Università della Calabria

Dipartimento di Matematica

Dottorato di Ricerca in Matematica ed Informatica

xxiv ciclo

Settore Disciplinare INF/01 – INFORMATICA

Tesi di Dottorato

Parallel Evaluation of ASP programs: Techniques and Implementation

Marco Sirianni

Supervisori
Prof. Francesco Ricca
Prof. Nicola Leone

Coordinatore
Prof. Nicola Leone

A.A. 2010 – 2011

Università della Calabria

Dipartimento di Matematica

Dottorato di Ricerca in Matematica ed Informatica

XXIV CICLO

Settore Disciplinare INF/01 – INFORMATICA

Tesi di Dottorato

Parallel Evaluation of ASP programs: Techniques and Implementation

[illegible]

Marco Sirianni

Supervisori

Prof. Francesco Ricca [illegible]

Prof. Nicola Leone [illegible]

Coordinatore

Prof. Nicola Leone [illegible]

A.A. 2010 – 2011

Dedicated to my family:
my mother Fiorella, my father Antonio,
my sister Miriam, and my brother Paolo,
for their moral support and unconditional love

Acknowledgments

First of all, I would like to express my deep and sincere gratitude to my supervisor Francesco Ricca, whose patient guidance, constant support, and warm encouragement have made possible to achieve the results presented in this thesis. His energy and enthusiasm in research gave me all the inspiration and motivation I needed to complete my doctoral studies. It has been a privilege and honour to work under his supervision during these years.

I am deeply grateful to Simona Perri for her constant help and brilliant advice; her steadfast support to my studies has been greatly needed and deeply appreciated.

I am also thankful to my supervisor Nicola Leone for the precious advices he gave me, and for all the opportunities for personal and professional growth he ensured me.

I would like to thank also the research group of the Department of Mathematics at Unical, which has been a source of friendships as well as good advice and collaboration. Among them, a sincere and grateful thanks goes to all the colleagues who shared the office with me, especially Lucantonio Ghionna and Mario Alviano for their friendship and constant help.

I wish to thank also all the long-standing friends for always keeping in contact and do not let the miles set us apart. Their friendship has helped me to get through tough periods.

Lastly, and most importantly, I owe my deepest gratitude to my mother, my father, my sister, my brother, and my cat for their care, moral support, and love.

Abstract

Answer Set Programming (ASP) is a purely declarative programming paradigm based on nonmonotonic reasoning and logic programming. The idea of ASP is to represent a given computational problem by a logic program such that its answer sets correspond to solutions, and then, use an answer set solver to find such solutions. The ASP language is very expressive and allows the representation of high-complexity problems (i.e., every problem in the second level of the polynomial hierarchy); unfortunately, the expressive power of the language comes at the price of an elevated cost of answer set computation. Even if this fact has initially discouraged the development of ASP system, nowadays several implementations are available, and the interest for ASP is growing in the scientific community as well as in the field of industry.

In the last few years, significant technological enhancements have been achieved in the design of computer architectures, with a move towards the adoption of multi-core microprocessors. As a consequence, Symmetric Multi-Processing (SMP) has become available even on non-dedicated machines. Indeed, at the time of writing, the majority of computer systems and even laptops are equipped with (at least one) dual-core processor. However, in the ASP context, the available systems were not designed to exploit parallel hardware; thus significant improvements can be obtained by developing ASP evaluation techniques that allow the full exploitation of the computational resources offered by modern hardware architectures.

The evaluation of ASP programs is traditionally carried out in two steps. In the first step an input program $\mathcal{P}$ undergoes the so-called instantiation process, which produces a program $\mathcal{P}'$ semantically equivalent to $\mathcal{P}$ but not containing any variable; in turn, $\mathcal{P}'$ is evaluated by using a backtracking search algorithm in the second step.

The aim of this thesis is the design and the assessment of a number of parallel techniques devised for both steps of the evaluation of ASP programs. In particular, a three-level parallel instantiation technique is presented for improving the efficiency of the instantiation process which might become a bottleneck in common situations, especially when huge input data has to been dealt with. Moreover, a parallel multi-heuristics search algorithm and a parallel lookahead technique have been conceived for optimizing the second phase of the evaluation.

The mentioned parallel techniques has been implemented in the state-of-the-art ASP system DLV. An extensive experimental analysis has been carried out in order to assess the performance of the implemented prototypes. Experimental results have confirmed the efficiency of the implementation and the effectiveness of those techniques.

Sommario

L'Answer Set Programming (ASP) è un paradigma di programmazione dichiarativo basato sul ragionamento non-monotono e la programmazione logica. L'idea alla base del ASP è di permettere la rappresentazione di un problema complesso mediante un programma logico i cui answer set, determinati tramite un ASP solver, corrispondono alle soluzioni ricercate. Il linguaggio dell'ASP è molto espressivo, ed infatti, permette di rappresentare problemi di elevata complessità computazionale (cioè, tutti i problemi appartenenti al secondo livello della gerarchia polinomiale); ovviamente, la sua elevata espressività si accompagna ad una elevata complessità del calcolo degli answer set. Anche se questo fatto ha inizialmente scoraggiato lo sviluppo di sistemi ASP, esistono oggi diverse implementazioni, e l'interesse per l'ASP è in forte crescita tanto nella comunità scientifica quanto nel campo dell'industria.

Nell'ultimo decennio, gli sviluppi nella produzione del hardware hanno portato al passaggio dal modello a singolo processore a quello multi-processore/ multi-core. Questo ha reso possibile l'adozione di sistemi paralleli di tipo SMP anche per sistemi non dedicati. Infatti, ad oggi, la quasi totalità dei computer in produzione, computer portatili compresi, è equipaggiato con almeno un processore dual-core.

Nel campo dell'ASP, tuttavia, la maggior parte dei sistemi disponibili è stata disegnata senza tener in conto la possibilità di usufruire di sistemi multiprocessore e non è in grado di sfruttare appieno la potenza computazionale offerta delle moderne architetture hardware; per questo motivo, significativi miglioramenti posso essere ottenuti sviluppando tecniche di valutazione parallela di programmi ASP in grado di trarre vantaggio da tali architetture.

La valutazione di programmi ASP viene tradizionalmente effettuata in due passi. Il primo passo, chiamato istanziazione , consta in un processo di traduzione del programma in input $\mathcal{P}$ in uno $\mathcal{P}'$ semanticamente equivalente a $\mathcal{P}$ ma privo di variabili. Tale programma $\mathcal{P}'$ viene successivamente valutato mediante l'utilizzo di un algoritmo di backtracking, al fine di produrre le soluzioni di $\mathcal{P}$.

L'obiettivo di questa tesi è lo studio, la progettazione e la valutazione di tecniche parallele per la valutazione di programmi ASP. Più in dettaglio, presentiamo una tecnica a tre livelli per l'istanziazione parallela dei programmi, al fine di migliorare l'efficienza di questa fase della valutazione, che, come è noto, può diventare un collo di bottiglia specialmente nel caso di input di grandi dimensioni. Presentiamo inoltre due tecniche volte a potenziare la seconda fase della valutazione di programmi ASP: una ricerca parallela multi-euristica ed il calcolo parallelo dei valori della funzione usata dall'euristica di branching. È stata inoltre realizzata una implementazione di tutte le tecniche sopra elencate nel sistema DLV, uno dei sistemi ASP più diffusi allo stato dell'arte.

I risultati di una esaustiva analisi sperimentale, effettuata al fine di valutare sul campo le nuove tecniche, hanno confermato sia l'efficienza dell'implementazione che l'efficacia delle tecniche proposte.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Answer Set Programming (ASP) [1, 2, 3, 4, 5, 6] is a purely declarative formalism for knowledge representation and reasoning rooted in the fields of logic programming and nonmonotonic reasoning. ASP is a rule-based language, which allows for both disjunction in the head of rules and nonmonotonic negation in the body of rules. Problem solving with ASP is a declarative approach, thus, no procedural information is encoded into an ASP program. More in detail, given a computational problem P a set of ASP rules $\mathcal{P}$, which is called ASP program, has to be provided such that the solutions of P correspond to the answer sets of $\mathcal{P}$.

The language of ASP is characterized by a relatively high expressive power [7, 8], since it allows the representation of all problems belonging to the second level of the polynomial hierarchy [2].[1] This language expressivity is needed for solving several significant complex problems, such as tasks in AI, in Knowledge Representation and Reasoning, etc. (see [10]), that are non-polynomially reducible to SAT. Unfortunately, the expressive power of the language is associated with an elevated computational cost of evaluation that makes the implementation of effective and efficient ASP systems a difficult task. This issue has been addressed since the birth of ASP, and has initially discouraged the implementation of scalable systems. However, after some pioneering work [11, 12], there are nowadays a number of systems that support ASP and its variants [7, 13, 14, 15, 16, 17, 18]. The availability of efficient systems made ASP profitably exploitable in real-world applications [19, 20, 21] and, recently, also in industry [22, 23, 24].

The answer set computation engines traditionally operate on a ground transformation of the original program, i.e., a program that does not contain any variable, but is semantically equivalent to the original input. Therefore, the computation of ASP programs is commonly carried out as a two-step process. In the first step, an input program $\mathcal{P}$ undergoes the so-called *instantiation* (or *grounding*) process, which produces a ground program $\mathcal{P}'$ semantically equivalent to $\mathcal{P}$. This task is usually performed starting from the ground information of $\mathcal{P}$ that is exploited for generating new ground rules, and then, iterating this process until no new information can be derived. The second step, which amounts to computing the answer sets of $\mathcal{P}'$, is performed using a backtracking search algorithm that generates candidate solutions; those, in turn, undergo a stability

[1]It is usual for logic-based languages to refer to data complexity [9].

check that allows the filtering of the actual answer sets. Both the instantiation and the propositional search phase are complex, i.e., computationally-expensive tasks [2, 8, 25]), and providing efficient evaluation techniques is crucial for the development of performant ASP systems.

Concerning the first task, nowadays it is widely recognized that having an efficient instantiation procedure is crucial for the performance of the entire ASP system. Many optimization techniques have been proposed for this purpose [26, 27, 28]; nevertheless, the performance of instantiators can be further improved in many cases, especially when the input data are significantly large (real-world instances, for example, may count hundreds of thousands of logic facts).

Concerning the propositional search task, a crucial role is played by the heuristic criteria employed for guiding the backtracking search for answer sets. The computation of the heuristic values in an answer set solver might be, however, computational expensive [29], and might become a bottleneck for the overall process of answer set computation. Moreover, the choice of the most suitable heuristics for the program at hand is still a fundamental point since it can determine a fast traversal of the search space. However, the choice of the heuristics is usually made at the beginning of the search process and remains a static input parameter in most modern ASP Solvers. Impressive speedups could be obtained if the solver can adopt several competing heuristics at the same time.

Nonetheless, the majority of ASP solvers rely on serial algorithms for both the instantiation and the propositional search; however, the application of parallel computation techniques can be fruitfully exploited to increase the performances of the solvers in both phases. Note also that, in the last decade, Moore's law on the progress of computers computational power has started to fail in the case of single processor machines. Indeed, the technological enhancement in computer architecture has moved from increasing the computational power of single CPUs to the adoption of multiple-core/multi-processor machines. The advantages of Symmetric Multi-Processing (SMP) [30] are now available also on non-dedicated machines, whereas, in the past, this parallel computation paradigm was supported only by expensive servers and workstations. Nowadays, the majority of the commercial personal computers and even laptops are equipped with (at least one) dual-core processor. This means that the benefits of true parallel processing can be enjoyed also by entry-level systems and PCs.

However, the performance-hungry ASP systems are mainly based on serial algorithms (especially in the case of the instantiation task) and, thus, are not able to exploit fully the computational power offered by modern hardware architectures.

This thesis addresses this issue, and is about the design and the implementation of a number of techniques for the parallel evaluation of ASP programs. The main contribution is related to the instantiation process for which we designed:

1. A three-level parallel evaluation technique, which allows:
    - breaking up the input program into independent sub-programs and computing their instantiation in parallel,
    - parallelizing the instantiation of the rules within the sub-programs,
    - breaking up the computation of a single rule in order to compute its instantiation concurrently;

2. a heuristics for optimizing the workload distribution among processors;

3. a heuristics for granularity control.

Besides the design of the parallel instantiation strategy, we also investigated possible methods for the exploitation of parallelism during the propositional search phase, and designed:

4. A multi-heuristics parallel search, which allows the exploration of the search space concurrently with different heuristic criteria;

5. a parallel lookahead technique, which allows the concurrent computation of the heuristic function values;

6. a technique for the parallelization of the model checking phase.

These techniques were implemented into the state-of-the-art ASP system DLV, using a multi-threaded approach, thus obtaining a *parallel instantiator* and a *parallel model generator*.

An extensive experimental analysis were carried out considering problem instances commonly employed for benchmarking ASP systems. More in detail, a scalability analysis was performed for the parallel instantiator in order to assess its capability of handling a large amount of input data; and, a set of heterogeneous benchmarking problems were used for assessing the parallel model generator performances and verifying the flexibility of the approach. The experimental results confirmed the efficacy of the proposed techniques.

The remainder of this thesis is structured as follows: in Chapter 2 the ASP framework is introduced, including a description of syntax and semantics, and a discussion of its usage as a tool for knowledge representation; in Chapter 3 we discuss the existing ASP program serial evaluation techniques, in particular, we detail the ones exploited in the ASP system DLV; the main contribution of the thesis is presented in Chapter 4, where the parallel instantiation techniques are presented; then, in Chapter 5, the parallel techniques for the propositional search phase are described; implementation issues and experimental results are illustrated in Chapter 6; finally, related works are discussed in Chapter 7 and conclusions are drawn in Chapter 8.

Chapter 2

Answer Set Programming

In this chapter we introduce the basics of Answer Set Programming (ASP). The chapter is structured as follows: in Section 2.1 we describe the syntax of the language extended with aggregate functions, and in Section 2.2 we present the semantics; in Section 2.3 we illustrate some relevant subclasses of ASP programs; finally, in Section 2.4 the usege of ASP as a tool for Knowledge Representation (KR) is described.

2.1 Syntax

Let $\mathcal{V}$ be a set of *variables*, $\mathcal{C}$ be a set of *constants*, and $\mathcal{S}$ be a set of *predicates symbols*. Hereafter in the remainder of this thesis we assume variables to be strings starting with uppercase letters and constants to be non-negative integers or strings starting with lowercase letters. Moreover, predicates are strings starting with lowercase letters. An *arity* (non-negative integer) is associated with each predicate. Moreover, the language allows the use of built-in predicates (i.e., predicates with a fixed meaning) for the common arithmetic operations (i.e., $=$, $\leq$, $\geq$, $+$, $\times$, etc.; usually written in infix notation).

A variable or a constant is a *term*. A *standard atom* is an expression $p(t_1, \ldots, t_n)$, where p is a *predicate* of arity n and $t_1, \ldots, t_n$ are terms. An atom $p(t_1, \ldots, t_n)$ is ground if $t_1, \ldots, t_n$ are constants.

A *set term* is either a symbolic set or a ground set. A *symbolic set* is a pair $\{\mathit{Terms} : \mathit{Conj}\}$, where *Terms* is a list of terms (variables or constants) and *Conj* is a conjunction of standard atoms, that is, *Conj* is of the form $a_1, \ldots, a_k$ and each a_i $(1 \leq i \leq k)$ is a standard atom.

Example 2.1.1. For example, consider the set term

$$\{X : p(X, a), q(X)\}.$$

Assume that we know the ground atoms $p(1, a)$, $p(2, a)$, $q(1)$ and $q(2)$ to be true.

The set term stands for the set of all the possible values of the variable X such that the conjunction $a(X, a), p(X)$ is true, i.e., $\{X \,|\, p(X, a)$ *and* $q(X)$ *are* *true*}. In this case, the possible values are

$$\{X : p(X, a), q(X)\} \quad X \in \{1, 2\}.$$

□

A *ground set* is a set of pairs of the form $\langle consts : conj \rangle$, where *consts* is a list of constants and *conj* is a conjunction of ground standard atoms.

We define an *aggregate function* an expression of the form $f(S)$, where S is a set term, and f is an *aggregate function symbol.*

Intuitively, an aggregate function can be thought of as a (possibly partial) function mapping multisets of constants to a constant. Throughout the rest of the thesis, we will adopt the notation of the DLV system [7] for representing aggregates.

Example 2.1.2. Below is a list of the most common aggregate function; all these functions consider the element of the set term as integers.

- #`min`: The function identifies the minimal term among the elements of the set term. This function is undefined for the empty set.
- #`max`: The function identifies the maximal term among the elements of the set term. This function is undefined for the empty set.
- #`count`: The function determines the number of elements of the set term.
- #`sum`: The function determines the sum of the elements of the set term.
- #`times`, The function determines the product of the elements of the set term.
- #`avg`, The function determines the average value of the elements of the set term. This function is undefined for the empty set.

□

An *aggregate atom* is a structure of the form $f(S) \prec T$, where $f(S)$ is an aggregate function, $\prec \in \{<, \leq, >, \geq\}$ is a comparison operator, and T is a term (variable or constant). An aggregate atom $f(S) \prec T$ is ground if T is a constant and S is a ground set.

Example 2.1.3. The following are aggregate atoms in DLV notation:

$$\#\mathtt{max}\{X : p(X, Y), q(X)\} > Z$$
$$\#\mathtt{max}\{\langle 2 : p(2, a), q(2)\rangle, \langle 2 : p(2, b), q(2)\rangle\} > 1$$

□

A *literal* is either (i) a standard atom, or (ii) a standard atom preceded by the *negation as failure* symbol `not`, or (iii) an aggregate atom. Complementary standard literals are of the form a and `not` a, where a is a standard atom. For a standard literal ℓ, we denote by $\neg.\ell$ the complement of ℓ. With a little abuse of notation, if L is a set of standard literals, we denote with $\neg.L$ the set $\{\neg.\ell \mid \ell \in L\}$.

Example 2.1.4. Example of literals are $person(joe)$, $father(joe, john)$, $\texttt{not}\ father(joe, joseph)$, $\#\texttt{max}\{X : age(X, Y), person(Y)\} < 18$.

□

A *rule* r is a construct of the form

$$a_1 \ \vee \cdots \vee \ a_n \text{ :- } \ell_1, \ \cdots, \ \ell_m.$$

where $a_1 \ \cdots \ a_n$ are standard atoms, $\ell_1, \ \ldots, \ \ell_m$ are literals, $n \geq 0$, and $m \geq 0$. The disjunction $a_1 \ \vee \cdots \vee \ a_n$ is referred to as the *head* of r, and the conjunction $\ell_1, \ \ldots, \ \ell_m$ as the *body* of r. If the body is empty ($m = 0$) then the rule is called *fact*. If the head is empty ($n = 0$) the the rule is called *integrity constraint* (or simply *constraint*). We denote the set of head atoms by

$$H(r) = \{a_1, \ \ldots, \ a_n\},$$

and the set of body literals by

$$B(r) = \{\ell_1, \ \ldots, \ \ell_m\}.$$

Moreover, the set of positive standard body literals is denoted by $B^+(r)$, the set of negative standard body literals by $B^-(r)$, and the set of aggregate body literals by $B^{\mathcal{A}}(r)$. A rule r is ground if all the literals in $H(r)$ and in $B(r)$ are ground. A *program* is a set of rules. A program is ground if all its rules are ground. Accordingly with the database terminology, a predicate occurring only in facts is referred to as an *EDB* predicate, all others as *IDB* predicates; the set of facts of P is denoted by $EDB(P)$.

The variables of a rule can be *local* or *global*. A *local* variable of a rule r is a variable appearing solely in sets terms of r; otherwise the variable is stated as *global*. A rule r is *safe* if both the following conditions hold: (i) for each global variable X of r there is a positive standard literal $\ell \in B^+(r)$ such that X appears in ℓ; (ii) each local variable of r appearing in a symbolic set $\{Terms : Conj\}$ also appears in $Conj$. Note that condition (i) is the standard safety condition adopted in LP to guarantee that the variables are range restricted [31], while condition (ii) is specific for aggregates. A program is safe if all its rules are safe.

Example 2.1.5. Consider the following rules:

$$\begin{array}{l}
person(X) \text{ :- } father(X, Y). \\
animal(X) \ \vee mineral(X) \text{:- } \texttt{not}\ plant(X). \\
p(X) \text{ :- } q(X, Y, V), \ \#\texttt{max}\{Z : r(Z), \ a(Z, V)\} > Y. \\
p(X) \text{ :- } q(X, Y, V), \ \#\texttt{sum}\{Z : r(X), \ a(X, S)\} > Y.
\end{array}$$

The first rule is safe, while the second is not because it violates condition (i) due to variable X. The third rule is safe, while the fourth is not because variable Z violates condition (ii). □

2.2 Semantics

In this section we describe the semantics of ASP programs, which is defined in terms of the set of its answer Sets. Answers Sets are defined only for ground programs. To determine the answer sets of a non-ground program $\mathcal{P}$, the same undergoes a process known as *program instantiation* which produces a ground program $\mathcal{P}$' semantically equivalent to $\mathcal{P}$. Before describing the instantiation process we introduce some basic notions.

Given an ASP program $\mathcal{P}$, the *universe* of $\mathcal{P}$, denoted by $U_{\mathcal{P}}$, is the set of constants appearing in $\mathcal{P}$. The *base* of $\mathcal{P}$, denoted by $B_{\mathcal{P}}$, is the set of standard atoms constructible from predicates of $\mathcal{P}$ with constants in $U_{\mathcal{P}}$.

A *substitution* is a mapping from a set of variables to $U_{\mathcal{P}}$. We define a *global substitution* for a rule r a substitution from the set of the global variables of r to $U_{\mathcal{P}}$; we define a *local substitution* for a rule r the substitution from the set of local variables of r to $U_{\mathcal{P}}$. Given a set term without global variables $S = \{\mathit{Terms}\colon \mathit{Conj}\}$, the *instantiation of* S is the following ground set:

$$\mathit{inst}(S) = \{\langle \sigma(\mathit{Terms}) : \sigma(\mathit{Conj})\rangle \mid \sigma \text{ is a local substitution for } S\}.$$

A *ground instance* of a rule r is obtained in two steps: First, a global substitution σ for r is applied, and then every set term S in $r\sigma$ is replaced by its instantiation $\mathit{inst}(S)$. The instantiation $\mathit{Ground}(\mathcal{P})$ of a program $\mathcal{P}$ is the set of instances of all the rules in $\mathcal{P}$.

Example 2.2.1. Consider the following program $\mathcal{P}_1$:

$$\begin{array}{ll} q(1) \text{ :- } \mathtt{not}\ p(2,2). & q(2) \text{ :- } \mathtt{not}\ p(2,1). \\ p(2,2) \text{ :- } \mathtt{not}\ q(1). & p(2,1) \text{ :- } \mathtt{not}\ q(2). \\ \multicolumn{2}{l}{t(X) \text{ :- } q(X),\ \#\mathtt{sum}\{Y : p(X,Y)\} > 1.} \end{array}$$

The instantiation $\mathit{Ground}(\mathcal{P}_1)$ of $\mathcal{P}_1$ is the following program:

$$\begin{array}{ll} q(1) \text{ :- } \mathtt{not}\ p(2,2). & q(2) \text{ :- } \mathtt{not}\ p(2,1). \\ p(2,2) \text{ :- } \mathtt{not}\ q(1). & p(2,1) \text{ :- } \mathtt{not}\ q(2). \\ \multicolumn{2}{l}{t(1) \text{ :- } q(1),\ \#\mathtt{sum}\{\langle 1 : p(1,1)\rangle,\ \langle 2 : p(1,2)\rangle\} > 1.} \\ \multicolumn{2}{l}{t(2) \text{ :- } q(2),\ \#\mathtt{sum}\{\langle 1 : p(2,1)\rangle,\ \langle 2 : p(2,2)\rangle\} > 1.} \end{array}$$

$\square$

Given a set X, let $\overline{2}^X$ denote the set of all multisets over elements from X. The domain of an aggregate function is the set of multisets on which the function is defined. Without loss of generality, we assume that aggregate functions map to $\mathbb{Z}$ (the set of integers).

Example 2.2.2. Let us look at common domains for the aggregate functions of Example 2.1.2: #`count` is defined over $\overline{2}^{U_{\mathcal{P}}}$, #`sum` and #`times` over $\overline{2}^{\mathbb{Z}}$, #`min`, #`max` and #`avg` over $\overline{2}^{\mathbb{Z}} \setminus \{\emptyset\}$. $\square$

An *interpretation* I for an ASP program $\mathcal{P}$ is a consistent set of standard ground literals, that is, $I \subseteq B_{\mathcal{P}} \cup \neg.B_{\mathcal{P}}$ and $I \cap \neg.I = \emptyset$. We denote by I^+ and I^- the set of standard positive and negative literals occurring in I, respectively. An interpretation can be *total* or *partial*: it is *total* if $I^+ \cup \neg.I^- = B_{\mathcal{P}}$; otherwise I is *partial*. The set of all the interpretations of $\mathcal{P}$ is denoted by $\mathcal{I}_{\mathcal{P}}$.

The evaluation of a standard literal ℓ with respect to an interpretation I results in one of the following alternatives: (i) if $\ell \in I$, then ℓ is true with respect to I; (ii) if $\neg.\ell \in I$, then ℓ is false with respect to I; (iii) otherwise, if $\ell \notin I$ and $\neg.\ell \notin I$, then ℓ is undefined with respect to I.

Set terms, aggregate functions and aggregate literals can be also evaluated with respect to an interpretation, giving rise to a multiset, a value, and a truth value, respectively. We first consider a total interpretation I. The evaluation $I(S)$ of a set term S with respect to I is the multiset $I(S)$ defined as follows: Let $S^I = \{\langle t_1, ..., t_n\rangle \mid \langle t_1, ..., t_n : Conj\rangle \in S$ and all the atoms in $Conj$ are true with respect to $I\}$; $I(S)$ is the multiset obtained as the projection of the tuples of S_I on their first constant, that is, $I(S) = [t_1 \mid \langle t_1, ..., t_n\rangle \in S^I]$. The evaluation $I(f(S))$ of an aggregate function $f(S)$ with respect to I is the result of the application of f on $I(S)$.[1] If the multiset $I(S)$ is not in the domain of f, then $I(f(S)) = \bot$ (where $\bot$ is a fixed symbol not occurring in $\mathcal{P}$). A ground aggregate literal $\ell = f(S) \prec k$ is true with respect to I if both $I(f(S)) \neq \bot$ and $I(f(S)) \prec k$ hold; otherwise, ℓ is false.

Example 2.2.3. Let I_1 be a total interpretation having $I_1^+ = \{p(3), q(2,3), q(3,3), q(4,3)r(2), r(3)\}$. Assuming that all variables are local, we can check that:

- $\#\texttt{count}\{Y : q(X,Y)\} < 2$ is false; indeed, if S_1 is the corresponding ground set, then $S_1^{I_1} = \{\langle 3\rangle\}$, $I_1(S_1) = [3]$ and $\#\texttt{count}([3]) = 1$.
- $\#\texttt{count}\{X,Y : q(X,Y)\} < 2$ is true; indeed, if S_2 is the corresponding ground set, then $S_2^{I_1} = \{\langle 2,3\rangle, \langle 3,3\rangle, \langle 4,3\rangle\}$, $I_1(S_2) = [2,3,4]$ and $\#\texttt{count}([2,3,4]) = 3$.
- $\#\texttt{times}\{X : q(X,Y), p(Y)\} < 25$ is true; indeed, if S_3 is the corresponding ground set, then $S_3^{I_1} = \{\langle 2\rangle, \langle 3\rangle, \langle 4\rangle\}$, $I_1(S_3) = [2,3,4]$ and $\#\texttt{times}([2,3,4]) = 24$.
- $\#\texttt{times}\{Y,X : q(X,Y), p(Y)\} < 24$ is false; indeed, if S_3 is the corresponding ground set, then $S_3^{I_1} = \{\langle 3,2\rangle, \langle 3,3\rangle, \langle 4,3\rangle\}$, $I_1(S_3) = [3,3,3]$ and $\#\texttt{times}([3,3,3]) = 27$.
- $\#\texttt{sum}\{X : q(X,Y), r(X)\} <= 5$ is true; indeed, if S_4 is the corresponding ground set, then $S_4^{I_1} = \{\langle 2,3\rangle, \langle 3,3\rangle\}$, $I_1(S_4) = [2,3]$ and $\#\texttt{sum}([2,3]) = 5$.
- $\#\texttt{sum}\{X,Y : q(X,Y), r(Y)\} <= 5$ is false; indeed, if S_5 is the corresponding ground set, then $S_5^{I_1} = \{\langle 2,3\rangle, \langle 3,3\rangle, \langle 4,3\rangle\}$, $I_1(S_5) = [2,3,4]$ and $\#\texttt{sum}([2,3,4]) = 9.$;
- $\#\texttt{min}\{X,Y : q(X,Y), r(Y)\} <= 5$ is true; indeed, if S_5 is the corresponding ground set, then $S_5^{I_1} = \{\langle 2,3\rangle, \langle 3,3\rangle, \langle 4,3\rangle\}$, $I_1(S_5) = [2,3,4]$ and $\#\texttt{min}([2,3,4]) = 2.$;
- $\#\texttt{min}\{X : f(X), h(X)\} >= 2$ is false; indeed, if S_6 is the corresponding ground set, then $S_6^{I_1} = \emptyset$, $I_1(S_6) = \emptyset$, and $I_1(\#\texttt{min}(\emptyset)) = \bot$ (we recall that $\emptyset$ is not in the domain of $\#\texttt{min}$).

□

We now consider a partial interpretation I and refer to an interpretation J such that $I \subseteq J$ as an *extension* of I. If a ground aggregate literal ℓ is true (resp. false) with respect to *each* total interpretation J extending I, then ℓ is true (resp. false) with respect to I; otherwise, ℓ is undefined.

