USING CRITICALITIES AS A HEURISTIC FOR ANSWER SET PROGRAMMING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

ORKUNT SABUNCU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

THE DEPARTMENT OF COMPUTER ENGINEERING

SEPTEMBER 2002

Approval of the Graduate School of Natural and Applied Sciences.

Prof. Dr. Tayfur Öztürk
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Ayşe Kiper
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Ferda Nur Alpaslan
Supervisor

Examining Committee Members

Prof. Dr. Varol Akman

Assoc. Prof. Dr. Ferda Nur Alpaslan

Assoc. Prof. Dr. İsmail Hakkı Toroslu

Assist. Prof. Dr. Bilge Say

Dr. Ayşenur Birtürk

ABSTRACT

USING CRITICALITIES AS A HEURISTIC FOR ANSWER SET PROGRAMMING

Sabuncu, Orkunt
M.Sc., Department of Computer Engineering
Supervisor: Assoc. Prof. Dr. Ferda Nur Alpaslan

September 2002, 55 pages

Answer Set Programming is a new paradigm based on logic programming. The main component of answer set programming is a system that finds the answer sets of logic programs. Stable model semantics is used to find the stable models of the program which correspond to the answer sets. Generally, systems utilize some heuristics to choose new literals so that they can augment a candidate model with the chosen literals in order to find an answer set. The heuristic used in this process is one of the key points for the performance of the system.

In this thesis, a new heuristic for answer set programming has been developed. The main idea of this heuristic is inspired by hierarchical planning. The notion of criticality, which was introduced for generating abstraction hierarchies in hierarchical planning, is used as the new heuristic. The resulting system called CSMODELS uses this new heuristic and is based on the system SMODELS. The experimental results show that this new heuristic is promising for answer set programming. CSMODELS generally takes less time than SMODELS to find an answer set.

Keywords: answer set programming, logic programming, SMODELS, stable model semantics, criticalities, abstraction hierarchy

ÖZ

KRİTİKLERİN YANIT KÜMESİ PROGRAMLAMADA BULUŞSAL YÖNTEM OLARAK KULLANILMASI

Sabuncu, Orkunt
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Ferda Nur Alpaslan

Eylül 2002, 55 sayfa

Yanıt kümesi programlama temeli mantık programlama olan yeni bir yaklaşımdır. Ana bileşeni mantık programlarının yanıt kümelerini bulan dizgedir. Programın yanıt kümelerine karşılık gelen durağan modeller (stable models) durağan model anlambilimi kullanılarak bulunur. Sistemler yeni harfler (literals) seçebilmek için genelde bazı buluşsal yöntemler (heuristics) kullanırlar ve seçilen harfleri aday modele (candidate model) yanıt kümesi bulmak için eklerler. Kullanılan buluşsal yöntemler sistemin performansı açısından önemli bir unsurdur.

Bu çalışmada yanıt kümesi programlama sistemleri için yeni bir buluşsal yöntem geliştirilmiştir. Bu yöntemin ana fikri için hiyerarşik planlamadan esinlenilmiştir. Soyutlama hiyerarşilerinin oluşturulmasında kullanılan kritik kavramı (criticality) geliştirilen buluşsal yöntemin temelini oluşturur. SMODELS sistemi üzerinde geliştirdiğimiz CSMODELS bu yeni yöntemi kullanır. Deneysel sonuçlar yeni buluşsal yöntemin ümit verici olduğunu gösteriyor. Genelde CSMODELS'ın yanıt kümesi bulması SMODELS'dan daha az zaman alıyor.

Anahtar Kelimeler: yanıt kümesi programlama, mantık programlama, SMOD-ELS, durağan model anlambilimi, kritikler, soyutlama hiyerarşisi

To my family,

for their never-ending support and encouragement

ACKNOWLEDGMENTS

I would like to thank Assoc. Prof. Dr. Ferda Nur Alpaslan for her supervision and guidance throughout the development of this thesis. I would also like to thank Prof. Dr. Varol Akman for his comments and suggestions in the TAG research group meetings. I am grateful to Prof. Dr. Vladimir Lifschitz for his comments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Answer Set Programming is a new programming paradigm. It is based on logic programming but solutions of a problem are not extracted from a proof session [1]. Problems are represented in a declarative way and the input language is highly expressive [2]. It is a model based approach; the logic program representing the problem is formed in such a way that intended models of it correspond to solutions of the problem. Several semantics for logic programs are defined to select models; *stable model semantics* is used in answer set programming. Models found by the stable model semantics are called *stable models* or *answer sets* and each of them correspond to a solution of the problem.

The main component of answer set programming is the system that finds the answer sets. Generally, it takes a ground logic program as input and searches for answer sets of the program according to the stable model semantics. In this sense they can be seen as the implementations of stable model semantics.

The main algorithm of finding an answer set is common for almost all systems. The idea is simple; model generation and model verification are the main procedures [2]. In model generation, a candidate model is formed for the input logic program. The system tries to transform the candidate model into an answer set in model generation also. The candidate model at hand is checked regularly whether it is an answer set or not by the model verification procedure.

The candidate model is usually small and does not cover all the atoms of the program when it is first formed by model generation procedure (i.e., there are still uncovered atoms which have not been interpreted as true or false by the model). So, in model generation, system tries to augment the candidate model at hand deterministically by analyzing the semantics of the rules of the logic program. However, sometimes it cannot augment the model deterministically any more. At these times, which are called *choice points*, it has to augment the model by selecting an uncovered atom and assigning it true or false as an interpretation value. Heuristics are usually used to select an uncovered atom.

The literal chosen is an important point for the computation of an answer set. Some choices cause the system find an answer set very quickly, but some cause it to enter an incorrect search path and to consume lots of time before it backtracks. So, heuristics used for choosing literals greatly affect the performance of the system.

Working on new heuristics is important for developing better systems for answer set programming. In this work, a new system called CSMODELS (Criticality SMODELS), which is based on SMODELS [3], is developed. SMODELS is the mostly used system in answer set programming. CSMODELS uses a new heuristic whose foundation is *criticality*. Criticality has been used in hierarchical planning.

Hierarchical planning is a way to deal with search space complexity of planning problems. Hierarchical planners attack the problem at different levels of detail. Some parts of the original problem can be neglected at a level to form an abstract problem. *Abstraction hierarchy* is used to generate all the abstract problems at all levels. The plan found for an abstract problem is called an *abstract plan* and is used as a guide to find the plans at the lower levels. This will ease the job of the planner at lower levels, but sometimes there are no plans and it becomes a necessity to backtrack to upper levels to search for another abstract plan.

There are good abstraction hierarchies that improve the performance of the hierarchical planner, but there are also bad ones which cause a lot of backtrackings. In a good abstraction hierarchy, the upper levels (the most abstract ones) should deal with the hardest part of the planning problem so that planner tries to solve them first. This will limit the number of backtrackings between the levels during plan finding process [4].

The notion of criticality was introduced in [5] for automatically generating good abstraction hierarchies based on the property about a good abstraction hierarchy stated previously. Criticality of a literal in a planning problem is a numerical value and approximates the difficulty of finding a plan that achieves this literal. A method for numerical simulation of the plan finding process was developed to calculate the criticalities in [5].

There is also a difficulty of finding a literal in an answer set for an answer set programming system. The information of the difficulties of all literals of a logic program can be useful during computation of the answer sets of that program. In hierarchical planning, trying to attack the hardest parts of the problem first leads to efficient plan finding and limited number of backtrackings. Similarly, in answer set programming, selecting the hardest literals at the first choice points can limit backtrackings and can help the system find an answer set quickly. This is the main motivation of our work. In CSMODELS, criticalities of the literals of the logic program are calculated to approximate the difficulty of finding them in an answer set. Then, this information is used as an heuristic at the choice points. It is not trivial to calculate the criticalities this time because originally criticality calculation was defined for planning problems, not for logic programs. A method for applying criticality calculation for answer set programming is also developed.

The experimental results obtained with CSMODELS are encouraging. Generally, CSMODELS finds an answer set of a program more efficiently than SMODELS. This increase in performance is significant for especially large problems.

Background information about criticalities, how to calculate them, and answer set programming are presented in Chapter 2. Chapter 3 describes the method for applying the notion of criticality to answer set programming. How to use criticalities as a heuristic for answer set programming systems is presented in Chapter 4. This chapter also gives the implementation details of CSMODELS. Experimental results of CSMODELS and comparisons with those of SMODELS are included in Chapter 5. The conclusions and future work follow in Chapter 6. Finally, logic program representations of the test problems used in experiments are presented in Appendix A.

CHAPTER 2

BACKGROUND INFORMATION

2.1 Criticalities

2.1.1 Abstraction Hierarchies in Planning

The main components of planning are the states of the world and the operators. Operators change the state of the world and form the possible connections between states. In a planning problem there are two special states: initial state and goal state. The whole search space is composed of all the states and the operators. *Planning* is searching for a path starting from the initial state to the goal state. The path found is called a *plan*. So, planning is a kind of search problem.

Abstraction is decomposition of a problem to reduce the complexity of it [6, 7]. Some parts of the problem are considered as detail and neglected in some subproblems formed after decomposition. In general, abstraction is used as a way to deal with search complexity of search problems. The system, which uses abstraction, attacks the problem in different levels of granularity. Since planning is search, abstraction can reduce the runtime required to find a plan. There are several planning systems that use abstraction, e.g. ABSTRIPS, ABTWEAK, ALPINE, and RESISTOR [6]. These planning systems use *abstraction hierarchies* and are called *hierarchical planners*. Some hierarchical planners generate abstraction hierarchies automatically, but in some systems necessary information related with the hierarchy is given as an input.

Abstraction hierarchy divides the whole problem into pieces. In subproblems, some portion of information in the original problem is considered as detail and is neglected during the process of finding a solution. It is expected that subproblems are easier to solve than the original problem. Hierarchy levels indicate which parts of the problem should be neglected or taken into account. Using the subsolutions, a solution for the whole problem is formed easily. This is easier than trying to solve the original one at first since subsolutions will guide the system.

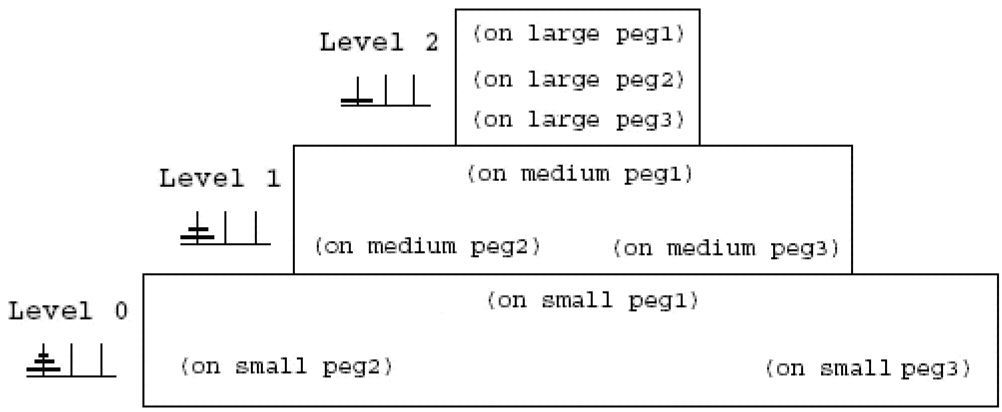

Figure 2.1: Abstraction Hierarchy for Towers of Hanoi problem generated by ALPINE [8].

