QUAECUMQUE VERA

– Motto of The University of Alberta

University of Alberta

Logic Programming with Constraints

by

Guohua Liu

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

Doctor of Philosophy

Department of Computing Science

Spring 2010
Edmonton, Alberta

Examining Committee

Randy Goebel, Department of Computing Science

Martin Mueller, Department of Computing Science

Liyan Yuan, Department of Computing Science

Jia-Huai You, Department of Computing Science

Marek Reformat, Department of Electrical and Computer Engineering

Tran Cao Son, Department of Computer Science, New Mexico State University

To my parents Shufang Wang and Zhiyong Liu and my sister Guojing Liu
for their measureless love and patience

Abstract

Answer set programming (ASP), namely logic programming under the answer set semantics, provides a promising constraint programming paradigm, due to its simple syntax, high expressiveness, and effective computational properties. This programming paradigm can be viewed as a variant of SAT. A main drawback of ASP, like SAT, is that the framework is primarily formulated for reasoning with individuals, where each is represented by an atom in the traditional sense. Recently, ASP has been extended to include constraints to facilitate reasoning with sets of atoms in an explicit manner. These constraints include weight constraints, aggregates, and abstract constraints. Logic programs with these constraints are referred to as weight constraint, aggregate and abstract constraint programs.

We investigate the properties and computations of weight constraint, aggregate, and abstract constraint programs, respectively. For properties, we improve the formulation of loop formulas for weight constraint and aggregate programs, propose loop formulas for abstract constraint programs, and characterize strong equivalence of these programs. For computations, we refine the existing constraint propagation mechanism in the computation of weight constraint programs and advocate a new approach to computing aggregate programs.

Acknowledgements

I am indebted to my supervisor Dr. Jia-Huai You for providing me with the opportunity to work with him. I have benefited a lot from his guidance on how to motivate, carry out and present a research, and his continuous support made it possible for me to try various ideas. Without his help, this dissertation would not have been possible.

I am grateful to Drs. Martin Mueller, Randy Goebel, Li Yan Yuan, Tran Cao Son, and Marek Reformat for their detailed review of my thesis and valuable suggestions on the revision.

I appreciate the administrative and technical support teams of the Computing Science Department including Edith Drummond, Sunrose Ko, Linda Jiao, and Broderick Wood. They created a really effective and friendly work environment.

Special thanks to many friends, including Baochun Bai, Zhaochen Guo, Yisong Wang, and Qian Yang for their help and encouragement during my study and the preparation for the defense.

Finally, I would like to thank my parents Shufang Wang and Zhiyong Liu and my sister Guojing Liu for their measureless love and patience.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Answer set programming

An intelligent agent must have the ability to acquire knowledge and reason with it. In order for an artificial intelligence (AI) system to possess these abilities, a computer readable language for knowledge representation and reasoning (KR) is needed. *Logic programming*, which uses logic as the basis for a programming language, has been a powerful tool for KR.

Initially, logic programming focuses on a subset of classical logic, called Horn logic. A Horn clause is an implication with an atom in the head and a conjunction of atoms in the body. This allows fast and simple inferences through resolution. But it was quickly realized that Horn logic was not adequate for common-sense reasoning. The reason is that Horn logic is *monotonic*: a conclusion entailed by a body of knowledge stubbornly remains valid no matter what additional knowledge is added, but common-sense reasoning is *nonmonotonic*: conclusions made with currently available knowledge may be withdrawn in the presence of additional knowledge. This led to the development of the field of *nonmonotonic logic*, and several nonmonotonic logics such as *circumscription*, *default logic*, *auto-epistemic logic*, and *nonmonotonic modal logics* were proposed. The AI journal special issue of 1980 (volume 13, number 1 and 2) contained initial articles on some of these logics.

The *closed world assumption* in databases was then imported to logic programming to make it able to perform nonmonotonic reasoning [13]. The basic idea of the closed world assumption in this context is that a logic program contains all the positive information about the objects in its domain, and a negative conclusion is drawn if the proof of its positive counterpart failed. This way of generating negative information is referred to as *negation-as-failure*. An operator `not` was introduced to represent the negative information.

Logic programming under the *answer set semantics*, coined as *answer set programming*

(ASP), was proposed in late 1980s as an alternative language for knowledge representation and nonmonotonic reasoning. In ASP, a problem is represented by a logic program and the *answer sets* (also called *stable models*) of the program correspond to the solutions to the problem. Intuitively, an answer set of a program is a set of atoms that satisfies every rule of the program and each atom in it has a *non-circular justification* by the program.

As an example, consider the 3-coloring problem, which is to color a graph with three colors: red, blue, and yellow, so that any two nodes connected by an edge have different colors. A typical logic program in ASP is as follows.

Example 1.1.

$$\begin{aligned} col(X, red) &\leftarrow node(X), \texttt{not}\ col(X, blue), \texttt{not}\ col(X, yellow) \\ col(X, blue) &\leftarrow node(X), \texttt{not}\ col(X, red), \texttt{not}\ col(X, yellow) \\ col(X, yellow) &\leftarrow node(X), \texttt{not}\ col(X, red), \texttt{not}\ col(X, blue) \\ &\leftarrow edge(X, Y), color(C), col(X, C), col(Y, C) \end{aligned}$$

The first rule in Example 1.1 says that a node is colored by red if it is not colored by blue or yellow. The second and third rules are similar. These rules together express all of the possible colorings of the nodes in the graph. The fourth rule guarantees that adjacent nodes get different colors.

Along with the facts that specify the nodes, edges and colors, such as, $node(a)$, $node(b)$, $node(c)$, $edge(a, b)$, $color(red)$, $color(blue)$ and $color(yellow)$, an answer set of this program looks like

$$\{col(a, red),\ col(b, blue),\ col(c, yellow),\ \ldots\}.$$

It means that coloring the nodes a, b, c with the colors red, blue, and yellow, respectively, is a valid solution to the problem. □

Some programs may have no answer sets. This means that the corresponding problem has no solution.

In Example 1.1, the upper case letters are variables. Variables are allowed in implemented ASP systems. But the semantics, properties and computations of a program are studied only on *ground* programs, which are obtained by instantiating the non-ground programs with the constants in the underlying language.[1] In order to make sure that such a

[1]Recently, there have been approaches to the computation of programs with variables based on partial or dynamic grounding [56].

program is finite, no function symbols are allowed. In the rest of this document, we only discuss ground programs.

In general, a logic program consists of rules of the form

$$l_0 \leftarrow l_1, \ldots, l_m, \texttt{not}\ l_{m+1}, \ldots, \texttt{not}\ l_n \tag{1.1}$$

where each l_i, $(0 \leq i \leq n)$ is an atom. Intuitively, a rule of the form (1.1) says that l_0 must be true if $l_1, \ \ldots, \ l_m$ are true and $l_{m+1}, \ldots, l_n$ can be safely assumed to be false (negation-as-failure).

It has been proved that all problems in NP can be solved within the paradigm of ASP [52], i.e. ASP is sufficiently expressive for a wide range of important computational problems including many combinatorial and optimization problems and constraint satisfaction problems (CSP).

The previous research on ASP has laid a solid theoretical foundation with a large body of building block results, e.g., equivalence between programs, programming methodologies, and relationships to other nonmonotonic formalisms. These results are very useful in the development of large knowledge bases.

There have been a number of ASP implementations, e.g. `SMODELS` [74], `DLV` [38], `CMODELS` [2], `ASSAT` [43], `CLASP` [30]. The performance of these systems has progressed rapidly. They have been used in a number of applications such as planning [22, 36], product configuration [75], phylogeny reconstruction in computational biology [82], and wire routing in computer aided design [22], etc. The latest progress in systems and applications is reported in [24].

In summary, ASP, as an alternative logic programming paradigm, has become an interesting topic in AI research. It has high expressiveness, solid theoretical foundations, and plenty of applications and implemented systems.

1.2 Answer set programming with constraints

Initially, ASP supported only *normal logic programs*, i.e., each literal in the rule (1.1) is either an atom or the negation of an atom. When working towards applications, it has been observed that normal logic programs are inadequate for many interesting domains. They cannot compactly express many commonly used constraints such as choices over subsets, cardinality and weight constraints.

In order to represent this type of knowledge, logic programs are extended to include

weight constraints [62, 75]. A weight constraint is of the form

$$l[a_1 = w_1,\ \ldots, a_m = w_m,\ \texttt{not}\ a_{m+1} = w_{m+1}\ \ldots,\ \texttt{not}\ a_n = w_n]u \tag{1.2}$$

where $a_1, ...a_n$ are atoms. Atoms $a_1, ..., a_m$ and atoms that are preceded by not , not a_{m+1}, ..., not a_n, are called *literals*. They are also called *positive literals* and *negative literals*, respectively. The w_i, l, and u are real numbers. w_i is the *weight* of the literal a_i (or not a_i). If a weight is 1, it is usually omitted. l and u are lower and upper bounds respectively. Either of the bounds can be omitted in which case the missing lower bound is taken to be $-\infty$ (no constraint on the lower bound) and the missing upper bound ∞ (no constraint on the upper bound).

Intuitively, a weight constraint is satisfied by a set of atoms S, if the summation of the weights of the literals among $\{a_1,\ \ldots,\ a_m\}$ that are in S plus the summation of the weights of literals among $\{a_{m+1},\ \ldots,\ a_n\}$ that are not in S is in between l and u, inclusive. Logic programs with weight constraints are called *weight constraint programs*.

Below are two examples of the use of weight constraint programs.

Example 1.2. [75] Consider the product configuration problem, which can be roughly defined as the problem of producing a configuration of a product as a collection of predefined components. The input of the problem is a *configuration model*, which describes the components that can be included in the configuration, the rules on how they can be combined to form a working product, and the requirements that specify some properties that the product should have. The output is a *configuration* which is an accurate description of a product consisting of components that satisfies all of the rules and requirements in the configuration model. The following example demonstrates some typical forms of knowledge in a configuration model.

A configuration model of a PC could consist of a number of components which could be different types of key boards, display units, hard disks, CD drives, floppy drives, extension cards and so on. It may contain rules such as a PC should have a mass storage which must be chosen from IDE hard disks, SCSI hard disks and floppy drives, and a PC needs a keyboard which could be either a Finish or UK layout. The configuration model may also contain a requirement, which is that no computer may have more than 4 IDE hard disks.

The product configuration problem can be encoded as follows.

$$
\begin{aligned}
computer &\leftarrow \\
1[idedisk,\ scsidisk,\ floppydisk]1 &\leftarrow computer \\
1[finnishKB, ukKB] &\leftarrow computer \\
1[idedisk_1,\ \ldots,\ idedisk_n]1 &\leftarrow idedisk \\
0[idedisk_1, \ldots, idedisk_n]4 &\leftarrow computer
\end{aligned}
$$

where $idedisk_1, \ldots, idedisk_n$ are IDE hard disks to be chosen from. □

Example 1.3. The graph coloring program in Example 1.1 can be concisely written as follows.

$$
\begin{aligned}
&1[color(X, red), color(X, blue), color(X, yellow)]1 \leftarrow node(X) \\
&\leftarrow edge(X, Y), color(C), col(X, C), col(Y, C)
\end{aligned}
$$

□

Besides weight constraints, logic programs is also extended to include *aggregates* [76]. Aggregates specify, in a nature way, more constraints than weight constraints, such as the number, the summation, average, maximum, and minimum of the atoms (numbers) in the set. An aggregate is a constraint on a set of atoms. A set of atoms satisfies an aggregate if the constraints specified by the aggregate is satisfied by the set. For example, let A be the aggregate $COUNT(\{X \mid p(X)\}) = 2$, where $D(X) = \{a, b\}$. A specifies a constraint on $D(A) = \{p(a), p(b)\}$, which can (and can only) be satisfied by the sets whose number of atoms in $D(A)$ equals to 2, i.e., A is satisfied by sets whose intersection with $D(A)$ is the set $\{p(a), p(b)\}$.

The standard aggregates are $COUNT$, SUM, AVG, MIN and MAX. The name of an aggregate naturally denotes the constraint it specifies. The sets that satisfy an aggregate are called the *models* of the aggregate. Logic programs with aggregates are called *aggregate programs*.

There are a number of proposed semantics for logic programs with aggregates. The semantics proposed by Faber et al [25] define the answer sets of a logic program with aggregates as the least models of the *reduct* of the program; Ferraris [27] define the answer sets of a logic program as the answer sets of a propositional theory translated from the program. Son et al [76] define answer set as the fixpoint of a monotone operator applied to the program; The semantics proposed by Denecker [18] and Pelov [63] deal with aggregates

by using an approximation theory and three-valued logic, building the semantics on a three-valued immediate consequence operator. It has been shown that the semantics of [76] and [18, 63] coincide. The set of answer sets admitted by them is a subset of that admitted by the semantics of [25].

Several answer set programming systems that support aggregates have been developed, including `DLV` by Lenoe et al [38], `SMODELS`^{A} by Elkabani et.al. [21], and `CLASP` by Gebser et al [31]. They implemented the semantics proposed by Faber et.al. [25], Son et al [76], and Ferraris [27], respectively.

The concept of *abstract constraint atoms* (or *c-atoms*) was proposed by Marek, Remmel and Truszczyński in [53, 54]. Logic programs with c-atoms are called *abstract constraint programs*. Abstract constraint programs provide an elegant theoretical framework for investigating, in a uniform fashion, various extensions of logic programming, including weight constraint and aggregate programs.

A c-atom is a pair of sets (D, C), where D is a finite set of atoms and C is a collection of subsets of D. It expresses a constraint on the atoms in D such that C is the collection of admissible solutions to the constraint. Many kinds of constraints can be represented as c-atoms. For instance, the aggregate $COUNT(\{X|p(X)\}) = 1$, where $D(X) = \{a, b\}$, can be represented by the c-atom $(\{p(a), p(b)\}, \{\{p(a)\}, \{p(b)\}\})$.

Intuitively, a c-atom A represents a constraint on models of the program containing A and the description of A includes an explicit description of what conditions an interpretation has to meet in order to satisfy A. Abstract constraint programs subsume weight constraint and aggregate programs.

The semantics of abstract constraint programs, i.e., the definition of answer sets of abstract constraint programs, has been a topic of intensive interest. Marek and Remmel [54] proposed the first explicit definition of answer sets for positive programs (programs without the operator `not`). An arguable shortcoming in this proposal is that unintuitive answer sets (e.g. non-minimal answer sets) can be derived in some cases and it does not coincide with some well-agreed semantics for aggregates [18, 25, 27, 67, 76].

Marek and Truszczyński [53] revisited the semantics of abstract constraint programs by focusing on programs with *monotone* constraints. A c-atom A is *monotone* if, for any two sets of atoms I and I' such that $I \subseteq I'$, we have that whenever I satisfies A, I' satisfies A. The semantics is extended to deal with *convex constraints* [49]. A c-atom is *convex* if, for any three sets of atoms I, W, and J such that $I \subseteq W \subseteq J$, we have that whenever both I and J satisfy A, W satisfies A. Programs with convex constraints can be encoded as

programs with monotone programs, as pointed out in [49].

The main advantage of focusing on monotone and convex c-atoms lies in that monotonicity provides a relative simple way to define answer sets and some results on monotone c-atoms can be easily extended to convex c-atoms. But many important constraints cannot be expressed directly by monotone c-atoms. Consider the aggregate $A = MIN(\{X|p(X)\}) > 2$, where $D(X) = \{1, 2, 3\}$. The aggregate says that A is satisfied by sets of atoms M of the form $p(X)$, where the minimal number of the intersection of $D(X)$ and the set of arguments of p is greater than 2. Obviously, A is not monotone, because for interpretations $I = \{p(3)\}$ and $I' = \{p(3), p(1)\}$, we have $I \subseteq I'$ and I satisfies A, but I' does not satisfy A.

Son et al [77] investigate the semantics of logic programs with arbitrary c-atoms, i.e., the c-atoms in a program could be nonmonotone as well as monotone. They introduce the concept of *conditional satisfaction*. Based on this concept, a fixpoint semantics for abstract constraint programs is developed. The semantics seems more intuitive than that proposed in [54].

Liu et al [48] introduce several notions of *computations*. They propose to use the *results* of the computations as the different definitions of answer sets for abstract constraint programs. They give the relationships among these different computations and among the corresponding definitions of answer sets. The semantics based on computations are generalizations of semantics previously proposed in [49, 53, 77].

You et al [87] propose an unfolding approach to study the semantics of abstract constraint programs. In the approach, an abstract constraint program is transformed to a normal logic program and the answer sets of the abstract constraint program are defined as the answer sets of the translated normal program. The semantics defined by this approach coincides with the semantics based on conditional satisfaction in [77]. The unfolding approach can also be used to study properties of abstract constraint programs. It makes it possible to characterize the properties of abstract constraint programs, using the known properties of the unfolded normal logic programs. This approach also leads to a definition of answer sets of abstract constraint programs with disjunctive head.

Shen and You introduce a compact representation of c-atoms [71]. Unlike the commonly used power set form representation of c-atoms, the new approach represents a c-atom by a pair of sets, one being the prefix of the other. In many cases, the new representation results in a substantial reduction of the size from its power set representation. Based on this representation, the Gelfond-Lifschitz transformation [33] is generalized to define an-

swer sets for abstract constraint programs. This definition is proved to coincide with the semantics proposed in [77].

Several properties of logic programs with constraints have been studied. The definition of loop formulas for a program is one of the most important properties. There are two reasons for the adoption of loop formulas. One is that loop formulas can be regarded as a characterization of semantics. It is known that every stable model of a program is a supported model of the program [13], but the converse is not true in general. This is because an atom in a supported model may be circularly justified, i.e., justified by itself. Loop formulas can capture the circular justification of atoms. Different semantics have different loop formulas. Another is that loop formulas are the key part of the *loop formula approach* to answer set computation [43, 49]. In the approach answer sets of a program are computed in the following way: firstly, computing the *supported models* (models of the *completion*[13]) of the given program; then filtering out the supported models that do not satisfy the loop formulas. Finding efficient methods for constructing loop formulas has been of much research interest. We study loop formulas for aggregate and abstract constraint programs.

Strong equivalence is another property of interest. Two programs are said to be *equivalent* if they have the same set of answer sets. The notion of *strong equivalence* is proposed since the notion of equivalence is inadequate for some applications such as program optimization, where changes made on a program are often restricted within a part of the program.

Two programs are said to be *strongly equivalent* if after adding any other program to each of them the resulting programs are still equivalent. Lifschitz et al [40] propose to use the logic *Here-and-There* to characterize strong equivalence between programs. Lin [41] shows that the notion of strong equivalence of logic programs can be reduced to the entailment of classic propositional logic theories. Turner [79] uses model-theoretic means to characterize strong equivalence: two programs are strongly equivalent if they have the same set of *strong equivalent models*. Liu and Truszczyński [49] extend the characterizations of strong equivalence to logic programs with monotone c-atoms. We give characterizations of strong equivalence for logic programs with arbitrary c-atoms.

A number of systems have been implemented to support abstract constraint programs. `ASP-CLP` [20], developed by Elkabani and his colleagues, integrates the answer set solver `SMODELS` with the constraint solver `ECLiPSe`. But it does not guarantee the minimality of answer sets and the cost of communication between `SMODELS` and `ECLiPSe` is significant. System `SMODELS`^{A} [21], developed by the same group addresses the above

problems. In SMODELS^A , an aggregate program is transformed to a normal logic program, then SMODELS is applied to compute the answer sets for the normal logic program. The system DLV [38] supports aggregate programs but does not allow aggregates to appear in the heads of the rules. We propose an approach to compute aggregate programs and implement it in the system ALPARSE .

Liu and Truszczyński [49] extend the loop formula approach proposed in [43] to logic programs with monotone and convex c-atoms. They show that a logic program with convex c-atoms can be transformed to a pseudo-Boolean theory so that the answer sets of the program can be computed as the models of the pseudo-Boolean theory, using off-the-shelf pseudo-Boolean constraint solvers. This approach is implemented in Pbmodels .

1.3 Contributions

We study three main classes of logic programs with constraints: weight constraint, aggregate, and abstract constraint programs. Our work focuses on two aspects of these programs: the properties and computations. On properties, we study the loop formulas and strong equivalence. On computations, we improve the existing methods for weight constraint programs and propose a new method for aggregate programs.

In more detail, the main contributions of this thesis are as follows.

- For weight constraint programs, we
  - present the level mapping characterization of stable models and propose loop formulas for weight constraint programs based on the characterization (the main result is reported in [44]);
  - revise the constraint propagation mechanism *lookahead* to *adaptive lookahead* thus make the computation of weight constraint programs more efficient for SMODELS (the work and subsets of experiments were presented in [45] and [47]). The approach is also applicable to SAT solvers that employ lookahead.
- For aggregate programs, we
  - present the level mapping characterization of answer sets and propose the loop formulas (this work is published in [44]);
  - propose an approach to computing the aggregate programs as weight constraint programs (the work is reported in [46]);

- For abstract constraint programs, we
  - propose loop formulas (the work is published in [84]);
  - give two characterizations of strong equivalence.

For future work, we propose the integration of *global constraints* to ASP, as a promising direction. We provide the experiments that show the efficiency of using global constraints for problem solving. We then present a property of global constrains which is useful for the study of the properties and computations of programs with global constraints.

1.4 Outline of the thesis

The first chapter is an introduction. Chapter 2 to Chapter 10 report main work of the thesis, which consists of three parts, dealing with weight constraint programs, aggregate programs and abstract constraint programs, respectively. Each part starts with a chapter of background knowledge, followed by the chapters where our work is presented.

Part one begins with Chapter 2 that introduces the semantics of weight constraint programs. Then Chapter 3 is the study of level mapping and loop formulas for weight constraint programs. In Chapter 4, we improve the computation of weight constraint programs by proposing the adaptive lookahead mechanism.

In Part two, we introduce the semantics of aggregate programs based on conditional satisfaction and then briefly survey other semantics in Chapter 5. We take the semantics based on conditional satisfaction as the semantics of aggregate programs. The loop formulas for aggregate programs are put forward in Chapter 6, and an approach to the computation of aggregate programs are proposed in Chapter 7.

In Part three, we present various semantics of abstract constraint programs in Chapter 8. Similar to Part two, we choose the semantics based on conditional satisfaction as the semantics of abstract constraint programs. We propose loop formulas for abstract constraint programs in Chapter 9. The characterizations of the strong equivalence of abstract constraint programs are presented in Chapter 10.

Chapter 11 discusses the future work and we conclude the thesis in Chapter 12.

Chapter 2

Stable model semantics

We first introduce the stable model semantics for normal logic programs, followed by the stable model semantics for weight constraint programs.

2.1 Normal logic programs

A *(normal) logic program* is a set of rules of the form

$$h \leftarrow a_1,\ \ldots,\ a_m, \texttt{not}\ b_1,\ \ldots,\ \texttt{not}\ b_n \tag{2.1}$$

where h, $a_1, \ldots, a_m$, $b_1, \ldots, b_n$ are *atoms*. The atom h is called the *head* of the rule. The set $\{a_1, ...a_m, ..., \texttt{not}\ b_1, ..., \texttt{not}\ b_n\}$ is called the *body* of the rule. The head of a rule may be empty, in which case the rule serves as a constraint specifying that the body must be false in any intended model. The body of a rule may be empty in which case we may drop the symbol '$\leftarrow$'. Such a rule serves as a fact. We call the expression $\texttt{not}\ b$ a *not-atom*. Atoms and not-atoms are both referred to as *literals*. When necessary, we refer to atoms and not-atoms as *positive literals* and *negative literals*, respectively. We use $At(P)$ to denote the set of the atoms that appear in the program P. A program is called a *positive program* if every literal in it is positive.

In answer set programming, variables are allowed to appear in programs. In this document, we assume that each rule of the form (2.1) in a program has been replaced by all its ground instances, so that all the atoms in the resulting program are ground. Such a program is usually called a *ground* program. To ensure that ground programs are finite, function symbols are not allowed in answer set programming.[1]

A set of atoms (or an interpretation) M *satisfies* an atom a, denoted $M \models a$, if $a \in M$; M satisfies $\texttt{not}\ a$, denoted $M \not\models a$, if $a \notin M$. M satisfies a rule r if it satisfies the head

[1] Recently, some languages re-introduce functions [9, 42].

of r whenever it satisfies the body. M is called a *model* of a program P if it satisfies every rule in P.

Let P be a program and M a set of atoms. The *reduct* of P with respect to M, denoted P^M, is a program obtained by

1. deleting each rule in P that has a negative literal $\texttt{not}\ x$ in its body such that $x \in M$, and
2. deleting all negative literals in the remaining rules.

The reduct of a program is a positive program. It is known that for a positive program P, there is an unique minimal model. This model, denoted by $Cl(P)$, is called the *deductive closure* of P.

Definition 2.1. Let P be a logic program. A set of atoms M is a *stable model* of P if and only if $M = Cl(P^M)$.

Example 2.1. Let P_1 be a program.

$$\begin{aligned} p &\leftarrow \texttt{not}\ q, r \\ q &\leftarrow \texttt{not}\ p \\ r &\leftarrow \texttt{not}\ s \\ s &\leftarrow \texttt{not}\ p \end{aligned}$$

The set $M_1 = \{r, p\}$ is a stable model of P because the reduct $P_1^{M_1}$ of P_1 with respect to M_1 is

$$\begin{aligned} p &\leftarrow r \\ r &\leftarrow \end{aligned}$$

and $M_1 = Cl(P_1^M)$. Similarly, $M_2 = \{q, s\}$ is also a stable model of P_1.

But $M_3 = \{p, s\}$ is not a stable model of P_1 because the reduct $P_1^{M_3}$ is $\{p \leftarrow r\}$ and its deductive closure is $\emptyset$. □

A program may have no stable model. The following is an example.

Example 2.2. Consider the program P_2.

$$\begin{aligned} f' &\leftarrow \texttt{not}\ f', f \\ f & \end{aligned}$$

Assume that P_2 has a stable model S. Then $f \in S$. If $f' \in S$, then the reduct is $\{f \leftarrow\}$ and its deductive closure does not include f'. If $f' \notin S$, then the reduct is

$$\begin{array}{rcl} f' & \leftarrow & f \\ & & f \end{array}$$

whose deductive closure contains f'. Thus, P_2 does not have a stable model. □

For a program P, if there is no negative literal in it, then for any set of atoms M, P^M is identical to P and $Cl(P)$ is the unique stable model of P.

The definition of stable model captures the key properties of stable models:

- Stable models are *founded*: each atom in a stable model has a justification in terms of the program, i.e., it is derivable from the reduct of the program with respect to the model.
- Stable models are minimal: no proper subset of a stable model is a stable model.

The requirement of foundedness guarantees that a stable model of a program is a subset of atoms appearing in the head of the rules in the program. Consider a program $P = \{a \leftarrow a\}$, the possible stable models are $\emptyset$ and $\{a\}$. But $\{a\}$ is not minimal, i.e., $\{a\}$ is not the deductive closure of $P^{\{a\}}$. (The deductive closure of $P^{\{a\}}$ is $\emptyset$.) Therefore, the only stable model of P is $\emptyset$.

2.2 Weight constraint programs

A *weight constraint* is of the form

$$l\,[a_1 = w_{a_1}, ..., a_m = w_{a_m}, \texttt{not}\ b_1 = w_{b_1}, ..., \texttt{not}\ b_n = w_{b_n}]\,u \tag{2.2}$$

where each a_i, b_j is an atom, and each atom and not-atom is associated with a *weight*. Atoms and not-atoms are also called *literals* (the latter may be emphasized as *negative* literals). The numbers l and u are the *lower* and *upper bounds*, respectively. The weights and bounds are real numbers. [2] A weight may be omitted when it is 1. Either of the bounds may be omitted in which case the missing lower bound is taken to be $-\infty$ and the missing upper bound by ∞.

Given a weight constraint W of the form (2.2), we define:

[2] In the current implementations only integer weights are supported.

- The literal set of W, denoted $lit(W)$, is the set of literals occurring in W, i.e., $lit(W) = \{a_1, ..., a_m, \texttt{not}\ b_1, ..., \texttt{not}\ b_n\}$
- The atom set of W, denoted $At(W)$, is the set of atoms occurring in W, i.e., $At(W) = \{a : a \in lit(W)\ or\ \texttt{not}\ a \in lit(W)\}$.

A set of atoms M satisfies a weight constraint W of the form (2.2), denoted $M \models W$, if (and only if) $l \leq w(W, M) \leq u$, where

$$w(W, M) = \sum_{a_i \in M} w_{a_i} + \sum_{b_i \notin M} w_{b_i} \tag{2.3}$$

M satisfies a set of weight constraints Π if $M \models W$ for every $W \in \Pi$.

Example 2.3. Let $W = 2[a = 8, b = 2, \texttt{not}\ c = 4]11$ be a weight constraint and $M_1 = \{a, b, c\}$ and $M_2 = \{a\}$ be two sets. We have $w(W, M_1) = 10$ and $w(W, M_2) = 12$, therefore $M_1 \models W$ and $M_2 \not\models W$. □

A weight constraint W is *monotone* if for any two sets R and S, if $R \models W$ and $R \subseteq S$, then $S \models W$; otherwise, W is *nonmonotone*. There are some special classes of nonmonotone weight constraints. W is *antimonotone* if for any R and S, $S \models W$ and $R \subseteq S$ imply $R \models W$; W is *convex* if for any R and S such that $R \subseteq S$, if $S \models W$ and $R \models W$, then for any I such that $R \subseteq I \subseteq S$, we have $I \models W$.

A weight constraint that contains neither negative literals nor negative weights is of interest. We call them *positive* weight constraints. Positive weight constraints are convex; Positive weight constraints without the upper bound are monotone; Positive weight constraints without the lower bound are antimonotone.

When a weight constraint contains negative literals or negative weights, it may be nonmonotone and doesn't fall into any of the special classes. For example, let $W = 1\ [a = 1, \texttt{not}\ b = 1]\ 2$ be a weight constraint. W is nonmonotone, but neither antimonotone nor convex, as shown below: consider $M_1 = \emptyset$, $M_2 = \{b\}$, and $M_3 = \{a, b\}$, and it's clear $M_1 \models W$, $M_2 \not\models W$, and $M_3 \models W$.

As pointed out by [74], negative weights and negative literals are closely related in that they can replace each other and that one is inessential when the other is available.

Negative weights can be eliminated by applying the following transformation: Given a weight constraint W of the form (2.2), if $w_{a_i} < 0$, then replace $a_i = w_{a_i}$ with $\texttt{not}\ a_i = |w_{a_i}|$ and increase the lower bound to $l + |w_{a_i}|$ and the upper bound to $u + |w_{a_i}|$; if $w_{b_i} < 0$, then replace $\texttt{not}\ b_i = w_{b_i}$ with $b_i = |w_{b_i}|$ and increase the lower bound to $l + |w_{b_i}|$ and the upper bound to $u + |w_{b_i}|$.

For instance, the weight constraint

$$-1\ [a_1 = -1, a_2 = 2, \texttt{not}\ b_1 = 1, \texttt{not}\ b_2 = -2]\ 1 \tag{2.4}$$

can be transformed to

$$2\ [\texttt{not}\ a_1 = 1, a_2 = 2, \texttt{not}\ b_1 = 1, b_2 = 2]\ 4 \tag{2.5}$$

The transformation above is equivalence-preserving, in the sense that for any weight constraint W and set of atoms M, $M \models W$ iff $M \models W'$, where W' is obtained from W by applying the transformation.

We assume that weights are non-negative if not said otherwise.

The *reduct* of a weight constraint W of the form (2.2) w.r.t. a set of atoms M, denoted by W^M, is the constraint

$$l^M\ [a_1 = w_{a_1}, ..., a_n = w_{a_n}] \tag{2.6}$$

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

Example 2.4. Let $W = 3[a_1 = 1, \texttt{not}\ b_1 = 2, \texttt{not}\ b_2 = 3]5$ be a weight constraint and $S_1 = \{b_1\}$ and $S_2 = \{b_2\}$ be two sets. Then $W^{S_1} = 0[a_1 = 1]$ and $W^{S_2} = 1[a_1 = 1]$. □

The reduct of a weight constraint is monotone, since it is positive and has no upper bound.

A *weight constraint program* is a finite set of rules of the form

$$W_0 \leftarrow W_1, ..., W_n \tag{2.7}$$

where each W_i is a weight constraint. In a rule r of the form (2.7), W_0 is the head of r, denoted $hd(r)$; the sets of atoms in W_0 is the headset of r, denoted $hset(r)$; the set $\{W_1, ..., W_n\}$ is the body of r, denoted $bd(r)$.

Given a program P, we define

- The atom set of P, denoted $At(P)$, as the set of atoms appearing in P.
- The head set of P, denoted $hset(P)$, as the union of the headsets of all rules in P, i.e., $hset(P) = \cup_{r \in P} hset(r)$.

A weight constraint of the form $1\ [l\ =\ 1]\ 1$ will be simply written as l. A weight constraint program, where each weight constraint is of the form $1\ [l = 1]\ 1$ is essentially a normal program.

Let P be a weight constraint program and M a set of atoms. The reduct P^M of P, w.r.t. M, is defined by

$$P^M = \{p \leftarrow W_1^M, \ldots, W_n^M \mid W_0 \leftarrow W_1, \ldots W_n \in P, \\ p \in lit(W_0) \cap M \; and \; w(W_i, M) \leq u \; for \; all \; i \geq 1\} \tag{2.8}$$

For a rule $r \in P$, we call its counterpart in P^M the reduct of r, denoted r^M.

Definition 2.2. [74] Let P be a weight constraint program and $M \subseteq At(P)$. M is a *stable model* of P iff the following two conditions hold:

1. $M \models P$,
2. M is the deductive closure of P^M.

Example 2.5. Let P be a program consisting of the following rules.

$$1[a = 1, b = 1]2 \quad \leftarrow \quad 0[a = 1, b = 1, \texttt{not}\ c = 1]2 \tag{2.9}$$

$$1[a = 1, b = 1]2 \quad \leftarrow \quad 0[a = 1, \texttt{not}\ b = 1]1 \tag{2.10}$$

Let $M = \{a, b\}$. P^M is the program

$$a \leftarrow 0[a = 1] \tag{2.11}$$

$$b \leftarrow 0[a = 1] \tag{2.12}$$

We have $M \models P$ and M is the deductive closure of P^M. So, M is a stable model of P. □

Note that P^M is a weight constraint program where all constraints are monotone and the head of each rule is an atom. Thus its deductive closure can be computed by a fixpoint construction, using the operator T_P defined in [49] as follows.

Let P be a weight constraint program, where the head of each rule is an atom and the body contains no negative literals. The operator T_P is defined as:

$$T_P(S) = \{h \mid \exists r \in P \text{ of the form } h \leftarrow bd(r) \text{ and } S \models bd(r)\}. \tag{2.13}$$

The least fixpoint of T_P can be constructed by

$$T_P^0(\emptyset) = \emptyset \tag{2.14}$$

$$T_P^{i+1}(\emptyset) = T_P(T_P^i(\emptyset)) \tag{2.15}$$

$$T_P^{\infty}(\emptyset) = \cup_{i=0}^{\infty} T_P^i(\emptyset) \tag{2.16}$$

Then, we have the following proposition.

Proposition 2.1. Given a weight constraint program P, a set of atoms M is a stable model of P iff $M \models P$ and $M = T_{P^M}^{\infty}(\emptyset)$.

Given a program P, the least fixpoint of T_{P^M} is the deductive closure of P^M, according to the definition of T_P. The proof of Proposition 2.1 is straightforward.

We will use the concept of *supported model* later.

Definition 2.3. Let P be a weight constraint program and M a subset of $At(P)$. M is a supported model of P if

- $M \models P$ and
- For any atom $a \in M$, there exists a rule $r \in P$ such that $a \in hset(r)$ and $M \models bd(r)$.

Example 2.6. Let P be the program $\{a \leftarrow 1[b = 1]\}$ and $M = \{a\}$. M is a model of P, but not a supported model of P. □

It is easy to show the following proposition.

Proposition 2.2. Let P be a weight constraint program. If a set $M \subseteq At(P)$ is a stable model of P, then M is a supported model of P.

The converse of Proposition 2.2 may not be true, i.e., a supported model of a program may not be a stable model. The following is an example.

Example 2.7. Let P be the program $\{a \leftarrow 1[a = 1]\}$ and $M = \{a\}$. M is a supported model of P but not a stable model. □

Chapter 3

Level mapping induced loop formulas for weight constraint programs

3.1 Motivation

It is known that every stable model of a program is a supported model of the program, but the converse is not true in general. The reason is that an atom in a supported model may be circularly justified, i.e., justified by itself. For example, for the program $P = \{a \leftarrow a\}$, the set $M = \{a\}$ is a supported model of P. But M is not a stable model, since a is justified by itself.

Loop formulas constitute a propositional theory that specifies the requirement that an atom must not be circularly justified. For the above example program $P = \{a \leftarrow a\}$, a loop formula is $a \rightarrow \bot$, which prevents a to be in any stable model.

Lin and Zhao [43] propose an approach to the computation of stable models of a normal program, where stable models are computed by firstly computing supported models of the program and then using *loop formulas* not only to filter out the models that are not stable models but also to generate new constraints on the rest of the computation to make it more efficient.

Liu and Truszczyński [49] extend the approach to logic programs with positive weight constraints. They define loop formulas for such programs. However, to transform an arbitrary weight constraint to a positive weight constraint, new propositional atoms are needed [49, 51]. The extra atoms may exert a heavy toll on SAT solvers when the number of them gets too big (for instance, the default setting in the SAT solver SATO allows only a maximum of 30,000 atoms). In the worst case, each extra atom may double the search space.

In this chapter, we address the question of whether the loop formulas for arbitrary

weight constraint programs can be formulated without extra atoms.

The method of level mapping has been used to characterize stable models of normal logic programs in [23, 26]. We observe that such a characterization is closely related to the formulation of loop formulas. From the level mapping point of view, a loop formula can be satisfied by a supported model of a program if an atom in the loop can be derived by the atoms that are not in the loop and have a strictly lower level.

We present a level mapping characterization of the stable models of weight constraint programs. The characterization leads to a formulation of loop formulas for arbitrary weight constraint programs, where no extra atoms are introduced.

The level mapping characterization of stable models is given in Section 3.2. Section 3.3 presents the loop formulas for weight constraint programs. We relate our work to previous work in Section 3.4 . Section 3.5 contains additional remarks on our approach.

3.2 Level mapping characterization of stable models

Notation: Given a weight constraint W of the form (2.2) and a set of atoms M, we define

- $M_a(W) = \{a_i \in M \mid a_i \in lit(W)\}$
- $M_b(W) = \{b_i \in M \mid \texttt{not}\ b_i \in lit(W)\}$.