[1]In this thesis, we only consider aggregate functions the value of which is polynomial-time computable with respect to the input multiset.

Example 2.2.4. Let S_7 be the ground set in the literal $\ell_1 = \#\texttt{sum}\{\langle 1 : p(2,1)\rangle, \langle 2 : p(2,2)\rangle\} > 1$, and consider a partial interpretation $I_2 = \{p(2,2)\}$. Since each total interpretation extending I_2 contains either $p(2,1)$ or $\texttt{not}\ p(2,1)$, we have either $I_2(S_7) = [2]$ or $I_2(S_7) = [1,2]$. Thus, the application of $\#\texttt{sum}$ yields either $2 > 1$ or $3 > 1$, and thus ℓ_1 is true with respect to I_2. □

Remark 1. Observe that our definitions of interpretation and truth values preserve "knowledge monotonicity": If an interpretation J extends I (i.e., $I \subseteq J$), each literal which is true with respect to I is true with respect to J, and each literal which is false with respect to I is false with respect to J as well.

An interpretation I satisfies a rule r if at least one of the following conditions is satisfied: (i) $H(r)$ is true with respect to I; (ii) some literal in $B(r)$ is false with respect to I; (iii) all the atoms in $H(r)$ and some literal in $B(r)$ are undefined with respect to I. An interpretation M is a *model* of an ASP program $\mathcal{P}$ if all the rules r in $Ground(\mathcal{P})$ are satisfied with respect to M. A model M for $\mathcal{P}$ is (subset) minimal if no model N for $\mathcal{P}$ exists such that $N^+ \subset M^+$. Note that, under these definitions, the word *interpretation* refers to a possibly partial interpretation, while a *model* is always a total interpretation.

Example 2.2.5. Consider again the program $\mathcal{P}_1$ of Example 2.2.1. Let I_3 be a total interpretation for $\mathcal{P}_1$ such that $I_3^+ = \{q(2), p(2,2), t(2)\}$. Then I_3 is a minimal model of $\mathcal{P}_1$. □

Definition 1 ([32])**.** Given a ground ASP program $\mathcal{P}$ and a total interpretation I, let $\mathcal{P}^I$ denote the transformed program obtained from $\mathcal{P}$ by deleting all rules in which a body literal is false w.r.t. I. I is an *answer set* of a program $\mathcal{P}$ if it is a minimal model of $Ground(\mathcal{P})^I$.

The set of all answer sets of a program $\mathcal{P}$ will be denoted in the remainder of this thesis with $ANS(\mathcal{P})$.

Example 2.2.6. Consider the following two programs:

$$\mathcal{P}_2\text{: } \{p(a) \text{:-} \#\texttt{count}\{X : p(X)\} > 0.\}$$
$$\mathcal{P}_3\text{: } \{p(a) \text{:-} \#\texttt{count}\{X : p(X)\} < 1.\}$$

then, we have that

$$Ground(\mathcal{P}_2) = \{p(a) \text{:-} \#\texttt{count}\{\langle a : p(a)\rangle\} > 0.\}$$
$$Ground(\mathcal{P}_3) = \{p(a) \text{:-} \#\texttt{count}\{\langle a : p(a)\rangle\} < 1.\}$$

and two interpretation $I_1 = \{p(a)\}$, $I_2 = \emptyset$. Then, $Ground(P_1)^{I_1} = Ground(P_1)$, $Ground(P_1)^{I_2} = \emptyset$, and $Ground(P_2)^{I_1} = \emptyset$, $Ground(P_2)^{I_2} = Ground(P_2)$ hold.

I_2 is the only answer set of P_1 (because I_1 is not a minimal model of $Ground(P_1)^{I_1}$), while P_2 admits no answer set (I_1 is not a minimal model of $Ground(P_2)^{I_1}$, and I_2 is not a model of $Ground(P_2) = Ground(P_2)^{I_2}$). □

Note that any answer set A of $\mathcal{P}$ is also a model of $\mathcal{P}$ because $Ground(\mathcal{P})^A \subseteq Ground(\mathcal{P})$, and rules in $Ground(\mathcal{P}) - Ground(\mathcal{P})^A$ are satisfied w.r.t. A.

Another possible characterization is given by the notion of *supportedness.* Given an interpretation I for a ground program $\mathcal{P}$, we say that a ground atom A is *supported* in I if there exists a *supporting* rule r in the ground instantiation

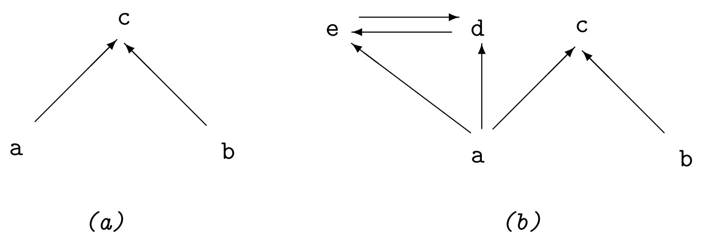

Figure 2.1: *Graphs (a) $G(\mathcal{P}_4)$, and (b) $G(\mathcal{P}_5)$*

of $\mathcal{P}$ such that the body of r is true w.r.t. I and A is the only true atom in the head of r.

Proposition 2.2.7. *[33, 34, 10]* If M is an answer set of a program $\mathcal{P}$, then all atoms in M are supported.

2.3 Relevant Sub-Classes

In this section, we describe relevant syntactic sub-classes of ASP programs. The definition of *dependency graph* is given first; then we define the class of *stratified* programs, and the class of *head-cycle free* programs.

Dependency Graph With every ground program $\mathcal{P}$, we associate a directed graph $G(\mathcal{P}) = (N, E)$, called the *dependency graph* of $\mathcal{P}$, where (i) the nodes in N are the atoms of $\mathcal{P}$, (ii) there is an arc in E from a node a to a node b iff there is a rule r in $\mathcal{P}$ such that b appears in the head of r and a appears in the positive body of r.

The graph $G(\mathcal{P})$ singles out the dependencies of the head atoms of a rule r from the positive atoms in its body. Negative literals cause no arc in $G(\mathcal{P})$.

Example 2.3.1. Consider the following two programs:

$$\mathcal{P}_4 = \{a \vee b.\;\; c{:-}a.\;\; c{:-}b.\}$$
$$\mathcal{P}_5 = \mathcal{P}_4 \cup \{d \vee e{:-}a.\;\; d{:-}e.\;\; e{:-}d, \texttt{not}\ b.\}$$

The dependency graph $G(\mathcal{P}_4)$ is depicted in Figure 2.1 (a), while the dependency graph $G(\mathcal{P}_5)$ is depicted in Figure 2.1 (b).

□

Stratified Programs. We introduce now the class of *stratified* ASP programs.

Definition 2. Functions from $\| \; \| : B_{\mathcal{P}} \rightarrow \{0, 1, \dots\}$ from the ground set of literal $B_{\mathcal{P}}$ to finite ordinals are called level mappings of P.

Level mapping is used now for defining (locally) stratified programs.

Definition 3. A disjunctive ASP program $\mathcal{P}$ is called *(locally) stratified* [35, 36] if there is a level mapping$\| \ \|_s$ of $\mathcal{P}$ such that, for every rule r of $Ground(\mathcal{P})$,

(1) for any $l \in B^+(r)$, and for any $l' \in H(r)$, $\| l\|_s \leq \| l'\|_s$;

(2) for any $l \in B^-(r)$, and for any $l' \in H(r)$, $\| l\|_s < \| l'\|_s$;

(3) for any $l,\ l' \in H(r)$, $\| l\|_s = \| l'\|_s$.

Example 2.3.2. Consider the following two programs.

$\mathcal{P}_6$: $p(a) \lor p(c)$:− `not` $q(a)$.
$p(b)$:− `not` $q(b)$.

$\mathcal{P}_7$: $p(a) \lor p(c)$:− `not` $q(b)$.
$q(b)$:− `not` $q(a)$.

It is easy to see that program $\mathcal{P}_6$ is stratified, while program $\mathcal{P}_7$ is not. A suitable level mapping for $\mathcal{P}_6$ is the following:

$$\|p(a)\|_s = 2 \ \|p(b)\|_s = 2 \ \|p(c)\|_s = 2 \ \|q(a)\|_s = 1 \ \|q(b)\|_s = 1 \ \|q(c)\|_s = 1$$

As for $\mathcal{P}_6$, an admissible level mapping would need to satisfy $\|p(a)\|_s < \|q(b)\|_s$ and $\|q(b)\|_s < \|p(a)\|_s$, which is impossible. □

An important property of locally stratified ASP programs is given by the following proposition.

Proposition 2.3.3. *[6] A locally stratified normal (non-disjunctive) ASP programs has at most one answer set.*

It is worthwhile noting that the presence of disjunction invalidates Proposition 2.3.3. Indeed, the program $\{a \lor b.\}$ has two answer sets, namely $\{a\}$ and $\{b\}$.

Head-Cycle Free Programs. Another relevant property of disjunctive ASP programs is head-cycle freeness (HCF) [37].

The dependency graphs allow us to define HCF programs. [37]. A program $\mathcal{P}$ is *HCF* iff there is no rule r in $\mathcal{P}$ such that two atoms occurring in the head of r occur in a single cycle of $G(\mathcal{P})$.

Example 2.3.4. The dependency graphs given in Figure 2.1 reveal that program $\mathcal{P}_4$ of Example 2.3.1 is HCF and that program $\mathcal{P}_5$ is not HCF, as rule $d \lor e$:−a contains in its head two atoms belonging to the same cycle of $G(\mathcal{P}_5)$.
□ □

It has been shown that HCF programs are computationally easier than general (non-HCF) programs.

Proposition 2.3.5. *[37, 2]* Deciding whether an atom belongs to some answer set of a ground HCF program $\mathcal{P}$ is NP-complete, while deciding whether an atom belongs to some answer set of a ground (non-HCF) program $\mathcal{P}$ is Σ_2^P-complete.

2.4 Knowledge Representation

Answer Set Programming is employed as a tool for knowledge representation and common sense reasoning in several application domains, ranging form classical deductive databases to artificial intelligence. ASP is particular suitable for handling incomplete knowledge and non-monotonic reasoning, and allows for encoding problems in a declarative fashion. Thanks to this approach, writing an ASP program is as easy as describing the problem domain, while the complexity of the reasoning task is hidden by using a dedicated ASP system. In addition, the (optional) separation of a fixed non-ground program from an input database allows one to obtain uniform solutions over varying instances.

ASP is a powerful formalisms, and allows complex problems to be expressed; its expressive power captures all problems belonging to the second level of the polynomial hierarchy (the complexity class Σ_2^P). This high expressive power is significantly relevant for approaching hard problems; for example, in solving planning and diagnosis problems, or, in the field of Artificial Intelligence, for solving problems not reducible to SAT instances.

ASP allows the encoding of problems in an intuitive and concise fashion following a "Guess&Check" programming methodology (originally introduced in [38] and refined in [7]). According to this approach a program $\mathcal{P}$ which encodes a problem $\mathbf{P}$ consists of the following parts:

Input Instance: An instance F of the problem $\mathbf{P}$ is specified in input using a database of facts.

Guess Part: A set of disjunctive rules $G \subseteq \mathcal{P}$, referred to as the "guessing part", is used the define the search space .

Check Part: The search space is then pruned by the "checking part", consisting of a set of constraints $C \subseteq \mathcal{P}$ which impose some properties to be verified.

Basically, the first two parts of the program, that is, the input instance and the guessing part, represent the "candidate solutions" to the problem. By adding the check part those solutions are filtered in order to guarantee that the answer sets of the resulting program represent exactly the admissible solutions for the input instance. The following example represents the typical application of the Guess&Check methodology.

Example 2.4.1. Suppose that we want to partition a set of people into two groups, but we also know that some pairs of people dislike each other, thus we have to keep those two in different groups. Assume that the input instance consists of the following facts:

$$person(bob).\ person(eve).\ dislike(bob, eve).$$

So as, applying the guess&check methodology, the guess part would model the possible assignments of persons to groups:

$$group(P, 1) \lor group(P, 2) \text{ :- } person(P).$$

The resulting program (input instance + guess) produces the following answer set:

$\{person(bob), person(eve), dislike(bob, eve), group(bob, 1), group(eve, 1)\}$
$\{person(bob), person(eve), dislike(bob, eve), group(bob, 1), group(eve, 2)\}$
$\{person(bob), person(eve), dislike(bob, eve), group(bob, 2), group(eve, 1)\}$
$\{person(bob), person(eve), dislike(bob, eve), group(bob, 2), group(eve, 2)\}$

However, we want to discard assignments in which people that dislike each other belong to the same group. To this end, we add the checking part by writing the following constraint:

$$:- \ group(P1, G),\ group(P2, G),\ dislike(P1, P2).$$

Now, adding the constraint to the original program allows us to obtain the intended answer sets, as the checking part acted as a sort of filter:

$\{person(bob), person(eve), dislike(bob, eve), group(bob, 1), group(eve, 2)\}$
$\{person(bob), person(eve), dislike(bob, eve), group(bob, 2), group(eve, 1)\}$

□

In the following, we illustrate the use of ASP as a tool for knowledge representation by example. More in details, we present a number of problems that can be naturally encoded using ASP: the first one is a problem taken from classic deductive database applications; the others are well-known hard problems that can be solved applying the "Guess&Check" programming style.

Reachability. Given a finite directed graph $G = (V, A)$, we want to compute all pairs of nodes $(a, b) \in V \times V$ such that b is reachable from a through a nonempty sequence of arcs in A. In different terms, the problem amounts to computing the transitive closure of the relation A.

The input graph is encoded by assuming that A is represented by the binary predicate $arc(X, Y)$, where a fact $arc(a, b)$ means that G contains an arc from a to b, i.e., $(a, b) \in A$; whereas, the set of nodes V is not explicitly represented, since the nodes appearing in the transitive closure are implicitly given by these facts.

The following program then defines a predicate $reachable(X, Y)$ containing all facts $reachable(a, b)$ such that b is reachable from a through the arcs of the input graph G:

$$r_1\colon\ reachable(X, Y) :- arc(X, Y).$$
$$r_2\colon\ reachable(X, Y) :- arc(X, U),\ reachable(U, Y).$$

The first rule states that that node Y is reachable from node X if there is an arc in the graph from X to Y, whereas the second rule represents the transitive closure by stating that node Y is reachable from node X if there is a node U such that U is directly reachable from X (there is an arc from X to U) and Y is reachable from U.

As an example, consider a graph represented by the following facts:

$$arc(1,2). \quad arc(2,3). \quad arc(3,4).$$

The single answer set of the program reported above together with these three facts program is $\{reachable(1,2), reachable(2,3), reachable(3,4), reachable(1,3), reachable(2,4), reachable(1,4), arc(1,2), arc(2,3), arc(3,4)\}$. The first three reported literals are inferred by exploiting the rule r_1, whereas the other literals containing the predicate *reachable* are inferred by using rule r_2.

Hamiltonian Path. Given a finite directed graph $G = (V, A)$ and a node $a \in V$ of this graph, does there exist a path in G starting at a and passing through each node in V exactly once?

This is a classical NP-complete problem in graph theory. Suppose that the graph G is specified by using facts over predicates *node* (unary) and *arc* (binary), and the starting node a is specified by the predicate *start* (unary). Then, the following program $\mathcal{P}_{hp}$ solves the *Hamiltonian Path* problem:

r_1: $inPath(X,Y) \ \vee \ outPath(X,Y) \text{:-} arc(X,Y).$
r_2: $reached(X) \text{:-} start(X).$
r_3: $reached(X) \text{:-} reached(Y), inPath(Y,X).$
r_4: $\text{:-} inPath(X,Y),\ inPath(X,Y1), Y <> Y1.$
r_5: $\text{:-} inPath(X,Y),\ inPath(X1,Y), X <> X1.$
r_6: $\text{:-} node(X),$ `not` $reached(X),$ `not` $start(X).$

The disjunctive rule (r_1) guesses a subset S of the arcs to be in the path, while the rest of the program checks whether S constitutes a Hamiltonian Path. Here, an auxiliary predicate *reached* is defined, which specifies the set of nodes which are reached from the starting node. Doing this is very similar to reachability, but the transitivity is defined over the guessed predicate *inPath* using rule r_3. Note that *reached* is completely determined by the guess for *inPath*, no further guessing is needed.

In the checking part, the first two constraints (namely, r_4 and r_5) ensure that the set of arcs S selected by *inPath* meets the following requirements, which any Hamiltonian Path must satisfy: (i) there must not be two arcs starting at the same node, and (ii) there must not be two arcs ending in the same node. The third constraint enforces that all nodes in the graph are reached from the starting node in the subgraph induced by S.

Let us next consider an alternative program $\mathcal{P}'_{hp}$, which also solves the *Hamiltonian Path* problem, but intertwines the reachability with the guess:

r_1: $inPath(X,Y) \ \vee \ outPath(X,Y) \text{:-} reached(X),\ arc(X,Y).$
r_2: $inPath(X,Y) \ \vee \ outPath(X,Y) \text{:-} start(X),\ arc(X,Y).$
r_3: $reached(X) \text{:-} inPath(Y,X).$
r_4: $\text{:-} inPath(X,Y),\ inPath(X,Y1), Y <> Y1.$
r_5: $\text{:-} inPath(X,Y),\ inPath(X1,Y), X <> X1.$
r_6: $\text{:-} node(X),$ `not` $reached(X),$ `not` $start(X).$

Here, the two disjunctive rules (r_1 and r_2), together with the auxiliary rule r_3, guess a subset S of the arcs to be in the path, while the rest of the program checks whether S constitutes a Hamiltonian Path. Here, *reached* is defined in a different way. In fact, *inPath* is already defined in a way that only arcs reachable from the starting node will be guessed. The remainder of the checking part is the same as in $\mathcal{P}_{hp}$.

Ramsey Numbers. In the previous example, we have seen how a search problem can be encoded in an ASP program whose answer sets correspond to the problem solutions. We now build a program whose answer sets witness that a property does not hold, i.e., the property at hand holds if and only if the program has no answer set. We next apply the above programming scheme to a well-known problem of number and graph theory.

The Ramsey number $R(k, m)$ is the smallest integer n such that, no matter how we colour the arcs of the complete undirected graph (clique) with n nodes using two colours, say red and blue, there is a red clique with k nodes (a red k-clique) or a blue clique with m nodes (a blue m-clique).

Ramsey numbers exist for all pairs of positive integers k and m [39]. We next show a program $\mathcal{P}_{ra}$ that allows us to decide whether a given integer n is *not* the Ramsey Number $R(3, 4)$. By varying the input number n, we can determine $R(3, 4)$, as described below. Let $\mathcal{F}_{ra}$ be the collection of facts for input predicates *node* and *arc* encoding a complete graph with n nodes. $\mathcal{P}_{ra}$ is the following program:

r_1: $blue(X, Y) \;\vee\; red(X, Y) \text{:-} arc(X, Y).$
r_2: $\text{:-}\; red(X, Y),\; red(X, Z),\; red(Y, Z).$
r_3: $\text{:-}\; blue(X, Y),\; blue(X, Z),\; blue(Y, Z),\; blue(X, W),\; blue(Y, W),$
$blue(Z, W).$

Intuitively, the disjunctive rule r_1 guesses a colour for each edge. The first constraint (r_2) eliminates the colourings containing a red clique (i.e., a complete graph) with 3 nodes, and the second constraint (r_3) eliminates the colourings containing a blue clique with 4 nodes. The program $\mathcal{P}_{ra} \cup \mathcal{F}_{ra}$ has an answer set if and only if there is a colouring of the edges of the complete graph on n nodes containing no size 3 red clique and no size 4 blue clique. Thus, if there is an answer set for a specific n, then n is *not* $R(3, 4)$, that is, $n < R(3, 4)$. On the other hand, if $\mathcal{P}_{ra} \cup \mathcal{F}_{ra}$ has no answer set, then $n \geq R(3, 4)$. Thus, the smallest n such that no answer set is found is the Ramsey number $R(3, 4)$.

Team Building A project team has to be built from a set of employees according to the following specifications

(p_1) The team consists of a certain number of employees.

(p_2) At least a given number of different skills must be present in the team.

(p_3) The sum of the salaries of the employees working in the team must not exceed the given budget.

(p_3) The salary of each individual employee is within a specified limit.

(p_3) The number of women working in the team has to reach at least a given number.

Suppose that our employees are provided by a number of facts of the form $emp(EmpId, Sex, Skill, Salary)$; the size of the team, the minimum number of different skills, the budget, the maximum salary, and the minimum number of women are specified by the facts $nEmp(N)$, $nSkill(N)$, $budget(B)$, $maxSal(M)$, and $women(W)$. We then encode each property p_i above by an aggregate atom A_i , and enforce it by an integrity constraint containing $\mathtt{not} A_i$.

r_1: $in(I) \vee out(I)$:−$emp(I, Sx, Sk, Sa)$.
r_2: :−$nEmp(N), \mathtt{not}\#\mathtt{count}\{I : in(I)\} = N$.
r_3: :−$nSkill(M), not\#\mathtt{count}\{Sk : emp(I, Sx, Sk, Sa), in(I)\} \geq M$.
r_4: :−$budget(B), not\#\mathtt{sum}\{Sa, I : emp(I, Sx, Sk, Sa), in(I)\} \leq B$.
r_5: :−$maxSal(M), not\#\mathtt{max}\{Sa : emp(I, Sx, Sk, Sa), in(I)\} \leq M$.
r_6: :−$women(W), not\#\mathtt{count}\{I : emp(I, f, Sk, Sa), in(I)\} \geq W$.

Intuitively, the disjunctive rule “guesses” whether an employee is included in the team or not, while the five constraints correspond one-to-one to the five requirements p_1-p_5 . Thanks to the aggregates the translation of the specifications is surprisingly straightforward. The example highlights the usefulness of representing both sets and multisets in our language; the latter can be obtained by specifying more than one variable in $Vars$ of a symbolic set $\{Vars : Conj\}$. For instance, the encoding of p_2 requires a set, as we want to count different skills; two employees in the team having the same skill, should count once w.r.t. p_2 . On the contrary, p_3 requires summing the elements of a multiset; if two employees have the same salary, both salaries should be summed for p_3 . This is obtained by adding the variable I, which uniquely identifies every employee, to $Vars$. The valuation of $\{Sa, I : emp(I, Sx, Sk, Sa), in(I)\}$ yields the set $S = \{Sa, I : Sais\ the\ salary\ of\ employee\ I\ in\ the\ team\}$. The sum function is then applied on the multiset of the first components S_a of the tuples $\langle S_a, I\rangle$ in S.

Strategic Companies In the examples considered so far, the complexity of the problems is located at most on the first level of the Polynomial Hierarchy [40] (in NP or co-NP). We next demonstrate that also more complex problems, located at the second level of the Polynomial Hierarchy, can be encoded in ASP. To this end, we now consider a knowledge representation problem, inspired by a common business situation, which is known under the name *Strategic Companies* [41].

Suppose there is a collection $C = \{c_1, \ldots, c_m\}$ of companies c_i owned by a holding, a set $G = \{g_1, \ldots, g_n\}$ of goods, and for each c_i we have a set $G_i \subseteq G$ of goods produced by c_i and a set $O_i \subseteq C$ of companies controlling (owning) c_i. O_i is referred to as the *controlling set* of c_i. This control can be thought of as a majority in shares; companies not in C, which we do not model here, might have shares in companies as well. Note that, in general, a company might have more than one controlling set. Let the holding produce all goods in G, i.e. $G = \bigcup_{c_i \in C} G_i$.

A subset of the companies $C' \subseteq C$ is a *production-preserving* set if the following conditions hold: (1) The companies in C' produce all goods in G, i.e., $\bigcup_{c_i \in C'} G_i = G$. (2) The companies in C' are closed under the controlling relation, i.e. if $O_i \subseteq C'$ for some $i = 1, \ldots, m$ then $c_i \in C'$ must hold.

A subset-minimal set C', which is *production-preserving*, is called a *strategic set*. A company $c_i \in C$ is called *strategic*, if it belongs to some strategic set of C.

This notion is relevant when companies should be sold. Indeed, intuitively, selling any non-strategic company does not reduce the economic power of the holding. Computing strategic companies is on the second level of the Polynomial Hierarchy [41].

In the following, we consider a simplified setting as considered in [41], where each product is produced by at most two companies (for each $g \in G$ $|\{c_i \mid g \in G_i\}| \leq 2$) and each company is jointly controlled by at most three other companies, i.e. $|O_i| \leq 3$ for $i = 1, \ldots, m$. Assume that for a given instance of Strategic Companies, $\mathcal{F}_{st}$ contains the following facts:

- $company(c)$ for each $c \in C$,
- $prod\_by(g, c_j, c_k)$, if $\{c_i \mid g \in G_i\} = \{c_j, c_k\}$, where c_j and c_k may possibly coincide,
- $contr\_by(c_i, c_k, c_m, c_n)$, if $c_i \in C$ and $O_i = \{c_k, c_m, c_n\}$, where c_k, c_m, and c_n are not necessarily distinct.

We next present a program $\mathcal{P}_{st}$, which characterizes this hard problem using only two rules:

$$r_1\colon\ strat(Y) \ \vee \ strat(Z) {:-} prod\_by(X, Y, Z).$$
$$r_2\colon\ strat(W) {:-} contr\_by(W, X, Y, Z),\ strat(X),\ strat(Y),\ strat(Z).$$

Here $strat(X)$ means that company X is a strategic company. The guessing part of the program consists of the disjunctive rule r_1, and the checking part consists of the normal rule r_2. The program $\mathcal{P}_{st}$ is surprisingly succinct, given that Strategic Companies is a hard problem.

The program $\mathcal{P}_{st}$ exploits the minimization which is inherent in the semantics of answer sets for the check whether a candidate set C' of companies that produces all goods and obeys company control is also minimal with respect to this property.

The guessing rule r_1 intuitively selects one of the companies c_1 and c_2 that produce some item g, which is described by $prod\_by(g, c_1, c_2)$. If there was no company control information, minimality of answer sets would naturally ensure that the answer sets of $\mathcal{F}_{st} \cup \{r_1\}$ correspond to the strategic sets; no further checking would be needed. However, when control information is available, the rule r_2 checks that no company is sold that would be controlled by other companies in the strategic set, by simply requesting that this company must be strategic as well. The minimality of the strategic sets is automatically ensured by the minimality of answer sets.

The answer sets of $\mathcal{F}_{st} \cup \mathcal{P}_{st}$ correspond one-to-one to the strategic sets of the holding described in $\mathcal{F}_{st}$; a company c is thus strategic iff $strat(c)$ is in some answer set of $\mathcal{F}_{st} \cup \mathcal{P}_{st}$.

An important note here is that the checking "constraint" r_2 interferes with the guessing rule r_1: applying r_2 may "spoil" the minimal answer set generated by r_1. For example, suppose the guessing part gives rise to a ground rule

$$r_3\text{: } strat(c1) \ \vee \ strat(c2) \text{:-} prod\_by(g, c1, c2)$$

and the fact $prod\_by(g, c1, c2)$ is given in $\mathcal{F}_{st}$. Now suppose the rule is satisfied in the guessing part by making $strat(c1)$ true. If, however, in the checking part an instance of rule r_2 is applied which derives $strat(c2)$, then the application of the rule r_3 to derive $strat(c1)$ is invalidated, as the minimality of answer sets implies that rule r_3 cannot justify the truth of $strat(c1)$, if another atom in its head is true.

Chapter 3

Evaluation of Disjunctive ASP Programs

In the previous chapter we introduced the ASP framework, and described its usae as a tool for knowledge representation. In this chapter we illustrate the main algorithms for the evaluation of ASP programs exploited by the state-of-the-art ASP system DLV; these algorithms are sequential, and feature several optimization for the evaluation of the programs on a single processor machine.

The evaluation of ASP programs is a two-step process: in the first step the input program $\mathcal{P}$ undergoes the so-called *instantiation* (or *grounding*) process, which produces a program $\mathcal{P}'$ semantically equivalent to $\mathcal{P}$, but not containing any variable. The instantiation could be performed by simply applying every possible substitution of variables; however, several optimizations can be applied to the process in order to avoid the creation of ground atoms that have no chance to appear in $\mathcal{P}'$. The DLV instantiator applies a number of "intelligent grounding" techniques in order to keep the size of the ground program as small as possible. In the second step, the grounded program p' undergoes the model generation phase, in which answer sets are generated. This process implies the application of a procedure similar to the DPLL [42], variations of which are often employed in solving instances of the Satisfability problem. The technique adopted for ASP programs relies on a classical backtracking algorithm enhanced with dedicated branching heuristics, which produces a candidate solution. The candidate models are then checked for stability in order to filter the actual answer sets.

This chapter is structured as follows: in Section 3.1 we illustrate the instantiation process, and describe the algorithm implemented in DLV; then, in Section 3.2, we describe the model generation basic operations, discuss the heuristics adopted, and the process of stability check.

3.1 Instantiation

In this section we describe the instantiation phase, and in particular its implementation in the ASP systems DLV.

Given an input program $\mathcal{P}$, the full instantiation $Ground(\mathcal{P})$ contains all the ground rules that can be generated applying every possible substitution of

$$\begin{tikzcd}
p \arrow[d] & t \arrow[l] \arrow[d] \\
q \arrow[u, bend left] & s \arrow[l]
\end{tikzcd}$$

$G(\mathcal{P})$

Figure 3.1: Dependency Graph.

variables. The DLV instantiator generates a ground instantiation that has the same answer sets as the full one, but is much smaller in general [7]. Note that the size of the instantiation is a crucial aspect for efficiency, since the answer set computation takes an exponential time (in the worst case) in the size of the ground program received as input (i.e., produced by the instantiator). In order to generate a small ground program equivalent to $\mathcal{P}$, the DLV instantiator generates ground instances of rules containing only atoms which can possibly be derived from $\mathcal{P}$, and thus avoiding the combinatorial explosion that may occur in the case of a full instantiation [27]. This is obtained by taking into account some structural information of the input program, concerning the dependencies among IDB predicates. Such dependencies can be identified by mean of the *Dependency Graph* of $\mathcal{P}$ (see subsection 2.3).