Figure 2.1 is an abstraction hierarchy generated by ALPINE for Towers of Hanoi planning problem [8]. The information in hierarchy levels is the preconditions of the planning problem. In the most abstract level (i.e., highest level in the hierarchy) the problem is reduced to moving only the largest disk. This subproblem is certainly easier than the original problem. The plan found for it is called "abstract plan" [4]. The next step is solving the subproblem in the level below having an abstract plan at hand. A hierarchical planner uses the abstract plan as a guide in the level below. The information that is neglected in the first level relative to the second level is now considered as part of the problem. The neglected information in the previous level for the above hierarchy specifically is the preconditions related with the medium disk. While searching for a plan in the level below, ignored preconditions have to be achieved. This is called "refinement" [6, 4]. This process continues level by level until the ground level (i.e. the lowest level) is reached. Figure 2.2 shows refinement for the hierarchy in Figure 2.1; the figure is a slightly modified version of the one in [8]. The problem in the ground level is in fact the original problem, but the abstract plan found in the higher level will guide the search in ground level and plan finding will be easier. The plan found in the ground level is a solution of the original problem.

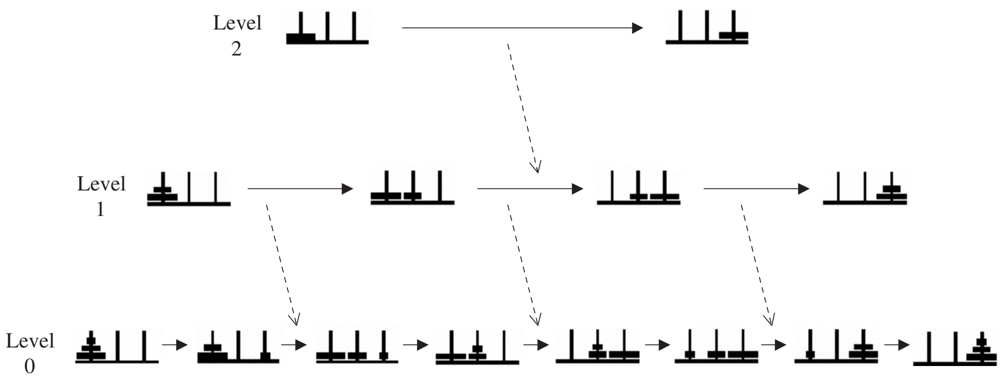

Figure 2.2: Plan finding using refinement.

Results of using abstraction in planning are generally encouraging. The above hierarchy for Towers of Hanoi problem leads to exponential reduction in the search space; thus improves the time needed for finding a plan [8].

There are also discouraging results [9, 10]. In [4], those negative results are considered as worst case analysis. The main cause of inefficiency is *trashing* [4] between levels of abstraction hierarchy. During refinement, if no plan can be found in a level, backtracking to more abstract level (upper one) is inevitable. The planner has to search for another abstract plan in the upper level and try to refine it. There may be many possible abstract plans which cannot be refined. This will cause many backtracking, and this situation is called "trashing".

A good abstraction hierarchy should avoid trashing, but finding such a hierarchy for a problem is not easy. To improve the efficiency of hierarchical problem solving, a system should try to solve a subproblem which covers the hardest part of the original problem first. This will minimize the possibility of trashing [4]. Thus, in a good abstraction hierarchy for planning, an abstract plan in the most abstract level achieves the hardest-to-achieve preconditions of the planning problem. The level below should deal with the hardest-to-achieve preconditions of the remaining ones since some are achieved in the most abstract level. This goes on until ground level is reached. In ground level, the preconditions that are easy-to-achieve are considered. The planner considers those preconditions as detail parts of the problem.

2.1.2 Criticalities and their Calculation

Section 2.1.1 explains a property of a good abstraction hierarchy; but it does not explain how to construct that kind of hierarchy. Bundy, Giunchiglia, Sebastiani, and Walsh [5] provide a method for generating good hierarchies and give information about an implementation called RESISTOR. This section is about the method in [5].

If one can sort the precondition literals of a planning problem according to their difficulty to achieve, a good abstraction hierarchy can be generated using these sorted literals. To obtain such a sorting we need to capture difficulty of preconditions. A hard-to-achieve precondition is a costly one for the planner. Calculating costs of preconditions will yield difficulty of them. Costs can be calculated by running the planner to find a plan that achieves the precondition and selecting a resource consumed (for example, time elapsed for finding a plan). This method is impractical since it needs running the planner extensively.

However, a method discussed in [5] uses numerical simulation of the plan finding process; it is more practical than extensive use of the planner directly. This method has introduced the notion of *criticality*. Criticality captures the cost of a literal. More clearly, criticality of a literal is an approximation of cost of achieving it. Criticalities have numerical values; smaller values correspond to easier to achieve literals.

Criticalities are calculated in an iterative way. The *criticality function C(p,n)* gives criticality value of literal p at the n-th iteration. There are some properties a criticality function must have. These properties form the outline of the candidate definitions for criticality functions. So, there is not only one criticality function.[1]

One important property of a criticality function is *convergence.* Criticality values must converge as the number of iterations increase. Also; there are the properties of *operator monotonicity* and *precondition monotonicity.*

Operator monotonicity ensures that literals having few operators that achieve them are the ones that are hard to achieve. A criticality function which has operator monotonicity property will give greater criticality value to a literal than another one if all the operators that achieve the former are also the operators that achieve the latter and the latter has additional operators in addition to the common ones. If two literals have the same operators then they will have the same criticality value.[2] This is called the *precondition equivalence property* and it is proved that operator or precondition monotonicity imply precondition equivalence [5].

Precondition monotonicity ensures the fact that operators with many preconditions are hard to occur; as a result literals that are achieved by these operators are hard to achieve. They must have greater criticality values. A criticality function which has the precondition monotonicity property will give greater criticality value to a literal a than another one b if for every operator that achieves a there exists an operator achieves b whose preconditions form a subset of the preconditions of the first operator achieving a.[3]

[1] Reference [5] explains two implemented systems, RESISTOR and PROBABILITY, that use different criticality function definitions.

[2] Formally, a set of operators S is *equivalent* to a set of operators T iff $|S| = |T|$ and for any $op_1 \in S$ there exists $op_2 \in T$, $Pre(op_1) = Pre(op_2)$. $Pre(op)$ is defined as the set of preconditions of operator op [5].

[3] Formally, the set of operators of a is *subsumed by* set of operators of b. Set of operators S is subsumed by set of operators T iff $|S| = |T|$ and for any $op_1 \in S$ there exists $op_2 \in T$, $Pre(op_1) \supseteq Pre(op_2)$ [5].

There is an interpretation of $C(p, n)$ as *the difficulty of finding a plan of length 0 to n achieving p.* This interpretation leads to a group of criticality function definitions.[4]

RESISTOR's definition of $C(p, n)$ is inspired by electrical resistors. Based on the above interpretation of $C(p, n)$, to calculate the difficulty of achieving p, operators whose effect is p can be seen as electrical resistors connected in parallel. The more operators there are, the more paths there will be for achieving p. So, finding p becomes less difficult. This is appropriate for the operator monotonicity property. $C(p, n)$ is calculated by parallel sum (i.e. total resistance of parallel connected resistors) of the criticalities of operators whose effect is p and the initial criticality of p (i.e., $C(p, 0)$ which corresponds to the difficulty of finding a plan of length 0). Figure 2.3 shows the electrical circuit generated by connecting resistors of operator criticalities in parallel. Total resistance of the circuit is used to calculate the criticality $C(p, n)$ in the equation (2.1).

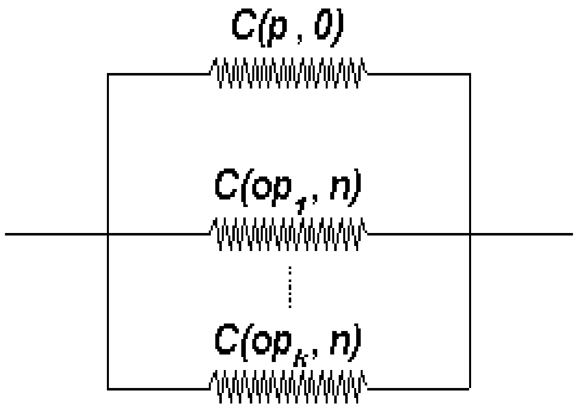

Figure 2.3: Criticality circuit for the precondition p.

$$\frac{1}{C(p,n)} = \frac{1}{C(p,0)} + \frac{1}{C(op_1,n)} + \ldots + \frac{1}{C(op_k,n)} \tag{2.1}$$

In the definition and equation for criticality of a precondition above, *the notion of criticality of an operator* is introduced. Criticality function $C(op, n)$ for an

4 This group of functions are called additive criticality functions [5].

operator *op* is not necessary but is defined to make calculations clear and simple. Similarly; it is interpreted as *difficulty of finding a plan of length 1 to n which ends with occurrence of operator op*. For calculating the criticality of an operator, preconditions of that operator can be seen as electrical resistors connected in series. Figure 2.4 shows the circuit generated by connecting resistors of precondition criticalities in serial. Total resistance equation (2.2) is used to calculate the criticality $C(op, n)$.

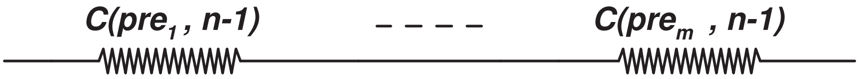

Figure 2.4: Criticality circuit for the operator *op*.

$$C(op, n) = C(pre_1, n-1) + \ldots + C(pre_m, n-1) \tag{2.2}$$

$C(op, n)$ is calculated by summing criticalities of preconditions of operator *op* (i.e., total resistance of serial connected resistors). Since interpretation of $C(op, n)$ forces occurrence of *op* at the n-th step, all the preconditions must be achieved up to the n-th step. That is why criticalities of preconditions at the $(n-1)$-th iteration are used in the definition of $C(op, n)$.

Using the equations (2.1) and (2.2), criticalities of all the precondition literals are calculated iteratively. Initial criticality values for every literal (i.e. $C(p, 0)$ terms) are assigned first. RESISTOR assigns the same value a_0 ($a_0 = 1$) to all literals' initial criticalities.

It can be shown that criticality values calculated using this method are *monotonically decreasing* and *convergent* [5]. Their limiting values will be in $[0, a_0]$. Experiments show convergence is rapid; values converge after few iterations [5]. RESISTOR continues iteration until no change occurs in the values up to some predefined *accuracy*. This is a computationally practical way for terminating iterations.

