Università degli Studi della Calabria

Dipartimento di Matematica

Dottorato di Ricerca in Matematica ed Informatica

XX Ciclo

Settore Disciplinare INF/01 INFORMATICA

Tesi di Dottorato

Efficient Evaluation of Disjunctive Logic Programs

Gelsomina Catalano

Supervisori

Prof. Nicola Leone

Dott.ssa Simona Perri

Coordinatore

Prof. Nicola Leone

Anno Accademico 2007 - 2008

Efficient Evaluation of Disjunctive Logic Programs

Gelsomina Catalano

Dipartimento di Matematica,
Università della Calabria
87036 *Rende, Italy*
email : catalano@mat.unical.it

Sommario

Agli inizi degli anni ‘80, Jack Minker propose di accrescere la potenza della programmazione logica consentendo l’utilizzo della disgiunzione nelle teste delle regole e specificando come l’assunzione di mondo chiuso potesse essere estesa al linguaggio risultante, chiamato Programmazione Logica Disgiuntiva (DLP) [Minker, 1982; 1994]. Più tardi, Michael Gelfond e Vladimir Lifschitz fornirono una semantica per la DLP, detta Answer Set Semantics [Gelfond and Lifschitz, 1991], che ha ricevuto larghi consensi nella comunità scientifica ed è ora generalmente adottata per la DLP (detta anche Answer Set Programming – ASP). In accordo a tale semantica un programma logico disgiuntivo può avere più modelli alternativi (ma anche nessuno) ognuno corrispondente a una possibile visione del mondo rappresentato dal programma.

Il linguaggio per la rappresentazione della conoscenza DLP è molto espressivo in un senso matematicamente preciso; la DLP può rappresentare *ogni* problema nella classe di complessità Σ_2^P (NP^{NP}) [Eiter *et al.*, 1997b]. Dunque, sotto assunzioni ampiamente accettate, la DLP risulta strettamente più espressiva della programmazione logica *normale (senza disgiunzione)*, la cui espressività è limitata alle proprietà decidibili in NP. L’espressività della Programmazione Logica Disgiuntiva, ha importanti implicazioni pratiche, poichè esistono problemi che possono essere rappresentati tramite un programma logico disgiuntivo, ma che non è possibile esprimere con programmi logici senza disgiunzione, considerata la loro complessità [Eiter *et al.*, 1997b]. Inoltre, la disgiunzione consente di rappresentare in modo più semplice e naturale problemi di classi di complessità più bassa. La DLP, con la Answer Set Semantics, è oggi ampiamente riconosciuta come uno strumento potente per la rappresentazione della conoscenza e il ragionamento di senso comune [Baral and Gelfond, 1994; Lobo *et al.*, 1992; Wolfinger, 1994; Eiter *et al.*, 1999; Gelfond and Lifschitz, 1991; Lifschitz, 1996; Minker, 1994; Baral, 2002].

L’elevata complessità della DLP ha scoraggiato per molti anni la realizzazione di sistemi che implementassero tutte le caratteristiche di tale linguaggio. Dopo alcuni anni di ricerca sia teorica che algoritmica, oggi esistono diversi sistemi che supportano la DLP o parte di essa. Oltre ai sistemi di programmazione logica (non disgiuntivi) Smodels [Simons *et al.*, 2002] e ASSAT [Lin and Zhao, 2002], sono disponibili anche alcuni sistemi di programmazione logica (disgiuntiva): DLV [Leone *et al.*, 2006], GnT [Janhunen *et al.*, 2003], and cmodels-3 [Lierler, 2005].

In questa tesi ci concentriamo sul sistema DLV, che è riconosciuto essere lo stato dell'arte della Programmazione Logica Disgiuntiva. DLV è ampiamente sfruttato in tutto il mondo sia a scopo di ricerca che didattico. Per esempio, è stato impiegato al CERN, il Laboratorio Europeo di Fisica delle Particelle di Ginevra, per un'applicazione di Basi di Dati deduttive che coinvolge la manipolazione di conoscenza complessa su basi di dati di grandi dimensioni.

La compagnia polacca Rodan Systems S.A. sfrutta DLV in uno strumento per scoprire le manipolazioni dei prezzi e l'uso non autorizzato di informazioni confidenziali. Noi crediamo che la forza di DLV – la sua espressività e l' implementazione solida – lo renda attrattivo per tali applicazioni complesse. Anche dal punto di vista dell'efficienza, esso è competitivo con i più avanzati sistemi in quest'area come confermano i recenti confronti e valutazione delle prestazioni [Leone *et al.*, 2006; Dix *et al.*, 2003; Arieli *et al.*, 2004], e i risultati della Prima Competizione di Sistemi Answer Set Programming `http://asparagus.cs.uni-potsdam.de/contest/`, in cui DLV è risultato vincitore per le categorie DLP e MGS Competition (detta anche categoria "Royal").

Lo sviluppo di DLV è iniziato nel 1996 al Politecnico di Vienna, nell'ambito di un progetto finanziato dalla Austrian Science Funds (FWF); oggi, DLV è oggetto di una cooperazione internazionale tra l'Università della Calabria e il Politecnico di Vienna.

Il presente lavoro di tesi è incentrato sullo studio della Programmazione Logica Disgiuntiva e l'ottimizzazione del sistema DLV, che implementa la DLP.

I nostri studi hanno evidenziato che negli ultimi anni, la disponibilità di sistemi DLP affidabili, ha indotto a sfruttare la DLP in diverse aree applicative, ma i sistemi attuali non sono sufficientemente efficienti per molte di queste applicazioni.

Questo lavoro affronta questo aspetto, proponendosi di superare questa limitazione, migliorando l'efficienza dei sistemi DLP e del sistema DLV tramite il progetto e l'implementazione di nuove tecniche di ottimizzazione.

I moduli della maggior parte dei sistemi DLP operano su un'istanziazione ground del programma in input, cioè un programma che non contiene alcuna variabile, ma è semanticamente equivalente all'input originale [Eiter *et al.*, 1997c]. Ogni programma P in input, inizialmente sottoposto alla cosiddetta procedura di istanziazione (detta anche istanziatore) che calcola, a partire da P, un programma ground P' semanticamente equivalente. Poichè questa fase può essere molto costosa, avere un buon istanziatore è un aspetto cruciale dei sistemi DLP. La ragione è dovuta al fatto che ogni atomo di ciascuna regola può essere istanziato utilizzando ogni costante dell'Universo di Herbrand del programma, con una evidente esplosione esponenziale. L'istanziatore dovrebbe essere in grado di produrre un programma ground P' avente gli stessi answer set di P e tale che: (i) P' sia calcolato efficientemente da P, e (ii) P' sia il più piccolo possibile, e quindi possa essere valutato più efficientemente da un solver DLP.

Alcune applicazioni della DLP in aree emergenti come la gestione della conoscenza e l'integrazione delle informazioni, in cui devono essere processate grandi quantità di dati, hanno reso evidente la necessità di migliorare significativamente gli istanziatori DLP.

La nostra attenzione è stata rivolta al modulo di instanziazione di DLV, investigando nuove possibili direzioni per aumentarne l'efficienza. In particolare, in questa tesi, presentiamo due proposte per migliorare la procedura di istanziazione:

- Una nuova tecnica di *Backjumping* per l'istanziatore di DLV e
- Nuove tecniche di *Indicizzazione* per l'istanziatore di DLV

Di seguito descriviamo brevemente queste due linee di ricerca.

Backjumping. Proponiamo di sfruttare tecniche di backjumping che riduce la taglia del programma ground generato e ottimizza il tempo di esecuzione necessario a produrlo. In particolare, data una regola r che deve essere resa ground, tale algoritmo sfrutta informazioni semantiche e strutturali su r, per calcolare efficientemente le istanze ground di r, evitando la generazione di regole "inutili". Cioè per ogni regola r si calcola solo un sottoinsieme rilevante delle sue istanze ground, preservandone la semantica.

Implementiamo questo algoritmo in DLV e conduciamo un'attività di sperimentazione su un'ampia collezione di problemi. I risultati sperimentali sono molto positivi: la nuova tecnica migliora sensibilmente l'efficienza del sistema DLV su molte classi di problemi.

Indicizzazione. Proponiamo di adoperare tecniche di indicizzazione per migliorare le performance della procedura di istanziazione di DLV, cioè tecniche per il progetto e l'implementazione di strutture dati che permettano di accedere più efficientemente a grandi datasets. In particolare, adattiamo al nostro contesto, una tecnica classica di indicizzazione sul primo argomento e proponiamo una strategia di indicizzazione “on demand” in base alla quale gli indici non sono predeterminati, ma piuttosto vengono calcolati su un argomento qualsiasi durante la valutazione (e solo se sfruttabili). In più definiamo due euristiche che possono essere usate per stabilire l’argomento più appropriato da indicizzare, quando esistono diverse possibilità.

Inoltre, implementiamo le tecniche di indicizzazione proposte in DLV e confrontiamo sperimentalmente le nostre strategie su una collezione di problemi provenienti da diversi domini comprese anche istanze di problemi reali. Il quadro generale risultante dagli esperimenti è molto positivo:

- Tutte le tecniche proposte e testate permettono di ottenere notevoli miglioramenti per l’esecuzione dell’istanziazione.

- Lo schema di indicizzazione on demand dà risultati migliori rispetto al classico schema sul primo argomento in un numero maggiore di casi e le performance migliorano particolarmente quando viene utilizzata una buona euristica.

