*

Proof. If the parent of the node had greater than N inconsistencies, then by virtue of the algorithm the current node will never be reached and bounding will occur at the parent node itself. Hence the proof. □

Unfortunately, it is not possible to give a static necessary characterization of the algorithm. Because during the course of the algorithm, if on the way, a solution is found with less than N inconsistencies, then N can be updated in such a case and this process can be dynamic. But if we know beforehand that any solution has a minimum of N_{min} inconsistencies, then it would be possible to give a necessary condition similar as above:

Theorem 5.12 *If a node is such that its parent is node with $< N_{min}$ inconsistencies, then the node is visited by DFBB.*

Proof. Because N_{min} represents the lowest number of inconsistencies in the entire search space of DFBB, any node visited by DFBB has either N_{min} or higher number inconsistencies. Hence the proof. □

The above results show that the condition analogous to the consistency of a node in backtracking case [31], is the condition that the node have less than N inconsistencies. This is explained by the fact that in DFBB, a failure is triggered by a partial solution with greater than N inconsistencies while in backtracking the failure is triggered by an inconsistency.

Backjumping

Before we give a sufficient condition characterizing the branch and bound algorithm with backjumping, we state the following lemma. We assume the presence of a bound N as in DFBB case.

Lemma 3 *Any node visited by BJBB to a_j after a_i such that $(i > j)$, is such that $(a_1, a_2, ..., a_j)$ along with any of the value of x_i will have $\geq N$ inconsistencies.*

Proof. The proof follows from the fact that backjump occurs from the node x_i to x_j. Irrespective of the fact whether x_j represents the deepest inconsistent node before x_i or the deepest variable in conflict with x_i, no variable between x_j and x_j is inconsistent or has conflict with x_j. So any of these variable instantiations do not add any further inconsistencies caused by $a_1, ..., a_j$ with a_i. And we know that backtracking occurs at a_i, so it follows the $a_1, a_2, ..., a_j$ with a_i have $\geq N$ inconsistencies. □

Based on the above condition the sufficient condition for the BJBB algorithm can be written as follows:

Theorem 5.13 *If BJBB visits a node, its parent node has less than N inconsistencies when combined with any other variable.*

Proof. The theorem directly follows from the lemma above. □

Again, it is difficult to elucidate a necessary condition for the BJBB algorithm, because of the same problem as in case of DFBB. We change the value of N whenever a partial solution having lesser number of inconsistencies is encountered. Once again if we are ensured that we have a measure of the best solution having N_{min} inconsistencies, the necessary condition for the BJBB algorithm can then be stated as follows:

Theorem 5.14 *If a node is such that its parent has less than N_{min} inconsistencies when combined with any other variable, then BJBB visits the node.*

Proof. Along the lines of the proof for DFBB. □

Conflict-directed backjumping

Strengthening the argument along the lines of DFBB and BJBB, for the case of CBJBB we deduce similar results for CBJBB too. As in DFBB and BJBB, we need to assume predetermined bounds N and N_{min}.

Extending the results along the same line of argument to CBJBB, we get the following necessary and sufficient conditions which we state without proof.

Theorem 5.15 *If CBJBB visits a node, its parent node has less than N inconsistencies when combined with any other set of variables. Conversely, if any node is such that its parent node has less than N_{min} inconsistencies in combination with any set of variables, then CBJBB visits the node.*

5.4.5 Conclusions on solution methods for max-CSP

In this section we studied the problem of max-CSP in detail with stress on the intelligent retrospective branch and bound techniques. We extend the observations made in [16] to conflict-directed backjumping. The other contribution of this section is a detailed theoretical analysis of the intelligent backtrack algorithms in the domain of over-constrained problems, along the line pursued in [31] for CSPs.

5.5 Summary of the chapter and discussion

In this chapter we have provided a critique of the existing semantic notions of solution in over-constrained problems. In particular we show two theoretically negative results pertaining to the existing semantics of over-constrained problems.

The first class of these results pertain to the computational complexity of these over-constrained problems. All the semantics studied for over-constrained systems were shown to fall in the second level of the polynomial hierarchy, which represents a class of problems computationally harder than SAT and other NP-complete problems.

The second result pertains to the translatability of non-binary to binary representations of the over-constrained problems. We show that in max-CSP the two most commonly used methods for translation from non-binary representations to binary representations, namely the dual graph method and the hidden variable method, fail to preserve the semantics of max-CSP in the translation process. Thus semantics of over-constrained problems is not preserved in translation from non-binary to binary CSP problems.