Abstraction hierarchies are generated by sorting the final criticality values; however what is important for this work is just calculating criticalities.

2.2 Answer Set Programming

Answer Set Programming is a new paradigm based on logic programming. Unlike classical logic programming, solutions to a problem are not extracted from a proof initiated by a query [1]. The problem is represented by a logic program in a declarative way, and intended models of the program correspond to solutions of the problem. The models form the semantics of the program and are called *"answer sets"* [2]. Figure 2.5 shows the key points in answer set programming.

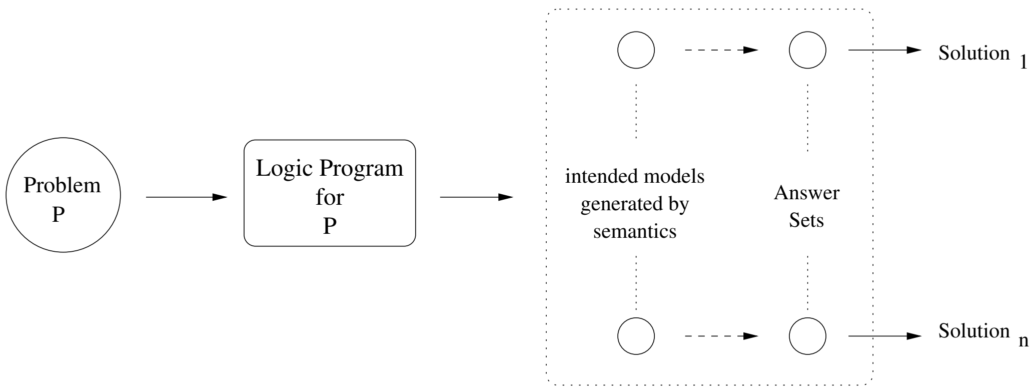

Figure 2.5: The flow in Answer Set Programming.

Various semantics for a logic program (well-founded semantics, perfect model semantics) have been defined to select a canonical model among the Herbrand models. Herbrand models contain ground atoms and a system can use the canonical model to answer a ground query [11].[5] Negation free programs have a unique minimal (w.r.t. set inclusion) Herbrand model [12] and semantics generally select this model as a canonical model.

However, programs with negation may have more than one minimal Herbrand model. So, choosing a canonical model is not trivial. Various semantics come

[5] One can answer a ground query as *yes* if it is in the canonical model otherwise *no*.

up with their ways of interpreting negation to make the semantics of a program with negation clear [13]. But, all the semantics mentioned above have some restrictions on the program related with the use of negation [11].

Gelfond and Lifschitz [11] define *stable model semantics* of logic programs that gives a clear meaning of negation. The information about this semantics is given in Section 2.2.1. There can be many stable models of the program unlike a unique intended model. At first one can think this may produce problems since stable model semantics does not choose a canonical model. In fact the stable model semantics does not worry about this and many possible models play an important role in answer set programming since they correspond to the solutions of the problem [1].

In answer set programming intended models of a program are found by stable model semantics. Each stable model corresponds to a solution of the problem to be solved. A logic program representation of the problem is constructed in a very simple way. Firstly, one writes rules in a straightforward and declarative way to represent a general form of solution for the problem domain. Then, some constraints are imposed on the general form of solution to shape it so that it becomes a solution for the problem specifically. In this sense, answer set programming has a relation with constraint programming and this relation constitutes an advantage of answer set programming over classical logic programming [13].

The *n-queens problem* is placing n queens in a $n \times n$ board so that no queen can attack another one. Here is an example logic program in [14]; it shows a representation of the n-queens problem in answer set programming.

$$d(1...n) \leftarrow \tag{2.3}$$

$$q(X,Y) \leftarrow d(X),\ d(Y),\ not\, negq(X,Y) \tag{2.4}$$

$$negq(X,Y) \leftarrow d(X),\ d(Y),\ not\, q(X,Y) \tag{2.5}$$

$$\leftarrow d(X),\ d(Y),\ d(X'),\ q(X,Y),\ q(X',Y),\ X' \neq X \tag{2.6}$$

$$\leftarrow d(X),\ d(Y),\ d(Y'),\ q(X,Y),\ q(X,Y'),\ Y' \neq Y \tag{2.7}$$

$$\begin{aligned}\leftarrow\ & d(X;Y;X';Y'),\ q(X,Y),\ q(X',Y'),\ X\neq X',\ Y\neq Y',\\ & abs(X-X')=abs(Y-Y')\end{aligned} \tag{2.8}$$

$$hasq(X) \leftarrow d(X),\ d(Y),\ q(X,Y) \tag{2.9}$$

$$\leftarrow d(X),\ not\ hasq(X) \tag{2.10}$$

Rule (2.3) forms the dimensions of the board using the $d(X)$ predicate. Rules (2.4) and (2.5) represents the general form of the solution by stating that there can be a queen at every position of the board. The predicate $q(X,Y)$ represents a queen at (X,Y) position of the board. Up to now no constraints are imposed. Rules (2.6), (2.7) and (2.8) are the constraints that shape the general model to generate one that corresponds to a solution. No two queens are allowed in the same row (2.6), in the same column (2.7), and in the same diagonal (2.8). Rules (2.9) and (2.10) impose another constraint that there must be a queen in each column. The predicate $hasq(X)$ represents that there is a queen in column X. Thus, the problem is represented in a straightforward way.

The program has variables, but while searching for stable models grounded program is used. Every stable model of the program (i.e., answer set) corresponds to a solution since ground instances of the predicate $q(X,Y)$ in the model show the legal positions of the queens on the board.

The main component of answer set programming is the system that process logic programs of that paradigm to find answer sets of them. These systems are *answer set finders*. Some of the answer set finders are SMODELS, DLV, DERES and CCALC [1].[6] SMODELS is the mostly used system and is explained in detail in Section 2.2.2. DLV uses the same semantics and similar algorithm as SMODELS uses, but its input language is disjunctive logic programs. DERES is based on default logic. CCALC is based on causal theory and can take action language as input.

[6] SMODELS: *http://www.tcs.hut.fi/Software/smodels/*
DLV: *http://www.dbai.tuwien.ac.at/dlv/*
DERES: *http://www.cs.engr.uky.edu/ai/deres.html*
CCALC: *http://www.cs.utexas.edu/users/tag/cc/*

2.2.1 Stable Model Semantics

Stable model semantics for logic programs is defined in [11], and it gives an explanation to the meaning of negation-as-failure.[7]

A logic program is composed of rules

$$A \leftarrow B_1, \ldots, B_k, not\, C_1, \ldots, not\, C_l$$

where A is the *head* atom and $B_1, \ldots, B_k, C_1, \ldots, C_l$ are atoms that form the *body* of the rule. A *literal* can be an atom or a not-atom; $not\, C_1$ is a negative literal of the atom C_1. If $k = l = 0$ then rule is called *fact* and if there is no head atom A (i.e., having false in the head) then rule is a *constraint rule.*

If the program has variables, P is considered as the program with variables are replaced by their ground forms. So, P is a *ground logic program.*

The *reduct of P for a set of atoms M*, P^M, is defined as P with the following modifications applied in order:

(i) Delete all the rules that have a negative literal $not\, C$ in the body with atom $C \in M$.

(ii) Delete all the negative literals in rule bodies.

P^M is a negation free logic program. So, it has a unique minimal Herbrand model. If this model is equal to set of atoms M then M is a stable model of the program P. Minimal Herbrand model of P^M also corresponds to deductive closure, so a set of atoms M is the stable model of P iff it is the deductive closure of P^M [3].

Here is a program.

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

[7] In explaining negation-as-failure, stable model semantics is more promising than other semantics like *completion of a logic program*[15] or *well-founded model semantics*[16]. [1]

In order to check whether $\{c\}$ is a stable model or not, take the reduct of the program w.r.t. $\{c\}$. $P^{\{c\}} = \{b \leftarrow d;\ c\}$. Since the deductive closure of $P^{\{c\}}$ is $\{c\}$, it is a stable model. Another stable model of the program is $\{a, b, d\}$.

Note that stable model $\{c\}$ implicitly assigns *false* as a truth value for atoms a, b and d. In this sense, stable model of a program covers all the atoms of it.

Another property of stable models is that every atom in the model has a ground to be true, in other words a ground for being in the model. Stable models are grounded models [3]. The ground for an atom is existence of a rule which has that atom in head and whose body is true. For the above program c has a reason to be in the stable model since there is a rule (c is a fact rule) in the reduct of the program. However, we cannot add atom b to the stable model $\{c\}$ because there are no rules that can generate atom b in $P^{\{b,c\}}$.

2.2.2 SMODELS: An Implementation of Stable Model Semantics

SMODELS has been developed at Helsinki University of Technology, and is now one of the mostly used answer set programming systems. The system implements stable model semantics for logic programs. Reference [3] explains SMODELS' main algorithm and implementation in detail.

Input to SMODELS is variable free logic programs. Programs with variables are transformed to ground logic programs by a front-end system called LPARSE[8]. The main work of SMODELS is a search process; search space is all subsets of atoms of the input program. It searches for a stable model. It uses a candidate model which is an empty set at the beginning. It tries to augment this candidate model by adding literals deterministically according to the program by using the properties of stable model semantics. Taking a candidate model and generating new literals deterministically according to the rules of the input program can be seen as *expanding* the model at hand. At an intermediate stage of the search process, interpretation of an atom can be one of the three different possible values [2]. An atom in the candidate model in the form of

[8] *Available at http://www.tcs.hut.fi/Software/smodels/.*

positive literal is *True*. This claims that atom will be in the stable model also. An interpretation value of an atom, which is in the candidate model in the form of negative literal (i.e., not-atom is in the model), is *False*. Similarly, this claims that atom will not be in the stable model. Third value for an atom is called *Undefined* (neither true nor false). For a particular candidate model, those literals of atoms that are not in it are undefined; in other words those atoms are *uncovered* by the candidate model.

The first expansion of the candidate model when it is an empty set will reveal the atoms that must be included in or excluded from all the stable models of the program. Usually; at the end of first expansion there will be still undefined atoms which are not covered by the candidate model. In SMODELS this candidate model at the end of first expansion corresponds to *well-founded model* of the program. There will be several times like this during the whole search process when there are uncovered atoms and SMODELS can not deterministically assign values to those atoms (i.e., true or false); these are called *choice points*. At choice points SMODELS should just select one of the uncovered atoms and give an interpretation value as true or false. Then, main algorithm will again try to expand the newly generated candidate model with the addition of chosen atom.

The aim is to expand the candidate model until it becomes a stable model. However expanding a model at hand can cause situations that both an atom and its negation, not-atom, are in the model at the same time because during expansion some rule of the program can generate a literal which is contradicting with an atom already in the model. This contradicting situation is called a *conflict*. A wrong choice at a choice point can be the reason for a conflict, so in case of a conflict SMODELS backtracks to the previous choice point. But, a conflict occuring at the first expansion is a result of a contradicting program; since there are no choices made the program will stop searching. That program has no well-founded model and no stable models either.