In definitiva, i metodi proposti migliorano sensibilmente l’efficienza dell’ istanziatore di DLV, consentendo l’utilizzo del sistema anche in applicazioni data-intensive. Comunque, per verificare ulteriormente la potenza del nuovo istanziatore conduciamo una profonda analisi sperimentale per confrontarlo con gli altri due più popolari istanziatori, Lparse [Niemelä and Simons, 1997; Syrjänen, 2002] e *GrinGo* [Gebser *et al.*, 2007b]. L’analisi conferma che, il nuovo istanziatore ha performance migliori degli altri su tutti i problemi testati, mentre il vecchio mostra performance simili agli altri.

I risultati presentati in questa tesi sono rilevanti anche per altri due aspetti: da una parte, l’istanziatore di DLV può essere sfruttato proficuamente da altri sistemi che non hanno un istanziatore proprio, per esempio ASSAT [Lin and Zhao, 2002] e Cmodels [Lierler and Maratea, 2004; Babovich, since 2002]. Infatti, questi sistemi possono usare DLV per ottenere il programma ground (lanciando DLV con l’opzione “-instantiate”), e poi applicare le proprie procedure per la valutazione del programma ground; d’altra parte, tutti i metodi proposti sono abbastanza generali e quindi, possono essere facilmente adattati per essere integrati nella fase di calcolo di altri istanziatori. In realtà la nostra tecnica di backjumping è stata già integrata in altri due istanziatori, *GrinGo* e FO+ [Wittocx *et al.*, 2008], con buoni risultati. Le buone performance di *GrinGo* in alcuni degli esperimenti condotti, infatti, sono dovuti all’utilizzo della nostra tecnica.

I principali contributi della tesi possono essere riassunti come segue:

1. Studiamo la DLP, la sua complessità e il suo utilizzo per la rappresentazione della conoscenza e il ragionamento non monotono.

2. Progettiamo un nuovo metodo, basato su una tecnica di backjumping, che consente di ridurre sia la taglia dell'istanziazione dei programmi DLP che il tempo necessario per generarla. Implementiamo il metodo proposto in DLV e conduciamo un'attività di sperimentazione.

3. Definiamo due nuove strategie di indicizzazione, per ottimizzare il tempo di istanziazione di DLV. Inoltre, implementiamo il metodo proposto nel sistema DLV ed effettuiamo un'analisi sperimentale.

4. Confrontiamo l'istanziatore di DLV con altri due istanziatori, Lparse e *GrinGo* e discutiamo i risultati.

Contents

Introduction

Context

Disjunctive Logic Programming At the beginning of the Eighties, Jack Minker proposed to empower logic programming by allowing for disjunction in the head of the rules, and specified how the Closed World Assumption could be extended to the resulting language, called Disjunctive Logic Programming (DLP) [Minker, 1982; 1994]. Later on, Michael Gelfond and Vladimir Lifschitz provided a semantics for DLP, called Answer Set Semantics [Gelfond and Lifschitz, 1991], which received a large consensus in the research community, and is now generally adopted for DLP (which is also called Answer Set Programming – ASP). According to this semantics, a disjunctive logic program may have several alternative models (but possibly none) each corresponding to a possible view of the world.

The knowledge representation language DLP is very expressive in a precise mathematical sense; DLP can represent *every* problem in the complexity class Σ_2^P (NP^{NP}) [Eiter *et al.*, 1997b]. Thus, under widely believed assumptions, DLP is strictly more expressive than *normal (disjunction-free)* logic programming, whose expressiveness is limited to properties decidable in NP. The expressiveness of Disjunctive Logic Programming has practical implications, since relevant practical problems can be represented by disjunctive logic programs, while they cannot be expressed by logic programs without disjunction, given current complexity beliefs [Eiter *et al.*, 1997b]. In addition, disjunction often allows for representing problems of lower complexity in a simpler and more natural fashion.

DLP, with Answer Set Semantics, is now widely recognized as a valuable tool for knowledge representation and commonsense reasoning [Baral and Gelfond, 1994; Lobo *et al.*, 1992; Wolfinger, 1994; Eiter *et al.*, 1999; Gelfond and Lifschitz, 1991; Lifschitz, 1996; Minker, 1994; Baral, 2002].

The DLV System The high complexity of DLP discouraged for many years the development of systems implementing all the features of this powerful language. After some years of considerable efforts on both theoretical and algorithmic research, several systems exist which support DLP or part of it. Besides the (non-disjunctive) logic programming systems like Smodels [Simons *et al.*, 2002], and ASSAT [Lin and Zhao, 2002], some full (disjunctive) logic programming systems are available: DLV [Leone *et al.*, 2006], GnT [Janhunen *et al.*, 2003], and cmodels-3 [Lierler, 2005].

In this thesis we focus on the DLV system which is generally recognized to be the state-of-the-art implementation of Disjunctive Logic Programming. DLV is widely exploited all over the world for both research and educational purposes. For instance, it has been employed at CERN, the European Laboratory for Particle Physics located near Geneva, for an advanced deductive database application that involves complex knowledge manipulation on large-sized databases. The Polish company Rodan Systems S.A. exploits DLV in a tool for the detection of price manipulations and unauthorized use of confidential information, which is used by the Polish Securities and Exchange Commission. We believe that the strengths of DLV – its expressivity and solid implementation – make it attractive for such hard applications. Also from the viewpoint of efficiency, it is competitive with the most advanced systems in this area as confirmed by recent comparison and benchmarks [Leone *et al.*, 2006; Dix *et al.*, 2003; Arieli *et al.*, 2004], and by the results of the First Answer Set Programming System Competition `http://asparagus.cs.uni-potsdam.de/contest/`, where DLV won the both the DLP and the MGS Competition (also called the "Royal" category).

The development of DLV started in 1996 at the Vienna University of Technology, in a research project funded by the Austrian Science Funds (FWF); at present, DLV is the subject of an international cooperation between the University of Calabria and the Vienna University of Technology.

Motivation and Main Contribution

This thesis concerns the study of Disjunctive Logic Programming and the optimization of the DLV system, which implements the DLP itself.

Our studies pointed out that in the latest years, the availability of reliable DLP systems induced many people to start exploiting DLP in several application areas, but the current systems are not efficient enough for many of these applications.

This work faces this issue, aiming at overcoming this limitation by increasing the efficiency of the DLP systems, and of the DLV system in particular, through the design and the implementation of new optimization techniques.

The kernel modules of most DLP systems operate on a ground instantiation of the input program, i.e., a program that does not contain any variable, but is (semantically) equivalent to the original input [Eiter *et al.*, 1997c]. Indeed, any given program P first undergoes the so called instantiation process, that computes from P a semantically equivalent ground program P'. Since this instantiation phase may be computationally very expensive, having a good instantiation procedure (also called instantiator) is a key feature of DLP systems. The instantiator should be able to produce a ground program P' having the same answer sets as P such that: (i) P' is computed efficiently from P, and (ii) P' is as small as possible, and thus can be evaluated efficiently by a DLP solver (recall that, in the worst case, every DLP solver takes exponential time in the size of P' – a polynomial reduction in the size of P', may thus give an exponential gain in the computational time).

Some emerging application areas of DLP, like knowledge management and information integration,[1] where large amount of data are to be processed, make very evident the need of improving DLP instantiators significantly.

We spent our work on the instantiation module of DLV, investigating new directions for improving its efficiency. In particular, in this thesis, we present two proposals for improving the instantiation procedure:

- A new *Backjumping* technique for the DLV Instantiator, and
- New *Indexing* techniques for the DLV Instantiator.

In the following, we briefly describe these two research lines.

Backjumping. We propose to exploit backjumping techniques in the rule instantiation process of DLV. We design a new structure-based backjumping algorithm for rule instantiation, which reduces the size of the generated ground instantiation and optimizes the execution time which is needed to generate it. In particular, given a rule r to be grounded, our algorithm exploits both the semantical and the structural information about r for computing efficiently the ground instances of r,

[1]The application of DLP in these areas has been investigated also in the EU projects INFOMIX IST-2001-33570, and ICONS IST-2001-32429, and is profitably exploited by Exeura s.r.l., a spin-off of University of Calabria having precisely this mission.

avoiding the generation of “useless” rules. That is, from each general rule r, we compute only a relevant subset of its ground instances, avoiding the generation of “useless” instances, while fully preserving the semantic of the program.

We implement this algorithm in DLV and we carry out an experimentation activity on an ample collection of benchmark problems. The experimental results are very positive: the new technique improves sensibly the efficiency of the DLV system on many program classes.

Indexing. We propose to employ main-memory indexing techniques for enhancing the performance of the instantiation procedure of DLV instantiator, that is techniques for the design and the implementation of data structures that allow to efficiently access to large datasets. In particular, we adapt a classical first argument indexing schema to our context, and propose an more general indexing strategy where anu argument can be indexed and indexes are not pre-established rather they are computed during the evaluation (and only if exploitable). Moreover, we define two heuristics which can be used for determining the most appropriate argument to be indexed, when more than one possibility exists.

Furthermore, we implement the proposed indexing techniques in DLV and we experimentally compare our strategies on a collection of benchmark problems taken from different domains including also a number of real-world instances. The overall picture resulting from the experiments is very positive:

- All tested indexing techniques cause the istantiation stage to achieve noticeable improvements.

- The on demand indexing schema gives better results w.r.t the classical first argument schema in a wider range of cases and performance improve notably when a good heuristics is utilized.