Finally, we critically examine the solution methods for max-CSP based on intelligent branch and bound. We provided theoretical characterizations for these algorithms in terms of necessary and sufficient conditions for these algorithms to visit a node.

Let us briefly discuss the causes leading to the first two results in the chapter.

The diversity of the types of priority information present in over-constrained systems studied in this chapter stresses the need to distinguish between global and local optimality criteria of solutions. All the existing over-constrained frameworks max-CSP, max-weighted CSP, and comparator based HCLP, base their definition of solution on global comparison measures. These global comparison measures define a concept of a preferred solution among a set of candidate solutions. The measures often used in such systems are extra-logical in the sense that they are external measures outside the logical semantics of the problem. The extra-logical nature of such measures is precisely the reason for their erratic behavior in otherwise semantics-preserving logical transformations.

It is the requirement of comparison among multiple solutions, which is responsible for the higher complexity class of max-CSP, max-weighted CSP and other similar over-constrained semantics. On the other hand, the extra-logical nature of the measures is responsible for the non-preservation of semantics in translation from non-binary to binary over-constrained representations. These two factors suggest the need to look for an alternative reasonable semantics for over-constrained problems which is neither extra-logical nor involves any inter-solution comparison.

Chapter 6

Conclusions and Scope for Future Work

6.1 Contributions

In the thesis our goals have been two-fold - (i) to explore the viability of the use of non-monotonic logic programming based on stable models as a constraint programming paradigm, and (ii) to explore the semantic notions of of solution in existing over-constrained frameworks.

6.1.1 Logic programming with stable models for constraint programming

Logic programming with stable models has been evaluated in this thesis in the context of its capacity to represent and solve constraints. This thesis extends the recent work in the area of logic programming [37, 41, 43, 56] on the promotion of stable model based logic programming as a constraint programming paradigm. The task is made easier by the development of effective implementations for computing stable models of ground logic programs like **smodels** [42, 43], followed by introduction of languages based on stable models like LP_{SM} and SLP [37]. Though these languages are restricted versions of the general class of logic programs with variables, they have been shown to be capable of capturing a wide class of AI problems. Fundamental to these languages is also the interpretation of logic program clauses as *constraints* on the solution sets.

As a representative language we concerned ourselves with the language LP_{SM} and its underlying stable model computation engine **smodels**.

On the representation angle, we extended the observations made by Niemela in [41] on the capacity of LP_{SM} to model constraint satisfaction problems. Further any constraint problem which can be captured in LP_{SM} can be captured by a function-free normal logic program (FFNLP). LP_{SM} has been shown to be declarative in nature, an essential requirement for a good modeling language. This feature it shares with other constraint programming langauges like Oz [53], Eclipse [38], and OPL [26]. Further, certain classes of constraint programming situations like dynamic CSPs [56] seemed to be better captured in LP_{SM}.

On the efficiency angle, we studied the important techniques incorporated in the implementation of **smodels**, which is by far the most competitive implementation developed for computing stable models. We find that the three main techniques used in **smodels** (constraint propagation, **lookahead**, and **backjumping**) are mappings from well known efficient techniques in finite constraint satisfaction problems (CSPs) because of their having originated from erstwhile SAT techniques and SAT itself being a specific type of CSP. Our investigation of these techniques was two-fold: (i) to measure the relative effectiveness of each technique in terms of contribution to the efficiency of **smodels**, and (ii) to conduct a preliminary comparison of the efficiency of these techniques in **smodels** with the corresponding techniques in finite CSPs. This study was performed in the context of logic programs modeling finite CSPs. Our investigation reveals that **lookahead** dominates the other two techniques in **smodels**, and that **smodels** employing **lookahead** can be as efficient as some of the efficient techniques in finite CSPs.

This corroborates some of the contentions which motivated our search for the correspondence between finite CSP techniques and **smodels** techniques. Some contentions which are given added credence based on our observations are:

1. Techniques which render efficiency to a specialized representation can be generalized to more general representations.

2. General purpose knowledge representation systems can be built which are as efficient as some special domain specific solvers.