The decision criteria for a candidate model to be a stable model is that all the atoms of the program should be covered by it and there should be no conflict. Below is a simplified pseudocode of SMODELS' main algorithm in reference [3]. SMODELS also uses a well-known technique, called *lookahead*, for reducing the search space. If expanding a model after adding the literal x (**expand**$(M \cup \{x\})$) raises conflict, that model must contain the literal *not* x. Lookaheading uncovered atoms in this way, i.e. choosing them temporarily and expanding, can be useful since raising conflicts help augmenting the model by opposite interpretation of the literal lookaheaded.

```
function smodels(CM)   {CM: Candidate Model}
  CM := expand(CM)
  if conflict(CM) then
      return false
  else if no atom is undefined in CM then
      return true   {CM is a stable model}
  else
      c := Choose an uncovered atom
      if smodels(CM ∪ {c}) then
          return true
      else
          return smodels(CM ∪ {not c})
      end if
  end if
```

Figure 2.6: Main algorithm of SMODELS

The input of SMODELS is in fact the output of LPARSE since LPARSE does the grounding and preprocessing of the input logic program. Detailed information about the syntax of the input logic program can be found in the user manual of LPARSE [17]. General rules of a logic program are like,

$$a \leftarrow b,\ not\ c \tag{2.11}$$

$$\leftarrow k,\ l \tag{2.12}$$

Rule (2.12), which has no head atom, is used as a constraint rule claiming that no stable model can include both k and l. SMODELS 2.X versions introduce new and more expressive rules; *choice*, *cardinality* and *weight rules* [3].

The head atom must be true if the body of a normal rule is true. But, this is not a necessity for a choice rule. Even when the body of the choice rule (2.13) is true (i.e., both d and e are true), c may be false.

$$\{\, c \,\} \leftarrow d,\ e \tag{2.13}$$

There can be more than one literal in the head. Also, *cardinality constraints* can be used for stating the minimum and maximum number of literals that must be true if the body of the choice rule is true.

$$1\ \{c,\ k,\ not\, l\}\ 2 \leftarrow d,\ e \tag{2.14}$$

The cardinality constraint in Rule (2.14) states that at least 1 at most 2 literals in the head must be true.

Cardinality rule is used to apply cardinality constraints to literals in the body of a normal rule. The cardinality constraints are like the one used in Rule (2.14), but this time they are in the body part.

$$b \leftarrow 1\ \{a,\ b,\ not\ c\},\ d \tag{2.15}$$

If literal d and at least one of the literals a, b and *not c* are in the model,Rule (2.15) states that literal b must be in the model also. Complex literal like $1\ \{a,\ b,\ not\ c\}$ is called a *cardinality literal.* Interpretation value of a cardinality literal depends on the cardinality constraints and the interpretation values of the basic literals that form it.

Weight rules are the general forms of cardinality rules. A cardinality rule is a special case of weight rule where the weights of the literals in a cardinality literal are all 1. In a weight rule one can give more importance to some literals by giving them different weights.

$$b \leftarrow 3\ [c = 2,\ d = 3,\ not\ e = 1]\ 5 \tag{2.16}$$

The constraints in a weight rule like rule (2.16) are in terms of weights and are called weight constraints. The weights of the true literals are added and compared with the minimum and maximum values of weight constraints.

CHAPTER 3

APPLYING CRITICALITY NOTION FOR ANSWER SET PROGRAMMING

Criticality of a literal in a planning problem approximate the cost of finding a plan achieving that literal. Its purpose is to create good abstraction hierarchies. Criticalities make it possible to find hard to achieve literals and to work on them firstly during hierarchical plan finding. This will reduce the backtracking between levels.

The same idea can also be applied to answer set programming. During answer set finding process choosing the *hard to find literals* at the early choice points can reduce the backtracking and cause the system find an answer set quickly. For an answer set programming system there are costs of finding literals in an answer set similar to the costs of achieving literals in a planning problem. Knowing the costs of literals of an input logic program can be useful when computing the answer sets of the program; a system can use that knowledge at choice points. This intiution is the main motivation for the heuristic developed in this work.

A new heuristic can utilize costs of the uncovered literals at a choice point. Chapter 4 explains this new heuristic defined particularly for SMODELS. It uses the criticality notion to approximate costs, but this time criticality will be approximation of costs of finding a literal in a stable model of the input logic program. This section explains how method of calculating criticalities is applied for logic programs of answer set programming.

Logic programs are composed of rules. If an atom is in the stable model of the logic program (i.e., if the model gives interpretation value *true* for that atom), there must be at least one rule that has generated it; in other words every positive literal in the stable model has to be *grounded.* A rule generates an atom if the atom is in the head position and body of the rule is interpreted as true. For example the rule,

$$a \leftarrow b,\ not\ c \tag{3.1}$$

is a generator for a. However, it generates a only when we know b is true and c is false in the model. Under these conditions a must be true and we must add literal a to the candidate model.

The constraint rule,

$$\leftarrow b,\ d \tag{3.2}$$

only claims that in a stable model both b and d can not be true. So, constraint rules can not be generator rules for any literal. The rule below shows this statement more clearly.

$$\leftarrow b,\ not\ c \tag{3.3}$$

Let b is in the model. Rule (3.3) only is not sufficient to add literal c to the model. It claims that c must be true when b is true, but there must be at least one generator rule that have generated c even though we know b is true. If no rule can generate c in this situation, a conflict will raise and system can not find an answer set.

The only condition for a rule to generate the literal in its head is that its body must be true. So, if we see the rule's body as a *precondition* we can transform a rule into a *planning operator (action).*[1] The *effect* of this planning operator is the head atom. We can see the corresponding planning operator for rule (3.1) below.

[1] Lin and Reiter [18] define logic rules as planning actions in the context of defining semantics for logic programs using situation calculus.

$$a \xleftarrow{r_1} b,\ not\ c \quad \Longrightarrow \quad \begin{array}{l} \text{Operator}:\ \text{r}_1 \\ \text{Preconditions}:\ \text{b, not c} \\ \text{Effect}:\ \text{a} \end{array}$$

By considering generator rules of all literals of the program as actions, we can calculate the criticalities of all literals. Let the following generator rules for literal a form a portion of a logic program.

$$a \leftarrow b,\ not\ c \quad (rule_1) \tag{3.4}$$

$$a \leftarrow d \quad (rule_2) \tag{3.5}$$

$$a \leftarrow not\ e \quad (rule_3) \tag{3.6}$$

When we consider the above rules as planning actions, equations (3.7–3.8) are used for calculating the criticality of literal a (see Section 2.1.2 for calculating criticalities). Note that initial criticality value of a, $C(a, 0)$, is set to 1 like all the other literals' initial criticalities.

$$C(rule_1, n) = C(b, n-1) + C(not\ c, n-1) \tag{3.7}$$

$$C(rule_2, n) = C(d, n-1) \tag{3.8}$$

$$C(rule_3, n) = C(not\ e, n-1) \tag{3.9}$$

$$\frac{1}{C(a,n)} = \frac{1}{C(a,0)} + \frac{1}{C(rule_1,n)} + \frac{1}{C(rule_2,n)} + \frac{1}{C(rule_3,n)} \tag{3.10}$$

3.1 Criticality for Negation-as-failure

There are terms representing criticality value of a negative literal in equations (3.7) and (3.9). How can criticality of a not-atom be found? Further, what are the generator rules for a negative literal? There are no rules that have a not-atom in the head, so no rule can generate a negative literal in the sense of generating positive ones. In fact *negation-as-failure* means that if it is not possible to achieve an atom a, then *not a* is true. Considering the meaning of negation-as-failure and the properties of stable model semantics, one can define the necessary conditions needed for a negative literal to be in the stable model. If literal *not a* is in the stable model of a program, no generator rule can have a chance to generate a. Otherwise, there will be a conflict. So, none of the preconditions of generators of a can be satisfied. All bodies of those rules must be interpreted as false, so literal *not a* is interpreted as true in the stable model. In that sense, here is the only rule generating *not a* formed by taking all the generator rules (3.4–3.6) of a into consideration.[2]

$$not\ a \leftarrow (not\ b \vee c),\ not\ d,\ e \quad (rule_{not\ a}) \tag{3.11}$$

A generator rule of negative literal may contain *compound literals* with disjunction in the body as in rule (3.11) and this does not comply with the syntax of logic programs. However, its purpose is just to show the preconditions for generating *not a*. Based on the rule (3.11) criticality of literal *not a* can be calculated using the following equations.

$$C(rule_{not\ a}, n) = C((not\ b \vee c), n-1) + C(not\ d, n-1) + C(e, n-1) \tag{3.12}$$

$$\frac{1}{C(not\ a, n)} = \frac{1}{C(not\ a, 0)} + \frac{1}{C(rule_{not\ a}, n)} \tag{3.13}$$

Calculating criticality value of a negative literal is not different than calculation for a positive literal except the criticalities of compound literals with disjunction. It is sufficient that for a compound literal to be true at least one of the combined literals must be true since compound literal is a disjunction of basic literals. Achieving a compound literal as a whole should be easier than achieving each one of the basic literals in disjunctive form individually. Criticality values are monotonically decreasing and reside in the interval [0,1].[3] Calculating the criticality of compound literal $(not\ b \vee c)$ by the equation $C((not\ b \vee c), n) = C(not\ b, n) \times C(c, n)$ is a rational approximation, since the multiplication of two or more real numbers in the interval [0,1] is always smaller

[2] Body of the rule (3.11) is the inverse of the completion of literal a [15]. We can see the inverse of the completion of literal x as a generator rule for literal *not x*.

[3] Actually, if the initial criticality value a_0 is 1, then all the criticalities will be in this interval.

than each number or equal to the smallest one at least. This arithmetic property makes the following general equation rational according to the mentioned hardness property of compound literals.

$$C(a_1 \vee \ldots \vee a_n, n) = C(a_1, n) \times \ldots \times C(a_n, n) \tag{3.14}$$

CHAPTER 4

USING CRITICALITIES AS A HEURISTIC

Chapter 3 has showed how criticalities of literals of a logic program can be found in answer set programming. The main point of this thesis is to use the criticality information in answer set finder systems like SMODELS. That information is used at choice points as a heuristic and this chapter describes how it is achieved in SMODELS.

4.1 Heuristic of SMODELS

At choice points SMODELS choose literals to add to the candidate model. The pseudocode of the main algorithm of SMODELS (Figure 2.6) has the statement,

'*c := Choose an uncovered atom*',

representing choice points. But, this statement is nondeterministic; there still exists a question to be answered of which literal should we choose from uncovered ones. To answer this question SMODELS uses *heuristic* at the choice points.