Since W is always clear by context, we will simply write M_a and M_b.

In general, an atom may appear both positively and negatively. We call such an atom a *dual* atom, e.g. atom a is a dual atom in $1[a = 1, \texttt{not}\ a = 2]1$. Given a set of atoms M, if there are no dual atoms in a weight constraint W, then $M_a(W) \cap M_b(W) = \emptyset$, otherwise, $M_a(W) \cap M_b(W) \neq \emptyset$.

Following the notation in [77], for a set of atoms X and a mapping λ from X to positive integers, we define

$$H(X) = max(\{\lambda(a) \mid a \in X\}). \tag{3.1}$$

For the empty set $\emptyset$, we define $\max(\emptyset) = 0$ and $\min(\emptyset) = \infty$.

Given a set of atoms X, a *level mapping* of X is a function λ from X to positive integers.

Definition 3.1. Let W be a weight constraint of the form (2.2), M a set of atoms and λ a level mapping of M. The *level* of W w.r.t. M, denoted $L(W, M)$, is defined as:

$$L(W, M) = min(\{H(X_a) \mid X \subseteq M,\ and\ w(W, X_a) \geq l + \sum_{b_i \in M \setminus X_a} w_{b_i}\}). \qquad (3.2)$$

If there are no dual atoms in W, $M_b \cap X_a = \emptyset$. Thus $\sum_{b_i \in M \setminus X_a} w_{b_i}$ in formula (3.2) becomes $\sum_{b_i \in M} w_{b_i}$.

Example 3.1. Let $W = 1[a = 1, \texttt{not}\ a = 1, \texttt{not}\ b = 1]$ be a weight constraint, $M = \{a\}$ a set, and λ a level mapping of M. Considering the subset of M, $X = \emptyset$, we have $w(W, X_a) = 2$ and $l + \sum_{b_i \in M \setminus X_b} w_{b_i} = 1 + w_a = 2$. $w(W, X_a) \geq l + \sum_{b_i \in M \setminus X_b} w_{b_i}$. Similarly, we can check that the other subset of M, $X' = M$, also satisfies the inequality in formula (3.2). Thus, $L(W, M) = min(\{H(\emptyset), H(\{a\})\})$. Note that the level of any atom is a positive number. Then we have $L(W, M) = 0$. □

Proposition 3.1. Let W be a weight constraint of the form (2.2), M and X be two sets of atoms. $w(W^M, X_a) \geq l^M$ iff $w(W, X_a) \geq l + \sum_{b_i \in M \setminus X_a} w_{b_i}$.

Proof. The reasoning is as follows.

(1) $w(W, X_a) = w(W^M, X_a) + \sum_{b_i \notin X_a} w_{b_i}$.

(2) $w(W^M, X_a) \geq l - \sum_{b_i \notin M} w_{b_i}$ if and only if $w(W, X_a) \geq l - \sum_{b_i \notin M} w_{b_i} + \sum_{b_i \notin X_a} w_{b_i}$, due to (1).

(3) $w(W, X_a) \geq l^M$ if and only if $w(W, X_a) \geq l + \sum_{b_i \in M \setminus X_a} w_{b_i}$, due to (2).

□

Definition 3.2. Let P be a weight constraint program and M a set of atoms. M is said to be *level mapping justified* by P if there is a level mapping λ of M, such that for each $b \in M$, there is a rule $r \in P$, such that $b \in lit(hd(r))$, $M \models bd(r)$, and for each $W \in bd(r)$, $\lambda(b) > L(W, M)$.

In this case, we say that the level mapping λ *justifies* M *by* P.

Using Proposition 3.1, we can prove the following theorem.

Theorem 3.1. Let P be a weight constraint program and M a set of atoms. M is a stable model of P iff M is a model of P and level mapping justified by P.

Proof. ($\Rightarrow$)

(1) M is a stable model of P.

(2) $M = T_{P^M}(\emptyset)$, due to (1).

(3) There is a level mapping $\lambda : M \to \mathbb{Z}^+$: $\lambda(b) = k$ if $b \in T^k_{P^M}(\emptyset)$ and $b \notin T^{k-1}_{P^M}(\emptyset)$, due to (2).

(4) λ justifies M by P, due to (2) and (3).

(5) M is level mapping justified, due to (4).

($\Leftarrow$)

(1) $M \models P$ and M is level mapping justified.

(2) Suppose that λ justifies M by P.

(3) $\forall b \in M$, there is a rule $r \in P$ s.t. $b \in lit(hd(r))$ and $M \models r$, due to (1).

(4) There is a rule $r^M \in P^M$, s.t. $b \in lit(hd(r^M))$ and $M \models bd(r^M)$, due to (3).

(5) $b \in Cl(P^M)$, due to (4).

(6) $M \subseteq Cl(P^M)$.

(7) $M \models P^M$, due to (1).

(8) $Cl(P^M) \subseteq M$, due to (7).

(9) $M = Cl(P^M)$, due to (6) and (8).

(10) M is a stable model of P, due to (1) and (9).

□

Example 3.2. Let P_1 be the program $\{a \leftarrow 1[a = 1, \texttt{not}\ a = 1, \texttt{not}\ b = 1]\}$. Let $W = 1[a = 1, \texttt{not}\ a = 1, \texttt{not}\ b = 1]$, $M = \{a\}$ and λ be a level mapping of M. We have $L(W, M) = 0$ (cf. Example 3.1). Therefore $\lambda(a) > L(W, M)$ and M is level mapping justified by P_1. Thus, M is a stable model of P_1. □

3.3 Completion and loop formulas

3.3.1 Completion

To characterize stable models by loop formulas, we need the concept of *completion* of a program. The models of the completion of a program are the supported models of the

program. Following [49], the completion of a weight constraint program P is defined as a set of formulas built from weight constraints by means of Boolean connectives $\wedge$, $\vee$ and $\rightarrow$[1].

Let $S = \{f_1, ..., f_n\}$ be a set of weight constraints or formulas built from weight constraints. We denote the conjunction $f_1 \wedge ... \wedge f_n$ by $\wedge S$ and the disjunction $f_1 \vee ... \vee f_n$ by $\vee S$.

Let P be a weight constraint program. The completion of P, denoted $Comp(P)$, consists of the following formulas.

(1). $\wedge bd(r) \rightarrow hd(r)$, for every rule $r \in P$.

(2). $x \rightarrow \vee\{\wedge bd(r) \mid r \in P, x \in lit(hd(r))\}$, for every atom $x \in At(P)$.

3.3.2 Loop formulas

The formulation of loop formulas consists of two steps: construction of a dependency graph and then establishing of a formula for each loop in the graph.

We now define the dependency graph for a weight constraint program.

Let P be a weight constraint program. The *dependency graph* of P, denoted $G_P = (V, E)$, is a directed graph, where

- $V = At(P)$,
- (u, v) is a directed edge from u to v in E, if there is a rule of the form (2.7) in P, such that $u \in lit(W_0)$ and $v \in lit(W_i)$, for some i ($1 \leq i \leq n$).

Let $G = (V, E)$ be a directed graph. A set $L \subseteq V$ is a *loop* in G if the subgraph of G induced by L is strongly connected. Recall that a directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.

For a loop in the dependency graph, the level mapping induced loop formula is established to enforce that the atoms in the loop must be justified by the atoms that are not in the loop and have a strictly lower level. Considering the definition of $L(W, M)$ (see the formula (3.2)), the condition requires that an atom in a loop be derivable by a subset X of M which contains no atoms in the loop and satisfies the inequality in formula (3.2).

To enforce the above requirements, we define the *restriction* of a weight constraint w.r.t. a loop.

[1]The connective $\neg$ is not needed, since no negative weight constraints (`not` W) are allowed in weight constraint programs.

Let W be a weight constraint and L be a set of atoms. The *restriction* of W w.r.t. L, denoted $W_{|L}$, is a conjunction of weight constraints $W_{l|L} \wedge W_{u|L}$, where

- $W_{l|L}$ is obtained by removing the upper bound, all positive literals that are in L and their weights from W;
- $W_{u|L}$ is obtained by removing the lower bound from W.

Definition 3.3. Let P be a weight constraint program and L a loop in G_P. The *loop formula* for L, denoted $LF(P, L)$, is defined as

$$LF(P, L) = \bigvee L \rightarrow \bigvee \{ \bigwedge_{W \in bd(r)} W_{|L} \mid r \in P, L \cap lit(hd(r)) \neq \emptyset \} \tag{3.3}$$

Let P be a weight constraint program. The *loop completion* of P is defined as

$$LComp(P) = Comp(P) \cup \{LF(P, L) \mid L \; is \; a \; loop \; in \; G_P\} \tag{3.4}$$

With the definition of loop completion, we can prove

Theorem 3.2. Let P be a weight constraint program and M a set of atoms. M is a stable model of P iff M is a model of $LComp(P)$.

Proof. We consider rules of the form $b \leftarrow W$, where b is an atom and W is a weight constraint of the form (2.2). The proof can be extended straightforwardly to rules with conjunctive body. In the proof below, we only show the satisfaction of the lower bound $W_{l|L}$; the satisfaction of the upper bound of $W_{u|L}$ is trivial.
($\Rightarrow$)

(1) $M \models LF(P, L)$.

(2) $\forall b \in M \cap L, \exists r \in P$, such that $hd(r) \cap L \neq \emptyset$ and $M \models W_{l|L}$, due to (1).

(3) $w(W_{l|L}, M) \geq l$, due to (2).

(4) Let $X = M_a \setminus L_a$.

(5) $w(W_{l|L}, M) = \sum_{a_i \in X_a} w_{a_i} + \sum_{b_i \notin M} w_{b_i}$.

(6) $\sum_{a_i \in X_a} w_{a_i} + \sum_{b_i \notin M} w_{b_i} \geq l$, due to (3) and (5).

(7) $w(W_{l|L}, X_a) = \sum_{a_i \in X_a} w_{a_i} + \sum_{b_i \notin X_a} w_{b_i}$

(8) $w(W_{l|L}, X_a) \geq l - \sum_{b_i \notin M} w_{b_i} + \sum_{b_i \notin X_a} w_{b_i}$, due to (6) and (7).

(9) $w(W_{l|L}, X_a) \geq l + \sum_{M \setminus X_a} w_{b_i}$, due to (8).

(10) $w(W_{l|L}, X_a) = w(W, X_a)$, due to (4).

(10) $w(W, X_a) \geq l + \sum_{b_i \in M \setminus X_a} w_{b_i}$, due to (8) and (9).

(11) The level mapping λ where $\lambda(b) > max(\{\lambda(a_i) \mid a_i \in X\})$ justifies M by P.

(12) $M \models Comp(P)$.

(13) $M \models P$, due to (12).

(14) M is a stable model of P, due to (11) and (13).

($\Leftarrow$)

(1) M is a stable model of P.

(2) There is a level mapping λ from $Atom(P)$ to positive integers satisfying $\forall b \in M, \exists r \in P$, such that $\lambda(b) > L(W, M)$, due to (1).

(3) Let L be a loop and $b \in L$. Let X be a set of atoms, such that $X \subseteq M \setminus L$, such that $b \in lit(hd(r))$ and $w(W, X_a) \geq l + \sum_{b_i \in M \setminus X_a} w_{b_i}$.

(4) $w(W_{l|L}, X_a) = w(W, X_a)$.

(5) $w(W_{l|L}, M) = \sum a_i \in M \setminus L w_{a_i} + \sum_{b_i \notin M} w_{b_i}$.

(6) $w(W_{l|L}, X_a) = \sum a_i \in M \setminus L w_{a_i} + \sum_{b_i \notin X_a} w_{b_i}$, due to (3).

(7) $w(W_{l|L}, M) \geq w(W_{l|L}, X_a) - \sum_{b_i \in M \setminus X_a} w_{b_i}$, due to (5) and (6).

(8) $w(W_{1|L}, M) \geq w(W, X_a) - \sum_{b_i \in M \setminus X_a} w_{b_i}$, due to (4) and (7).

(9) $w(W_{1|L}, M) \geq l$, due to (3) and (8).

(10) $M \models LF(P, L)$, due to (9).

(11) $M \models Comp(P)$, due to (1).

(12) $M \models LComp(P)$, due to (10) and (11).

□

Example 3.3. Consider the program P_1 in Example 3.2. There is a loop in $L = \{a\}$ in G_{P_1}. The loop formula is $F : a \rightarrow 1[\texttt{not}\ a = 1, \texttt{not}\ b = 1] \wedge [a = 1, \texttt{not}\ a = 1, \texttt{not}\ b = 1]$ (the second weight constraint is irrelevant, since it has no bounds and can be satisfied by any set). Let $M = \{a\}$. M satisfies F and is therefore a stable model of P_1. □

3.4 Relation to weakly tight programs

A weight constraint program can be translated to a program with *nested expressions* [28], such that its stable models are precisely the stable models of the original program.

In [86], the stable models of programs with nested expressions are characterized by the concept of *weak tightness*, that is, given a program with nested expressions, a set of atoms is a stable model of the program if and only if it is a supported model[2] and the program is weakly tight on it.

It is easy to show that, if a weight constraint program is transformed to a program with nested expressions using the transformation proposed in [28], the level mapping characterization for stable models coincides with that based on weak tightness in [86], as stated in the following theorem.

Theorem 3.3. Let P be a weight constraint program and M a supported model of P. M is level mapping justified by P iff $[P]$ is weakly tight on M, where $[P]$ is the program with nested expressions obtained by the transformation in [28].

3.5 Conclusion

We present a level mapping characterization of stable models. Based on the characterization, we define loop formulas for arbitrary weight constraint programs. To construct the loop formulas, it is not needed to transform arbitrary weight constraints to positive weight constraints. Therefore, the extra atoms introduced by the transformation are avoided.

[2] A model M of a program P is a supported model of P if for any $a \in M$, there is a rule in P, such that $a \in lit(hd(r))$ and $M \models bd(r)$.

Chapter 4

An improvement in computation of weight constraint programs

4.1 Motivation

Complete SAT/ASP solvers are typically variants of the DPLL search algorithm [16], in which *unit propagation*, sometimes also called *Boolean constraint propagation* (BCP), is considered a critical component. In the popular answer set solver SMODELS [74], the algorithm that corresponds to BCP is called the *expand* function.

On top of BCP/*expand*, other deductive mechanisms have been proposed. One of them is called *lookahead* [29] - before a decision on a choice point is made, for each unassigned atom, if fixing the atom's truth value leads to a contradiction, the atom gets the opposite truth value. In this way, an atom may get a truth value from the truth value propagation of already assigned atoms without going through a search process. The above process is carried out repeatedly until no unassigned atoms can be fixed a truth value in this way.

Lookahead, however, incurs high overhead. In the case of SAT, the full employment of lookahead has the worst case complexity $O(mn^2)$, where m is the size of the SAT instance and n the number of distinct atoms in it. For the ASP solver SMODELS , the complexity becomes $O(mn^3)$ [83], due to the computation of *unfounded atoms* in the *expand* function.

The (in)effectiveness of lookahead in SAT solvers was studied in [34]. The main conclusion is that lookahead does not pay off when integrated with look-back methods. However, that is not necessarily the case. As an polynomial time constraint propagation scheme, lookahead has the potential to allow the search to avoid searching subtrees that might take it exponential time to explore. Thus it is always possible to contrive examples where the (time) cost of lookahead pays off because it allows the search to avoid the exponential costs of searching a particularly expensive subtree. The issue in practice is how often this hap-

pens and how much work is saved. In fact, lookahead could be very efficient for some SAT benchmarks as shown in [3].

The high pruning power, along with non-ignorable overhead, has made lookahead a somewhat controversial technique. There are two camps of SAT/ASP solvers. In one of them lookahead is employed during the search [3, 35, 39, 74], and in the other it is not (e.g. [38, 59, 72, 88]).

In this chapter, on one hand we show that for some extremely hard problem instances, lookahead can indeed significantly improve the search. On the other hand, we provide some characterizations and identify the representative benchmarks for which lookahead downgrades the search efficiency.

Based on these observations, we propose an *adaptive lookahead* mechanism and implement it in the ASP solver `SMODELS`. The resulting system is called `ASMODELS`, in which lookahead is carried out only when it tends to be beneficial to do so. Our experiments show that, adaptive lookahead adapts well to different search environments it is going through - it performs like (better than, for most cases) `SMODELS` with lookahead for the problems that benefit from the use of lookahead, and without lookahead for those problems for which the use of lookahead tends to slow down the search.

In the direction of efficiently using lookahead in ASP solvers, it is proposed in [38] and [60] to use lookahead in a limited manner, performing lookahead on a subset instead of all of the unassigned atoms. While [60] deals with normal logic programs, [38] focuses on disjunctive logic programs.

Adaptive lookahead deals with weight constraint programs (normal programs can be considered as special cases of weight constraint programs). The main difference between adaptive lookahead and the approach called *limited lookahead* in [60] is that adaptive lookahead turns on/off lookahead according to the observed information - the frequencies of conflicts and dead-ends discovered during the search. Limited lookahead on the other hand depends on the status of literals i.e. it chooses a literal for lookahead if assuming its value leads to at least one inference. A more detailed comparison will be given later in this chapter.

The general idea of adaptive constraint propagation is also of interest in the constraint satisfaction problem (CSP) [8, 37, 70]. The approach in [37] incrementally applies arc-consistency to obtain higher consistency and [8, 70] switch between different consistency algorithms. But none of them deals with the lookahead technique. The relationship between lookahead and some consistency techniques in CSP has been studied in [85]. Adap-

tive lookahead is similar to [8] in the sense that they both use some thrashing prediction mechanism instead of the domain information as in [70].

The section 4.2 presents the answer set computation in system SMODELS . Section 4.3 provides the characterizations by which we identify problems that run much slower with lookahead. The adaptive lookahead algorithm is given in Section 4.4. Section 4.5 provides experimental results. The summary and future directions are given in Section 4.6.

4.2 Answer set computation in SMODELS

4.2.1 Weight constraint rules

Following [74], the system SMODELS deals with *basic constraint rules* which include two type of rules

- A *weight* rule of the form

$$h \leftarrow l[a_1 = w_{a_1}, ..., a_m = w_{a_m}, \texttt{not}\ b_1 = w_{b_1}, ..., \texttt{not}\ b_n = w_{b_n}] \tag{4.1}$$

  where the lower bound l and all weights are non-negative.

- A *choice* rule of the form

$$\{h_1, ..., h_k\} \leftarrow a_1, ..., a_m, \texttt{not}\ b_1, ..., \texttt{not}\ b_n. \tag{4.2}$$

  Recall that a weight may be omitted when it is 1. The rule (4.2) is actually the short hand for

$$0[h_1, ..., h_k] \leftarrow m + n[a_1, ..., a_m, \texttt{not}\ b_1, ..., \texttt{not}\ b_n]. \tag{4.3}$$

A weight rule captures the lower bound condition for a single weight constraint. A choice rule encodes a conditional choice stating that if the body of the rule is satisfied, then any subset (including the empty one) can be selected from the set in the head but if the body is not satisfied, only the empty subset can be chosen.

Any weight constraint program can be translated to a set of basic constraint rules as pointed out by [74].

The following are short hand notations for frequently used special forms of basic constraint rules.

- A *cardinality* rule $h \leftarrow k[a_1, ..., a_m, \texttt{not}\ b_1, ..., \texttt{not}\ b_n]$ corresponds to

$$h \leftarrow k[a_1 = 1, ..., a_m = 1, \texttt{not}\ b_1 = 1, ..., \texttt{not}\ b_n = 1]. \tag{4.4}$$

- A *normal* rule $h \leftarrow a_1, ..., a_m, ..., \texttt{not}\ b_1, ..., \texttt{not}\ b_n$ corresponds to

$$h \leftarrow m + n[a_1, ..., a_m, \texttt{not}\ b_1, ..., \texttt{not}\ b_n]. \tag{4.5}$$

- An *integrity constraint* $\leftarrow a_1, ..., a_m, \texttt{not}\ b_1, ..., \texttt{not}\ b_n$ corresponds to

$$f \leftarrow n + m + 1[a_1, ..., a_m, \texttt{not}\ b_1, ..., \texttt{not}\ b_n, \texttt{not}\ f]. \tag{4.6}$$

  where f is a new atom used only in integrity constraints. Notice that if a program contains such an integrity constraint, then it cannot have a stable model S such that $\{a_1, ..., a_n\} \subseteq S$ and $\{b_1, ..., b_m\} \cap S = \emptyset$.

4.2.2 Lookahead in SMODELS

Notations: Given a program P, $Literal(P)$ denotes the set of literals appearing in P. A *conflict* in a set of literals is a pair of complement literals, e.g. a and $\texttt{not}\ a$. A set of literals is *consistent* if there is no conflict in the set. A *partial assignment* is a consistent subset of $Literal(P)$. $At(P)$ denotes the set of atoms appearing in P (excluding the special atom $\perp$). The expression $not(\texttt{not}\ a)$ is identified with a, and $not(a)$ is $\texttt{not}\ a$. Given a set of literals A, $A^+ = \{a \mid a \in A\}$ and $A^- = \{a \mid \texttt{not}\ a \in A\}$. Given a set of atoms S, we define $not(S) = \{\texttt{not}\ a \mid a \in S\}$. A *choice point* is a point during search where the branching heuristic picks a literal to assign a truth value to it. In the literature, this is also referred to as *making a decision*.

Given a program P, SMODELS begins with an empty partial assignment, and attempts to extend the current partial assignment possibly to an answer set. Before making a decision, SMODELS performs constraint propagation. When lookahead is not involved, constraint propagation is carried out by a function called $expand(P, A)$, where P is a program and A a partial assignment. When lookahead is employed, constraint propagation is carried out as follows: for each unassigned atom x, assume a truth value for it, if it leads to a conflict, then x gets the opposite truth value. This process continues, repeatedly, until no atom can be fixed a truth value by lookahead (See Algorithms 1 and 2 for the details).

Algorithm 1 lookahead(P, A)

```
1: repeat
2:     A' = A
3:     A := lookahead_once(P, A)
4: until A = A'
5: return A
```

Algorithm 2 lookahead_once(P,A)

1: B := $At(P) - Atoms(A)$
2: B := $B \cup \texttt{not}\ (B)$
3: **while** $B \neq \emptyset$ **do**
4: Take any literal $x \in B$
5: A' := $expand(P, A \cup \{x\})$
6: B := $B - A'$
7: **if** conflict(P,A') **then**
8: **return** $expand(P, A \cup \{\texttt{not}\ (x)\})$
9: **return** A

In lookahead_once, the function $conflict(P, A)$ returns true if $A^+ \cap A^- \neq \emptyset$ and false otherwise. We say a conflict is detected if $conflict(P, A)$ returns true.

Truth values are propagated in lookahead by function $expand(P, A)$. The function $expand(P, A)$ consists of two functions: $Atleast(P, A)$ and $Atmost(P, A)$.

Algorithm 3 expand(P, A)

1: **repeat**
2: A':= A
3: A := $Atleast(P, A)$
4: A := $A \cup \{\texttt{not}\ x \mid x \in At(P)\ and\ x \notin Atmost(P, A)\}$
5: **until** $A = A'$
6: **return** A

Below, we follow the description given in [74].

Let P be a set of rules and let A be a set of literals. For a weight rule $r \in P$ of the form (4.1) and a set of literals B, let

$$min_r(B) = \begin{cases} \{h\} & \text{if } \sum_{a_i \in B} w_{a_i} + \sum \texttt{not}\ _{b_i \in B} w_{b_i} \geq l; \\ \emptyset & \text{otherwise} \end{cases}$$

be the *inevitable consequence* of B. Similarly, for a choice rule $r \in P$ of the form (4.2), let

$$min_r(B) = \begin{cases} \{h_1, ..., h_k\} \cap B & \text{if } a_1, .., a_m, \texttt{not}\ b_1, ..., \texttt{not}\ b_n \in B; \\ \emptyset & \text{otherwise.} \end{cases}$$

In addition, for a weight rule $r \in P$ let

$$max_r(B) = \begin{cases} \{h\} & \text{if } \sum_{a_i \notin B^-} w_{a_i} + \sum \texttt{not}\ _{b_i \notin B^+} w_{b_i} \geq l; \\ \emptyset & \text{otherwise} \end{cases}$$

be the *possible consequence* of B. For a choice rule $r \in P$ let

$$max_r(B) = \begin{cases} \{h_1, ..., h_k\} - B^- & \text{if } a_1, ..., a_m \notin B^- \text{ and } b_1, ..., b_n \notin B^+; \\ \emptyset & \text{otherwise.} \end{cases}$$

Set

$$f_P^1(B) = \{a \in min_r(B) \mid a \in At(P) \; and \; r \in P\}, \tag{4.7}$$

$$f_P^2(B) = \{\texttt{not}\ a \mid a \in At(P) \; and \; \forall r \in P, a \notin max_r(B)\}, \tag{4.8}$$

$$f_P^3(B) = \{\texttt{not}\ x \mid \exists a \in B \; such \; that \; a \in max_r(B) \; for \; only \; one \; rule \; r \in P \; and \; a \notin max_r(B \cup \{x\})\} \tag{4.9}$$

$$f_P^4(B) = \{\texttt{not}\ x \mid \exists \texttt{not}\ a \in B \; and \; r \in P \; such \; that \; a \in min_r(B \cup \{x\})\} \tag{4.10}$$

and

$$f_P^5(B) = \begin{cases} At(P) & \text{if } B^+ \cap B^- \neq \emptyset; \\ \emptyset & \text{otherwise.} \end{cases}$$

Define $Atleast(P, A)$ as the least fixpoint of

$$f(B) = A \cup B \cup f_P^1(B) \cup f_P^2(B) \cup f_P^3(B) \cup f_P^4(B) \cup f_P^5(B). \tag{4.11}$$

Example 4.1. Let P be the program

$$a \leftarrow b, \texttt{not}\ c \tag{4.12}$$

$$d \leftarrow \texttt{not}\ a \tag{4.13}$$

$$e \leftarrow \texttt{not}\ b \tag{4.14}$$

We will compute $A = Atleast(P, \{d\})$. Since c does not appear in the head of any rule in P, $\texttt{not}\ c \in A$ by f_P^2. As $d \in A$, $\texttt{not}\ a \in A$ by f_P^3. It follows that $\texttt{not}\ b \in A$ by f_P^4. Finally, $e \in A$ by f_P^1. Hence, $Atleast(P, \{d\}) = \{\texttt{not}\ a, \texttt{not}\ b, \texttt{not}\ c, d, e\}$. □

Let P be a set of rules and A a set of literals. For a weight rule $r \in P$ of the form (4.1) and a set of atoms B, let

$$f_r(B) = \begin{cases} \{h\} & \text{if } \sum_{a_i \in B - A^-} w_{a_i} + \sum_{b_i \notin A^+} w_{b_i} \geq l; \\ \emptyset & \text{otherwise.} \end{cases}$$

For a choice rule $r \in P$ of the form (4.2), let

$$f_r(B) = \begin{cases} \{h_1, ..., h_k\} & \text{if } a_1, ..., a_m \in B - A^- \text{ and } b_1, ..., b_n \notin A^+; \\ \emptyset & \text{otherwise.} \end{cases}$$

Define $Atmost(P, A)$ as the least fix point of $f(B) = \cup_{r \in P} f_r(B) - A^-$.

Example 4.2. Let P be the program

$$a \leftarrow \texttt{not}\ b \tag{4.15}$$

$$c \leftarrow \texttt{not}\ a \tag{4.16}$$

Then, $Atmost(P, \emptyset) = \{a, c\}$ but $Atmost(P, \{\texttt{not}\ a\}) = \emptyset$. □

4.3 Ineffectiveness of lookahead

We give two characterizations under which the use of lookahead is totally or nearly totally wasted. The first characterization describes the situations where no pruning is ever generated by lookahead in the course of solving some parts of a problem, which are called *easy sub-programs*. The second is called *spurious pruning*, which describes the situations where a literal added by lookahead to a partial assignment is immaterial to the rest of the search.

4.3.1 Easy sub-programs

Let P' and P be two ground programs, P' is called a *sub-program* of P if $P' \subseteq P$. A program P is said to be *easy* if any partial assignment can be extended to an answer set.

Proposition 4.1. Given a program P, if it is an easy program then lookahead is totally wasted in that, if $expand(P, A) = A$ then $lookahead(P, A) = A$, for any partial assignment A generated during the search.

Proposition 4.1 can be shown as follows. Let P be an easy program, and A a partial assignment. We have that $expand(P, A)$ generates no conflict, since any partial assignment can be extended to an answer set. The eighth line of Algorithm 2 is never executed, thus $lookahead\_once(P, A) = A$. Then we have $lookahead(P, A) = A$. This is the case where lookahead does not do anything more than what $expand(P, A)$ does.

Examples of easy programs include the pigeon-hole problem where the number of holes is greater than or equal to the number of pigeons; the graph coloring problem where there are as many colors as nodes, and in scheduling problems where the available resources are always more than what are required. These problems are easy, in the sense that a solution can be computed without backtracking, with or without lookahead.

A program for solving a hard problem may contain many nontrivial easy sub-programs for which lookahead in the search for answer sets is totally wasted. Consider the pigeon-hole problem, which is to put N pigeons into M holes so that there is at most one pigeon in a hole and every pigeon must take some hole. A typical weight constraint encoding is as below.

$$1\{pos(P, H) : hole(H)\}1 \leftarrow pigeon(P). \tag{4.17}$$
$$\{pos(P, H) : pigeon(P)\}1 \leftarrow hole(H). \tag{4.18}$$

Figure 4.1: A program for the pigeon-hole problem

The literals $pos(P, H) : hole(H)$ and $pos(P, H) : pigeon(P)$ in above program are *conditional literals*. A conditional literal is of the form

$$p(X) : q(X) \tag{4.19}$$

where $p(X)$ is a literal and $q(X)$ is a domain predicate. If the extension of q is $\{q(a_1),\ ...,\ q(a_n)\}$, the conditional literal $p(X) : q(X)$ is semantically equivalent to writing $p(a_1),\ ...,\ p(a_n)$.

The first rule specifies that each pigeon has to be placed in exactly one hole. The second rule states that each hole can only be occupied by at most one pigeon.

The problem can be very hard for DPLL-based solvers when $M = N - \delta$, for some small positive integer δ.

To see that lookahead is totally wasted in solving some easy sub-programs, consider any partial assignment A in the course of computing an answer set by `SMODELS` . Here, let A be such that some pigeons already take holes, e.g., suppose $pos(p_i, h_j) \in A$, for some i and j. Note that when $lookahead(P, A)$ is invoked, propagation by the $expand(P, A)$ function is completed, i.e., $expand(P, A) = A$. Thus, we have `not` $pos(p_i, h_k) \in A$, for any hole h_k different from h_j, due to the rule (4.18); and `not` $pos(p_m, h_j) \in A$, for any other pigeon p_m, due to the rule (4.17). Now, suppose $lookahead(P, A)$ is called, which calls $expand(P, A \cup \{pos(p', h')\})$ inside $lookahead\_once(P, A)$. This is to assume $pos(p', h')$ and attempt to derive a conflict. It can be verified that whenever there are at least three holes left unoccupied, no conflict can be found by lookahead.

In fact, lookahead begins to detect conflicts only when there are two holes left. When one of them is assumed to be taken by one pigeon, all the other pigeons that do not have a hole will be competing for the only remaining hole (by the definition of $expand(P, A)$, the first rule makes the unassigned $pos(p, h)$ true for the only remaining h). In other words, lookahead begins to find conflicts only when most of the holes have been assigned.

In this example, the sub-programs obtained by removing some facts about pigeons so that the resulting number of pigeons equals to the number of the holes are typical easy sub-programs that the search is going through.

Besides the pigeon-hole problem, the clique coloring problem is another typical problem that has nontrivial easy sub-programs [19]. The pigeon-hole and clique coloring problems, among others, are known to be exponentially difficult for any conventional resolution-based provers (including any DPLL implementation).

4.3.2 Spurious pruning

When lookahead finds a conflict, some search space is pruned. But this pruning may be immaterial to the rest of the search. Suppose, by an invocation of lookahead, a literal, say l, is added to the current partial assignment A. The addition of l may not contribute to further constraint propagation. This can be described by an equation

$$expand(P, A \cup \{l\}) = expand(P, A) \cup \{l\}$$

for any partial assignment A generated during search, such that the superset returned by $expand(P, A \cup \{l\})$ is not an answer set. This is what we mean by *spurious pruning*. In this case, lookahead is unnecessary since the decision on l can be delayed to any later choice point.

We can use the number of calls to the $expand(P, A)$ function during search to measure the effectiveness of lookahead. Suppose N_{lh} and N_{nlh} denote the numbers of calls to $expand(P, A)$ in SMODELS with and without lookahead respectively. By the definition of spurious pruning, we can easily derive its effect as stated in the following proposition.

Proposition 4.2. Given a program P, if all the pruning by lookahead are spurious, then $N_{nlh} \leq N_{lh}$.

An example where the search only generates spurious pruning is the Hamiltonian cycle problem for complete graphs. Given a graph, a Hamiltonian cycle of the graph is a path that visits each vertex of the graph exactly once and returns to the starting vertex. A complete graph is a graph in which every pair of distinct vertices is connected by an edge. It is known that solving the Hamiltonian cycle problem for complete graphs can be very hard for ASP solvers [43]. A typical program is given in Figure 4.2, which is taken from [14].

$$\leftarrow \quad 2\{hc(X,Y) : edge(X,Y)\}, node(Y). \tag{4.20}$$
$$\leftarrow \quad 2\{hc(X,Y) : uedge(X,Y)\}, node(X). \tag{4.21}$$
$$reach(U) \leftarrow \quad edge(V,U),\ hc(V,U),\ reach(V),\ \texttt{not}\ initial(V). \tag{4.22}$$
$$reach(U) \leftarrow \quad edge(V,U),\ hc(V,U),\ initial(V). \tag{4.23}$$
$$\leftarrow \quad vertex(V),\ \texttt{not}\ reach(V). \tag{4.24}$$

Figure 4.2: A program for the Hamiltonian cycle problem

The rules (4.20), (4.21) ensure that for each node exactly one outgoing and incoming arc belong to the path. The rules (4.22), (4.23) and (4.24) state that the path forms a cycle

which visits all nodes and returns to the initial node. The literal $hc(u, v)$ being true in an answer set means $edge(u, v)$ (from u to v) belongs to a Hamiltonian cycle.

Consider a complete graph on nodes $v_1, v_2, ..., v_n$, for some sufficiently large n. Suppose v_1 is the initial node and A is the current partial assignment, where a path $v_1 \leftarrow v_2 \leftarrow v_3$ is established. Note that `not` $hc(v_2, v_3) \in expand(P, A)$, due to the rule (4.21). At this point, in the call $lookahead(P, A)$, $expand(P, A \cup \{hc(v_1, v_3)\})$ will derive a conflict, since there must be a v_i, $i \geq 4$, such that both $hc(v_1, v_i)$ and `not` $hc(v_1, v_i)$ are derived due to rule (4.23) and rule (4.20), respectively. Then, `not` $hc(v_1, v_3)$ is added to A. However, it is clear that the addition of `not` $hc(v_1, v_3)$ to A is immaterial to the rest of the search, and choosing any other $hc(v_i, v_j)$ is guaranteed to lead to an answer set.

4.4 Adaptive lookahead

4.4.1 Algorithm

Adaptive lookahead is designed to avoid lookahead when its use tends to be ineffective. Two pieces of information are useful for this purpose. One is the number of conflicts and the other is the number of *dead-ends* detected during the search. In the SMODELS system, conflicts are generated by *failed literals* whose addition to the current partial assignment causes both a literal and its negation to be included in the partial assignment by constraint propagation. Note that conflicts do not necessarily mean that backtracking is needed since the negation of the failed literal may be consistent with the current partial assignment. A dead-end is a point during the search where both a literal and its negation are failed literals. In this case backtracking is needed.

The idea in adaptive lookahead is that if after some runs, the conflicts have been rare, it is likely the search is in a space where pruning is insignificant, and likely to remain so for some time to come, so lookahead is turned off; if dead-ends have been frequently encountered after some runs, it is likely that the search has entered into a space where pruning can be significant, so lookahead is turned on.

SMODELS with adaptive lookahead is called ASMODELS (Algorithm 4). The control of lookahead is realized by manipulating two scores, $look\_score$ and $dead\_end\_counter$. The $look\_score$ is initialized to be some positive number, then deducted each time lookahead does not detect any conflict. When it becomes zero, lookahead will be turned off. The $dead\_end\_counter$ is initialized to be zero and increased each time a dead-end is encountered. Lookahead will be turned on if $dead\_end\_counter$ reaches some thresh-

old. The counter $look\_score$ will be reset after lookahead is turned on and the counter $dead\_end\_counter$ will be reset after lookahead is turned off.

In addition to the on/off control, lookahead will never be employed in the later search processes if it cannot detect any conflicts after a number of atoms have been assigned. This is because the search efficiency cannot be improved much by lookahead if the conflicts it detects only happen in late stages of the search. The lateness is measured by a ratio of number of assigned literals to all literals in the program. When the ratio gets to a threshold before any conflict was found by lookahead, lookahead will be shut down permanently

The initial value of $look\_score$, the amounts of increase and decrease, and the thresholds are determined empirically. We set the amount of increase/decrease to 1, $look\_score$ to 10, the thresholds of $dead\_end\_counter$ and $ratio$ to 1 and 0.8 respectively. Note that there may not be a setting of these parameters that works universally well for all kinds of problems. The current setting is effective for the problems that we have experimented on and likely to work well for similar problems.

The correctness of the search algorithm of SMODELS has been proved in [73]. ASMODELS does not change the search process of the algorithm, hence its correctness is guaranteed. ASMODELS makes use of the existing data structures in the implementation of SMODELS . It has the same computational complexity as the SMODELS algorithm.

4.4.2 Comparison with *limited lookahead*

The idea of limited lookahead [60] is that if assuming a literal to be true does not lead to any inference, lookahead is guaranteed to be wasted. As such, lookahead is performed only on what are called *propagating literals*, the literals whose assignment leads to at least one inference.