Summarizing, the proposed methods sensibly improve the performance of the DLV Instantiator, allowing for the exploitation of the system also in case of data intensive applications. However, to further check the strength of the enhanced instantiator, we also carry out a deep experimental analysis for comparing it w.r.t. the other two popular instantiators, namely Lparse [Niemelä and Simons, 1997; Syrjänen, 2002] and *GrinGo* [Gebser *et al.*, 2007b]. The analysis confirm that, the new DLV instantiator outperforms them on all the tested problems, while the old one showed similar performance.

The results presented in this thesis are relevant also for two other aspects: on the one hand, the DLV istantiator can profitably be exploited by other systems, which do not have their own instantiators like, e.g., ASSAT [Lin and Zhao, 2002]and Cmodels [Lierler and Maratea, 2004; Babovich, since 2002]. Indeed, these systems can use DLV to obtain the ground program (by running DLV with option “-instantiate”), and then apply their own procedures for the evaluation of the ground program.[2]; on the other hand, all the proposed methods are quite general and, thus, they can be easily adapted in order to be integrated in the computation phase of other instantiators. Actually, our new backjumping technique has been already integrated in two other instantiators, namely *GrinGo* and FO+ [Wittocx *et al.*, 2008], with very good results. The good performance of *GrinGo* in some of the conducted experiments are, indeed, due to the use of our technique.

Briefly, the main contribution of the thesis is the following:

1. We study DLP, its complexity and its exploitation for knowledge representation and reasoning.

2. We design a new structure based backjumping method for reducing both the size and the time of the instantiation. We implement the proposed method in DLV and carry out an experimental activity.

3. We define new indexing strategies, for reducing the instantiation time of DLP programs. We implement the proposed techniques in the grounding engine of DLV system and perform an experimental analysis.

4. We compare the enhanced DLV instantiator with other two instantiators, namely Lparse and *GrinGo* and discuss the results.

Structure of the thesis

The thesis is organized as follows.

- In Chapter 1 we present Disjunctive Logic Programming extended by weak constraint. In particular, we provide a formal definition of the syntax of this language and its associated semantics, the *Answer Set Semantics*. Then, we describe the computation complexity of this language and we illustrate the usage of Disjunctive Logic Programming for knowledge representation and reasoning.

[2]Recall that DLV instantiator can deal also with normal nondisjunctive programs.

- In Chapter 2 we introduce the DLV system and give an overview of its architecture and implementation. The theoretical foundations of the implementation of DLV are also briefly discussed. The main procedure for the computation of the answer set semantics is then described.
- In Chapter 3 we provide a short description of the overall instantiation module of DLV, the Intelligent Grounding, and focus on the "heart" procedure of this module which produces the ground instances of a given rule.
- In Chapter 4 we present a new kind of structure-based backjumping algorithm for rule instantiation that can be used in order to improve the efficiency of the instantiation procedure of DLV. In particular, this algorithm allows to reduce the size of the generated ground instantiation and optimize the execution time which is needed to generate it.
- In Chapter 5 we propose to employ main-memory indexing techniques for enhancing the performance of the instantiation procedure of the ASP system DLV. In particular, we adapt a classical first argument indexing schema to our context, and propose an on demand indexing strategy where indexes are computed during the evaluation (and only if exploitable). Moreover, we define two heuristics which can be used for determining the most appropriate argument to be indexed, when more than one possibility exists. We discuss some key issues for its implementation into the DLV system and we then report the results of our experimentation activity on a number of benchmark problems.
- In Chapter 6 we compare the enhanced DLV instantiator with Lparse and *GrinGo*, other two instantiators and report the results of our experimental analysis.

Chapter 1

Disjunctive Logic Programming

In this chapter we present Disjunctive Logic Programming[1]. In particular, we provide a formal definition of the syntax of this language and its associated semantics, the *Answer Set Semantics*. This programming framework is also referred to as Answer Set Programming. (For further background, see [Lobo *et al.*, 1992; Eiter *et al.*, 1997b; Gelfond and Lifschitz, 1991]). Then, we describe the computation complexity of this language and we illustrate the usage of Disjunctive Logic Programming for knowledge representation and reasoning.

1.1 The language

In this section, we illustrate syntax and semantics of Disjunctive logic programming.

1.1.1 Syntax

A *term* is either a variable or a constant. An *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. A *literal* is a *positive literal* p or a *negative literal* not p, where p is an atom.

A *disjunctive rule* (*rule*, for short) r is a formula

$$a_1 \texttt{ v } \cdots \texttt{ v } a_n \text{ :- } b_1, \cdots, b_k, \text{ not } b_{k+1}, \cdots, \text{ not } b_m. \tag{1.1}$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are atoms and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \texttt{ v } \cdots \texttt{ v } a_n$ is called *head* of r, while the conjunction $b_1, \cdots, b_k$,

[1]From now on, when talking about DLP we actually refer to a rather recent extension of DLP itself by *weak constraints* [Buccafurri *et al.*, 2000] which are a powerful tool to express optimization problems.

$\text{not } b_{k+1}, \cdots, \text{ not } b_m$ is the *body* of r. We denote by $H(r)$ the set $\{a_1, ..., a_n\}$ of the head atoms, and by $B(r)$ the set of the body literals. In particular, $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r)$ (the *positive body*) is $\{b_1, \ldots, b_k\}$ and $B^-(r)$ (*the negative body*) is $\{b_{k+1}, \ldots, b_m\}$. A rule having precisely one head literal (i.e. $n = 1$) is called a *normal rule*. If the body is empty (i.e. $k = m = 0$), it is called a *fact*, and we usually omit the " :– " sign.

An *(integrity) constraint* is a rule without head literals (i.e. $n = 0$)

$$:\!-\ b_1, \cdots, b_k, \text{ not } b_{k+1}, \cdots, \text{ not } b_m. \tag{1.2}$$

A weak constraint wc is an expression of the form

$$:\sim\ b_1, \ldots, b_k, \text{ not } b_{k+1}, \ldots, \text{ not } b_m. \, [w : l] \tag{1.3}$$

where for $m \geq k \geq 0$, $b_1, \ldots, b_m$ are atoms, while w (the *weight*) and l (the *level*, or *layer*) are positive integer constants or variables. For convenience, w and/or l might be omitted and are set to 1 in this case.

The sets $B(wc)$, $B^+(wc)$, and $B^-(wc)$ of a weak constraint wc are defined in the same way as for integrity constraints.

A *disjunctive logic program* (often simply DLP program) $\mathcal{P}$ is a finite set of rules (possibly including integrity constraints) and weak constraints. $WC(\mathcal{P})$ denotes the set of weak constraints in $\mathcal{P}$, and $Rules(\mathcal{P})$ denotes the set of rules (including integrity constraints) in $\mathcal{P}$. A not-free program $\mathcal{P}$ (i.e., such that $\forall r \in \mathcal{P} : B^-(r) = \emptyset$) is called *positive*, and a v-free program $\mathcal{P}$ (i.e., such that $\forall r \in \mathcal{P} : |H(r)| \leq 1$) is called *normal logic program*. A program that does not contain weak constraints (i.e., such that $WC(\mathcal{P})$=$\emptyset$) is called *regular*.

A rule is *safe* if each variable in that rule also appears in at least one positive literal in the body of that rule. A program is safe, if each of its rules is safe, and in the following we will only consider safe programs.

A term (an atom, a rule, a program, etc.) is called *ground*, if no variable appears in it. A ground program is also called a *propositional* program.

1.1.2 Semantics

The semantics provided in this section extends the Answer Set Semantics of regular disjunctive logic programs, originally defined in [Gelfond and Lifschitz, 1991], to deal with weak constraints.

Let $\mathcal{P}$ be a disjunctive logic program. The Herbrand Universe of $\mathcal{P}$, denoted as $U_{\mathcal{P}}$, is the set of all constants appearing in $\mathcal{P}$. In case no constant appears in $\mathcal{P}$, an arbitrary constant ψ is added to $U_{\mathcal{P}}$. The Herbrand Base of $\mathcal{P}$, denoted as $B_{\mathcal{P}}$, is the set of all ground atoms constructible from the predicate symbols appearing in $\mathcal{P}$ and the constants of $U_{\mathcal{P}}$.

For any rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of $U_{\mathcal{P}}$. In a similar way, given a weak constraint w, $Ground(w)$ denotes the set of weak constraints obtained by applying all possible substitutions σ from the variables in w to elements of $U_{\mathcal{P}}$. For any program $\mathcal{P}$, the ground instantiation $Ground(\mathcal{P})$ is the set $GroundRules(\mathcal{P}) \cup GroundWC(\mathcal{P})$, where

$$GroundRules(\mathcal{P}) = \bigcup_{r \in Rules(\mathcal{P})} Ground(r)$$

and

$$GroundWC(\mathcal{P}) = \bigcup_{w \in WC(\mathcal{P})} Ground(w).$$

Note that for propositional programs, $\mathcal{P} = Ground(\mathcal{P})$ holds.

Answer Sets For every program $\mathcal{P}$, we define its answer sets using its ground instantiation $Ground(\mathcal{P})$ in three steps:

First we define the answer sets of positive regular disjunctive logic programs, then we give a reduction of disjunctive logic programs containing negation to positive ones and use it to define answer sets of arbitrary disjunctive logic programs. Finally, we specify the way how weak constraints affect the semantics, defining the semantics of general DLP programs.