SMODELS' heuristic has a basis about minimizing the remaining search space after a choice made [3]. Heuristic chooses the literal which reduce the search space maximally. To find the literal satisfying this criteria, SMODELS at a choice point selects every uncovered literal temporarily and adds it to the candidate model. The idea is trying to see what happens if SMODELS chooses that uncovered literal actually. After expanding the candidate model according to the input program, there will possibly be newly generated literals covered by the candidate model. The number of these newly generated literals is the heuristic score of the chosen literal. If the chosen literal is positive (i.e., an *atom*) then the score of the literal is called *positive score* of that atom. Similarly the score of negative literal of the same atom, an *not atom*, is also computed; it is *negative score* of that atom.

When atoms are sorted according to minimum of their positive and negative scores, SMODELS' heuristic chooses the maximum one. Formally; in the sort it is $A < B$ for two atoms A and B if,

$$min(PosScore(A),\ NegScore(A)) < min(PosScore(B),\ NegScore(B)).$$

If the minimums are equal, heuristic chooses the one with greater maximum of positive and negative scores [2].[1] So, $A < B$ if,

$$min(PosScore(A),\ NegScore(A)) = min(PosScore(B),\ NegScore(B)) \text{ and}$$
$$max(PosScore(A),\ NegScore(A)) < max(PosScore(B),\ NegScore(B)).$$

After choosing the atom, heuristic decides whether positive or negative literal of the atom will be added to the candidate model. If positive score is greater than the negative score for atom x system will add literal x; otherwise it will add literal $not\ x$.

Heuristic of SMODELS is a *dynamic heuristic* [2]. Heuristic function computes the score of a literal for a specific choice point by evaluating it at that situation exactly. At a different choice point heuristic evaluates the same literal again if it is still uncovered and may compute a different score this time. At every choice point it recomputes the scores which makes the heuristic dynamic.

Generally all answer set finder systems use heuristics. Heuristic function is one of the main points that affect the efficiency of finding answer sets. A good

[1] In fact, SMODELS does not find all of the sorted list. It only finds the atom with the maximum score according to the above criteria; this is the practical way of implementing it. Also, lookahead and heuristic procedures are integrated for optimization. Details can be found in [3].

choice can cause the system follow a fruitful path along which a stable model is found quickly in the search space. But, a bad choice can cause the system select a wrong path. It may take a lot of computation to understand the choice made leads to no stable model actually. The system may go deeper in levels by going through a lot of choice points after the initial bad choice made. At the end system faces with lots of conflicts and backtracking all the way to the initial choice point is necessary.

4.2 Calculating Criticalities in CSMODELS

In this thesis, a system called CSMODELS (Criticality SMODELS) is developed to use criticalities as a heuristic based on SMODELS system. SMODELS is chosen since it is the best-known and mostly used system. It is written in C++.

Chapter 3 describes the method of how criticality calculation can be applied to answer set programming; CSMODELS implements this method for the logic programs of SMODELS by modifying the source code of SMODELS. This section gives information about the implementation details of applying this method.

CSMODELS can calculate criticalities of all the literals of a logic program if all the rules in it are normal rules. But, SMODELS enlarges the syntax and semantics of logic programs by extended rules [3]. These extended rules are *choice, cardinality,* and *weight* rules.

Here is a choice rule.

$$\{a\} \leftarrow b,\ not\ c \tag{4.1}$$

A choice rule's head may not be true (i.e., may not be in the model) although its body is satisfied by the model unlike normal rules. In other words, its head may be or may not be in the stable model if its body is true. There can be more than one literal in the head and choice rules are used for representing subsets of the set of literals in the head [3].

One type of problem domain that choice rules are especially useful for is planning problems. They are usually used to represent planning actions in a logic program. A very basic planning action can be represented by the following rule.

$$\{open\_door\} \leftarrow door\_is\_unlocked,\ robot\_is\_near\_door \tag{4.2}$$

Note that a planning agent may execute an action or just do something else if the action's preconditions are satisfied. This is in agreement with the semantics of a choice rule when the body part represents preconditions and the head part is the planning action.

One can rewrite the choice rule using only normal rules. But, this translation introduces new atoms to the input program [3]. Choice rule (4.1) can be rewritten by two rules.

$$\begin{aligned} a &\leftarrow not\ c\_a,\ b,\ not\ c \\ c\_a &\leftarrow not\ a,\ b,\ not\ c \end{aligned} \tag{4.3}$$

Atom c_a is a new one. Since choice rules are common, CSMODELS accepts logic programs with choice rules and handles them. Actually, rules of type (4.1) in a logic program do not have to be transformed into rules (4.3) by the user explicitly. When CSMODELS finds a choice rule it treats the rule as if there are two normal rules of the form (4.3). So, when calculating the criticality of the rule generating literal a it adds the criticality value of literal $not\ c\_a$ in addition to b and $not\ c$. Criticality value of literal $not\ c\_a$ is calculated by using the only rule generating c_a; $c\_a \leftarrow not\ a,\ b,\ not\ c$.[2] If there are other choice rules that generate a, CSMODELS handles them in the same way and introduces new atoms for each one. Criticalities of new atoms due to choice rules will be used only in finding the criticality heuristic score in section 4.3, but original form of the choice rules are actually used during computation of answer sets. Answer sets in output will not contain these new atoms. So, CSMODELS can take advantage of choice rules without increasing the number of atoms or rules.

CSMODELS does not handle cardinality or weight rules. Weight rules are not very common in logic programs; in the experiments part of this work there are no representations using weight rules. However, cardinality rules are

[2] Criticality of literal $not\ c\_a$ is calculated using the generator rule formed by taking the inverse completion of c_a as described in Section 3.1.

common in logic programs and using them generally leads to a more efficient and compact representation for the problem. A cardinality rule is of the form,

$$a \leftarrow 1\{b,\ c,\ not\ d\}2 \tag{4.4}$$

and it is possible to rewrite cardinality rules using the normal logic rules without changing the answer sets of the problem. But, transformation may increase the number of rules or the number of atoms. A trivial translation is enumerating all the subsets of literals in the cardinality group (i.e., literals b, c and $not\ d$ in rule (4.4)) satisfying the limit constraints. Unfortunately, this enumeration can generate exponential number of rules in the number of literals; this makes the transformation impractical. Some of the experiments carried out with CSMODELS in this work had originally cardinality rules, but they are eliminated by additional normal rules without effecting the answer sets of the problem.

CSMODELS calculates the criticalities of literals just before making the first choice (i.e., at the first choice point) during the computation of an answer set. In the flow of main algorithm of SMODELS first choice points corresponds to the time just after the first expand procedure call finished its computation. According to SMODELS the candidate model found by expand procedure at the first choice point corresponds to *well-founded model (WFM)* of the program (see Section 2.2.2).

Well-founded model covers some of the atoms of the input program; the ones that have the interpretation value *undefined* are the uncovered atoms. So, there is no need to calculate the criticality values of the already covered atoms. Actually, we can assume the criticality value of a literal in well-founded model as 0, indicating that it is very easy to achieve that literal or it is already achieved. Similarly we can assume the criticality value of the inverse of that literal in the model as 1, indicating that it is impossible or very hard to achieve it.[3]

$$\text{If literal l} \in \text{WFM} \quad \Longrightarrow \quad \begin{array}{l} C(l) = 0 \\ C(not\ l) = 1 \end{array}$$

[3] Note that 1 is the maximum criticality value a literal can have.

Criticality calculation is an iterative process. When to stop iterating is computationally important for CSMODELS. Convergence rate of the criticality values is high; in other words they converge to their limiting values quickly [5]. However, computationally practical way of handling the stopping condition for iterations is to use a predefined *accuracy value* like RESISTOR system uses. CSMODELS uses a predefined decimal digit for accuracy check. At the end of each iteration it checks whether each criticality value is equal to its value at the previous iteration up to a decimal digit; if not iterations continue. Default value for the decimal digit is 1. So, for the default case CSMODELS continues iteration until first decimal digit for all values stabilize.

4.3 Criticality Heuristic Score

After calculating criticality values of all uncovered literals CSMODELS finds the *criticality heuristic scores* to select a literal at the first choice point. CSMODELS uses the same heuristic scores also if needed at the next choice points.

The system finds the criticality heuristic score using the criticality values of literals. CSMODELS temporarily selects every uncovered literal one by one and adds them to the candidate model just like SMODELS finds heuristic scores (see Section 4.1). Criticality heuristic scores are only calculated at the first choice point, so CSMODELS adds the temporarily selected literal to well-founded model.

Let x be an uncovered literal by the well-founded model and CSMODELS selects it to find the criticality heuristic score of literal x. The model at hand becomes WFM $\cup \{x\}$ after adding the literal x. CSMODELS expands the model at hand according to the logic program. This expansion will probably generate new literals that were uncovered by the well-founded model previously. New literals generated by the selection of x is represented by $NewLiterals(x)$. Let the set of newly generated literals for x is,

$$NewLiterals(x) = \{a,\ b, \ldots,\ not\ k,\ not\ l, \ldots\}. \tag{4.5}$$

CSMODELS finds the criticality heuristic score of literal x by the following equation.

$$\begin{aligned} Score(x) = \quad & C(a) + (1 - C(not\ a)) + \\ & C(b) + (1 - C(not\ b))\ + \ldots + \\ & C(not\ k) + (1 - C(k)) + \\ & C(not\ l) + (1 - C(l))\ + \ldots \end{aligned} \tag{4.6}$$

The purpose of criticalities was to approximate the hardness of achieving a literal in an answer set. Similarly, main purpose of the score of a literal is to approximate the hardness of choosing that literal at a choice point. We know choosing x generates literal a. Consider that if a were in the well-founded model without selecting x, its criticality values should be considered as 0. Also criticality value of literal *not* a should be 1. However, a is not in the well-founded model actually and criticality value of it is an effort to be given by the system when choosing x. The system has to make the criticality of a 0 in order to choose x. Also, it has to make the criticality of *not* a 1; that is why term $1 - C(not\ a)$ is added to the score. Repeating this as in the equation (4.6) for all the generated literals CSMODELS finds the total effort needed to choose x in order to approximate the hardnessof choosing.

If x is a positive literal, then its criticality heuristic score is called *positive criticality heuristic score* of atom x. If it is a negative literal (i.e., x is a not-atom), then the score is called *negative criticality heuristic score* of atom x. They are represented by $PosCScore(x)$ and $NegCScore(x)$ respectively.

CSMODELS takes the choice rules into consideration when finding heuristic scores. Consider that generated literal a in $NewLiterals(x)$ is a head atom of choice rule (4.1). During computation of criticalities, CSMODELS uses two normal rules (4.3) instead of that choice rule and calculates the criticality values of the newly introduced atom *c_a*. Now, criticalities of the new atom should be used when finding the heuristic score of literal x since generating literal a means generating literal *not c_a*. Just like the case of literal *not* k in $NewLiterals(x)$, the term $C(not\ c\_a) + (1 - C(c\_a))$ is added to score of x. If there are more choice rules generating a or other literals in $NewLiterals(x)$, criticality values of corresponding new atoms which are introduced when transforming the choice rules are added to the score in the same way.