The existence of some inferences is a condition that appears to be too weak. It is easier to incur inferences than a conflict or dead-end, so performing lookahead on propagating literals may not generate any space pruning. The pigeon-hole problem is an illustrative example. It is known to be hard when the number of pigeons is greater than the number of holes by one. Suppose we have 10 pigeons and 9 holes, and the current partial assignment is $A = \{pos(1,1)\}$. Consider the following rule

$$\bot \leftarrow pos(2,2), pos(2,3).$$

Both $pos(2,2)$ and $pos(2,3)$ will be identified as a propagating literal. Because they are unassigned, the head of the rule is $false$; assuming either $pos(2,2)$ or $pos(2,3)$ to be

Algorithm 4 ASMODELS (P, A, $look$, $shut\_down$)

```
1:  reset dead_end_counter
2:  A ← expand(P, A)
3:  if (look) then
4:     A ← lookahead(P, A)
5:     if (no conflict detected by lookahead) then
6:        decrease look_score
7:        if (look_score = 0) then
8:           look ← false   {lookahead is turned off}
9:  if (conflict()) then
10:    conflict_found ← true   {a conflict is found}
11:    if (a dead-end is found) then
12:       increase dead_end_counter
13:       if (!look and dead_end_counter > threshold_1 and !shut_down) then
14:          look ← true   {lookahead is turned on}
15:          reset look_score
16:    return false
17: else if (A covers At(P)) then
18:    return true{A^+ is a stable model}
19: else
20:    ratio ← ||A|| / ||Literal(P)||
21:    if (ratio > threshold_2 and !conflict_found) then
22:       shut_down ← true  {shut down lookahead in later search}
23:    choose a new atom x to assign
24:    if (ASMODELS (P, A ∪ {x}, look, shut_down)) then
25:       return true
26:    else
27:       return ASMODELS (P, A ∪ {not x}, look, shut_down)
```

true allows us to infer the other is false. But as we have commented in Subsection 4.3.1, lookahead on pos(2,2) or pos(2,3) does not lead to any conflict because there are more than two holes left unoccupied. Actually, the same situation arises for any $pos(i, j)$, where $2 \leq i \leq 10$, $2 \leq j \leq 9$. It is easy to see that limited lookahead reduces very little overhead in this case.

The above observation is also applicable to the computation of Hamiltonian cycles on complete graphs (this is due to the rules (4.21) and (4.22) of the program given in Subsection 4.3.2).

Instead of identifying propagating literals, adaptive lookahead uses the information developed during the search about conflicts and dead-ends. In solving the pigeon-hole instances like the one above, adaptive lookahead will turn off lookahead after some invocations since it cannot detect any conflicts (w.r.t. the current partial assignment A). For the problem of computing Hamiltonian cycles on a complete graph, lookahead is also turned off since the dead-end is rarely encountered because every choice point during the search may lead to a solution.

4.5 Experiments

To test how well adaptive lookahead works, we have conducted a series of experiments. In Subsection 4.5.1, the general performance of ASMODELS is compared with SMODELS using random logic programs, which are provided as benchmarks for the first ASP solver contest [14]. The experiments reported in Subsection 4.5.2 serve three purposes. First, they confirm our findings of the problems where the performance is significantly deteriorated by the use of lookahead; second, they show that lookahead tends to be very effective for a number of hard problems, especially for the problems that lie in the known regions of phase transition; third, they suggest that adaptive lookahead behaves as if it "knows" when to employ lookahead and when not to. Finally in Subsection 4.5.3, we report experimental results on a deployed application, the USA-Advisor project.

We run SMODELS 2.32 with and without lookahead[1], and ASMODELS , respectively. By default, SMODELS runs with lookahead. All of the experiments are run on Red Hat Linux AS release 4 with 2GHz CPU and 2GB RAM. The cutoff time is set to two hours. In the tables that report the experimental results, we will use the symbol "-" to indicate that no result has been produced within two hours of running time.

[1]The SMODELS system can be downloaded from `http://www.tcs.hut.fi/Software/smodels/`.

We shall mention that the branching heuristic in SMODELS is related to lookahead. The heuristic value of each atom is initialized according to how many rules they are involved in. During the computation, the heuristic value of an atom is updated according to how many other atom's values can be determined by the addition of the atom to the current partial assignment. With lookahead, SMODELS computes the heuristic value of every atom each time lookahead is invoked. Without lookahead, the heuristic value will not be computed until the atom is chosen to be assigned. So the effectiveness of lookahead should be considered on both the search procedure and the branching heuristics.

4.5.1 Random logic programs

There are two sets of random logic programs provided in [14]. The first is just called *random logic programs* (RLPs) and the second *random non-tight logic programs* (RNLPs). We run ASMODELS and SMODELS on both of them. The results are plotted in Figures 4.4 and 4.5. The graphs show the ratio of the running time of ASMODELS to SMODELS . The instances are arranged in descending order of this ratio.

For RLPs, ASMODELS outperforms SMODELS on all of them except for three. The average improvement is 35% and maximum is 62%.

For RNLPs, ASMODELS is faster than SMODELS except for five. The average improvement is 34% and maximum is 66%. This result is better than [60], where the benefit of limited lookahead is not so obvious.

For these benchmarks, ASMODELS turned on and off lookahead once, respectively, during the search. The above instances include both solvable and unsolvable ones. No discernible difference between the performances of ASMODELS on them is observed.

4.5.2 Adaptiveness of adaptive lookahead

The logic programs used in this section (except the one for the 3-SAT problem) are taken from [61].

Cases that benefit from lookahead

Graph coloring The problem is to find an assignment of one of 3 colors to each vertex of a graph such that vertices connected with an edge do not have the same color. We use the weight constraint program in Example 1.3.

We use Culberson's flat graph generator [15] to generate graph instances. By this generator, each pair of vertices is assigned an edge with independent identical probability p.

We use the suggested value of p to sample across the "phase transition" region [12]. The number of nodes of the graph is 400. For each data point, the average running time of 100 instances is reported.

The extremely hard instances happen when p is in [0.018, 0.020]. In this region, lookahead speeds up the search drastically (Table 4.1). Note that SMODELS without lookahead cannot finish the search in the cutoff time (for $P = 0.018$, it runs for two days without completion).

It is known that every DPLL-based solver spends the majority of its time on BCP; in the case of SMODELS , on the *expand* function. The savings by lookahead can also be measured by the number of calls to the *expand* function[2]. We collect the number of *expand* calls. The result is consistent with the running time. The reduction in the number of calls to *expand* is orders of magnitudes, by using lookahead (adaptively) in the phase transition region. This is shown in Fig. 4.6. The savings of calls to the *expand* function can also be observed for other benchmarks where lookahead speeds up the search.

3-SAT The problem is to determine the satisfiability of a Boolean formula in conjunctive normal form (CNF), where each clause in the conjunction consists of at most three literals, i.e., a CNF of n clauses is of the form

$$x_{11} \vee x_{12} \vee x_{13} \bigwedge ... \bigwedge x_{n1} \vee x_{n2} \vee x_{n3} \tag{4.25}$$

A CNF of form (4.25) can be translated to a logic program in the following way. For each literal x_{ij}, two rules as below are introduced.

$$x_{ij} \quad \leftarrow \quad \texttt{not}\ neg\_x_{ij}. \tag{4.26}$$

$$neg\_x_{ij} \quad \leftarrow \quad \texttt{not}\ x_{ij}. \tag{4.27}$$

For each clause c_i, a literal sat_c_i and the following rules are included in the program.

$$sat\_c_i \quad \leftarrow \quad x_{ik}. \quad k = 1, 2, 3. \tag{4.28}$$

Note that for a negative literal $\neg x_{ik}$ in a CNF, we have $\texttt{not}\ x_{ik}$ instead of x_{ik} in the formula (4.28).

In addition to the rules from (4.26) to (4.28), a new literal sat and the following rules are included in the program.

$$sat \quad \leftarrow \quad sat\_c_1,\ ...\ , sat\_c_1. \tag{4.29}$$

$$\bot \quad \leftarrow \quad \texttt{not}\ sat. \tag{4.30}$$

[2]In comparison, the number of choice points is usually not a good indicator, as a reduction may be achieved in the expense of a huge overhead.

It can be verified that the CNF of form (4.25) is satisfiable if and only if the logic program consists of rules from (4.26) to (4.30) has an answer set.

Random 3-SAT is another well-known problem with phase transition. The instances are extremely hard when the ratio $\frac{c}{a}$ is around 4.3 [58], where c is the number of clauses, and a the number of variables in the formula.

We fixed the number of variables to 300 and randomly generated conjunctive normal formulas by the ratio from 1 to 10. For each ratio, we generate 10 instances and transform them to logic programs. We compute answer sets for the programs and report the average running time in Table 4.2. Similar to the graph coloring problem, lookahead substantially speeds up the search in the hard region.

Queens The queens problem is to place n queens to an $n \times n$ board so that no queen checks against any other queen. It is a standard example in the CSP literature, where the constraints have a natural representation as logic program rules [61].

The experimental results are given in Table 4.3. It is clear that SMODELS with lookahead is significantly faster than SMODELS without lookahead; further, adaptive lookahead is tens of times faster than the original lookahead.

Blocks-world The problem is to re-arrange a number of blocks on a table from an initial configuration to a goal configuration. The blocks world problem is a standard planning benchmark.

The problem instances are generated as follows. For n blocks, b_1,...,b_n, the initial configuration is b_1 on the table and b_{i+1} on b_i for $i = 1...n-1$. The goal is b_n on the table, b_1 on b_n and b_{i+1} on b_i for $i = 1...n-2$. We use this setting because, under it, each block in the initial state has to be moved to get to the goal state, so the problem turns out to be nontrivial. The minimum number of steps needed is $2n-2$.

The results are reported in Table 4.4. The upper sub-row of each row is the solvable case and the lower one is the unsolvable case (similarly in Table 4.5).

Gripper The goal of the gripper problem is to transport balls from room $R1$ to room $R2$. To accomplish this, a robot with two grippers is allowed to move from one room to the other, pick up, and put down a ball. Each gripper of the robot can hold one ball at a time.

In our experiments two kinds of settings are used. In the first, all of the balls are in $R1$ initially and should be $R2$ in the goal state. In the second, there is an equal number of balls in $R1$ and $R2$ initially and in the goal state, balls initially in $R1$ are in $R2$ and vice versa. The results are reported in Table 4.5, where the first two columns are the number of balls in each room initially, and the third is the number of steps allowed.

The results suggest that lookahead generates more performance gains when the instances become harder especially for the unsolvable instances.

For all of the problems in this section, ASMODELS performs as well as or better than SMODELS with lookahead (Tables 4.1, 4.2, 4.3, 4.4, and 4.5). For most instances of the graph coloring, queens, 3-SAT, and gripper problems, lookahead is turned on and off once, respectively. The blocks world problem is interesting. For the solvable instances, ASMODELS turns off lookahead automatically and performs several times faster than SMODELS . Especially for the instance where the blocks number is 16, lookahead is turned off twice and on once. A drastic improvement in the performance by this action can be observed. This suggests that turning off lookahead when it is unnecessary can effectively speed up the search even if it is useful most of the time.

Seating The seating problem is to generate a sitting arrangement for a number of guests, with m tables and n chairs per table. Guests who like each other should sit at the same table; guests who dislike each other should not sit at the same table.

In [1] the seating problem was chosen to evaluate the performance of DLV , a well-known ASP solver. The problem can be naturally encoded as an weight constraint program as given in Figure 4.3.

$$1[at(P,T) : table(T)]1 \leftarrow person(P).$$
$$\bot \leftarrow table(T), likes(P1,P2), at(P1,T), \texttt{not } at(P2,T).$$
$$\bot \leftarrow table(T), dislikes(P1,P2), at(P1,T), at(P2,T).$$
$$\bot \leftarrow n+1[at(P,X) : person(P)], table(X).$$

Figure 4.3: A program for the seating problem

The first rule says each guest sits at a table. The next two rules guarantee that guests who like each other sit at the same table and guests who dislike each other does not sit at the same table. The last rule ensures the number of people sit at the table cannot be more than the number of chairs around the table.

We consider 5 and 10 seats per table, with increasing numbers of tables and persons (with the number of persons equals to the number of tables times the number of seats per table), respectively. For each problem size, i.e., seats/tables configuration, we test the classes with different numbers of like and dislike constraints that are used by DLV^A. They are : 1) no like/dislike constraints at all; 2) 25% like constraints; 3) 25% like and 25% dislike constraints; 4) 50% like constraints; 5) 50% like and 50% dislike constraints, where the percentages are relative to the maximum number of dislike (like, respectively) constraints.

For the seating problem, lookahead significantly improves the search efficiency. SMODELS

without lookahead couldn't solve any instance in two hours; all of them can be solved in tens of seconds with lookahead. ASMODELS is about 2-3 times faster than SMODELS with lookahead (Table 4.6).

Cases that suffer from lookahead

For the problems with easy sub-programs, like pigeon-hole, lookahead can detect conflicts only near the end of the search. Employing lookahead for these problems makes the search several times slower (Table 4.7). ASMODELS is 2 times faster than SMODELS without lookahead and about 10 times faster than SMODELS with lookahead.

As for computing Hamiltonian cycles on complete graphs, lookahead cannot reduce the depth of the search tree while bringing on extra constraint propagation. The experimental results (Table 4.8) show SMODELS can be hundreds of times slower than SMODELS without lookahead.

For the above two problems, ASMODELS works largely as SMODELS without lookahead. Its performance is slightly better than SMODELS without lookahead for the pigeon-hole problem (Table 4.7). ASMODELS turned off lookahead permanently after some literals are assigned during the process of solving this problem.

For the Hamiltonian cycle problem, the performance of ASMODELS is in between SMODELS and SMODELS without lookahead - it is about 20 to 30 times faster than SMODELS and 2 to 10 times slower than SMODELS without lookahead (Table 4.8). We notice that if the parameter $look\_score$ is set to be 2 and $ratio$ 0.1, ASMODELS will perform two times faster than what we reported here but it is still several times slower than SMODELS without lookahead. The reason for that could be the effect of lookahead on branching heuristic as we mentioned before. More investigation on this is needed.

The Hamiltonian cycle problem on complete graphs has been extensively tested by a number of answer set systems in [43]. It turns out that for large instances, thousands of seconds are needed to solve them. As shown in Table 4.8, with adaptive lookahead, SMODELS can solve large instances in hundreds of seconds, and if lookahead is completely shut down, only tens of seconds are needed to solve them.

4.5.3 Experiments on a real application

The *reaction control system* (RCS) is the system used to maneuver the Space Shuttle while it is in orbit. It consists of jets, fuel tanks, pipes, and valves used to deliver fuel to the jets, and associated circuitry required to control the system. In order for the Space Shuttle to

perform a given maneuver, a set of jets must be prepared to fire. To prepare a jet to fire, an open, non-leaking path must be provided for the fuel to flow from pressurized fuel tanks to the jet. The flow of fuel is controlled by opening and closing valves. Valves are opened and closed by either having an astronaut flip a switch or by instructing the computer to issue special commands.

When everything is operating correctly, there are pre-scripted plans for each maneuver. When some components of the system such as switches, valves, or circuits fail, a planner is needed to generate plans to finish the required maneuver. The system USA-Advisor is such a planner with a user-friendly interface, developed by Balduccini et al. [4]. The reasoning module that is responsible for the plan generation in USA-Advisor is `SMODELS`.

Following [4], by *test instances* we mean a maneuver to be performed by the shuttle together with a collection of system faults. We randomly generated 10 groups of instances by setting the target maneuver. Each group contains 20 instances, generated by randomly set 5 system faults, of which 2 are mechanical and 3 are electrical (this situation is referred to as the most interesting one from the standpoint of the USA-Advisor experts). The reported running time of each group is the average running time of the instances in the group.

From the experiment results (Table 4.9), one can see that there is no major difference between the performance of `SMODELS` with and without lookahead. But `ASMODELS` is better than both of them.

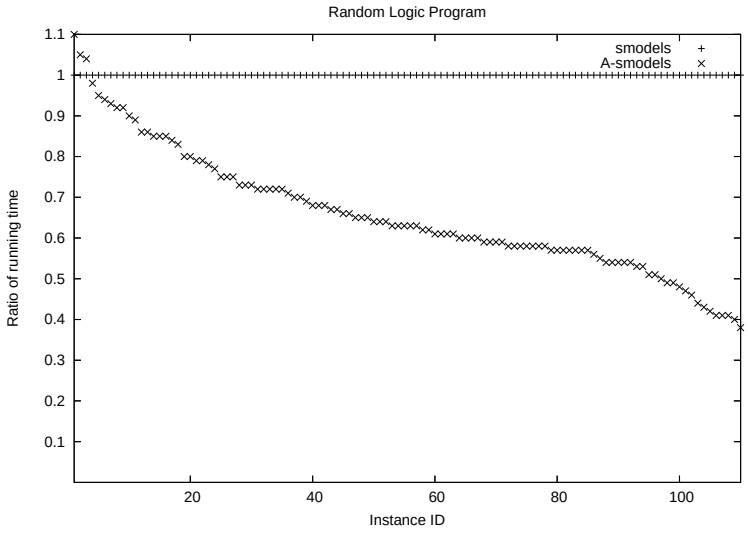

Figure 4.4: Random logic programs

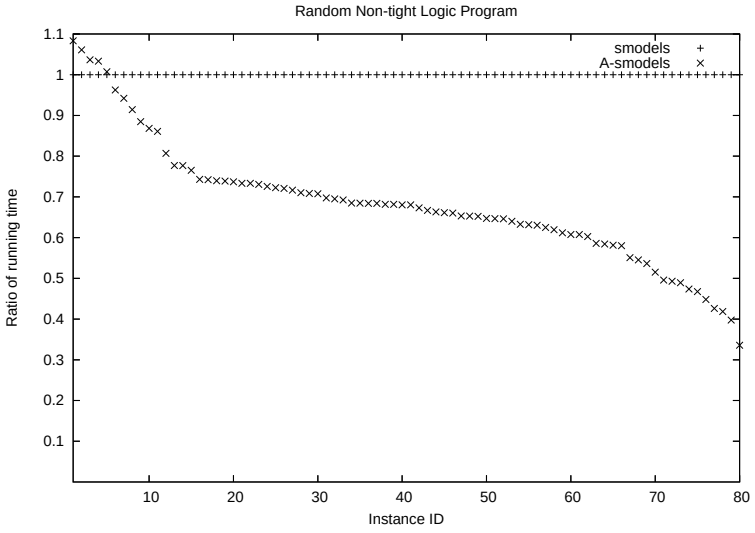

Figure 4.5: Random non-tight logic programs

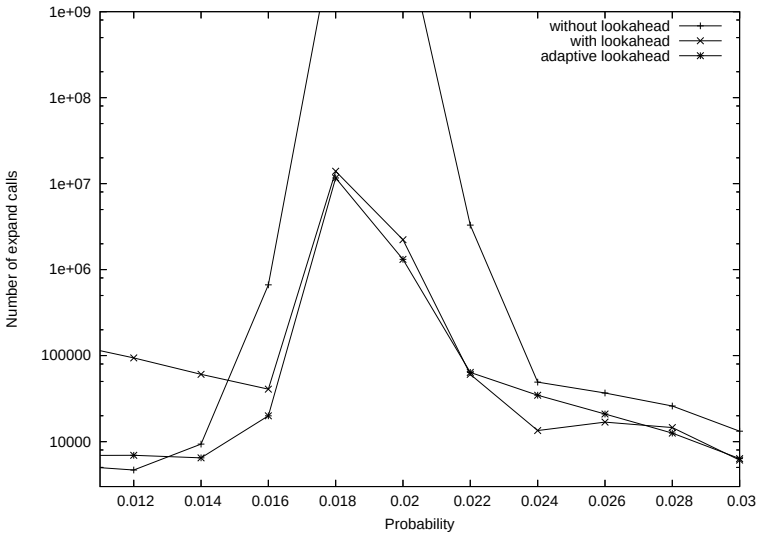

Figure 4.6: $expend$ calls in graph coloring

p	No Lookahead	Lookahead	A-Lookahead
.014	0.33	0.33	0.09
.015	1.44	0.33	0.15
.016	53.55	0.37	0.32
.017	97.12	0.24	0.12
.018	–	209.68	183.41
.019	–	43.30	40.15
.020	–	55.38	23.34
.021	2649.80	9.47	10.14
.022	299.61	0.96	1.06
.023	18.89	0.92	1.05
.024	4.19	0.25	0.67
.025	3.52	0.23	0.22
.026	3.26	0.26	0.22
.027	1.97	0.21	0.29
.028	2.13	0.21	0.26
.029	2.06	0.17	0.29
.030	1.25	0.19	0.22

Table 4.1: Graph coloring ("-" indicates no result is generated within two hours of running time)

c/a	No Lookahead	Lookahead	A-Lookahead
1	0.01	0.04	0.01
2	0.02	0.05	0.02
3	0.08	0.06	0.05
4.3	6475.43	137.64	139.88
5	650.85	20.20	19.72
6	45.09	1.64	1.55
7	7.12	0.55	0.54
8	3.15	0.38	3.25
9	1.68	0.27	1.61
10	0.94	0.21	1.04

Table 4.2: Random 3-SAT

n	No Lookahead	Lookahead	A-Lookahead
14	32.975	0.43	0.18
15	205.898	0.86	0.27
16	1471.878	2.35	0.37
17	–	5.36	0.74
18	–	9.08	2.11
19	–	3.92	2.4
20	–	54.33	2.10
21	–	98.49	5.73
22	–	852.09	47.60
23	–	2465.09	418.22
24	–	1009.556	76.31
25	–	–	370.56
26	–	–	3349.53
27	–	–	–
28	–	–	4051.11

Table 4.3: Queens

n	s	No Lookahead	Lookahead	A-Lookahead
11	20	13.58	4.62	2.02
	19	16.52	1.56	1.56
12	22	37.49	9.38	3.36
	21	30.70	2.42	2.41
13	24	106.31	17.26	4.85
	23	62.31	3.81	3.76
14	26	165.88	30.49	7.34
	25	359.65	5.45	5.46
15	28	335.11	53.07	10.54
	27	4673.35	7.35	7.35
16	30	375.20	6276.28	14.66
	29	–	10.15	10.50
17	32	1197.65	145.59	21.24
	31	701.86	16.48	16.40
18	34	–	145.48	29.28
	33	–	21.42	21.39

Table 4.4: Blocks-world problem

R1	R2	s	No Lookahead	Lookahead	A-Lookahead
4	0	7	0.10	0.43	0.41
		6	0.09	0.17	0.17
5	0	11	8.31	77.35	64.94
		10	1824.55	189.40	97.12
6	0	11	263.75	1117.02	1084.31
		10	2345.49	490.94	487.65
3	3	11	0.70	15.80	15.98
		10	77.93	15.64	15.52
4	4	12	1749.35	419.42	412.64
		11	5463.57	175.61	175.73

Table 4.5: Gripper problem

p	h	No Lookahead	Lookahead	A-Lookahead
5	30	–	2.67	1.40
5	35	–	4.89	2.47
5	40	–	8.50	4.04
5	45	–	13.99	6.54
5	50	–	20.96	9.24
10	30	–	8.94	3.33
10	35	–	16.88	5.72
10	40	–	28.43	8.89
10	45	–	47.12	13.71
10	50	–	71.94	20.74

Table 4.6: Seating

p	h	No Lookahead	Lookahead	A-Lookahead
5	4	0.01	0.01	0.00
6	5	0.00	0.03	0.00
7	6	0.05	0.15	0.00
8	7	0.03	0.11	0.02
9	8	0.27	0.92	0.13
10	9	2.56	8.58	1.27
11	10	28.27	91.19	13.86
12	11	337.32	1006.83	167.99

Table 4.7: Pigeon hole by weight constraint program

n	No Lookahead	Lookahead	A-Lookahead
30	0.46	3.62	1.07
40	1.11	16.90	3.19
50	2.25	64.92	8.57
60	3.91	183.59	19.74
70	6.31	467.28	40.18
80	9.58	986.85	74.02
90	13.77	1862.86	123.66
100	19.12	3522.24	194.24
110	25.56	6129.59	288.53
120	33.36	–	412.58

Table 4.8: Hamiltonian cycle

I	No Lookahead	Lookahead	A-Lookahead
1	1.24	1.31	0.84
2	0.75	0.95	0.75
3	1.39	1.23	0.82
4	0.63	0.63	0.58
5	2.50	1.46	0.93
6	0.81	1.70	0.72
7	1.07	1.30	0.89
8	1.21	1.30	0.89
9	1.27	1.26	0.90
10	1.94	1.59	1.04

Table 4.9: USA-Advisor

4.6 Conclusion

In this chapter, we have shown that lookahead could be a burden in, as well as an accelerator to, the DPLL search. This may offer some clues for the appropriate use of lookahead in constraint solving systems. By experiments, we show that lookahead may significantly speed up the answer set computation for hard problems, especially those in phase transition regions. We also analyze why lookahead sometimes slows down the search and characterize some reasons as embedded easy sub-programs and spurious pruning. We shall mention that this analysis is applicable to DPLL-based SAT solvers; that is, similar SAT encodings do not benefit from the use of lookahead under these situations.

Based on these observations, we propose an adaptive lookahead mechanism, by which the decision on whether lookahead is invoked or not is made dynamically upon the observed information developed during the search. It takes advantage of lookahead while avoiding the unnecessary overhead caused by it. We have conducted a series of experiments on benchmarks written as normal logic programs as well as logic programs with weight constraints. We have also applied adaptive lookahead to the planning component of the USA-Advisor project. Our experiments show that adaptive lookahead adapts well to different search environments.

An interesting direction is *selective lookahead*: instead of the entire set of unassigned atoms, some subset of it is selected by lookahead for testing. Limited lookahead is a kind of selective lookahead. But as we have shown, the selection of propagating literals may not be very effective for the purpose of space pruning. The heuristic for the selection of literals for testing in selective lookahead is worth of investigation in future work.

After this work had been done, the ASP system `CLASP` was developed. It is currently admitted as the fastest solver, where some state-of-the-art techniques from SAT solvers are used, such as *conflict-directed learning*, *restart*, and *deletion of recorded conflicts*. `ASMODELS` is usually not as fast as `CLASP`. But we find that, for the pigeon-hole problem, `ASMODELS` is several times faster than `CLASP`. The reason for that, and the combination of lookahead with the established techniques in SAT solvers is of interest for further study.

Chapter 5

Answer set semantics for aggregate programs

There are different semantics proposed for aggregate programs [25, 27, 66, 76]. Among them, the semantics based on conditional satisfaction is considered the most *conservative*, in the sense that any answer set under this semantics is an answer set under others, but the converse may not hold [77]. We take this semantics as the semantics for aggregate programs.

In the next section, we present the semantics based on conditional satisfaction, called *answer set semantics* for aggregate programs. Then, we briefly introduce other semantics and provide a comparison to them.

5.1 Answer set semantics

Following [76], we define the syntax and semantics of aggregate programs.

An aggregate is a constraint on sets of atoms taking the form

$$aggr(\{X \mid p(X)\}) \ \mathtt{op} \ Result \tag{5.1}$$

where $aggr$ is an *aggregate function*. The standard aggregate functions are SUM, $COUNT$, AVG, MAX, and MIN. The set $\{X \mid p(X)\}$ is called an *intentional set*, where p is a predicate, and X is a variable, which takes its value from a set $D(X) = \{a_1, ..., a_n\}$, called *variable domain*. The comparison operator op is from $\{=, \neq, <, >, \leq, \geq\}$ and $Result$ is either a variable or a numeric constant.

The *domain* of an aggregate A, denoted $Dom(A)$, is the set of atoms $\{p(a) \mid a \in D(X)\}$. The size of an aggregate is $|Dom(A)|$.

Let M be a set of atoms. M is a *model* of an aggregate A, denoted $M \models A$, if $aggr(\{a \mid p(a) \in M \cap Dom(A)\}) \ \mathtt{op} \ Result$ holds, otherwise M does not satisfy A,

denoted $M \not\models A$.

Example 5.1. Let $A = SUM(\{X|p(X)\}) \geq 1$ be an aggregate, where $D(X) = \{-1, 0, 1\}$. Let $M_1 = \{p(0), p(1)\}$ and $M_2 = \{p(-1), p(0)\}$ be two sets. We have $M_1 \models A$ and $M_2 \not\models A$. □

An aggregate program is a set of rules of the form

$$h \leftarrow A_1, ..., A_n \tag{5.2}$$

where h is an atom and $A_1, ..., A_n$ are aggregates[1]. For a rule r of the form (5.2), $hd(r)$ and $bd(r)$ denote h and $\{A_1, ..., A_n\}$, respectively.

The definition of *answer set* of aggregate programs is based on the notion of *conditional satisfaction.*

Definition 5.1. Let A be an aggregate, R and S two sets of atoms. R conditionally satisfies A, w.r.t. S, denoted $R \models_S A$, iff $R \models A$ and for every set I s.t. $R \cap Dom(A) \subseteq I \subseteq S \cap Dom(A)$, $I \models A$.

Example 5.2. Let $A = COUNT(\{X|p(X)\} \leq 2$ be an aggregate, where $D(X) = \{1, 2, 3, 4\}$. Let $R = \{p(1)\}$, $S_1 = \{p(1), p(2)\}$, and $S_2 = \{p(1), p(2), p(3), p(4)\}$. Then $R \models_{S_1} A$, since $R \models A$ and $S_1 \models A$; $R \not\models_{S_2} A$, since for the set $I = \{p(1), p(2), p(3)\}$, we have $R \subseteq I \subseteq S_2$ and $I \not\models A$. □

Let $\mathcal{A}$ be the set of aggregates $\{A_1, ..., A_n\}$ or the conjunction $(A_1 \wedge ... \wedge A_n)$ and R and S two sets of atoms. We define $R \models_S \mathcal{A}$ iff $R \models_S A_i$, $1 \leq i \leq n$.

Given two sets R and S, and an aggregate program P, the operator $K_P(R, S)$ is defined as:

$$K_P(R, S) = \{hd(r) \mid \exists r \in P,\ R \models_S bd(r)\}.$$

The sequence of $K_P^i(R, S)$ is defined as

$$K_P^0(\emptyset, M) = \emptyset \text{ and } K_P^i(\emptyset, M) = K_P(K_P^{i-1}(\emptyset, M), M), \text{ for all } i \geq 0.$$

It is easy to see that K_P is monotone w.r.t. its first argument, given that the second argument is fixed. We have the following corollary.

[1] In general, A_i could also be atoms or negative atoms. We focus on positive aggregates in this chapter. The results can be extended to the general case, where the (negative) atoms are treated in exactly the same way as in normal logic programs [76].

Proposition 5.1 ([76])**.** Let P be an aggregate program and M a set of atoms. Then

$$K_P^0(\emptyset, M) \subseteq K_P^1(\emptyset, M) \subseteq ... \subseteq M \tag{5.3}$$

Definition 5.2 ([76])**.** Let P be an aggregate program and M a set of atoms. M is an *answer set* of P iff M is the least fixpoint of $K_P(\emptyset, M)$, i.e., $M = K_P^\infty(\emptyset, M)$

Example 5.3. Let P be the program

$$p(1) \tag{5.4}$$

$$p(2) \tag{5.5}$$

$$p(3) \tag{5.6}$$

$$p(5) \leftarrow q \tag{5.7}$$

$$q \leftarrow SUM(\{X \mid p(X)\} > 10 \tag{5.8}$$

Any answer set of P must contain $p(1)$, $p(2)$, and $p(3)$. Actually, $M = \{p(1), p(2), p(3)\}$ is an answer set of P, since

$$K_P^0(\emptyset, M) = \emptyset \tag{5.9}$$

$$K_P^1(\emptyset, M) = \{p(1), p(2), p(3)\} \tag{5.10}$$

$$K_P^2(\emptyset, M) = \{p(1), p(2), p(3)\}. \tag{5.11}$$

□

5.2 Other Semantics

5.2.1 FLP-answer set semantics

The notion of answer set proposed by Faber et al. [25] is based on a new notion of reduct. Given a program P and a set of atoms M, the *reduct* of P with respect to M, denoted by $\Gamma(M, P)$, is obtained by removing the rules whose body is false w.r.t. M, i.e.

$$\Gamma(M, P) = \{r \mid r \in P, M \models bd(r)\}. \tag{5.12}$$

Then, an answer set of program P is defined as the minimal model of $\Gamma(M, P)$. Following [76], we call such an answer set FLP-answer set to distinguish it from the answer set based on conditional satisfaction given in Definition 5.2. It has been shown that any answer set is an FLP-answer set. But an FLP-answer set may not be an answer set.

Example 5.4. Let P be the program that consists of the rules:

$$p(1) \leftarrow SUM(\{X \mid p(X)\}) \geq 0 \tag{5.13}$$

$$p(-1) \leftarrow p(1) \tag{5.14}$$

$$p(1) \leftarrow p(-1) \tag{5.15}$$

It can be checked that $M = \{p(1), p(-1)\}$ is an FLP-answer set, but we have that $K_P^\infty(\emptyset, M) = \emptyset$. So M is not an answer set of P. □

The FLP-answer set semantics has been implemented in the answer set programming system DLV .

5.2.2 Ferraris' semantics

Ferraris [27] generalizes the answer set semantics to equilibrium logic and uses a generalized concept of answer set to define the semantics of aggregate programs. In the definition, the concept of reduct in the definition of answer sets of logic programs is extended to propositional theories. The answer sets of a propositional theory are defined as the minimal models of the reduct of the theory. An aggregate program is first translated to a set of propositional theories and then the answer sets of the aggregate program are defined as the answer sets of the translated propositional theories.

It has been shown in [27] that the semantic proposed by Ferraris is an extension of FLP-answer set semantics. For the aggregate programs that consist of rules of the form (5.2), Ferraris' semantics is identical to FLP-answer set semantics.

5.2.3 PDB-answer set semantics

Pelov et al. [65] use an approximation theory to define answer sets for aggregate programs. In particular, they describe a fixpoint operator, called Φ_P^{aggr}, operating on 3-valued interpretations and parameterized by the choice of approximating aggregates. The PDB-answer set is defined as the first component of the least fixpoint (*lfp*) of Φ_P^{aggr}, where the ultimate approximating aggregates are employed, i.e., for a programs P and a set of atoms M, M is a PDB-answer set of P if and only if $M = lfp(\Phi_P^{aggr,1}(\emptyset, M))$, that is M is the first component of $\Phi_P^{aggr}(\emptyset, M)$.

As pointed out in [76], PDB-answer sets coincide with the answer sets based on conditional satisfaction given in Definition 5.2.

Chapter 6

Level mapping induced loop formulas for aggregate programs

6.1 Motivation

Aggregate programs are closely related to weight constraint programs, since many aggregates can be encoded by weight constraints (this will be shown later). The study on the loop formulas for weight constraint programs and the effectiveness of the loop formula approach proposed in [49] motivates us to study the loop formulas for aggregate programs.

Loop formulas for answer set semantics are presented in [84]. In the approach, given a program, the construction of the dependency graph requires computing what is called "local power set" for the constraints in the program in order to capture conditional satisfaction. The process takes exponential time in the size of the program. The question that we address in this chapter is whether the construction of dependency graph can be done in polynomial time for aggregate programs. Again, we tackle the problem by means of level mapping.

Son et. al. [77] give a level mapping characterization of the answer set semantics. In their approach, to compute the level of an aggregate, a checking process is needed to capture the conditional satisfaction of aggregates. The time complexity of this process is exponential in the size of the domain of the aggregate.

We investigate level mapping for aggregate programs and find that, for aggregates, the conditional satisfaction checking can be reduced to the polynomial time standard satisfaction checking. Based on this finding, we define the levels of aggregates. The definition induces a formulation of loop formulas, where local power sets are not needed and the exponential process to compute them is avoided.

In the next section, we show the encoding of the aggregate SUM by weight constraints. In Sections 6.3 and 6.4, respectively, we present the level mapping characterization of an-

swer sets and the loop formulas for aggregate programs. Section 6.5 manifests the difference between the stable model and answer set semantics that is revealed by the level mapping characterizations. Section 6.6 is a conclusion.

6.2 Encoding of aggregates

Let $A = SUM(\{X|p(X)\}) \leq k$ be an aggregate, where $D(X) = \{a_1, ...a_m, b_1, ..., b_n\}$, and $a_i's$ and $b_i's$ are positive and negative numbers, respectively. A can be encoded by the following weight constraint.

$$W = [p(a_1) = a_1, ..., p(a_m) = a_m, p(b_1) = b_1, ..., p(b_n) = b_n]k, \tag{6.1}$$

Using the transformation in [74], W can be translated to

$$W(A) = [p(a_1) = a_1, ..., p(a_m) = a_m, \texttt{not}\ p(b_1) = |b_1|, ..., \texttt{not}\ p(b_n) = |b_n|]k' \tag{6.2}$$

where $k' = k + \sum_{i=1}^{n} b_i$.

It is easy to show that for any set of atoms M, $M \models A$ iff $M \models W(A)$. We call $W(A)$ the *weight constraint encoding* of A.

The notations in this chapter are the same as that used for the level mapping characterization of stable models for weight constraint programs, presented at the beginning of Section 3.2.

A good property of aggregates is that their weight constraint encoding contains no dual atoms. The property will be used in the proof later.

In Chapter 7, we will show that most standard aggregates can be encoded by weight constraints. In this chapter, we focus on aggregates SUM only. The reasons are:

- It is a representative aggregate in that the aggregates $COUNT$ and AVG are special cases of SUM;
- It is the most commonly used aggregate in the current practice of answer set programming, e.g., this is the case for the benchmarks in the first answer set programming system competition [14];
- There is no technical difficulty in extending the results to aggregate programs with aggregates MAX and MIN, since they can be encoded by aggregate SUM.

Note that although the aggregate SUM is essentially the same as weight constraints, the semantics that we consider in this chapter are different from that in Chapter 3, i.e., we

consider answer set semantics for aggregate programs in this chapter but the stable model semantics for weight constraint programs in Chapter 3.

6.3 Level mapping characterization of answer sets

Recall that given a set of atoms X, a level mapping of X is a function λ from X to positive integers.

Definition 6.1. Let A be an aggregate, M a set of atoms and λ a level mapping of M. The *answer set level* of A w.r.t. M, denoted $L^*(A, M)$, is defined as:

$$L^*(A, M) = min(\{H(X) \mid X \subseteq M, w(W(A), X_a) \geq l + \sum_{b_i \in M} w_{b_i}, \quad and\ w(W(A), X_b) \leq u - \sum_{a_i \in M} w_{a_i}\}). \tag{6.3}$$

Example 6.1. Let $A = SUM(\{X \mid p(X)\}) \leq 0$ be an aggregate, where $D(X) = \{-1, 1\}$. Then $W(A) = [p(1) = 1, \texttt{not}\ p(-1) = 1]1$. Let $M = \{p(-1), p(1)\}$ and λ be a level mapping of M, where $\lambda(p(-1)) = 1$ and $\lambda(p(1)) = 2$. It can be seen that the subsets of M that satisfy the inequalities in formula (6.3) are $X_1 = \{p(-1)\}$ and $X_2 = M$. So, $L^*(A, M) = min(\{H(\{p(-1)\}), H(\{p(-1), p(1)\})\}) = \lambda(p(-1)) = 1$. □

Definition 6.2. Let P be an aggregate program and M a set of atoms. M is said to be *strongly level mapping justified* by P if there is a level mapping λ of M satisfying that for each $b \in M$, there is a rule $r \in P$, such that $b = hd(r)$, $M \models bd(r)$, and for each $A \in bd(r)$, $\lambda(b) > L^*(A, M)$.