The graph $G(\mathcal{P})$ induces a subdivision of $\mathcal{P}$ into subprograms (also called *modules*) allowing for a modular evaluation. We say that a rule $r \in \mathcal{P}$ *defines* a predicate p if p appears in the head of r. For each strongly connected component (SCC)[1] C of $G(\mathcal{P})$, the set of rules defining all the predicates in C is called *module* of C and is denoted by $\mathcal{P}_c$.[2]

More in detail, a rule r occurring in a module $\mathcal{P}_c$ (i.e., defining some predicate $q \in C$) is said to be *recursive* if there is a predicate $p \in C$ occurring in the positive body of r; otherwise, r is said to be an *exit rule.*

Example 3.1.1. Consider the following program $\mathcal{P}$, where a is an EDB predicate:

$$\begin{array}{rcl@{\qquad}rcl}
p(X,Y) \vee s(Y) & :- & t(X), t(Y), not\ q(X,Y). & t(X) & :- & a(X). \\
p(X,Y) & :- & t(X), q(X,Y). & q(X,Y) & :- & p(X,Y), s(Y).
\end{array}$$

Graph $G(\mathcal{P})$ is illustrated in Figure 3.1; the strongly connected components of $G(\mathcal{P})$ are $\{s\}$, $\{t\}$ and $\{p,q\}$. They correspond to the three following modules:

- $\{\ p(X,Y) \vee s(Y) {:-} t(X), t(Y), not\ q(X,Y).\ \}$
- $\{\ t(X) {:-} a(X).\ \}$
- $\{\ p(X,Y) {:-} t(X), q(X,Y).\qquad p(X,Y) \vee s(Y) {:-} t(X), t(Y), not\ q(X,Y).$ $q(X,Y) {:-} p(X,Y), s(Y).\ \}$

[1]We briefly recall here that a strongly connected component of a directed graph is a maximal subset of the vertices, such that every vertex is reachable from every other vertex.

[2]Note that, since integrity constraints are considered as rules with exactly the same head (which is a special symbol appearing nowhere in the program), they all belong to the same module.

```
**Algorithm Instantiate** ($\mathcal{P}$; $G(\mathcal{P})$)
**Input**: a program $\mathcal{P}$, the dependency graph $G(\mathcal{P})$ associated to $\mathcal{P}$
**Output**: a ground program $\Pi$
**Var**:
  $S$: **set** of atoms;
  $C$: **set** of predicates;
  $\Pi$: **set** of ground rules;
**begin**
  1.    $S := EDB(\mathcal{P})$;   $\Pi := \emptyset$;
  2.    **while** $G(\mathcal{P}) \neq \emptyset$ **do**
  3.            Remove a SCC $C$ from $G(\mathcal{P})$ without incoming edges;
  4.            $\Pi := \Pi \cup$ **InstantiateComponent**($\mathcal{P}, C, S$);
  5.    **end while**
  6.    **return** $\Pi$;
**end**;
```

Figure 3.2: The DLV Instantiation Algorithm.

Moreover, the first and second module do not contain recursive rules, whereas the third one contains one exit rule, namely $p(X,Y) \vee s(Y)$:– $t(X)$, $t(Y)$, *not* $q(X,Y)$., and two recursive rules. □

The dependency graph[3] induces a partial ordering among its SCCs, defined as follows: for any pair of SCCs A, B of $G(\mathcal{P})$, we say that B *directly depends on* A (denoted $A \prec B$) if there is an arc from a predicate of A to a predicate of B; and, B *depends* on A if $A \prec_s B$, where $\prec_s$ denotes the transitive closure of relation $\prec$.

Example 3.1.2. Consider the dependency graph $G(\mathcal{P})$ shown in Figure 3.1; it is easy to see that component $\{p,q\}$ depends on components $\{s\}$ and $\{t\}$, while $\{s\}$ depends only on $\{t\}$. □

This ordering can be exploited to pick out an ordered sequence $C_1, \ldots, C_n$ of SCCs of $G(\mathcal{P})$ (which is not unique, in general) such that whenever C_j depends on C_i, C_i precedes represent a partial ordering which can be used to perform a layered evaluation of the program one module at time so that all data needed for the instantiation of a module C_i have been already generated by the instantiation of the modules preceding C_i.

In the following we illustrate in detail the instantiation algorithm based on the principles discussed above.

The algorithm *Instantiate* shown in Figure 3.2 takes as input both a program $\mathcal{P}$ to be instantiated and the dependency graph $G(\mathcal{P})$, and outputs a set Π of ground rules containing only atoms which can possibly be derived from $\mathcal{P}$, such that $ANS(\mathcal{P}) = ANS(\Pi \cup EDB(\mathcal{P}))$. As already pointed out, the input program $\mathcal{P}$ is divided into modules corresponding to the SCCs of the dependency graph $G(\mathcal{P})$. Such modules are evaluated one at a time according to an ordering induced by the dependency graph. We recall here that this ordering is, in

[3]It is worth remembering that, according to its definition (see subsection 2.3 in the previous chapter), the dependency graph does not take into account negative dependencies.

```
Function InstantiateComponent ($\mathcal{P}$; $C$; $S$)
Input: a program $\mathcal{P}$, a set of predicates $C$, a set of atoms $S$
Output: a set of ground rules $\Pi_C$
Var:
  $\mathcal{NS}$: set of atoms;
  $\Delta S$: set of atoms;
  $\Pi_C$: set of ground rules;
begin
  1.  $\mathcal{NS} := \emptyset$;  $\Delta S := \emptyset$;  $\Pi_C := \emptyset$;
  2.  for each $r \in Exit(C, \mathcal{P})$; do
  3.          $\Pi_C := \Pi_C \cup$ InstantiateRule($r, S, \Delta S, \mathcal{NS}$);
  4.  end for
  5.  do
  6.          $\Delta S := \mathcal{NS}$; $\mathcal{NS} = \emptyset$;
  7.          for each $r \in Recursive(C, \mathcal{P})$; do
  8.                  $\Pi_C := \Pi_C \cup$ InstantiateRule($r, S, \Delta S, \mathcal{NS}$);
  9.          $S := S \cup \Delta S$;
  10. while $\mathcal{NS} \neq \emptyset$
  11. return $\Pi_C$;
end;
```

Figure 3.3: The DLV Instantiation Component Procedure.

general, not unique; however, the order in which program modules that do not depend on each other are instantiated is irrelevant.

The algorithm creates a new set of atoms S that will contain the subset of the base $B_{\mathcal{P}}$ significant for the instantiation. Initially, $S = EDB(\mathcal{P})$, and $\Pi = \emptyset$. Then, a strongly connected component C, with no incoming edge, is removed from $G(\mathcal{P})$, and the program module corresponding to C is evaluated by invoking *InstantiateComponent*, which returns the set of ground rules obtained instantiating such module; those rules are then added to the set of ground rules Π. This ensures that modules are evaluated one at a time so that whenever $C_1 \prec_s C_2$, $\mathcal{P}_{C_1}$ is evaluated before $\mathcal{P}_{C_2}$. The *Instantiate* algorithm runs on until all the components of $G(\mathcal{P})$ have been evaluated.

Example 3.1.3. Let $\mathcal{P}$ be the program of Example 3.1.1. The unique component of $G(\mathcal{P})$ having no incoming edges is $\{t\}$. Thus the program module $\mathcal{P}_t$ is evaluated first. Then, once $\{t\}$ has been removed from $G(\mathcal{P})$, $\{s\}$ becomes the (unique) component of $G(\mathcal{P})$ having no incoming edge and is therefore taken. Once $\{s\}$ has been evaluated and thus removed from $G(\mathcal{P})$,$\{p, q\}$ is processed at last, completing the instantiation process. □ □

The procedure *InstantiateComponent*, in turn, takes as input the component C to be instantiate, the set of atoms S, and performs the instantiation of the module corresponding C. Firstly, the set of atoms $\mathcal{NS}$ and ΔS, which are used in the instantiation of the recursive rules, and the set of ground rules Π_C are initialized to $\emptyset$. Then, following the instantiation scheme previously described, the procedure proceeds instantiating the *exit* rules first, and then the *recursive* ones; at the same time, it updates the set S with the atoms occurring in the heads of the rules of Π. To this end, each rule r in the program module of C is processed by calling procedure *InstantiateRule*. This, given the set of atoms which are known to be significant up to now, builds all the ground instances of r, marks as significant the head atoms of the newly generated ground rules, and return the set of ground rules generated. The instantiation of a rule r is performed in such a way that only atoms that can possibly be derived from $\mathcal{P}$ are taken into account. In case of exit rules, those atoms are exactly those contained in the set S. It is worth noting that a disjunctive rule r may appear in the program modules of two different components. In order to deal with this, before processing r, *InstantiateRule* checks whether it has been already grounded during the instantiation of another component. This ensures that a rule is actually processed only within one program module.

Concerning recursive rules, they are processed several times according to a semi-naïve evaluation technique [31], where at each iteration n only the significant information derived during iteration $n-1$ has to be used. This is implemented by partitioning significant atoms into three sets: ΔS, S, and $\mathcal{N}S$. $\mathcal{N}S$ is filled with atoms computed during current iteration (say n); ΔS contains atoms computed during previous iteration (say $n-1$); and, S contains the ones previously computed (up to iteration $n-2$). Initially, ΔS and $\mathcal{N}S$ are empty, and the exit rules contained in the program module of C are evaluated by a single[4] call to procedure *InstantiateRule*; then, the recursive rules are evaluated (do-while loop). At the beginning of each iteration, $\mathcal{N}S$ is assigned to ΔS, i.e. the new information derived during iteration n is considered as significant information for iteration $n+1$. Then, *InstantiateRule* is invoked for each recursive rule r, and, at the end of each iteration, ΔS is added to S (since it has already been exploited). The procedure stops whenever no new information has been derived (i.e. $\mathcal{N}S = \emptyset$).

Proposition 3.1.4. [43] Let $\mathcal{P}$ be an ASP program, and Π be the ground program generated by the algorithm *Instantiate*. Then $ANS(\mathcal{P}) = ANS(\Pi \cup EDB(\mathcal{P}))$ (i.e. $\mathcal{P}$ and $\Pi \cup EDB(\mathcal{P})$ have the same answer sets).$\Box$

Example 3.1.5. Let $\mathcal{P}$ be the program of Example 3.1.1. Assume that, initially, the set of facts is represented by $EDB = \{a(1), a(2)\}$; the algorithm depicted in Figure 3.2 proceeds as follows: the set S is filled in with the content of EDB, and Π is initialized to $\emptyset$; then the instantiation of the program modules is performed.

The first component to be processed is $\{t\}$, because it is the only one which has no incoming edges. Then, the procedure *InstantiateComponent* is invoked: the correspondent program module contains only the exit rule $\{t(X){:-}a(X).\}$, which is instantiated invoking the procedure *InstantiateRule*. The instantiation of this rule produce the ground rules

$$t(1){:-}a(1). \quad t(2){:-}a(2).$$

The set S is updated with the atoms $t(1), t(2)$.

Then, the algorithm proceeds instantiating the component $\{a\}$, which is the only one having no incoming edges; again the corresponding program module contains only one exit rule, namely $\{p(X,Y) \vee s(Y){:-}t(X), t(Y), not\ q(X,Y).\}$.

[4]Since no recursive atom occurs in the body of exit rules, a single call to *InstantiateRule* is sufficient for completely evaluating them.

In this case the procedure *InstantiateComponent* returns the following ground rules

$$p(1,1) \vee s(1) \text{:-} t(1), t(1), not\ q(1,1).$$
$$p(1,2) \vee s(2) \text{:-} t(1), t(2), not\ q(1,2).$$
$$p(2,1) \vee s(1) \text{:-} t(2), t(1), not\ q(2,1).$$
$$p(2,2) \vee s(2) \text{:-} t(2), t(2), not\ q(2,2).$$

The atoms $p(1,1), p(1,2), p(2,1), p(2,2), s(1), s(2)$ are added to the set S.

Lastly, the component $\{p, q\}$ can be instantiated; this contains one exit rule and the two recursive rules $\{p(X,Y)\text{:-}t(X), q(X,Y)., q(X,Y)\text{:-}p(X,Y), s(Y).\}$. As the exit rule has already been instantiated, the invocation of the function *InstantiateRule* for this rule has no effect. The instantiation of the recursive rules, on the other hand, takes two iteration of the seminaïve algorithm. Initially, the set $\mathcal{NS}$ contains all the new predicates introduced by the instantiation of the exit rule; the content of ΔS is then replaced with the content of $\mathcal{NS}$, and the first iteration of the semiñaive is performed which produces the following ground rules

$$q(1,1) \text{:-} p(1,1), s(1).$$
$$q(2,1) \text{:-} p(2,1), s(1).$$
$$q(1,2) \text{:-} p(1,2), s(2).$$
$$q(2,2) \text{:-} p(2,2), s(2).$$

The set $\mathcal{NS}$ is $\{q(1,1), q(1,2), q(2,1), q(2,2)\}$. Before the second iteration starts, the set S is filled with the atoms contained in ΔS; then ΔS is set to the content of $\mathcal{NS}$, and the second iteration is performed producing the ground rules

$$p(1,1) \text{:-} t(1), q(1,1).$$
$$p(1,2) \text{:-} t(1), q(1,2).$$
$$p(2,1) \text{:-} t(2), q(2,1).$$
$$p(2,2) \text{:-} t(2), q(2,2).$$

In this case no new information is derived, that is, all the heads of the ground rules produces are already contained in S; as a consequence, the seminaïve algorithm ends, and the function *InstantiateComponent* returns the set of ground rules that has been produced.

□

3.2 Computation of Answer Sets for Propositional Programs

In this section, we describe the general algorithm for computing the answer sets of a propositional program. In particular we refer to the kernel module (called *Model Generator*) of the ASP system DLV system[34, 44, 7]; other ASP systems like Smodels[14] and Clasp [15] employs similar procedures for the evaluation of

```
Function: ModelGenerator($I$)
Input: An Interpretation $I$
Output: A boolean (true if $I$ is and Answer Set)
Var:
 $L$: Literal
 contradiction: boolean
begin
 1.  $I$ = DetCons(I);
 2.  if I = $\mathcal{L}$ then (* inconsistency *)
 3.      return $FALSE$;
 4.  endif
 5.  if no atom is undefined in I then
 6.      return IsStable(I); (* stability check *)
 7.  endif
 8.  $L$ = Select(I);
 9.  if ModelGenerator($I \cup \{L\}$) then
 10.     return $TRUE$;
 11. else
 12.     return ModelGenerator($I \cup \{\texttt{not}.L\}$);
 13. endif
end;

Function: DetCons($I$)
Input: An Interpretation $I$
Output: an extended interpretation
(* Extend $I$ with literals that can be deterministically inferred and or
   $I$ becomes the set of all literals $\mathcal{L}$ upon inconsistency.
   Possible implementations are described in [47, 44, 48]. *)
```

Figure 3.4: Computation of answer sets in DLV

the programs [5].

The algorithm employed for the generation of the candidate answer sets is sketched in Figure 3.4. In the sake of clarity the description of the model generator module is simplified; the actual implementation implies several optimizations and complex data structures whose description is out of the scope of this thesis. Moreover, the actual implementation computes and outputs all or a fixed number of answer sets rather than just deciding whether answer sets exist; however, the extension is quite straightforward and would only add details which are not relevant. Interested readers may find a more detailed description in [44].

As already stated, the *Model Generator* algorithm is aimed at producing some candidate answer sets; each candidate I is then tested by mean of the function IsStable(I), which checks whether I is a minimal model of the program $\mathcal{P}^I$ obtained by applying the GL-transformation w.r.t. I. This operation is performed by the sub-sequent module called *Model Checker*, which is discussed later on in this section.

[5]Other solvers like Cmodels[45] and ASSAT[46] use a different architecture based on transformations to SAT.

Initially, the *Model Generator* function is called with parameter I set to the empty interpretation; this is because, at the beginning, all the literals are undefined[6]. Hereafter, for simplicity we suppose that the input program $\mathcal{P}$ is a global variable of the system. If $\mathcal{P}$ has at least one answer set, then the function returns true and the (total) interpretation I is an answer set; otherwise it returns false. Notably, the Model Generator is similar to the Davis-Putnam procedure, whose variants are very commonly employed in the design of SAT solvers. The function proceeds as follows: first of all, the extension of the partial interpretation I with those literals that can be deterministically inferred is performed by calling the function DetCons(I); whenever an inconsistency is detected, the function returns the set of all all literals $\mathcal{L}$. This function is similar to a unit propagation procedure employed by SAT solvers, but exploits the peculiarities of ASP for making further inferences, e.g., it exploits the knowledge that every answer set is a minimal model [47, 44, 48] .

If *DetCons* detects an inconsistency, the function *ModelGenerator* returns false and a backtracking step is performed; otherwise, a literal L is selected according to a heuristic criterion (by a call to the *Select* procedure, see next section) and *ModelGenerator* is called on both $I \cup \{L\}$ and $I \cup \{\texttt{not}.L\}$. The literal L corresponds to a *branching variable* in SAT solvers. And indeed, like for SAT solvers, the selection of a "good" literal L is crucial for the performance of a ASP system.

In the following, we describe the framework for evaluating heuristic criteria as adopted in the DLV system and then describe a number of heuristic criteria for the selection of the branching literal; then, we discuss how the model checking is performed by the function *isStable*.

3.2.1 Evaluation of Heuristic Functions

The Model Generator of DLV can be configured with a number of "dynamic" heuristic functions; they are "dynamic" in a sense that the heuristic values associated to possible choices are determined during the evaluation of the program. In particular, the heuristic value of a literal L depends on the result of taking L true as well as false and computing its consequences, respectively. Unfortunately, evaluating the heuristic value of each possible choice it is an expensive operation; so as, in order to reduce the number of look-aheads, the DLV system does not evaluate the heuristic value of *all* undefined literals; rather, it considers only a subset of the undefined literals called *possibly-true* literals. The correctness of this strategy has been shown in [34].

Definition 4. (*Definition 1 in* [29]) (**PT literal**) Let I be a partial interpretation for the (ground) program $\mathcal{P}$.

A *positive Possibly-True (PT) literal* of $\mathcal{P}$ w.r.t. I is an undefined positive literal l such that there exists a rule $r \in \mathcal{P}$ for which all of the following conditions hold:

1. l is in the head of r: $l \in H(r)$;

2. the head of r is not true w.r.t. I: $H(r) \cap I = \emptyset$;

[6]Note again that the interpretations handled by the Model Generator are partial interpretations where any literal is either one of true, false, or undefined w.r.t. an interpretation I.

3. the body of r is true w.r.t. I: $B(r) \subseteq I$.

A *negative PT literal* of $\mathcal{P}$ w.r.t. I is an undefined negative literal $\texttt{not}\ l$ such that there exists a rule $r \in \mathcal{P}$ for which all of the following conditions hold:

1. $\texttt{not}\ l$ is in the body of r: $\texttt{not}\ l \in B(r)$;
2. the head of r is not true w.r.t. I: $H(r) \cap I = \emptyset$;
3. the positive body of r is true w.r.t. I: $B^+(r) \subseteq I$;
4. no negative body literal is false w.r.t. I: $I \cap \texttt{not}.B^-(r) = \emptyset$.

The set of all PT literals of $\mathcal{P}$ w.r.t. I is denoted by $PT_{\mathcal{P}}(I)$. □

Example 3.2.1. Consider the following program $\mathcal{P}$

$$a \lor b{:-}\ c, d.$$

$$e{:-}\ d, \texttt{not} f.$$

and let $I = \{c, d\}$ be an interpretation for $\mathcal{P}$, then $PT_{\mathcal{P}}(I) = \{a, b, \texttt{not}\ f\}$.

□

Note that, the set of PT literals is always not strictly smaller than the set of all the atoms that may be selected. Indeed, it can happen that all the undefined atoms of a program are PT literals at some point. However, this technique is very effective when it applies; for example, in the program HAMPATH presented in Section 2.4 of Chapter 2, at any given stage of the computation, the PTs are those literals of the form $inPath(a, b)$ or $outPath(a, b)$, where a is a node already reached from the start ($reached(a)$ is true) and (a, b) is an arc of the input graph.

The function *select* shown in Figure 3.5 is used to choose the heuristically best literal. The function iterates over PT literals (line 2); for each literal A look-aheads for $I \cup \{A\}$ and for $I \cup \{\texttt{not}.A\}$ are performed by calling function *DetCons*, and results are stored in $\mathrm{I}_{\mathrm{A}}^{+}$ and $\mathrm{I}_{\mathrm{A}}^{-}$,respectively (lines 3-11). If either the assumption of A or the assumption of $\texttt{not}.A$ leads to an inconsistency, then the complementary literal is deterministically added to the interpretation, calling again *DetCons* (lines 5,9). Otherwise, A is compared with the previously best literal L by exploiting an heuristic criterion h_C (lines 13-16).

Once all PTs have been considered, the best literal according to the heuristic is returned in the parameter L and then assumed in the *ModelGenerator* Function.

We present now a number of heuristic criterion that have been implemented in DLV in order to determine the comparison operator h_C (for a more detailed description see [29]).

3.2.2 Heuristics

In order to define an heuristics for this framework, it is sufficient to define an appropriate comparison operator $<_{h_C}$, which may depend on $\mathrm{I}_{\mathrm{A}}^{+}$ and/or $\mathrm{I}_{\mathrm{A}}^{-}$, or rather on some values that were collected during the computation of $\mathrm{I}_{\mathrm{A}}^{+}$ and/or $\mathrm{I}_{\mathrm{A}}^{-}$.

```
Function: Select($I$);
Input: An interpretation $I$
Output: A literal $L$
Var:
  $L$: A Literal
  $\mathrm{I}^+_\mathrm{A}, \mathrm{I}^-_\mathrm{A}$: Interpretation;
begin
  1.  $L := NULL$;
  2.  foreach $A \in PT_\mathcal{P}(I)$ do
  3.      $\mathrm{I}^+_\mathrm{A}$ := DetCons($I \cup \{A\}$);    (* look-ahead for $A$ *)
  4.      if $\mathrm{I}^+_\mathrm{A} = \mathcal{L}$ then
  5.          $I := DetCons(I \cup \{\texttt{not}.A\})$;
  6.      else
  7.          $\mathrm{I}^-_\mathrm{A}$ := DetCons($I \cup \{\texttt{not}.A\}$); (* look-ahead for not.$A$ *)
  8.          if $\mathrm{I}^-_\mathrm{A} = \mathcal{L}$ then
  9.              $I := I \cup \{A\}$;
  10.         endif
  11.     endif
  12.     if $\mathrm{I}^+_\mathrm{A} \neq \mathcal{L}$ and $\mathrm{I}^-_\mathrm{A} \neq \mathcal{L}$ then   (* no inconsistency has arisen *)
  13.         if $L = NULL$ then
  14.             $L := A$; (* first literal, no comparison *)
  15.         elseif $L <_{h_C} A$ then  (* compare $A$ against L w.r.t. the heuristic *)
  16.             $L := A$;
  17.         endif
  18.     endif
  19. endfor
  20. return $L$;
end;
```

Figure 3.5: Framework for the selection of the branching literal in DLV

Before describing the heuristics we introduce some general definitions. Given a literal L, from now we denote with $ext(I, L)$ the interpretation resulting from the application of a deterministic consequence operator on $I \cup \{L\}$. In DLV, this amounts to a call to *DetCons*, i.e. $ext(I, L) = DetCons(I \cup L)$. We usually assume that $ext(I, L)$ is consistent, otherwise the framework for heuristic evaluation of DLV (see Figure 3.5) deterministically assumes `not.`L and the heuristic is not evaluated on L at all. In the following paragraphs we describe some of the heuristics that may be computed by DLV.

Heuristic $\mathbf{h_1}$ This is an extension of the branching rule adopted in the system SATZ [49] – an efficient SAT solver – to fit ASP settings.

The *length* of a rule r (w.r.t. an interpretation I), is the number of undefined literals occurring in r. Let $Unsat_k(L)$ denote the number of unsatisfied rules[7] of length k w.r.t. $ext(I, L)$, which have a greater length w.r.t. I. In other words, $Unsat_k(L)$ is the number of unsatisfied rules whose length shrinks to k if L is

[7]Recall that a rule r is satisfied w.r.t. an interpretation J if the body of r is false w.r.t. J or the head of r is true w.r.t. J.

assumed and propagated in the interpretation I. The weight $w_1(L)$ is:

$$w_1(L) = \Sigma_{k>1}\ Unsat_k(L) * 5^{-k}$$

Thus, the weight function w_1 prefers literals introducing a higher number of short unsatisfied rules. Intuitively, the introduction of a high number of short unsatisfied rules is preferred because it creates more and stronger constraints on the interpretation so that a contradiction can be found earlier [49]. The weight of L is combined with the weight of its complement $\texttt{not}.L$ to favour L such that $w_1(L)$ and $w_1(\texttt{not}.L)$ are roughly equal, in order to avoid that a possible failure leads to a very bad state. To this end, as in SATZ, the combined weight *comb*-$w_1(L)$ of L is defined as follows:

$$comb\text{-}w_1(L) = w_1(L) * w_1(\texttt{not}.L) * 1024 + w_1(L) + w_1(\texttt{not}.L).$$

The idea is that for values $w_1(L)$ and $w_1(\texttt{not}.L)$ which are closer together, the product is greater than for values which are further apart but have the same sum. Both constant values 5 and 1024 are the "empirically optimal" value that have been determined in [49]

Given two literals A and B, heuristic h_1 prefers B over A ($A <_{h1} B$) if:

1. *comb*-$w_1(A) <$ *comb*-$w_1(B)$ when $A \neq \texttt{not}.B$;

2. $w_1(A) < w_1(B)$ when $A = \texttt{not}.B$.[8]

Heuristic $\mathbf{h_2}$ The second heuristic is inspired by the branching rule of Smodels [14]. Let $|J|$ denote the number of literals in a (three-valued) interpretation J. Then, define

$$w_2(L) = |ext(I, L)|.$$

Since w_2 maximizes the size of the resulting interpretation, it minimizes the literals which are left undefined. Intuitively, this minimizes the size of the remaining search space [14] (which is 2^u, where u is the number of undefined atoms w.r.t. $ext(I, L)$). Similar to Smodels, the heuristic h_2 cautiously maximizes the minimum of $w_2(L)$ and $w_2(\texttt{not}.L)$. More precisely, the preference relationship $<_{h2}$ of h_2 is defined as follows. Given two literals A and B:

1. $A <_{h2} B$ if $min(w_2(A), w_2(\texttt{not}.A)) < min(w_2(B), w_2(\texttt{not}.B))$;

2. otherwise, $A <_{h2} B$ if $min(w_2(A), w_2(\texttt{not}.A)) = min(w_2(B), w_2(\texttt{not}.B))$, and $max(w_2(A), w_2(\texttt{not}.A)) < max(w_2(B), w_2(\texttt{not}.B))$

Heuristic $\mathbf{h_4}$ This is the basic heuristics used in the DLV system.

A peculiar property of answer sets is *supportedness*, cf. Section 2.3. Since a ASP system must eventually converge to a supported interpretation, it makes sense to keep the interpretations "as much supported as possible" during the intermediate steps of the computation. To this end, the number of *UnsupportedTrue (UT)* atoms is counted, i.e., atoms which are true in the current interpretation but still miss a supporting rule (further details on UTs can be found in [47, 44] where they are called must-be-true atoms or MBTs). For

[8]Note that *comb*-$w_1(A) =$ *comb*-$w_1(B)$ if $A = \texttt{not}.B$.

instance, the rule $:\!-$*not* x implies that x must be true in every answer set of the program; but it does not give a "support" for x. Thus, *DetCons* derives x in order to satisfy the rule and adds it to the set *UnsupportedTrue*. It will be removed from this set once a supporting rule for x is found (e.g., $x \vee b{:-}c$ would be a supporting rule for x in the interpretation $I = \{x, \texttt{not}\ b, c\}$). Given a literal L, let $UT(L)$ be the number of UT atoms in $ext(I, L)$. Moreover, let $UT_2(L)$ and $UT_3(L)$ be the number of UT atoms occurring, respectively, in the heads of exactly 2 and 3 unsatisfied rules w.r.t. $ext(I, L)$. Intuitively, these are the most constrained UTs. Note that a UT atom in the of head only one unsatisfied rule will cause *DetCons* to make appropriate derivations in order to turn that rule into a supporting rule, therefore such atoms will not occur anymore during the evaluation of the heuristic.

The heuristic h_4 of DLV considers $UT(L)$, $UT_2(L)$ and $UT_3(L)$ in a prioritized way, to favor atoms yielding interpretations with fewer $UT/UT_2/UT_3$ atoms (which should more likely lead to a supported model). If all UT counters are equal, then the heuristic considers the total number $Sat(L)$ of rules which are satisfied w.r.t. $ext(I, L)$. If also SAT counters are equals, the degree of supportedness is considered, which is introduced next.

Given a literal L, let $True(L)$ be the number of true non-HCF atoms in $ext(I, L)$, and let $SuppRules(L)$ be the number of all supporting rules for non-HCF atoms w.r.t. $ext(I, L)$; let DS be the degree of supportedness of the interpretation intended as the ratio between the number of supporting rules and the number of true atoms (i.e. $DS(L) = SuppRules(L)/True(L)$[9])

The degree of supportedness is considered to cover the case of non-HCF programs, for which it is not guaranteed that supported models are answer sets. Therefore stability check may be performed several time unfruitfully leading to models which are not answer sets. Unfortunately, stability checking is computationally expensive (co-NP-complete), and may consume a large portion of the resources needed for computing a answer set. To this aim, the heuristic h_4 tries to drive the computation toward supported models with a better chance of being answer sets, preferring models with a greater degree of supportedness (the average number of supporting rules for a true atom).