6.1.2 Semantics of over-constrained systems

In the latter part of the thesis we started out with an interest in representing and solving the over-constrained problems by logic programming with stable models. However the computation and representation of these problems would be impossible without a thorough understanding of their semantics. During the pursuit we observed that the semantics of over-constrained problems has not been adequately dealt with in the literature. Since time only permits us to investigate the computational and semantic difficulties with the standard notions of solution in the over-constrained context, our goal is only partially achieved.

Our studies of over-constrained semantics nevertheless produced interesting conclusions with leads for new directions of research. In particular we have shown and corroborated that the existing notions of solutions in over-constrained problems suffer from the following semantic problems:

1. ad-hoc semantics;

2. higher computational complexity;

3. semantics not preserved in translation from non-binary to binary representations; and

4. techniques used in solving finite CSPs cannot be used directly.

We therefore leave the original goal of this part of the thesis to future investigation.

6.2 Scope for future work

The work presented in the thesis can be extended along many directions. Some important directions are discussed here.

6.2.1 Alternative semantics for over-constrained problems

The discussion in Chapter 5 motivates us to look for an alternative reasonable semantics for over-constrained problems which is neither extra-logical nor involves any inter-solution comparison.

One possible candidate which fits the bill for such a semantics is the idea of a solution based on *maximal consistency.* Instead of comparing between candidate solutions, in any solution semantics based on maximal consistency, we just verify if a particular partial solution can be expanded any further without generating a contradiction. The notion of a maximal consistent solution does not resort to any extra-logical theory solution preference criterion involving a global measure of comparison among solutions. Clearly, such a *maximally consistent* partial solution can be computed in polynomial number of steps by expanding a partial assignment till no more variables can be added to it without generating inconsistency. At the same time the notion of *maximal consistency* must *respect* the CSP, i.e. it must be ensured that if there is a solution S satisfying all the constraints, then S must be the desired answer. This is analogous to the requirement of a comparator in HCLP to *respect* the constraint hierarchy (i.e. if there is a solution satisfying all constraints till a level k in the constraint hierarchy then any answer returned by a comparator must also satisfy all constraints till level k). Based on the above, we can conclude that the complexity of computing a *maximal consistent* solution is related to the following decision problem of D:

Definition 12 (Maximal consistency decision problem) *Does a finite CSP P have no solution satisfying all the constraints in P?*

Clearly the decision problem P is co-NP-complete because the complement of P is the finite CSP decision problem which is known to be NP-complete [35].

Theorem 6.1 *The maximal consistency decision problem is co-NP-complete.*

Proof. Trivially follows from the fact that finite CSP decision problem is NP-complete. □

Thus the semantics based on *maximal consistency* restricts the complexity to within the first level of the polynomial hierarchy (Π_1 = co-NP). In spite of this, there are some potential disadvantages of using *maximal consistency*. The main disadvantage of using *maximal consistency* is the number of solutions that can be generated. *Maximal consistency* can generate a large number of partial solutions. One possible way to restrain the number of solutions is by using some kind of priority information which allows filtering of candidate solutions. Thus the search for alternative reasonable semantics of over-constrained problems is one future direction of research.

6.2.2 Relationship between over-constrained semantics and logic programming with stable models

Our original goal in the latter part of the thesis was to extend the logic programming techniques to capture over-constrained semantics. We leave this original goal as a future direction of research. We identified the semantic problems with existing notions of solution in over-constrained systems.

The paradigm of logic programming with stable models bears a close relationship to other paradigms of non-monotonic reasoning like default logic [49]. In the literature relationships have been established between non-monotonic reasoning formalisms and over-constrained formalisms. Notable among them are [1, 52, 2, 6]. In [1, 2] a relationship is established between default logic and partial constraint satisfaction (an over-constrained formalism). In [6] different variants of constraint relaxation have been related to non-monotonic reasoning. In [51] a relationship is established between HCLP and circumscription.

Given the abundance of these relationships it is a worthwhile task to look into the feasibility of stable model based logic programming to capture over-constrained problem semantics.

6.2.3 Comparing smodels techniques with non-binary CSP techniques

In this thesis, most of the finite CSP techniques which were mapped to **smodels** techniques were restricted to binary representations of constraints. In the finite CSP literature however, there are two schools of thought in terms of generalization of these techniques to non-binary representations.

- Solving the non-binary problems by converting them to binary representations first and then solving the binary problems.
- Solving the non-binary problems by directly employing the generalizations of the techniques used in solving binary finite CSPs.