An interpretation I is a set of ground atoms, i.e. $I \subseteq B_{\mathcal{P}}$ w.r.t. a program $\mathcal{P}$. An interpretation $X \subseteq B_{\mathcal{P}}$ is called *closed under* $\mathcal{P}$ (where $\mathcal{P}$ is a positive disjunctive logic program), if, for every $r \in Ground(\mathcal{P})$, $H(r) \cap X \neq \emptyset$ whenever $B(r) \subseteq X$. An interpretation $X \subseteq B_{\mathcal{P}}$ is an *answer set* for a positive disjunctive logic program $\mathcal{P}$, if it is minimal (under set inclusion) among all interpretations that are closed under $\mathcal{P}$.

Example 1.1 The positive program $\mathcal{P}_1 = \{a \,\mathrm{v}\, b \,\mathrm{v}\, c.\}$ has the answer sets $\{a\}$, $\{b\}$, and $\{c\}$. Its extension $\mathcal{P}_2 = \{a \,\mathrm{v}\, b \,\mathrm{v}\, c. \ ; \ :\!-\, a.\}$ has the answer sets $\{b\}$ and $\{c\}$. Finally, the positive program $\mathcal{P}_3 = \{a \,\mathrm{v}\, b \,\mathrm{v}\, c. \ ; \ :\!-\, a. \ ; \ b :\!-\, c. \ ; \ c :\!-\, b.\}$ has the single answer set $\{b, c\}$.

Definition 1.2 *[Gelfond and Lifschitz, 1991] The* GL-reduct *or* Gelfond-Lifschitz transform *of a ground program $\mathcal{P}$ w.r.t. a set $X \subseteq B_{\mathcal{P}}$ is the positive ground program $\mathcal{P}^X$, obtained from $\mathcal{P}$ by*

- *deleting all rules $r \in \mathcal{P}$ for which $B^-(r) \cap X \neq \emptyset$ holds;*
- *deleting the negative body from the remaining rules.*

Definition 1.3 *An* answer set *of a program $\mathcal{P}$ is a set $X \subseteq B_{\mathcal{P}}$ such that X is an answer set of $Ground(\mathcal{P})^X$.*

Example 1.4 Given the general program $\mathcal{P}_4 = \{a \,\mathrm{v}\, b \,\text{:-}\, c. \;\; ; \; b \,\text{:-}\, \mathrm{not}\; a, \mathrm{not}\; c. \;\; ; \; a \,\mathrm{v}\, c \,\text{:-}\, \mathrm{not}\; b.\}$ and $I = \{b\}$, the GL-reduct $\mathcal{P}_4^I$ is $\{a \,\mathrm{v}\, b \,\text{:-}\, c. \;\; ; \; b.\}$. It is easy to see that I is an answer set of $\mathcal{P}_4^I$, and for this reason it is also an answer set of $\mathcal{P}_4$.

Now consider $J = \{a\}$. The GL-reduct $\mathcal{P}_4^J$ is $\{a \,\mathrm{v}\, b \,\text{:-}\, c. \;\; ; \; a \,\mathrm{v}\, c.\}$ and it can be easily verified that J is an answer set of $\mathcal{P}_4^J$, so it is also an answer set of $\mathcal{P}_4$.

If, on the other hand, we take $K = \{c\}$, the GL-reduct $\mathcal{P}_4^K$ is equal to $\mathcal{P}_4^J$, but K is not an answer set of $\mathcal{P}_4^K$: for the rule $r : a \,\mathrm{v}\, b \,\text{:-}\, c$, $B(r) \subseteq K$ holds, but $H(r) \cap K \neq \emptyset$ does not. Indeed, it can be verified that I and J are the only answer sets of $\mathcal{P}_4$.

Recently, an alternative definition of reduct has been proposed in [Faber *et al.*, 2004]. This definition is a generalization and simplification of the original definition of the Gelfond-Lifschitz transform , but is fully equivalent to it for the definition of answer sets.

Definition 1.5 *Given a ground program $\mathcal{P}$ and an interpretation I, the* FLP-reduct *of $\mathcal{P}$ w.r.t. I is the subset $\mathcal{P}^I$ of $\mathcal{P}$, which is obtained from $\mathcal{P}$ by deleting rules in which a body literal is false w.r.t. I.*

Definition 1.6 *[Przymusinski, 1991; Gelfond and Lifschitz, 1991] Let I be an interpretation for a program $\mathcal{P}$. I is an* answer set *for $\mathcal{P}$ if $I \in \mathrm{MM}(\mathcal{P}^I)$ (i.e., I is a minimal model for the program $\mathcal{P}^I$).* □

The main difference between Definitions 1.3 and 1.6 is that the answer sets are defined directly on top of the notion of models of DLP programs, rather than transforming them to a positive program.

Theorem 1.7 *Given a DLP program $\mathcal{P}$ an interpretation I is an answer set of $\mathcal{P}$ according to FLP-reduct iff it is an answer set of $\mathcal{P}$ according to the standard definition via the classic GL- reduct (Definitions 1.2, 1.6).*

Example 1.8 *Given the general program $\mathcal{P}_1 = \{a \mathrm{v} b$:- c., b:-not a, not c., $a \mathrm{v} c$:-not $b.\}$ and $I = \{b\}$, the FLP-reduct $\mathcal{P}_1^I$ is $\{b$:- not a, not $c.\}$. It is easy to see that I is a minimal model of $\mathcal{P}_1^I$, and for this reason it is also an answer set of $\mathcal{P}_1$. Now consider $J = \{a\}$. The FLP-reduct $\mathcal{P}_1^J$ is $\{a \mathrm{v} c$:-not $b.\}$ and it can be easily verified that J is an answer set of $\mathcal{P}_1$. If, on the other hand, we take $K = \{c\}$, the FLP-reduct $\mathcal{P}_1^K$ is $\{a \mathrm{v} b$:-$c.$, $a \mathrm{v} c$:- not $b.\}$ and K is not an answer set of $\mathcal{P}_1^K$: the rule $a \mathrm{v} b$:-$c.$, is not true w.r.t K and hence K is not a model for $\mathcal{P}_1^K$. Indeed, it can be verified that I and J are the only answer sets of $\mathcal{P}_1$.*

Given a ground program $\mathcal{P}$ with weak constraints $WC(\mathcal{P})$, we are interested in the answer sets of $Rules(\mathcal{P})$ which minimize the sum of weights of the violated (unsatisfied) weak constraints in the highest priority level,[2] and among them those which minimize the sum of weights of the violated weak constraints in the next lower level, etc. Formally, this is expressed by an objective function $H^{\mathcal{P}}(A)$ for $\mathcal{P}$ and an answer set A as follows, using an auxiliary function $f_{\mathcal{P}}$ which maps leveled weights to weights without levels:

$$\begin{aligned} f_{\mathcal{P}}(1) &= 1, \\ f_{\mathcal{P}}(n) &= f_{\mathcal{P}}(n-1) \cdot |WC(\mathcal{P})| \cdot w_{max}^{\mathcal{P}} + 1, \quad n > 1, \\ H^{\mathcal{P}}(A) &= \textstyle\sum_{i=1}^{l_{max}^{\mathcal{P}}} (f_{\mathcal{P}}(i) \cdot \sum_{w \in N_i^{\mathcal{P}}(A)} weight(w)), \end{aligned}$$

where $w_{max}^{\mathcal{P}}$ and $l_{max}^{\mathcal{P}}$ denote the maximum weight and maximum level over the weak constraints in $\mathcal{P}$, respectively; $N_i^{\mathcal{P}}(A)$ denotes the set of the weak constraints in level i that are violated by A, and $weight(w)$ denotes the weight of the weak constraint w. Note that $|WC(\mathcal{P})| \cdot w_{max}^{\mathcal{P}} + 1$ is greater than the sum of all weights in the program, and therefore guaranteed to be greater than the sum of weights of any single level.

Intuitively, the function $f_{\mathcal{P}}$ handles priority levels. It guarantees that the violation of a single constraint of priority level i is more “expensive” then the violation of *all* weak constraints of the lower levels (i.e., all levels $< i$).

For a DLP program $\mathcal{P}$ (possibly with weak constraints), a set A is an *(optimal) answer set* of $\mathcal{P}$ if and only if (1) A is an answer set of $Rules(\mathcal{P})$ and (2) $H^{\mathcal{P}}(A)$ is minimal over all the answer sets of $Rules(\mathcal{P})$.

Example 1.9 Consider the following program $\mathcal{P}_{wc}$, which has three weak constraints:

[2]Higher values for weights and priority levels mark weak constraints of higher importance. E.g., the most important constraints are those having the highest weight among those with the highest priority level.

$a \,\mathtt{v}\, b.$
$b \,\mathtt{v}\, c.$
$d \,\mathtt{v}\, e$ `:-` $a, c.$
`:-` $d, e.$
$:\sim b.$ $[1:2]$
$:\sim a, e.$ $[4:1]$
$:\sim c, d.$ $[3:1]$

$Rules(\mathcal{P}_{wc})$ admits three answer sets: $A_1 = \{a, c, d\}$, $A_2 = \{a, c, e\}$, and $A_3 = \{b\}$. We have: $H^{\mathcal{P}_{wc}}(A_1) = 3$, $H^{\mathcal{P}_{wc}}(A_2) = 4$, $H^{\mathcal{P}_{wc}}(A_3) = 13$. Thus, the unique (optimal) answer set is $\{a, c, d\}$ with weight 3 in level 1 and weight 0 in level 2.