To this end, given a literal L, let $UT'(L) = UT(L) + UT(\texttt{not}.L)$, $UT'_2(L) = UT_2(L) + UT_2(\texttt{not}.L)$, $UT'_3(L) = UT_3(L) + UT_3(\texttt{not}.L)$, $Sat'(L) = Sat(L) + Sat(\texttt{not}.L)$, and $DS'(L) = DS(L) + DS(\texttt{not}.L)$. Note that heuristic h_4 is "balanced" (i.e. it takes into account both the look-ahead for the branching literal and its complement).

The preference relationship $<_{h4}$ of h_4 is defined as follows. Given two literals A and B:

1. $A <_{h4} B$ if $UT'(A) > UT'(B)$;
2. otherwise, $A <_{h4} B$ if $UT'(A) = UT'(B)$ and $UT'_2(A) > UT'_2(B)$;
3. otherwise, $A <_{h4} B$ if $UT'(A) = UT'(B)$, $UT'_2(A) = UT'_2(B)$ and $UT'_3(A) > UT'_3(B)$;
4. otherwise, $A <_{h4} B$ if $UT'(A) = UT'(B)$, $UT'_2(A) = UT'_2(B)$, $UT'_3(A) = UT'_3(B)$ and $Sat'(A) < Sat'(B)$.

[9]In the implementation, the denominator is increased by 1, in order to avoid possible divisions by zero.

5. otherwise, $A <_{h4} B$ if $UT'(A) = UT'(B)$, $UT'_2(A) = UT'_2(B)$, $UT'_3(A) = UT'_3(B)$, $Sat'(A) = Sat'(B)$ and $DS'(A) < DS'(B)$.

3.2.3 Model Checking

The goal of *model checking* is to verify whether a model M is an answer set for an input program. This task is very hard in general, because checking the stability of a model is well-known to be a co-NP-complete [2] problem. However, the task is polynomial [34] for the class of HCF-free programs. To this purpose, the DLV system adopts a strategy that performs a polynomial time check for easy problems (HCF); in case of hard problems (non-HCF), an additional check is carried out by translating the program into a SAT formula and checking whether it is unsatisfiable.

Before describing this approach, let us consider some preliminary notions. First of all we define the notions of *unfounded set* and *unfounded-free* programs.

Definition 5. (Definition 3.1 in [34]) Le I be a total interpretation for a program $\mathcal{P}$. A set $X \subseteq B_{\mathcal{P}}$ of ground atoms is an *unfounded set* for $\mathcal{P}$ w.r.t. I if, for each rule $r \in Ground(\mathcal{P})$ such that $X \cap H(r) \neq \emptyset$, at least one of the following condition holds:

C_1. $(B^+(r) \nsubseteq I) \;\vee\; (B^-(r) \cap I \neq \emptyset)$, that is, the body of r is false w.r.t. I.

C_2. $(B^+(r) \cap I \neq \emptyset)$, that is, some positive body literal belongs to X.

C_3. $(H(r) - X) \cup I \neq \emptyset$, that is, an atom in the head of r, distinct from the element in X, is true w.r.t. I.

Example 3.2.2. Let $\mathcal{P} = \{a \vee b.\}$ and $I = \{a, b\}$. Owing to Condition 3, both $\{a\}$ and $\{b\}$ are unfounded set w.r.t. $\mathcal{P}$. □

Based on Definition 5, an interpretation I for a program $\mathcal{P}$ is *unfounded-free* iff no non-empty subset of I is an unfounded set for $\mathcal{P}$ w.r.t. I. Most interesting, the unfounded-free condition singles out precisely the answer sets as confirmed by the following proposition.

Proposition 3.2.3. *(Theorem 4.6 in [34]) Let M be a model for a program $\mathcal{P}$. M is an answer set for $\mathcal{P}$ iff M is unfounded-free.*

Example 3.2.4. Let $\mathcal{P} = \{a \vee b.\}$. $M_1 = \{a\}$ is an answer set for $\mathcal{P}$, since there is no non-empty of M_1 which is an unfounded set. As shown in Example 3.2.2, $M_2 = \{a, b\}$ is not unfounded-free, and therefore it is not an answer set.

□

Let us now define the $\mathcal{R}$ operator which is exploited in the stability check.

Definition 6. [50] Let $\mathcal{P}$ be a program I an interpretation. Then we define an operator $\mathcal{R}_{\mathcal{P},\mathcal{I}}$ as follow

$$\mathcal{R}_{\mathcal{P},\mathcal{I}}: 2^{B_{\mathcal{P}}} \to 2^{B_{\mathcal{P}}}$$

$$X \mapsto \{a \in X \mid \forall r \in Ground(\mathcal{P}) \; with \; a \in H(r)\},$$

$$B(r) \cap (\neg.I \cup X) \neq \emptyset \text{ or } (H(r) - \{a\}) \cup I \neq \emptyset\}$$

were $\neg.I$ denotes the set of (ground) literals $\{\texttt{not}l \mid l \in I\}$.

```
Function: isStable($I$);
Input: An interpretation $I$
Output: A boolean true if I is an answer set, false otherwise
Var:
  $X$: set of atoms
begin
  1.  $X := \mathcal{R}^{\omega}_{\mathcal{P},\mathcal{I}}$;
  2.  if($X \neq \emptyset$) then
  3.      if( $\mathcal{P}$ is Head-Cycle-Free) then
  4.          return false;
  5.      else
  6.          if($unfoundedFree(I, X) = false$) then
  7.              return false;
  8.      end if;
  9.  end if;
  10. return true;
end;
```

Figure 3.6: The isStable function for performing stability checking

It can be easily verified that the operator described above is monotonic. Indeed, given a set $X \subseteq B_{\mathcal{P}}$, the sequence $R_0 = X$, $R_n = \mathcal{R}_{\mathcal{P},\mathcal{I}}(R_{n-1})$ decreases monotonically and converges finitely to a limit denoted by $\mathcal{R}^{\omega}_{\mathcal{P},\mathcal{I}}$. It has been shown in [34] that, given a program $\mathcal{P}$ and a model M, all the unfounded sets contained in M are subsets of $\mathcal{R}^{\omega}_{\mathcal{P},\mathcal{I}}$

Proposition 3.2.5. *[50] Let $\mathcal{P}$ be a program (not necessary HCF) and M a model for $\mathcal{P}$. Then $\mathcal{R}^{\omega}_{\mathcal{P},\mathcal{I}} = \emptyset$ implies that M is unfounded-free w.r.t. $\mathcal{P}$.*

Proposition 3.2.6. *(Theorem 6.9 in [34]) Let $\mathcal{P}$ be an HCF program and M a total interpretation for it. Than M is unfounded-free iff $\mathcal{R}^{\omega}_{\mathcal{P},\mathcal{I}} = \emptyset$.*

Those results suggest that computing the fixpoint of the operator $\mathcal{R}$ and checking whether it is empty it is sufficient for verifying the stability of a model in the case of HCF programs. For non-HCF programs an additional Co-NP check is necessary, which is performed by a mapping to UNSAT.

The function *isStable* described in Figure 3.6 summarizes the behaviour of the Model Checker module of DLV. The function receives in input a model M and outputs whether it is an answer set or not. Basically, the fixpoint $\mathcal{R}^{\omega}_{\mathcal{P},\mathcal{I}}$ of the operator $\mathcal{R}$ is computed; if the empty set is returned then the input model is an answer set, and the function returns true. Otherwise, if the program is HCF then the input model is not stable, and the function returns false; and, in case of non-HCF programs, and additional check is performed calling the function *unfoundedFree* (Figure 3.7) that performs a call to an integrated SAT solver: if the formula is satisfiable, than the model is unstable; otherwise it is an answer set. The algorithm *generateSatFormula* depicted in Figure 3.7 describes how the SAT formula is obtained. The algorithm receives in input a candidate model M, and a set of atoms X, which represent an unfounded set; this unfounded set is exploited in order reduce the size of the transformation of $\mathcal{P}$ into a SAT instance. First of all the program $\mathcal{P}$ is simplified removing useless

```
Algorithm: generateSatFormula($M$,$X$);
Input: An interpretation $M$, a set of atoms $X$
Output: A Sat formula
begin
  1.  $\forall r \in \mathcal{P}$ delete $r$ from $P$ if $B(r)$ is false w.r.t $M$;
  2.  $\forall r \in \mathcal{P}$ remove all false atoms (w.r.t. $M$) from the $H(r)$;
  3.  $\forall r \in \mathcal{P}$ delete $r$ from $P$ if $H(r) \not\subseteq X$.
  4.  $\forall r \in \mathcal{P}$ remove all negative literals from $B(r)$;
  5.  $\forall r \in \mathcal{P}$ remove all literals $l$ from the bodies of $r$ such that $l \notin X$;
  6.  $S := \emptyset$
  7.  Let $\mathcal{P}'$ be the program resulting from steps 1–5;
  8.  for each $r$ $a_0 \vee \cdots \vee a_n$:- $b_0, \cdots, b_m \in \mathcal{P}'$ do
          $S := S \cup \{b_0 \vee \cdots \vee b_m$:- $a_0, \cdots, a_n\}$;
  9.  end for
  10. $\Gamma_{M,X}(\mathcal{P}) := \bigwedge_{c \in S} c \wedge (\bigvee_{y \in M} y)$
  11. return $\Gamma_{M,X}(\mathcal{P})$;
end;
```

```
Function: unfoundedFree($I$,$X$);
Input: An interpretation $M$, a set of atoms $X$
Output: A boolean true if I is an answer set, false otherwise
begin
  1.  $\Gamma = generateSatFormula(M, X)$;
  2.  if($SAT(\Gamma)$) then return false;
  3.  else return true;
end;
```

Figure 3.7: Model checking algorithm for no-HCF programs

rules; every rule r whose body is false or such that $H(r) \cup M \not\subseteq X$ is removed from $\mathcal{P}$. The remaining rules are then processed removing any body literal l such that $l \notin X$. Lines 8-10 correspond to the conversion of the obtained rules in the corresponding SAT clauses. Note that, as stated previously, all the possible unfounded sets of M are subsets of $\mathcal{R}^{\omega}_{\mathcal{P},\mathcal{I}}$, and this latter is also exploited for simplifying (Line 3 and Line 5) the SAT formula to be evaluated. The correctness of this algorithm is shown in [50] (Theorem 4.2 and Lemma 5.3).

Chapter 4

Parallel Grounding

In this chapter we present the main contribution of this thesis, that is, the design of a strategy for the parallel instantiation of ASP programs. More in detail, this strategy exploits parallelism in three different points of the instantiation process. The parallel instantiation process we defined employs techniques preliminary presented in [51] and integrates them with a novel strategy which has a larger application field, covering many situations in which the previous techniques do not apply.

This chapter is structured as follows: in Section 4.1 we provide a general description of the techniques mentioned above; in Section 4.2 we give a formal definition of the designed algorithm; in Sections 4.3 and 4.4 we discuss the heuristics employed in the third level of parallelism for partitioning the workload, and the heuristics devised for load balancing and granularity control.

4.1 Pushing Parallelism in the Instantiator

In this section we illustrate a three-level concurrent instantiation technique conceived for pushing parallelism into the instantiation process. In particular, parallelism is exploited in three different points of the computation in such a way that they form a chain. The first level of parallelism, the *components* level allows the instantiation in parallel subprograms of the program in input induced by the SCCs of the dependency graph: it is especially useful when handling programs containing parts that are, somehow, independent. The second one, namely the *rules* level, allows the parallel evaluation of rules within a given subprogram: it is useful when the number of rules in the subprograms is large. The third one, namely the *single rule* level, allows the parallel evaluation of each single rule by partitioning the workload needed to perform its instantiation; it is crucial for the parallelization of programs with few rules, where the first two levels are almost not applicable. In the following paragraphs, we discuss separately each of the three level of parallelism.

Components Level [51]. The first level of parallelism, called *Components Level*, consists in dividing the input program $\mathcal{P}$ into subprograms, according to the dependencies among the IDB predicates of $\mathcal{P}$, and by identifying which of them can be evaluated in parallel.

According to the partial ordering that can is obtained exploiting the dependencies among the SCCs of $G(\mathcal{P})$(see Section 3.1), the instantiation of the input program $\mathcal{P}$ can be carried out by separately evaluating its modules; if the evaluation order of the modules respects the mentioned partial ordering then a small ground program is produced [51]. Indeed, this gives the possibility of computing ground instances of rules containing only atoms that can possibly be derived from $\mathcal{P}$ (thus, avoiding the combinatorial explosion that can be obtained by naively considering all the atoms in the Herbrand base).

Intuitively, this partial ordering guarantees that a component A precedes a component B if the program module corresponding to A has to be evaluated before the one of B, because the evaluation of A produces data that are needed for the instantiation of B. Moreover, the partial ordering allows for determining which modules can be evaluated in parallel. Indeed, if two components A and B, do not depend on each other, then the instantiation of the corresponding program modules can be performed simultaneously, because the instantiation of A does not require the data produced by the instantiation of B and vice versa. The dependency among components is thus the principle underlying the first level of parallelism. At this level subprograms can be evaluated in parallel, but still the evaluation of each subprogram can be further parallelized.

Rules Level [51]. The second level of parallelism, called the *Rules Level*, allows the concurrent evaluation of the rules within each module. A rule r occurring in the module of a component C (i.e., defining some predicate in C) is said to be *recursive* if there is a predicate $p \in C$ occurring in the positive body of r; otherwise, r is said to be an *exit rule.* Rules are evaluated following a semi-naïve schema [31] and the parallelism is exploited for the evaluation of both exit and recursive rules. More in detail, for the instantiation of a module M, first all exit rules are processed in parallel by exploiting the data (ground atoms) computed during the instantiation of the modules which M depends on (according to the partial ordering induced by the dependency graph). Only afterwards, recursive rules are processed in parallel several times by applying a semi-naïve evaluation technique in which, at each iteration n, the instantiation of all the recursive rules is performed concurrently and by exploiting only the significant information derived during iteration $n-1$.

Single Rule Level. The first two levels of parallelism are effective when handling large programs. However, when the input program consists of few rules, their efficacy is drastically reduced, and there are cases where components and rules parallelism are not exploitable at all. For instance the following program $\mathcal{P}$ encoding the well-known 3-colorability problem:

$$\begin{array}{ll} (r) & col(X, red) \;\vee\; col(X, yellow) \;\vee\; col(X, green) \;\text{:-}\; node(X). \\ (c) & \text{:-}\; edge(X, Y), col(X, C),\; col(Y, C). \end{array}$$

The two levels of parallelism described above have no effects on the evaluation of $\mathcal{P}$. Indeed, this encoding consists of only two rules which have to be evaluated sequentially, since, intuitively, the instantiation of (r) produces the ground atoms with predicate *col*, which are necessary for the evaluation of (c).

For the instantiation of this kind of programs a third level is necessary for the parallel evaluation of each single rule, which is therefore called *Single Rule*

Level.

In the following we present a strategy for parallelizing the evaluation of a rule. The idea is to partition the extension of a single rule literal (hereafter called *split* literal) into a number of subsets. Thus the rule instantiation is divided into a number of smaller similar tasks each of which considers as extension of the split literal only one of those subsets. For instance, the evaluation of rule (c) in the previous example can be performed in parallel by partitioning the extension of one of its literals, let it be *edge*, into n subsets, thus obtaining n instantiation tasks for (c), working with different ground instances of *edge*. Note that, in general, several body literals are possible candidates to be split up (for instance, in the case of (c), *col* can be split up instead of *edge*) and the choice of the most suitable literal to split has to be carefully made, since it may strongly affect the cost of the instantiation of rules. Indeed, a "bad" split might reduce or neutralize the benefits of parallelism, thus making the overall time consumed by the parallel evaluation not optimal (and, in some corner cases, even worse than the time required to instantiate the original encoding). Note also that, the partitioning of the extension of the split literal has to be performed at run-time. Indeed, if the predicate to split is an IDB predicate, as in the case of *col*, the partitioning can be made only when the extension of the IDB predicate has already been computed; in our example, only after the evaluation of rule (r).

Example 4.1.1. In the following is reported a complete example of the application of the single rule parallelism for computing in parallel the instantiation of a rule. Consider the following program $\mathcal{P}$ encoding the 3-Colorability problem:

$$\begin{array}{ll} (r) & col(X, red)\ \vee\ col(X, yellow)\ \vee\ col(X, green) \text{ :- } node(X). \\ (c) & \text{:- } edge(X,Y), col(X,C),\ col(Y,C). \end{array}$$

Assume that, after the instantiation of rule r, the extensions of predicate *node* and predicate *col*, are the ones reported in Table 4.1, and that the extension of the predicate *edge* is the one reported in Table 4.2.

Suppose now that the heuristics suggests to perform the single rule level of parallelism for the instantiation of the constraint (c), and suppose that the extension of predicate *edge* is split in two. Then, the extension of the predicate *edge* is partitioned into two subsets which appear divided by an horizontal line in Table 4.2. The instantiation of constraint (c) is carried out in parallel by two separate processes, say p_1, and p_2, which will consider as extension of *edge*, respectively, the two splits depicted in Table 4.2. Process p_1 produces the following ground constraints:

$$\begin{array}{l} \text{:-}edge(a,b), col(a,red),\ col(b,red). \\ \text{:-}edge(a,b), col(a,yellow),\ col(b,yellow). \\ \text{:-}edge(a,b), col(a,green),\ col(b,green). \\ \text{:-}edge(b,c), col(a,red),\ col(b,red). \\ \text{:-}edge(b,c), col(a,yellow),\ col(b,yellow). \\ \text{:-}edge(b,c), col(a,green),\ col(b,green). \end{array}$$

whereas, process p_2 produces the following ground constraints:

	Predicates extension
1	$node(a)$
2	$node(b)$
3	$node(c)$
4	$node(d)$
1	$col(a, red)$
2	$col(a, yellow)$
3	$col(a, green)$
4	$col(b, red)$
5	$col(b, yellow)$
6	$col(b, green)$
7	$col(c, red)$
8	$col(c, yellow)$
9	$col(c, green)$
10	$col(d, red)$
11	$col(d, yellow)$
12	$col(d, green)$

Table 4.1: Extension of the predicate *node* and *col*.

$$
\begin{array}{l}
:- edge(b, d), col(b, red),\ col(d, red).\\
:- edge(b, d), col(b, yellow),\ col(d, yellow).\\
:- edge(b, d), col(b, green),\ col(d, green).\\
:- edge(c, d), col(c, red),\ col(d, red).\\
:- edge(c, d), col(c, yellow),\ col(d, yellow).\\
:- edge(c, d), col(c, green),\ col(d, green).
\end{array}
$$

	Predicate extension
1	$edge(a, b)$
2	$edge(b, c)$
3	$edge(b, d)$
4	$edge(c, d)$

Table 4.2: Extension of the predicate *edge*.

□

4.2 The Algorithms

In this section we present the algorithms for the three levels of parallelism mentioned above, which build upon the one presented for serial instantiation in Section 3.1. The parallel algorithms, detailed in Figure 4.1, and Figure 4.2, repeatedly apply a pattern similar to the classical producer-consumers problem. A *manager* thread (acting as a producer) identifies the tasks that can be performed in parallel and delegates their instantiation to a number of *worker* threads (acting as consumers).

More in detail, the *Components_Instantiator* procedure (see Fig. 4.1), acting as a manager, implements the first level of parallelism, that is, the parallel evaluation of program modules. It receives as input both a program $\mathcal{P}$ to be instantiated and its Dependency Graph $G(\mathcal{P})$; and it outputs a set of ground rules Π, such that $ANS(\mathcal{P}) = ANS(\Pi \cup EDB(\mathcal{P}))$. First of all, the algorithm creates a new set S of atoms that will contain the subset of the base significant for the instantiation; more in detail, S will contain, for each predicate p in the program, the extension of p, that is, the set of all the ground atoms having the predicate name of p (significant for the instantiation).

Initially, $S = EDB(\mathcal{P})$, and $\Pi = \emptyset$. Then, the manager checks whether some SCC C can be instantiated; in particular, it checks whether there is any other component C' that has not been evaluated yet and such that C depends on C'. As soon as a component C is processable, a new thread is created, by a call to threading function *Spawn*, running procedure *Rules_Instantiator*.

Procedure *Rules_Instantiator* (see Fig. 4.1), implementing the second level of parallelism, takes as input, among the others, the component C to be instantiated and the set S; for each atom a belonging to C, and for each rule r defining a, it computes the ground instances of r containing only atoms that can possibly be derived from $\mathcal{P}$. At the same time, it updates the set S with the atoms occurring in the heads of the rules of Π. To this end, each rule r in the program module of C is processed by calling procedure *SingleRule_Instantiator*.

It is worth noting that exit rules are instantiated by a single call to *SingleRule_Instantiator*, whereas recursive ones are processed several times according to a semi-naïve evaluation technique [31], where at each iteration n only the significant information derived during iteration $n-1$ is used. This is implemented by partitioning significant atoms into three sets: ΔS, S, and $\mathcal{N}S$. $\mathcal{N}S$ is filled with atoms computed during current iteration (say n); ΔS contains atoms computed during previous iteration (say $n-1$); and, S contains the ones previously computed (up to iteration $n-2$).

Initially, ΔS and $\mathcal{N}S$ are empty; the exit rules contained in the program module of C are evaluated and, in particular, one new thread identified by $\mathcal{I}_r$ for each exit rule r, running procedure *SingleRule_Instantiator*, is spawned. Since the evaluation of recursive rules has to be performed only when the instantiation of all the exit rules is completed, a synchronization barrier is exploited. This barrier is encoded (à la POSIX) by several calls to threading function *join_with_thread* forcing *Rules_Instantiator* to wait until all *SingleRule_Instantiator* threads are done. Then, recursive rules are processed (do-while loop). At the beginning of each iteration, $\mathcal{N}S$ is assigned to ΔS, i.e. the new information derived during iteration n is considered as significant information for iteration $n+1$. Then, for each recursive rule, a new thread is spawned, running procedure *SingleRule_Instantiator*, which receives as input S and ΔS; when all threads terminate (second barrier), ΔS is added to S (since it has already been exploited). The evaluation stops whenever no new information has been derived (i.e. $\mathcal{N}S = \emptyset$). Eventually, C is removed from $G(\mathcal{P})$.

The third level of parallelism (see Fig. 4.2), concerning the parallel evaluation of each single rule, is then implemented by Procedure *SingleRule_Instantiator*, which given the sets S and ΔS of atoms that are known to be significant up to now, builds all the ground instances of r and adds them to Π. Initially, *SingleRule_Instantiator* selects, according to a heuristics for load balancing (see Section 4.4) the number s of parts into which the evaluation has to be divided;

```
**Procedure** *Components_Instantiator*($\mathcal{P}$; $G(\mathcal{P})$)
**Input**: a program $\mathcal{P}$, the dependency graph $G(\mathcal{P})$ associated with $\mathcal{P}$
**Output**: a ground program $\Pi$
**Var**:
  $S$: **set** of atoms;
  $C$: **set** of predicates;
  $\Pi$: **set** of ground rules;
**begin**
  1.  $S = EDB(\mathcal{P})$; $\Pi := \emptyset$;
  2.  **while** $G(\mathcal{P}) \neq \emptyset$ **do**
  3.      take a SCC $C$ from $G(\mathcal{P})$ that does not depend on other SCCs of $G(\mathcal{P})$
  4.      $Spawn(Rules\_Instantiator, \mathcal{P}, C, S, \Pi, G(\mathcal{P}))$
  5.  **end while**
  6.  **return** $\Pi$;
**end**;

**Procedure** *Rules_Instantiator*($\mathcal{P}$; $C$; $S$; $\Pi$; $G(\mathcal{P})$)
**Input**: a program $\mathcal{P}$, a set of predicates $C$, a set of atoms $S$, a ground program $\Pi$,
        the dependency graph $G(\mathcal{P})$ associated with $\mathcal{P}$
**Var**:
  $\Delta S$, $\mathcal{N}S$: **set** of atoms;
**begin**
  1.  $\Delta S := \emptyset$; $\mathcal{N}S := \emptyset$ ;
  2.  **for each** $r \in Exit(C, \mathcal{P})$ **do**          // evaluation of exit rules
  3.      $\mathcal{I}_r = Spawn\ (SingleRule\_Instantiator, r, S, \Delta S, \mathcal{N}S, \Pi)$;
  4.  **for each** $r \in Exit(C, \mathcal{P})$ **do**          // synchronization barrier
  5.      $join\_with\_thread(\mathcal{I}_r)$;
  6.  **do**
  7.      $\Delta S := \mathcal{N}S$; $\mathcal{N}S := \emptyset$ ;
  8.      **for each** $r \in Recursive(C, \mathcal{P})$ **do**    // evaluation of recursive rules
  9.          $\mathcal{I}_r = Spawn\ (SingleRule\_Instantiator, r, S, \Delta S, \mathcal{N}S, \Pi)$;
  10.     **for each** $r \in Recursive(C, \mathcal{P})$ **do**    // synchronization barrier
  11.         $join\_with\_thread(\mathcal{I}_r)$;
  12.     $S := S \cup \Delta S$;
  13. **while**  $\mathcal{N}S \neq \emptyset$
  14. Remove $C$ from $G(\mathcal{P})$;                        // to process C only once
**end**;
```

Figure 4.1: Components and Rules parallelism

then *SingleRule_Instantiator* heuristically selects a positive literal to split in the body of r, say L (see Section 4.3). A call to function *SplitExtension* (Figure 4.2) partitions the extension of L (stored in S and ΔS) into s equally sized parts, called splits. In order to avoid useless copies, each split is virtually identified by means of iterators over S and ΔS, representing ranges of instances. More in detail, for each split, a VirtualSplit is created containing two iterators over S (resp. ΔS), namely *S_begin* and *S_end* (resp. Δ*S_begin* and Δ*S_end*), indicating the instances of L from S (resp. ΔS) that belong to the split. Note that, in general, a split may contain ground atoms from both S and ΔS. Once the extension of the split literal has been partitioned, then a number of threads

```
// A split is virtually identified by four iterators to $S$ and $\Delta S$ identifying
ranges of instances.
struct VirtualSplit { iterator⟨Atom⟩ S_begin, S_end, ΔS_begin, ΔS_end; }

Procedure SingleRule_Instantiator($r$; $S$; $\Delta S$; $\mathcal{N}S$; $\Pi$)
Input: a rule $r$, sets of atoms $S$, $\Delta S$, $\mathcal{N}S$, a ground program $\Pi$;
Var:
  $s$: integer;
  Splits: vector of VirtualSplit;
begin
  1.   $s$ := numberOfSplits($B(r), S, \Delta S$);
  2.   SelectLiteralToSplit($L$,$B(r)$,$s$);          // according to a heuristics
  3.   Splits[$s$];
  4.   SplitExtension($L$, $s$, $S$, $\Delta S$, Splits);        // distribute extension of L
  5.   for each sp in Splits
  6.        $\mathcal{I}_{sp}$ = Spawn (InstantiateRule, $r, L, sp, S, \Delta S, \mathcal{N}S, \Pi$);
  7.   for each sp in Splits do               // synchronization barrier
  8.        join_with_thread($\mathcal{I}_{sp}$);
end;

Procedure InstantiateRule( $r$; $L$; $sp$; $S$; $\Delta S$; $\mathcal{N}S$; $\Pi$)
Input: a rule $r$, literal $l$, a VirtualSplit sp, sets of atoms $S$, $\Delta S$, $\mathcal{N}S$,
a ground program $\Pi$;
/*  Given $S$ and $\Delta S$ builds all the ground instances of $r$, adds them to $\Pi$,
    and add to $\mathcal{N}S$ the new head atoms of the generated ground rules.
    For $L$ only the ground atoms belonging to the ranges {S_Begin,S_End} .
    and {ΔS _begin,ΔS _end} indicated by sp are used .
*/
```

Figure 4.2: Single Rule parallelism

running procedure *InstantiateRule*, are spawned. *InstantiateRule*, given S and ΔS builds all the ground instances of r that can be obtained by considering as extension of the split literal L only the ground atoms indicated by the iterators in the virtual split at hand. *SingleRule_Instantiator* terminates (last barrier) once all splits are evaluated.

The correctness of the algorithm follows from the consideration that, whatever the split literal L, the union of the outputs of all the s concurrent *InstantiateRule* procedures is the same as the output produced by a single call to *InstantiateRule* working with the entire extension of L ($s = 1$). Note that, if the split predicate is recursive, its extension may change at each iteration. This is considered in our approach by performing different splits of recursive rules at each iteration. This ensures that at each iteration the virtual splits are updated according to the actual extension of the literal to split.