In this case, we say that the level mapping λ *strongly justifies* M *by* P.

The main theorem of this section states that a strongly level mapping justified model is exactly also an answer set. It is based on the following two lemmas by which the conditional satisfaction of an aggregate is reduced to the standard satisfaction of its weight constraint encoding.

Lemma 6.1. Let A be an aggregate and X a set of atoms. $X \models A$ iff $w(W(A), X_a) \geq l + \sum_{b_i \in X} w_{b_i}$ and $w(W(A), X_b) \leq u - \sum_{a_i \in X} w_{a_i}$, where l and u are the lower and upper bounds of $W(A)$, respectively.

Proof. Since $w(W(A), X) = w(W(A), X_a) - \sum_{b_i \in X} w_{b_i}$ and $w(W(A), X_a) \geq l + \sum_{b_i \in X} w_{b_i}$, we have $w(W(A), X) \geq l$. Since $w(W(A), X) = w(W(A), X_b) + \sum_{a_i \in X} w_{a_i}$ and $w(W(A), X_b) \leq u - \sum_{a_i \in X} w_{a_i}$, we have $w(W(A), X) \leq u$. Therefore, $X \models W(A)$. □

Lemma 6.2. Let A be an aggregate and X and M two sets of atoms such that $X \subseteq M$. $X \models_M A$ iff $w(W(A), X_a) \geq l + \sum_{b_i \in M} w_{b_i}$ and $w(W(A), X_b) \leq u - \sum_{a_i \in M} w_{a_i}$, where l and u are the lower and upper bounds of $W(A)$, respectively.

Proof. ($\Leftarrow$). By Lemma 6.1 we have $X \models A$. We now show that for any S such that $X \subseteq S \subseteq M$, we have $S \models A$.

(1) Let S be any set such that $X \subseteq S \subseteq M$. We have $X_a \subseteq S_a$ and $S_b \subseteq M_b$.

(2) $w(W(A), S) \geq w(W(A), X_a) - \sum_{b_i \in M} w_{b_i}$, due to (1).

(3) $w(W(A), S) \geq l$, due to (2).

(4) $\forall S$ such that $X \subseteq S \subseteq M$, we have $S_a \subseteq M_a$ and $X_b \subseteq S_b$.

(5) $w(W(A), S) \leq w(W(A), X_b) + \sum_{a_i \in M} w_{a_i}$, due to (4).

(6) $w(W(A), S) \leq u$, due to (5).

(7) $S \models W(A)$, due to (3) and (6).

(8) $S \models A$, due to (7).

($\Rightarrow$).

(1) Let $S = M_b \cup X_a$. We have $X \subseteq S \subseteq M$ and $S \models W(A)$.

(2) $w(W(A), S) = w(W(A), X_a) - \sum_{b_i \in M} w_{b_i}$, due to (1).

(3) $w(W(A), S) \geq l$ Due to (1).

(4) $w(W(A), X_a) \geq l + \sum_{b_i \in M} w_{b_i}$, due to (3).

(5) Let $S' = M_a \cup X_b$. We have $X \subseteq S' \subseteq M$ and $S' \models W(A)$.

(6) $w(W(A), S') = \sum_{a_i \in M} w_{a_i} + \sum_{b_i \notin X_b} w_{b_i}$, due to (5).

(7) $\sum_{b_i \notin X_b} w_{b_i} = w(W(A), X_b)$.

(8) $w(W(A), S') = \sum_{a_i \in M} w_{a_i} + w(W(A), X_b)$, due to (6) and (7).

(9) $w(W(A), S') \leq u$, due to (5).

(10) $w(W(A), X_b) \leq u - \sum_{a_i \in M} w_{a_i}$, due to (8) and (9).

□

Now, we are ready to present the main theorem.

Theorem 6.1. Let P be an aggregate program and M a set of atoms. M is an answer set of P iff M is a model of P and strongly level mapping justified by P.

Proof. ($\Rightarrow$)

(1) M is an answer set of P.

(2) M is a model of P, due to (1).

(3) $M = K_P^\infty(\emptyset, M)$, due to (1).

(4) There is a level mapping $\lambda : M \rightarrow \mathbb{Z}^+$: $\lambda(b) = k$ if $b \in K_P^k(\emptyset, M)$ and $b \notin K_P^{k-1}(\emptyset, M)$, due to (3) and Lemma 6.2.

(5) λ strongly level mapping justifies M by P, due to (3) and (4).

(6) M is strongly level mapping justified, due to (5).

($\Leftarrow$)

(1) $M \models P$ and M is strongly level mapping justified.

(2) Suppose that λ strongly justifies M by P.

(3) $\forall b \in M$, there is a rule $r \in P$ and a set $R \subseteq M$ s.t. $b = hd(r)$ and $b \notin R$, and $R \models_M bd(r)$, due to Lemma 6.2 and (2).

(4) $b \in K_P^\infty(\emptyset, M)$, due to (3).

(5) $M \subseteq K_P^\infty(\emptyset, M)$, due to (4).

(6) $K_P^\infty(\emptyset, M) \subseteq M$, due to Proposition 5.1 in Chapter 5.

(7) $M = K_P^\infty(\emptyset, M)$, due to (5) and (6).

(8) M is an answer set of P.

□

Example 6.2. Let P_2 be the program

$$p(-1) \leftarrow \tag{6.4}$$

$$p(1) \leftarrow SUM(\{X \mid p(X)\}) \leq 0. \tag{6.5}$$

Let $M = \{p(-1), p(1)\}$ and λ be a level mapping of M, where $\lambda(p(-1)) = 1$ and $\lambda(p(1)) = 2$. We have $L^*(A, M) = 1$ (c.f. Example 6.1). Therefore $\lambda(p(1)) > L^*(A, M)$ and M is strongly level mapping justified by P_2. It can also be verified that M is an answer set of P_2 by Definition 5.2. □

6.4 Loop formulas

6.4.1 Completion

The completion of aggregate programs consists of the same set of formulas as that of weight constraint programs (Chapter 3), except that the weight constraints in the formulas are weight constraint encodings of aggregates.

Let P be an aggregate. The completion of P, denoted $Comp(P)$, consists of the following formulas.

(1). $\wedge_{A_i \in bd(r)} w(A_i) \rightarrow hd(r)$, for every rule $r \in P$.

(2). $x \rightarrow \vee\{\wedge_{A_i \in bd(r)} W(A_i) \mid r \in P, x = hd(r)\}$, for every atom $x \in Atom(P)$.

Theorem 6.2. Let P be a weight constraint program and M a set of atoms. M is a supported model of P iff M is a model of $Comp(P)$.

Proof. Recall that for an aggregate A and a set of atoms M, $M \models A$ iff $M \models W(A)$, where $W(A)$ is the weight constraint encoding of A.

Let P be a program and M a set of atoms, M satisfies the formulas of the form (1) in $Comp(P)$ if and only if $M \models P$ and further, according to the definition of supported model, for any atom $a \in M$, there exists a rule $r \in P$ such that $a = hd(r)$ and $M \models bd(r)$ if and only if M satisfies the formulas of the form (2) in $Comp(P)$. □

6.4.2 Loop formulas

Let P be an aggregate program. The *dependency graph* of P, denoted $G^*_P = (V, E)$, is a directed graph, where

- $V = At(P)$,
- (u, v) is a directed edge from u to v in E, if there is a rule of the form (5.2) in P, such that $u = hd(r)$, and either v or `not` $v \in lit(W(A_i))$, for some i ($1 \leq i \leq n$).

Now we give the *strong restriction* of an aggregate w.r.t. a loop by defining the strong restriction of a weight constraint. Let W be a weight constraint and L a set of atoms.

The strong restriction of W, w.r.t. L, denoted $W^*_{|L}$, is a conjunction of weight constraints $W^*_{l|L} \wedge W^*_{u|L}$, where

- $W^*_{l|L}$ is obtained by removing from W the upper bound, all positive literals that are in L and their weights;
- $W^*_{u|L}$ is obtained by removing from W the lower bound, $\texttt{not}\ b_i = w_{b_i}$ for each $b_i \in L$, and changing the upper bound to be $u - \sum_{b_i \in L} w_{b_i}$.

The strong restriction of an aggregate A w.r.t. a loop L is defined as the strong restriction of its weight constraint encoding w.r.t. the loop $W^*_{|L}(A)$.

Definition 6.3. Let P be an aggregate program and L a loop in G^*_P. The loop formula for L, denoted $LF^*(P, L)$, is defined as

$$LF^*(P, L) = \bigvee L \rightarrow \bigvee \{ \bigwedge_{A \in bd(r)} W^*_{|L}(A) \mid r \in P, hd(r) \in L \} \tag{6.6}$$

Let P be an aggregate program. The loop completion of P, denoted $LComp^*(P)$, is defined as

$$LComp^*(P) = Comp(P) \cup \{LF^*(P, L) \mid L\ is\ a\ loop\ in\ G^*_P\} \tag{6.7}$$

We can prove the following theorem.

Theorem 6.3. Let P be an aggregate program and M a set of atoms. M is an answer set of P iff it is a model of $LComp^*(P)$.

Proof. For simplification, we use respectively W, $W^*_{l|L}$, and $W^*_{u|L}$ to represent $W(A)$, $W^*_{l|L}(A)$, and $W^*_{u|L}(A)$, since the aggregate A in consideration is always clear in context. We consider a rule of form $h \leftarrow A$. The proof is applicable to rules with conjunctive body.

The proof about $W^*_{l|L}$ is similar to that for Theorem 3.2. We show the part for $W^*_{u|L}$ in the following.

($\Leftarrow$)

(1) $M \models LF^*(P, L)$.

(2) $\forall b \in M \cap L$, $\exists r \in P$, such that $hd(r) \in L$, and $M \models W^*_{u|L}$, due to (1).

(3) $w(W^*_{u|L}, M) \leq u - \sum_{b_i \in L_b} w_{b_i}$, due to (2).

(4) $w(W^*_{u|L}, M) = w(W, M) - \sum_{b_i \in L_b} w_{b_i} + \sum_{b_i \in L_b \cap M_b} w_{b_i}$.

(5) $w(W, M) \leq u - \sum_{b_i \in L_b \cap M_b} w_{b_i}$, due to (3) and (4).

(6) Let $X_b = M_b \setminus (M_b \cap L_b)$.

(7) $w(W, X_b) = w(W, M) - \sum_{a_i \in M} w_{a_i} + \sum_{b_i \in L_b \cap M_b} w_{b_i}$, due to (6).

(8) $w(W, X_b) \leq u - \sum_{a_i \in M} w_{a_i}$, due to (7).

(9) The level mapping λ where $\lambda(b) \geq max(\{\lambda(b_i) \mid b_i \in X_b\})$ strongly justifies M by P.

(10) M is strongly level mapping justified.

(11) $M \models Comp(P)$.

(12) $M \models P$, due to (11).

(13) M is an answer set of P, due to and (10) and (12).

($\Leftarrow$)

(1) M is an answer set of P.

(2) There is a level mapping λ from $At(P)$ to positive integers satisfying $\forall b \in M, \exists r \in P$, such that $\lambda(b) > L^*(A, M)$.

(3) Let L be a loop and $b \in L$. $\exists X \subseteq M \setminus L$ and a rule $r \in P$ such that $w(W, X_b) \leq u - \sum_{a_i \in M} w_{a_i}$, due to (2).

(4) $w(W, M_b \setminus (M_b \cap L_b)) \leq w(W, X_b)$ Due to (3).

(5) $w(W, M_b \setminus (M_b \cap L_b)) = w(W, M) - \sum_{a_i \in M} w_{a_i} + \sum_{b_i \in L_b \cap M_b} w_{b_i}$.

(6) $w(W, M) + \sum_{b_i \in L \cap M} w_{b_i} \leq u$, due to (3), (4) and (5).

(7) $w(W^*_{u|L}, M) = w(W, M) - \sum_{b_i \in L} w_{b_i} + \sum_{b_i \in L \cap M} w_{b_i}$.

(8) $w(W^*_{u|L}, M) \leq u - \sum_{b_i \in L} w_{b_i}$, due to (6) and (7).

(9) $M \models W^*_{u|L}$.

(10) $M \models LF^*(P, L)$.

(11) $M \models Comp(P)$, due to (1).

(12) $M \models LComp^*(P)$, due to (10) and (11).

□

Example 6.3. Consider the program P_2 in Example 6.2. There is a loop $\{p(1)\}$ in $G^*_{P_2}$. The loop formula is $p(1) \rightarrow [p(1) = 1, \texttt{not}\ p(-1) = 1]1$ and satisfied by the set $M = \{p(-1), p(1)\}$. So, M is an answer set of P_2. □

6.5 Comparison

Weight constraint programs and aggregate programs are both logic programs with constraints, but they have different semantics. Our level mapping characterizations and loop formulas reveal the difference between the stable model semantics for weight constraint programs and answer set semantics for aggregate programs.

Comparing the definition of the level of weight constraints in Chapter 3 formula (3.2) to that of the answer set level of aggregates in formula (6.3), we may see that the atoms in X_b that are negative in W and the upper bound of W are not considered in formula (3.2), while they are constrained by the second inequality in formula (6.3).

Naturally, the difference is also manifested by the constructions of dependency graphs and formulations of loop formulas (the definitions of restriction and strong restriction) for weight constraint and aggregate programs, respectively. For stable model semantics, negative dependencies between atoms do not contribute to the loops and loop formulas, while they do for the answer set semantics. The following example demonstrates how level mapping and loop formulas capture this difference.

Example 6.4. Let P_3 be the aggregate program $\{p(-1) \leftarrow SUM(\{X|p(X)\}) \leq -1\}$, where $D(X) = \{-1\}$. We denote the aggregate in P_3 by A. We have $W(A) = [\texttt{not}\ p(-1) = 1]0$. The weight constraint program counterpart of P_3 is P'_3: $\{p(-1) \leftarrow [\texttt{not}\ p(-1) = 1]0\}$.

Consider the set $M = \{p(-1)\}$. For program P_3 and any level mapping λ, we have $L^*(A, M) = \lambda(p(-1))$, since the only subset of M that satisfies the inequalities in formula (6.3) is M itself. Namely, $p(-1)$ depends on itself, even though it appears negatively in the body of the rule. There is a loop $\{p(-1)\}$ in $G^*_{P_3}$. The loop formula is: $p(-1) \rightarrow [\,] - 1$[1]. The loop formula is not satisfied by M. Thus M is not an answer set of P_3.

For program P'_3 and any level mapping α, we have $L(W(A), M) = 0$, since $\emptyset$ satisfies the inequality in formula (3.2). There is no loop in $G_{P'_3}$. It can be verified that M is a stable

[1] $[\,] - 1$ is a weight constraint where the literal set is empty and upper bound is -1. It can not be satisfied by any set.

model of P_3'. □

6.6 Conclusion

In this chapter, we give a level mapping characterization of answer sets. Based on the characterization, we develop an approach to build the loop formulas for aggregate programs. In the approach, for a program P, the dependency graph G_P^* can be constructed by going through each rule and building an edge in G_P^* from the head of the rule to each literal in the domain of aggregates in the body of the rule. The process takes time linear in the size of P (number of rules plus the number of atoms in P). The exponential time process required in the approach [84] is therefore avoided. The establishment of the formula for a loop is also linear in the size of P, since the restriction of an aggregate can be obtained in linear time according to the definition.

It can be shown that any loop defined in [84] is a loop defined here, but the reverse does not hold. This is because we count negative dependencies as edges. As a result, there are more loops under this definition. However, for many of these loops, their loop formulas can be satisfied by any supported model (need not to be constructed in practice). For example, given the following program:

$$a \leftarrow 1[\texttt{not } b = 1] \qquad b \leftarrow 1[\texttt{not } a = 1]$$

$L = \{a, b\}$ is a loop by our definition. The loop formula of L is satisfied by any supported model of the program.

Future works are needed for the aggregate programs. Firstly, the level mapping and loop formulas are defined on aggregate SUM. For the aggregates MAX and MIN, direct definition may be desired for intuitiveness. Secondly, as we have mentioned, there may be many redundant loops. Thus, our result may be more appropriately regarded as evidence of the existence of a polynomial time construction of dependency graph in order to formulate loop formulas, rather than a practical proposal, ready for implementation. The impact of the redundant loops on efficiency and methods to remove them are worth further study.

Chapter 7

Computing aggregate programs as weight constraint programs

7.1 Motivation

We propose a computation approach to aggregate programs by investigating the relationship between stable model and the answer set semantics[1].

Weight constraint programs and aggregates are both logic programs with constraints. They are closely related but have different semantics. A question arises as how the stable model semantics of weight constraint programs is related to the answer semantics of aggregate programs. If differences exist, what is the nature of the differences and their potential implications in applications. These questions are important since weight constraint programs and aggregate programs have been used for benchmarks and serious applications (e.g. [32, 82]), and will likely be adopted in further endeavors in applying the ASP technology to real world applications.

Our study in this chapter show that for a subclass of weight constraint programs, called *strongly satisfiable programs*, the stable model semantics (w.r.t. Definition 2.2) agrees with the answer set semantics (w.r.t. Definition 5.2). For example, weight constraint programs where weight constraints are upper bound free are all strongly satisfiable. This result is useful in that we are now sure that the known properties of the latter semantics also hold for these programs. One important property is that any answer set is a *well-supported model* [77], ensuring that any conclusion must be supported by a non-circular justification in the sense of Fages [26].

We further reveals that for weight constraint programs where the two semantics disagree, stable models may be circularly justified, based on a formal notation of circular jus-

[1] In the literature, stable model and answer set are interchangeable. We refer to them for different semantics.

tification. We then show that there exists a transformation from weight constraint programs to strongly satisfiable programs, which provides a way to run weight constraint programs under the current implementation of systems that implemented stable model semantics (e.g. SMODELS and CLASP) without generating circular models.

We also find that most aggregates proposed for ASP can be encoded by weight constraints and the size of the encoding is linear in the size of the domain of the aggregates. Accordingly, an aggregate program can be effectively translated to a weight constraint program. This leads to an approach to computing aggregate programs as weight constraint programs using any system that implements stable model semantics. We implement the approach in the system called ALPARSE (Aggregates to Lparse programs, which are weight constraint programs.). The system ALPARSE consists of a translator and an ASP solver for weight constraint programs. The translator transforms a given aggregate program to a strongly satisfiable program. The ASP solver then computes the answer sets of the resulting program.

To evaluate this approach, we have conducted a series of experiments. In the first part of the experiments, we use the well-known system SMODELS as the solver of ALPARSE and compare it with the solvers SMODELS^A and DLV on aggregate programs, respectively. Our experiments show that ALPARSE often runs faster, sometimes substantially faster, than the two aggregate systems for the benchmarks tested.

The ASP system CLASP is an efficient solver for weight constraint programs. In the second part of the experiments, we use CLASP as the ASP solver for ALPARSE and compare with DLV on the benchmarks from the first ASP system competition. The results show that ALPARSE constantly outperforms DLV . This suggests that representing aggregates by weight constraints is a promising alternative to the explicit handling of aggregates in logic programs.

Besides efficiency, another advantage is at the system level: an aggregate language can be built on top of an ASP solver supporting weight constraints with a simple front end that essentially transforms standard aggregates to weight constraints in linear time. This is in contrast with the state-of-the-art in handling aggregates in ASP, which typically requires an explicit implementation of each aggregate.

In Section 7.2, we relate stable model semantics to answer set semantics. We show that they coincide for strongly satisfiable programs, and that an arbitrary weight constraint program can be transformed to a strongly satisfiable program. Section 7.3 provides the transformation from standard aggregates to weight constraints and from aggregate programs

to weight constraint programs. The effectiveness of the transformation is demonstrated by experiments in Section 7.4 and final remarks are given in Section 7.5.

7.2 A study on semantics

Notation: Given a weight constraint W of the form (2.2) and a set of atoms M, we define $M_a(W) = \{a_i \in M \mid a_i \in lit(W)\}$ and $M_b(W) = \{b_i \in M \mid \texttt{not}\ b_i \in lit(W)\}$. Since W is always clear by context, we will simply write M_a and M_b.

7.2.1 Coincidence between semantics

Definition 7.1. Let M be a set of atoms and W a weight constraint of the form (2.2). W is said to be *strongly satisfiable by* M if $M \models W$ implies that for any $V \subseteq M_b$, $w(W, M \setminus V) \leq u$.

Intuitively, strong satisfaction by M requires that even if some b_i's are removed from M (thus the value of the summation for W increases), the upper bound of W is still satisfied.

W is *strongly satisfiable* if for any set of atoms M, W is strongly satisfiable by M. A weight constraint program is *strongly satisfiable* if every weight constraint that appears in the body of a rule in it is strongly satisfiable.

Example 7.1. The following constraints are all strongly satisfiable: $1\ [a = 1, b = 2]\ 2$, $1\ [a = 1, \texttt{not}\ b = 2]\ 3$, and $1\ [a = 1, \texttt{not}\ b = 2]$. But $1\ [a = 1, \texttt{not}\ b = 2]\ 2$ is not, since it is satisfied by $\{a, b\}$ but not by $\{a\}$. □

Strongly satisfiable programs constitute a nontrivial class of programs. In particular, weight constraints W that possess one of the following syntactically checkable conditions are strongly satisfiable.

- $lit(W)$ contains only atoms;
- $\sum_1^n w_{a_i} + \sum_1^m w_{b_i} \leq u$.

Strongly satisfiable constraints are not necessarily convex or monotone.

Example 7.2. Let $A = 2[a = 1, b = 1, \texttt{not}\ c = 1]$ be a weight constraint. Since A is upper bound free, it is strongly satisfiable. But A is neither monotone nor convex, since $\{a\} \models A$, $\{a, c\} \not\models A$, and $\{a, b, c\} \models A$. □

To present our results, it is notationally important to lift the concept of conditional satisfaction to weight constraints. Let W be a weight constraint and R and S be two sets of atoms. R conditionally satisfies W, w.r.t. S, denoted $R \models_S W$ if $\forall I$ such that $R \subseteq I \subset S$, we have $I \models W$. Answer sets of a weight constraint program are defined as least fixpoints of the operator K_P in Chapter 5.

Theorem 7.1. Let P be a weight constraint program and $M \subseteq At(P)$. Suppose for any weight constraint W appearing in the body of a rule in P, W is strongly satisfiable by M. Then, M is a stable model of P iff M is an answer set for P.

Proof. Let M and S be two sets of atoms such that $S \subseteq M$, and P be a program in which the weight constraints that appear in the bodies of rules in P are strongly satisfiable by M. We will prove a key lemma below which relates $S \models_M W$ with $S \models W^M$. The goal is to ensure a one-to-one correspondence between the derivations based on conditional satisfaction (Definition 5.2) and the derivations in the construction of the least model (Definition 2.1). Then it can be shown, by induction on the length of derivations, that a stable model of P is an answer set for an instance of P, and vice versa. □

Lemma 7.1. Let W be a weight constraint of the form (2.2), and S and M be sets of atoms such that $S \subseteq M$. Then,

(i) If $S \models_M W$ then $S \models W^M$ and $w(W, M) \leq u$.

(ii) If $S \models W^M$ and W is strongly satisfiable by M, then $S \models_M W$.

Proof. We prove (i) and (ii), respectively as follows.

(i) We prove it by contraposition. That is, we show that if $w(W, M) > u$ or $S \not\models W^M$, then $S \not\models_M W$. The case of $w(W, M) > u$ is simple. It leads to $M \not\models W$ hence $S \not\models_M W$.

Assume $S \not\models W^M$. By definition, the lower bound is violated, i.e., $w(W^M, S) < l'$, where $l' = l - \sum_{b_i \notin M} w_{b_i}$. Let $I = I_a \cup I_b$, where $I_a = S_a$ and $I_b = M_b$. Since $w(W^M, S) = w(W^I, S)$ and $w(W^M, S) < l'$, we have $w(W^I, S) < l'$. Then, from $I_b = M_b$ and the assumption $S \not\models W^M$, we get $S \not\models W^I$. It then follows from $I_a = S_a$ that $I \not\models W$. By construction, we have $S \cap dom(W) \subseteq I \subseteq M$, and therefore we conclude $S \not\models_M W$.

(ii) Assume $S \not\models_M W$ and W is strongly satisfiable by M. We show $S \not\models W^M$.

We have either $S \models W$ or $S \not\models W$. If $S \not\models W$ then clearly $S \not\models W^M$. Assume $S \models W$. Then from $S \not\models_M W$, we have $\exists I$, $S \cap dom(W) \subset I \subseteq M$, such that $I \not\models W$. Since W is strongly satisfiable by M, if $M \models W$ then for any $R = M \setminus V$, where $V \subseteq M_b$, $w(W, R) \leq u$. Assume $M \models W$. Let R be such that $R_b = I_b$ and $I_a \subseteq R_a$. It's clear that $w(W, R) \leq u$ leads to $w(W, I) \leq u$. Thus, since $M \models W$, $I \not\models W$ follows due to the violation of the lower bound, i.e., $w(W, I) < l$.

Now consider $I' = S_a \cup M_b$; i.e., we restrict I_a to S_a and expand I_b to M_b. Note that by construction, it still holds that $S \cap dom(W) \subset I' \subseteq M$. Clearly, $I \not\models W$ leads to $I' \not\models W$, which is also due to the violation of the lower bound, as $w(W, I') \leq w(W, I)$, i.e., we have $w(W, I') < l$. By definition, we have $w(W^{I'}, I') < l'$, where $l' = l - \sum_{b_i \notin I'} w_{b_i}$. Note that since $I'_b = M_b$, we have $l' = l - \sum_{b_i \notin M} w_{b_i}$. Since $I'_a = S_a$, it follows that $w(W^{I'}, S) < l'$. Now since $W^{I'}$ is precisely the same constraint as W^M, we have $w(W^{I'}, S) = w(W^M, S)$, and therefore $w(W^M, S) < l'$. This shows $S \not\models W^M$.

□

By Theorem 7.1 and the definition of strongly satisfiable programs, we can show that for strongly satisfiable programs, the stable model semantics coincides with the answer set semantics.

Theorem 7.2. Let P be a strongly satisfiable weight constraint program, and $M \subseteq At(P)$ be a set of atoms. M is a stable model of P iff M is an answer set for P.

7.2.2 When the semantics disagree

The following theorem can be proved using Lemma 7.1 and Example 7.3 below.

Theorem 7.3. Every answer set of a weight constraint program P is a stable model of P, but the converse does not hold.

In this section, we show what happens to the weight constraint programs that are not strongly satisfiable.

Example 7.3. Let P be a program consisting of a single rule:

$$a \leftarrow [\texttt{not}\ a = 1]\ 0. \tag{7.1}$$

Let $M_1 = \emptyset$ and $M_2 = \{a\}$ be two sets. The weight constraint $[\texttt{not}\ a = 1]0$ in P is not strongly satisfiable, since although M_2 satisfies the upper bound, its subset M_1 does not.

By Definition 2.2, P has two stable models: M_1 and M_2. But, by Definition 5.2, M_1 is an answer set for P and M_2 is not. Note that M_2 is not a minimal model.

The reason that M_2 is not an answer set for P is due to the fact that a is derived by its being in M_2. This kind of circular justification can be seen more clearly below.

- The weight constraint is substituted with an equivalent aggregate:
$$a \leftarrow COUNT(\{X \mid X \in D\}) = 1, \text{ where } D = \{a\}.$$
- The weight constraint is transformed to an equivalent one without negative literal, but with a negative weight, according to [74]:[2] $a \leftarrow [a = -1]{-}1$.

For the claim of equivalence, note that for any set of atoms M, we have: $M \models [\texttt{not}\ a = 1]0$ iff $[a = -1]{-}1$ iff $M \models COUNT(\{X \mid X \in D\}) = 1$ □

The type of circular justification observed here is similar to "answer sets by reduct" in dealing with nonmonotone c-atoms [77]. But the constraint $[\texttt{not}\ a = 1]\ 0$ is actually monotone! One may think that the culprit for M_2 above is because it is not a minimal model. However, the following example shows that stable models that are minimal models may still be circularly justified.

Example 7.4. Consider the following weight constraint program P (obtained from the one in Example 7.3 by adding the second rule):

$$a \leftarrow [\texttt{not}\ a = 1]0 \tag{7.2}$$
$$f \leftarrow \texttt{not}\ f, \texttt{not}\ a \tag{7.3}$$

Now, $M = \{a\}$ is a minimal model of P, and also a stable model of P, but clearly a is justified by its being in M. □

We now give a more formal account of *circular justification* for stable models, borrowing the idea of *unfounded sets* previously used for normal programs [80] and logic programs with monotone and antimonotone aggregates [10].

Definition 7.2. Let P be a weight constraint program and M a stable model of P. M is said to be *circularly justified*, or simply *circular*, if there exists a non-empty set $U \subseteq M$ such that $\forall \phi \in U$, $M \setminus U$ does not satisfy the body of any rule in P where ϕ is in the literal set of the head of the rule.

[2] Caution: Due to an internal bug, SMODELS produces $\emptyset$ as the only stable model, which is inconsistent with the stable model semantics defined in [74].

Theorem 7.4. Let P be a weight constraint program and M a stable model of P. If M is an answer set for P, then M is not circular.

Proof. Let P be a weight constraint program and M be an answer set of P. Consider any atom $a \in M$. According to the definition of answer set, there is an non-negative number k such that $a \in K_P^k(\emptyset, M)$ and $a \notin K_P^{k-1}(\emptyset, M)$. Then there must be a rule $r \in P$ such that $K_P^{k-1}(\emptyset, M) \models bd(r)$ and $a \in hset(r)$. Therefore M is not circular. □

Example 7.3 shows that extra stable models (stable models that are not answer sets) of a program may be circular. However, not all extra stable models are necessarily circular.

Example 7.5. Consider a weight constraint program P that consists of three rules.

$$a \leftarrow \tag{7.4}$$

$$b \leftarrow 2[a = 1, \texttt{not}\ b = 1] \tag{7.5}$$

$$b \leftarrow [a = 1, \texttt{not}\ b = 1]1 \tag{7.6}$$

$M = \{a, b\}$ is a stable model but not an answer set for P. However, it can be verified that M is not circular under our definition: b is derived by the last rule, given a, and M stabilizes by the first two rules.[3] □

7.2.3 Transformation to strongly satisfiable programs

In this section we show that all weight constraint programs can be transformed to strongly satisfiable programs. This is achieved by replacing each weight constraint of form (2.2) in a given program by two upper bound-free weight constraints.

Let W be a weight constraint of form (2.2). The *strongly satisfiable encoding* of W, denoted by (W_1, W_2) consists of the following constraints:

$$W_1 \ : \ l[a_1 = w_{a_1}, ..., a_n = w_{a_n}, \texttt{not}\ b_1 = w_{b_1}, \texttt{not}\ b_1 = w_{b_m}]$$

$$W_2 \ : \ -u + \sum_{i=1}^{n} w_{a_i} + \sum_{i=1}^{m} w_{b_i}[\texttt{not}\ a_1 = w_{a_1}, ..., \texttt{not}\ a_n = w_{a_n}, b_1 = w_{b_1}, ..., b_m = w_{b_m}]$$

Intuitively, W_1 and W_2 are to code the lower and upper bound constraints of W, respectively. The encoding is satisfaction preserving, as shown in the following lemma.

Lemma 7.2. Let W be a weight constraint, (W_1, W_2) be its strongly satisfiable encoding, and M be a set of atoms. $M \models W$ iff $M \models W_1$ and $M \models W_2$.

[3] M is an answer set under the notion of $\rhd_C^s$ computation [48]. However, it appears that the notion of circular justification is still an open issue; there could be different intuitions and definitions.

Proof. The satisfaction of W_1 is trivial, since W_1 is simply the lower bound part of the W. Next, we show that W_2 is the upper bound part of W.

The upper bound part of W is

$$\sum_{1\leq i\leq m} a_i \cdot w_{a_i} + \sum_{1\leq i\leq n} b_i \cdot (-w_{b_i}) \leq u \tag{7.7}$$

which is equivalent to

$$-u \leq \sum_{1\leq i\leq m} a_i \cdot (-w_{a_i}) + \sum_{1\leq i\leq n} b_i \cdot w_{b_i}. \tag{7.8}$$

By the transformation that eliminates the negative weights (introduced in Section 2.2), the constraint (7.8) is equivalent to the weight constraint W_2. □

Using Lemmas 7.1 and 7.2, we establish the following theorem.

Theorem 7.5. Let W be a weight constraint, (W_1, W_2) be the strongly satisfiable encoding of W, and S and M be two sets of atoms, such that $S \subseteq M$. $S \models_M W$ iff $S \models W_1^M$ and $S \models W_2^M$.

Proof. For the $\Rightarrow$ part, W_1^M is the same as W^M. By (i) of Lemma 7.1, we have $S \models W_1^M$. In the following, we show $S \models W_2^M$.
The $\Rightarrow$ part:

(1) $S \models_M W$ and $S \subseteq M$.

(2) $S \models W$ and $\forall I$, s.t. $S \cap lit(W) \subseteq I$ and $I \subseteq M \cap lit(W)$, $I \models W$, due to (1).

(3) Let $I = I_a \cup I_b$ such that $I_b = S_b$, $I_a = M_a$.

(4) $S \cap lit(W) \subseteq I$ and $I \subseteq M \cap lit(W)$, due to (1) and (3).

(5) $w(W, I) \leq u$, that is, $\sum_{a_i \in I_a} w_{a_i} + \sum_{b_i \notin I_b} w_{b_i} \leq u$, due to (2), (4).

(6) $\sum_{a_i \in M_a} w_{a_i} + \sum_{b_i \notin S_b} \leq u$, due to (3) and (5).

(7) $\sum_{i=1}^{n} w_{a_i} - \sum_{a_i \notin M_a} w_{a_i} + \sum_{i=1}^{m} w_{b_i} - \sum_{b_i \in S_b} w_{b_i} \leq u$, due to (6).

(8) $\sum_{b_i \in S_b} w_{b_i} \geq -u + \sum_{i=1}^{n} w_{a_i} + \sum_{i=1}^{m} w_{b_i} - \sum_{a_i \notin M_a} w_{a_i}$, due to (7).

(9) $\sum_{b_i \in S} w_{b_i} \geq -u + \sum_{i=1}^{n} w_{a_i} + \sum_{i=1}^{m} w_{b_i} - \sum_{a_i \notin M} w_{a_i}$, due to (8).

(10) $S \models W_2^M$, due to (9).

The $\Leftarrow$ part:

(1) $S \models W_1^M$, $S \models W_2^M$ and both W_1 and W_2 are strongly satisfiable.

(2) $S \models_M W_1$ and $S \models_M W_2$, due to (1) and (ii) of Lemma 7.1.

(3) $\forall I$ s.t. $S \cap lit(W) \subseteq I$ and $I \subseteq M \cap lit(W)$, $I \models W_1$ and $I \models W_2$, due to (2).

(4) $\forall I$ s.t. $S \cap lit(W) \subseteq I$ and $I \subseteq M \cap lit(W)$, $I \models W$, due to (3) and Theorem 7.2.

(5) $S \models_M W$.

□

Theorem 7.5 guarantees the one-to-one correspondence between the derivations based on conditional satisfaction (Definition 5.2) and the derivations in the construction of the least model (Definition 2.2).

Theorem 7.6. Let P be a weight constraint program, $Tr(P)$ be the program obtained by replacing each W in the body of rules in P by the strongly satisfiable encoding of W, and M be a set of atoms. M is an answer set for P iff M is a stable model of $Tr(P)$.

Example 7.6. Consider a program P with a single rule: $a \leftarrow 0[\texttt{not}\ a = 3]2$. Then, $Tr(P)$ consists of

$$a \leftarrow 0[\texttt{not}\ a = 3],\ 1[a = 3].$$

The weight constraints in $Tr(P)$ are all upper bound-free, hence $Tr(P)$ is strongly satisfiable. Both $\emptyset$ and $\{a\}$ are stable models of P, but $\emptyset$ is the only stable model of $Tr(P)$, which is also the only answer set for P. □

7.3 An approach to computing aggregate programs

7.3.1 Encode aggregates as weight constraints

This section shows that aggregates can be encoded as weight constraints.

Definition 7.3. Let A be an aggregate in the form (5.1). A set of weight constraints $\{W_1, ..., W_n\}$ is an *weight constraint encoding* (or *encoding*) of A, denoted $e(A)$, if for any model M of A, there is a model M' of $e(A)$ such that $M'_{|Dom(A)} = M$ and for any model M' of $e(A)$, $M'_{|Dom(A)}$ is a model of A, where $M'_{|S}$ denotes $M' \cap S$.

We show the encodings of aggregates of the form (5.1), where the operator $\texttt{op}$ is $\geq$. The encodings can be easily extended to other relational operators except for the aggregate SUM with operator $\neq$ (more on $\neq$ later in this section). For example, aggregate $SUM(\{X \mid p(X)\}) > k$ can be encoded as $SUM(\{Y \mid p(Y)\}) \geq k + 1$.

The encodings work for the aggregates whose variable domain contains only integers. For the aggregates whose variable domain contains real numbers, each real number can be converted to an integer by multiplying a factor (e.g. convert 1.5 to 15 by multiplying 10). In this case, the $Result$ also needs to be processed correspondingly.

For convenience, below we may write negative weights in weight constraints. Recall that negative weights can be eliminated by a simple transformation.

SUM, COUNT, and AVG

These aggregates can be encoded by weight constraints rather directly.

For instance, $SUM(\{X \mid p(X)\}) \geq k$ can be represented by

$$k\ [p(a_1) = a_1, ..., p(a_n) = a_n]. \tag{7.9}$$

The aggregates $COUNT(\{X \mid p(X)\}) \geq k$ and $AVG(\{X \mid p(X)\}) \geq k$ can be encoded simply by substituting the weights in (7.9) with 1 and $a_i - k$ (for AVG the lower bound k is also replaced by zero), respectively.