1.2 Computational Complexity

n this section, we describe the computational complexity of Disjunctive Logic Programming. We first provide some preliminaries on complexity theory. Then, describe relevant syntactic properties of disjunctive logic programs, which allow us to single out computationally simpler subclasses of the language. Finally, we define the main computational problems under consideration and illustrate their precise complexity.

1.2.1 Preliminaries

We assume here that the reader is familiar with the concepts of NP-completeness and complexity theory and provide only a very short reminder of the complexity classes of the Polynomial Hierarchy which are relevant to this chapter. For further details, the reader is referred to [Papadimitriou, 1994].

The classes Σ_k^P, Π_k^P, and Δ_k^P of the *Polynomial Hierarchy* (PH, cf. [Johnson, 1990]) are defined as follows:

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

$$\text{and for all } k \geq 1,\ \Delta_k^P = \mathrm{P}^{\Sigma_{k-1}^P}, \Sigma_k^P = \mathrm{NP}^{\Sigma_{k-1}^P}, \Pi_k^P = \text{co-}\Sigma_k^P,$$

where NP^C denotes the class of decision problems that are solvable in polynomial time on a nondeterministic Turing machine with an oracle for any decision problem π in the class C. In particular, $\mathrm{NP} = \Sigma_1^P$, co-NP $= \Pi_1^P$, and $\Delta_2^P = \mathrm{P}^{\mathrm{NP}}$.

The oracle replies to a query in unit time, and thus, roughly speaking, models a call to a subroutine for π that is evaluated in unit time.

Observe that for all $k \geq 1$,

$$\Sigma_k^P \subseteq \Delta_{k+1}^P \subseteq \Sigma_{k+1}^P \subseteq \text{PSPACE}$$

where each inclusion is widely conjectured to be strict. By the rightmost inclusion above, all these classes contain only problems that are solvable in polynomial space. They allow, however, a finer grained distinction among NP-hard problems that are in PSPACE.

1.2.2 Relevant Classes of Programs

In this section, we introduce syntactic classes of disjunctive logic programs with interesting properties. First we need the following:

Definition 1.10 Functions $|| \; || : B_{\mathcal{P}} \rightarrow \{0, 1, \ldots\}$ from the Herbrand Base $B_{\mathcal{P}}$ to finite ordinals are called *level mappings* of $\mathcal{P}$.

Level mappings give us a useful technique for describing various classes of programs.

Definition 1.11 A disjunctive logic program $\mathcal{P}$ is called *(locally) stratified* [Apt *et al.*, 1988; Przymusinski, 1988], 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 1.12 Consider the following two programs.

$$\begin{array}{llcll} \mathcal{P}_1: & p(a) \,\text{v}\, p(c) \text{ :- not } q(a). & \quad & \mathcal{P}_2: & p(a) \,\text{v}\, p(c) \text{ :- not } q(b). \\ & p(b) \text{ :- not } q(b). & & & q(b) \text{ :- not } p(a). \end{array}$$

It is easy to see that program $\mathcal{P}_1$ is stratified, while program $\mathcal{P}_2$ is not. A suitable level mapping for $\mathcal{P}_1$ is the following:

$$\begin{array}{lll} ||p(a)||_s = 2 & ||p(b)||_s = 2 & ||p(c)||_s = 2 \\ ||q(a)||_s = 1 & ||q(b)||_s = 1 & ||q(c)||_s = 1 \end{array}$$

As for $\mathcal{P}_2$, an admissible level mapping would need to satisfy $||p(a)||_s < ||q(b)||_s$ and $||q(b)||_s < ||p(a)||_s$, which is impossible.

Another interesting class of problems consists of head-cycle free programs.

Definition 1.13 A program $\mathcal{P}$ is called *head-cycle free (HCF)* [Ben-Eliyahu and Dechter, 1994], if there is a level mapping $||\ ||_h$ of $\mathcal{P}$ such that, for every rule r of $Ground(\mathcal{P})$,

1. For any $l \in B^+(r)$, and for any $l' \in H(r)$, $||l||_h \leq ||l'||_h$;
2. For any pair $l, l' \in H(r)$ $||l||_h \neq ||l'||_h$.

Example 1.14 Consider the following program $\mathcal{P}_3$.

$$\begin{array}{ll} \mathcal{P}_3: & a \vee b. \\ & a \coloneq b. \end{array}$$

It is easy to see that $\mathcal{P}_3$ is head-cycle free; an admissible level mapping for $\mathcal{P}_3$ is given by $||a||_h = 2$ and $||b||_h = 1$. Consider now the program

$$\mathcal{P}_4 = \mathcal{P}_3 \cup \{b \coloneq a.\}$$

$\mathcal{P}_4$ is not head-cycle free, since a and b should belong to the same level by Condition (1) of Definition 1.13, while they cannot by Condition (2) of that definition. Note, however, that $\mathcal{P}_4$ is stratified.

1.2.3 Main Problems Considered

Three important decision problems, corresponding to three different reasoning tasks, arise in the context of Disjunctive Logic Programming:

> **Brave Reasoning**. Given a program $\mathcal{P}$, and a ground atom A, decide whether A is true in some answer set of $\mathcal{P}$ (denoted $\mathcal{P} \models_b A$).
>
> **Cautious Reasoning**. Given a program $\mathcal{P}$, and a ground atom A, decide whether A is true in all answer sets of $\mathcal{P}$ (denoted $\mathcal{P} \models_c A$).
>
> **Answer Set Checking.** Given a program $\mathcal{P}$, and a set M of ground literals as input, decide whether M is an answer set of $\mathcal{P}$.

We study the complexity of these decision problems for ground (i.e., propositional) DLP programs; we shall address the case of non-ground programs at the end of this chapter.

An interesting issue is the impact of syntactic restrictions on the logic program $\mathcal{P}$. Starting from normal positive programs (without negation and disjunction), we consider the effect of allowing the (combined) use of the following constructs:

- stratified negation (not_s),
- arbitrary negation (not),
- head-cycle free disjunction ($\mathtt{v_h}$),
- arbitrary disjunction ($\mathtt{v}$),
- weak constraints (w).[3]

Given a set X of the above syntactic elements (with at most one negation and at most one disjunction symbol in X), we denote by DLP[X] the fragment of DLP where the elements in X are allowed. For instance, DLP[$\mathtt{v_h}, \mathrm{not}_s$] denotes the fragment allowing head-cycle free disjunction and stratified negation, but no weak constraints.

1.2.4 Complexity Results and Discussion

We report here, with the help of some tables, results proved in [Eiter *et al.*, 1997b; Gottlob, 1994; Buccafurri *et al.*, 2000; Eiter *et al.*, 1998; Eiter and Gottlob, 1995; Perri, 2004; Leone *et al.*, 2006].

It is worth that we consider the ground case, i.e., we assume that programs and, unless stated otherwise, also atoms, literals etc. are ground. Furthermore, for the sake of the presentations, we disregard integrity constraints in programs. However, this is not significant since the results in presence of these constructs are the same (see, e.g., [Buccafurri *et al.*, 2000]). Some remarks on the complexity and expressiveness of non-ground programs are then provided.

The complexity of Brave Reasoning and Cautious Reasoning from ground DLP programs are summarized in Table 1.1 and Table 1.2, respectively. In Table 1.3, we report the results on the complexity of Answer Set Checking.

The rows of the tables specify the form of disjunction allowed; in particular, $\{\}$ = no disjunction, $\{\mathtt{v_h}\}$ = head-cycle free disjunction, and $\{\mathtt{v}\}$ = unrestricted (possibly not head-cycle free) disjunction. The columns specify the support for negation and weak constraints. For instance, $\{w, \mathrm{not}_s\}$ denotes weak constraints and stratified negation. Each entry of the table provides the complexity of the

[3]Following [Buccafurri *et al.*, 2000], possible restrictions on the support of negation affect $Rules(\mathcal{P})$, that is, the rules (including the integrity constraints) of the program, while weak constraints, if allowed, can freely contain both positive and negative literals in any fragment of DLP we consider.

	$\{\}$	$\{\mathbf{w}\}$	$\{\mathrm{not_s}\}$	$\{\mathrm{not_s}, \mathbf{w}\}$	$\{\mathrm{not}\}$	$\{\mathrm{not}, \mathbf{w}\}$
$\{\}$	P	P	P	P	NP	Δ_2^P
$\{\mathtt{v_h}\}$	NP	Δ_2^P	NP	Δ_2^P	NP	Δ_2^P
$\{\mathtt{v}\}$	Σ_2^P	Δ_3^P	Σ_2^P	Δ_3^P	Σ_2^P	Δ_3^P

Table 1.1: The Complexity of Brave Reasoning in fragments of DLP

	$\{\}$	$\{\mathbf{w}\}$	$\{\mathrm{not_s}\}$	$\{\mathrm{not_s}, \mathbf{w}\}$	$\{\mathrm{not}\}$	$\{\mathrm{not}, \mathbf{w}\}$
$\{\}$	P	P	P	P	co-NP	Δ_2^P
$\{\mathtt{v_h}\}$	co-NP	Δ_2^P	co-NP	Δ_2^P	co-NP	Δ_2^P
$\{\mathtt{v}\}$	co-NP	Δ_3^P	Π_2^P	Δ_3^P	Π_2^P	Δ_3^P

Table 1.2: The Complexity of Cautious Reasoning in fragments of DLP

corresponding fragment of the language, in terms of a completeness result. For instance, $(\{\mathtt{v_h}\}, \{\mathrm{not}_s\})$ is the fragment allowing head-cycle free disjunction and stratified negation, but no weak constraints. The corresponding entry in Table 1.1, namely NP, expresses that brave reasoning for this fragment is NP-complete. The results reported in the tables represent completeness under polynomial time (and in fact LOGSPACE) reductions. All results have either been proved in [Perri, 2004] or emerge from [Eiter *et al.*, 1997b; Gottlob, 1994; Eiter *et al.*, 1998; Eiter and Gottlob, 1995; Buccafurri *et al.*, 2000]. Note that the presence of weights besides priority levels in weak constraints does not increase the complexity of the language, and thus the complexity results reported in [Buccafurri *et al.*, 2000] remain valid also for our more general language. Furthermore, not all complexity results in the quoted papers were explicitly stated for LOGSPACE reductions, but can be easily seen to hold from (suitably adapted) proofs.