In addition, this choice has a relevant side-effect: at each iteration the workload is dynamically re-distributed among instantiators, thus inducing a form of dynamic load balancing in case of the evaluation of recursive rules.

```
**Procedure** *SplitExtension*($l$; $n$; $S$; $\Delta S$; $V$)
**Input**: a literal $l$, an integer $n$, sets of atoms $S$, $\Delta S$, a vector of VirtualSplit $V$;
**Var**:
  $size$, $i$, $k$: **integer**;
  $it$: **iterator**$\langle Atom \rangle$;
**begin**
 1.   $size$:= $\lfloor$ ($S$.size() + $\Delta S$.size())/ $n$ $\rfloor$;
 2.   $i$:= 0;   $it$ := $S$.begin();
 3.   while $i < \lfloor S$.size()$/size \rfloor$ do   // possibly, build splits with atoms from S
 4.        $V[i]$.SetIterators_S($it$, $it+size$);   $it$ := $it$ + $size$;   $i$=$i$+1;
 5.   end while
 6.   if $it < S$.end() then   // possibly, build a split mixing S and ΔS atoms
 7.        $V[i]$.SetIterators_S($it$, $S$.end());   $it$ := $\Delta S$.begin();
 8.        $k$ := $size$ - Size($V[i]$);
 9.        if $\Delta S$.size() $< k$
10.             $V[i]$.SetIterators_$\Delta S$ ($it$, $\Delta S$.end());   $it$ = $\Delta S$.end();
11.        else
12.             $V[i]$.SetIterators_$\Delta S$ ($it$, $it+k$);   $it$ = $it+k$;   $i$ = $i$+1;
13.   while $i < \lfloor (S$.size()+$\Delta S$.size())$/size \rfloor$ do   // possibly, from ΔS
14.        $V[i]$.SetIterators_$\Delta S$ ($it$, $it+size$);   $it$ := $it$ + $size$;   $i$=$i$+1;
15.   end while
16.   if $it < \Delta S$.end() then
17.        $V[i]$.SetIterators_$\Delta S$ ($it$, $\Delta S$.end());
**end**
```

Figure 4.3: Splitting the extension of a literal.

4.3 Selection of the Literal to Split

In this section we discuss the problem of determining the literal to be split. Indeed, this choice has to be carefully made, since it may strongly affect the cost of the instantiation of rules. It is well-known that this cost strongly depends on the order of evaluation of body literals, since computing all the possible instantiations of a rule is equivalent to computing all the answers of a conjunctive query joining the extensions of literals of the rule body. However, the choice of the split literal may influence the time spent on instantiating each split rule, whatever the join order. In the light of these observations, we have devised a new heuristics for selecting the split literal *given an optimal ordering* (which can be obtained as explained in [27]).

Intuitively, suppose we have a rule r containing n literals in the body in a given order, and suppose that any body literal allows for the target number of splits, say s, then: to obtain work for s threads it is sufficient to split the first literal (whatever the join order); nonetheless, moving forward, say splitting the third literal, would cause a replicate evaluation of the join of the first two literals in each split thread possibly increasing parallel time. It is easy to see that the picture changes if all/some body literal does not allow for the target number of splits, in this case one should estimate the cost of splitting a literal different from the first and select the best possible choice.

In the following, we first introduce some metrics for estimating the work needed for instantiating a given rule, and then we describe the new heuristics. In detail, we use the following estimation for determining the size of the joins of the body literals: given two relations R and S, with one or more common variables, the size of $R \bowtie S$ can be estimated as follows:

$$T\left(R \bowtie S\right) = \frac{T\left(R\right) \cdot T\left(S\right)}{\prod_{X \in var(R) \cap var(S)} \max\left\{V\left(X, R\right), V\left(X, S\right)\right\}} \tag{4.1}$$

where $T\left(R\right)$ is the number of tuples in R, and $V\left(X, R\right)$ (called selectivity) is the number of distinct values assumed by the variable X in R. Given an evaluation order of body literals, one can repeatedly apply this formula to pairs of body predicates for estimating the size of the join of a body. A more detailed discussion on this estimation can be found in [31].

Let r be a rule with n body literals $L_1, L_2, \ldots, L_n$, where L_i precedes L_j for each $i < j$ in a given evaluation order, an estimation of the cost of instantiating the first k literals in $B(r)$ is:

$$\mathcal{C}(k) \;=\; \begin{cases} 0 \quad if\ k < 2 \\ T(L_1) \cdot T(L_2) \quad if\ k = 2 \\ \mathcal{C}(k-1) + T(L_1 \bowtie \cdots \bowtie L_{k-1}) \cdot T(k) \quad if\ k > 2 \end{cases} \tag{4.2}$$

Now, let s be the number of splits to be performed; the following is an estimation of the work of the instantiation tasks obtained by the split of the i-th literal L_i:

$$\mathcal{C}^i \;=\; \frac{\mathcal{C}(n) - \mathcal{C}(i-1)}{s^i} + \mathcal{C}(i-1), \quad 1 \leq i \leq n \tag{4.3}$$

where, s^i is equal to s (if the extension of L_i is sufficiently large) or the maximum number of splits allowed by L_i. Intuitively, if L_i is the split literal, the work of each instantiation task is composed of two parts: a part to be performed serially, common to all tasks, which consists in the instantiation of the first $i-1$ literals, whose cost is represented by $\mathcal{C}(i-1)$; and the instantiation of the remaining literals, which is divided among the s^i tasks, whose cost is represented by $\frac{\mathcal{C}(n)-\mathcal{C}(i-1)}{s^i}$. The estimation $\mathcal{C}^i$ can be used for determining the split literal, by choosing the one with minimum cost.

Note that, in the search for the best one, we can skip over each body literal L_k, with $k > j$, if L_j allows for s splits since $\mathcal{C}^j \leq \mathcal{C}^k$ holds. Indeed, if $n = 2$, $\mathcal{C}^1 = \mathcal{C}^2$; while for $n \geq 3$, $k = j+1$ and $s^j = s^k = s$ (worst case) we have that

$$\mathcal{C}^k \;=\; \frac{\mathcal{C}(n) - \mathcal{C}(k-1)}{s^k} + \mathcal{C}(k-1) = \frac{\mathcal{C}(n) - \mathcal{C}(j)}{s} + \mathcal{C}(j).$$

By applying (4.2):

$$\mathcal{C}^k = \frac{\mathcal{C}(n) - \mathcal{C}(j-1)}{s} - \frac{Q}{s} + \mathcal{C}(j-1) + Q = \mathcal{C}^j + \frac{s-1}{s} \cdot Q$$

where $Q = T(L_1 \bowtie \cdots \bowtie L_{j-1}) \cdot T(j)$. Thus, by induction, if L_j allows for s splits, $\mathcal{C}^j \leq \mathcal{C}^k$, for $k > j$. Intuitively, this can be explained by considering that splitting a literal L_k after one allowing for s splits L_j has the effect of evaluating serially the join of literals between L_k and L_j thus leading to a

Table 4.3: Number of splits and costs of the instantiation tasks

s	s^1	s^2	s^3	s^4	$\mathcal{C}^1$	$\mathcal{C}^2$	$\mathcal{C}^3$	$\mathcal{C}^4$
5	5	5	5	5	**11400**	11400	12200	17000
50	20	50	50	50	2850	**1140**	2120	8000
100	20	50	100	100	2850	**1140**	1560	7500
500	20	50	500	500	2850	1140	**1112**	7100

greater evaluation time. Clearly, even a literal L whose extension cannot be split in s parts can be chosen, provided that L allows for a minor (estimated) work for each instantiation task. Moreover, if $s^1 = s$ (L_1 allows for s splits), it holds that $\mathcal{C}^1 \leq \mathcal{C}^i$, for each $1 < i \leq n$; in this case, L_1 can be chosen without computing any cost.

As an example, suppose that we have to instantiate the constraint $:\!-a(X,Y)$, $b(Y,Z)$, $c(Z,X)$, $d(V,Z)$. Suppose also that the extensions of the body literals are $T(a) = 20$, $T(b) = 50$, $T(c) = 1000$, $T(d) = 1000$, and that the estimations of the costs of instantiating the first i literals with $1 \leq i \leq 4$ are the following: $\mathcal{C}(1) = 0$, $\mathcal{C}(2) = 1000$, $\mathcal{C}(3) = 7000$, $\mathcal{C}(4) = 57000$. Table 4.3 shows the estimations $\mathcal{C}^i$ of the works of the instantiation tasks obtained by the split of the i-th literal with $1 \leq i \leq 4$, by varying the target number s of splits. In particular, the first column shows the target number of splits, the following four columns show the maximum number of splits s^i allowed for each literal, and the remaining four columns show the costs $\mathcal{C}^i$ computed according to the s^i values; in bold face we outline, for each target number of splits, the minimal values of $\mathcal{C}^i$. It can be noted that, in our example, increasing the value of s corresponds to different choices of the literal to split. Moreover, in each row, the choice is always restricted to the first i literals, where the ith literal is the first one allowing for the target number s of splits; indeed, $\mathcal{C}^i \leq \mathcal{C}^k$, for each $k > i$. Furthermore, even a literal that does not allow for s splits can be chosen; this is the case for $s = 100$, where the chosen literal is $b(Y,Z)$, which allows for 50 splits. Notice that the choice of the literal to split may be influenced by the body ordering in some cases, which in turn considerably affects the serial evaluation time (which is the amount to be divided by parallel evaluation). For example, all body orderings having $d(V,Z)$ as first literal have d as the chosen literal, since its extension is sufficiently large to allow the four target numbers of splits considered. However, if such orderings determine a higher evaluation cost w.r.t. the body order exploited in the serial evaluation, then the effect of parallel evaluation could be overshadowed. Thus, we apply the selection of the literal to split after reordering the body with a strategy that minimizes the heuristic cost of instantiating the body.

Summarizing, our heuristics consists in determining an ordering of the body literals exploiting the already assessed technique described in [27] and splitting the first literal in the body if it allows for the target number of splits (without computing any cost). Otherwise, the estimations of the costs are determined and the literal allowing for the minimum one is chosen.

4.4 Load Balancing and Granularity Control

An advanced parallelization technique has to deal with two important issues that strongly affect the performance of a real implementation: load balancing and granularity control. Indeed, if the workload is not uniformly distributed to the available processors then the benefits of parallelization are not fully obtained; moreover, if the amount of work assigned to each parallel processing unit is too small then the (unavoidable) overheads due to creation and scheduling of parallel tasks might overcome the advantages of parallel evaluation (in corner cases, adopting a sequential evaluation might be preferable).

As an example, consider the case in which we are running the instantiation of a rule r on a two processor machine and, by applying the technique for Single Rule parallelism described above, the instantiation of r is divided into two smaller tasks, by partitioning the extension of the split predicate of r into two subsets with, approximatively, the same size. Then, each of the two tasks will be processed by a thread; and the two threads will possibly run separately on the two available processors. For limiting the inactivity time of the processors, it would be desirable that the threads terminate their execution almost at the same time. Unfortunately, this is not always the case, because subdividing the extension of the split predicate into equal parts does not ensure that the workload is equally spread among threads. However, if we consider a larger number of splits, a further subdivision of the workload would be obtained, and, the inactivity time would be more likely limited.

Clearly, it is crucial to guarantee that the parallel evaluation of a number of tasks is not more time-consuming than their serial evaluation (granularity control); and that an unbalanced workload distribution does not introduce significant delays and limits the overall performance (load balancing).

Granularity Control. Our method for granularity control is based on the use of a heuristic value $\mathcal{W}(r)$, which acts as a litmus paper indicating the amount of work required for evaluating each rule of the program, and so, its "hardness", just before its instantiation. $\mathcal{W}(r)$ denotes the value $\mathcal{C}(n)$ (see Section 4.3), for each rule r having n body literals.

At the rules level, rather than assigning each rule to a different thread, a set of rules R is determined and assigned to a thread. R is such that the total work for instantiating its rules is enough for enjoying the benefits of scheduling a new thread. In practice, R is constructed by iterating on the rules of the same component, and stopping if either $\sum_{r \in R} \mathcal{W}(r) > w_{seq}$ or when no further rules can be added to R, where w_{seq} is an empirically-determined threshold value. At the single rule level, a rule r is scheduled for parallel instantiation (i.e. its evaluation can be divided into smaller tasks that can be performed in parallel) if $\mathcal{W}(r) > w_{seq}$; otherwise, for r the third level of parallelism it is not applied at all.

Note that, for simplifying the presentation of the algorithms in Section 2.3, we have not considered the management of the granularity control in the second level of parallelism, which would have added noisy technical details and made the description more involved. However, they can be suitably adapted by modifying procedure *Rules_Instantiator* (see Fig.4.1) in order to build a set of "easy" rules, and by adding a *SetOfRules_Instantiator* procedure, which instantiates each rule in the set. Note also that, granularity control in the third level of parallelism is obtained by setting the number of splits of a given rule to 1.

Load Balancing. In our approach load balancing exploits different factors. On the one hand, in the case of the evaluation of recursive rules, a dynamic load redistribution of the extension of the split literal at each iteration is performed. On the other hand, the extension of the split literal is divided by a number which is greater than the number of processors (actually, a multiple of the number of processors is enough) for exploiting the preemptive multitasking scheduler of the operating system. Moreover, in case of "very hard" rules, a finer distribution is performed in the last splits. In particular, when a rule is assessed to be "hard" by comparing the estimated work (the value $\mathcal{W}(r)$ described above) with another empirically-determined threshold ($\mathcal{W}(r) > w_{hard}$), a finer work distribution (exploiting a unary split size) is performed for the last $s - n_p$ splits, where s is the number of splits and n_p is the number of processors. The intuition here is that, if a rule is hard to instantiate then it is more likely that its splits are also hard, and thus an uneven distribution of the splits to the available processors in the last part of the computation might cause a perceptible loss of efficiency. Thus, further subdividing the last "hard" splits, may help to distribute the workload in a finer way in the last part of the computation.

Chapter 5

Parallel Computation of Answer Sets for Propositional Programs

In this chapter we describe the design of parallel techniques devised for the computation of the answer sets.

The propositional answer sets computation engine is an important module of an ASP system; surely, its functioning is very expensive in terms of computational resources. In Section 3.2 is described the propositional search algorithm implemented in the ASP system DLV. This algorithm is serial, and, even if it employs sophisticated techniques, and effective heuristics, the advantages of applying parallelism to this phase are still worthy of attention.

This Chapter is structured as follows: in Section 5.1 we introduce the problem of pushing parallelism into the model generation process; in Section 5.2 we describe the parallel multi-heuristics search we have devised for the ASP system DLV; in Section 5.3 we present a parallel lookahead technique; finally, in Section 5.4, we introduce a technique for performing answer set checking in parallel.

5.1 Parallel techniques for propositional ASP solvers

In general, the answer set computation is a two-step process: the instantiation step produces a propositional program which is then the input of a subsequent propositional search phase which finds the answer sets. More in detail, the propositional search consists in the generation of candidate solutions (model generation), which are than checked to verify whether they are answer sets (model checking).

In this chapter we discuss the issues related to both tasks of the propositional search process. In particular, we analyse the crucial operations performed during the computation of the answer sets of propositional programs in order to point out which of them may particularly benefit from the application of parallelism. In the following we explicitly refer to the propositional search algorithm employed by the ASP system DLV [34].

First of all, note that practical implementations devised to solve difficult problems (i.e., NP-Complete or even harder problems), are based on heuristic criteria that drive the search in order to rapidly reach solutions immersed in a large search space. As a matter of fact, the heuristics are decisive for the design of a competitive and performant system. The majority of ASP systems (including DLV) allows the static selection of a heuristic function when invoking the solver; this makes the solvers dependent on human supervision. Indeed, a deep understanding of the problem domain is fundamental for selecting the most appropriate heuristics for the problem at hand. Knowing the best possible heuristic criterion in advance is not possible (even if there are approaches based on machine learning that partially address this issue [52]), but parallelism can help here allowing for the concurrent involvement of several criteria. Moreover, the computation of the heuristic values is one of the most expensive tasks of model generation in the case of look-ahead-based solvers such as DLV. Indeed, the number of lookahead steps, which suppose the computation of these values, performed during the search space traversal is quite large (roughly, note that every branching variable is, in principle, a candidate choice to be looked-ahead thus the number of calls might be exponential in the size of the input program). Parallelism can be exploited in this step by performing a concurrent measure of heuristic values.

The last step of the computation is answer set checking, which amounts to verifying, given a candidate model, whether it represents an answer set or not. The stability check is a complex task: in the case of a non-HCF programs it is Co-NP-complete, whereas for HCF programs it is polynomial time computable. Concerning the Co-NP task, described in Section 3.2.3, it is traditionally performed translating the input program into an SAT formula and evaluating its satisfiability with an integrated SAT Solver; the polynomial time check is performed computing the fixpoint of the $\mathcal{R}$ operator described in Subsection 3.2.3. Nonetheless, the number of model checks performed during the propositional search process is significant, thus exploiting parallelism , which is worthwhile in both cases.

Given these premises, we designed the following parallel evaluation strategies for the ASP solver DLV:

- a parallel multi-heuristics search in which a number of search tasks are run concurrently, each one employing a different heuristics; the fastest among them computes and outputs the answer set;

- a parallel lookahead technique that allows the concurrent computation of the heuristic values.

- a parallel model checking strategy including both the parallel computation of the the fixpoint of the $\mathcal{R}$ operator (see Chapter 3), and the use of a parallel SAT solver in the cases in which model checking is Co-NP.

In the following sections we describe the techniques mentioned before in detail.

```
Function MultiHeuristicsSearch($H$)
Input: An array of heuristic function $H$;
Output: An answer set $A$ or the set of all literals $\mathcal{L}$;
Var:
 $I$: a array of interpretation
 $A$: a thread safe interpretation
begin
 1.    foreach $j < size(H)$ do
 2.        $I[j] := \emptyset$;
 3.        Spawn(ModelGenerator, $I[j]$, $A$, $H[j]$);
 4.    $wait(someAnswerSetHasBeenFound)$;
 5.    $broadcast(searchEnd)$;
 6.    return $A$;
end
```

Figure 5.1: Framework for the parallel multi heuristics search DLV

5.2 Multi-Heuristics Parallel Search Strategy

The heuristics for the selection of the branching literal dramatically affects the performance of an ASP system; and, obviously, there is no heuristics performing well in all cases, rather one heuristics can be more suitable than another for a given problem typology, or even for a specific problem instance. This problem may be overcome adopting a parallel strategy for exploring the search space. To this end, we designed a multi-heuristics search strategy that builds upon the model generator schema described in Section 3.2. More in details (see Figure 5.1), we allow the system to run n of instance of answer set computation engines running n processes (Lines 1-3) executing the function *ModelGenerator* depicted in Figure 3.4. Each one of these processes adopts a different branching criterion among the ones described in Section 3.2.2. To this aim, the *ModelGenerator* function is modified in order to receive also a heuristic function in input and a thread safe interpretation. The computation stops when the first process running *modelGenerator* finds a solution or determines an inconsistency. In particular, the thread safe interpretation is used to store the result of the multi-heuristics search, and is written only by the thread that terminate the computation; in case of inconsistency, A becomes the set of all literals $\mathcal{L}$. Basically, we address the problem of finding a single answer set of the input program, if it exists. Thus, as soon as a process stops its computation a terminating message is sent to the remaining processes (Line 5), and the function return the solution. This multi-heuristics parallel search strategy ensures that the best possible performance for the given problem instance is obtained as long as enough computational resources are employed. Indeed, the first propositional search task that finds a solution is the one employing the most suitable heuristic criterion for the input problem. A prototype of this solver was implemented; implementation details, and performances will be discussed in Chapter 6.

```
Function SelectPar($I$,$s$)
Input: An interpretation $I$, an integer $s$;
Output: a literal $L$;
Var:
  $L$: a literal
  $id$: thread_id;
  $thList$: list of thread id;
  $AS$: set of literals;
  $DetChoices$: thread safe set of literals;
begin
 1.   $thList := \emptyset$;
 2.   $DetChoices := \emptyset$;
 3.   foreach Literal $A \in PT_{\mathcal{P}}(I)$ do          (* spawn lookahead threads *)
 4.        if $AS$.size = $\lfloor PT_{\mathcal{P}}(I)/s \rfloor$ then
 5.             $AS$.add($A$);
 6.        else
 7.             $id$ = Spawn(Lookahead, $I$, $A$, $detChoices$);
 8.             $thList$.add($id$);
 9.        endif
10.   foreach thread_id $thread \in thList$ do     (* barrier *)
11.        join_with_thread($thread$);
12.   $I :=$ Propagate($I \cup detChoices$);
13.   if $I = \mathcal{L}$ then
14.        return $NULL$;
15.   endif
16.   $L := NULL$;
17.   foreach Literal $A \in PT_{\mathcal{P}}(I)$ do
18.        if $L = NULL$ then
19.             $L := A$;                             (* first literal, no comparison *)
20.        else if $L <_{h_C} A$ then
21.             $L := A$;
22.        endif
23.   return $L$;
end
```

Figure 5.2: The function selectPar for the parallel selection of the branching literal

5.3 Parallel Lookahead

As described in Chapter 3, the Model Generator computation proceeds by heuristically selecting a branching literal until the end of the computaion is reached. The selection is made by comparing the heuristic values computed in a look-ahead step for each possible choice. In the following we describe a technique for exploiting parallelism in the look-ahead step.

Our approach builds upon the serial algorithms described in Section 3.2.1. In particular, Figure 5.2 shows the general procedure for evaluating the heuristically best literal in parallel, and is an enhancement of the function *Select* depicted in Figure 3.5. In order to insert our parallel lookahead technique into

```
**Procedure** *Lookahead*($I$;$A$;*detChoices*)
**Input**: An interpretation $I$, a set of literals $AS$, thread-safe set of literals *detChoices*;
**begin**
 1.  **foreach** Literal $A \in AS$ **do**
 2.      $I_A^+ :=$ Propagate($I \cup \{A\}$);            (* look-ahead for A *)
 3.      **if** $I_A^+ = \mathcal{L}$
 4.          detChoices.add(not.A);
 5.      **endif**
 6.      $I_A^- :=$ Propagate($I \cup \{\texttt{not}.A\}$);      (* look-ahead for not.A *)
 7.      **if** $I_A^- = \mathcal{L}$
 8.          detChoices.add(A);
 9.      **endif**
 10.     **if** $I_A^+ = \mathcal{L} \wedge I_A^- = \mathcal{L}$ **then**
 11.         **return**;
 12.     **endif**
**end**
```

Figure 5.3: The Lookahead function the parallel evaluation of the heuristic values

the general ASP computational framework, the algorithm shown in Figure 3.4 is modified by substituting the invocation of the function *Select* with the function *SelectPar*.

The function *SelectPar* receives an interpretation I as input, and an integer s, and proceeds by creating exactly s subsets of size $\lfloor PT_{\mathcal{P}}(I)/s \rfloor$ (lines 4-5) of the set $PT_{\mathcal{P}}(I)$, which contains all the possible choices at a given stage of the computation. If $PT_{\mathcal{P}}(I)$ contains exactly t literals such that $t < s$, only t subsets of size 1 will be generated. For each subset AS a thread [1] is spawned running the function *Lookahead* (line 7), which performs the lookahead for the all the literals contained in AS (Figure 5.3). Lines 10-11, refer to the synchronization that has to be made after all threads finish the computation.

Lookahead threads, either calculate the heuristic values (results are stored in I_A^+ and I_A^-) by calling function Propagate for each literal L in AS, or if an inconsistency is encountered ($I_A^+ = \mathcal{L}$ or $I_A^- = \mathcal{L}$), stores the complement of the propagated literal in the *detChoices* set; note that the set *detChoices* is shared among threads and has to be implemented by exploiting a thread-safe collection. If a thread detects an inconsistency it stops the computation, in order to avoid useless operations (lines 10-12 of function *Lookahead*). Once all threads terminate (they all reach the barrier in Line XX), literals in *detChoices*, which can be deterministically assumed, are propagated. If an inconsistency arises at this point, then no literal can be chosen at this level (the assumption of both A and its complement $\texttt{not}.A$ leads to inconsistency) and the function Select returns false, in order to cause a backtracking. Otherwise, the best literal according to a given heuristic criterion h_C is selected among the candidate ones as in the serial version of this algorithm.

[1] In the description of the thread function calls we adopted here a syntax similar to the one of lightweight POSIX thread (calls to *Spawn* and *join_with_thread*).

5.4 Parallel Model Checking

The serial algorithm used for checking the stability of candidate models has been described in Subsection 3.2.3. Recall that, the stability check is carried out as a two-step process: the first step is a polynomial time check which amounts to compute the fixpoint of the $\mathcal{R}$ described in Subsection 3.2.3. In the case of HCF programs, the first step is sufficient for determining the stability of the model; for non-HCF problems an additional step is necessary.

More in detail, the second step is carried out translating the ASP program into a SAT formula and testing its unsatisfability (which is a Co-NP-complete problem). The technique described in Subsection 3.2.3 employs the invocation of an external SAT solver for the evaluation of the so-obtained formula. This technique might take advantage of parallelism making use of a modern parallel sat solver. Nowadays there are a number of efficient implementations which make use of multi-heuristics [53] approaches as well as parallel tree traversal technique for the parallel evaluation of the satisfiability of a formula [54]. Since proving the unsatisfability is a complex (Co-NP complete) problem, we argue that the use of a parallel SAT solver may produce significant speed up for checking the stability of non-HCF programs.

We describe now a novel parallel technique for enhancing the first step of the stability check. The main idea is to design a parallel strategy for the evaluation the operator $\mathcal{R}$; in particular, a new definition of fixpoint of $\mathcal{R}_{\mathcal{S},\mathcal{I}}$ is given which is based on the function $FixpointOfR$, depicted in Figure 5.4.

Whenever a model check is performed, the $FixpointOfR$ is called instead of $\mathcal{R}^{\omega}_{\mathcal{P},\mathcal{I}}$ at line 1 in the procedure in Figure 3.6 of Chapter 3. This function works under the assumption that a partition of the input program $\mathcal{P}$ is made at the beginning of the propositional search phase, such that n sub-programs are defined which are indicated with P_i. Basically, a cyclic procedure is performed until the fixpoint condition is reached; n thread are spawned performing the fixpoint of the $\mathcal{R}_{\mathcal{S},\mathcal{I}}$, such that each thread works on a partition of the program $S = \mathcal{P}_i$ (Lines 2-7), and modify only a local copy of the global set X (each thread i modifies $XLocal[i]$). Afterwards, a synchronization step is executed, and a check is performed to verify whether at least one of the n $XLocal$ set is different from the global set X (Lines 10-15); as long as this happens another iteration has to be performed, with X sets to the intersection of all the $XLocal$. The computation stops as soon as all the $XLocal$ sets are set to X (fixpoint condition), and X is returned.

In the following we show the correctness of the technique.

Proposition 5.4.1. *The FixpointOfR procedure is equivalent to the computation of the fixpoint of the $\mathcal{R}$ operator.*

Proof. Let consider first a variant of the $FixpointOfR$ function of Figure 5.4, in which the computation of $\mathcal{R}^{\omega}_{\mathcal{S},\mathcal{X}}$ (Line 6) is replaced by single application of the $\mathcal{R}$ operator denoted by $\mathcal{R}^{1}_{S,X}$. The union of all the $\mathcal{R}^{1}_{S,X}$ at the $m-th$ iteration is equal to $\mathcal{R}^{m}_{\mathcal{P},X}$ (a serialized execution of all the $\mathcal{R}^{1}_{S,X}$ coincides with the execution of $\mathcal{R}^{m}_{\mathcal{P},X}$), thanks to the synchronization step (Lines 8-9). Thus an execution of this modified algorithm employing parallelism is equivalent to the serial execution of $\mathcal{R}^{\omega}_{\mathcal{P},\mathcal{X}}$. Now, considering that $\mathcal{R}^{\omega}_{S,X} \subseteq \cdots \subseteq \mathcal{R}^{2}_{S,X} \subseteq \mathcal{R}^{1}_{S,X}$, and that the application of the $\mathcal{R}$ operator can only remove atoms from X,

```
Function FixpointOfR(P,I,n)
Input: A program P, an interpretation I;
Output: a set of atoms X;
Var:
  i: an integer
  continue: a boolean
  S: a sub-program
  L: a literal
  X:set of atoms
  XLocal: an array of set of atoms
  id: thread_id;
  thList: list of thread id;
begin
 1.   $X := I^+$;
 2.   do
 3.       foreach $i < n$ do
 4.           $XLocal[i] := \emptyset$;
 5.           $S := P_i$;
 6.           $id := Spawn(\mathcal{R}^{\omega}_{\mathcal{S},\mathcal{X}}, XLocal[i])$;
 7.           thList.add(id);
 8.       foreach thread_id $thread \in thList$ do          (* barrier *)
 9.           join_with_thread(thread);
10.       $continue := false$;
11.       foreach $i < n$ do
12.           if $X \cap XLocal[i] \neq X$
13.               $continue := true$;
14.               break;
15.           endif
16.       if continue
17.           $X := \bigcap_{0<i<n} XLocal[i]$;
18.       endif
19.   while continue
20.   return X
end
```

Figure 5.4: Parallel computation of the $\mathcal{R}^{\omega}_{\mathcal{P},\mathcal{I}}$ operator.

then $\mathcal{R}^{\omega}_{\mathcal{S},\mathcal{X}}$ removes from X only atoms that would have been removed in a later iteration by the algorithm employing $\mathcal{R}^{1}_{S,X}$. Thus, the algorithm of Figure 5.4 is equivalent to the computation $\mathcal{R}^{\omega}_{\mathcal{P},\mathcal{X}}$.

□

Chapter 6

Experiments and Implementation

The algorithms devised for parallel grounding and parallel model generation have been implemented into the ASP system DLV. In this Chapter we discuss the implementation issues and report the results of the experimental analysis carried out to assess the performances of the implemented prototypes.

This Chapter is structured as follows: in Section 6.1 we describe the implementation details; in Section 6.2 we report the results of the experimental analysis carried out for the parallel instantiator, while in section 6.3 we report the results of the assessment on the parallel model generator.

6.1 Implementation in DLV

The techniques described in Chapter 4–5 have been implemented in the ASP system DLV. In particular, we came out with two prototypes, each one dealing with a different phase of the answers set computation, thus obtaining a *parallel instantiator* and a *parallel model generator*. Both them have been implemented by extending the corresponding module of DLV. The systems are implemented in the C++ language by exploiting the Linux POSIX Thread API, shipped with the GCC 4.3.3 compiler. The description of the algorithm reported in Chapter 4–5 is simplified for presentation reasons; a number of optimization have been adopted in the real implementation in order to make the most of parallelism. In the following we sketch the implementation details for both systems.