MAX

Let $A = MAX(\{X \mid p(X)\}) \geq k$ be an aggregate. The idea in the encoding of A is that for a set of numbers $S = \{a_1, ..., a_n\}$, the maximum number in S is greater than or equal to k if and only if

$$\sum_{i=1}^{n} (a_i - k + 1) > -\sum_{i=1}^{n} |a_i - k + 1|. \tag{7.10}$$

For each atom $p(a_i)$, two new literals $p^+(a_i)$ and $p^-(a_i)$ are introduced. The encoding $e(A)$ consists of the following constraints.

$$0\ [p(a_i) = -1, p^+(a_i) = 1, p^-(a_i) = 1]\ 0,\ 1 \leq i \leq n \tag{7.11}$$

$$0\ [p(a_i) = -d_i, p^+(a_i) = d_i],\ 1 \leq i \leq n \tag{7.12}$$

$$0\ [p(a_i) = d_i, p^-(a_i) = -d_i],\ 1 \leq i \leq n \tag{7.13}$$

$$1\ [p(a_1) = d_1, p^+(a_1) = d_1, p^-(a_1) = -d_1,$$
$$..., p(a_n) = d_n, p^+(a_n) = d_n, p^-(a_n) = -d_n] \tag{7.14}$$

$$1\ [p(a_1) = 1, ..., p(a_n) = 1] \tag{7.15}$$

where $d_i = a_i - k + 1$.

In the following presentation, for any model M of the encoding, $a = 1$ means $a \in M$ and $a = 0$ means $a \notin M$.

The constraints (7.11), (7.12) and (7.13) are used to encode $|a_i - k + 1|$. Clearly, if $a_i > k-1$, we have $p^+(a_i) = p(a_i)$ and $p^-(a_i) = 0$; if $a_i < k-1$, we have $p^-(a_i) = p(a_i)$ and $p^+(a_i) = 0$; and if $a_i = k - 1$, we have $p^+(a_i) = p(a_i)$ or $p^-(a_i) = p(a_i)$.

The constraint (7.14) encodes the relation (7.10) and the constraint (7.15) guarantees that a model of $e(A)$ is not an empty set.

In the following we prove that the weight constraints from (7.11) to (7.15) is the weight constraint encoding of A. We denote them W_1, W_2, W_3, W_4 and W_5, respectively. We have

Theorem 7.7. Let $A = MAX(\{X \mid p(X)\}) \geq k$ be an aggregate and $e(A) = \{W_1, W_2, W_3, W_4, W_5\}$. Then for any model of A, there is a model M' of $e(A)$ such that $M'_{|Dom(A)} = M$ and for any model M' of $e(A)$, $M'_{|Dom(A)}$ is a model of A, where $M'_{|S}$ denotes $M' \cap S$.

Proof. Let M be a set of atoms and $M \models A$. Suppose $p(a_1) \in M$ and $a_1 \geq k$. Then, we can construct M' as:

- $p(a_i) \in M'$ and $p^+(a_i) \in M'$, if $p(a_i) \in M$ and $a_i \geq k$;
- $p(a_i) \in M'$ and $p^-(a_i) \in M'$, if $p(a_i) \in M$ and $a_i < k$;

It is easy to check that the weight constraints W_1, W_2 and W_3 are satisfied by M'. Since $a_1 \geq k$, we have $p(a_1) \in M'$ and $p^+(a_1) \in M'$. Therefore W_4 and W_5 are also satisfied by M'. So $M' \models e(A)$.

Let M' be a set and $M' \models e(A)$. Since M' models W_1, W_2 and W_3, we have $p^+(a_i) = p(a_i)$ and $p^-(a_i) = 0$, for $a_i \geq k$; $p^-(a_i) = p(a_i)$ and $p^+(a_i) = 0$, for $a_i < k - 1$; and $p^+(a_i) = p(a_i)$ or $p^-(a_i) = p(a_i)$, if $a_i = k - 1$. Since $M' \models W_4$, there must be i, such that $a_i \leq k$ and $p(a_i) = 1$. That is $p(a_i) \in M'_{|Dom(A)}$. Then, we have $M \models A$. □

MIN

Let $A = MIN(\{X \mid p(X)\}) \geq k$ be an aggregate. The idea in the encoding of A is that for a set of numbers $S = \{a_1, ..., a_n\}$, the minimal number in S is greater than or equal to k if and only if

$$\sum_{i=1}^{n} (a_i - k) = \sum_{i=1}^{n} |a_i - k|. \tag{7.16}$$

Similar to MAX, the constraint in (7.16) can be encoded by weight constraints.

$$0\ [p^+(a_i) = 1, p^-(a_i) = 1, p(a_i) = -1]\ 0 \tag{7.17}$$

$$0\ [p^+(a_i) = d_i, p(a_i) = -d_i] \tag{7.18}$$

$$0\ [p^-(a_i) = -d_i, p(a_i) = d_i] \tag{7.19}$$

$$\begin{aligned} 0\ [&p(a_1) = d_1, p^+(a_1) = -d_1, p^-(a_1) = d_1, \\ &..., p(a_n) = d_n, p^+(a_n) = -d_n, p^-(a_n) = d_n] \end{aligned} \tag{7.20}$$

$$\begin{aligned} 0\ [&p(a_1) = -d_1, p^+(a_1) = d_1, p^-(a_1) = -d_1, \\ &..., p(a_n) = -d_n, p^+(a_n) = d_n, p^-(a_n) = -d_n] \end{aligned} \tag{7.21}$$

$$1\ [p(a_1) = 1, ..., p(a_n) = 1] \tag{7.22}$$

where $d_i = a_i - k$.

Similar to Theorem 7.7, we can show that the weight constraints from (7.17) to (7.22) are an encoding of the aggregate $A = MIN(\{X \mid p(X)\}) \geq k$.

We note that all the encodings above result in weight constraints whose collective size is linear in the size of the domain of the aggregate being encoded.

In the encoding of MAX (similarly for MIN), the first three constraints are the ones between the newly introduced literals $p^+(a_i)$, $p^-(a_i)$ and the literal $p(a_i)$. We call them *auxiliary constraints*. The last two constraints code the relation between $p(a_i)$ and $p(a_j)$, where $i \neq j$. We call them *relation constraints*. Let A be an aggregate, we denote the set of auxiliary constraints in $e(A)$ by $a(A)$ and the relation constraints by $r(A)$. If A is aggregate SUM, $COUNT$, or AVG, we have that $r(A) = e(A)$, because no new literals are introduced in the encodings.

For a given aggregate A, the constraints in $e(A)$ can be transformed to strongly satisfiable weight constraints. In the sequel, we assume $e(A)$ contains only strongly satisfiable weight constraints.

7.3.2 Aggregate programs to weight constraint programs

We translate a logic program with aggregates P to a weight constraint program, denoted $\tau(P)$, as follows:

1. For each rule of form (5.2) in P, we have a weight constraint rule of the form

$$h \leftarrow r(A_1), ..., r(A_n) \tag{7.23}$$

in $\tau(P)$. In the formula (7.23), we use $r(A_i)$ to denote the conjunction of all the weight constraints in $r(A_i)$, and

2. If there are newly introduced literals in the encoding of aggregates, the *auxiliary rule* of the form

$$W \leftarrow p(a_i) \tag{7.24}$$

is included in $\tau(P)$, for each auxiliary constraint W of each atom $p(a_i)$ in the aggregates.

By Theorem 7.5, it is easy to show the following theorem.

Theorem 7.8. Let P be an aggregate program where the relational operator is not $\neq$. For any stable model M of $Tr(\tau(P))$, $M_{|At(P)}$ is an answer set for P. For any answer set M for P, there is a stable model M' of $Tr(\tau(P))$ such that $M'_{|At(P)} = M$.

Remark. For an aggregate where the relation operator is not '$\neq$', the aggregate can be encoded by a conjunction of weight constraints as we have presented in this section. In this case, logic equivalence leads to equivalence under conditional satisfaction. That is why we only need to ensure that an encoding is satisfaction-preserving.

For an aggregate where the relation operator is '$\neq$', two classes are distinguished. One class consists of aggregates of the forms $COUNT(.) \neq k$, $MAX(.) \neq k$ and $MIN(.) \neq k$. For these aggregates, the operator '$\neq$' can be treated as the disjunction of the operators '$>$' and '$<$'. Consider the aggregate $A = MAX(.) \neq k$. A is logically equivalent to $A_1 \vee A_2$, where $A_1 = MAX(.) > k$ and $A_2 = MAX(.) < k$. Let R and S be two sets of atoms, it is easy to show that $R \models_S A$ iff $R \models_S A_1$ or $R \models_S A_2$. The other class is the aggregate $SUM(.) \neq k$ and the related aggregate $AVG(.) \neq k$. For these aggregates, the operator '$\neq$' cannot be treated as the disjunction of '$>$' and '$<$', since the conditional satisfaction may not be preserved. Below is an example.

Example 7.7. Let $A = SUM(\{X \mid p(X)\}) \neq -1$, $A_1 = SUM(\{X \mid p(X)\}) > -1$ and $A_2 = SUM(\{X \mid p(X)\}) < -1$. Note that A is logically equivalent to $A_1 \vee A_2$. Consider $S = \{p(1)\}$ and $M = \{p(1), p(2), p(-3)\}$. While S conditionally satisfies A w.r.t. M (i.e., $S \models_M A$), it is not the case that S conditionally satisfies A_1 w.r.t. M or S conditionally satisfies A_2 w.r.t. M. □

To compute the answer sets of the logic programs with aggregates in the second class, the transformation approach proposed in [87] may be used, which in the worst case will

transform an aggregate program to a normal program whose size is exponential in the size of the aggregate program. This is closely related to a result of [76], which shows that the answer set existence problem of logic programs with aggregates $SUM(.) \neq k$ or $AVG(.) \neq k$, is of a complexity higher than NP.

7.4 Experiments

Our theoretical studies show that an aggregate program can be translated to weight constraint programs for which the answer sets are exactly the stable models. This leads to an approach for answer set computation for aggregate programs, where the answer sets of an aggregate program are computed as the stable models of its corresponding strongly satisfiable weight constraint program. We develop the system ALPARSE to evaluate the efficiency of the approach. ALPARSE consists of two parts: a front-end translator which translates an aggregate program to a strongly satisfiable program using the translation in Section 7.3.2 and an ASP solver which supports weight constraints. In the next two sections, respectively, we use SMODELS and CLASP as the ASP solvers for ALPARSE and compare ALPARSE with other implementations for aggregate programs.

The experiments are run on Scientific Linux release 5.1 with 3GHz CPU and 1GB RAM. The reported execution time of ALPARSE consists of the transformation time (from aggregates to weight constraints), the grounding time (calling LPARSE for SMODELS and GRINGO for CLASP), and the search (by SMODELS or CLASP) time. The execution time of SMODELS^A consists of grounding time, search time and unfolding time (computing the solutions to aggregates). The execution time of DLV includes grounding time and search time (the grounding phase is not separated from the search in DLV).

7.4.1 ALPARSE based on SMODELS

We code logic programs with aggregates as weight constraint programs and use SMODELS 2.32 for the stable model computation. If a benchmark program is not already strongly satisfiable, it will be transformed into one, thus we can use the current implementation of SMODELS for our experiments.

We compare our approach with two systems, SMODELS^A and DLV .

A Comparison with SMODELS^A

We compare our approach to the unfolding approach implemented in the system SMODELS^A [21].[4]

The experimental results are reported in Table 7.1, where the sample size is measured by the argument used to generate the test cases. The execution times are the average of one hundred randomly generated instances for each sample size. The results show that SMODELS is often faster than SMODELS^A, even though both use the same search engine.

The first set of problems in Table 7.1 is the company control problem. In the problem, a collection of companies and the percentage of one company which is owned by another company are given. A company A controls another company B if the sum of shares of B owned either directly by A or by companies controlled by A, is more than 50%. The problem is to determine the control relationship between the companies involved. The aggregate used in the program is SUM.

The second set of problems is the employee raise problem. In this problem, the working hours for each quarter of a year of a set of employees are given. The problem is to choose a number of employees to be promoted. The requirement for the promotion of an employee is that the number of working hours of the employee in a year is beyond a given threshold. The aggregate used in the program is SUM.

The third set of problems is the party invitations problems. In this problem, a number of persons are given. A person may be a friend of some other person. We are planning a party and want people that come to the party as more as possible. A person will come to the party only if the number of his or her friends that come to the party is greater than some number. The aggregate used in the program is $COUNT$.

The fourth (NM1) and fifth (NM2) problems are benchmarks created by the authors of smodels^A. The aggregates used in the programs are MAX and MIN, respectively.

Scale-up could be a problem for SMODELS^A , due to the exponential blowup of the size of the unfolded program for a given aggregate program. For instance, given an aggregate $COUNT(\{a \mid a \in S\}) \geq k$, SMODELS^A would list all *aggregate solutions* in the unfolded program, whose number is $C^k_{|S|}$. For a large domain S and k being around $|S|/2$, this is a huge number. If one or a few solutions are needed, ALPARSE takes little time to compute the corresponding weight constraints.

[4]The benchmarks and programs can be found at www.cs.nmsu.edu/∼ielkaban/asp-aggr.html.

A Comparison with DLV

In [1] the seating problem was chosen to evaluate the performance of DLV [5]. The problem is to generate a sitting arrangement for a number of guests, with m tables and n chairs per table. Guests who like each other should sit at the same table; guests who dislike each other should not sit at the same table. The aggregate used in the problem is $COUNT$. We use the same setting to the problem instances as in [1]. The results are shown in Table 7.2. The instance size is the number of atom occurrences in the ground programs. We report the result of the average over one hundred randomly generated instances for each problem size.

The experiments show that, by encoding logic programs with aggregates as weight constraint programs, ALPARSE solves the problem efficiently. For large instances, the execution time of ALPARSE is about one order of magnitude lower than that of DLV and the sizes of the instances are also smaller than those in the language of DLV .

Weight constraint programs vs. normal programs for global constraints

Some global constraints can be encoded by weight constraints compactly. We have experimented with the pigeon-hole problem modeled by the `alldifferent` constraint. The weight constraint program that encodes `alldifferent` is about one order of magnitude smaller than the normal program encoding [61]. The execution of the weight constraint program is 6-7 times faster than its normal program counterpart for hard unsatisfiable instances where the number of holes is one less than the number of pigeons, on the same machine under default settings. See Table 7.3 for the results.

7.4.2 ALPARSE based on CLASP

We use the published programs for the benchmarks [6]. We set the cutoff time to 600 seconds. The instances that are solved in the cutoff time are called “solvable”, otherwise “unsolvable”. In the figures of the experiment results, the running time of unsolvable instances are just plotted as 600 seconds. In the summary of the experiments Table 7.4, the “Execution Time “ is the average running time in seconds for the solvable instances.

The experimental results are shown in Figures 7.1 to 7.12. They are summarized in Table 7.4. It can be seen that ALPARSE is constantly faster than DLV by several orders of magnitude, except for the Towers of Hanoi benchmark.

[5] The program contains a disjunctive head, but can be easily transformed to a non-disjunctive program.

[6] The benchmarks and programs can be found at `http://asparagus.cs.uni-potsdam.de/contest/`

During the writing of the thesis, CLASP is progressed to support aggregates SUM, MIN and MAX. The weight constraints are implemented as the aggregate SUM. The aggregates used in the benchmarks are SUM except for Towers of Hanoi, where the aggregate MAX is used. We also tried the CLASP program, where aggregate MAX is not replaced by its weight constraint encoding (Note that, the answer sets of this aggregate program coincide with the corresponding weight constraint program.). The performances of CLASP on these two programs are similar.

As we have mentioned, the transformation approach indicates that it is important to focus on an efficient implementation of aggregate SUM rather than on implementing other aggregates such as AVG and $TIMES$ in CLASP , since they can be encoded by SUM.[7]

The following is a description of the benchmarks.

15-Puzzle

In 15-puzzle, we have a 4×4 grid where there are 15 numbers (1 to 15) and one blank. The goal is to arrange the numbers from their initial configuration to the goal configuration by sliding one number at a time to its adjacent blank position. Let (x, y) be the coordinates of a number on the grid and (i, j) be those of the blank. Then (x, y) and (i, j) are adjacent if $|x - i| + |y - j| = 1$.

Schur Numbers

The input of the Schur number problem consists of two integers m and n. We need to distribute integers from 1 to n to m disjoint sets so that all of the sets are sum-free. By sum-free, we mean if x and y both belong to the set, then $x + y$ is not in the set.

Blocked N-queens

The blocked N-queens problem is a variant of the N-queens problem. In the blocked N-queens problem we have an $N \times N$ board and N queens. Each square on the board can hold at most one queen. Some squares on the board are blocked and cannot hold any queen. A conflict arises when any two queens are assigned to the same row, column or diagonal.

A blocked N-queens is an assignment of the N queens to the non-blocked squares of the board in a conflict-free manner.

[7] The aggregate $TIMES$ can be translated to SUM, using the logarithm transformation.

Weighted Spanning Tree

Let G be a directed graph. Let every edge in the graph be assigned a weight. A spanning tree is w-bounded if for every vertex the sum of the weights of the outgoing edges at this vertex is at most w.

In the weighted spanning tree problem we are given a directed graph $G = (V, E, W)$, where V is the set of vertices, E is the set of edges and W is a function that maps each edge in the graph to an integer weight. We are also given an integer bound w. The goal is to find a w-bounded spanning tree in G.

Bounded Spanning Tree

Let G be a directed graph. A spanning tree is d-bounded if for every vertex the number of outgoing edges at this vertex is at most d.

In the bounded spanning tree problem we are given a directed graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. We are also given a bound d. The goal is to find a d-bounded spanning tree in G.

Hamiltonian Cycle

A Hamiltonian cycle (or HC for short) in an undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. A set of edges C is a HC in G if every vertex v in V occurs exactly once in C. The input of the HC problem is an undirected graph. The goal is to find a HC in the graph.

Towers of Hanoi

The classic Tower of Hanoi (ToH) problem has three pegs and n disks of different size. Initially, all n disks are on the left-most peg in sorted order. The goal is to move all n disks to the right-most peg with the help of the middle peg. The rules are:

1. move one disk at a time
2. only the top disk on a peg can be moved
3. larger disk cannot be placed on top of a smaller one

It is known that for a classic ToH problem with n disks, the shortest plan for moving all n disks from the left-most peg to the right-most peg consists of $2^n - 1$ moves.

Social Golfer

In the problem, we have n golf players, each of whom play golf once a week, and always in groups of m. The goal is to find a p-week schedule with 'maximum socialization'; that is, as few repeated pairs as possible, and finding a schedule of minimum length such that each golfer plays with every other golfer at least once ('full socialization').

Weighted Latin Square

The weighted Latin-square problem is a variant of the Latin-square problem. Given an $n \times n$ weights $wt(i, j)$ and a bound w, a weighted Latin-square is an $n \times n$ array a such that

1. each entry contains an integer from $\{1, 2, ..., n\}$
2. no row contains the same integer twice
3. no column contains the same integer twice
4. $\forall i$, $1 \leq i \leq n$, $a(i, 1) \times wt(i, 1) + a(i, 2) \times wt(i, 2) + a(i, n) \times wt(i, n) \leq w$

The goal in the problem is to find a weighted Latin square.

Weight-bounded Dominating Set

Let $G = (V, E)$ be a directed graph. Each edge (u, v) in G is associated with a positive weight $w_{u,v}$. A subset D of V is a w-dominating set of G if, for every vertex v in V, at least one of the following conditions holds:

1. v is in D;
2. $\sum_{\{y \mid (v,y) \in E \text{ and } y \in D\}} w_{v,y} \geq w$;
3. $\sum_{\{z \mid (z,v) \in E \text{ and } z \in D\}} w_{z,v} \geq w$.

The goal is to find a w-dominating set of G of size at most k, where k is an integer parameter.

Traveling Salesperson

Given an integer w and an undirected graph $G = (V, E)$, where every edge (u, v) is associated with a positive weight $w_{u,v}$, the goal is to find a Hamiltonian cycle in G (a simple cycle that visits all vertices in G exactly once) such that the sum of the edges in the cycle is at most w.

Car Sequencing

The goal in the problem is to produce a number of cars. The cars are not identical, because different options are available as variants on the basic model. The assembly line has different stations which install the various options (air-conditioning, sun-roof, etc.). These stations have been designed to handle at most a certain percentage of the cars passing along the assembly line. Furthermore, the cars requiring a certain option must not be bunched together, otherwise the station will not be able to cope. Consequently, the cars must be arranged in a sequence so that the capacity of each station is never exceeded. For instance, if a particular station can only cope with at most half of the cars passing along the line, the sequence must be built so that at most 1 car in any 2 requires that option. The problem has been shown to be NP-complete (Gent 1999).

The following two constraints must be satisfied.

1. The given number of cars for each class must be produced (a class is a subset of options) .

2. The cars must be arranged in a sequence so that the capacity of each station is never exceeded.

Program	Sample Size	ALPARSE	SMODELS^A
Company Control	20	0.03	0.09
Company Control	40	0.18	0.36
Company Control	80	0.87	2.88
Company Control	120	2.40	12.14
Employee Raise	15/5	0.01	0.69
Employee Raise	21/15	0.05	4.65
Employee Raise	24/20	0.05	5.55
Party Invitation	80	0.02	0.05
Party Invitation	160	0.07	0.1
NM1	125	0.61	0.21
NM1	150	0.75	0.29
NM2	125	0.65	2.24
NM2	150	1.08	3.36

Table 7.1: Benchmarks used by SMODELS^A

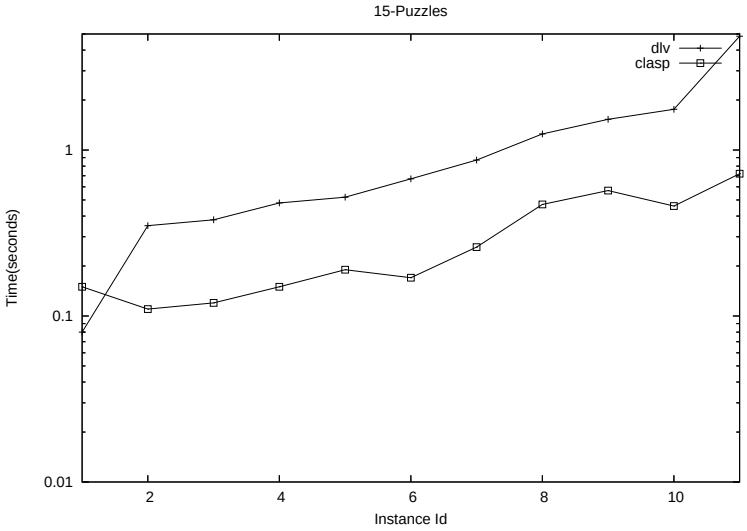

Figure 7.1: 15-Puzzle

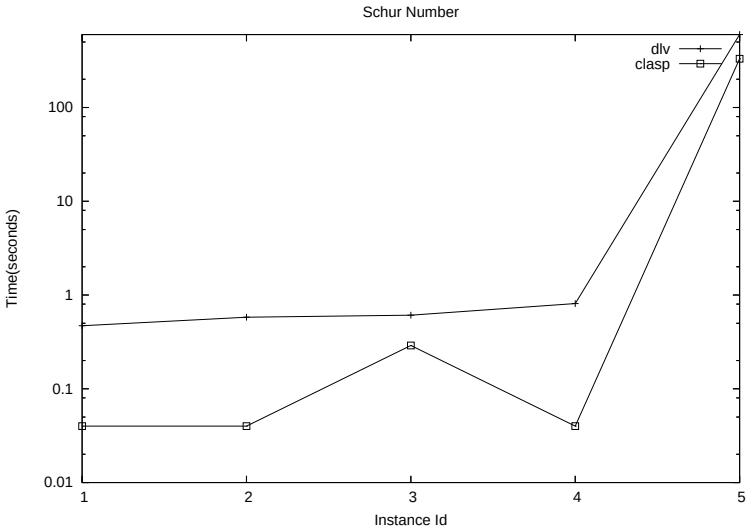

Figure 7.2: Schur Number

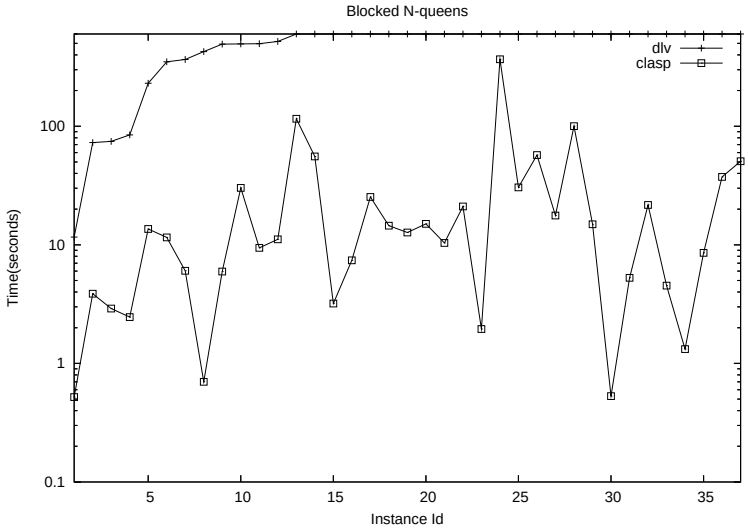

Figure 7.3: Blocked N-queens

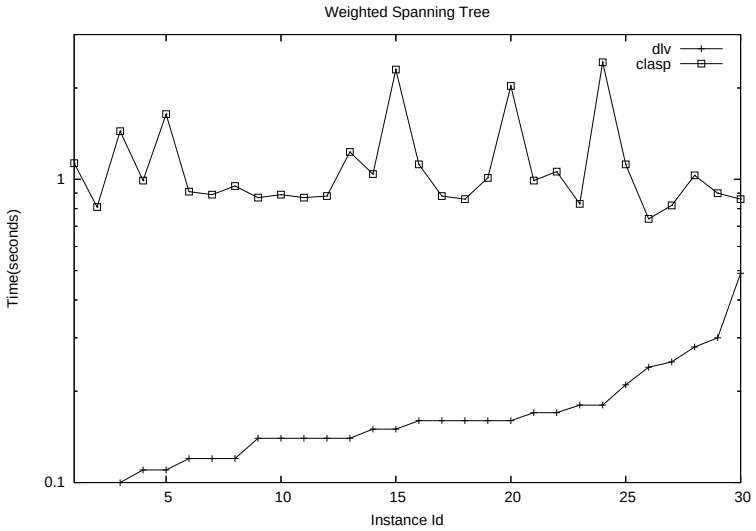

Figure 7.4: Weighted Spanning Tree

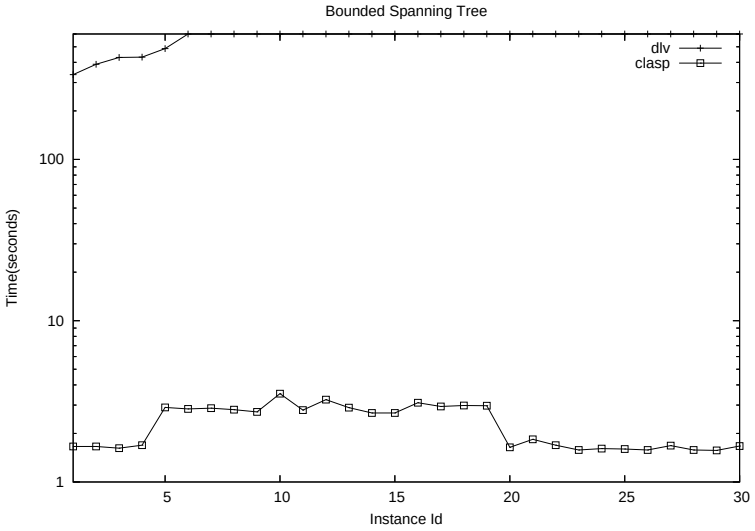

Figure 7.5: Bounded Spanning Tree

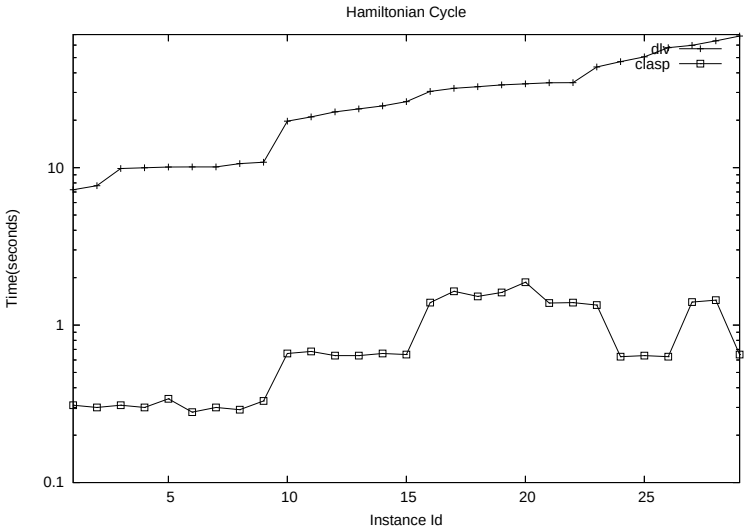

Figure 7.6: Hamiltonian Cycle

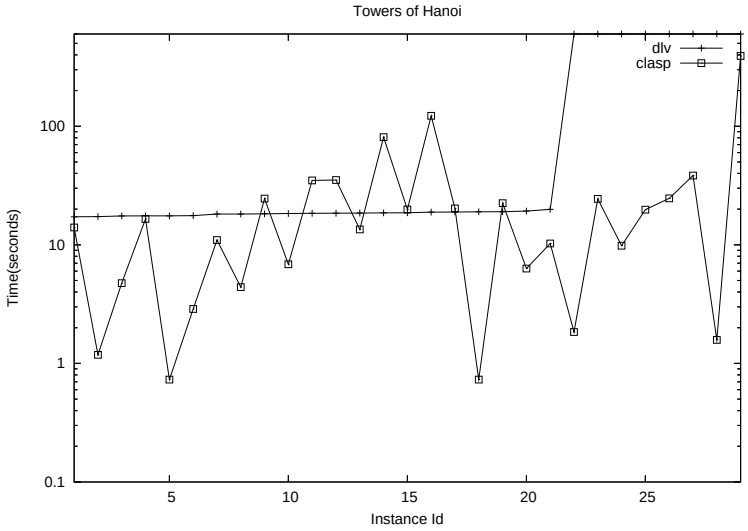

Figure 7.7: Towers of Hanoi

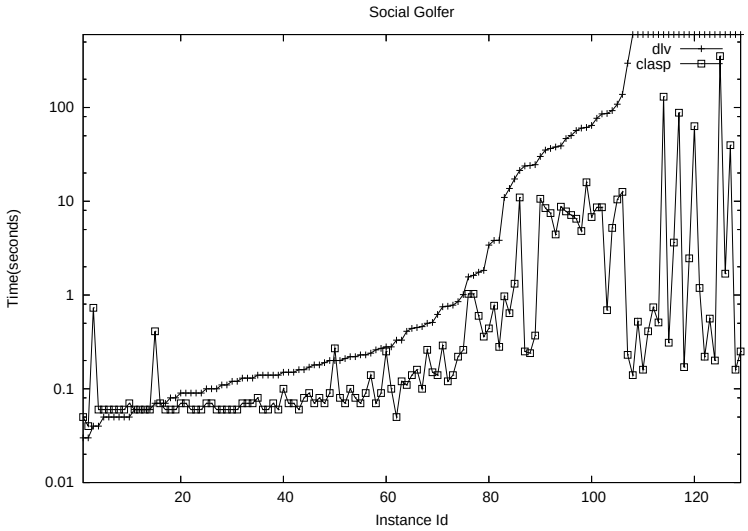

Figure 7.8: Social Golfer

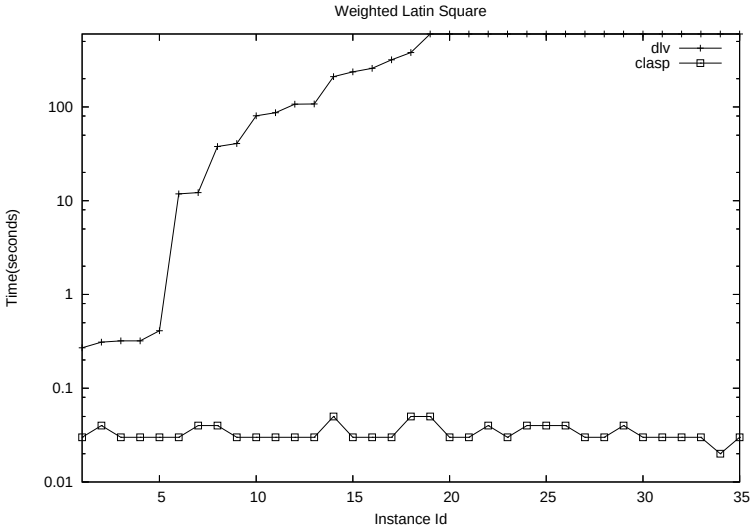

Figure 7.9: Weighted Latin Square

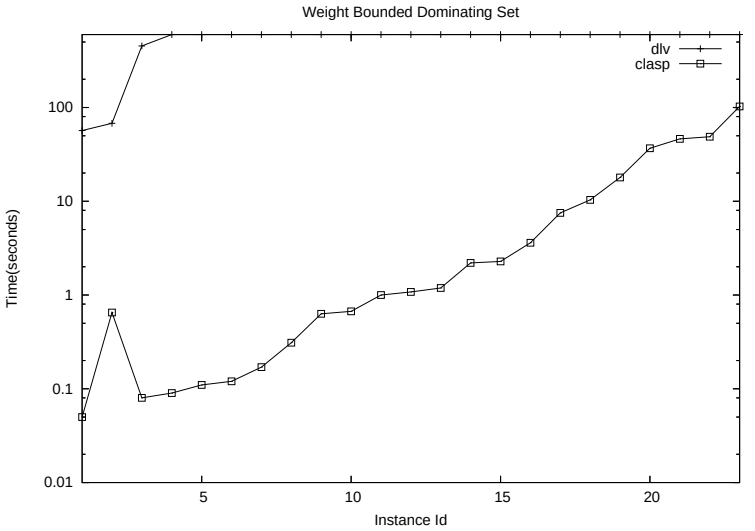

Figure 7.10: Weight Bounded Dominating Set

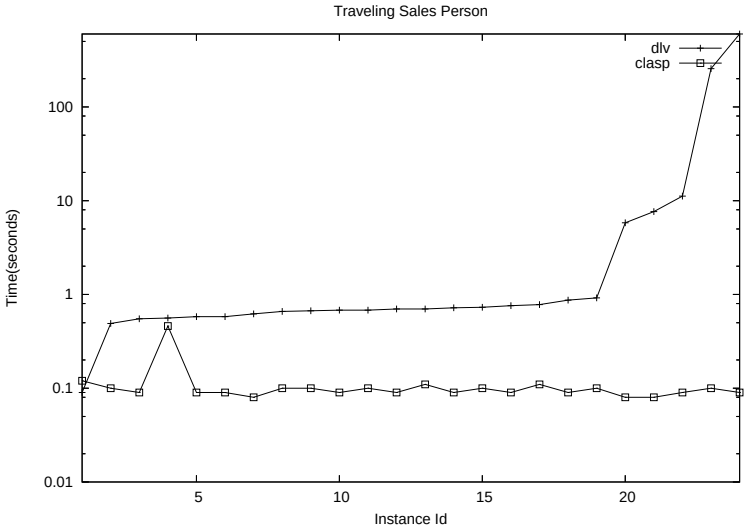

Figure 7.11: Traveling Sales Person Problem

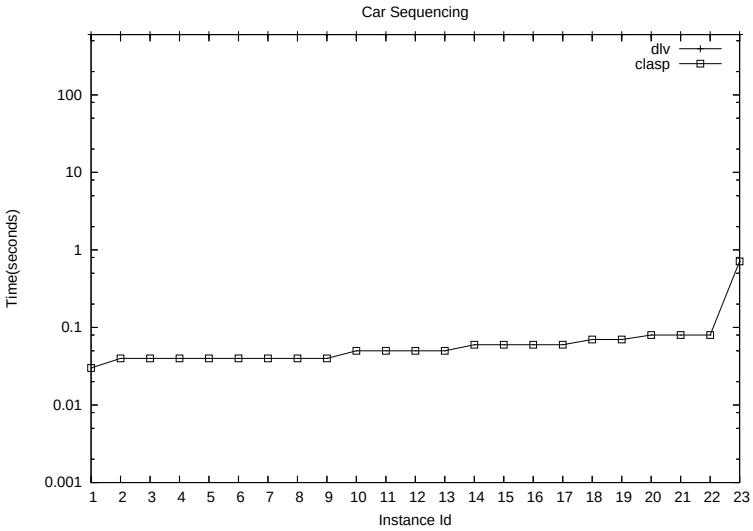

Figure 7.12: Car Sequencing

C	T	Execution Time		Instance Size	
		ALPARSE	DLV	ALPARSE	DLV
4	3	0.1	0.01	293	248
4	4	0.2	0.01	544	490
5	5	0.58	0.02	1213	1346
5	10	0.35	0.31	6500	7559
5	15	1.24	1.88	18549	22049
5	20	3.35	7.08	40080	47946
5	25	8.19	64.29	73765	88781
5	30	16.42	152.45	12230	147567

Table 7.2: Seating

p	h	Execution Time		Instance Size	
		ALPARSE	Normal	ALPARSE	Normal
5	4	0.00	0.01	98	345
6	5	0.01	0.01	142	636
7	6	0.01	0.06	194	1057
8	7	0.09	0.49	254	1632
9	8	0.74	4.38	322	2385
10	9	6.89	43.66	398	3340
11	10	71.92	480.19	482	4521
12	11	827.85	5439.09	574	5952

Table 7.3: Pigeon hole

7.5 Conclusion

We show that for a large class of programs the stable models semantics coincides with the answer set semantics. In general, answer sets are all stable models. When a stable model is not an answer set, it may be circularly justified. We propose a transformation, by which a weight constraint program can be translated to strongly satisfiable programs, for which all stable models are answer sets and thus well-supported models.

As an issue of methodology, we have shown that most standard aggregates can be encoded by weight constraints. Therefore the ASP systems that support weight constraints can be applied to efficiently compute the answer sets of logic programs with almost all standard aggregates. The system ALPARSE demonstrates the efficiency of this approach.