The future direction of research suggested in this section is to study empirically and theoretically the results of comparison of **smodels** techniques and non-binary finite CSP techniques. The representation of **smodels** is general enough to capture non-binary constraints. In view of the comparable performance of some **smodels** techniques restricted to logic programs representing binary finite CSPs (when compared to their corresponding binary CSP techniques), our conjecture is that **smodels** should perform as well as the finite CSP techniques generalized to non-binary representations. We also conjecture that the performance of **smodels** should be better than the performance of the two-stage non-binary solvers involving the additional overhead of translation to binary representations.

The study of non-binary constraint satisfaction problems is still in a nascent stage where significant methods need to be developed for the generalized representation scheme followed in non-binary CSPs. In view of the large scope for development of comprehensively efficient techniques for non-binary CSPs, **smodels** holds promise in terms of solving generalized representations of finite CSPs.

6.2.4 Enhancing the smodels proof procedure

The present version of the implementation of the **smodels** proof procedure employs three main speedup techniques: constraint propagation, **lookahead**, and **backjumping**. In contrast, in the finite CSP literature many additional techniques have been developed with significant results. To name a few like generalized arc-consistency (GAC) [34], generalized arc-consistency with conflict-directed backjumping (GAC-CBJ) [47], and iterative repair.

The constraint propagation in **smodels** has been shown to be a restricted version of the technique of arc-consistency when applied to binary CSPs. We saw how the propagation achieved by **expand** can be enriched by incorporating arc-consistency for the binary CSPs. In GAC the arc-consistency for binary CSPs has been generalized to take care of non-binary representations of constraints. So it would be interesting to see if GAC can be mapped to a more efficient and general version of the **expand** procedure in **smodels**. The idea is to exploit the structure of the programs to be able to achieve a more effective propagation than that achieved by the generic propagation rules used by **expand** in the current version of **smodels**.

Iterative repair based techniques have been proved to be successful for a large class of scheduling [66] and planning problems encoded as CSPs. The iterative repair methods start from an imperfect solution and gradually move towards a perfect solution by a constant repair and mix process. It would be interesting to see if **smodels** can accommodate iterative repair to arrive at a solution faster.

Thus the line of research advocated in this section is to look into the feasibility of application of more general and efficient CSP techniques to enhance the efficiency of **smodels**.

6.2.5 Performance and structural analysis of stable model computation methods

Comprehensive research of finite CSPs has revealed the intricate structure of the search space of CSP problems. It makes the possibility of identification of the regions of *hardest* CSPs possible. Further, structural characteristics of CSPs have been identified to provide sufficient conditions under which polynomial time algorithms exist for CSPs.

Similar studies needs to be undertaken for the stable models. Although **smodels** and similar implementations of stable models are proving to be successful, there is a lot of work remaining in the structural and performance analysis of these systems. A comprehensive set of benchmarks need to be developed for the evaluation of stable model implementations. This type of performance analysis is only possible if it is possible to analyze the structures of stable models in a comprehensive manner. This is an area of research which requires immediate attention if the paradigm of logic programming were to be pushed further as a practical problem-solving paradigm.

Bibliography

[1] Aditya K. Ghose Abdul Sattar and Randy Goebel. Specifying overconstrained problems in default logic. In *Proc. Workshop on Over Constrained Systems,CP 95*, pages 253–263, 1995.

[2] Abdul Sattar Aditya Ghose and Randy Goebel. Default reasoning as partial constraint satisfaction. In *Proc. of Australasian Joint conference on Artificial Intelligence*, Armidale,NSW,Australia, 1994. World Scientific Publishing Co.

[3] Fahiem Bacchus and Peter van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *Proceedings 15th National Conference on Artificial Intelligence*, Madison,Wisconsin, 1998.

[4] Alan Borning, Robert Duisberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint hierarchies. In *OOPSLA'87*, pages 48–60, October 1987.

[5] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5:223–270, 1992.

[6] Gerhard Brewka, Hans Werner Guesgen, and Joachim Hertzberg. Constraint relaxation and nonmonotonic reasoning. Technical Report TR-92-002, ICSI, Berkeley, 1992.

[7] Pawel Cholewinski, Victor W. Marek, and Miroslaw Truszczynski. Default reasoning system DeReS. In Luigia Carlucci Aiello, Jon Doyle, and Stuart

Shapiro, editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, pages 518–528, San Francisco, November 5–8 1996. Morgan Kaufmann.