6.1.1 Instantiator Implementation

An overview of the system architecture is depicted in Figure 6.1. An input program P first undergoes the parsing procedure, which maps P into the internal data structures of DLV. Moreover, the dependency graph associated with the program is created, and the corresponding program modules identified. Then, a number of threads are spawned: n_c threads working at the components level,

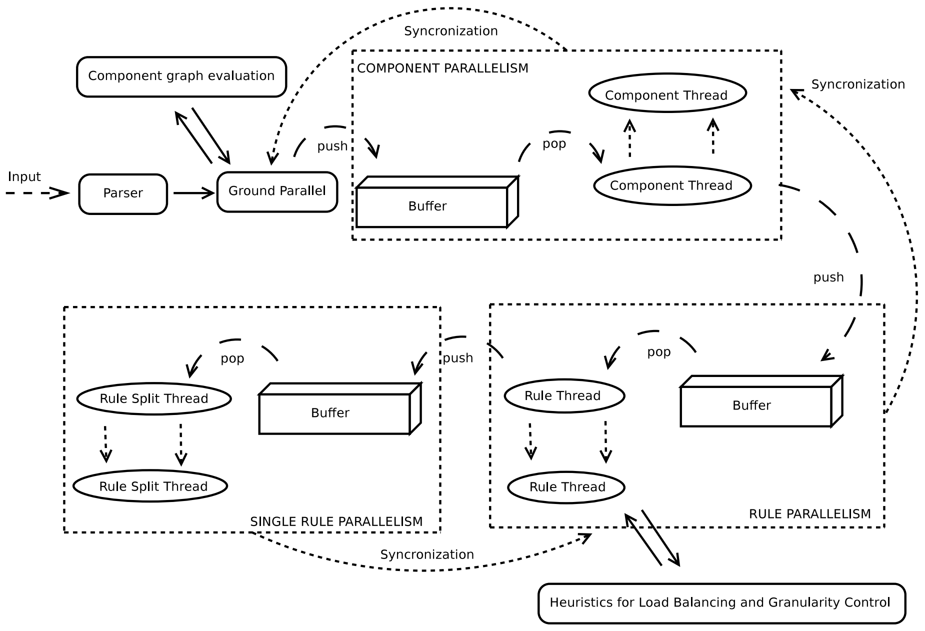

Figure 6.1: System Architecture of the Parallel Instantiator

n_r threads working at the rules level, and n_s threads working at the single rule level. Afterwards the master thread, which is responsible for managing the whole evaluation process, starts the instantiation process. Basically, it exploits the information on modules dependency to schedule independent components in the component buffer. The component threads process the components placed in the component buffer according to the strategy explained in the previous section. Each rule of the component that has to be processed is pushed into the rule buffer. Then each component thread waits until all scheduled rules are instantiated; and, in presence of recursive rules it reiterates the rules scheduling until the end of the semi-naïve algorithm. When all the rules of a component are completely instantiated the component thread notifies that the component has been processed to the master thread that might continue by scheduling the remaining components (if any) or stop the instantiation. Similarly, rule threads take the rules to be processed from the rule buffer, notify to the originating component threads when the rule-instantiation task is concluded. In accordance with the heuristics described in the previous section each rule instantiator thread chooses among sequential evaluation or dynamic rewriting. Rewritten rules are pushed into a third buffer and processed by split threads. Split threads, in turn, take a split rule from the buffer and call the original rule instantiation procedure of the DLV system; when the the instantiation task is concluded the originating rule thread is notified.

6.1.2 Model Generator Implementation

The parallel model generator takes a ground program $\mathcal{P}$ as input and outputs its answer sets (or one solution in case the multi-heuristics is applied). Concerning the multi-heuristics implementation, a number of thread is spawned at the beginning of the computation, each of which runs an instance of the serial propositional search solver instructed to exploit a specific branching heuristic criterion; in particular, each thread runs the *ModelGenerator* function described in Section 3.2. Thus, the main process waits for the threads to terminate their task; a synchronized global flag variable is used to share the information that a solution has been found by a concurrent solver thread. As soon as a thread finds the solution, it acquires a lock, sets the termination flag to *true* and prints the answer set. In particular, the termination flag is checked every time a choice point is reached to determine whether the computation has to continue. Once every thread terminates the computation is stopped.

The implementation of the parallel lookahead follows the pseudo-code algorithm described in Section 5.3; however, in the real implementation the spawn and join instructions (lines 7,11) are replaced by an equivalent but more efficient data structure. Threads are spawned at the beginning of the computation and are "recycled " at each invocation of the lookahead procedure to save execution time. The syntonization is handled by exploiting a task buffer filled by the candidate choices to be looked ahead. Note that, in the case that the two parallel techniques are combined, each model generator has its own task buffer and data structures for handling the parallel lookahead process.

6.2 Assessment of the Parallel Instantiator

In this section we report the results of the experimental analysis carried out for assessing the parallel instantiator. First of all we describe benchmarks problems and data in the next subsection. Then, a description of the results of the analysis is presented, which is divided into two trunks: first of all, a discussion on the effect of the third level of parallelism is presented, which prove the effectiveness of the technique. Then, scalability results of the system are reported, showing a sub-optimal behavior for the system.

6.2.1 Benchmark Problems and Data

In our experiments, several well-known combinatorial problems as well as real-world problems are considered. These benchmarks have already been used for assessing ASP instantiator performance [7, 55, 56]. Many of them are particularly difficult to parallelize due to the compactness of their encodings; note that concise encodings are quite common given the declarative nature of the ASP language which allows to compactly encode even very hard problems. About data, we considered five instances (where the instantiation time is non-negligible) of increasing difficulty for each problem, except for the Hamiltonian Path and 3-Colorability problem, for which generators are available, and we could generate several instances of increasing size.[1]

A detailed description of each benchmark is provided below. Some of the problems has been already described in Chapter 2; for these problems only a short description is provided. Moreover, the encoding of some of the benchmark problems is omitted due to the large number of rules (hundreds, thousands of rules).

n-Queens. The 8-queens puzzle is the problem of putting eight chess queens on an 8x8 chessboard so that none of them is able to capture any other using the standard chess queen's moves. The n-queens puzzle is the more general problem of placing n queens on an nxn chessboard ($n \geq 4$). The predicate

$$queens(X, Y)$$

represents the position of a queen; in particular if $queen(X, Y)$ is true then there is a queen at row X and column Y. The encoding of the problem follow the guess&check paradigm. The disjunctive rule

$$queen(X, 1) \ \vee \ \cdots \vee \ queen(X, n) \ {:-}\#int(X), \ X > 0.$$

allows the guess of the position of the queens; then, three constraints are used in order to verify that the assignment is a solution. In particular the constraint

$$:-q(X, Y), \ q(Z, Y), \ X <> Z.$$

verifies that no row contains two queens; the constraints

$$:-q(X, Y), \ q(A, B), \ N = X - A, \ B = Y + N, \ N > 0.$$
$$:-q(X, Y), \ q(A, B), \ N = X - A, \ Y = B + N, \ N > 0.$$

verifies that no queen can capture another attacking in diagonal. There is no need to verify that two queen are on the same column because the guess rule does not allow this possibility. Furthermore, note that the encoding avoid the explicit representation of the chessboard.

Instances were considered having $n \in \{37, 39, 41, 43, 45\}$.

Ramsey Numbers. The Ramsey number $ramsey(k, m)$ is the least integer n such that, no matter how the edges of the complete undirected graph (clique) with n nodes are colored using two colors, say red and blue, there is a red clique with k nodes (a red k-clique) or a blue clique with m nodes (a blue m-clique). The encoding of this problem has already been presented in Section 2.4, and consists of one rule and two constraints. For the experiments, the problem was considered of deciding whether, for $k = 7$, $m = 7$, and $n \in \{31, 32, 33, 34, 35\}$, n is $ramsey(k, m)$.

Clique. A clique in an undirected graph $G = (V, E)$ is a subset of its vertices such that every two vertices in the subset are connected by an edge. We

[1]Encodings and instances are available at `http://www.mat.unical.it/ricca/downloads/parallelground10.zip`.

considered the problem finding the cliques in a given input graph. The graph is made up starting from the predicate $edge(X, Y)$, then rules

$$\begin{array}{l} edge(X, Y) :- edge(Y, X). \\ node(X) :- edge(X, Y). \end{array}$$

are used to set up the graph representation. Then a disjunctive rule is used to guess a clique

$$inclique(X) \;\; \vee \;\; outclique(X) :- node(X).$$

where the $node(X)$ is in the clique if $inclique(X)$ is true. Minimality guaranties that only one between $inclique(X)$ or $outclique(X)$ is true for each answer set. To verify that all the nodes within a clique are connected a constraint is added

$$:- inclique(X),\; inclique(Y),\; notedge(X, Y),\; X! = Y.$$

Five graphs of increasing size were considered.

Timetabling. The problem of determining a timetable for some university lectures which have to be given in a week to some groups of students. The timetable must respect a number of given constraints concerning availability of rooms, teachers, and other issues related to the overall organization of the lectures. We do not provide a description of the encoding of this problem due to the large number or rules; however it is publicly available online together with all the other benchmark problem we considered in this thesis. Many instances were provided by the University of Calabria; they refer to different numbers of student groups $g \in \{15, 17, 19, 21, 23\}$.

Sudoku. Given an NxN grid board, where N is a square number $N = M^2$, fill it with integers from 1 to N so that each row, each column, and each of the N MxM boxes contains each of the integers from 1 to N exactly once. Suppose the rows are numbered 1 to N from left to right, and the columns are numbered 1 to N from top to bottom. The boxes are formed by dividing the rows from top to bottom every M rows and dividing the columns from left to right every M columns. Encoding and instances were used for testing the competitors solvers in the ASP Competition 2009 [56]. We do not provide a description of the encoding of this problem due to the large number or rules; however it is publicly available online together with all the other benchmark problem we considered in this thesis. For assessing our system we considered the instances $\{sudoku.in5$, $sudoku.in6$, $sudoku.in7$, $sudoku.in9$, $sudoku.in10\}$, where $N = 25$.

Golomb Ruler. A Golomb ruler is an assignment of marks to integer positions along a ruler so that no pair of two marks is the same distance from each other. The number of marks is the order of the ruler. The first mark is required to be at position 0, the position of the highest mark is the length of the ruler. The problem is finding the shortest ruler of a given order. Input instances are described by predicate $mark(M)$ and $position(P)$ representing marks and positions. A fact

$$non_free(0).$$

is added to force a mark to be at position 0. Then, a disjunctive rule is used to guess the position occupied by a mark

$$free(P) \ \vee non\_free(P) \text{ :- } position(P).$$

In order to verify that exactly N non-free position are guessed the following rule and constraint are encoded

$$\begin{array}{l} num(N) \text{ :- } \#\texttt{count}\{M : mark(M)\} \ = \ N. \\ \text{:- } num(N), \ \texttt{not} \ \#\texttt{count}\{P : non\_free(P)\} = N. \end{array}$$

To check that no pair of two marks is at the same distance from each other we compute, for each non-free position P1, the distance with each successive non-free position P2.

$$d(P1, D) \text{ :- } non_free(P1), \ non_free(P2), P1 < P2, D = P2 - P1.$$

then we discard models in which more than one pair of non-free position have the same distance.

$$\text{:- } d(P1, D), \ d(P2, D), \ P1 < P2.$$

As long as we are interested in finding the shorter ruler, an optimization part is added to the encoding. This is done by associating a cost to each non-free position P and each position left of P

$$\begin{array}{l} cost(P)\text{:- } non_free(P). \\ cost(P1)\text{:- } cost(P), P1 = P - 1. : \ cost(P).[P :] \end{array}$$

The last rule is called *weak* constraint, and is native DLV construct used for optimization issues; a description of this construct can be found in []

Encoding and instances have been used for testing the competitors solvers in the ASP Competition 2009 [56]. Instances are described by a couple (m, p) where m is the number marks and p is the number of positions: we considered the values $(10, 125)$, $(13, 150)$, $(14, 175)$,$(15, 200)$ and $(15, 225)$.

Reachability. Given a directed graph $G = (V, E)$, we want to compute all pairs of nodes $(a, b) \in V \times V$ (i) such that b is reachable from a through a nonempty sequence of edges in E. The encoding of this problem has already been presented in Section 2.4, and consists of one exit rule and a recursive one. Five trees were generated with a pair (number of levels, number of siblings): (9,3), (7,5), (14,2), (10,3) and (15,2), respectively.

Food. The problem here is to generate plans for repairing faulty workflows. That is, starting from a faulty workflow instance, the goal is to provide a completion of the workflow such that the output of the workflow is correct. Workflows may comprise many activities. Repair actions are compensation, (re)do and replacement of activities. We do not provide a description of the encoding of this problem due to the large number or rules; however it is publicly available online together with all the other benchmark problem we considered in this thesis. A single instance was provided related to a workflow containing 63 predicates, 56 components and 116 rules.

3-Colorability. This well-known problem asks for an assignment of three colors to the nodes of a graph, in such a way that adjacent nodes always have different colors. The input graph is represented by a set of predicates $edge(X, Y)$, and rules

$$node(X)\ derives\ edge(X, Y).$$
$$node(Y)\ derives\ edge(X, Y).$$

are used to single out the nodes of the graph. To guess a possible nodes coloration the following disjunctive rule is encoded

$$colored(r, X)\ \vee\ colored(g, X)\ \vee\ colored(b, X) \text{ :- } node(X).$$

To verify that all the nodes connected by an edge are assigned to different colours the following constraint is added

$$\text{:- } edge(X, Y),\ colored(C, X),\ colored(C, Y).$$

Concerning instances, a number of simplex graphs were generated with the Stanford GraphBase library [57], by using the function $simplex(n, n, -2, 0, 0, 0, 0)$, where $80 \leq n \leq 250$.

Hamiltonian Path. A classical NP-complete problem in graph theory, which can be expressed as follows: given a directed graph $G = (V, E)$ and a node $a \in V$ of this graph, does there exist a path in G starting at a and passing through each node in V exactly once. TThe encoding of this problem has already been presented in Section 2.4, and consists of several rules, one of these is recursive. Instances were generated, by using a tool by Patrik Simons (cf. [58]), with n nodes with $1000 \leq n \leq 12000$.

The machine used for the experiments is a two-processor Intel Xeon "Woodcrest" (quad core) 3GHz machine with 4MB of L2 Cache and 4GB of RAM, running Debian GNU Linux 4.0. Since our techniques focus on instantiation, all the results of the experimental analysis refer only to the instantiation process rather than the whole process of computing answer sets; in addition, the time spent before the grounding stage (parsing) is obviously the same both for parallel and non-parallel version. In order to obtain more trustworthy results, each single experiment was repeated five times and the average of performance measures are reported.

6.2.2 Experimental Results

Effect of Single Rule Parallelism In this section we report the results of an experimental analysis aimed at comparing the effects of the single rule parallelism with the first two levels. More in detail, we considered four versions of the instantiator: (*i*) `serial` where parallel techniques are not applied, (*ii*) $\texttt{levels}_{1+2}$ where components and rules parallelism are applied, (*iii*) $\texttt{level}_3$

Problem	`serial`	`levels`$_{1+2}$	`level`$_3$	`levels`$_{1+2+3}$
$queen_1$	4.64 (0.00)	2.19 (0.06)	0.71 (0.01)	0.69 (0.01)
$queen_2$	5.65 (0.00)	3.29 (0.51)	0.89 (0.01)	0.86 (0.02)
$queen_3$	6.83 (0.00)	3.85 (0.50)	1.08 (0.00)	1.03 (0.02)
$queen_4$	8.19 (0.00)	4.50 (0.48)	1.25 (0.01)	1.22 (0.00)
$queen_5$	9.96 (0.00)	4.72 (0.11)	1.45 (0.02)	1.43 (0.01)
$ramsey_1$	258.52 (0.00)	131.61 (0.23)	40.53 (0.21)	36.23 (0.34)
$ramsey_2$	328.68 (0.00)	168.03 (1.38)	51.35 (0.25)	46.09 (0.33)
$ramsey_3$	414.88 (0.00)	211.60 (0.55)	68.49 (0.10)	58.06 (0.20)
$ramsey_4$	518.28 (0.00)	265.58 (2.82)	83.32 (0.24)	75.19 (2.41)
$ramsey_5$	643.65 (0.00)	327.25 (0.79)	103.14 (0.44)	92.28 (0.65)
$clique_1$	16.06 (0.00)	15.88 (0.20)	3.34 (0.04)	2.35 (0.01)
$clique_2$	29.98 (0.00)	29.97 (0.01)	4.41 (0.12)	4.34 (0.07)
$clique_3$	49.11 (0.00)	49.18 (0.05)	7.11 (0.03)	7.09 (0.02)
$clique_4$	78.05 (0.00)	78.70 (0.35)	11.35 (0.14)	11.29 (0.11)
$clique_5$	119.48 (0.00)	118.66 (0.07)	17.08 (0.14)	17.09 (0.16)
$timetab_1$	15.48 (0.00)	7.28 (0.04)	2.35 (0.04)	2.29 (0.01)
$timetab_2$	17.49 (0.00)	8.30 (0.07)	2.61 (0.01)	2.61 (0.02)
$timetab_3$	21.65 (0.00)	10.22 (0.05)	3.22 (0.04)	3.20 (0.01)
$timetab_4$	17.75 (0.00)	8.24 (0.04)	2.61 (0.01)	2.64 (0.05)
$timetab_5$	23.69 (0.00)	11.09 (0.01)	3.52 (0.01)	3.50 (0.03)
$sudoku_1$	5.42 (0.00)	4.15 (0.04)	0.98 (0.01)	0.88 (0.01)
$sudoku_2$	9.87 (0.00)	7.75 (0.01)	1.59 (0.02)	1.51 (0.02)
$sudoku_3$	10.28 (0.00)	8.01 (0.05)	1.62 (0.00)	1.57 (0.01)
$sudoku_4$	10.56 (0.00)	8.38 (0.01)	1.75 (0.02)	1.63 (0.03)
$sudoku_5$	11.08 (0.00)	8.25 (0.04)	1.67 (0.02)	1.63 (0.05)
gol_ruler_1	6.58 (0.00)	6.32 (0.07)	0.96 (0.01)	0.94 (0.02)
gol_ruler_2	13.74 (0.00)	12.57 (0.04)	1.87 (0.04)	1.84 (0.09)
gol_ruler_3	24.13 (0.00)	22.67 (0.05)	3.29 (0.06)	3.25 (0.13)
gol_ruler_4	40.64 (0.00)	37.50 (0.10)	5.44 (0.21)	5.51 (0.10)
gol_ruler_5	62.23 (0.00)	59.04 (0.04)	8.32 (0.12)	8.36 (0.17)
$reach_1$	52.21 (0.00)	52.07 (0.07)	8.25 (0.06)	8.28 (0.01)
$reach_2$	147.34 (0.00)	148.34 (0.01)	22.60 (0.16)	22.67 (0.18)
$reach_3$	258.01 (0.00)	240.17 (0.13)	39.59 (0.29)	39.57 (0.44)
$reach_4$	522.09 (0.00)	517.97 (0.59)	77.21 (0.20)	77.52 (0.31)
$reach_5$	1072.00 (0.00)	1069.86 (1.04)	160.66 (0.21)	160.31 (0.25)
$Food$	684.95 (1.19)	0.18 (0.15)	104.6 (1.01)	0.08 (0.01)

Figure 6.2: Average instantiation times in seconds (standard deviation)

where only the single rule level is applied, and (*iv*) `levels`$_{1+2+3}$ in which all the three levels are applied. Results are shown in Figures 6.2-6.3-6.4; more in detail, Figure 6.3-6.4 report the average instantiation times for the Hamiltonian Path and the 3-Colorability problem, respectively; while the table in Figure 6.2 reports the average instantiation times in seconds for the remaining benchmarks.

More in detail, in Figure 6.2, the first column reports the problem considered, whereas the next columns report the results for the four instantiators. Looking at the third column in the table, benchmarks can be classified in three different groups according to their behavior: the benchmarks in which the first two levels of parallelism apply, those where these first two levels apply marginally, and

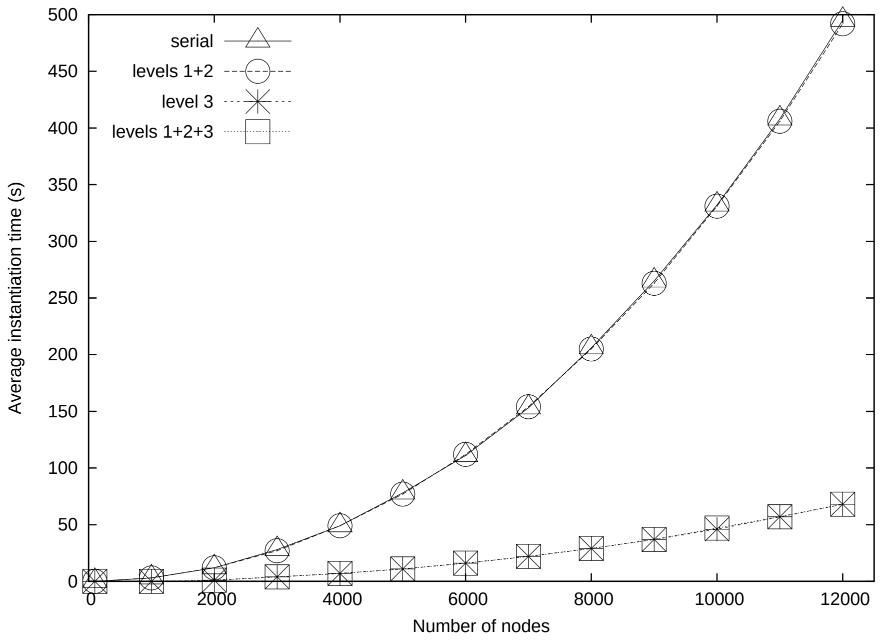

Figure 6.3: Instantiation times(s) - Hamiltonian Path

those where they do not apply at all. In the first group, we have the n-Queens problem, Ramsey Numbers, and Timetabling, where `levels`$_{1+2}$ is about twice faster than `serial`; however, considering that the machine on which we ran the benchmarks has eight core available, `levels`$_{1+2}$ is not able to exploit all the computational power at hand. The reason, is that the encodings of these benchmarks either have a small number of rules (n-Queens, Ramsey Numbers), or they show an intrinsic dependency among components/rules (Timetabling), that limits the efficacy of the first two levels of parallelism. These considerations explain also the behavior of the other two groups of benchmarks. More in detail, for the second group (which contains only Sudoku) a small improvement is obtained due to few rules which are evaluated in parallel, while the benchmarks belonging to the third group, whose encodings have very few interdependent rules (e.g. Reachability), proved hard to parallelize. Looking at the graphs, Hamiltonian Path and 3-Colorability clearly belongs to the third group, indeed the lines of `serial` and `levels`$_{1+2}$ overlap.

A special case is the Food problem, showing an impressive performance, which proved to be a case very easy to parallelize. This behavior can be explained by a different scheduling of the constraints performed by the serial version and the `levels`$_{1+2}$ one. In particular, this instance is inconsistent (there is a constraint always violated) and both versions stop the computation as soon as they recognize this fact. The scheduling performed by the parallel version allows the identification of this situation before the serial one, since constraints are evaluated in parallel, while the latter evaluates the inconsistent constraint

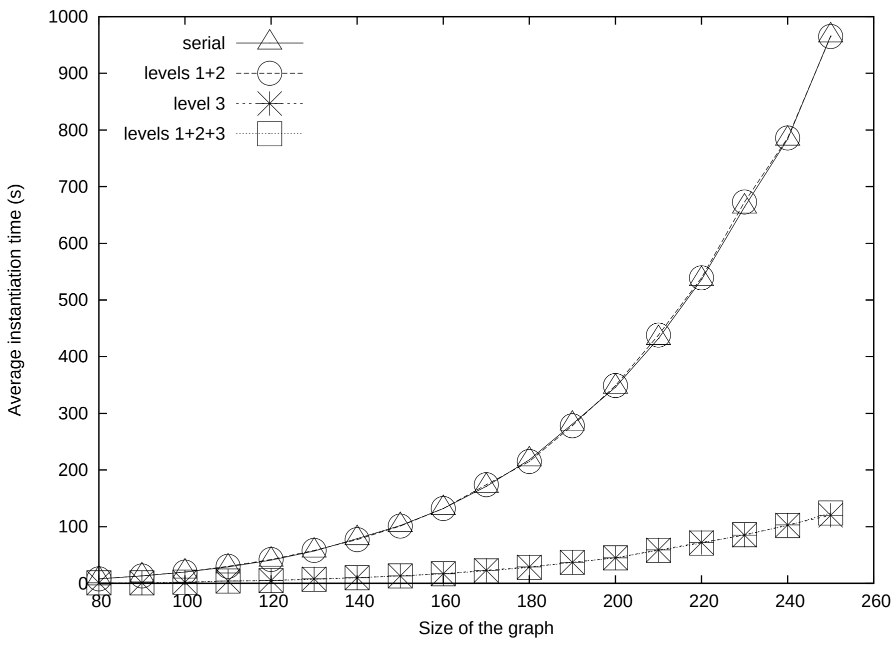

Figure 6.4: Instantiation times(s) - 3-Colorability

later on.

Concerning the behavior of $\mathtt{level}_3$, we notice that it always performs very well (always more than 7.5x faster than `serial`), and in all cases but Food, it outperforms $\mathtt{levels}_{1+2}$. Basically, the third level of parallelism applies to every single rule, and thus it is effective on all problems, even those with very small encodings. In the case of Food, even if $\mathtt{level}_3$ is about 7x faster than `serial`, it evaluates rules in the same order than `serial` thus recognizing the inconsistent constraint later than $\mathtt{levels}_{1+2}$.

The good news is that the three levels of parallelism always combine (even in the case of Food). This can be easily seen by looking at the last column of table and at the two graphs. Note that most of the advantages are due to the third level of parallelism. Indeed in the graphs, the lines for $\mathtt{level}_3$ and $\mathtt{levels}_{1+2+3}$ overlap, and $\mathtt{levels}_{1+2+3}$ shows only marginal gains w.r.t. $\mathtt{level}_3$, in the benchmarks where $\mathtt{levels}_{1+2}$ applies.

Scalability of the Approach We conducted a scalability analysis on the instantiator $\mathtt{levels}_{1+2+3}$ which exploits all the three parallelism levels. Moreover, we considered the effects of increasing both the size of the instances and the number of available processors (from 1 up to 8 CPUs).[2] The results of the analysis are summarized in Table 6.1, Figure 6.6, and Figure 6.8, which report

[2]Available processors can be disabled (respectively enabled) by running the bash Linux commands: *echo 0 >> /sys/devices/system/cpu/cpu-n/online* (resp. *echo 1 >> /sys/devices/system/cpu/cpu-n/online*).

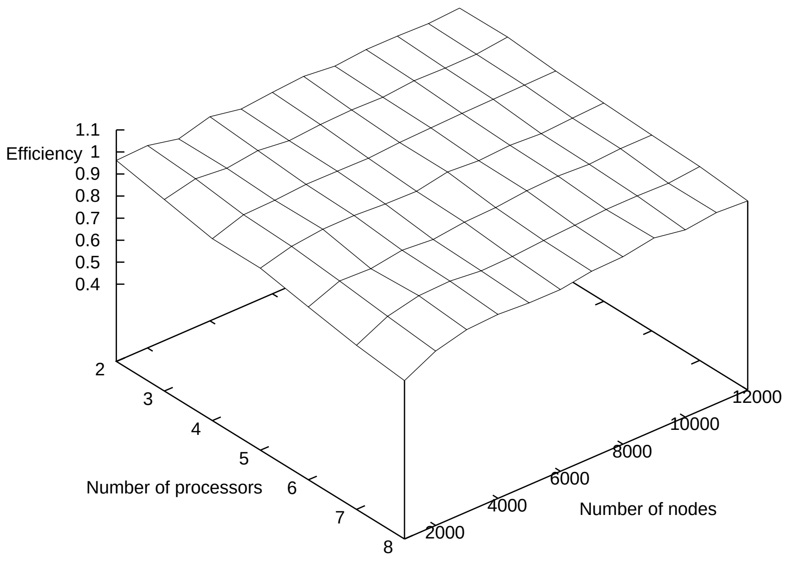

Figure 6.5: Efficiency - Hamiltonian Path

the average instantiation time; in Table 6.2, Figure 6.5, and Figure 6.7, which report the efficiencies. As before, the graphs show results for Hamiltonian Path and 3-Colorability, while the results of the remaining problems are reported in the table. The overall picture is very positive: the performance of the instantiator is very good in all cases and average efficiencies vary between 0.85 and 0.95 when all the available CPUs are enabled. As one would expect, the efficiency of the system slightly decreases when the number of processors increases –still remaining at a good level– and rapidly increases going from very small instances (<2 seconds) to larger ones.