As we have shown that stable models that are not sanctioned by answer set semantics may or may not be circular under our definition of circular justification. This left open the question of what would be the desired semantics for weight constraint programs. It seems that the notion of unfounded sets can serve as a definition for a new semantics for

Benchmarks	Number of Instances	Solved Instances		Execution Time	
		ALPARSE	DLV	ALPARSE	DLV
15 Puzzle	11	11	11	0.31	1.16
Schur Number	5	5	4	0.10	0.62
Blocked N-queens	37	37	12	8.94	328.92
Wt. Spanning Tree	30	30	30	0.12	0.17
Bd. Spanning Tree	30	30	5	1.91	414.42
Hamiltonian Cycle	29	29	29	0.84	29.22
Towers of Hanoi	29	29	21	21.61	18.35
Social Golfer	168	129	107	1.52	14.69
Wt. Latin Square	35	35	18	0.03	105.01
Wt. Dominating Set	30	23	3	0.26	192.53
Traveling Sales	24	24	23	0.11	12.74
Car Sequencing	54	23	0	0.08	–

Table 7.4: Summary ALPARSE and DLV

weight constraint programs, since it appears to separate the desired stable models from the undesired ones. Then, a question is whether a transformation exists that eliminates only circular models.

Another question of interest is on the expressiveness of weight constraints. We know there are difficulties representing aggregates with the operator $\neq$. Then it is interesting to investigate characterizations of the constraints that can be encoded by weight constraints.

Chapter 8

Semantics for abstract constraint programs

Logic programs with *abstract constraint atoms* (or *c-atoms*) is proposed as a general framework for investigating, in a uniform fashion, various extensions of logic programming, including weight constraint and aggregate programs. A c-atom expresses a constraint over a set of atoms. It takes the form (D, C), where D is a set of atoms, the domain of the constraint, and C a collection of the subsets from the power set of D serving as the set of solutions to the constraint.

Different semantics have been proposed for logic programs with c-atoms. Similarly to the case in aggregate programs, we take the semantics based on conditional satisfaction (also called answer set semantics) as the semantics of logic programs with c-atoms. We present the answer set semantics in Section 8.1. In Section 8.2, we introduce other semantics and the relationships between these semantics and the answer set semantics.

8.1 Answer set semantics

We assume a propositional language with a countable set of propositional *atoms*. Once a program P is defined, we will denote by $At(P)$ the set of atoms appearing in P.

A *abstract constraint atom* (*c-atom*) is of the form (D, C), where D is a finite set of atoms (the *domain* of the c-atoms) and $C \subseteq 2^D$ (the *admissible solutions*, or simply *solutions*, of the c-atom). Intuitively, a c-atom (D, C) is a constraint on the set of atoms D, and C represents its solutions. Given a c-atom $A = (D, C)$, we use A_d and A_c to refer to D and C, respectively.

Example 8.1. Consider the aggregate $A = SUM(\{X \mid p(X)\}) \geq -1$, where $D(X) = \{1, -2\}$. A can be written as a c-atom $B = (D, C)$ whose domain is $D = \{p(1), p(-2)\}$

and

$$C = \{S \mid S \subseteq D, SUM(S) \geq -1\} = \{\emptyset, \{p(1)\}, \{p(-2), p(1)\}\}.$$

□

Let A be a c-atom. A is said to be *elementary* if it is of the form $(\{a\}, \{\{a\}\})$, which is just written as a; A is *monotone* if for every $X \subseteq Y \subseteq A_d$, $X \in A_c$ implies that $Y \in A_c$; A is *nonmonotone* if it is not monotone; A is *antimonotone* if A_c is closed under subset, i.e., for every $X, Y \subseteq A_d$, if $Y \in A_c$ and $X \subseteq Y$ then $X \in A_c$; A is *convex* if for every $X, Y, Z \subseteq A_d$ such that $X \subseteq Y \subseteq Z$ and $X, Z \in A_c$, then $Y \in A_c$.

A set of atoms M satisfies a c-atom A, written $M \models A$, if $M \cap A_d \in A_c$. Otherwise M does not satisfy A, written $M \not\models A$. $M \models \texttt{not}\ A$ (or $M \models \neg A$) if $M \not\models A$. A c-atoms of the form $(D, \emptyset)$ is not satisfiable. It is often denoted by $\bot$. On the other hand, some c-atoms are tautologies. For example, all monotone c-atoms of the form $(D, 2^D)$ are tautologies.

Satisfiability naturally extends to conjunctions of c-atoms (sometimes written as a set) and disjunctions of c-atoms.

A logic program with c-atoms, also called a *(abstract) constraint program* (or just *program* in this chapter), is a finite set of rules of the form

$$A \leftarrow A_1, \ldots, A_k, \texttt{not}\ A_{k+1}, \ldots, \texttt{not}\ A_n. \tag{8.1}$$

where A and A_i's are arbitrary c-atoms. The literals $\texttt{not}\ A_j$ are called *negative c-atoms*.

For a rule r of the form (8.1), we define

$$hd(r) = A \text{ and } bd(r) = \{A_1, ..., A_k, \texttt{not}\ A_{k+1}, \ldots, \texttt{not}\ A_n\},$$

which are called the *head* and the *body* of r, respectively.

A rule r is said to be *basic* if every c-atom in it occurs positively, and either $hd(r)$ is elementary or r is a constraint. A program P is *basic* if every rule in it is basic; P is a *normal program* if every c-atom in it is elementary; P is a *monotone-constraint program* if every c-atom in it is monotone.

Following [77], a negative c-atom $\texttt{not}\ A$ in a program is interpreted by its complement, and substituted by c-atom $\overline{A}$, where $\overline{A}_d = A_d$ and $\overline{A}_c = 2^{A_d} \setminus A_c$. For example, the negative elementary c-atom $\texttt{not}\ (\{a\}, \{\{a\}\})$ will be replaced by c-atom $(\{a\}, \{\emptyset\})$.

Due to this assumption, in the sequel, if not said otherwise, constraint programs contain only positive c-atoms.

A set of atoms $M \subseteq At(P)$ is a *model* of a program P if for each rule $r \in P$, $M \models hd(r)$ whenever $M \models bd(r)$. M is a *supported* model of P if for any $a \in M$, there is $r \in P$ such that $a \in hd(r)_d$ and $M \models bd(r)$.

Answer sets for constraint programs are defined in two steps. In the first step, answer sets for basic programs are defined, based on the *conditional satisfaction* of a c-atom.

Definition 8.1. [77] Let M and S be sets of atoms. The set S *conditionally satisfies* a c-atom A, w.r.t. M, denoted by $S \models_M A$, if $S \models A$ and for every $I \subseteq A_d$ such that $S \cap A_d \subseteq I$ and $I \subseteq M \cap A_d$, we have that $I \in A_c$.

An operator T_P is then defined.

Definition 8.2. Let P be a basic program and R and S two sets of atoms. $T_P(R, S)$ is defined as:

$$T_P(R, S) = \{a : \exists r \in P, hd(r) = a \neq \bot,\ R \models_S bd(r)\}.$$

The operator T_P is monotonic at its first argument, with respect to its second argument.

Proposition 8.1. Let M be a model of P, and let $S \subseteq U \subseteq M$. Then $T_P(S, M) \subseteq T_P(U, M) \subseteq M$.

Definition 8.3. Let M be a model of a basic program P. M is an *answer set* for P iff $M = T_P^\infty(\emptyset, M)$, where $T_P^0(\emptyset, M) = \emptyset$ and $T_P^{i+1}(\emptyset, M) = T_P(T_P^i(\emptyset, M), M)$, for all $i \geq 0$.

Next, a constraint program is represented by its instances in the form of basic programs, and the answer sets of the former are defined in terms of the answer sets of the latter.

Definition 8.4. Let P be a constraint program and $r \in P$. For each $\pi \in hd(r)_c$, the *instance* of r w.r.t. π is the set of rules consisting of

1. $b \leftarrow bd(r)$, for each $b \in \pi$, and
2. $\bot \leftarrow d, bd(r)$, for each $d \in hd(r)_d \backslash \pi$.

An *instance of* P is a basic program obtained by replacing each rule of P with one of its instances.

Definition 8.5. [77] Let P be a constraint program and M a set of atoms. M is an answer set for P iff M is an answer set for one of its instances.

Example 8.2. Let P be the program consists of the following rules.

$$c \leftarrow a \tag{8.2}$$

$$(\{a,b\},\{\{a\},\{b\}\}) \leftarrow (\{a,b\},\{\emptyset,\{a\},\{b\}\}) \tag{8.3}$$

The instances of P, denoted by P_1 and P_2, respectively are P_1:

$$c \leftarrow a \tag{8.4}$$

$$a \leftarrow (\{a,b\},\{\emptyset,\{a\},\{b\}\}) \tag{8.5}$$

$$\bot \leftarrow b,(\{a,b\},\{\emptyset,\{a\},\{b\}\}) \tag{8.6}$$

and P_2:

$$c \leftarrow a \tag{8.7}$$

$$b \leftarrow (\{a,b\},\{\emptyset,\{a\},\{b\}\}) \tag{8.8}$$

$$\bot \leftarrow a,(\{a,b\},\{\emptyset,\{a\},\{b\}\}) \tag{8.9}$$

It can be checked that $\{a,c\}$ and $\{b\}$ are answer sets of P_1 and P_2 respectively. Thus, both of them are answer sets of P. □

8.2 Other semantics

8.2.1 MR-answer set semantics

The concept of logic programs with c-atoms was originally introduced by Marek and Remmel [54] where the programs with c-atoms were named *set constraint programs* (SC-programs).

The proposed semantics (let us call it MR-answer set semantics) is based on the upper-closure of a c-atom, with respect to the domain of the c-atom. For a c-atom $A = (D, C)$, the upper closure of A, denoted $\hat{A}$ is a c-atom, where $\hat{A}_d = D$ and

$$\hat{A}_c = \{S \mid S \subseteq D \; and \; \exists Z \in C \; such \; that \; Z \subseteq S\}.$$

Given an SC-program P and a set of atoms M. The NSS transform, NSS(P, M), is obtained by firstly removing from P all rules whose body are not satisfied by M and then, for each rule r of form $A \leftarrow A_1, ..., A_n$ and an atom $a \in C \cap M$, generating a rule

$$a \leftarrow \hat{C}_1, ..., \hat{C}_n$$

to replace r.

Let P be a SC-program and M a set of atoms. M is an MR-answer set of P if $M \models P$ and M is the least model of NSS(P, M).

It was shown in [77] that any answer set of a program is an MR-answer set of the program.

The MR-answer set semantics does not guarantee the minimality of answer sets.

Example 8.3. Consider the program P:

$$\begin{aligned} a &\leftarrow \\ c &\leftarrow \\ d &\leftarrow (\{a,c,d\}, \{\{a\}, \{a,c,d\}\}) \end{aligned}$$

According to the conditional satisfaction semantic, this program has only one answer set $M_1 = \{a, c\}$. But according to the MR-answer set semantics, $M_2 = \{a, c, d\}$ is another answer set, since the reduct of P w.r.t. M_2 is

$$\begin{aligned} a &\leftarrow \\ c &\leftarrow \\ d &\leftarrow (\{a,c,d\}, \{\{a\}, \{a,c\}, \{a,d\}, \{a,c,d\}\}) \end{aligned}$$

and {a,c,d} is a minimal model of the reduct.

The reason that causes the non-minimal answer set is that the c-atom $\{a, c, d\}, \{\{a\}, \{a, c, d\}\})$ in the last rule of P is replaced by its upper-closure $\{\{a\}, \{a, c\}, \{a, d\}, \{a, c, d\}\})$. □

8.2.2 MT-answer set semantics

Marek and Truszczyński in [54] propose semantics for logic programs with monotone c-atoms (monotone constraint programs). In the proposal (let us call it MT-answer set semantics), the MT-answer sets are defined as the fixpoint of a *nondeterministic one-step* operator $T_P^{nd}(P, M)$, where P is a monotone constraint program and M a set of atoms, that is,

$$T_P^{nd}(P, M) = \{S \mid S \subseteq hset(P(M)) \; and \; S \models hd(r), \; for \; each \; rule \; r \in P(M)\} \quad (8.10)$$

where $P(M)$ is the set of rules in P whose body is satisfied by M. Recall that, for a set of rules P, $hset(P)$ is the union of the atoms appearing in the heads of rules in P.

Let P be a monotone constraint program where all c-atoms are positive. A P-computation is a sequence $\langle X_k \rangle$ (indexed with non-negative integers) such that (1). $X_0 = \emptyset$, (2). $X_{k+1} \in T_P^{nd}(X_k)$, and (3). $X_k = \bigcup_{i<k} X_i$.

Let P be a monotone constraint program with only positive c-atoms and M a set of atoms. M is an MT-answer set of P if $M \models P$ and M is the least fixpoint of P-computation. If P contains negative c-atoms, a model M of P is an MT-answer sets if M is the least fixpoint of P-computation on the program reduct P^M, which is obtained by firstly removing the rules from P that are not satisfied by M and then removing all negative c-atoms from the bodies of the remaining rules.

It has been shown that, for monotone constraint programs, the answer set semantics, MR-answer set semantics and MT-answer set semantics coincide.

Chapter 9

Loop formulas for abstract constraint programs

9.1 Motivation

In this chapter, we present a method for answer set computation for logic programs with arbitrary c-atoms c-atoms, by formulating loop formulas for these programs, for the answer set semantics. This semantics is known to coincide with the ultimate stable semantics for logic programs with aggregates [18]. Our work follows the previous work on loop formulas for normal logic programs [43], and loop formulas for logic programs with monotone and convex constraints [49]. We show that answer sets of a logic program with arbitrary c-atoms are precisely the models of completion that satisfy our loop formulas. Since the semantics based on conditional satisfaction agrees with the semantics for logic programs with monotone and convex c-atoms, our loop formulas are applicable to the latter. Actually, for this class of programs, our loop formulas are simpler (sometimes *exponentially* simpler) than those proposed in [49]. Our results can be applied to any constraint solver, where models of completion can be computed. The next question is which constraint solvers are suitable for this task.

In [49], it is shown how logic programs with cardinality and weight constraints, called *lparse* programs, can be encoded by pseudo-Boolean (PB) theories. Here, we advocate in addition that PB constraint solvers (e.g. [11]) are attractive candidates for computing models of completion for programs with arbitrary c-atoms, especially for programs with aggregates and global constraints. This is due to the observation that typical aggregates in logic programs, as well as some global constraints, can be encoded as PB theories compactly. That is, the size of the PB encoding of such a constraint is linear in the size of the constraint's domain. This is in contrast with the unfolding approach [21, 64] - translating

a logic program with aggregates to a normal program, where the resulting normal program could be of exponential size for the same constraint. We show some encodings to illustrate this connection.

In the next section, we introduce the concept of completion of abstract constraint programs. The loop formulas are presented in Section 9.3. The relation to previous work is given in Section 9.4. In Section 9.5, we show that most aggregates and global constraints can be encoded by PB theories. Section 9.6 contains the conclusions.

9.2 Completion

In the sequel, for a set of atoms $S = \{a_1, \ldots, a_n\}$, we use $S^\wedge$ to denote the conjunction $a_1 \wedge \ldots \wedge a_n$, and $\neg S$ to denote the conjunction $\neg a_1 \wedge \ldots \wedge \neg a_n$.

For a c-atom A, we may be interested only in the domain atoms that actually appear in some admissible solution of A. We thus define $ASet(A) = \{a : a \in \pi, \text{ for some } \pi \in A_c\}$.

Following [49], the *completion* of a constraint program P, denoted by $Comp(P)$, consists of the following formulas

- $[bd(r)]^\wedge \rightarrow hd(r)$, for each $r \in P$;
- $x \rightarrow \bigvee\{[bd(r)]^\wedge : r \in P, x \in ASet(hd(r))\}$, for each atom $x \in At(P)$.

The first formula above captures the if definition in a rule, whereas the second completes the definition by adding the only if part (Please refer Example 9.3).

The completion of a constraint program is a set of formulas with c-atoms. The notion of satisfaction and models for constraint programs extends in a natural way to formulas with c-atoms.

Note that in the definition of [49], a negative literal `not` A in completion is interpreted as $\neg A$, while in this paper it is interpreted by its complement $\overline{A}$. The two are consistent since for any set of atoms $I \subseteq At(P)$, $I \models \neg A$ if and only if $I \models \overline{A}$.

Theorem 9.1. [49] Let P be a constraint program. A set of atoms M is a supported model of P if and only if M is a model of $Comp(P)$.

9.3 Loop formulas

To define the loop formulas for a constraint program, we introduce the notion of *local power set*.

Definition 9.1. Let A be a c-atom. A pair of sets $\langle B, T\rangle$, where $B \subseteq T \subseteq A_d$, is called a *local power set (LPS)* of A, if $B \in A_c$, $T \in A_c$ and for any set I such that $B \subseteq I \subseteq T$, we have $I \in A_c$.

Intuitively, $\langle B, T\rangle$ represents an "extension" of the power set $2^{T \setminus B}$, where B is the "bottom" element and T is the "top" element.

A local power set $\langle B, T\rangle$ of a c-atom A is said to be *maximal* if there is no other local power set $\langle B', T'\rangle$ of A such that $B' \subseteq B$ and $T \subseteq T'$.

Example 9.1. Let $A = (\{a, b, c\}, \{\emptyset, \{a\}, \{a, b, c\}\})$ be a c-atom. The maximal local power set of A are $\langle \emptyset, \{a\}\rangle$ and $\langle \{a, b, c\}, \{a, b, c\}\rangle\})$. Note that the local power sets $\langle \emptyset, \emptyset\rangle$ and $\langle \{a\}, \{a\}\rangle$ are not maximal. □

The LPS representation of a c-atom is defined as follows.

Definition 9.2. Let A be a c-atom. The LPS representation of A, denoted by A^*, is (A_d, A_c^*), where A_c^* = $\{\langle B, T\rangle : \langle B, T\rangle$ is a maximal LPS of $A\}$.

Example 9.2. Let $A = (\{a, b, c\}, \{\emptyset, \{a\}, \{a, b, c\}\})$. Then,

$$A^* = (\{a, b, c\}, \{\langle \emptyset, \{a\}\rangle, \langle \{a, b, c\}, \{a, b, c\}\rangle\}).$$

□

Let M be a set of atoms and A be a c-atom. We say M satisfies A^*, denoted $M \models A^*$, if $M \models \bigvee\{B^\wedge \wedge \neg(A_d \setminus T) : \langle B, T\rangle \in A_c^*\}$.

The satisfaction of a c-atom can be characterized by the satisfaction of its LPS representation.

Proposition 9.1. Let A be a c-atom and M be a set of atoms. We have $M \models A$ iff $M \models A^*$.

The proof of the proposition is straightforward by the definition of the LPS representation of c-atoms.

A^* can be seen as a compact representation of A. For example, given a monotone c-atom $A = (D, 2^D - \{\emptyset\})$, $A^* = (A_d, \{\langle B, A_d\rangle : B \subseteq A_d, B$ is singleton$\})$.

To capture the loops in constraint programs, we define the dependency graph for a constraint program.

Definition 9.3. Let P be a constraint program. The *dependency graph* of P, denoted by $G_P^a = (V, E)$ (a stands for arbitrary c-atoms), is a directed graph, where

- $V = At(P)$
- (u, v) is a directed edge from u to v in E if there is a rule $r \in P$ such that $u \in ASet(hd(r))$ and $v \in B$, for some $\langle B, T\rangle \in A^*_c$ and $A \in bd(r)$.

Let $G = (V, E)$ be a directed graph. A set $L \subseteq V$ is a *loop* in G if the subgraph of G induced by L is strongly connected. A loop is *maximal* if it is not a proper subset of any other loop in G. A maximal loop is *terminating* if there is no edge in G from L to any other maximal loop.

For any nonempty set $X \subseteq At(P)$, we denote by $G^a_P[X]$ the subgraph of G^a_P *induced* by X.

The idea of loop formula is that for any atom in a loop to be in an answer set of a program, it must be supported by atoms that are not in the loop. To capture this idea, the *restriction* of the LPS representation of a c-atom is defined.

Definition 9.4. Let P be a constraint program, A be a c-atom, and $L \subseteq At(P)$. The *restriction* of A^* to L, denoted by $A^*_{|L}$, is $(A_d, A^*_{c|L})$, where

$$A^*_{c|L} = \{\langle B, T\rangle \in A^*_c : L \cap B = \emptyset\}.$$

Definition 9.5. Let L be a set of atoms and r be a rule $A \leftarrow A_1, \ldots, A_n$. We define the *body formula* of r w.r.t. L as

$$\alpha_L(r) = \pi_1 \wedge \ldots \wedge \pi_n,$$

where $\pi_i = \bigvee\{B^\wedge \wedge \neg(A_{i_d} \setminus T) : \langle B, T\rangle \in A^*_{i_{c|L}}\}$, for each i, $1 \leq i \leq n$. If there is a c-atom A_i such that $A^*_{i_{c|L}} = \emptyset$, then $\alpha_L(r) = false$.

We are ready to give our definition of loop formulas.

Definition 9.6. Let P be a constraint program and L be a loop in G^a_P. The *loop formula* for L, denoted by $LP^a(L)$, is defined as

$$\bigvee L \rightarrow \bigvee\{\alpha_L(r) : r \in P,\ L \cap ASet(hd(r)) \neq \emptyset\}. \tag{9.1}$$

The *loop completion* of a constraint program P, denoted $LComp(P)$, combines the completion of P with the loop formulas for loops in G^a_P, and is defined as

$$LComp(P) = Comp(P) \cup \{LP^a(L) : L \text{ is a loop in } G^a_P\}. \tag{9.2}$$

Example 9.3. Consider the following program P.

$$r_1: \ p \leftarrow (\{a,b,c\}, \{\emptyset, \{a,b\}, \{a,b,c\}\}).$$
$$r_2: \ a \leftarrow p.$$
$$r_3: \ b \leftarrow p.$$

Let $A = (\{a,b,c\}, \{\emptyset, \{a,b\}, \{a,b,c\}\})$. The completion of P is

$$Comp(P) = \{A \rightarrow p, p \rightarrow A, p \rightarrow a, a \rightarrow p, p \rightarrow b, b \rightarrow p\}.$$

The only model of $Comp(P)$ is $M = \{a,b,p\}$. We will see that M is not an answer set for P.

$$A^* = (\{a,b,c\}, \{\langle \emptyset, \emptyset \rangle, \langle \{a,b\}, \{a,b,c\} \rangle\}).$$

$L = \{a,b,p\}$ is a loop in G^a_P, and its loop formula is:

$$LP^a(L) = a \vee b \vee p \rightarrow \alpha_L(r_1) \vee \alpha_L(r_2) \vee \alpha_L(r_3).$$

Since $\alpha_L(r_1) = \neg a \wedge \neg b \wedge \neg c$ and $\alpha_L(r_2) = \alpha_L(r_3) = false$, we get

$$LP^a(L) = a \vee b \vee p \rightarrow (\neg a \wedge \neg b \wedge \neg c).$$

As $M \not\models LP^a(L)$, we conclude that M is not an answer set for P. □

Example 9.4. Consider the program P:

$$d \leftarrow (\{a,b,c,d\}, \{\{a\}, \{a,b\}, \{a,c\}, \{a,b,c\}, \{d\}, \{a,d\}\}).$$
$$a \leftarrow .$$

Let A be the c-atom in the body of the first rule. We have

$$A^* = (\{a,b,c,d\}, \{\langle \{a\}, \{a,b,c\} \rangle, \langle \{a\}, \{a,d\} \rangle, \langle \{d\}, \{a,d\} \rangle\}).$$

The loop formula for the loop $L = \{d\}$ is:

$$LP^a(L) = d \rightarrow (a \wedge \neg d) \vee (a \wedge \neg b \wedge \neg c)$$

As $M = \{a,d\}$ is a model of $Comp(P)$ and $M \models PL^a(L)$, for the only loop L in G^a_P, it is an answer set for P. □

To establish the main theorem, we need the following lemma.

Lemma 9.1. Let P be a basic program. A set $M \subseteq At(P)$ is an answer set for P iff M is a model of $LComp(P)$.

Proof. ($\Rightarrow$) Let M be an answer set for P. Then, M is a supported model of P, so $M \models Comp(P)$. Let L be a loop in G^a_P. If $L \cap M = \emptyset$, then $M \models LP^a(L)$. If $L \cap M \neq \emptyset$ there is a smallest k such that $L \cap T^k_P(\emptyset, M) \neq \emptyset$ and $L \cap T^{k-1}_P(\emptyset, M) = \emptyset$. Let $x \in L \cap T^k_P(\emptyset, M)$. We have $x \notin T^{k-1}_P(\emptyset, M)$ and $x \in T^k_P(\emptyset, M)$. By the definition of T_P, there is a rule $r \in P$ with an elementary head such that $hd(r) = x$, and $T^{k-1}_P(\emptyset, M) \models_M bd(r)$. From Proposition 9.1, we have for any c-atom $A \in bd(r)$, $T^{k-1}_P(\emptyset, M) \models_M A^*_{|L}$. Then, by the monotonicity of the operator T_P (Proposition 8.1 in Chapter 8), we have $T^{k-1}_P(\emptyset, M) \subseteq T^i_P(\emptyset, M) \subseteq T^{i+1}_P(\emptyset, M)$, for $i \geq k$, and $T^\infty_P(\emptyset, M) = M$. It follows from the definition of conditional satisfaction $T^i_P(\emptyset, M) \models A^*_{|L}$, for $i \geq k$. So, $M \models \alpha_L(r)$. Therefore $M \models LP^a(L)$.

($\Leftarrow$) Assume M is a model of $LComp(P)$ and we show $M = T^\infty_P(\emptyset, M)$. The proof of $T^\infty_P(\emptyset, M) \subseteq M$ is routine by an induction on the construction of $T^\infty_P(\emptyset, M)$. We show the other direction, $M \subseteq T^\infty_P(\emptyset, M)$.

Suppose the statement doesn't hold, i.e., for some $x \in M$, $x \notin T^\infty_P(\emptyset, M)$. Then, since $T^\infty_P(\emptyset, M) \subseteq M$, it follows $T^\infty_P(\emptyset, M) \subset M$. Let $M^- = M \setminus T^\infty_P(\emptyset, M)$. In $G^a_P[M^-]$, there are either loops or no loops. If $G^a_P[M^-]$ doesn't have a loop, it's easy to show that $x \in T^\infty_P(\emptyset, M)$, causing a contradiction. Suppose $G^a_P[M^-]$ has a loop, then it has a terminating loop $L \subseteq M^-$. Since $M \models LComp(P)$, there exist $x \in M^-$ and $r \in P$, where $hd(r) = x$, such that $M \models bd(r)$, and for any $A \in bd(r)$,

(1) $M \cap A_d = t$, for some $t \in A_c$ (due to $M \models bd(r)$).

(2) $\exists \langle B, T\rangle \in A^*_c$ such that $B \subseteq t \subseteq T$, $B \in A_c$ (due to (1)), and

  (2.1) $B \models_M A$ (due to (1), (2))

  (2.2) $M \models B^\wedge \wedge \neg(A_d \setminus T)$ and $B \cap L = \emptyset$ (due to $M \models LP^a(L)$)

If we can show $B \subseteq T^j_P(\emptyset, M)$, for some $j > 0$, then from (2.1) above, and since it is the case for any $A \in bd(r)$, we will have $x \in T^{j+1}_P(\emptyset, M)$, resulting in a contradiction.

Consider any $y \in B$. From (2.2), we know $y \notin L$. From (1) and (2) above, it is clear $y \in M$. It follows that either $y \in M^- \setminus L$, or $y \in T^j_P(\emptyset, M)$, for some $j > 0$. In the former case, since L is a terminating loop in $G^a_P[M^-]$, y is not in any loop in $G^a_P[M^-]$ and y doesn't depend on any loop in $G^a_P[M^-]$ (i.e., there is no path from y to any loop in $G^a_P[M^-]$). It is then easy to show that $y \in T^j_P(\emptyset, M)$, for some $j > 0$. Since y is arbitrary, we conclude that $B \subseteq T^j_P(\emptyset, M)$, for some $j > 0$. □

Theorem 9.2. Let P be a constraint program. A set $M \subseteq At(P)$ is an answer set for P iff M is a model of $LComp(P)$.

Proof. ($\Rightarrow$) Let M be an answer set of program P. It is an answer set of an instance of P, say P_1. The proof of the satisfaction of $Comp(P)$ is trivial. We show that M satisfies loop formulas of P. By lemma 9.1, we have $M \models Lcomp(P_1)$. Then $M = T_{P_1}^{\infty}(\emptyset, M)$, that is any atom in M can be derived without itself. Therefore, we have M models any loop formula of P.

($\Leftarrow$) Let P be a program and M be a set of atoms that $M \models LComp(P)$. Let P_1 be a instance of P. The proof of satisfaction of $Comp(P_1)$ is easy. Since any loop in the dependency graph of a instance of P is a loop in the dependency graph of P and M satisfies the loop formula of P, we have M satisfies the loop formula of P_1. Therefore M is an answer set of P_1. By Lemma 9.1, M is an answer set of P_1. We therefore have that M is an answer set of P. □

9.4 Relation to previous works

9.4.1 Local power set representation

Similar constructions or definitions have been presented in the literature for different purposes. In [64], *indexed pairs* are used to translate aggregate programs to normal logic programs; the *aggregate solutions* for aggregates is given in [76] to capture conditional satisfaction of aggregates; and In [77], the existence of a *level mapping*, w.r.t. a model, is formulated as a sufficient condition for the model to be an answer set. Local power sets are variants of *prefixed power sets* [71], which are used to define a generalized form of Gelfond-Liftchitz reduction.

Let A be a c-atom. It takes polynomial time, in the size of A, to construct A^*. A naive algorithm would examine each pair $X, Y \in A_c$ such that $X \subseteq Y$, in the partial order of *inclusiveness* – a pair (X, Y) *includes* another (X', Y') if $X \subseteq X'$ and $Y' \subseteq Y$, whereas at least one of the $\subseteq$ is proper, to see if for any I such that $X \subseteq I \subseteq Y$, $I \in A_c$. Whenever such a pair (X, Y) is identified, it is maximal, and thus all the included pairs are dropped from consideration.

For some special classes of c-atoms, the construction of A^* is much simpler. Below, we show this for the classes of monotone, antimonotone, and convex c-atoms.

Given a set S of sets, $\pi \in S$ is said to be *minimal* in S if there is no $\pi' \in S$ such that $\pi' \subset \pi$; similarly for maximal sets in S.

- A is monotone: $A_c^* = \{\langle B, A_d\rangle : \ B \text{ is minimal in } A_c\}$
- A is antimonotone: $A_c^* = \{\langle \emptyset, T\rangle : \ T \text{ is maximal in } A_c\}$
- A is convex: $A_c^* = \{\langle B, T\rangle : \ B \subseteq T, B \text{ is minimal}, T \text{ is maximal in } A_c\}$

9.4.2 Dependency graph

In [49], *positive dependency graph* is defined for programs with monotone c-atoms. Let P be a program with monotone c-atoms which consists of rules of the form (8.1). The *positive dependency graph* of P is the directed graph $G_P^m = (V, E)$, where $V = At(P)$ and $\langle u, v\rangle$ is an edge in E if there exists a rule $r \in P$ such that $u \in hd(r)_d$ and $v \in A_d$, for some positive c-atom $A \in bd(r)$.

We make two remarks. First, the construction of G_P^m is not directly applicable to programs with nonmonotone c-atoms. For these programs, a head atom in a rule may also *positively* depend on the atoms appearing in a *negative* body c-atom of the same rule. Second, there are in general more loops in G_P^m than in G_P^a.

Example 9.5. Consider a program with nonmonotone c-atoms, denoted P:

$$a \ \leftarrow \ \texttt{not}\ (\{a, b\}, \{\emptyset, \{b\}\}) \tag{9.3}$$

$$b \ \leftarrow \ \texttt{not}\ (\{a, b\}, \{\emptyset, \{a\}\}) \tag{9.4}$$

G_P^m contains no edges. So, it cannot be used to deny a model of completion to be an answer set. Indeed, $M = \{a\}$ is a model of $Comp(P)$, but not an answer set for P.

Our dependency graph is based on the program where negative c-atoms are replaced by their complements. In this case, we have the following program, denoted P'

$$a \ \leftarrow \ (\{a, b\}, \{\{a\}, \{a, b\}\}) \tag{9.5}$$

$$b \ \leftarrow \ (\{a, b\}, \{\{b\}, \{a, b\}\}) \tag{9.6}$$

where $L_1 = \{a\}$ and $L_2 = \{b\}$ are loops in $G_{P'}^a$, but $L_3 = \{a, b\}$ is not. Note that L_3 is also a loop in $G_{P'}^m$, in addition to L_1 and L_2.

The loop formula for L_1 in our case, for example, is $LP^a(L_1) = a \rightarrow false$. $M \not\models LP^a(L_1)$, which shows that M is not an answer set for P. □

Proposition 9.2. Let P be a constraint program. Any loop in G_P^a is a loop in G_P^m, but the converse does not hold.

Proof. Given a c-atom A, we have that $ASet(A) \subseteq Dom(A)$. Let P be a constraint program, by the definition of G^a_P and G^m_P, we have that any edge in G^a_P is an edge in G^m_P. Therefore any loop in G^a_P is a loop in G^m_P. The loop L_3 in Example 9.5 shows that a loop in G^m_P may not be a loop in G^m_P. □

For monotone-constraint programs P, the definition of G^a_P is consistent with that of G^m_P. By definition, the complement of a monotone c-atom is antimonotone. As shown earlier, if A is antimonotone, then A^* is such that for every $\langle B, T\rangle \in A^*_c$, we have $B = \emptyset$. It follows from the definition of G^a_P that a head atom does not depend on any body atoms in an antimonotone c-atom.

However, since the extra loops in G^m_P are non-essential (e.g. the loop L_3 in Example 9.5), our loop formulas also work with the definition of loops in G^m_P.

Theorem 9.3. Let P be a constraint program, and $LC(P) = Comp(P) \cup \{LP^a(L) \mid L \text{ is a loop in } G^m_P\}$. A set $M \subseteq At(P)$ is an answer set for P iff M is a model of $LC(P)$.

9.4.3 Loop formulas

The loop formulas in [49] are defined as follows. For a rule of the form

$$A \leftarrow A_1, \ldots, A_k, \texttt{not}\ A_{k+1}, \ldots, \texttt{not}\ A_m,$$

define

$$\beta_L(r) = A_{1|L} \wedge \ldots \wedge A_{k|L} \wedge \neg A_{k+1} \wedge \ldots \wedge \neg A_m$$

where $A_{|L} = (A_d \setminus L, \{Y : Y \in A_c, Y \cap L = \emptyset\})$.[1]

The loop formula for L, denoted $LP^m(L)$, is defined as

$$\bigvee L \rightarrow \bigvee \{\beta_L(r) : r \in P, L \cap hd(r)_d \neq \emptyset\}$$

For monotone-constraint programs (note that they do not contain negative c-atoms), our loop formula is equivalent to the one above. To show this result, we need the following lemma.

Lemma 9.2. Let A be a monotone c-atom and M a set of atoms. For any set of atoms L, $M \models A^*_{|L}$ iff $M \models A_{|L}$.

[1] The definition given in [49] is actually $A_{|L} = (A_d, \{Y \mid Y \in A_c, Y \cap L = \emptyset\})$, which didn't work completely. As an example, assume a monotone-constraint program $P = \{a \leftarrow (\{a, b\}, \{\emptyset, \{a\}, \{b\}, \{a, b\}\})\}$. $M = \{a\}$ is an answer set for P, but $L = \{a\}$ is a loop in G^m_P and its loop formula is $a \rightarrow (\{a, b\}, \{\emptyset, \{b\}\})$, which is not satisfied by M.

Proof. ($\Rightarrow$)

(1) $M \models A^*_{|L}$,

(2) There exists $\langle B, A_d\rangle \in A^*_c$ s.t. $L \cap B = \emptyset$ and $M \cap A_d \models B^\wedge$, due to (1).

(3) $A_{|L}$ is monotone, due to that A is monotone.

(4) $B \subseteq M$, due to (2).

(5) $B \subseteq (A_{|L})_d$, due to (2).

(6) $B \subseteq M \cap (A_{|L})_d$, due to (4) and (5).

(7) $B \in (A_{|L})_c$, due to (2).

(8) $M \cap (A_{|L})_d \in (A_{|L})_c$, due to (3) and (7).

(9) $M \models A_{|L}$, due to (8).

($\Leftarrow$)

(1) $M \cap (A_d \setminus L) \in (A_{|L})_c$.

(2) $\exists S$ s.t. $S \subseteq M$, $S \subseteq A_d$, $S \cap L = \emptyset$, $S \in A_c$, due to (1).

(3) Let B be the minimal set s.t. $B \subseteq S$.

(4) $M \models B^\wedge$, due to (3) and the monotonicity of A.

(5) $M \models A^*_{|L}$, duet to (4).

□

Theorem 9.4. Let P be a monotone-constraint program and $M \subseteq At(P)$. For any loop $L \in G^m_P$, $M \models LP^a(L)$ iff $M \models LP^m(L)$.

Proof. We consider the rule of form $A \leftarrow A_1$. The result can be extended to the rule with conjunctive body.

Let P be a program and L a loop in G^m_P. The case where L is not a loop in G^a_P is trivial. We consider that L is a loop in G^a_P. Suppose $M \models \bigvee L$. By Proposition 9.2, we have a one-to-one correspondence between the satisfaction of $\alpha_L(r)$ and $\beta_L(r)$ for the rule $r \in P$ and $L \cap hd(r)_d \neq \emptyset$. Thus, Theorem 9.4 holds. □

Corollary 9.1. Let P be a monotone-constraint program. For any loop $L \in G_P^m$, no atom in $LP^a(L)$ occurs negatively.

Thus, for any monotone-constraint program P (without negative c-atoms), for any loop L in G_P^m or in G_P^a, the size of $LP^a(L)$ is no larger than that of $LP^m(L)$. In many cases, the former is substantially smaller; in some cases, the former is exponentially smaller.

Example 9.6. Consider the following monotone program P:

$$c \leftarrow (\{a, b, c\}, \{\{a\}, \{a, b\}, \{a, c\}, \{a, b, c\}\}). \tag{9.7}$$

There is no loop in G_P^a, but one loop $L = \{c\}$ in G_P^m. Our loop formula for this loop is

$$LP^a(L) = c \rightarrow a \tag{9.8}$$

while the loop formula of $LP^m(L)$ is

$$LP^m(L) = c \rightarrow (\{a, b\}, \{\{a\}, \{a, b\}\}). \tag{9.9}$$

To scale up, replace the rule above with $a_n \leftarrow A$, where $A = (\{a_1, ..., a_n\}, \{\pi : \pi \subseteq A_d, a_1 \in \pi\})$. □

9.5 Application

In this section, we show that some c-atoms (aggregates and global constraints) can be encoded by pseudo-Boolean (PB) theories. The encodings presented in this section are different from that in Chapter 7. Firstly, the goal of the encodings in this section is to illustrate the possibility of computing answer sets by PB constraint solvers while the encodings in Chapter 7 is for the computation of answer sets by weight constraint program solvers. Secondly, the encodings in this section is more complex than that in Chapter 7, since we have to use *PB variables* to encode the truth and falsity of c-atoms, and enforce the equivalence between c-atoms and their PB encoding.