4.4 Sorting Scores and Choosing a Literal

After finding heuristic scores of all uncovered literals, CSMODELS is really close to do the actual selection of a literal at the choice point. It has to select one literal among uncovered ones, add it to the candidate model, and call the expand procedure. If there is a need for another choice point, same procedure is applied. This cycle continues until an answer set is found, but note that there may be conflicts. In the case of a conflict CSMODELS backtracks to the previous choice point and is forced to add the negation of that literal chosen incorrectly. In fact, the main algorithm of CSMODELS is the same as of SMODELS except the heuristic used since its implementation is based on SMODELS.

In order to limit number of backtrackings CSMODELS selects hard to choose literals at first. Criticality heuristic scores are numerical approximations of "hardness to choose", so the system sorts the uncovered atoms according to their heuristic scores. Heuristic procedure of CSMODELS chooses the first uncovered atom in the sorted list. The next atom in the list becomes the one to be selected at the next choice point. However, CSMODELS at every choice point looks for an uncovered atom in the list starting from the place where it left off at the previous choice point. The reason is that the atom at the place where it left off can be already covered by the choices made previously.

CSMODELS' heuristic is not *dynamic* unlike SMODELS'. The system does not compute the criticality heuristic scores at every choice point. It only computes at the first choice point and forms the sorted list of atoms only once. During the computation of an answer set it uses the same sorted list at all choice points.

Sorting criteria is one of the things that effects the performance of the heuristic deeply. This is very natural because a change in the order of atoms will cause CSMODELS choose different atoms.

Main sorting criteria of CSMODELS, which is called *maxmin* sorting criteria, is the same as of SMODELS. Atoms are sorted according to minimum of their positive and negative criticality heuristic scores in descending order. Formally; for two uncovered atoms A and B, CSMODELS considers $A < B$ in the list if,

$$min(PosCScore(A), NegCScore(A)) < min(PosCScore(B), NegCScore(B)).$$

If the minimums are equal, criticality heuristic considers the one with a greater maximum of positive and negative scores. So, $A < B$ if,

$$\begin{aligned} &min(PosCScore(A), NegCScore(A)) = min(PosCScore(B), NegCScore(B)), \\ &max(PosCScore(A), NegCScore(A)) < max(PosCScore(B), NegCScore(B)) \end{aligned}$$

When the heuristic chooses an atom, the decision of adding positive or negative literal form of that atom is made according to its positive and negative heuristic score. If $PosCScore(x) > NegCScore(x)$ for an atom x, heuristic chooses the literal x; otherwise it chooses the literal *not* x. This ensures that CSMODELS selects the literal form of an atom that is hardest to choose.

Suppose that a, b, c and d are some uncovered atoms in a logic program and CSMODELS has calculated their heuristic scores as in Table 4.1. The sorted

Table 4.1: Criticality heuristic scores of some atoms of an example program

Atom	*PosCScore*	*NegCScore*
a	20.45	102.61
b	180.80	1.05
c	22.21	24.98
d	1.30	10.82

list of these atoms according to the maxmin sorting criteria is,

$$c,\ a,\ d,\ b.$$

If these atoms are on top of the whole sorted list, at the first choice point heuristic selects the atom c. Since $NegCScore(c) > PosCScore(c)$ ($24.98 > 22.21$), CSMODELS adds the literal *not* c to the candidate model.

Experiments showed that different sorting criteria can lead to more efficient way of finding answer set for some problems. Sorting atoms according to the sum of positive and negative heuristic scores in descending order is another criteria named *maxsum* sorting criteria. Formally; in the sorted list it is $A < B$ for two atoms A and B if,

$$PosCScore(A) + NegCScore(A) < PosCScore(B) + NegCScore(B).$$

This sorting criteria is useful especially for problems that have many answer sets. Experiments showed that using maxsum criteria helped CSMODELS to find an answer set in less time than maxmin sorting criteria in these kind of problems. If we sort the atoms in Table 4.1 according to maxsum, the list will be,

$$b,\ a,\ c,\ d.$$

Fortunately using maxsum criteria in problems that have not got many answer sets usually leads to conflict at the first choice point. This makes it possible for CSMODELS to behave in the same way for all problems. It uses the maxsum criteria first. If the first choice point made ends up with an immediate conflict, it changes the sorting criteria and sorts the uncovered atoms again according to the maxmin criteria; during the rest of the computation newly sorted list is used. But, if the first choice does not result with a conflict CSMODELS continues computation with maxsum criteria. So, to decide which sorting criteria to use occurs after first choice has been made.

Figure 4.1 shows the pseudocode of CSMODELS' main algorithm based on the SMODELS' algorithm in Figure 2.6.

```
function csmodels(CM, SL)   {CM: Candidate Model}
  {SL: Sorted List of uncovered atoms with next_to_choose index}
  CM := expand(CM)
  if conflict(CM) then
      if conflict occurred after first choice made then
          SL := sort(maxmin_criteria)
      end if
      return false
  else if no atom is undefined in CM then
      return true   {CM is a stable model}
  else
      if first choice point then   {CM is well-founded model}
          Calculate criticalities
          Find criticality heuristic scores of uncovered atoms
          SL := sort(maxsum_criteria)
      end if
      c := criticality_choose(SL, CM)
      if csmodels(CM ∪ {c}, SL) then
          return true
      else
          return csmodels(CM ∪ {not c}, SL)
      end if
  end if


function criticality_choose(SL, CM)
  c := Find first atom in SL not covered by CM starting from next_to_choose
  Update next_to_choose index to point to the next atom after c
  return c
```

Figure 4.1: Main algorithm of CSMODELS

CHAPTER 5

EXPERIMENTAL RESULTS

CSMODELS has been tested to find out its performance. Tests are from common application domains of answer set programming. We compare CSMODELS with SMODELS. Logic programs with variables are first grounded by LPARSE. Since both CSMODELS and SMODELS take the same grounded logic program for the experiments, grounding time is the same for both.

The main measure for the tests is the duration which states how long the search for an answer set took in seconds. The durations are taken from the output messages of CSMODELS which are given in the same way SMODELS does. Another measure for comparison is the number of choice points. This number shows how many times a system used its heuristic to find an answer.

Planning domain is one of the main areas where answer set programming systems are used. Several well-known planning problems are tested. These are Towers of Hanoi, Simple Robot-box domain, and Blocks-world planning. In addition to planning problems, several tests of colorability and n-queens problems have been done. Tables 5.1, 5.3 and 5.5 report the search times (in seconds) of CSMODELS and SMODELS for all the problem instances. Tables 5.2, 5.4 and 5.6 show the number of choice points for the same problems. All of the logic programs constituting the test problems can be found in Appendix A.

Tests for planning problems have been performed on a 200 MHz Pentium computer running Linux 2.4.5; for colorability and n-queens problems a 733 MHz Pentium III computer running Linux 2.2.19 has been used.

Towers of Hanoi problem with 4 disks has been tested. Tables 5.1 and 5.2 shows the search time and the number of choice points for this test problem. Initial logic program written for representing the problem has cardinality rules. Since cardinality rules are not handled by CSMODELS they are rewritten as normal rules without effecting the answer sets of the problem (see Section 4.2). Cardinality rules are especially suitable and are usually used for representing no-concurrency (i.e., actions are not allowed to occur concurrently) in planning problems. Here is a rule with cardinality literal representing no-concurrency in Towers of Hanoi problem stating that only one disk can move at a step.

$$\leftarrow 2\ \{move(D, From, To, S) : disk(D) : peg(From) : peg(To)\},\ step(S) \quad (5.1)$$

In the logic program used for testing the purpose of this rule, i.e. no-concurrency, is achieved by rewritting it as the following normal rules according to the nature of the problem.

$$\begin{aligned} &\leftarrow changes(D, S),\ changes(D1, S),\ disk(D; D1),\ step(S),\ D < D1 \\ &\leftarrow on(D, P, T),\ on(D, P1, T),\ disk(D),\ peg(P; P1),\ time(T),\ P < P1 \end{aligned} \quad (5.2)$$

Also, in the other planning problems similar changes have been made by rewritting the cardinality rules.

The logic program representing the Simple Robot–Box domain is based on the reference [6]. There are rooms and boxes in this domain. Doors connect the rooms and they can be open or closed. A planning problem can be formed in this domain to find the needed actions for the robot in order to move the boxes and achieve a desired configuration of them. These actions can be moving, opening doors, loading, and carrying boxes. The specific problem used for testing can be found in the representation of the domain in Appendix A and the experimental results of it is shown in Tables 5.1 and 5.2.

Three different domain representations have been tested for the Blocks-world planning problems. In this problem, there are boxes on a table. A box is

Table 5.1: Search time comparison for planning problems (in seconds)

	SMODELS	CSMODELS
Hanoi	4.630	5.540
Robot–Box	39.940	40.800
BW__1 (11)	46.350	36.690
BW__2 (11)	73.400	57.040
BW__3 (15)	38.500	37.390
BW__3 (17)	77.330	61.550
BW__3 (19)	174.660	90.620

only moveable if it is on top of a box pile or on table with no box on it. A planning problem instance is formed to find the necessary moves of boxes from an initial configuration to a desired one. One representation is from the reference [13]. In this domain representation concurrency is allowed. Three different problem instances from the reference [14] are tested; they are problems with 15, 17 and 19 blocks.[1] In Tables 5.1 and 5.2 rows BW__3 (15), BW__3 (17) and BW__3 (19) show the results for the problem instances with 15, 17 and 19 blocks respectively. Another representation is the one used in reference [19]. A problem instance with 11 blocks from the same reference has been used.[2] Unlike the first representation, this one does not allow concurrent moves; problem instance has a solution with 9 moves and is shown by the rows BW__2 (11) in Tables 5.1 and 5.2. Third representation is in fact the first representation with no concurrent moves allowed. The problem instance with 11 blocks has been tested with the third representation also; the rows BW__1 (11) show the results in Tables 5.1 and 5.2.

The results in Table 5.1 show that for small size problems like Towers of Hanoi and Simple Robot-Box domains, CSMODELS' performance is almost the same as SMODELS' performance. When the sizes of the problems become larger criticality heuristic of CSMODELS helps the system solve the problems more efficiently than the heuristic of SMODELS does. This claim is supported

[1] They correspond to I. Niemela's bw-large.c, bw-large.d and bw-large.e problems, respectively [14].

[2] This problem instance corresponds to E. Erdem's P4 in [19].

Table 5.2: Number of choice points for planning problems

	SMODELS	CSMODELS
Hanoi	12	8
Robot–Box	11	8
BW__1 (11)	8	5
BW__2 (11)	7	5
BW__3 (15)	29	7
BW__3 (17)	27	10
BW__3 (19)	4111	6

especially by the problem instances (15, 17 and 19) of the Blocks-world representation BW__3. By increasing the number of blocks from 15 to 19 the size of the problem becomes larger and also the performance difference between CSMODELS and SMODELS becomes more clear. These problems have many answer sets since concurrent moves of blocks are allowed; CSMODELS used maxsum sorting criteria for these 3 problems (see Section 4.4). Compared to them the other 4 planning problems (first four rows in Table 5.1) have much less number of answer sets; CSMODELS switch to the maxmin sorting criteria from maxsum during the computation of an answer set.