The granularity control mechanism resulted to be effective in the n-Queens problem, where all the considered instances required less than 10 seconds of serial execution time. Indeed, the "very easy" disjunctive rule was always sequentially-evaluated in all the cases. Since the remaining constraints are strictly determined by the result of the evaluation of the disjunctive rule, the unavoidable presence of a sequential part limited the final efficiency to a remarkable 0.9 in the case of 8 processors. A similar scenario can be observed in the case of Ramsey Numbers, where the positive impact of the load balancing and granularity control heuristics becomes very evident. In fact, since the encoding is composed of few "very easy" disjunctive rules and two "very hard" constraints, the heuristics selects a sequential evaluation for the rules, and dynamically applies the finer distribution of the last splits for the constraints. As a result, the system produces a well-balanced work subdivision, which allows steady results to be obtained with an average efficiency greater or almost equal to 0.9 in all tested configurations. Analogously for Clique, which has a short

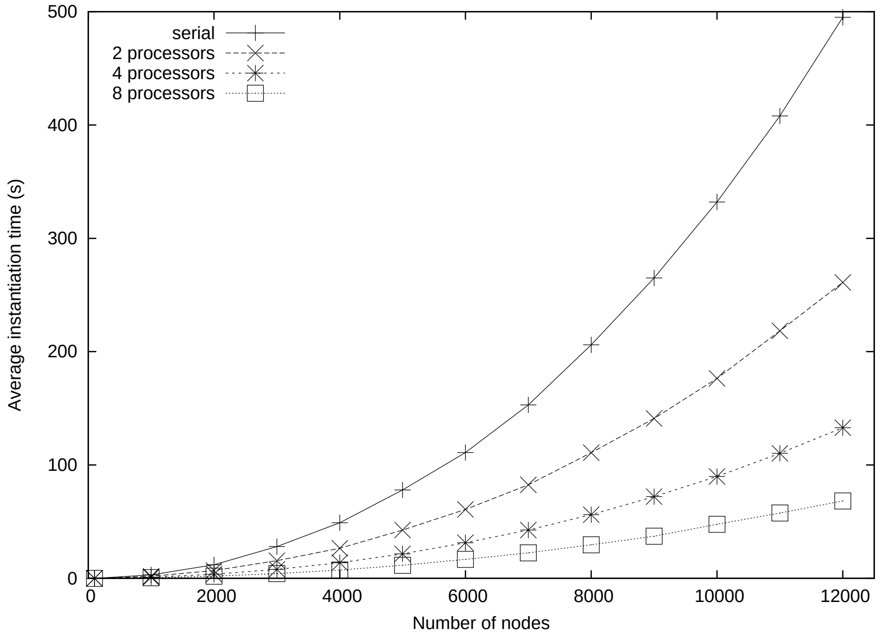

Figure 6.6: Instantiation times - Hamiltonian Path

encoding consisting of only three easy rules, for which granularity control schedules a serial execution, and one "hard" constraint which can be split and thus evaluated in parallel.

A good performance is also obtained in the case of Reachability. This problem is made up of only two rules; the first one is caught by granularity control which schedules its serial execution. The second one is a heavy recursive rule, that requires several iterations to be grounded. In this case a good load balancing is obtained thanks to the redistributions applied (with possibly different split sizes) at each iteration of the semi naïve algorithm.

The instantiator is effective also in Golomb Ruler, Timetabling and Sudoku where the performance results to be good also thanks to a well-balanced workload distribution.

About Food, a super-linear speedup (owing to the first levels of parallelism) is already evident with two-processors and efficiency peaks when three processors are enabled, where the execution times becomes negligible. The behaviour of the system for instances of varying sizes was analysed in more detail in the case of Hamiltonian Path and 3-Colorability; this was made possible by the availability of generators.

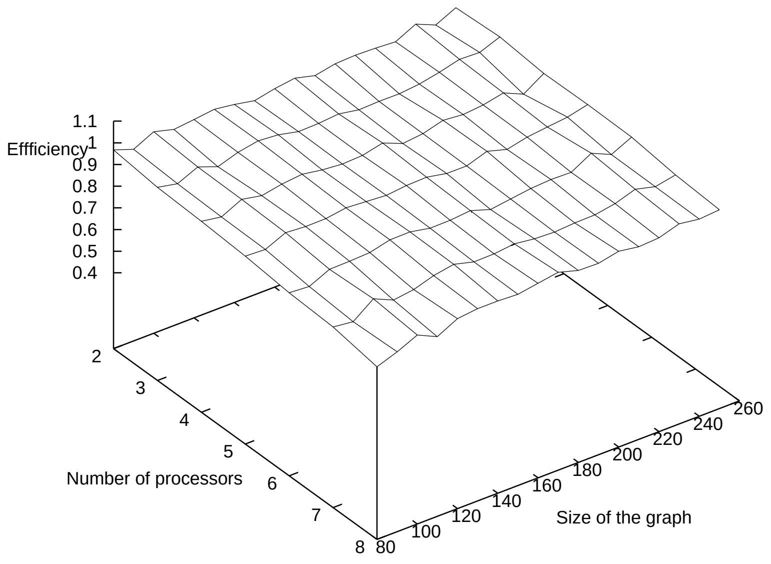

Figure 6.7: Efficiency - 3-Colorability

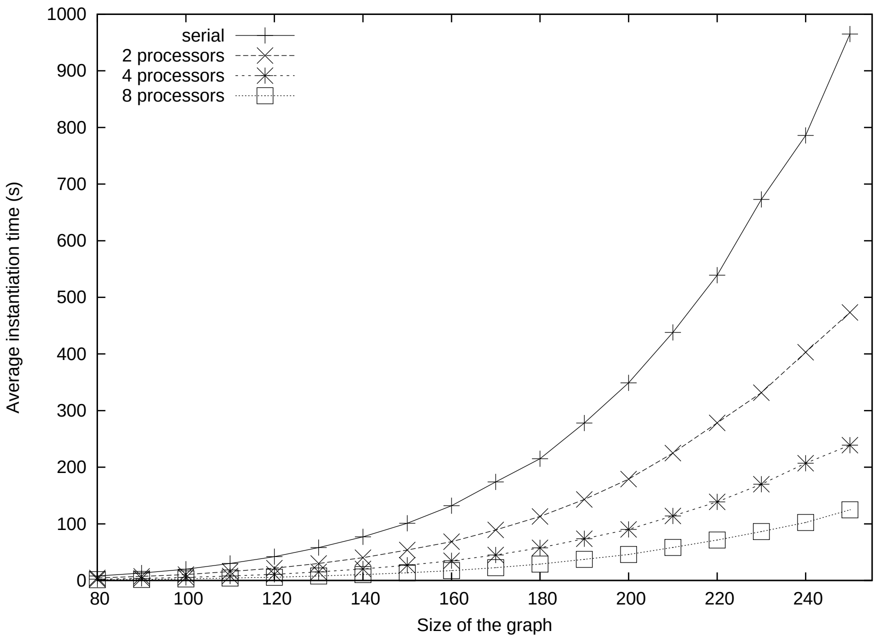

Figure 6.8: Instantiation times - 3-Colorability

	Average instantiation time (standard deviation)							
Problem	serial	2 proc	3 proc	4 proc	5 proc	6 proc	7 proc	8 proc
$queen_1$	4.64 (0.00)	2.53 (0.01)	1.71 (0.01)	1.31 (0.01)	1.07 (0.00)	0.91 (0.01)	0.78 (0.01)	0.69 (0.01)
$queen_2$	5.65 (0.00)	3.11 (0.00)	2.11 (0.01)	1.60 (0.00)	1.31 (0.01)	1.11 (0.01)	0.97 (0.00)	0.86 (0.02)
$queen_3$	6.83 (0.00)	3.79 (0.01)	2.57 (0.01)	1.97 (0.01)	1.60 (0.01)	1.35 (0.02)	1.17 (0.01)	1.03 (0.02)
$queen_4$	8.19 (0.00)	4.54 (0.00)	3.06 (0.01)	2.35 (0.01)	1.90 (0.01)	1.62 (0.02)	1.41 (0.01)	1.22 (0.00)
$queen_5$	9.96 (0.00)	5.57 (0.19)	3.68 (0.01)	2.81 (0.02)	2.26 (0.02)	1.92 (0.00)	1.69 (0.01)	1.43 (0.01)
$ramsey_1$	258.52 (0.00)	131.72 (0.08)	89.04 (0.41)	67.10 (0.46)	55.14 (0.93)	46.62 (0.19)	39.98 (0.12)	36.23 (0.34)
$ramsey_2$	328.68 (0.00)	167.47 (0.16)	112.97 (0.94)	85.90 (0.15)	70.64 (1.74)	58.70 (0.82)	51.21 (0.18)	46.09 (0.33)
$ramsey_3$	414.88 (0.00)	210.98 (0.38)	142.85 (0.68)	108.00 (0.38)	88.13 (0.51)	74.83 (0.22)	65.25 (0.59)	58.06 (0.20)
$ramsey_4$	518.28 (0.00)	264.69 (1.82)	178.67 (2.39)	137.42 (1.89)	111.09 (2.15)	95.27 (2.02)	81.45 (0.45)	75.19 (2.41)
$ramsey_5$	643.65 (0.00)	327.06 (0.36)	222.81 (0.20)	169.37 (0.86)	135.94 (0.17)	115.78 (0.92)	101.21 (1.33)	92.28 (0.65)
$clique_1$	16.06 (0.0)	8.51 (0.13)	5.84 (0.08)	4.45 (0.17)	3.64 (0.04)	3.08(0.1)	2.67 (0.03)	2.35 (0.01)
$clique_2$	29.98 (0.0)	15.92 (0.23)	10.69 (0.18)	8.27 (0.09)	6.77 (0.11)	5.71 (0.10)	4.94 (0.40)	4.34 (0.07)
$clique_3$	49.11 (0.00)	25.81 (0.41)	17.31 (0.06)	13.39 (0.20)	10.92 (0.20)	9.23 (0.02)	7.98 (0.03)	7.09 (0.02)
$clique_4$	78.05 (0.00)	41.68 (0.07)	27.91 (0.28)	21.10 (0.02)	17.33 (0.20)	14.60 (0.04)	12.76 (0.06)	11.29 (0.11)
$clique_5$	119.48 (0.00)	62.87 (0.13)	42.62 (0.15)	32.46 (0.04)	26.14 (0.21)	22.24 (0.00)	19.14 (0.00)	17.09 (0.16)
$timetab_1$	15.48 (0.00)	7.98 (0.10)	5.41 (0.02)	4.16 (0.00)	3.37 (0.00)	2.93 (0.05)	2.59 (0.03)	2.29 (0.01)
$timetab_2$	17.49 (0.00)	9.26 (0.34)	6.30 (0.23)	4.68 (0.01)	3.89 (0.04)	3.41 (0.14)	2.92 (0.04)	2.61 (0.02)
$timetab_3$	21.65 (0.00)	11.12 (0.03)	7.54 (0.02)	5.98 (0.23)	4.84 (0.09)	4.08 (0.05)	3.64 (0.02)	3.20 (0.01)
$timetab_4$	17.75 (0.00)	9.32 (0.36)	6.13 (0.02)	4.75 (0.06)	3.86 (0.02)	3.33 (0.01)	3.01 (0.15)	2.64 (0.05)
$timetab_5$	23.69 (0.00)	12.16 (0.01)	8.28 (0.01)	6.35 (0.02)	5.34 (0.20)	4.47 (0.03)	3.94 (0.03)	3.50 (0.03)
$sudoku_1$	5.42 (0.00)	2.84 (0.01)	2.14 (0.21)	1.54 (0.00)	1.29 (0.02)	1.10 (0.00)	0.98 (0.02)	0.88 (0.01)
$sudoku_2$	9.87 (0.00)	5.09 (0.02)	3.53 (0.02)	2.72 (0.04)	2.25 (0.01)	1.90 (0.02)	1.68 (0.03)	1.51 (0.02)
$sudoku_3$	10.28 (0.00)	5.45 (0.17)	3.63 (0.01)	2.81 (0.02)	2.31 (0.01)	1.96 (0.01)	1.78 (0.00)	1.57 (0.01)
$sudoku_4$	10.56 (0.00)	5.50 (0.03)	3.80 (0.03)	2.92 (0.03)	2.41 (0.02)	2.03 (0.02)	1.81 (0.04)	1.63 (0.03)
$sudoku_5$	11.08 (0.00)	5.52 (0.12)	3.73 (0.01)	2.93 (0.11)	2.35 (0.01)	2.05 (0.04)	1.82 (0.03)	1.63 (0.05)
gol_ruler_1	6.58 (0.00)	3.34 (0.01)	2.26 (0.00)	1.73 (0.02)	1.42 (0.02)	1.24 (0.02)	1.06 (0.03)	0.94 (0.02)
gol_ruler_2	13.74 (0.00)	6.63 (0.02)	4.60 (0.18)	3.41 (0.04)	2.86 (0.10)	2.43 (0.04)	2.11 (0.02)	1.84 (0.09)
gol_ruler_3	24.13 (0.00)	12.11 (0.02)	8.15 (0.06)	6.34 (0.06)	5.06 (0.10)	4.34 (0.17)	3.79 (0.05)	3.25 (0.13)
gol_ruler_4	40.64 (0.00)	20.27 (0.05)	13.51 (0.11)	10.35 (0.10)	8.64 (0.19)	7.13 (0.25)	6.35 (0.31)	5.51 (0.10)
gol_ruler_4	62.23 (0.00)	31.54 (0.29)	21.30 (0.16)	16.03 (0.09)	12.95 (0.20)	11.03 (0.27)	9.67 (0.15)	8.36 (0.17)
$reach_1$	52.21 (0.00)	29.52 (0.36)	20.41 (0.25)	15.38 (0.02)	12.73 (0.16)	10.81 (0.03)	9.63 (0.05)	8.28 (0.01)
$reach_2$	147.34 (0.00)	84.93 (0.35)	57.14 (0.07)	43.56 (0.07)	35.16 (0.19)	29.90 (0.02)	26.02 (0.10)	22.67 (0.18)
$reach_3$	258.01 (0.00)	144.36 (0.60)	97.06 (0.39)	74.88 (0.25)	61.05 (0.21)	52.38 (0.09)	45.64 (0.21)	39.57 (0.44)
$reach_4$	522.09 (0.00)	301.44 (0.48)	201.90 (0.47)	153.00 (0.28)	123.83 (0.16)	104.74 (0.54)	90.44 (0.10)	77.52 (0.31))
$reach_5$	1072.00 (0.00)	618.68 (0.71)	412.29 (0.05)	311.96 (0.98)	253.22 (0.98)	213.31 (0.85)	185.08 (0.26)	160.31 (0.25)
$Food$	684.95 (1.19)	0.22 (0.15)	0.08 (0.01)	0.07 (0.01)	0.06 (0.01)	0.06 (0.00)	0.06 (0.00)	0.08 (0.01)

Table 6.1: Benchmark Results: average instantiation times in seconds (standard deviation)

	Efficiency						
Problem	2 proc	3 proc	4 proc	5 proc	6 proc	7 proc	8 proc
$queen_1$	0.98	0.97	0.95	0.93	0.91	0.91	0.90
$queen_2$	0.99	0.97	0.96	0.94	0.92	0.91	0.90
$queen_3$	0.99	0.97	0.95	0.94	0.92	0.91	0.91
$queen_4$	0.99	0.98	0.96	0.95	0.92	0.91	0.92
$queen_5$	0.97	0.97	0.96	0.95	0.93	0.94	0.94
$ramsey_1$	0.98	0.97	0.96	0.94	0.92	0.92	0.89
$ramsey_2$	0.98	0.97	0.96	0.93	0.93	0.92	0.89
$ramsey_3$	0.98	0.97	0.96	0.94	0.92	0.91	0.89
$ramsey_4$	0.98	0.97	0.94	0.93	0.91	0.91	0.86
$ramsey_5$	0.98	0.96	0.95	0.95	0.93	0.91	0.87
$clique_1$	0.94	0.92	0.90	0.88	0.87	0.86	0.85
$clique_2$	0.94	0.93	0.91	0.89	0.88	0.87	0.86
$clique_3$	0.95	0.95	0.92	0.90	0.89	0.88	0.87
$clique_4$	0.94	0.93	0.92	0.90	0.89	0.87	0.86
$clique_5$	0.95	0.93	0.92	0.91	0.90	0.89	0.87
$timetab_1$	0.97	0.95	0.93	0.92	0.88	0.85	0.84
$timetab_2$	0.94	0.93	0.93	0.90	0.85	0.86	0.84
$timetab_3$	0.97	0.96	0.91	0.89	0.88	0.85	0.85
$timetab_4$	0.95	0.97	0.93	0.92	0.89	0.84	0.84
$timetab_5$	0.97	0.95	0.93	0.89	0.88	0.86	0.85
$sudoku_1$	0.95	0.84	0.88	0.84	0.82	0.79	0.77
$sudoku_2$	0.94	0.94	0.91	0.89	0.87	0.83	0.82
$sudoku_3$	0.97	0.93	0.91	0.88	0.87	0.84	0.82
$sudoku_4$	0.96	0.93	0.90	0.88	0.87	0.83	0.81
$sudoku_5$	1.00	0.99	0.95	0.94	0.90	0.87	0.85
gol_ruler_1	0.99	0.97	0.95	0.93	0.88	0.89	0.88
gol_ruler_2	1.04	1.00	1.01	0.96	0.94	0.93	0.93
gol_ruler_3	1.00	0.99	0.95	0.95	0.93	0.91	0.93
gol_ruler_4	1.00	1.00	0.98	0.94	0.95	0.91	0.92
gol_ruler_4	0.99	0.97	0.97	0.96	0.94	0.92	0.93
$reach_1$	0.86	0.85	0.85	0.82	0.80	0.77	0.79
$reach_2$	0.87	0.86	0.85	0.84	0.82	0.81	0.81
$reach_3$	0.87	0.87	0.86	0.85	0.82	0.81	0.82
$reach_4$	0.87	0.86	0.85	0.84	0.83	0.82	0.84
$reach_5$	0.87	0.87	0.86	0.85	0.84	0.83	0.84
$Food$	1556	2853	2446	2283	1902	1630	1223

Table 6.2: Benchmark Results: efficiency

Looking at Figures 6.5 and 6.7, it is evident that the efficiency of the system rapidly reaches a good level (ranging from 0.9 up to 1), moving from small instances (requiring less than 2s) to larger ones, and remains stable (the surfaces are basically plateaux). The corresponding gains are visible by looking at Figures 6.6 and 6.8, where, e.g. an Hamiltonian Path (3-Colorability) instance is evaluated in 332.78s (965.36s) by the serial system, and requires only 68.26s (124.70s) with $\texttt{levels}_{1+2+3}$ with 8-processor enabled.

Summarizing, the parallel instantiator behaved very well in all the considered instances. It showed superlinear speedups in the case of easy-to-parallelize instances and, in the other cases its efficiency rapidly reaches good levels and remains stable when the sizes of the input problem grow. Importantly, the system offers a very good performance already when only two CPUs are enabled (i.e. for the largest majority of the commercially-available hardware at the time of this writing), and efficiency remains at a very good level when up to 8 CPUs are available.

6.3 Assessment of the Parallel Propositional Search Strategies

In this section we provide a description of the results of the experimental analysis carried out for assessing the parallel model generator. Benchmarks problems and data are described first. Then, a picture of the results of the analysis is presented, which is divided into two trunks: description of the multi-heuristic system performances is provided first; then, we discuss the effect of the parallel lookahead technique. Experimental results show that both the parallel methods can be used for exploiting parallelism for improving the computation of the answer sets.

6.3.1 Benchmark Problems and Instances

We considered three well-known problems which are usually exploited for evaluating Model Generator performances.

3SAT. is a special case of SAT, one of the best researched problems in AI and frequently used for solving many other problems by translating them to SAT, solving the SAT problem, and transforming the solution back to the original domain.

> Let Φ be a propositional formula in conjunctive normal form (CNF) $\Phi = \bigwedge_{i=1}^{n}(d_{i,1} \vee d_{i,2} \vee d_{i,3})$ where the $d_{i,\cdot}$ are literals over propositional variables $x_1, \ldots, x_m$.
>
> Φ is satisfiable, iff there exists a consistent conjunction I of literals such that $I \models \Phi$.

3SAT is a classical NP-complete problem [40] and can be easily represented in ASP as follows:

For every propositional variable x_i $(1 \leq i \leq m)$, we add the following rule which ensures that we either assume that variable x_i or its complement nx_i is true: $x_i \vee nx_i$. And for every clause $d_1 \vee d_2 \vee d_3$ in Φ we add the constraint $:- \bar{d_1}, \bar{d_2}, \bar{d_3}$. where $\bar{d_k}$ $(1 \leq k \leq 3)$ is nx_j if $d_k = x_j$, and $\bar{d_k} = x_j$ if $d_k = \neg x_j$.

The instances for 3SAT were randomly generated by using a tool by Selman and Kautz [59]. For each size we generated 20 such instances, where we kept the ratio between the number of clauses and the number of variables at 4.3, which is near the cross-over point for random 3SAT [60].

HAMPATH. A classical NP-complete problem in graph theory, which can be expressed as follows: given a directed graph $G = (V, E)$ and a node $a \in V$ of this graph, does there exist a path in G starting at a and passing through each node in V exactly once. The encoding of this problem has already been presented in Section 2.4. The instances for HAMPATH were generated by a tool by Patrik Simons (cf. [58]). For each problem size n we generated 20 instances, always assuming node 1 as the starting node.

STRATCOMP. Is the problem of computing companies that are "strategic" according with the following definition [41]. A holding owns companies $C(1), \ldots, C(c)$, each of which produces some goods. Some of these companies may jointly control another one. Now, some companies should be sold, under the constraint that all goods can still be produced, and that no company is sold which would still be controlled by the holding afterwards. A company is strategic, if it belongs to a *strategic set*, which is a minimal set of companies satisfying these constraints. The encoding of this problem has already been presented in Section 2.4. For STRATCOMP, we randomly generated 20 instances for each problem size n, with n companies and n products.

The first two benchmarks (3SAT and HAMPATH) are well-known NP-complete problems, while the third (STRATCOMP) is a Σ_2^P-complete problem. All the considered benchmarks problems have been frequently used to assess the efficiency of ASP systems (see, e.g., [7, 61]). Since disjunctive ASP can represent every problem in the second level of the polynomial hierarchy we also considered the strategic companies problem.

The machine used for the experiments is a multi-processor Intel Xeon "Woodcrest" (quad core) 3GHz machine with 4MB of L2 Cache and 4GB of RAM, running Debian GNU Linux 4.0.

We have allowed at most 600 seconds (ten minutes) of execution time for each instance. , and we set a memory usage limit to 1GB of memory for each process (by exploiting the *ulimit* command). The experimentation has been stopped (for each system) at the size at which some instance exceeded this time limit. [3].

6.3.2 Experimental Results

The results of our experiments are displayed in the graphs of the Figures from 6.9 to 6.14. For each problem domain we report two graphs, describing the behavior of the two tested parallel techniques: In both graphs the horizontal axis shows a parameter representing the size of the instance, while on the vertical axis we report the running time (expressed in seconds) averaged over the instances of the same size we ran.

Since our techniques focus on model generation, all the results of the experimental analysis refer only to the process of computing answer sets of ground programs.

Evaluation of Parallel Lookahead. In order to assess the impact of the parallel lookahead we considered a number of variants of the same prototype:

[3]Actually, the memory usage limit had never been exceeded.

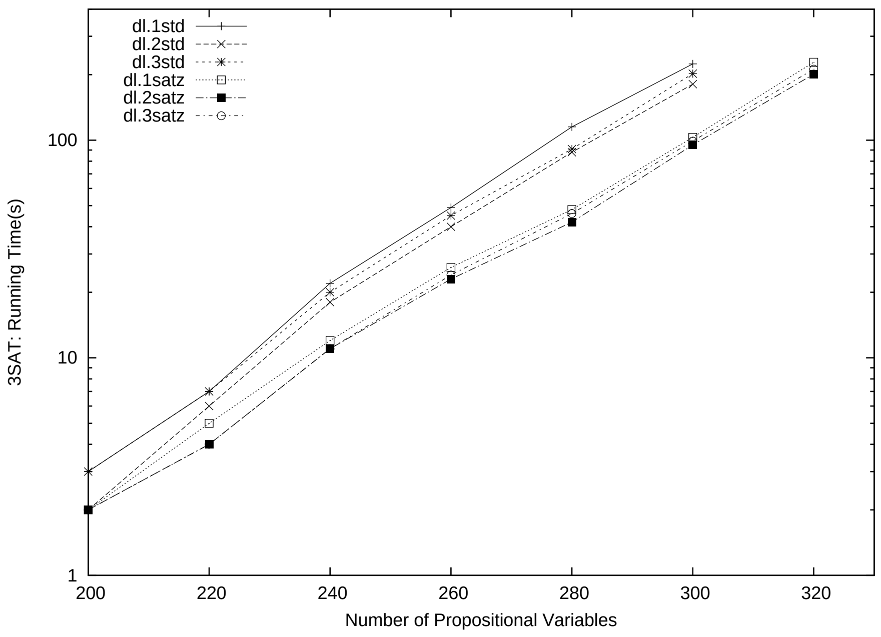

Figure 6.9: Parallel Lookhaead - 3-SAT

- **dl.X.satz** DLV with lookahead exploiting $\mathbf{h_{satz}}$ and $X = [1, 2, 3]$ threads.
- **dl.X.std** DLV with lookahead exploiting $\mathbf{h_{std}}$ and $X = [1, 2, 3]$ threads.

The results shown in Figure 6.9, in Figure 6.11 and in Figure 6.13 report the performance of those variants on the considered instances of 3SAT, HAMPATH and STRATCOMP, respectively.

Concerning 3SAT, the best heuristics is clearly $\mathbf{h_{satz}}$, the entire group of dl.X.satz performed better than dl.X.std group, solving all the generated instances in less time; whereas all the dl.X.std were stopped when considering instances having more than 300 variables. Moreover, it can be noted that dl.2.satz is the absolute best variant in this domain, and dl.2.std is the best among the ones exploiting $\mathbf{h_{std}}$. The lower performance of variants with three workers[4] is due to the larger amount of time spent by threads in synchronization (perhaps a technological problem of our prototype that can be probably overcome by improving the implementation).

As far as HAMPATH is concerned, the results are clearly in favor of the group exploiting the standard heuristics. Here the standard heuristics, which takes into account peculiar properties of ASP programs, has an edge on the sat-based one. Indeed, all the dl.X.satz variants were stopped before reaching 100 nodes, while the best versions equipped with standard heuristics could solve instances having up to 120 nodes. Unfortunately here the effect of parallel

[4]We tried also variants with more than three workers confirming this statement; results have been omitted for the sake of readability.

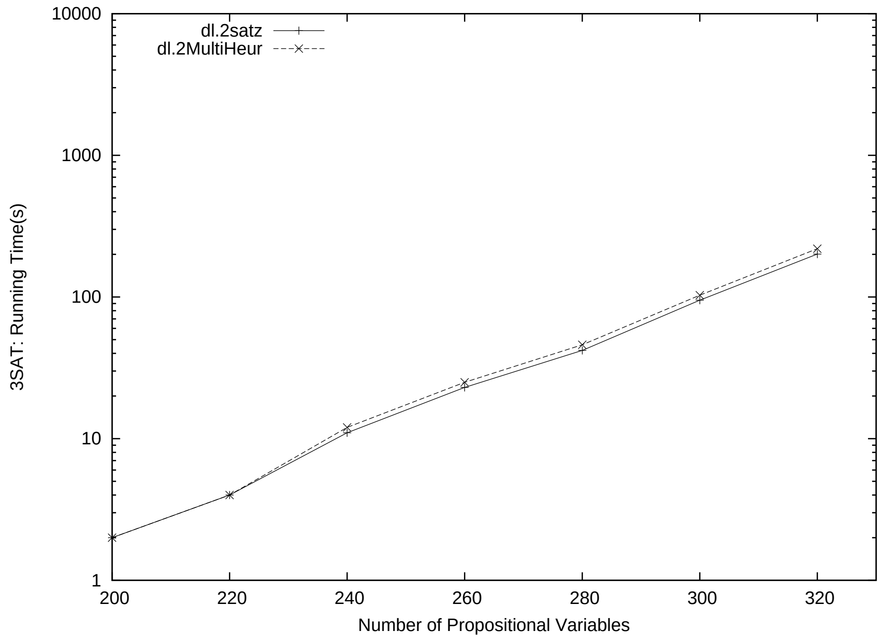

Figure 6.10: Parallel Multi-Heuristics - 3-SAT

lookahead is overshadowed by an internal optimization of DLV, which considers as selectable literals only a subset the available ones called PT literals (c.f.r. [29]), thus reducing the work to be divided among workers; this results in an emphasized effect of synchronization overhead which still remains acceptable (<0.9sec).

For STRATCOMP it is evident that the standard heuristics is the best, and that the effect of parallel lookahead made dl.2.std and dl.3.std to be the best variants (still dl.2 shows less overhead compared with dl.3), which were capable to solve instances having up to 290 companies; whereas, the $\mathbf{h_{satz}}$-based systems were stopped before reaching 150 companies, and dl.1.std reaches at most 170 companies.

Summarizing, results clearly show that the benefits of parallel lookahead are maximized when at most two workers are employed, and the technique can bring interesting speedups on hard instances.

Evaluation of the Multi-Heuristics Approach. The results reported above, as one can expect, show that there is no heuristics performing well in all cases (e.g. $\mathbf{h_{satz}}$ is the best for 3SAT, while in the other two domains $\mathbf{h_{std}}$ is the winner). In order to evaluate the second strategy proposed in this thesis, we considered a variant implementing the multi-heuristics approach and exploiting the best parallel lookahead settings (named dl2.MultiHeu). We compared dl2.MultiHeu with the best performer in each domain among the ones considered before; the result are shown in Figure 6.10, in Figure 6.12 and in Fig-

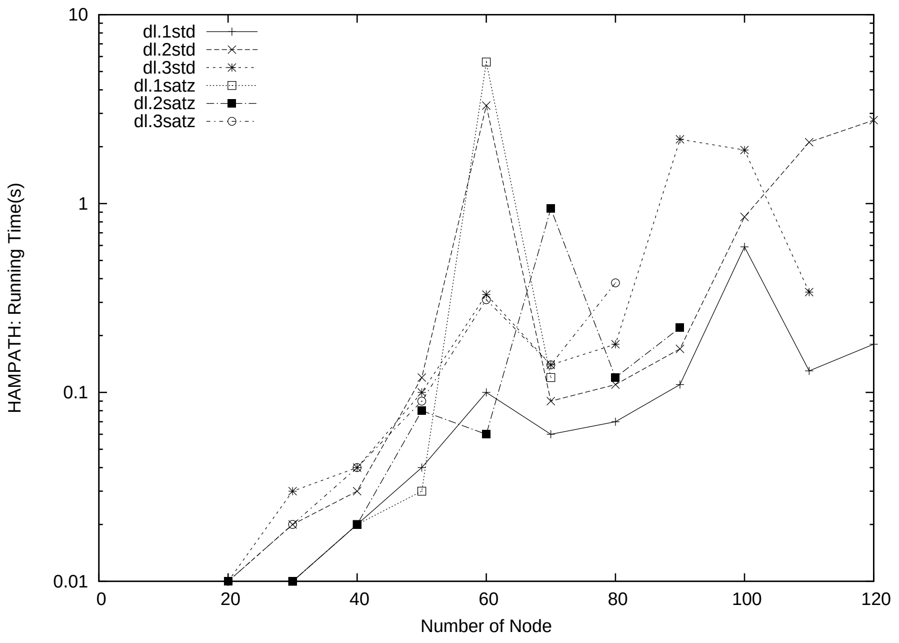

Figure 6.11: Parallel Lookhaead - Hamiltonian Path

ure 6.14, which reports the performance of those variants on 3SAT, HAMPATH and STRATCOMP, respectively.