We contrast the encoding approach with the unfolding approaches in [21, 64]. The advantage is shown in Example 9.7.

9.5.1 Pseudo-Boolean constraints

PB constraints are integer programming constraints in which variables have 0-1 domains. They are generally written in the form of inequalities

$$w_1 \times x_1 + \ \ldots \ + w_n \times x_n \, \mathcal{R} \, w \tag{9.10}$$

where $\mathcal{R} \in \{\leq, \geq, <, >, =\}$, w_i and w are integer coefficients, and x_i are PB variables taking values from the domain $\{0, 1\}$.

A *PB theory* is a set of PB constraints. Let's denote the set of variables in a PB theory by X. An assignment $\lambda : X \rightarrow \{0, 1\}$ is a *solution* to a PB theory if it satisfies every constraint in the theory.

Given a program P, the scheme for the encoding of $Comp(P)$ by PB constraints is given in [49], where all c-atoms need to be encoded.

There is a straightforward encoding of any c-atom A in terms of a PB theory by enumerating all admissible solutions in A_c. We are interested in the cases where the resulting PB theories are linear in the size of a c-atom's domain.

Let λ be a solution to a PB theory and A a c-atom whose domain is $S = \{a_1, \ldots, a_n\}$. In to encode A, we will introduce a PB variable x_i for each a_i, and define the *membership set* of λ to be $M_\lambda = \{a_i \in S : \lambda(x_i) = 1\}$.

To represent the satisfaction of a c-atom A, we introduce a PB variable y. The intention is that an assignment λ is a solution with $y = 1$ (resp. $y = 0$) to the encoded PB theory if and only if the corresponding membership set satisfies (resp. does not satisfy) A.

Definition 9.7. Let A be a c-atom. A PB theory is the *PB encoding* (or *encoding*) of A, denoted by $\tau_{pb}(y \equiv A)$, iff for any solution λ to $\tau_{pb}(y \equiv A)$, we have $\lambda(y) = 1$ iff $M_\lambda \models A$.

9.5.2 Encoding of aggregates

The syntax of aggregates is presented in Chapter 5. For the simplicity in the presentation, we consider the aggregate of form

$$aggr(\{a \mid \ a \in S\}) \ \mathtt{op} \ Result \tag{9.11}$$

where S is the domain of the aggregate, in this section. The form 9.11 is equivalent to form 5.1 in Chapter 5, considering a and S as X and $p(X)$, respectively.

For the aggregates SUM, $COUNT$, AVG, MIN, $mathitMAX$ and their corresponding PB encodings that we will introduce, the following theorem holds (A detailed proof will be given for the aggregate MIN) .

Theorem 9.5. Let A be an aggregate whose domain is $S = \{a_1, ..., a_n\}$ and $\tau(y \equiv A)$ be the PB encoding of A. We have:

(I) There is a solution to $\tau(y \equiv A)$;

(II) Let λ be a solution to $\tau(y \equiv A)$. $\lambda(y) = 1$ if and only if $M_\lambda \models A$;

(III) The size of $\tau(y \equiv A)$ is linear in $|S|$.

SUM, COUNT, AVG

Let A be the aggregate $SUM(\{a|a \in S\}) \geq k$ and M be a subset of S. $M \models A$ if and only if the summation of the elements in M is not less than k. Let y be a PB variable. The PB encoding $\tau_{pb}(y \equiv A)$ consists of the following constraints.

$$x_i^+ + x_i^- = x_i, \ for\ each\ i,\ 1 \leq i \leq n \tag{9.12}$$

$$x_i^+ \cdot a_i \geq x_i \cdot a_i, \ for\ each\ i,\ 1 \leq i \leq n \tag{9.13}$$

$$x_i^- \cdot (-a_i) \geq x_i \cdot (-a_i), \ for\ each\ i,\ 1 \leq i \leq n \tag{9.14}$$

$$\sum_{i=1}^{n} (x_i \cdot a_i) \geq y \cdot k \tag{9.15}$$

$$\sum_{i=1}^{n} (x_i \cdot a_i) \leq (1-y) \cdot (k-1) + y \cdot \sum_{i=1}^{n} (x_i^+ \cdot a_i + x_i^- \cdot (-a_i)) \tag{9.16}$$

The PB variables x_i^+ and x_i^- are introduced for each x_i to encode $|a_i|$. The constraints (9.12) to (9.14) guarantees that $x_i^+ = x_i$ and $x_i^- = 0$, if $a_i > 0$; $x_i^- = x_i$ and $x_i^+ = 0$, if $a_i < 0$; and $x_i^+ = x_i$ or $x_i^- = x_i$, if $a_i = 0$. Therefore $x_i^+ \cdot a_i + x_i^- \cdot (-a_i) = |a_i|$.

It can be verified that $\tau_{pb}(y \equiv A)$ always have a model. Let λ be a model of $\tau_{pb}(y \equiv A)$. The membership set $M_\lambda \models A$ if and only if $\lambda(y) = 1$.

The aggregate $COUNT(\{a|a \in S\}) \geq k$ is the special case of $SUM(\{a|a \in S\}) \geq k$, where each a_i in S equals to 1. The encoding for $COUNT(\{a|a \in S\}) \geq k$ can be constructed simply by substituting 1 for each a_i in constraints (9.12) and (9.13).

The aggregate $AVG(\{a|a \in S\}) \geq k$ is essentially the aggregate $SUM(\{b|b \in S'\} \geq 0$, where S' is $\{a_1 - k, ..., a_n - k\}$.

The proofs of the Theorem 9.5 for above aggregates are straightforward.

To contrast with the unfolding approach, consider the following example.

Example 9.7. Let P be a program with a single rule:

$$h \leftarrow A \tag{9.17}$$

where $A = COUNT(\{a : a \in \{a_1, a_2, a_3, a_4\}) \geq 3$.

We introduce PB variables y and z for the aggregate A and atom h, respectively. The number of constraints in $\tau(y \equiv A)$ is linear in the size of $|S|$, according to the formulas from 9.12 to 9.16 (Note that each a_i is replaced by 1 here). The rule can be encoded by PB

constraint $z + (1 - y) = 1$. Thus, P can be encoded by a PB theory of size linear in the size of $|S|$.

The unfolded normal program of P would be:

$$h \leftarrow a_1, a_2, a_3 \tag{9.18}$$

$$h \leftarrow a_1, a_2, a_4 \tag{9.19}$$

$$h \leftarrow a_1, a_3, a_4 \tag{9.20}$$

$$h \leftarrow a_2, a_3, a_4. \tag{9.21}$$

For an aggregate $COUNT(\{a : a \in S\}) \geq k$, the number of rules in the unfolded normal program is $C^k_{|S|}$, which is in general exponential in $|S|$. □

MIN

Let A be the aggregate $MIN(\{a|a \in S\}) \geq k$ and M be a subset of S. $M \models A$ if and only if the minimal element in M is greater than or equal to k.

Let y be a PB variable. In addition to x_i, we introduce PB variables x_i^+ and x_i^- for each atom a_i. The PB encoding $\tau_{pb}(y \equiv A)$ consists of the following constraints.

$$x_i^+ + x_i^- = x_i, for\ each\ i,\ 1 \leq i \leq n \tag{9.22}$$

$$x_i^+ \cdot (a_i - k) \geq (x_i - 1) \cdot (a_i - k),\ for\ each\ i,\ 1 \leq i \leq n \tag{9.23}$$

$$x_i^- \cdot (k - a_i) \geq (x_i - 1) \cdot (k - a_i),\ for\ each\ i,\ 1 \leq i \leq n \tag{9.24}$$

$$T_2 - T_1 \leq 2 \cdot (1 - y) \cdot \sum_{i=1}^{n} (a_i - k) \tag{9.25}$$

$$s_1 + s_2 + y = 1 \tag{9.26}$$

$$T_2 - T_1 \geq s_1 \tag{9.27}$$

$$\sum_{i=1}^{n} x_i \leq (1 - s_2) \cdot n \tag{9.28}$$

$$\sum_{i=1}^{n} x_i \geq 1 - s_2 \tag{9.29}$$

where T_1 denotes $\sum_{i=1}^{n} x_i \cdot (a_i - k)$ and T_2 denotes $\sum_{i=1}^{n} (x_i^+ \cdot (a_i - k) + x_i^-(k - a_i))$.

The idea in the encoding of A is that for a set of numbers $S = \{a_1, ..., a_n\}$, the minimal number in S is greater than or equal to k if and only if

$$\sum_{i=1}^{n} (a_i - k) = \sum_{i=1}^{n} |a_i - k|. \tag{9.30}$$

Similar to the encoding of SUM, the first three constraints encode $|a_i - k|$. T_1 and T_2 are, respectively, the summation of $a_i - k$ and $|a_i - k|$ for a_i's whose corresponding PB variable x_i is assigned to be 1. The constrain (9.25) ensure that formula (9.30) holds, if $y = 1$. Formula (9.30) is not satisfied by a set, if $y = 0$. The constraints (9.26) and (9.27) encodes the case where the set is empty; the constraints (9.28) and (9.29) encodes the case where the set is not empty.

We give the proof of Theorem 9.5 for aggregate $MIN(\{a|a \in S\}) \geq k$. For convenience, we re-state the theorem for the aggregate $MIN(\{a|a \in S\}) \geq k$.

Theorem 9.6. Let A be the aggregate $MIN(\{a|a \in S\}) \geq k$ whose domain is $S = \{a_1, ..., a_n\}$ and $\tau(y \equiv A)$ be the PB encoding of A which consists of the PB constraints from (9.22) to (9.29). Then, we have:

(I) There is a solution to $\tau(y \equiv A)$.

(II) Let λ be a solution to $\tau(y \equiv A)$. $\lambda(y) = 1$ if and only if $M_\lambda \models A$.

(III) The size of $\tau(y \equiv A)$ is linear in $|S|$.

Proof.

- The proof of (I).

  If $\exists i$, $a_i \geq k$, say $a_1 \geq k$. The assignment λ, where $\lambda(x_1) = \lambda(x_1^+) = 1$, $\lambda(x_1^-) = 0$, $\lambda(x_i) = \lambda(x_i^+) = \lambda(x_i^-) = 0$ for all $i \neq 1$, $\lambda(y) = 1$, and $\lambda(s_1) = \lambda(s_3) = 0$ is a solution.

  If $\forall i$, $a_i < k$. The assignment λ, where $\lambda(x_i) = \lambda(x_i^+) = \lambda(x_i^-) = 0$ for all $1 \leq i \leq n$, $\lambda(y) = \lambda(s_1) = 0$ and $\lambda(s_2) = 1$ is a solution.

- The proof of (II).

  We show that, for any solution λ to $\tau(y \equiv A)$, if $\lambda(y) = 1$ then $M_\lambda \models A$.

  (1) Let λ be a solution to $\tau(y \equiv A)$ where $\lambda(y) = 1$.

  (2) $\lambda(s_1) = \lambda(s_2) = 0$, due to (9.26).

  (3) $T_2 - T_1 \geq 0$, due to (2) and (9.27).

  (4) $T_2 - T_1 \leq 0$, due to (1) and (9.25).

  (5) $T_2 = T_1$, due to (3) and (4).

  (6) $\sum_{i=1}^{n} x_i \geq 1$, due to (2) and (9.29).

(7) $M_\lambda \models A$, due to (5) and (6).

We show that, for any solution λ to $\tau(y \equiv A)$, if $\lambda(y) = 0$ then $min(M_\lambda) < k$.

(1) Let λ be a solution to $\tau(y \equiv A)$ where $\lambda(y) = 0$.

(2) There are two cases: (i). $\lambda(s_1) = 0$ and $\lambda(s_2) = 1$ and (ii). $\lambda(s_1) = 1$ and $\lambda(s_2) = 0$, due to (9.26).

(3) $\lambda(s_1) = 0$ and $\lambda(s_2) = 1$, which is the Case (i).

(4) $\sum_{i=1}^{n} x_i = 0$, due to (9.28) and (9.29).

(5) $M = \emptyset$ and $M \not\models A$, due to (4).

(6) $\lambda(s_1) = 1$ and $\lambda(s_2) = 0$, which is the case (ii).

(7) $T_2 > T_1$, due to (9.27).

(8) $\sum_{i=1}^{n} x_i \geq 1$, due to (9.29).

(9) $M_\lambda \not\models A$, due to (7) and (8).

- (III) holds obviously by the encoding. □

Example 9.8. Let A be the aggregate $MIN(\{a|a \in \{2,4\}) \geq 3$ and y be a PB variable. $\tau_{pb}(y \equiv A)$ contains the following constraints.

$$\begin{array}{lll} x_1^+ + x_1^- = x_1, & x_1^+ \leq x_1, & x_1^- \geq x_1, \\ x_2^+ + x_2^- = x_2, & x_2^+ \geq x_2, & x_2^- \leq x_2, \\ T_2 - T_1 \leq 12 \cdot (1-y), & s_1 + s_2 + y = 1, & T_2 - T_1 \geq s_1, \\ x_1 + x_2 \leq 2 \cdot (1 - s_2), & x_1 + x_2 \geq 1 - s_2. & \end{array}$$

where $T_1 = -x_1 + x_2$ and $T_2 = -x_1^+ + x_1^- + x_2^+ - x_2^-$. We use a 9-ary 0-1 tuple to represent the value assignment to the tuple $(x_1, x_1^+, x_1^-, x_2, x_2^+, x_2^-, y, s_1, s_2)$. It can be verified that $\tau_{pb}(y \equiv A)$ has four solutions. $\lambda_1 : (0,0,0,0,0,0,0,0,1)$, $\lambda_2 : (0,0,0,1,1,0,1,0,0)$, $\lambda_3 : (1,0,1,0,0,0,0,1,0)$, and $\lambda_4 : (1,0,1,1,1,0,0,1,0)$. Among the four solutions, only $\lambda_2(y) = 1$. Therefore, the only set that satisfies A is $M_{\lambda_2} = \{4\}$. □

MAX

Let A be the aggregate $mathitMAX(\{a|a \in S\}) \geq k$ and M be a subset of S. $M \models A$ if and only if the maximal element in M is greater than or equal to k.

Let y be a PB variable. $\tau_{pb}(y \equiv A)$ consists of the following constraints.

$$
\begin{aligned}
x_i^+ + x_i^- &= x_i, for\ each\ i,\ 1 \leq i \leq n && (9.31)\\
x_i^+ \cdot (a_i - k) &\geq (x_i - 1) \cdot (a_i - k),\ for\ each\ i,\ 1 \leq i \leq n && (9.32)\\
x_i^- \cdot (k - a_i) &\geq (x_i - 1) \cdot (k - a_i),\ for\ each\ i,\ 1 \leq i \leq n && (9.33)\\
T_1 + T_2 &\geq y && (9.34)\\
s_1 + s_2 + y &= 1 && (9.35)\\
T_1 + T_2 &\leq 2 \cdot (1 - s_1) \cdot \sum_{i=1}^{n} && (9.36)\\
\sum_{i=1}^{n} x_i &\leq (1 - s_2) \cdot n && (9.37)\\
\sum_{i=1}^{n} x_i &\geq 1 - s_2 && (9.38)
\end{aligned}
$$

where T_1 denotes $\sum_{i=1}^{n} x_i \cdot (a_i - k + 1)$ and T_2 denotes $\sum_{i=1}^{n}(x_i^+ \cdot (a_i - k + 1) + x_i^-(k - 1 - a_i))$.

The idea in the encoding of A is that for a set of numbers $S = \{a_1, ..., a_n\}$, the maximal number in S is greater than or equal to k if and only if

$$\sum_{i=1}^{n} (a_i - k + 1) > -\sum_{i=1}^{n} |a_i - k + 1|. \tag{9.39}$$

The proof of Theorem 9.5 for $mathitMAX$ is similar to that for MIN.

9.5.3 Encoding of global constraints

A global constraint is a constraint that specifies a relation between a set of variables. One of the most extensively studied global constraints is `alldifferent` [81], which specifies that each variable in the variable set X must take a value from the domain D and different variables must take distinct values.

Let $X = \{x_1, \ldots, x_m\}$ be a set of variables and $D = \{d_1, \ldots, d_n\}$ be the domain of the variables. We use atom $a(x_i, d_j)$ to represent that variable x_i is assigned the value d_j. The global constraint `alldifferent` (X) can be represented by a c-atom $AllDiff = (S, C)$,

where the domain of the constraint is $S = \{a(x,d) : x \in X, d \in D\}$, and C is the collection of sets satisfying, for each set π in C, (1) for each i, exactly one $a(x_i, d_j)$ is in π, for some d_j, and (2) if $a(x,d)$ and $a(x',d')$ are in π and $x \neq x'$, then $d \neq d'$.

Let G = `alldifferent` (S, C) and y be a PB variable. We introduce the following additional PB variables.

- x_{ij} for each atom $a(x_i, d_j)$.
- s_{i0}, s_{i1}, and s_{i2} for each variable x_i. Intuitively, $s_{i0} = 1$ indicates that x_i is not assigned to any value; $s_{i1} = 1$ indicates that x_i is assigned to exactly one value; and $s_{i2} = 1$ indicates that x_i is assigned to at least two values.
- t_j for each value d_j. Intuitively, if $t_j = 1$ then d_j is not assigned to any variable (not used) or assigned to one variable; if $t_j = 0$ then d_j is assigned to at least two variables.

The PB theory $\tau_{pb}(y \equiv G)$ consists of the following PB constraints.

$$s_{i0} + s_{i1} + s_{i2} = 1, \quad for\ each\ i,\ 1 \leq i \leq m \tag{9.40}$$

$$\sum_{j=1}^{n} x_{ij} \leq (1 - s_{i0}) \cdot n, \quad for\ each\ i,\ 1 \leq i \leq m \tag{9.41}$$

$$\sum_{j=1}^{n} x_{ij} \geq 2 \cdot s_{i2}, \quad for\ each\ i,\ 1 \leq i \leq m \tag{9.42}$$

$$\sum_{j=1}^{n} x_{ij} \leq n \cdot (1 - s_{i1}) + s_{i1}, \quad for\ each\ i,\ 1 \leq i \leq m \tag{9.43}$$

$$\sum_{j=1}^{n} x_{ij} \geq s_{i1}, \quad for\ each\ i,\ 1 \leq i \leq m \tag{9.44}$$

$$\sum_{i=1}^{m} x_{ij} \leq m \cdot (1 - t_j) + t_j, \quad for\ each\ j,\ 1 \leq j \leq n \tag{9.45}$$

$$\sum_{i=1}^{m} x_{ij} \geq 2 \cdot (1 - t_j), \quad for\ each\ j,\ 1 \leq j \leq n \tag{9.46}$$

$$\sum_{i=1}^{m} s_{i1} + \sum_{j=1}^{n} t_j \geq (m + n) \cdot y \tag{9.47}$$

$$\sum_{i=1}^{m} s_{i1} + \sum_{j=1}^{n} t_j \leq (m + n - 1) \cdot (1 - y) + (m + n) \cdot y \tag{9.48}$$

Theorem 9.7. Let $G = \mathit{AllDiff}(S, C)$ and y be a PB variable. The following statements hold for the encoding $\tau_{pb}(y \equiv G)$.

(I) The PB theory $\tau_{pb}(y \equiv G)$ has at least one model.

(II) Let λ be a model of $\tau_{pb}(y \equiv G)$. The membership set $M_\lambda \models A$ iff $\lambda(y) = 1$.

(III) The size of $\tau_{pb}(y \equiv G)$ is linear in $|S|$.

Proof.

- The proof of (I).

  If $m > n$, we construct an assignment λ, where $\lambda(x_{ij}) = 0$ for all $1 \leq i \leq m$ and $1 \leq j \leq n$; $\lambda(s_{i0}) = 1$, $\lambda(s_{i1}) = \lambda(s_{i2}) = 0$ for all $1 \leq i \leq m$; $\lambda(y) = 0$; and $\lambda(t_j) = 1$ for all $1 \leq j \leq n$. It can be verified that λ is a solution to $\tau(y \equiv G)$.

  If $m > n$, we construct an assignment λ, where $\lambda(x_{ij}) = 1$ for all $i = j$; $\lambda(x_{ij}) = 0$ for all $i \neq j$; $\lambda(s_{i0}) = 1$, $\lambda(s_{i1} = 1) = 1$, and $\lambda(s_{i2}) = 0$ for all $1 \leq i \leq m$; $\lambda(y) = 1$; $\lambda(t_j) = 1$ for all $1 \leq j \leq n$. It can be verified that λ is a solution to $\tau(y \equiv G)$.

- The proof of (II).

  Let λ be a solution to $\tau_{pb}(y \equiv G)$. If $\lambda(y) = 1$, then $\forall i.\ \lambda(s_{i1}) = 1$ and $\forall j.\ \lambda(t_j) = 1$, due to (9.47). Then each variable is assigned to exactly one value, due to (9.43) and (9.44), and each value is assigned to no more than one variable due to (9.45).

  If $\lambda(y) = 0$, then $\exists i.\ \lambda(s_{i1}) = 0$ or $\exists j.\ \lambda(t_j) = 0$, due to (9.48). If $\lambda(s_{i1}) = 0$, we have $\lambda(s_{i0}) = 1$ or $\lambda(s_{i2}) = 1$ due to (9.40). If $\lambda(s_{i0}) = 1$, then x_i is not assigned due to (9.41). If $\lambda(s_{i2}) = 1$, then x_i is assigned to more than one value due to (9.42). If $\lambda(t_j) = 0$, we have d_j is assigned to more than one variable, due to (9.46).

- Obviously, (III) holds by the encoding. □

Note that the membership set of λ in this case is: $M_\lambda = \{a(x_i, d_j) \in S : \lambda(x_{ij}) = 1\}$.

Clearly, the unfolding approach would be undesirable, as in the unfolded normal program it would have to list all solutions in C for the constraint $\mathit{AllDiff}(S, C)$, which is exponential in $|S|$.

The all-different constraint has many applications, for instance, to represent the pigeonhole problem. A typical encoding of this problem by a normal program [61] takes exponential time to decide unsatisfiability by an ASP solver. When the problem is encoded as a PB theory, unsatisfiability can be decided in polynomial time by a PB constraint solver [11]. Efficiency can be further improved by embedding special constraint propagators [50].

9.6 Conclusion

This chapter extends the work in [49] in two aspects. Firstly, we extend the loop formula characterization of answer sets for monotone constraint programs to arbitrary constraint programs. We show that the loop formulas in our approach are simpler than that in [49]. Secondly, we propose the PB encodings for some of the frequently used constraints: aggregates and global constraints. Our results provide the means to compute arbitrary constraint programs by PB constraint solvers, comparing to the approach in [49], which focus on monotone constraint programs.

Chapter 10

Strong equivalence of abstract constraint programs

10.1 Motivation

A logic program P_1 is said to be *equivalent* to a logic program P_2 under answer set semantics, if P_1 and P_2 have the same answer sets. P_1 and P_2 are *strongly equivalent* if for any program P, $P_1 \cup P$ and $P_2 \cup P$ are equivalent. Questions regarding whether two logic programs are strongly equivalent are interesting for a variety of reasons. For instance, in order to see whether a set of rules in a program can always be replaced by another one regardless of the other rules of the program, one should check whether the two sets of rules are strongly equivalent. For example, the set of a single rule $\{p \leftarrow p\}$ is strongly equivalent to the empty set (of rules), therefore it can always be removed from any logic program. However, the set P: $\{p \leftarrow q, q \leftarrow p\}$ is equivalent but not strongly equivalent to empty set, so the pair of rules cannot be eliminated regardless of the context: in the presence of q, the first rule of program P can be used to derive p.

Lifschitz et. al. [40] propose to use the logic of here-and-there (HT-logic) to characterize the strong equivalence for normal logic programs. The problem of determining the strong equivalence of two logic programs can be done by checking if the two programs have the same set of models in HT-logic.

Turner [78] proposes a model-theoretical method to characterize the strong equivalence between programs. Two programs are strongly equivalent if and only if they have the same set of HT-models. For a program, HT-models are exactly models of the program in HT-logic. So, Turner's approach is essentially the same as the approach of [40].

Lin [41] presents a transformation from logic programs to propositional theories. By the transformation, the strong equivalence of logic programs is converted to the entailment

of theories in classic logic.

For logic programs with constraints, Turner [79] shows that SE-models (which is essentially the same as the HT-models proposed in [78]) can be used to characterize the strong equivalence between weight constraint programs. Liu and Truszczyński extend the approach in [49] to monotone constraint programs. They show that two monotone constraint programs are strongly equivalent if and only if they have the same set of SE-models.

In this chapter, we further extend Turner's approach to arbitrary constraint programs. We present two characterizations of strong equivalence of arbitrary constraint programs under answer set semantics.

In Section 10.2, we define SE-models for abstract constraint programs and then use them to characterize strong equivalence. We give another characterization of strong equivalence in Section 10.3, which is simpler than the previous one. The characterization is based on the generalization of the concept of program reduct to abstract constraint programs. Section 10.4 concludes the work.

10.2 Characterizations of SE under answer set semantics

In this chapter, we consider the logic programs with c-atoms where each rule is of the form

$$a \leftarrow A_1, ..., A_n \tag{10.1}$$

where a is an atom and A_1,...A_n are c-atoms. For the programs whose head is an c-atom, the strong equivalence of two programs can be reduced to the strong equivalence of the instances of the two programs.

Definition 10.1. Let P and Q be two programs. P *strongly equivalent* to Q, denoted $P \equiv_s Q$, if for any program R, $AS(P \cup R) = AS(Q \cup R)$, where $AS(P)$ denote the set of answer sets of the program P.

We define the SE-models of a program.

Definition 10.2. Let P be a program. A pair of sets (X, Y) is a *SE-model* of P if the following conditions hold: (1) $X \subseteq Y$; (2) $Y \models P$; (3) $T_P^\infty(\emptyset, Y) \subseteq X$, where T_P is the operator based on conditional satisfaction given in Definition 8.2.

Lemma 10.1. Let P be a program and Y be an answer set of P. (Y, Y) is the unique SE-model of P whose second component is Y.

Proof. Note that if Y is a answer set of P, then (Y, Y) is an SE-model of P.

(1) Let Y be an answer set of P.

(2) $T_P^\infty(\emptyset, Y) = Y$, due to (1).

(3) Let (X, Y) be an SE-model of P.

(4) $Y \subseteq X$, due to (2) and (3).

(5) $X \subseteq Y$, due to (3).

(6) $X = Y$, due to (4) and (5).

□

Notation: Let P be a program. $SE(P)$ denotes the set of SE-models of P.

Lemma 10.2. Let P and Q be two programs. If $SE(P) = SE(Q)$, then $AS(P) = AS(Q)$.

Proof. We prove $AS(P) \subseteq AS(Q)$. $AS(Q) \subseteq AS(P)$ can be proved similarly.

(1) $SE(P) = SE(Q)$.

(2) $\forall Y \in AS(P)$, (Y, Y) is the unique SE-model of P whose second component is Y, due to Lemma 10.1.

(3) (Y, Y) is the unique SE-model of Q, whose second component is Y, due to (1) and (2).

(4) $(T_Q^\infty(\emptyset, Y), Y) \in SE(Q)$, due to (3), Definition 10.2, and Proposition 8.1.

(5) $T_Q^\infty(\emptyset, Y) = Y$, due to (3) and (4).

(6) $Y \in AS(Q)$, due to (5).

□

Lemma 10.3. Let P and Q be two programs. $SE(P \cup Q) = SE(P) \cap SE(Q)$.

Proof. We proof $\Rightarrow$ part. The other part is similar.

(1) $\forall (X, Y) \in SE(P \cup Q)$, we have $X \subseteq Y$, $Y \models P \cup Q$ and $T_{P \cup Q}^\infty(\emptyset, Y) \subseteq X$.

(2) $Y \models P$, $Y \models Q$, $T_P^\infty(\emptyset, Y) \subseteq X$, and $T_Q^\infty(\emptyset, Y) \subseteq X$, due to (1).

(3) $(X, Y) \in SE(P)$ and $(X, Y) \in SE(Q)$.

□

Theorem 10.1. Let P and Q be two programs. $P \equiv_s Q$ iff $SE(P) = SE(Q)$.

Proof. ($\Leftarrow$)

(1) $SE(P) = SE(Q)$.

(2) $\forall R$, $SE(P \cup R) = SE(Q \cup R)$, due to (1) and Lemma 10.3.

(3) $AS(P \cup R) = AS(Q \cup R)$, due to (2) and Lemma 10.2.

(4) $P \equiv_s Q$.

($\Rightarrow$) We prove $SE(P) \subseteq SE(Q)$. $SE(Q) \subseteq SE(P)$ can be proved similarly. Let (X, Y) be an SE-model of P but not an SE-model of Q. We consider two cases.

Case 1. $Y \not\models Q$.

(1) $(X, Y) \in SE(P)$.

(2) Let R be the program: $\{a \mid a \in Y\}$.

(3) $Y = T^{\infty}_{P \cup R}(\emptyset, Y)$, i.e., $Y \in ST(P \cup R)$, due to (1) and (2).

(4) $Y \not\models Q$.

(5) $Y \notin ST(Q \cup R)$, due to (4).

(6) A contradiction to $P \equiv_s Q$, due to (3) and (5).

(7) $Y \models Q$, due to (6).

Case 2. $T^{\infty}_{Q}(\emptyset, Y) \not\subseteq X$.

(1) Let $X' = T^{\infty}_{Q}(\emptyset, Y) \setminus X$.

(2) Let R be the program: $\{b \leftarrow a \mid a \in X' \; and \; b \in Y\}$, where $\langle X', Y\rangle = \{S \mid X' \subseteq S \subseteq Y\}$.

(3) $(X, Y) \in SE(P)$.

(4) $X = T^{\infty}_{P \cup R}(\emptyset, Y)$, due to (2) and (3).

(5) $T^{\infty}_{Q}(\emptyset, Y) \not\subseteq X$

(6) $X \neq Y$, due to (5).

(7) $Y \notin ST(P \cup R)$, due to (4) and (6).

(8) $Y \in ST(Q \cup R)$, due to (1) and (2).

(9) A contradiction to $P \equiv_s Q$, due to (7) and (8).

(10) $T_Q^\infty(\emptyset, Y) \subseteq X$

By case 1 and 2, we have $(X, Y) \in SE(Q)$. □

10.3 Yet another characterization

The SE-model defined in last section is based on the operator T_P. To check if a pair of sets (X, Y) is an SE-model of a program, a derivation process using T_P is needed. In this section, we will give a definition of SE-model that is based on the *reduct* of abstract constraint programs. By the new definition, the derivable process is not necessary for the verification of SE-models.

10.3.1 Reduct of abstract constraint programs and operator $\hat{T}_P$

Definition 10.3. Let A be a c-atom and M be an set of atoms. The *reduct* of A, w.r.t. M denoted A^M, is the c-atom (A_d^M, A_c^M), where $A_d^M = A_d$ and $A_c^M = \{s \mid s \in A_c, s \models_M A\}$. Note that if $A_c^M = \emptyset$, A^M is simply written as $\bot$.

Definition 10.4. Let r be a rule of the form (10.1) and M be a set of atoms. The *reduct* of r w.r.t. M, denoted r^M, is a rule of form $a \leftarrow A_1^M, ..., A_n^M$, where A_i^M is the reduct of A_i^M.

Definition 10.5. Let P be a program consists of rules of the form 10.1 and M be a set of atoms. The *reduct* of P, w.r.t. M denoted by P^M is a program consists of the reducts of the rules in P, whose body does not contain $\bot$.

Definition 10.6. Let P be a program with c-atoms and S a set of atoms. We define the operator $\hat{T}_P$.

$$\hat{T}_P(S) = \{a \mid \exists r \in P,\ s.\,t.\ a = hd(r)\ and\ S \models bd(r)\}. \tag{10.2}$$

By Definition 10.6, it is easy to show the following propositions.

Proposition 10.1. Let P be a program with c-atoms and M a model of P. Let R and S be sets such that $R \subseteq S \subseteq M$. Then $\hat{T}_{P^M}(R) \subseteq \hat{T}_{P^M}(S) \subseteq M$.

Proof. We consider the rule of the form $a \leftarrow A$, where A is a c-atom. The proof can be extended to rules with conjunctive body.

$\forall a \in \hat{T}_{P^M}(R)$, there exists a rule r^M of the form $a \leftarrow A^M$ in M, s.t. $R \models A^M$. Then there is a rule r of the form $a \leftarrow A$ in P, s.t. $R \models_M A$. Since $R \subseteq S$, we have $S \models_M A$. Thus $S \models A^M$. So $a \in \hat{T}_{P^M}(S)$.

The proof of $\hat{T}_{P^M}(S) \subseteq M$ is trivial. □

Similarly to the operator T_P, we define the sequence

$$\hat{T}_P^0(\emptyset) = \emptyset, \qquad \hat{T}_P^k(\emptyset) = \hat{T}_P(\hat{T}_P^{k-1}(\emptyset)).$$

Theorem 10.2. Let P be a logic program with c-atoms and M a model of P. The operator T_P and $\hat{T}_P$ are defined in Definition 8.2 and Definition 10.6, respectively. We have $T_P^\infty(\emptyset, M) = \hat{T}_{P^M}^\infty(\emptyset)$.

Proof. We consider the rule of the form $a \leftarrow A$, where A is a c-atom. The proof can be extended to rules with conjunctive body.

We show that for any k, $T_P^k(\emptyset, M) \subseteq \hat{T}_{P^M}^k(\emptyset)$ by induction. Base case holds since $T_P^0(\emptyset, M) = \hat{T}_{P^M}^0(\emptyset) = \emptyset$. The inductive hypothesis is for any $i \leq k-1$ $T_P^i(\emptyset, M) \subseteq \hat{T}_{P^M}^i(\emptyset)$. $\forall a \in T_P^k(\emptyset M)$, there is a rule $r: \ a \leftarrow A$ in P such that $T_P^{k-1}(\emptyset, M) \models_M A$. Thus $a \in \hat{T}_{P^M}(T_P^{k-1}(\emptyset, M))$. By the inductive hypothesis and Proposition 10.1, we have $T_P^k(\emptyset, M) \subseteq \hat{T}_{P^M}^k(\emptyset)$.

The statement that for any k, $\hat{T}_{P^M}^k(\emptyset) \subseteq T_P^k(\emptyset, M)$ can be proved in the same way. □

10.3.2 SE-models and strong equivalence

We re-define the SE-models of a program as follows.

Definition 10.7. Let P be a program. A pair of sets (X, Y) is a *SE-model* of P if the following conditions hold: (1) $X \subseteq Y$; (2) $Y \models P$; (3) $X \models P^Y$.

The only difference between Definition 10.2 and Definition 10.7 is that the third condition is re-defined using the reduct of program P. By Definitions 10.2, 10.7, and the Theorem 10.2 the following theorem holds.

Theorem 10.3. Let P and Q be two programs. $P \equiv_s Q$ iff $SE(P) = SE(Q)$.

10.4 Conclusion

The work in this chapter is an extension of the previous work on strong equivalence of normal programs [78] and monotone constraint programs [49]. We present two characterizations of strong equivalence of abstract constraint programs, using SE-models. In one characterization, SE-models are defined based on the operator T_P and in the other, the generalized reduct of logic programs. Given a pair of sets (X, Y), the first characterization needs a derivation process to check if (X, Y) is a SE-model and the second does not. Therefore, the second characterization is simpler than the first.

Chapter 11

Future work: Integrating global constraints to ASP

11.1 Introduction

Global constraints [57] have been developed by researchers in the field of constraint satisfaction problems (CSP). A global constraint is a constraint on a non-fixed number of variables which specifies the values that these variables can be assigned simultaneously. For example, the global constraint `alldifferent` $(x_1, ..., x_n)$ specifies that the values assigned to the variables $x_1, ..., x_n$ must be pairwise distinct.

Compared to the simple constraints like $=$, $\neq$, $<$, global constraints are more expressive since it is more convenient to define one constraint corresponding to a set of constraints than to define independently each constraint of this set. Consider the constraint `alldifferent` $(x_1, ..., x_n)$. It is obviously simpler and clearer than a set of constraints $\{x_1 \neq x_2$, ..., $x_{n-1} \neq x_n\}$.

In addition to expressiveness, global constraints are also advantageous in computation. Since a global constraint corresponds to a set of constraints, it is possible to deduce some information from the simultaneous presence of constraints. Thus powerful filtering algorithms can be designed by taking into account the set of constraints as a whole. By filtering algorithm, we mean an algorithm that removes from the domain of the variables the values that cannot take part in any solutions of a constraint. Suppose we have a CSP with 3 variables x_1, x_2, and x_3 and the sets of values that they can take are $\{a, b\}$, $\{a, b\}$ and $\{a, b, c\}$, respectively. Consider the constraint `alldifferent` (x_1, x_2, x_3). The filtering algorithm of `alldifferent` removes the values a and b from the domain of x_3, while the filtering algorithm (for establishing arc-consistency [17]) for the simple constraints $\{x_1 \neq x_2,\ x_1 \neq x_3,\ x_2 \neq x_3\}$ cannot delete any values.

In the practice of ASP, global constraints are employed for problem solving in a very limited way. ASP systems are usually less efficient to solve the problems that can be naturally modeled by global constraints. Let us use the well-known pigeon-hole benchmark problem as an example.

Example 11.1. The ASP encoding is as below.

$$1\{pos(P,H) : hole(H)\}1 \leftarrow pigeon(P). \tag{11.1}$$

$$\{pos(P,H) : pigeon(P)\}1 \leftarrow hole(H). \tag{11.2}$$

The first rule specifies that each pigeon has to be placed in exactly one hole. The second rule states that each hole can only be occupied by at most one pigeon.

Consider the case where the number of pigeons is n and the number of holes is $n-1$. After the first pigeon chooses a hole, the second one may take a hole from the remaining $n-1$ holes, then the third from the $n-2$ holes, ... The DPLL search must go through all of the $n \times n-1 \times ... \times 2$ assignments of holes to pigeons before it finds out there is no solution.

Using the global constraint `alldifferent` , the problem can be encoded by only one constraint specifying that each pigeon takes one and only one hole. The filtering algorithm specifically designed for `alldifferent` [68] can find out there is no solution in polynomial time. □

The effectiveness of global constraints for problem solving motivates us to incorporate them in ASP. Thanks to the formalism of logic programs with abstract constraints [53, 77], the theoretical foundation to integrate ASP with global constraints is ready. Essentially, constraints of all sorts, whenever they can be specified by a domain and a set of admissible solutions, can be embedded into a logic program.