Table 5.2 indicates that CSMODELS generally finds an answer set of a problem in less number of choice points than SMODELS does. The reduction in the number of choice points is consistent with the performance gains.

Other than the planning domain, colorability and n-queens problems have been used in testing. For colorability problem there is a graph and an assignment of one of 4 colors to each vertex of this graph has to be found (4-colorability problem instances are used in experiments). However, colors of the vertices that are connected with an edge have to be different. n-queens problem is placing n queens on a $n \times n$ chess board so that no queen attacks any other queen. The logic programs for these problems are based on I. Niemela's representations.[3] For both of the problems, using choice rules make their representations more

[3] The logic program for n-queens problem is actually from [13]; the one for colorability problem is adapted from [14] by rewriting using choice rules.

Table 5.3: Search time comparison for colorability problem (in seconds)

	p100	*p300*	*p600*	*p1000*	*p3000*
SMODELS	0.530	3.530	12.380	35.210	305.060
CSMODELS	0.540	3.050	10.080	26.070	230.370

Table 5.4: Number of choice points for colorability problem

	p100	*p300*	*p600*	*p1000*	*p3000*
SMODELS	32	89	177	310	860
CSMODELS	38	79	162	235	787

compact and efficient. Here is a choice rule in n-queens,

$$1\ \{q(X,Y) : d(Y)\}\ 1 \leftarrow d(X) \tag{5.3}$$

which represents the fact that there can be exactly one queen in each row of the board. The choice rule has cardinality numbers in the head for expressing only one queen. One may think these numbers in a choice rule can be problem for CSMODELS since during grounding, LPARSE converts it into group of rules in which there are cardinality rules. A 3×3 board problem has been considered for the sake of simplicity. LPARSE generates the following ground rules for the choice rule 5.3.

$$\{q(1,3),\ q(1,2),\ q(1,1)\} \tag{5.4}$$

$$\leftarrow tmp1 \tag{5.5}$$

$$tmp1 \leftarrow 3\ \{not\ q(1,3),\ not\ q(1,2),\ not\ q(1,1)\} \tag{5.6}$$

$$\leftarrow tmp2 \tag{5.7}$$

$$tmp2 \leftarrow 2\ \{q(1,3),\ q(1,2),\ q(1,1)\} \tag{5.8}$$

CSMODELS does not handle cardinality rules but the ones generating the temporary atoms $tmp1$ and $tmp2$ are cardinality rules. However, the constraint rules (5.5) and (5.7) cause the expand procedure at the first call to interprete both $tmp1$ and $tmp2$ as false. So, well-founded model will cover both of them and there is no need for CSMODELS to calculate their criticalities by handling the

Table 5.5: Search time comparison for n-queens problem (in seconds)

	8×8	*18×18*	*20×20*	*22×22*
SMODELS	0.020	1.830	9.360	117.730
CSMODELS	0.030	1.530	5.490	31.170

Table 5.6: Number of choice points for n-queens problem

	8×8	*18×18*	*20×20*	*22×22*
SMODELS	3	302	1483	16156
CSMODELS	7	152	571	2401

cardinality rules. CSMODELS accepts all kinds of choice rules with or without cardinality constraints in the head.

Several tests have been done from small size problem instances to large size ones for colorability and n-queens problems. Tables 5.3 and 5.4 show the experimental results for colorability problem. Test graphs (*p100–p3000*) from [14] have been used. Tables 5.5 and 5.6 show the experimental results for n-queens problems starting from 8×8 up to 22×22 instances.

Tables 5.3 and 5.5 show that CSMODELS is more efficient than SMODELS for colorability and n-queens problems. For larger size problem instances the performance gains obtained by CSMODELS are more significant.

CHAPTER 6

CONCLUSION

Answer set programming is a new paradigm based on logic programming. It has been applied to many application domains having basically search and decision problems [13]. Declarative way of writing input logic programs and nonmonotonic characteristic of the stable model semantics applied for finding models make answer set programming expressive and advantageous compared to classical logic programming.

There are some systems that can be used for finding answer sets of logic programs. The main algorithm of all the systems is almost the same. They utilize some heuristics during the computation of answer sets. Heuristics affect the search path that the system explores among the whole search space. As some paths lead to efficient solution finding and some not, heuristics are one of the key points for the performance of the system.

In this work, a new heuristic for answer set programming has been developed. The resulting system called CSMODELS uses this new heuristic. The main idea of the heuristic comes from hierarchical planning. Criticality, which is introduced for generating abstraction hierarchies, is used as the new heuristic. Criticality values approximate the difficulty of achieving a literal. A method for applying criticality calculation for answer set programming has been developed. The experimental results indicated that CSMODELS outperforms SMODELS in terms of efficiency. The performance difference becomes more significant when the size of the problem becomes bigger. Generally, the results show that this new heuristic is a promising one for answer set programming.

The choice rules from the extended rules of SMODELS' new versions are supported by CSMODELS. However, cardinality and weight rules are not supported. Using them usually makes the representation of the problems more compact. SMODELS benefits from these extended rules by improved search times. Calculating criticality of a literal generated by a cardinality or weight rule is not trivial. Rewriting them may generate many normal rules. Just enumerating the possible subsets of the literals used in a cardinality or a weight rule according to the cardinality or weight constraints will generate exponential number of normal rules. Approximating the criticality value of a literal generated by these extended rules can be a solution. Handling cardinality and weight rules in CSMODELS is a topic of future work.

Heuristics for answer set programming are like heuristics for satisfiability solvers. Although there has been much work done about them, there are not much about the heuristics for answer set programming [2]. New heuristics will help systems to be more efficient and make answer set programming more applicable.

REFERENCES

[1] V. Lifschitz. Answer set planning. In *International Conference on Logic Programming*, pages 23–37, 1999.

[2] W. Faber, N. Leone, and G. Pfeifer. A comparison of heuristics for answer set programming. In *Proc. of the 5th Dutch-German Workshop on Non-monotonic Reasoning Techniques and their Applications (DGNMR 2001)*, pages 64–75, 2001.

[3] P. Simons. Extending and implementing the stable model semantics. Research Report 58, Helsinki University of Technology, Helsinki, Finland, 2000.

[4] F. Giunchiglia. Using ABSTRIPS abstractions – where do we stand? Technical Report 9607-10, IRST, Trento, Italy, 1996.

[5] A. Bundy, F. Giunchiglia, R. Sebastiani, and T. Walsh. Calculating criticalities. *Artificial Intelligence*, 88(1-2):39–67, 1996.

[6] C. A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243–302, 1994.

[7] F. Giunchiglia, A. Villafiorita, and T. Walsh. Theories of abstraction. *AI Communications*, 10(3–4):167–176, 1997.

[8] C. A. Knoblock. Abstracting the Tower of Hanoi. In *Working Notes of AAAI-90 Workshop on Automatic Generation of Approximations and Abstractions*, pages 13–23, 1990.

[9] D. E. Smith and M. A. Peot. A critical look at Knoblock's hierarchy mechanism. In *Proc. of 1st International conference Artificial Intelligence Planning Systems (AIPS-92)*, pages 307–308, 1992.

[10] C. Backstrom and P. Jonsson. Planning with abstraction hierarchies can be exponentially less efficient. In *Proc. of the 14th International Joint Conference on Artificial Intelligence*, pages 1599–1604, 1995.

[11] M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In R. Kowalski and K. Bowen, editors, *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080. The MIT Press, 1988.

[12] G. Antoniou. *Nonmonotonic Reasoning.* The MIT Press, 1996.

[13] I. Niemela and M. Trunszczynski. Answer-set programming: a declarative knowledge representation paradigm. In Lecture notes of ESSLLI 2001 Summer School, 2001.

[14] I. Niemela. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.

[15] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.

[16] A. van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[17] T. Syrjanen. LPARSE 1.0 user's manual. Available at *http://www.tcs.hut.fi/Software/smodels/lparse.ps.*

[18] F. Lin and R. Reiter. Rules as actions: A situation calculus semantics for logic programs. *Journal of Logic Programming*, 31(1-3):299–330, 1997.

[19] E. Erdem. Applications of logic programming to planning: Computational experiments. Unpublished draft, http://www.cs.utexas.edu/users/esra/papers.html, 1999.

APPENDIX A

Problem Representations Used in Experiments

A.1 Towers of Hanoi

```
% Towers of Hanoi with 4 disks
%
% lparse -c n=15 han4.lp | {csmodels|smodels}

time(0..n).
step(0..n-1).

peg(peg1).
peg(peg2).
peg(peg3).
disk(1..4).

{ move(D,F,T,S) } :- disk(D), peg(F), peg(T), step(S),
                     on(D,F,S), top(D,F,S).

on(D,T,S+1) :- disk(D), peg(T), peg(F), step(S),
               move(D,F,T,S).

top(D,T,S+1) :- disk(D), peg(T), peg(F), step(S),
                move(D,F,T,S).

changes(D,S) :- move(D,F,T,S), disk(D), peg(F), peg(T), step(S).

% frame axiom
on(D,P,S+1) :- on(D,P,S), not changes(D,S), step(S), disk(D), peg(P).
```

```
% dont make a move from one peg to the same one.
:- move(D,F,T,S), disk(D), peg(F), peg(T), step(S), F=T.

% only one disk can move at a step
:- changes(D,S), changes(D1,S), D < D1, disk(D;D1), step(S).
:- on(D,P,T), on(D,P1,T), disk(D), peg(P;P1), time(T), P < P1.

% for every peg there must be at most one disk on top.
{ top(D,P,T):disk(D) } :- peg(P), time(T).

% there cannot be a disk smaller than the disk on top of a peg.
:- top(D,P,T), peg(P), disk(D), time(T), on(D1,P,T), disk(D1), D1 < D.

% a disk can only be on top of a peg if it is on that peg.
:- not on(D,P,T), top(D,P,T), disk(D), peg(P), time(T).

% if there is a disk on a peg there must be a disk on top of that peg.
:- not top(1,P,T), not top(2,P,T), not top(3,P,T), not top(4,P,T),
   on(Dany,P,T), disk(Dany), peg(P), time(T).

% initial condition
on(4, peg1, 0). on(3, peg1, 0). on(2, peg1, 0). on(1, peg1, 0).

% goal
compute 1 { on(4, peg3, n), on(3, peg3, n), on(2, peg3, n), on(1, peg3, n) }.
```