The picture here is very clear, dl2.MultiHeu is able to move forward the limit of the the largest instance solved in 10 minutes. Still the synchronization overhead, paid to stop the concurrent model generators remains evident, but the dramatic advantage of selecting the best possible criterion per instance allows for solving larger instances in less time. For instance, dl2.MultiHeu solved instances with up to 160 nodes in HAMPATH where the competitor stopped at 120 nodes; and dl2.MultiHeu solved up to 3000 companies vs 2900. The overall performance leap becomes dramatic when dl2.MultiHeu is compared with the original dl1.std: 300 vs 320 propositional variables for 3SAT; 120 vs 160 nodes for HAMPATH; and 1700 vs 3000 companies for STRATCOMP.

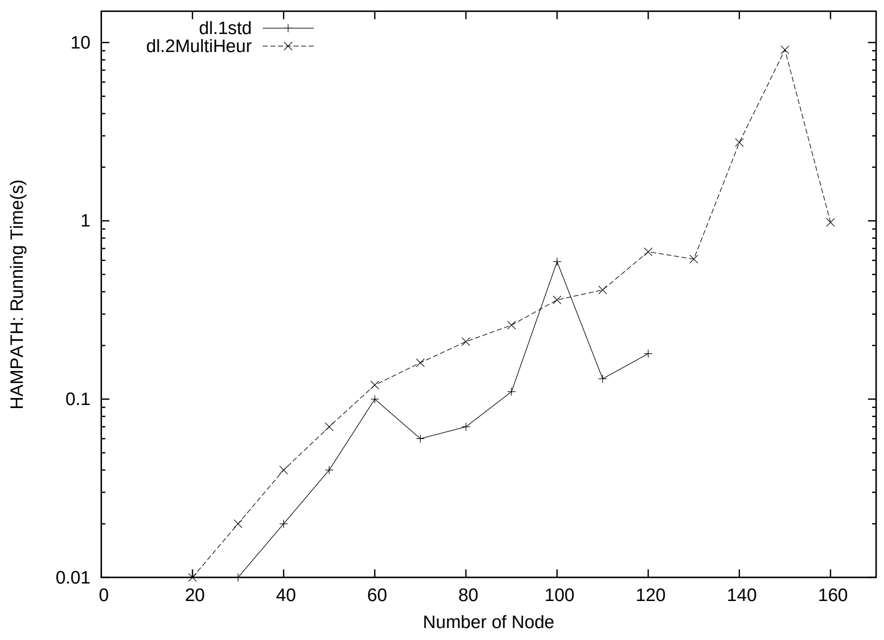

Figure 6.12: Parallel Multi-heuristics - Hamiltonian Path

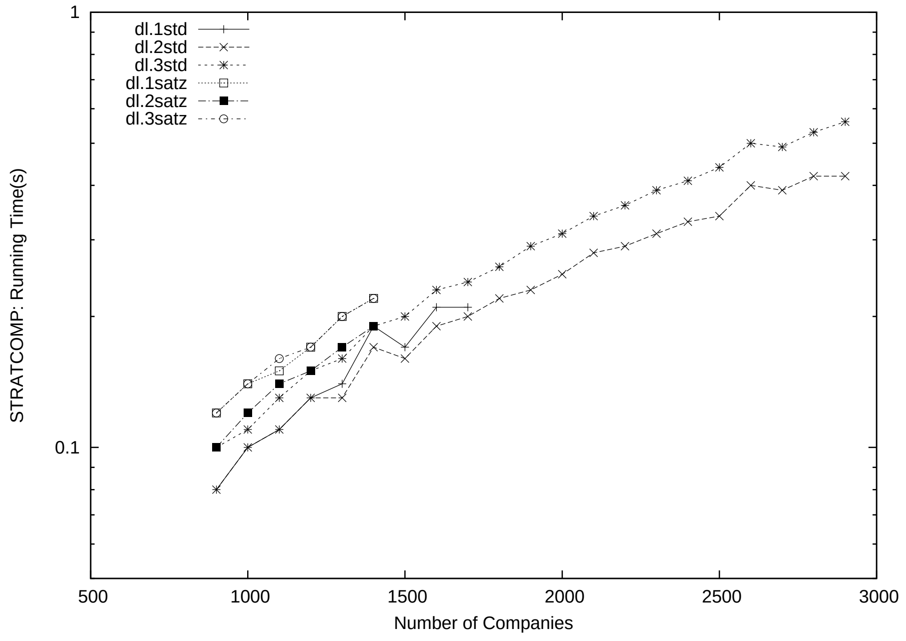

Figure 6.13: Parallel Lookhaead - Strategic Companies

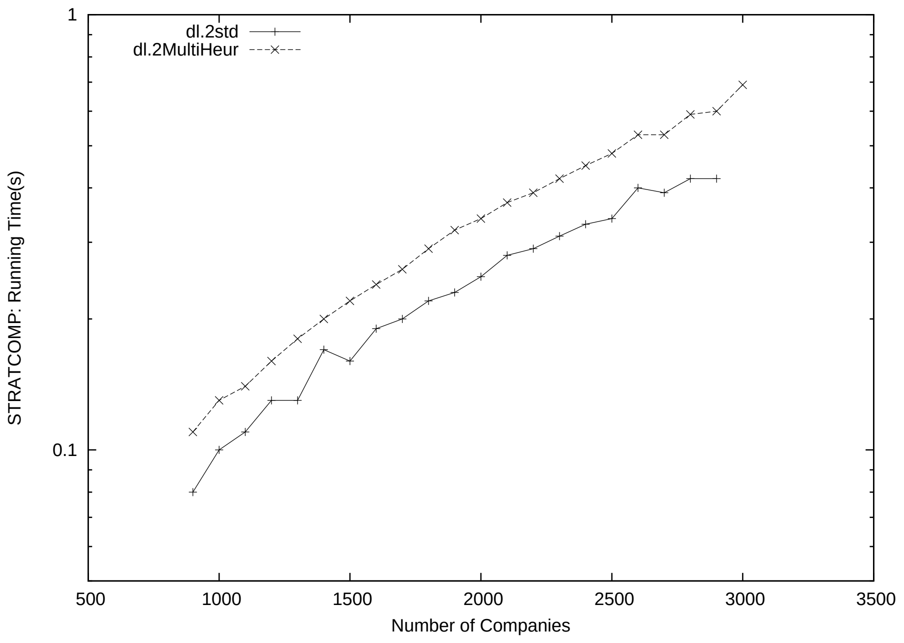

Figure 6.14: Parallel Multi-Heuristics - Strategic Companies

Chapter 7

Related Works

Several works about parallel techniques for the evaluation of ASP programs have been proposed, focusing on both the propositional (model search) phase [62, 63, 64, 65], and the instantiation phase [66, 51]. Similarities and differences with the work presented in this thesis on parallel instantiation and parallel propositional search are discussed next.

Concerning the parallelization of the instantiation phase, some preliminary studies were carried out in [66], as one of the aspects of the attempt to introduce parallelism in non-monotonic reasoning systems. However, there are crucial differences with our system regarding both the employed technology and the supported parallelization strategy. Indeed, our system is implemented by using POSIX threads APIs, and works in a shared memory architecture [30], while the one described in [66] is actually a Beowulf cluster working in local memory. Moreover, the parallel instantiation strategy of [66] is applicable only to a subset of the program rules (those not defining domain predicates), and is, in general, unable to exploit parallelism fruitfully in the case of programs with a small number of rules. Importantly, the parallelization strategy of [66] *statically* assigns a rule per processing unit; whereas, in our approach, both the extension of predicates and split sizes are dynamically computed (and updated at different iterations of the semi-naïve evaluation) while the instantiation process is running. Note also that our parallelization techniques could be adapted for improving other ASP instantiators like Lparse [67] and Gringo [68]. Concerning works related to parallel instantiation, it is worth remembering that, the Single Rule parallelism employed in our system is related to the *copy and constrain* technique for parallelizing the evaluation of deductive databases [69, 70, 71, 72, 73]. In many of the mentioned works (dating back to 90's), only restricted classes of Datalog programs are parallelized; whereas, the most general ones (reported in [70, 72]) are applicable to normal Datalog programs. Clearly, none of them consider the peculiarities of disjunctive programs and unstratified negation. More in detail, [70] provides the theoretical foundations for the *copy and constrain* technique, whereas [72] enhances it in such a way that the network communication overhead in distributed systems can be minimized. The copy and constrain technique works as follows: rules are replicated with additional constraints attached to each copy; such constraints are generated by exploiting a hash function and allow for selecting a subset of the tuples. The obtained restricted rules are evaluated in parallel. The technique employed in our system shares the idea of splitting the instantiation of each rule, but has several differences that allow for obtaining an effective implementation. Indeed, in [70, 72] copied rules are generated and statically associated to instantiators according to an hash function which is independent of the current instance in input. In contrast, in our technique, the distribution of predicate extensions is performed dynamically, before assigning the rules to instantiators, by taking into account the "actual" predicate extensions. In this way, the non-trivial problem [72] of choosing an hash function that properly distributes the load is completely avoided in our approach. Moreover, the evaluation of conditions attached to the rule bodies during the instantiation phase would require to modify either the standard instantiation procedure (for efficiently selecting the tuples from the predicate extensions according to added constraints) or to incur a possible non negligible overhead due to their evaluation. Focusing on the *heuristics* employed on parallel databases, we mention [73] and [74]. In [73] a heuristics is described for balancing the distribution of load in the parallel evaluation of PARULEL, a language similar to Datalog. Here, load balancing is done by a manager server that records the execution times at each site, and exploits this information for distributing the load according to predictive dynamic load balancing (PDLB) protocols that "update and reorganize the distribution of workload at runtime by modifying the restrictions on versions of the rule program"[73]. In [74] the proposed heuristics were devised for both minimizing communication costs and choosing an opportune site for processing sub-queries among various network-connected database systems. In both cases, the proposed heuristics were devised and tuned for dealing with data distributed in several sites and their application to other architectures might be neither viable nor straightforward.

Concerning the parallelization of the propositional search phase, several studies were carried out in the context of ASP and related fields. Parallel approaches related to this phase usually exploit a divide-and-conquer scheme. This scheme works by recursively breaking down the original problem into two (or more) sub-problems until they become easy enough to be solved directly. The application of this methodology to backtracking search algorithms is due to some pioneer work in the distributed tree search [75, 76]; afterwards the approach was employed also in SAT solving [77, 78, 79]. In the context of ASP programs, the search space can be broken down creating two search branches whenever a choice point is reached. More in detail, as soon as a literal L is chosen, two new search tasks can be initialized one assuming L as true, and the other assuming $\neg L$ as true; the new search tasks must consider the truth values of the literals chosen before L (or $\neg L$) as fixed. In this way, as long as no possible backtracking can be performed before the choice of L, the technique allows the splitting of the search space, the partition of which can be searched in parallel. This general description simplify the approaches adopted in several previous works (*Claspar* [63], *Parstab* [62], *Platypus* [64] , and [65, 66]). All these approaches are based a single-heuristics search; our work differs from theirs in that it adopts a concurrent multi-heuristics search rather than breaking down the search space of a single-heuristics search. Concerning the application of several heuristics to a single search task, the *Claspfolio* ASP system [80] allows the dynamic selection of the most appropriate heuristics for the program in input, using an approach based on machine learning; however no parallelism is applied here. Approaches similar to the multi-heuristics were carried out in the the field of constraint satisfaction problems [81] and in the SAT community [53]; both of them exploit a number of search tasks employing different competing heuristics criteria in order to find a solution of the input problem. Concerning the the parallel lookahead technique, it was firstly introduced in [66], where an ASP system based on the sequential system Smodels [67], is presented which featuered a cluster-based computation. In this thesis we extended their approach to our framework; moreover we faced the consequences of porting the technique to the case of multicore/multiprocessor architectures. To the best of our knowledge, the approach to the parallel answer set checking described in this thesis is novel and current state-of-the-art ASP systems do not exploit similar techniques.

Chapter 8

Conclusion

In this thesis we have presented a number of parallel techniques enhancing the computation of answer sets. Recall that, the computation of ASP programs is commonly carried out as a two-step process. The first step, called instantiation amounts to computing a ground program $\mathcal{P}'$ semantically equivalent to the input one. In the second step the answer sets of the propositional program $\mathcal{P}'$ are computed by exploiting a backtracking search algorithm endowed with a stability checking procedure (note that, this latter is a co-NP-complete task in general). All the above-mentioned steps were subject of investigation in this thesis, and a number of parallelization strategies were devised, implemented and assessed.

Instantiation. The main contribution of this thesis is a strategy for the parallel instantiation of programs. This strategy applies three levels of parallelism, in particular: (*i*) the input program is first decomposed in modules that can be safely evaluated in parallel without the need for enforcing mutual exclusion in the main data structures of the system; (*ii*) the rules of each module are evaluated in parallel according to a parallel semi naif algorithm; (*iii*) the instantiation of each single rule of the input program carried out in parallel. Heuristics for load balancing and granularity control were also designed in order to optimize the workload distribution among the processors.

Propositional search. In addition to the above-mentioned parallel instantiation strategy, we investigated the problem of pushing parallelism in the evaluation of propositional programs. In particular, (*a*) a multi-heuristics parallel search was designed that executes in parallel a number of instances of a propositional search solver, each of which is driven by a different branching heuristic criterion; whatever the program at hand, the system prints an answer set within the same time as the best performing static heuristic selection. Moreover, (*b*) a technique was designed for parallelizing the computation of heuristic functions for candidate branching choices in look-ahead-based solvers. Eventually, (*c*) we proposed also a parallel model checking approach for exploiting parallelism in the stability checking procedure.

Implementation and Experiments. These techniques were implemented in the state-of-the-art ASP solver DLV. An extensive experimental analysis was carried out for assessing the implemented prototypes. Experiments confirmed the efficacy of the proposed techniques for enhancing both the instantiation process and the propositional search. Indeed, the efficiency of implemented prototypes rapidly reaches good levels, and superlinear speedups were obtained in several cases. Importantly, the prototypes offered a very good performance already when only two CPUs were enabled (i.e. for the largest majority of the commercially-available hardware at the time of this writing), and efficiency remains at a very good level when up to 8 CPUs are available.

Summarizing, the techniques presented in this thesis allow the effective exploitation of parallelism in the evaluation of ASP programs.

Bibliography

[1] Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[2] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[3] Vladimir Lifschitz. Answer Set Planning. In Danny De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 23–37, Las Cruces, New Mexico, USA, November 1999. The MIT Press.

[4] Victor W. Marek and Mirosław Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In Krzysztof R. Apt, V. Wiktor Marek, Mirosław Truszczyński, and David S. Warren, editors, *The Logic Programming Paradigm – A 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.

[5] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, 2003.

[6] Michael Gelfond and Nicola Leone. Logic Programming and Knowledge Representation – the A-Prolog perspective . *Artificial Intelligence*, 138(1–2):3–38, 2002.

[7] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, July 2006.

[8] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[9] Moshe Y. Vardi. Complexity of relational query languages. In *Proceedings of the 14th Symposium on Theory of Computation (STOC)*, pages 137–146, 1982.

[10] Chitta Baral and Michael Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19/20:73–148, 1994.

[11] Colin Bell, Anil Nerode, Raymond T. Ng, and V.S. Subrahmanian. Mixed Integer Programming Methods for Computing Nonmonotonic Deductive Databases. *Journal of the ACM*, 41:1178–1215, 1994.

[12] V.S. Subrahmanian, Dana Nau, and Carlo Vago. WFS + Branch and Bound = Stable Models. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):362–377, June 1995.

[13] Tomi Janhunen, Ilkka Niemelä, Dietmar Seipel, Patrik Simons, and Jia-Huai You. Unfolding Partiality and Disjunctions in Stable Model Semantics. *ACM Transactions on Computational Logic*, 7(1):1–37, January 2006.

[14] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, June 2002.

[15] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 386–392. Morgan Kaufmann Publishers, January 2007.

[16] Fangzhen Lin and Yuting Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1–2):115–137, 2004.

[17] Yuliya Lierler and Marco Maratea. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7)*, volume 2923 of *LNAI*, pages 346–350. Springer, January 2004.

[18] Christian Anger, Kathrin Konczak, and Thomas Linke. `NoMoRe`: A System for Non-Monotonic Reasoning. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings*, volume 2173 of *Lecture Notes in AI (LNAI)*, pages 406–410. Springer Verlag, September 2001.

[19] Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Integrating Inconsistent and Incomplete Data Sources. In *Proceedings of SEBD 2002*, Portoferraio, Isola d'Elba, 2002. 299–308.

[20] Luigia Carlucci Aiello and Fabio Massacci. Verifying security protocols as planning in logic programming. *ACM Transactions on Computational Logic*, 2(4):542–580, 2001.

[21] Chitta Baral and Cenk Uyan. Declarative Specification and Solution of Combinatorial Auctions Using Logic Programming. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-01)*, volume 2173 of *Lecture Notes in AI (LNAI)*, pages 186–199. Springer Verlag, 2001.

[22] Giovanni Grasso, Salvatore Iiritano, Nicola Leone, Vincenzino Lio, Francesco Ricca, and Francesco Scalise. An asp-based system for team-building in the gioia-tauro seaport. In *PADL (2010), 12th International Symposium, Madrid, Spain, January 18-19, 2010*, volume 5937 of *LNCS*, pages 40–42. Springer, 2010.

[23] Salvatore Maria Ielpa, Salvatore Iiritano, Nicola Leone, and Francesco Ricca. An asp-based system for e-tourism. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR '09, pages 368–381. Springer-Verlag, 2009.

[24] G. Grasso, N. Leone, M. Manna, and F. Ricca. Asp at work: Spin-off and applications of the dlv system. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning: Essays in Honor of Michael Gelfond*, Lecture Notes in Computer Science, pages 1–20. Springer-Verlag, 2011.

[25] Simona Citrigno, Thomas Eiter, Wolfgang Faber, Georg Gottlob, Christoph Koch, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The `dlv` System: Model Generator and Application Frontends. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *Proceedings of the 12th Workshop on Logic Programming (WLP'97), Research Report PMS-FB10*, pages 128–137, München, Germany, September 1997. LMU München.

[26] Wolfgang Faber, Nicola Leone, Cristinel Mateis, and Gerald Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. In INAP Organizing Committee, editor, *Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99)*, pages 135–139. Prolog Association of Japan, September 1999.

[27] Nicola Leone, Simona Perri, and Francesco Scarcello. Improving ASP Instantiators by Join-Ordering Methods. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria*, volume 2173 of *Lecture Notes in AI (LNAI)*, pages 280–294. Springer Verlag, September 2001.

[28] Nicola Leone, Simona Perri, and Francesco Scarcello. BackJumping Techniques for Rules Instantiation in the DLV System. In *Proceedings of the 10th International Workshop on Non-monotonic Reasoning (NMR 2004), Whistler, BC, Canada*, pages 258–266, 2004.

[29] Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Francesco Ricca. On look-ahead heuristics in disjunctive logic programming. *Annals of Mathematics and Artificial Intelligence*, 51(2–4):229–266, 2007.

[30] William Stallings. *Operating systems (3rd ed.): internals and design principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[31] Jeffrey D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, 1989.

[32] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In José Júlio Alferes and João Leite, editors, *Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004)*, volume 3229 of *Lecture Notes in AI (LNAI)*, pages 200–212. Springer Verlag, September 2004.

[33] V. Wiktor Marek and V.S. Subrahmanian. The Relationship between Logic Program Semantics and Non-Monotonic Reasoning. In *Proceedings of the 6th International Conference on Logic Programming – ICLP'89*, pages 600–617. MIT Press, 1989.

[34] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. *Information and Computation*, 135(2):69–112, June 1997.

[35] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a Theory of Declarative Knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Publishers, Inc., Washington DC, 1988.

[36] Teodor C. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann Publishers, Inc., 1988.

[37] Rachel Ben-Eliyahu and Rina Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.

[38] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative Problem-Solving Using the DLV System. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

[39] Stanislaw P. Radziszowski. Small Ramsey Numbers. *The Electronic Journal of Combinatorics*, 1, 1994. Revision 9: July 15, 2002.

[40] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[41] Marco Cadoli, Thomas Eiter, and Georg Gottlob. Default Logic as a Query Language. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):448–463, May/June 1997.

[42] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7:201–215, 1960.

[43] Wolfgang Faber, Nicola Leone, Simona Perri, and Gerald Pfeifer. Efficient Instantiation of Disjunctive Databases. Technical Report DBAI-TR-2001-44, Institut für Informationssysteme, Technische Universität Wien, Austria, November 2001. Online at `http://www.dbai.tuwien.ac.at/local/reports/dbai-tr-2001-44.pdf`.

[44] Wolfgang Faber. *Enhancing Efficiency and Expressiveness in Answer Set Programming Systems*. PhD thesis, Institut für Informationssysteme, Technische Universität Wien, 2002.

[45] Yuliya Lierler. Disjunctive Answer Set Programming via Satisfiability. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings*, volume 3662 of *Lecture Notes in Computer Science*, pages 447–451. Springer Verlag, September 2005.

[46] Fangzhen Lin and Yuting Zhao. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-2002)*, Edmonton, Alberta, Canada, 2002. AAAI Press / MIT Press.

[47] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Pushing Goal Derivation in DLP Computations. In Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, volume 1730 of *Lecture Notes in AI (LNAI)*, pages 177–191, El Paso, Texas, USA, December 1999. Springer Verlag.

[48] Francesco Calimeri, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Pruning Operators for Disjunctive Logic Programming Systems. *Fundamenta Informaticae*, 71(2–3):183–214, 2006.

[49] Chu Min Li and Anbulagan. Heuristics Based on Unit Propagation for Satisfiability Problems. In *Proceedings of the Fourteen International Joint Conference on Artificial Intelligence (IJCAI) 1997*, pages 366–371, Nagoya, Japan, August 1997.

[50] Christoph Koch, Nicola Leone, and Gerald Pfeifer. Enhancing Disjunctive Logic Programming Systems by SAT Checkers. *Artificial Intelligence*, 15(1–2):177–212, December 2003.

[51] Francesco Calimeri, Simona Perri, and Francesco Ricca. Experimenting with Parallelism for the Instantiation of ASP Programs. *Journal of Algorithms in Cognition, Informatics and Logics*, 63(1–3):34–54, 2008.

[52] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub, Marius Thomas Schneider, and Stefan Ziller. A portfolio solver for answer set programming: preliminary report. In *Proceedings of the 11th international conference on Logic programming and nonmonotonic reasoning*, LPNMR'11, pages 352–357, Berlin, Heidelberg, 2011. Springer-Verlag.

[53] Youssef Hamadi and Lakhdar Sais. Manysat: a parallel sat solver. *JOURNAL ON SATISFIABILITY, BOOLEAN MODELING AND COMPUTATION (JSAT)*, 6, 2009.

[54] Tobias Schubert, Matthew Lewis, and Bernd Becker. Pamira - a parallel sat solver with knowledge sharing. In *Proceedings of the Sixth International Workshop on Microprocessor Test and Verification*, pages 29–36, Washington, DC, USA, 2005. IEEE Computer Society.

[55] Martin Gebser, Lengning Liu, Gayathri Namasivayam, André Neumann, Torsten Schaub, and Mirosław Truszczyński. The first answer set programming system competition. In Chitta Baral, Gerhard Brewka, and John Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning — 9th International Conference, LPNMR'07*, volume 4483 of *Lecture Notes in Computer Science*, pages 3–17, Tempe, Arizona, May 2007. Springer Verlag.

[56] M. Denecker, J. Vennekens, S. Bond, M. Gebser, and M. Truszczyński. The second answer set programming competition. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR '09, pages 637–654. Springer-Verlag, 2009.

[57] Donald E. Knuth. *The Stanford GraphBase : A Platform for Combinatorial Computing.* ACM Press, New York, 1994.

[58] Patrik Simons. *Extending and Implementing the Stable Model Semantics.* PhD thesis, Helsinki University of Technology, Finland, 2000.

[59] Bart Selman and Henry Kautz, 1997. `ftp://ftp.research.att.com/dist/ai/`.

[60] James M. Crawford and Larry D. Auton. Experimental Results on the Crossover Point in Random 3SAT. *Artificial Intelligence*, 81(1–2):31–57, March 1996.

[61] Ilkka Niemelä and Patrik Simons. Efficient Implementation of the Well-founded and Stable Model Semantics. In Michael J. Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming (ICLP'96)*, pages 289–303, Bonn, Germany, September 1996. MIT Press.

[62] Raphael A. Finkel, Victor W. Marek, Neil Moore, and Miroslaw Truszczynski. Computing stable models in parallel. In *Answer Set Programming, Towards Efficient and Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Intl. ASP'01 Workshop*, pages 72–76, Stanford, March 2001.

[63] Enrico Ellguth, Martin Gebser, Markus Gusowski, Benjamin Kaufmann, Roland Kaminski, Stefan Liske, Torsten Schaub, Lars Schneidenbach, and Bettina Schnor. A simple distributed conflict-driven answer set solver. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR '09, pages 490–495. Springer-Verlag, 2009.

[64] Jean Gressmann, Tomi Janhunen, Robert E. Mercer, Torsten Schaub, Sven Thiele, and Richard Tichy. Platypus: A Platform for Distributed Answer Set Solving. In *Proceedings of Logic Programming and Nonmonotonic Reasoning, 8th International Conference (LPNMR)*, pages 227–239, Diamante, Italy, September 2005.

[65] Enrico Pontelli and Omar El-Khatib. Exploiting Vertical Parallelism from Answer Set Programs. In *Answer Set Programming, Towards Efficient and*

Scalable Knowledge Representation and Reasoning, Proceedings of the 1st Intl. ASP'01 Workshop, pages 174–180, Stanford, March 2001.

[66] Marcello Balduccini, Enrico Pontelli, Omar Elkhatib, and Hung Le. Issues in parallel execution of non-monotonic reasoning systems. *Parallel Computing*, 31(6):608–647, 2005.

[67] Ilkka Niemelä and Patrik Simons. Smodels – An Implementation of the Stable Model and Well-founded Semantics for Normal Logic Programs. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in AI (LNAI)*, pages 420–429, Dagstuhl, Germany, July 1997. Springer Verlag.

[68] Martin Gebser, Torsten Schaub, and Sven Thiele. GrinGo : A New Grounder for Answer Set Programming. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271. Springer, 2007.

[69] Ouri Wolfson and Abraham Silberschatz. Distributed Processing of Logic Programs. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 329–336, Chicago, Illinois, USA, June 1988.

[70] Ouri Wolfson and Aya Ozeri. A new paradigm for parallel and distributed rule-processing. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 133–142, New York, NY, USA, 1990.

[71] Sumit Ganguly, Abraham Silberschatz, and Shalom Tsur. A Framework for the Parallel Processing of Datalog Queries. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 143–152, 1990.

[72] Weining Zhang, Ke Wang, and Siu-Cheung Chau. Data Partition and Parallel Evaluation of Datalog Programs. *IEEE Transactions on Knowledge and Data Engineering*, 7(1):163–176, 1995.

[73] Hasanat M. Dewan, Salvatore J. Stolfo, Mauricio Hernández, and Jae-Jun Hwang. Predictive dynamic load balancing of parallel and distributed rule and query processing. In Pascal Van Hentenryck, editor, *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 277–288, New York, NY, USA, 1994. ACM.

[74] Michael J. Carey and Hongjun Lu. Load balancing in a locally distributed db system. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, SIGMOD '86, pages 108–119, New York, NY, USA, 1986. ACM.

[75] Raphael Finkel and Udi Manber. Dib - a distributed implementation of backtracking. *ACM Trans. Program. Lang. Syst.*, 9:235–256, March 1987.

[76] Chris Ferguson and Richard E. Korf. Distributed tree search and its application to alpha-beta pruning. In *AAAI'88*, pages 128–132, 1988.

[77] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. Psato: a distributed propositional prover and its application to quasigroup problems. *J. Symb. Comput.*, 21:543–560, June 1996.

[78] Luís Gil, Paulo Flores, and Luís M. Silveira. PMSat: a parallel version of MiniSAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:71–98, 2008.

[79] Matthew Lewis, Tobias Schubert, and Bernd Becker. Multithreaded sat solving. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, ASP-DAC '07, pages 926–931, Washington, DC, USA, 2007. IEEE Computer Society.

[80] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub, Marius Schneider, and Stefan Ziller. A portfolio solver for answer set programming : preliminary report. In *Logic Programming and Nonmonotonic Reasoning*. 2011.

[81] Georg Ringwelski and Youssef Hamadi. Boosting distributed constraint satisfaction. In *In Int. Conf. on Principles and Practice of Constraint Programming (CP)*, pages 549–562, 2005.