Looking at Table 1.1, we see that limiting the form of disjunction and negation reduces the respective complexity. For disjunction-free programs, brave reasoning is polynomial on stratified negation, while it becomes NP-complete if we allow unrestricted (nonmonotonic) negation. Brave reasoning is NP-complete on head-cycle free programs even if no form of negation is allowed. The complexity jumps one level higher in the Polynomial Hierarchy, up to Σ_2^P-complexity, if full disjunction is allowed. Thus, disjunction seems to be harder than negation,

	$\{\}$	$\{\mathbf{w}\}$	$\{\mathrm{not_s}\}$	$\{\mathrm{not_s}, \mathbf{w}\}$	$\{\mathrm{not}\}$	$\{\mathrm{not}, \mathbf{w}\}$
$\{\}$	P	P	P	P	P	co-NP
$\{\mathtt{v_h}\}$	P	co-NP	P	co-NP	P	co-NP
$\{\mathtt{v}\}$	co-NP	Π_2^P	co-NP	Π_2^P	co-NP	Π_2^P

Table 1.3: The Complexity of Answer Set Checking in fragments of DLP

since the full complexity is reached already on positive programs, even without any kind of negation. Weak constraints are irrelevant, from the complexity viewpoint, if the program has at most one answer set (if there is no disjunction and negation is stratified). On programs with multiple answer sets, weak constraints increase the complexity of reasoning moderately, from NP and Σ_2^P to Δ_2^P and Δ_3^P, respectively.

Table 1.2 contains results for cautious reasoning. One would expect its complexity to be symmetric to the complexity of brave reasoning, that is, whenever the complexity of a fragment is C under brave reasoning, one expects its complexity to be co-C under cautious reasoning (recall that co-P $= \mathrm{P}$, co-$\Delta_2^P = \Delta_2^P$, co-$\Sigma_2^P = \Pi_2^P$, and co-$\Delta_3^P = \Delta_3^P$).

Surprisingly, there is one exception: while full disjunction raises the complexity of brave reasoning from NP to Σ_2^P, full disjunction alone is not sufficient to raise the complexity of cautious reasoning from co-NP to Π_2^P. Cautious reasoning remains in co-NP if default negation is disallowed. Intuitively, to disprove that an atom A is a cautious consequence of a program $\mathcal{P}$, it is sufficient to find *any model* M of $\mathcal{P}$ (which need not be an answer set or a minimal model) which does not contain A. For not-free programs, the existence of such a model guarantees the existence of a subset of M which is an answer set of $\mathcal{P}$ (and does not contain A).

The complexity results for Answer Set Checking, reported in Table 1.3, help us to understand the complexity of reasoning. Whenever Answer Set Checking for weak constraint-free programs is co-NP-complete for a fragment F, the complexity of brave reasoning jumps up to the second level of the Polynomial Hierarchy (Σ_2^P). In contrast, co-NP-completeness for Answer Set Checking involving weak constraints causes only a modest increase for brave reasoning, which stays within the same level (Δ_2^P). Indeed, brave reasoning on full DLP programs suffers from three sources of complexity:

(s_1) the exponential number of answer set "candidates",

(s_2) the difficulty of checking whether a candidate M is an answer set (the minimality of M can be disproved by an exponential number of subsets of M), and

(s_3) the difficulty of determining the optimality of the answer set w.r.t. the violation of the weak constraints.

Now, disjunction (unrestricted or even head-cycle free) or unrestricted negation preserve the existence of source (s_1), while source (s_2) exists only if full disjunction is allowed (see Table 1.3). Source (s_3) depends on the presence of weak constraints, but it is effective only in case of multiple answer sets (i.e., only if source (s_1) is present), otherwise it is irrelevant. As a consequence, e.g., the complexity of brave reasoning is the highest (Δ_3^P) on the fragments preserving all three sources of complexity (where both full disjunction and weak constraints are allowed). Eliminating weak constraints (source (s_3)) from the full language, decreases the complexity to Σ_2^P. The complexity goes down to the first level of PH if source (s_2) is eliminated, and is in the class Δ_2^P or NP depending on the presence or absence of weak constraints (source (s_3)). Finally, avoiding source (s_1) the complexity falls down to P, as (s_2) is automatically eliminated, and (s_3) becomes irrelevant.

We close this section with briefly addressing the complexity and expressiveness of non-ground programs. A non-ground program $\mathcal{P}$ can be reduced, by naive instantiation, to a ground instance of the problem. The complexity of this ground instantiation is as described above. In the general case, where $\mathcal{P}$ is given in the input, the size of the grounding $Ground(\mathcal{P})$ is single exponential in the size of $\mathcal{P}$. Informally, the complexity of Brave Reasoning and Cautious Reasoning increases accordingly by one exponential, from P to EXPTIME, NP to NEXPTIME, Δ_2^P to EXPTIME$^{\mathrm{NP}}$, Σ_2^P to NEXPTIME$^{\mathrm{NP}}$, etc. For disjunctive programs and certain fragments of DLP, complexity results in the non-ground case have been derived e.g. in [Eiter *et al.*, 1997b; 1998]. For the other fragments, the results can be derived using complexity upgrading techniques [Eiter *et al.*, 1997b; Gottlob *et al.*, 1999]. Answer Set Checking, however, increases exponentially up to co-NEXPTIME$^{\mathrm{NP}}$ only in the presence of weak constraints, while it stays in PH if no weak constraints occur. The reason is that in the latter case, the conditions of an answer set can be checked using small guesses, and no alternative (perhaps exponentially larger) answer set candidates need to be considered.

1.3 Knowledge Representation

In this section, we illustrate the usage of Disjunctive Logic Programming for knowledge representation and reasoning. We first present a new programming methodology, which allows us to encode search problems in a simple and highly declarative fashion; even optimization problems of complexity up to Δ_3^P can be declaratively encoded using this methodology. Then, we illustrate this methodology on a number of computationally hard problems.

1.3.1 The GCO Declarative Programming Methodology

Disjunctive Logic Programming can be used to encode problems in a highly declarative fashion, following a Guess/Check/Optimize (**GCO**) paradigm, which is an extension and refinement of the "Guess&Check" methodology in [Eiter *et al.*, 2000]. In this section, we will first describe the **GCO** technique and we will then illustrate how to apply it on a number of examples. Many problems, also problems of comparatively high computational complexity (Σ_2^P-complete and Δ_3^P-complete problems), can be solved in a natural manner by using this declarative programming technique. The power of disjunctive rules allows for expressing problems which are more complex than NP, and the (optional) separation of a fixed, non-ground program from an input database allows to do so in a uniform way over varying instances.

Given a set $\mathcal{F}_I$ of facts that specify an instance I of some problem $\mathbf{P}$, a **GCO** program $\mathcal{P}$ for $\mathbf{P}$ consists of the following three main parts:

Guessing Part The guessing part $\mathcal{G} \subseteq \mathcal{P}$ of the program defines the search space, such that answer sets of $\mathcal{G} \cup \mathcal{F}_I$ represent "solution candidates" for I.

Checking Part The (optional) checking part $\mathcal{C} \subseteq \mathcal{P}$ of the program filters the solution candidates in such a way that the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ represent the admissible solutions for the problem instance I.

Optimization Part The (optional) optimization part $\mathcal{O} \subseteq \mathcal{P}$ of the program allows to express a quantitative cost evaluation of solutions by using weak constraints. It implicitly defines an objective function $f : \mathcal{AS}(\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I) \rightarrow \mathbb{N}$ mapping the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ to natural numbers. The semantics of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I \cup \mathcal{O}$ optimizes f by filtering those answer sets having the minimum value; this way, the optimal (least cost) solutions are computed.

Without imposing restrictions on which rules $\mathcal{G}$ and $\mathcal{C}$ may contain, in the extremal case we might set $\mathcal{G}$ to the full program and let $\mathcal{C}$ be empty, i.e., checking is completely integrated into the guessing part such that solution candidates are always solutions. Also, in general, the generation of the search space may be guarded by some rules, and such rules might be considered more appropriately placed in the guessing part than in the checking part. We do not pursue this issue further here, and thus also refrain from giving a formal definition of how to separate a program into a guessing and a checking part.

In general, both $\mathcal{G}$ and $\mathcal{C}$ may be arbitrary collections of rules (and, for the optimization part, weak constraints), and it depends on the complexity of the problem at hand which kinds of rules are needed to realize these parts (in particular, the checking part).