[8] Stephen A. Cook. The complexity of theorem-proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Shaker Heights, Ohio, 3–5 1971 1971.

[9] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[10] Martin Davis and Hilary Putnam. A machine program for theorem proving. *CACM*, 5(7):394-397, 1962.

[11] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, January 1990.

[12] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38(3):353–366, April 1989.

[13] J. Freeman. Improvements to propositional satisfiability algorithms. Phd thesis, University of Pennsylvania, PA, 1995.

[14] E. C. Freuder. A sufficient condition for backtrack-free search. *Journal. of the A.C.M.*, 29(1):24–32, January 1982.

[15] Eugene Freuder. Partial constraint satisfaction. In *IJCAI-89: Proceedings 11th International Joint Conference on Artificial Intelligence*, pages 278–283, Detroit, 1989.

[16] Eugene C. Freuder and Richard J. Wallace. Partial constraint satisfaction. In *Proc. Workshop on Over Constrained Systems,CP 95*, pages 63–109, 1995.

[17] D. Frost and R. Dechter. Looking at full looking ahead. *Lecture Notes in Computer Science*, 1118:539–540, 1996.

[18] Daniel Frost and Rina Dechter. In search of the best constraint satisfaction search. In *National Conference of Artificial Intelligence, AAAI-94*, Seattle, WA, USA, August 1994.

[19] Daniel Frost and Rina Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'95*, pages 572–578, Montreal, Canada, August 1995.

[20] Michael R Garey and David S Johnson. *Computers and Intractibility, a Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., San Francisco, 1979.

[21] J. Gaschnig. A general backtrack algorithm that eliminates most redundant checks. In *Proc. IJCAI*, page 457, 1977.

[22] J. Gaschnig. Experimental case studies of backtrack vs. waltz-type vs. new algorithms for satisfying assignment problems. In *Proc. second biennial conference of the Canadian Society for Computational studies of intelligence*, pages 268–277, 1978.

[23] A Van Gelder, K. Ross, and J. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *Proc. of PODS*, 1988.

[24] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. 5th ICLP*, pages 1070–1080. MIT Press, 1988.

[25] R. M. Haralick and G. L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[26] Pascal Van Hentenryck. *The OPL Optimization Programming Language.* MIT Press, Massachussets, USA, 1999.

[27] J. Jaffar and M.J.Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, pages 503–581, 1994.

[28] Joxan Jaffar and Michael Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[29] A.C. Kakas and A. Michael. Integrating abductive and constraint logic programming. In *Proc. 12th ICLP*, pages 399–413, 1995.

[30] H. Kautz and B. Selman. Pushing the envelope: Planning propositional logic and stochastic search. In *Proc. of American Association for Artifical Intelligence*. AAAI Press, 1996.

[31] Gregorz Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. In *Proc. Fourteenth IJCAI*, pages 541–546, Montreal, Canada, August 1995.

[32] Mark W. Krentel. The complexity of optimization problems. *Journal of Computer and System Sciences*, 36(3):490–509, June 1988.

[33] A. Mackworth. Constraint Satisfaction. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 205–211. John Wiley and Sons, 1987.

[34] Alan Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[35] Alan Mackworth and Eugene Freuder. The complexity of constraint satisfaction revisited. *Artificial Intelligence*, 59(1–2):57–62, February 1993. Special Volume on Artificial Intelligence in Perspective.

[36] W. Marek and T.C. Przymusinski. Autoepistemic logic. *Journal of the ACM*, 38:588–619, 1991.

[37] W. Marek and M. Truszczynski. Stable models and an alternative logic programing paradigm. In *The Logic Programming Paradigm: a 25 year perspective*, pages 375–398. Springer-Verlag, 1999.

[38] Joachim Schimpf Mark Wallace, Stefano Novello. *ECLiPSe : A Platform for Constraint Logic Programming.* William Penney Laboratory, Imperial College, London SW7 2AZ, 1997.

[39] Eugene Freuder Michael Jampel and Michael Maher (eds.). *Over-Constrained Systems : Proceedings of the 1995 workshop.* Springer-Verlag, 1995.

[40] Ilkka Niemel, Patrick Simons, and Timo Soininen. Stable model semantics of weight constraint rules. In *Proceedings of Fifth International Conference on Logic Programming and Nonmonotonic Reasoning*, El Paso, TX, USA, 1999. Springer Verlag.