This chapter is a preliminary study on the integration of global constraints to ASP. In the next section, we provide motivating experiments that compare the efficiency of global constraints to that of ASP programs in problem solving. In Section 11.3, we give a property of global constraints which may be useful in the further study on the computations and properties of logic programs with global constraints.

11.2 Motivating experiments

11.2.1 The goal

In the study of constraint programming, many filtering algorithms for global constraints have been implemented. A well-known constraint programming system with these implementations is the `SICSTUS` PROLOG (or simply, `SICSTUS`) system. It is a PROLOG programming system featured with a library *constraint logic programming on finite domains (CLP(fd))*, where a number of global constraints and their corresponding filtering algorithms are ready to use.

The goal of this section is to show the efficiency of the CLP(fd) in `SICSTUS` as a promising candidate for the computation of global constraints. We experiment on three widely used global constraints `alldifferent` , `cumulative` , and `sort` . For each global constraint, we compare the performance of the `SICSTUS` encoding and the ASP encoding on randomly generated instances. The CLP(fd) encodings are run by the system `SICSTUS` and the ASP encodings are run by the state-of-the-art ASP solver `CLASP` .

All of the experiments are run on Scientific Linux release 5.1 with 3GHz CPU and 1GB RAM. The cutoff time is set to 600 seconds. The instances that are solved in the cutoff time are called “solvable”, otherwise “unsolvable”. The results are shown in Figures 11.1, 11.2, and 11.3. In the figures, the running time of the unsolvable instances are plotted as 600 seconds. A summary of the experimental results is reported in Table 11.1, where the “Execution Time “ is the average running time in seconds for the solvable instances.

11.2.2 The `alldifferent` constraint

The `alldifferent` (or `all_disticnt`) constraint is probably the most studied global constraint in constraint programming. It constrains the values taken by a set of variables to be pairwise different. This constraint is used in a lot of real world problems like rostering or resource allocation. It is quite useful to express that two things cannot be at the same place at the same moment. An efficient filtering algorithm is proposed in [68].

Definition 11.1. [69] Let $X = \{x_1, x_2, ..., x_n\}$ be a set of variables and D be the domain of the variables, respectively. The `alldifferent` constraint is (X, D, C), where

$$C = \{(d_1, ..., d_n) \mid \forall\, i\ d_i \in D,\ \forall\, i \neq j\ d_i \neq d_j\}. \tag{11.3}$$

The problem

The typical benchmark for the application of the `alldifferent` constraint is the pigeon-hole problem. It is to place m pigeons into n holes satisfying that each pigeon has a hole and one hole is occupied by more than one pigeon.

`SICSTUS` encoding

In `SICSTUS` , the `all_disticnt` constraint is described as of the form `all_disticnt` $(+Variables)$, where $Variables$ is a list of domain variables. Each variable is constrained to take a value that is unique among the variables.

Let the set of pigeons are $\{1, ..., m\}$ and the set of holes $\{1, ..., n\}$. We model the pigeon-hole problem by the following `SICSTUS` program.

$$place([P_1, ..., P_m], n) \quad :- \tag{11.4}$$

$$domain([P_1, ..., P_m], 1, n), \tag{11.5}$$

$$\texttt{all\_disticnt}\ ([P_1, ..., P_m]), \tag{11.6}$$

$$labeling([], [P_1, ..., P_m]). \tag{11.7}$$

The formula (11.5) specifies the possible assignment of hole to pigeons. The formula (11.6) constrains that each pigeon gets a hole and no more two pigeons get the same hole. The last formula starts the search for an assignment of holes to the pigeons that satisfies the constraint.

`CLASP` encoding

We use the program in Example 11.1.

Experimental results

The solvable instances where the number of pigeons is smaller than the number of holes can be done in a fraction of a second by both `CLASP` and `SICSTUS` . Our tests focus on the hard instances, where the number of pigeons is greater than the number of holes by one. We generate instances that the number of pigeons is from 5 to 14. The results show that `SICSTUS` solve all of them in almost no time, while `CLASP` solves them rather slowly. For the instances where the number of pigeons is greater than 12, `CLASP` even cannot solve them but `SICSTUS` can solve them instantly (Figure 11.1).

11.2.3 The `cumulative` constraint

An important application area of ASP is in solving NP-hard scheduling problems. The `cumulative` constraint developed in constraint programming is for such problems [6]. The `cumulative` constraint matches directly the single resource scheduling problem, where we are given a collection of tasks, such that each task is associated with a *start time*; a *duration* which is the amount of time that it takes to complete; and a *height* which is the amount of resources that it takes up while it executes. In addition, we are given a natural number l which is the total amount of available resources that can be shared by the different tasks. The `cumulative` constraint states that, at any instant i of the schedule, the summation of the amount of resources of the tasks that overlap does not exceed the limit l. A filtering algorithm is detailed in [5]

Definition 11.2. [57] Let $T = \{t_1, ..., t_n\}$ be a set of tasks. Each t_i is associated with 3 variables: s_i representing the start time of t_i which is in a range D, d_i representing the duration of t_i, and h_i representing the amount of resources needed during the execution of t_i. Let l be a natural number.

The `cumulative` constraint is the constrain (X, D, C), where

- $X = \{s_1, ..., s_n\}$.
- D is the range of possible start time.
- C=$\{(a_1, ..., a_n) \mid \forall i\ 1 \leq i \leq n,\ \sum_{\{j \mid a_i \leq a_j \leq a_i + d_i - 1\}} h_j \leq l\}$.

The Problem

We study the scheduling problem as follows. Let $T = \{t_1, ..., t_n\}$ be a set of tasks where each t_i is associated with a ternary tuple (s_i, d_i, h_i). Let l and u be natural numbers representing the limit of the resources and time, respectively. The goal is to find a schedule of the tasks that guarantees every task be finished before time u.

`SICSTUS` encoding

In `SICSTUS` the description of the `cumulative` constraint is the following: `cumulative` $(+Tasks)$, where $Tasks$ is a list of tasks. Each task is represented by a term $task(S_i, D_i, E_i, H_i, T_i)$ with S_i being the start time, D_i the non-negative duration, E_i the end time, H_i the non-negative resource consumption, and T_i the task identifier. All fields are domain variables with bounded domains, or integers.

Let n be the number of tasks and l the resources limit and:

$$H_{ij} = \begin{cases} H_i & \text{if } S_i \leq j < S_i + D_i; \\ 0 & \text{otherwise.} \end{cases}$$

The constraint holds if:

1. For every task i, $S_i + D_i = E_i$, and

2. For all instants j, $H_{1j} + ... + H_{nj} \leq l$.

We model the problem by the `SICSTUS` program.

$$schedule(Ss, Es) \quad :- \tag{11.8}$$

$$Ss = [S_1, ..., S_n], \tag{11.9}$$

$$Es = [E_1, ..., E_n], \tag{11.10}$$

$$Tasks = [T_1(S_1, d_1, E_1, h_1, 1), ..., \tag{11.11}$$

$$T_n(S_n, d_n, E_n, h_n, n)],$$

$$domain(Ss, 1, u), \tag{11.12}$$

$$domain(Es, 1, u), \tag{11.13}$$

$$\texttt{cumulative}\,(Tasks, [limit(l)]), \tag{11.14}$$

$$labeling([\,], Vars). \tag{11.15}$$

The formulas (11.9) and (11.10) specify the sets of start times and end times, respectively. The formula (11.11) gives the set of tasks. The formulas (11.13) and (11.14) specify the domains of the start times and end times, respectively. The formula (11.14) enforces the `cumulative` constraint. The formula (11.15) starts the `SICSTUS` system to search for an assignment of Ss that satisfies the constraint.

CLASP encoding

We model the problem by the following CLASP program.

$$1\{start(I,T) : time(T)\}1 \quad :- \quad task(I,D,H) \tag{11.16}$$

$$use(I,T,H) \quad :- \quad start(I,T), task(I,D,H) \tag{11.17}$$

$$use(I,T,H) \quad :- \quad start(I,T1), task(I,D,H), T1 < T, \quad T \leq T1 + D \tag{11.18}$$

$$use(T,R) \quad :- \quad R = sum[use(I,T,H) : task(I,D,H) = H] \tag{11.19}$$

$$end(I,T) \quad :- \quad task(I,D,H), start(I,T1), \quad T = T1 + D \tag{11.20}$$

$$:- \quad use(T,R), R > L \tag{11.21}$$

$$:- \quad task(I,D,H), end(I,T), T > u \tag{11.22}$$

The formula (11.16) enumerates all possible assignments of the start time of task I to time T. The formulas (11.17) and (11.18) define the use of resources by a task at an instant. The formula (11.19) defines the end time of a task. The formulas (11.21) and (11.22) guarantee that the resources used at any instant cannot be beyond the limit and each task is finished in the given time.

Experimental results

We randomly generate 100 instances half of them are satisfiable (there exists a schedule) and the other half are unsatisfiable (i.e. no schedule satisfying the constraint exists). For the satisfiable instances, we set the limits of the resources to be 15; the numbers of tasks are 8, 10 ,11, 12, and 13 and time limits are 80, 100, 110, 120 and 130, respectively. For each setting, we randomly generate 10 instances where the tasks have distinct durations and heights. For the unsatisfiable instances, we set the limit of resources to be 15 and the height of each task to be 14. By this setting, no tasks can overlap. We then set total durations of the tasks greater than the time limit.

The experimental results show that for most of the solvable instances, SICSTUS is faster than CLASP by several orders of magnitude except for two instances. The number of unsolvable instances by SICSTUS is much less than that by CLASP (Figure 11.2). Actually, we found that when the number of tasks is greater than 14, CLASP cannot solve the instances any more but SICSTUS can still solve them in fraction of a second.

11.2.4 The `sort` constraint

The `sort` constraint is proposed in [7]: "A sortedness constraint express that an n-tuple $(y_1, ..., y_n)$ is equal to the n-tuple obtained by sorting in increasing order the terms of another n-tuple $(x_1, ..., x_n)$." The best filtering algorithm has been the one proposed in [55].

Definition 11.3. Let $\{a_1, ..., a_n\}$ be a set of numbers, $X = \{x_{a_1}, ..., x_{a_n}\}$ be a set of variables and $D = \{1, ..., n\}$ be the domain of the variables. The `sort` constraint (X, D, C) is the constraint where

$$C = \{(d_1, ..., d_n) \mid \forall i \neq j \; 1 \leq i, j \leq n, \; d_i \leq d_j \; whenever \; i < j\}.$$

The problem

We model the problem of sorting a given set of numbers $X = \{a_1, ..., a_n\}$ in increasing order by `SICSTUS` and `CLASP`. Note that we suppose the numbers in X are distinct.

`SICSTUS` encoding

The `sort` constraint in `SICSTUS` is specified as follows: `sorting` $(+X, +P, +Y)$, where $X = [X_1, ..., X_n]$, $P = [P_1, ..., P_n]$, and $Y = [Y_1, ..., Y_n]$ are lists of domain variables. The constraint holds if the following are true:

1. Y is in ascending order,
2. P is a permutation of $[1..n]$, and
3. for any i in $1, ..., n$, $X_i = Y_{P_i}$.

The problem can be modeled by the program below.

$$sort(P) \; :- \tag{11.23}$$
$$X = [a_1, ..., a_n], \tag{11.24}$$
$$Y = [Y_0, ..., Y_n], \tag{11.25}$$
$$P = [P_1, ..., P_n], \tag{11.26}$$
$$domain(P, 1, n), \tag{11.27}$$
$$domain(Y, 1, u), \tag{11.28}$$
$$\texttt{sorting}\,(X, P, Y), \tag{11.29}$$
$$labeling([\,], P). \tag{11.30}$$

The formula (11.24) gives X, the set of numbers to sort. The formulas (11.25) and (11.26) specify that Y is the sorted sequence of the numbers and P is the positions of each number in X, respectively. Then the constraint `sort` is enforced in formula (11.29) and the formula (11.30) starts the search for the assignments to the variables in P.

`CLASP` encoding

We model the sorting problem as follows.

$$1\{place(X,Y) : position(Y)\}1 \quad :- \quad number(X). \tag{11.31}$$

$$1\{place(X,Y) : number(X)\}1 \quad :- \quad position(Y). \tag{11.32}$$

$$:- \quad place(X_1,Y_1), place(X_2,Y_2), \tag{11.33}$$
$$X_1 > X_2, Y_1 < Y_2,$$

The predicate $place(X,Y)$ represents that the number X is the Yth number in the ordered sequence. The first two formulas state that, in the ordered sequence, each number has a place and there is a number at each place. The last formula guarantees the ordered sequence be increasing.

Experimental results

In our experiments, we generate a series of sets of numbers with 10 different sizes: 20, 30, 40, 50, 60, 70, and from 72 to 75. We randomly generated 10 sets for each size. We run all of the 100 instances using `SICSTUS` and `CLASP` programs, respectively. `SICSTUS` solve all of the instances using almost no time but `CLASP` need considerable time when the size of the set is greater than 70. Actually, `CLASP` cannot sort the set if its size is beyond 80, while `SICSTUS` still can sort it in a fraction of a second (Figure 11.3).

11.3 A property of global constraints

We present a property of global constraints, called the *compactness* of local power sets. As an application of this property, we show that the property makes it easier to construct the dependency graph of a logic program with global constraints than that of logic programs with c-atoms in general case. Recall that the dependency graph is indispensable to the formulation of loop formulas.

11.3.1 Global constraints as c-atoms

In this section, we give the representation of global constraints as c-atoms. The concept of conditional satisfaction and answer sets of logic programs with c-atoms are naturally extended to logic programs with global constraints.

The *domain* of a variable x, denoted $D(x)$, is a finite set of elements that can be assigned to x. For a set of variables X, we denote the union of their domains by $D(X) = \cup_{x \in X} D(x)$.

A (global) constraint is a triple (X, D, C), where $X = \{x_1, ..., x_m\}$ is a set of variables; $D = D(X)$ is the union of the domains of variables in X; C is a subset of the Cartesian product of the domains of the variables in X, i.e., $C \subseteq D(x_1) \times D(x_2) \times ... \times D(x_m)$. A tuple $(d_1, ..., d_m) \in C$ is called a *solution* to C. Equivalently, we say that a solution $(d_1, ..., d_m) \in C$ is an assignment of the value d_i to the variable x_i, for all $1 \leq i \leq m$, and that this assignment satisfies C. If $C = \emptyset$, we say that it is inconsistent.

Let $G = (X, D, C)$ be a global constraint. The *c-atom representation* of G, denoted $A(G) = (A(G)_d, A(G)_c)$ is a c-atom as follows.

- $A(G)_d = \{x_i(d_j) \mid x_i \in X,\ d_j \in D(x_i)\}$. Intuitively, A_d consists of all possible assignments of the variables.
- $A(G)_c = \{\{x_1(d_{i1}), ..., x_m(d_{im})\} \mid (d_{i1}, ..., d_{im}) \in C\}$, that is, A_c consists of all solutions of C.

Since the global constraint G is usually clear in a context, we may write $A(G)$, $A(G)_d$, and $A(G)_c$ as A, A_d, and A_c, respectively.

Example 11.2. Let $G = (X, D, C)$ be a global constraint, where $X = \{x_1, x_2\}$, $D = \{1, 2\}$, and $C = \{(1, 2), (2, 1)\}$. Note that G is actually `alldifferent` (x_1, x_2). Then the c-atom representation of G is the c-atom

$$A = (\{x_1(1), x_1(2), x_2(1), x_2(2)\}, \{\{x_1(1), x_2(2)\}, \{x_1(2), x_2(1)\}\}).$$

□

In the sequel, when we talk about a global constraint, we refer to the c-atom representation of the constraint if not stated otherwise.

11.3.2 Compactness of local power sets

We present a property of global constraints, called the *compactness* of local power sets.

Given a global constraint $G = (X, D, C)$ and its c-atoms representation A, the sets of solutions in A_c are all of the same size, i.e. for all S_i and S_j such that $S_i \in A_c$ and $S_j \in A_c$, we have $|S_i| = |S_j| = |X|$. This fact leads to the following proposition.

Proposition 11.1. Let A be a global constraint and $R \subseteq A_d$ and $S \subseteq A_d$. If $R \in A_c$, $S \in A_c$ and $R \neq S$, then $R \not\subset S$ and $S \not\subset R$.

Theorem 11.1. Let A be a global constraint. The local power set (LPS) representation of A, A^* is a c-atom such that

- $A^*_d = A_d$, and
- $A^*_c = \{\langle S, S\rangle \mid S \in A_c\}$.

Proof. The theorem can be proved by contraposition. Suppose $\langle B, T\rangle$ is a local power set of A, where $B \neq T$. By Proposition 11.1, we have $B \not\subset T$. This contradicts that $\langle B, T\rangle$ is a local power set of A. □

Theorem 11.1 shows that, for a global constraint A, each LPS of A^* contains only one solution in A_c. Thus, we call this property as the *compactness* of LPS (or *compactness*) of global constraints. We may write $\langle S, S\rangle$ simply as S. Then A^* is exactly the same as A, i.e., $A^* = (A_d, A_c)$.

Example 11.3. Consider the `alldifferent` constraint A in Example 11.2, which is

$$A = (\{x_1(1), x_1(2), x_2(1), x_2(2)\}, \{\{x_1(1), x_2(2)\}, \{x_1(2), x_2(1)\}\}). \tag{11.34}$$

The LPS representation of A is

$$A^* = (\{x_1(1), x_1(2), x_2(1), x_2(2)\}, \{\langle\{x_1(1), x_2(2)\}, \{x_1(1), x_2(2)\}\rangle, \langle\{x_1(2), x_2(1)\}, \{x_1(2), x_2(1)\}\rangle\}). \tag{11.35}$$

For simplicity's sake, A^* can be written as A, i.e.

$$A^* = (\{x_1(1), x_1(2), x_2(1), x_2(2)\}, \{\{x_1(1), x_2(2)\}, \{x_1(2), x_2(1)\}\}) \tag{11.36}$$

□

11.3.3 Discussion

We take the answer set semantics based on conditional satisfaction (c.f. Section 8.1) as the semantics for logic programs with global constraints.

Dependency graph

The concept of dependency graph is indispensable to the definition of loop formulas, which have been used to characterize the semantics and compute the answer sets for logic programs.

In Chapter 9, we define the dependency graph for logic programs with c-atoms for the formulation of loop formulas. In the definition, atoms in the domain of the head of a rule depend on atoms that are in the first component of the local power set (LPS) representation of each c-atom in the body of the rule. Therefore, to construct the dependency graph, one has to compute the LPS representation of each c-atom in the body of the rules. For a c-atom A, the time complexity of the algorithm to compute the LPS representation of A is $O(|A_c|^3)$ [84].

If a c-atom is a global constraint, the computation of LPS representation is not needed, due to the compactness of global constraints (since the LPS representation of the c-atom is essentially the same as itself).

Example 11.4. Consider the program P which consists of the rule

$$a \leftarrow A, B \tag{11.37}$$

where A is the constraint in Example 11.2 and B is the c-atom $(\{a, b\}, \{\{a\}, \{a, b\}\})$. To construct the dependency graph, we do not need to compute the LPS representation of A. The LPS representation of B is $(\{a, b\}, \{\langle\{a\}, \{a, b\}\rangle\}$. Therefore the dependency graph $G_P = (V, E)$ where

- $V = \{a, b, x_1(1), x_2(2), x_2(1), x_2(2)\}$, and
- $E = \{(a, a), (a, x_1(1)), (a, x_1(2)), (a, x_2(1)), (a, x_2(2))\}$.

□

Transformation to normal programs

The unfolding approach has been used for the study of computations and properties of logic programs with c-atoms in the literature [20, 21]. In [87], we give an approach to unfold a logic program with c-atoms to a normal logic program. In the approach, three steps are needed to unfold c-atoms in rule bodies, where the last two steps are essentially to compute the first components of the local power sets of the c-atoms. For global constraints, the last two steps are not needed anymore, due to the compactness of global constraints. This makes the unfolding simpler.

Example 11.5. Let P be the program $\{a \leftarrow A\}$, where A is the `alldifferent` constraint in Example 11.2. It can be verified that the unfolding of P, denoted $U(P)$, is

$$a \leftarrow x_1(1), x_2(2), \texttt{not}\ x_1(2), \texttt{not}\ x_2(1) \tag{11.38}$$

$$a \leftarrow x_1(2), x_2(1), \texttt{not}\ x_1(1), \texttt{not}\ x_2(2) \tag{11.39}$$

□

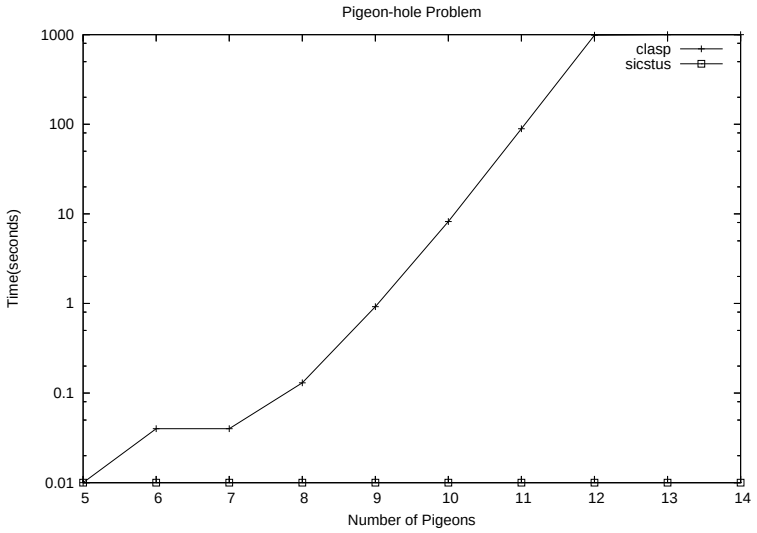

Figure 11.1: Pigeon-hole Problem

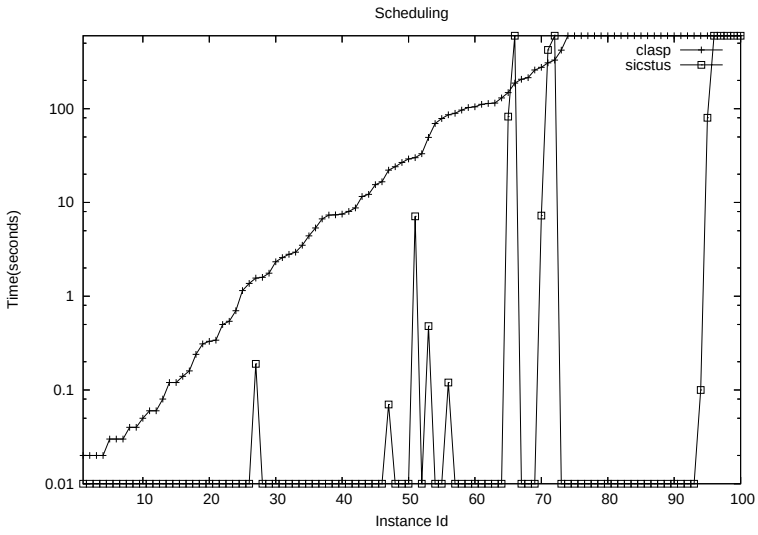

Figure 11.2: Scheduling

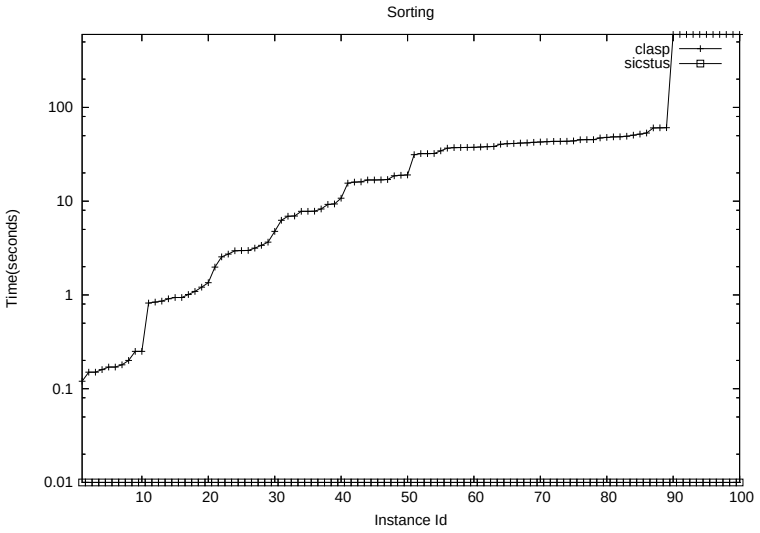

Figure 11.3: Sorting

Benchmarks	Number of Instances	Solved Instances		Execution Time	
		SICSTUS	CLASP	SICSTUS	CLASP
Pigeon-hole	10	10	8	0.01	135.64
Scheduling	100	93	73	1.35	40.75
Sorting	100	100	89	0.01	22.29

Table 11.1: Summary SICSTUS and CLASP

11.4 Summary and future work

This chapter is a preliminary study on the integration of global constraints to ASP. We present the experiments that demonstrate the efficiency of global constraints and their designated filtering algorithms. Then, we show that global constraints possess a property which makes it more convenient to study the computation and properties of logic programs with global constraints than c-atoms in the general case.

Further investigations on language design and computation are needed for the implementation of logic programs with global constraints.

Regarding the language, a problem of interest is the syntax of the built-in predicates in ASP language to represent the global constraints and model the real world problems.

For the computation, it is desirable to build a system that unifies the filtering algorithms for global constraints and the search procedure for answer sets. To implement such a system, the following issues have to be explored: interactions between the filtering algorithms and the main search procedure; constraint propagations involving global constraint

and other constraints; and related data structures and algorithms.

Chapter 12

Conclusion

Weight constraint, aggregate and abstract constraint programs are recent extensions to logic programs. We study the properties and computations of these programs. For properties we focus on loop formulas, since loop formulas are both a characterization of a semantics and key part of an approach to computing answer set of programs. For computation, we improve the existing methods for weight constraint programs and propose an approach to computing the aggregate programs under answer set semantics.

Specifically, the main contributions of the thesis are the following. For weight constraint programs, we present a formulation of loop formulas based on the level mapping characterization of stable model semantics. In the formulation, no new atoms are needed thus avoid the extra search space brought by them. We study the effectiveness of a constraint propagation scheme, lookahead and observe that, for some programs, lookahead is totally a waste and reduces the performance of ASP solvers. Based on this finding, we introduce an adaptive lookahead mechanism which invokes lookahead dynamically upon the information collected during the search. Adaptive lookahead exploits the pruning power of lookahead while avoiding the unnecessary overhead caused by it.

For aggregate programs, we introduce loop formulas based on the level mapping characterization of answer set semantics. The formulation averts the procedure to compute the local power sets of the aggregates, which takes exponential time in the size of the aggregate. We show that aggregate programs can be translated to weight constraint programs, so that the answer sets of the aggregate programs are exactly the stable models of the translated weight constraint programs. Thus, the answer sets of aggregate programs can be computed using the state-of-the-art solver implemented for the computation of stable models for weight constraint programs.

For abstract constraint programs, we define loop formulas for programs with arbitrary

constraints. We also characterize the strong equivalence of abstract constraint programs.

For the future work, we believe that the integration of global constraints to ASP is an encouraging topic. This integration will make ASP more effective for problem modeling and substantially more efficient for computation of answer sets. As a preliminary study in this direction, we provide motivating experiments. Many issues in the integration present interesting questions for future research.

Bibliography

[1] D. Armi, W. Faber, and G. Ielpa. Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in DLV*. In *Proc. IJCAI'03*, pages 847–852, 2003.

[2] Y. Babovich. Cmodels. http://www.cs.utexas.edu/users/tag/cmodels.html, 2002.

[3] F. Bacchus. Exploring the computational tradeoff of more reasoning and less searching. In *Proc. SAT'02*, pages 7–13, 2002.

[4] M. Balduccini, M. Gelfond, R. Watson, and M. Nogueira. The USA-advisor: A case study in answer set planning. In *Proc. LPNMR-01*, pages 439–442, 2001.

[5] N. Beldiceanu and M. Carlsson. A new multi-resource cumulatives constraint with negative heights. In *Proc. CP'02*, pages 63–79, 2002.

[6] N. Beldiceanu and E. Contejean. Introducing global constraints in chip. *Journal of Mathematical and Computer Modeling*, 20(12):97–123, 1994.

[7] N. Bleuzen-Guernalec and A. Colmerauer. Narrowing a *2n*-block of sortings in O*(n logn)*. In *Proc. CP'97*, pages 2–16, 1997.

[8] J. E. Borrett, E. P. Tsang, and N. R. Walsh. Adaptive constraint satisfaction: The quickest first principle. In *TR CSM-256, University of Essex*, 1995.

[9] P. Cabalar. Partial functions and equality in answer set programming. In *Proc. ICLP'08*, pages 392–406, 2008.

[10] F. Calimeri, W. Faber, N. Leone, and S. Perri. Declarative and computational properties of logic programs with aggregates. In *Proc. IJCAI'05*, pages 406–411, 2005.

[11] D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(3):305–317, 2005.

[12] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *Proc. IJCAI'91*, pages 331–337, 1991.

[13] K. L. Clark. Negation as failure. *Logics and Databases*, pages 293–322, 1978.

[14] The First ASP System Competition. http://www.asparagus.cs.uni-potsdam.de/.

[15] J. Culberson. http://web.cs.ualberta.ca/joe/coloring/index.html.

[16] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[17] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[18] M. Denecker, N. Pelov, and M. Bruynooghe. Ultimate well-founded and stable semantics for logic programs with aggregates. In *Proc. ICLP'01*, pages 212–226, 2001.

[19] H. E. Dixon and M. L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In *Proc. AAAI'02*, pages 635–641, 2002.

[20] I. Elkabani, E. Pontelli, and T. C. Son. Smodels with CLP and its applications: a simple and effective approach to aggregates in ASP. In *Proc. ICLP'04*, pages 73–89, 2004.

[21] I. Elkabani, E. Pontelli, and T. C. Son. $Smodels^A$ – a system for computing answer sets of logic programs with aggregates. In *Proc. LPNMR'05*, pages 427–431, 2005.

[22] E. Erdem. *Theory and Applications of Answer Set Programming*. PhD thesis, University of Texas, 2002.

[23] E. Erdem and V. Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3(4):499–518, 2003.

[24] E. Erdem, F. Lin, and T. Schaub, editors. *Proceedings of LPNMR'09*. Springer, 2009.

[25] W. Faber, N. Leone, and G. Pfeifer. Recursive aggregates in disjunctive logic programs. In *Proc. JELIA'04*, pages 200–212, 2004.

[26] F. Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

[27] P. Ferraris. Answer sets for propositional theories. In *Proc. LPNMR'05*, pages 119–131, 2005.

[28] P. Ferraris and V. Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5:45–74, 2005.

[29] J. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, 1995.

[30] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. http://potassco.sourceforge.net/.

[31] M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solver *clasp*: Progress report. In *Proc. LPNMR'09*, pages 509–514, 2009.

[32] M. Gebser, L. Liu, G. Namasivayam, A. Neumann, T. Schaub, and M. Truszczyński. The first answer set programming system competition. In *Proc. LPNMR'07*, pages 1–17, 2007.

[33] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. 5th ICLP*, pages 1070–1080, 1988.

[34] E. Ginuchiglia, M. Maratea, and A. Tacchella. (in)effectiveness of look-ahead techniques in a modern sat solver. In *Proc. CP'03*, pages 842–846, 2003.

[35] M. Heule, M. Dufour, and J. van Zwieten. March_eq:implementing additional reasoning into an efficient look-ahead sat solver. In *Proc. SAT'04*, pages 145–159, 2004.

[36] X. Jia, J. You, and L. Yuan. Adding domain dependent knowledge to answer set programs for planning. In *Proc. ICLP'04*, pages 400–415, 2004.

[37] E. Lamma, P. Mello, and M. Milano. An incremental consistency algorithm for adaptive constraint satisfaction. In *TR DEIS-LIA-97-002, University of Bologna*, 1997.

[38] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):1–57, 2006.

[39] C. M. Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proc. CP'97*, pages 341–355, 1997.

[40] V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic,2(4)*, pages 526–541, 2001.

[41] F. Lin. Reducing strong equivalence of logic programs to entailment in classical propositional logic. In *Proc. KR'02*, pages 170–176, 2002.

[42] F. Lin and Y. Wang. Answer set programming with functions. In *Proc. ICLP'08*, pages 454–465, 2008.

[43] F. Lin and Y. Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[44] G. Liu. Level mapping induced loop formulas for weight constraint and aggregate programs. In *Proc. LPNMR'09*, pages 444–449, 2009.

[45] G. Liu and J. You. On the effectiveness of looking ahead in search for answer sets. In *Proc. LPNMR'07*, pages 303–308, 2007.

[46] G. Liu and J. You. Lparse programs revisited: semantics and representation of aggregates. In *Proc. ICLP'08*, pages 347–361, 2008.

[47] G. Liu and Jia-Huai You. Adaptive lookahead for answer set computation. In *Proc. ICTAI'07*, pages 230–237, 2007.

[48] L. Liu, E. Pontelli, T. C. Son, and M. Truszczyński. Logic programs with abstract constraint atoms: The role of computations. In *Proc. ICLP'07*, pages 286–301, 2007.

[49] L. Liu and M. Truszczyński. Properties and applications of programs with monotone and convex constraints. *Journal of Artificial Intelligence Research*, 7:299–334, 2006.

[50] A. Lopez-Ortiz, C. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *Proc. IJCAI'03*, pages 245–250, 2003.

[51] V. Marek, I. Niemelä, and M. Truszczyński. Logic programs with monotone abstract constraint atoms. *Theory and Practice of Logic Programming*, 8(2):167–199, 2007.

[52] V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K.R. Apt et al., editor, *The Logic Programming Paradigm: A 25-Year Perspective*, pages 375–398. Springer, 1999.

[53] V. Marek and M. Truszczyński. Logic programs with abstract constraint atoms. In *Proc. AAAI '04*, pages 86–91, 2004.

[54] V. W. Marek and J. B. Remmel. Set constraints in logic programming. In *Proc. LPNMR'04*, pages 167–179, 2004.

[55] K. Mehlhorn and S. Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In *Proc. CP'00*, pages 306–319, 2000.

[56] V. S. Mellarkod, M. Gelfond, and Y. Zhang. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Integlligence*, 53(1-4):251–287, 2008.

[57] M. Milano, editor. *Constraints and Integer Programming Combined*, chapter Global Constraints and Filtering Algorithms. Kluwer, 2005.

[58] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of sat problems. In *Proc. of AAAI'92*, pages 459–465, 1992.

[59] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proc. DAC'00*, pages 530–535, 2000.

[60] G. Namasivayam and M. Truszczyński. An $smodels$ system with limited lookahead computation. In *Proc. LPNMR'07*, pages 278–284, 2007.

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

[62] I. Niemelä, P. Simons, and T. Soininen. Stable model semantics of weight constraint rules. In *Proc. LPNMR '99*, pages 317–331, 1999.

[63] N. Pelov. *Semantics of Logic Programs with Aggregates*. PhD thesis, Ketholieke Universiteit Leuven, 2004.

[64] N. Pelov, M. Denecker, and M. Bruynooghe. Translation of aggregate programs to normal logic programs. In *Proc. ASP'03*, pages 29–42, 2003.

[65] N. Pelov, M. Denecker, and M. Bruynooghe. Partial stable models for logic programs with aggregates. In *Proc. LPNMR'04*, pages 207–219, 2004.

[66] N. Pelov, M. Denecker, and M. Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming*, 7:301–353, 2007.

[67] N. Pelov and M. Truszczyński. Semantics of disjunctive programs with monotone aggregates an operator-based approach. In *Proc. LPNMR '04*, pages 327–334, 2004.

[68] J. C. Régin. A filtering algorithm for constraints of difference in csps. In *Proc. AAAI'94*, pages 362–367, 1994.

[69] F. Rossi, P. Van Beek, and T. Walsh, editors. *Hand Book of Constraint Programming*, chapter Global Constraints. Elsevier, 2006.

[70] H. E. Sakkout, M. G. Wallace, and E. B. Richards. An instance of adaptive constraint propagation. In *Proc. CP'96*, pages 164–178, 1996.

[71] Y. Shen and J. You. A generalized gelfond-lifschitz transformation for logic programs with abstract constraints. In *Proc. AAAI'07*, pages 483–488, 2007.

[72] J. Silva and K.A. Sakallah. Grasp–a new search algorithm for satisfiability. In *Proc. ICCAD'96*, pages 220–227, 1996.

[73] P. Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Helsinki, Finland, 2000.

[74] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[75] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Proc. PADL'99*, pages 305–319, 1999.

[76] T. C. Son and E. Pontelli. A constructive semantic characterization of aggregates in answer set programming. *Theory and Practice of Logic Programming*, 7:355–375, 2006.

[77] T. C. Son, E. Pontelli, and P. H. Tu. Answer sets for logic programs with arbitrary abstract constraint atoms. *Journal of Artificial Intelligence Research*, 29:353–389, 2007.

[78] H. Turner. Strong equivalence for logic programs and default theories (made easy). In *Proc. LPNMR'01*, pages 81–92, 2001.

[79] H. Turner. Strong equivalence made easy: nested expressions and weight constraints. *Theory and Practice of Logic Programming,3*, pages 609–622, 2003.

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

[81] W. J. van Hoeve and I. Katriel. Global constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 7. Elsevier, 2006.

[82] G. Wu, J. You, and G. Lin. Quartet based phylogeny reconstruction with answer set programming. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 4(1):139–152, 2007.

[83] J. You and G. Hou. Arc consistency + unit propagation = lookahead. In *Proc. ICLP'04*, pages 314–328, 2004.

[84] J. You and G. Liu. Loop formulas for logic programs with arbitrary constraint atoms. In *Proc. AAAI'08*, pages 584–589, 2008.

[85] J. You, G. Liu, L. Yuan, and C. Onuczko. Lookahead in Smodels compared to local consistencies in CSP. In *Proc. of LPNMR'05*, pages 266–278, 2005.

[86] J. You, L. Yuan, and M. Zhang. On the equivalence between answer sets and models of completion for nested logic programs. In *Proc. IJCAI'03*, pages 859–865, 2003.

[87] J. You, L. Y. Yuan, G. Liu, and Y. Shen. Logic programs with abstract constraints: Representation, disjunction and complexities. In *Proc. LPNMR'07*, pages 228–240, 2007.

[88] H. Zhang. SATO: an efficient propositional prover. In *Proc. CADE'97*, pages 272–275, 1997.