Problems in NP and Δ_2^P For problems with complexity in NP or, in case of optimization problems, Δ_2^P, often a natural **GCO** program can be designed with the three parts clearly separated into the following simple layered structure:

- The guessing part $\mathcal{G}$ consists of disjunctive rules that "guess" a solution candidate S.
- The checking part $\mathcal{C}$ consists of integrity constraints that check the admissibility of S.
- The optimization part $\mathcal{O}$ consists of weak constraints.

Each layer may have further auxiliary predicates, defined by normal stratified rules (see Section 1.2.2 for a definition of stratification), for local computations.

The disjunctive rules define the search space in which rule applications are branching points, while the integrity constraints prune illegal branches. The weak constraints in $\mathcal{O}$ induce a modular ordering on the answer sets, allowing the user to specify the best solutions according to an optimization function f.

Problems beyond Δ_2^P For problems which are beyond Δ_2^P, and in particular for Σ_2^P-complete problems, the layered program schema above no longer applies. If $\mathcal{G}$ has complexity in NP, which is the case if disjunction is just used for making the guess and the layer is head-cycle free [Ben-Eliyahu and Dechter, 1994], then an answer set A of $\mathcal{G} \cup \mathcal{F}_I$ can be guessed in polynomial time, i.e., nondeterministically created with a polynomial number of steps. Hence, it can be shown easily that computing an answer set of the whole program, $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I \cup \mathcal{O}$, is feasible in polynomial time with an NP oracle. Thus, applicability of the same schema to Σ_2^P-hard problems would imply $\Sigma_2^P \subseteq \Delta_2^P$, which is widely believed to be false.

Until now we tacitly assumed an intuitive layering of the program parts, such that the checking part $\mathcal{C}$ has no "influence" or "feedback" on the guessing part $\mathcal{G}$, in terms of literals which are derived in $\mathcal{C}$ and invalidate the application of rules in $\mathcal{G}$, or make further rules in $\mathcal{G}$ applicable (and thus change the guess). This can be formalized in terms of a "potentially uses" relation [Eiter *et al.*, 1997b] or a "splitting set" condition [Lifschitz and Turner, 1994]. Complexity-wise, this can be relaxed to the property that the union of the program parts is head-cycle free.

In summary, if a program solves a Σ_2^P-complete problem and has guessing and checking parts $\mathcal{G}$ and $\mathcal{C}$, respectively, with complexities below Σ_2^P, then $\mathcal{C}$ must either contain disjunctive rules or interfere with $\mathcal{G}$ (and in particular head-cycles must be present in $\mathcal{G} \cup \mathcal{C}$).

We close this section with remarking that the **GCO** programming methodology has also positive implications from the Software Engineering viewpoint. Indeed, the modular program structure in **GCO** allows for developing programs incrementally, which is helpful to simplify testing and debugging. One can start by writing the guessing part $\mathcal{G}$ and testing that $\mathcal{G} \cup \mathcal{F}_I$ correctly defines the search space. Then, one adds the checking part and verifies that the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ encode the admissible solutions. Finally, one tests that $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I \cup \mathcal{O}$ generates the optimal solutions of the problem at hand.

1.3.2 Applications of the GCO Programming Technique

In this section, we illustrate the declarative programming methodology described in Section 1.3.1 by showing its application on a number of concrete examples.

Exams Scheduling Let us start by a simple scheduling problem. Here we have to schedule the exams for several university courses in three time slots t_1, t_2, and t_3 at the end of the semester. In other words, each course should be assigned exactly one of these three time slots. Specific instances I of this problem are provided by sets $\mathcal{F}_I$ of facts specifying the exams to be scheduled. The predicate $exam$ has four arguments representing, respectively, the *identifier* of the exam, the *professor* who is responsible for the exam, the *curriculum* to which the exam belongs, and the *year* in which the exam has to be given in the curriculum.

Several exams can be assigned to the same time slot (the number of available rooms is sufficiently high), but the scheduling has to respect the following specifications:

$S1$ Two exams given by the same professor cannot run in parallel, i.e., in the same time slot.

$S2$ Exams of the same curriculum should be assigned to different time slots, if possible. If $S2$ is unsatisfiable for all exams of a curriculum C, one should:

($S2_1$) first of all, minimize the overlap between exams of the same year of C,

($S2_2$) then, minimize the overlap between exams of different years of C.

This problem can be encoded in DLP by the following **GCO** program $\mathcal{P}_{sch}$:

$$\left.\begin{array}{l} assign(Id,t_1) \mathrm{\ v\ } assign(Id,t_2) \mathrm{\ v\ } assign(Id,t_3) \text{ :- } \\ \quad exam(Id,P,C,Y). \end{array}\right\} \textbf{Guess}$$

$$\left.\begin{array}{l} \text{:- } assign(Id,T), assign(Id',T), \\ \quad Id <> Id', exam(Id,P,C,Y), exam(Id',P,C',Y'). \end{array}\right\} \textbf{Check}$$

$$\left.\begin{array}{l} :\sim assign(Id,T), assign(Id',T) \\ \quad exam(Id,P,C,Y), exam(Id',P',C,Y), Id <> Id'. \ [:2] \\ :\sim assign(Id,T), assign(Id',T) \\ \quad exam(Id,P,C,Y), exam(Id',P',C,Y'), Y <> Y'. \ [:1] \end{array}\right\} \textbf{Optimize}$$

The guessing part $\mathcal{G}$ has a single disjunctive rule defining the search space. It is evident that the answer sets of $\mathcal{G} \cup \mathcal{F}_I$ are the possible assignments of exams to time slots.

The checking part $\mathcal{C}$ consists of one integrity constraint, discarding the assignments of the same time slot to two exams of the same professor. The answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ correspond precisely to the admissible solutions, that is, to all assignments which satisfy the constraint $S1$.

Finally, the optimizing part $\mathcal{O}$ consists of two weak constraints with different priorities. Both weak constraints state that exams of the same curriculum should *possibly not* be assigned to the same time slot. However, the first one, which has higher priority (level 2), states this desire for the exams of the curriculum of the same year, while the second one, which has lower priority (level 1) states it for the exams of the curriculum of different years. The semantics of weak constraints, as given in Section 1.1.2, implies that $\mathcal{O}$ captures precisely the constraints $S2$ of the scheduling problem specification. Thus, the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I \cup \mathcal{O}$ correspond precisely to the desired schedules.

Hamiltonian Path Let us now consider a classical NP-complete problem in graph theory, namely Hamiltonian Path.

Definition 1.15 (HAMPATH) *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?*

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 **GCO** program $\mathcal{P}_{hp}$ solves the problem HAMPATH (no optimization part is needed here):

$$\left.\begin{array}{ll}
inPath(X,Y) \mathrel{\mathrm{v}} outPath(X,Y) \text{ :- } start(X), arc(X,Y). & \\
inPath(X,Y) \mathrel{\mathrm{v}} outPath(X,Y) \text{ :- } reached(X), arc(X,Y). & \\
reached(X) \text{ :- } inPath(Y,X). & \textbf{(aux.)}
\end{array}\right\} \textbf{Guess}$$

$$\left.\begin{array}{l}
\text{:- } inPath(X,Y), inPath(X,Y1), Y <> Y1. \\
\text{:- } inPath(X,Y), inPath(X1,Y), X <> X1. \\
\text{:- } node(X), \text{not } reached(X), \text{not } start(X).
\end{array}\right\} \textbf{Check}$$

The two disjunctive rules 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, an auxiliary predicate $reached$ is used, which is associated with the guessed predicate $inPath$ using the last rule. Note that $reached$ is completely determined by the guess for $inPath$, and no further guessing is needed.

In turn, through the second rule, the predicate $reached$ influences the guess of $inPath$, which is made somehow inductively: Initially, a guess on an arc leaving the starting node is made by the first rule, followed by repeated guesses of arcs leaving from reached nodes by the second rule, until all reached nodes have been handled.

In the checking part, the first two constraints 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. A less sophisticated encoding can be obtained by replacing the guessing part with the single rule

$$inPath(X, Y) \lor outPath(X, Y) \coloneq arc(X, Y).$$

that guesses for each arc whether it is in the path and by defining the predicate $reached$ in the checking part by rules

$$\begin{array}{l} reached(X) \coloneq start(X). \\ reached(X) \coloneq reached(Y), inPath(Y, X). \end{array}$$

However, this encoding is less preferable from a computational point of view, because it leads to a larger search space.

It is easy to see that any set of arcs S which satisfies all three constraints must contain the arcs of a path $v_0, v_1, \ldots, v_k$ in G that starts at node $v_0 = a$, and passes through distinct nodes until no further node is left, or it arrives at the starting node a again. In the latter case, this means that the path is in fact a Hamiltonian Cycle (from which a Hamiltonian path can be immediately computed, by dropping the last arc).

Thus, given a set of facts $\mathcal{F}$ for $node$, arc, and $start$, the program $\mathcal{P}_{hp} \cup \mathcal{F}$ has an answer set if and only if the corresponding graph has a Hamiltonian Path. The above program correctly encodes the decision problem of deciding whether a given graph admits a Hamiltonian Path or not.