A.2 Simple Robot-Box Domain

```
%
% Robot-box domain simplified version of the domain in Bacchus and Yang's paper
%
% lparse -c n=13 rb.lp | {csmodels|smodels}

time(0..n).
step(0..n-1).

%
% open_dr action
%
{ open_dr(D,S) } :- door_closed(D,S), at(R,S), connects(D,R,R1),
                    room(R;R1), door(D), step(S).

door_open(D,S+1) :- open_dr(D,S), door(D), step(S).
```

```
% Enumerate actions for representing no-concurrency without using cardinality
% rules. 05/03/2002
actions(1..6).
act_occured(1,S) :- open_dr(D,S), door(D), step(S).

abndoor(D,S) :- open_dr(D,S), door(D), step(S).

%
% close_dr action
%
{ close_dr(D,S) } :- door_open(D,S), at(R,S), connects(D,R,R1),
                     room(R;R1), door(D), step(S).

door_closed(D,S+1) :- close_dr(D,S), door(D), step(S).

act_occured(2,S) :- close_dr(D,S), door(D), step(S).

abndoor(D,S) :- close_dr(D,S), door(D), step(S).

%
% load box action
%
{ load(B,S) } :- inroom(B,R,S), at(R,S), not holding_abox(S), box(B), room(R),
                 step(S).

holding(B,S+1) :- load(B,S), box(B), step(S).

holding_abox(S+1) :- load(B,S), box(B), step(S).

act_occured(3,S) :- load(B,S), box(B), step(S).

abninroom(B,S) :- load(B,S), box(B), step(S).

%
% unload box action
%
{ unload(B,S) } :- holding(B,S), box(B), step(S).

inroom(B,R,S+1) :- unload(B,S), at(R,S), box(B), room(R), step(S).

act_occured(4,S) :- unload(B,S), box(B), step(S).
```

```
abnholding(B,S) :- unload(B,S), box(B), step(S).

abnholding_abox(S) :- unload(B,S), box(B), step(S).

%
% go through the door action. This action is not for carrying boxes.
%
{ go_thru_dr(D,R1,R2,S) } :- at(R1,S), door_open(D,S), connects(D,R1,R2),
                  not holding_abox(S), door(D), room(R1;R2), step(S).

at(R2,S+1) :- go_thru_dr(D,R1,R2,S), room(R1;R2), door(D), step(S).

act_occured(5,S) :- go_thru_dr(D,R1,R2,S), door(D), room(R1;R2), step(S).

abnat(R1,S) :- go_thru_dr(D,R1,R2,S), door(D), room(R1;R2), step(S).

%
% carry box through the door action.
%
{ carry_thru_dr(B,D,R1,R2,S) } :- holding(B,S), at(R1,S), door_open(D,S),
                 box(B), door(D), room(R1;R2), step(S), connects(D,R1,R2).

at(R2,S+1) :- carry_thru_dr(B,D,R1,R2,S), box(B), door(D), room(R1;R2), step(S).

act_occured(6,S) :- carry_thru_dr(B,D,R1,R2,S), box(B), door(D), room(R1;R2),
                    step(S).

abnat(R1,S) :- carry_thru_dr(B,D,R1,R2,S), box(B), door(D), room(R1;R2),
               step(S).

%
% FRAME AXIOMS
%
at(R,S+1) :- at(R,S), not abnat(R,S), room(R), step(S).
holding(B,S+1) :- holding(B,S), not abnholding(B,S), box(B), step(S).
holding_abox(S+1) :- holding_abox(S), not abnholding_abox(S), box(B), step(S).
door_open(D,S+1) :- door_open(D,S), not abndoor(D,S), door(D), step(S).
door_closed(D,S+1) :- door_closed(D,S), not abndoor(D,S), door(D), step(S).
inroom(B,R,S+1) :- inroom(B,R,S), not abninroom(B,S), box(B), room(R),
                   step(S).

connects(D,R2,R1) :- connects(D,R1,R2), room(R1;R2), door(D).
```

```
% robot can not be in two rooms at the same time.
:- at(R,T), at(R1,T), time(T), room(R;R1), R < R1.

% no concurrency
:- act_occured(A,S), act_occured(A1,S), step(S), actions(A;A1), A1 > A.
:- open_dr(D,S), open_dr(D1,S), door(D;D1), step(S), D < D1.
:- close_dr(D,S), close_dr(D1,S), door(D;D1), step(S), D < D1.
:- load(B,S), load(B1,S), box(B;B1), step(S), B < B1.

%
% problem domain literals
%
room(r1;r2;r3;r4;r5;r6;r7).
door(d12). connects(d12,r1,r2).
door(d25). connects(d25,r2,r5).
door(d26). connects(d26,r2,r6).
door(d23). connects(d23,r2,r3).
door(d56). connects(d56,r5,r6).
door(d36). connects(d36,r3,r6).
door(d67). connects(d67,r6,r7).
door(d34). connects(d34,r3,r4).
door_open(d12,0). door_closed(d25,0). door_closed(d26,0). door_closed(d23,0).
door_closed(d56,0). door_open(d36,0). door_closed(d67,0). door_closed(d34,0).

box(box1).
box(box2).
at(r5,0).
inroom(box1,r2,0).
inroom(box2,r3,0).

%
% goal condition
%
compute 1 { inroom(box1,r4,n), inroom(box2,r1,n) }.
```

A.3 Blocks-world Domain I

```
% Blocks-world domain representation from Niemela & Truszczynski ESSLLI 2001
% summer school handouts
%
% This domain is used for 4 tests in the experimental results; corresponding
% rows in result tables are BW_1 (11), BW_3 (15), BW_3 (17) and BW_3 (19).
% In the tests in BW_3 rows concurrency is allowed, so parts representing
```

```
% no-concurrency are deleted for them. No-concurrency is allowed for test BW_1.


% An example of initial and goal conditions
% Blocks
block(b0).
block(b1).
block(b2).
block(b3).
block(b4).
block(b5).
block(b6).
block(b7).
block(b8).
block(b9).
block(b10).

% Initial configuration
on(b0, floor,0). on(b3, floor,0). on(b5, floor,0). on(b1, b0,0).
on(b2, b1,0). on(b4, b3,0). on(b9, b4,0). on(b10, b9,0). on(b6, b5,0).
on(b7, b6,0). on(b8,b7,0).

% Goal conditions
goal(T) :- time(T), on(b9, floor, T), on(b3, floor, T),
on(b5,floor,T), on(b4, b9, T), on(b0, b4, T),
on(b8, b3, T), on(b7, b8, T), on(b6, b5, T),
on(b10, b6, T), on(b2, b10,T), on(b1, b2, T).

% Domain predicates
time(0..n).

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

% Auxiliary predicates
object(floor).
object(X) :- block(X).

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

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

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

% Select a moveop provided that preconditions hold
{ moveop(X,Y,T) } :-
        time(T), block(X), object(Y),
        X != Y, on_something(X,T),
        available(Y,T),
        not covered(X,T),
        not covered(Y,T).

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

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

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

% Concurrent moveop instances are allowed
% This part is uncommented for tests BW_3.
%:- moveop(X,Y,T), moveop(X,Y1,T), block(X), time(T), object(Y;Y1), Y < Y1.

% The destination cannot be moving
:- block(X), object(Y), time(T),
        moveop(X,Y,T),
        moving(Y,T).

% No two blocks moved onto the same block
% This part is uncommented for tests BW_3.
%:- moveop(X,Y,T), moveop(X1,Y,T), block(Y), time(T), block(X;X1), X < X1.

% no concurrency
% This part is commented for tests BW_3.
moveop1(X,T) :- moveop(X,Y,T), block(X), object(Y), time(T).
```

```
:- moveop1(X,T), moveop1(X1,T), block(X;X1), time(T), X < X1.
moveop2(Y,T) :- moveop(X,Y,T), block(X), object(Y), time(T).
:- moveop2(Y,T), moveop2(Y1,T), object(Y;Y1), time(T), Y < Y1.

% Exclude models where the goal has not been reached.
:- not goal.
goal :- time(T), goal(T).
goal(T2) :- nextstate(T2,T1), goal(T1).

% Optimizations
% Stop when the goal has been reached
:- block(X), object(Y), time(T),
        moveop(X,Y,T),
        goal(T).

% No move from floor to floor
:- block(X), time(T),
        moveop(X,floor,T), on(X,floor,T).

% No move on something and then to floor
:- nextstate(T2,T1), block(X), object(Y),
        moveop(X,Y,T1), moveop(X,floor,T2).
```

A.4 Blocks-world Domain II

```
% Blocks-world domain representation from E.Erdem's paper.
%
% This representation is used for the test BW_2 (11).



% non(B,L,T) represents the classical negation of on(B,L,T).


% effect of moving a block


on(B,L,T1) :- block(B), location(L), move(B,L,T), next(T,T1).


% a block can be moved only when it's clear


:- location(L), block(B), block(B1), time(T), move(B,L,T), on(B1,B,T).


% any two blocks cannot be on the same block at the same time


:- on(B1,B,T), on(B2,B,T), time(T), block(B;B1;B2), B1 < B2.
```

```
% wherever a block is, it's not anywhere else

non(B,L1,T) :- time(T), location(L1), location(L), block(B), on(B,L,T),
not eq(L,L1).

% every block is supported by the table

supported(B,T) :- block(B), time(T), on(B,table,T).
supported(B,T) :- block(B), block(B1), time(T), on(B,B1,T), supported(B1,T), not eq(B,B1).
:- block(B), time(T), not supported(B,T).

% no actions are executed at last time

:- block(B), location(L), move(B,L,lasttime).

% no concurrency

move1(B,T) :- move(B,L,T), block(B), location(L), time(T).
:- move1(B,T), move1(B1,T), block(B;B1), time(T), B < B1.
move2(L,T) :- move(B,L,T), block(B), location(L), time(T).
:- move2(L,T), move2(L1,T), block(L;L1), time(T), L < L1.

% inertia

on(B,L,T1) :- location(L), block(B), on(B,L,T), not non(B,L,T1), next(T,T1).

% initial values and actions are exogenous

{ on(B,L,0), non(B,L,0) } :- block(B), location(L).

{move(B,L,T)} :- block(B), location(L), time(T).

% consistency

:- block(B), location(L), time(T), on(B,L,T), non(B,L,T).

% auxiliary predicates

time(0..lasttime).
next(T,T+1) :- time(T), lt(T,lasttime).

location(L) :- block(L).
location(table).
```

```
hide on(B,L,T).
hide non(B,L,T).
hide supported(B,T).
```

A.5 Colorability Problem

```
% Colorability problem
% Rewritten based on Niemela's repr. using a choice rule.
% Niemelas's graphs have been used for testing.
%
% cat color.lp ../niemela/graphs/p1000 | lparse -c c=4 | {csmodels|smodels}


col(1..c).

1 { color(V,C):col(C) } 1 :- vertex(V).

:- arc(V1,V2), col(C), color(V1,C), color(V2,C).
```

A.6 N-queens Problem

```
%
% From lecture notes of Niemela an Truszczynski in ESSLLI 2001
%
% lparse -c n=8 queen.lp | {csmodels|smodels}


d(1..n).

1 { q(X,Y):d(Y) } 1 :- d(X).

1 { q(X,Y):d(X) } 1 :- d(Y).

:- d(X;Y;X1;Y1), q(X,Y), q(X1,Y1), X < X1, Y != Y1, abs(X-X1) == abs(Y-Y1).
```