[41] Ilkka Niemela. Logic programs with stable model semantics as a constraint programming paradigm. In *Workshop on computational aspects of nonmonotonic reasoning*, Trento,Italy, May 30 - June 1 1998.

[42] Ilkka Niemela and Patrick Simons. Smodels - an implementation of the stable model and the well-founded semantics for normal logic programs. In *Proceedings of 4th International conference on Logic Programming and Non-monotonic reasoning*, pages 420–429, 1997.

[43] Ilkka Niemela, Patrick Simons, and Timo Soininen. Stable model semantics of weight constraint rules. In *Proceedings of 5th International conference on Logic Programming and Non-monotonic reasoning*, pages 420–429, December 1999.

[44] C. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[45] David Poole, Randy Goebel, and Romas Aleliunas. Theorist: A logical reasoning system for defaults and diagnosis. Technical Report ?, Department of Computer Science, University of Waterloo, 1986.

[46] Patrick Prosser. Domain filtering can degrade intelligent backtracking. In *Proc. of Thirteenth IJCAI*, Chamberry, France, August 1993.

[47] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, August 1993. (Also available as Technical Report AISL-46-91, Stratchclyde, 1991).

[48] Patrick Prosser. An empirical study of phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.

[49] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[50] Francesca Rossi, Charles Petrie, and Vasant Dhar. On the equivalence of constraint satisfaction problems. In Luigia Carlucci Aiello, editor, *ECAI'90: Proceedings of the 9th European Conference on Artificial Intelligence*, pages 550–556, Stockholm, August 1990. Pitman.

[51] Ken Satoh. Formalizing soft constraints by interpretation ordering. In *Proceedings European Conference on Artifical Intelligence*, 1990.

[52] Abdul Sattar and Randy Goebel. Constraint satisfaction as hypothetical reasoning. In *Proc. Vth International Symposium on Artificial Intelligence*, Cancun,Mexico, 1992. AAAI Press.

[53] Christian Schulte and Gert Smolka. *Finite Domain Constraint Programming in Oz. A Tutorial.* DFKI, D-66123, Saarbrucken,Germany, 1994.

[54] Patrck Simons. Towards constraint satisfaction through logic programs and the stable model semantics. Technical Report A47, Helsinki University of technology, Helsinki, Finland, August 1997.

[55] Barbara M. Smith and Martin E. Dyer. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:155–181, 1996.

[56] Timo Soininen, Esther Gelle, and Ilkka Niemela. A fixpoint definition of dynamic constraint satisfaction. In *Proceedings of Principles and Practice of Constraint Programming - CP96*, Alexandria, VA, USA, 1999. Springer Verlag.

[57] Timo Soininen and Ilkka Niemel. Developing a declarative rule language for applications in product configuration. In *Proceedings of First International Workshop on Practical Aspects of Declarative Languages*, pages 305 – 319, San Antonio, TX, USA. Springer Verlag.

[58] Peter van Beek. CSP C library. Public domain software, University of Alberta, Edmonton, Canada, 1994. Available at ftp://ftp.cs.ualberta.ca/pub/vanbeek/software/.

[59] Paul van Run. Domain independent heuristics in hybrid algorithms for constraintsatisfaction problems. M. math thesis, University of Waerloo, Ontario , Canada, 1994.

[60] W.F.Dowling and J.H.Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulas. *Journal of Logic Programming*, 3:267–284, 1984.

[61] C.P. Williams and T. Hogg. Using deep structure to locate hard problems. In *Proc. of American Association for Artifical Intelligence*, pages 472–477. AAAI Press, 1992.

[62] C.P. Williams and T. Hogg. The typicality of phase transitions in search. *Computational Intelligence*, 9(3):221–238, 1993.

[63] Molly Wilson and Alan Borning. Hierarchical constraint logic programming. *Journal of Logic Programming*, 16(3):277–318, July 1993.

[64] J. You and L. Yuan. A three-valued semantics for deductive databases and logic programs. *J. Computer and System Sciences*, 49:334–361, 1994.

[65] L. Yuan. Autoepistemic logic of first order and its expressive power. *J. Automated Reasoning*, 13(1):88–116, 1994.

[66] M. Zweben, E. Davis, B. Daun, and M. Deale. Scheduling and rescheduling with iterative repair. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6):1588–1596, 1993.