This encoding is very flexible, and can be easily adapted to solve the *search problems* Hamiltonian Path and Hamiltonian Cycle (where the result has to be a tour, i.e., a closed path). If we want to be sure that the computed result is an *open* path (i.e., it is not a cycle), we can easily impose openness by adding a further constraint $\coloneq start(Y), inPath(\_, Y).$ to the program (like in Prolog, the symbol '_' stands for an anonymous variable whose value is of no interest). Then, the set S of selected arcs in any answer set of $\mathcal{P}_{hp} \cup \mathcal{F}$ constitutes a Hamiltonian Path starting at a. If, on the other hand, we want to compute the Hamiltonian cycles, then we just have to strip off the literal $\mathrm{not}\ start(X)$ from the last constraint of the program.

Traveling Salesperson The Traveling Salesperson Problem (TSP) is a well-known optimization problem, widely studied in Operation Research.

Definition 1.16 (TSP) *Given a weighted directed graph $G = (V, E, C)$ and a node $a \in V$ of this graph, find a minimum-cost cycle (closed path) in G starting at a and passing through each node in V exactly once.*

It is well-known that finding an optimal solution to the Traveling Salesperson Problem (TSP) is intractable. Computing an optimal tour is both NP-hard and co-NP-hard. In fact, in [Papadimitriou, 1984] it was shown that deciding whether the cost of an optimal tour is an even number is Δ_2^P-complete.

A DLP encoding for the Traveling Salesperson Problem (TSP) can be easily obtained from an encoding of Hamiltonian Cycle by adding optimization: each arc in the graph carries a weight, and a tour with minimum total weight is selected.

Suppose again that the graph G is specified by predicates $node$ (unary) and arc (ternary), and that the starting node is specified by the predicate $start$ (unary).

We first modify the HAMPATH encoding $\mathcal{P}_{hp}$ in Section 1.3.2 to compute Hamiltonian Cycles, by stripping off literal $\mathrm{not}\ start(X)$ from the last constraint of the program, as explained above. We then add an optimization part consisting of a single weak constraint

$$:\sim\ inPath(X,Y,C).\ [C:1]$$

which states the preference to avoid taking arcs with high cost in the path, and has the effect of selecting those answer sets for which the total cost of arcs selected by $inPath$ is the minimum.

The full **GCO** program $\mathcal{P}_{tsp}$ solving the TSP problem is thus as follows:

$$\left.\begin{array}{l}inPath(X,Y,C) \vee outPath(X,Y,C) :\!-\ start(X), arc(X,Y,C).\\ inPath(X,Y,C) \vee outPath(X,Y,C) :\!-\ reached(X), arc(X,Y,C).\\ reached(X) :\!-\ inPath(Y,X,C). \qquad\qquad\qquad\qquad \textbf{(aux.)}\end{array}\right\}\ \textbf{Guess}$$

$$\left.\begin{array}{l}:\!-\ inPath(X,Y,\_), inPath(X,Y1,\_), Y <> Y1.\\ :\!-\ inPath(X,Y,\_), inPath(X1,Y,\_), X <> X1.\\ :\!-\ node(X), \mathrm{not}\ reached(X).\end{array}\right\}\ \textbf{Check}$$

$$\left.:\sim\ inPath(X,Y,C).\ [C:1]\right\}\ \textbf{Optimize}$$

Given a set of facts $\mathcal{F}$ for $node$, arc, and $start$ which specifies the input instance, it is easy to see that the (optimal) answer sets of $\mathcal{P}_{tsp} \cup \mathcal{F}$ are in a one-to-one correspondence with the optimal tours of the input graph.

Winners in Combinatorial Auctions The weak constraints, specifying the Optimization module of a **GCO** program, permit to naturally express optimization criteria based on minimization. In this example, we show how also maximization can be expressed in the optimization part, by a suitable use of weak constraints.

A combinatorial auction has a set $\mathcal{O}$ of objects for sale and a set $\mathcal{B}$ of bidders. Bidders may offer bids on a set of items, in terms of a price they are willing to pay for them. For example, bidder b_1 may bid \$500 for items a, b, c together and \$200 for item a alone. Formally, a *bid* is a triple of the form (b, X, p) where $b \in \mathcal{B}$, $X \subseteq \mathcal{O}$, and $p > 0$ is a price. The auctioneer receives a set **bids** of bids. Without loss of generality, we may assume that **bids** does not contain two different triples (b_1, X, p_1) and (b_2, X, p_2) (i.e., with the same set X of items). This is because if two bids are received for the same set X of objects, the one with the lower price can be eliminated.[4]

The task of the auctioneer is to determine which bids to accept. Given **bids**, a *potential winner* is a subset $\mathbf{win} \subseteq \mathbf{bids}$ such that

$$(b_1, X_1, p_1), (b_2, X_2, p_2) \in \mathbf{win} \rightarrow X_1 \cap X_2 = \emptyset.$$

Each potential winner set represents a possible scenario, i.e., a possible outcome of the auction.

Potential winners may be of very different quality according to the auctioneer's point of view. Indeed, she clearly wants to maximize her revenue, and thus to determine a potential winner **win** maximizing the revenue $R(\mathbf{win})$, that is, maximizing the following function:

$$R(\mathbf{win}) = \sum_{(b,X,p) \in \mathbf{win}} p.$$

The *winner determination problem* is to find a potential winner $\mathbf{win}$ such that $R(\mathbf{win})$ is maximum.

We construct a **GCO** program $\mathcal{P}_{\mathbf{bids}}$ solving this problem as follows. Suppose that two input predicates *bids* and *requires* encode the input data about the bids $\mathbf{bids}$ received by the auctioneer. In particular, for each bid $(b, x, p) \in \mathbf{bids}$, $\mathcal{F}$ contains the fact $bids(b, x, p).$, and the facts describing the set of items x required by the bidder b, namely, a fact $requires(x, i).$, for each item $i \in x$.

The predicate *accept* encodes the potential winner set of bids chosen by the auctioneer. The rules of program $\mathcal{P}_{\mathbf{bids}}$ are:

$$\left.\begin{array}{l} accept(B, X) \vee reject(B, X) \text{ :- } bids(B, X, P). \end{array}\right\} \textbf{Guess}$$

$$\left.\begin{array}{l} \text{:- } accept(B_1, X_1), accept(B_2, X_2), \\ \quad X_1 <> X_2, requires(X_1, I), requires(X_2, I). \end{array}\right\} \textbf{Check}$$

[4]It is still possible that there are two bids of the form $(b_1, X, p_1), (b_2, X, p_2)$ with $b_1 \neq b_2$ and $p_1 = p_2$. In such a case most auctioneers eliminate one of the two bids using some pre-announced protocol – e.g. the bid received at a later time may be discarded.

$$:\sim reject(B,X), bids(B,X,P).\ [P:1] \quad \Big\}\ \textbf{Optimize}$$

It is easy to see that the answer sets of $Rules(\mathcal{P}_{\mathbf{bids}}) \cup \mathcal{F}$ correspond exactly to the potential winners of the auction. Then, the weak constraint $:\sim reject(B,X)$, $bids(B,X,P).\ [P:1]$ minimizes the sum of the prices of the rejected bids, and it therefore maximizes the revenue, since the maximum of $R(\mathbf{win})$ corresponds precisely to the minimum of

$$\hat{R}(\mathbf{win}) = \Big(\sum_{(b,X,p)\in\mathbf{bids}} p\Big) - R(\mathbf{win}).$$

Thus, the optimal answer sets of $\mathcal{P}_{\mathbf{bids}} \cup \mathcal{F}$ correspond precisely to the optimal potential winners.

Ramsey Numbers In the previous examples, we have seen how a search problem can be encoded in a DLP program whose answer sets correspond to the problem solutions. We next see another use of the **GCO** programming technique. We build a DLP program whose answer sets witness that a property does not hold, i.e., the property at hand holds if and only if the DLP program has no answer set. Such a programming scheme is useful to prove the validity of co-NP or Π_2^P properties. We next apply the above programming scheme to a well-known problem of number and graph theory.

Definition 1.17 (RAMSEY) *The Ramsey number $R(k,m)$ is the least integer n such that, no matter how we color the arcs of the complete undirected graph (clique) with n nodes 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).*

Ramsey numbers exist for all pairs of positive integers k and m [Radziszowski, 1994]. We next show a program $\mathcal{P}_{ramsey}$ that allows us to decide whether a given integer n is <u>not</u> the Ramsey Number $R(3,4)$. By varying the input number n, we can determine $R(3,4)$, as described below. Let $\mathcal{F}$ be the collection of facts for input predicates *node* and *arc* encoding a complete graph with n nodes. $\mathcal{P}_{ramsey}$ is the following **GCO** program:

$$blue(X,Y) \vee red(X,Y) :- arc(X,Y). \quad \Big\}\ \textbf{Guess}$$

$$\left.\begin{array}{l} :- red(X,Y),\ red(X,Z),\ red(Y,Z). \\ :- blue(X,Y),\ blue(X,Z),\ blue(Y,Z), \\ \quad blue(X,W),\ blue(Y,W),\ blue(Z,W). \end{array}\right\}\ \textbf{Check}$$

Intuitively, the disjunctive rule guesses a color for each edge. The first constraint eliminates the colorings containing a red clique (i.e., a complete graph) with 3 nodes, and the second constraint eliminates the colorings containing a blue clique with 4 nodes. The program $\mathcal{P}_{ramsey} \cup \mathcal{F}$ has an answer set if and only if there is a coloring of the edges of the complete graph on n nodes containing no red clique of size 3 and no blue clique of size 4. Thus, if there is an answer set for a particular n, then n is <u>not</u> $R(3, 4)$, that is, $n < R(3, 4)$. On the other hand, if $\mathcal{P}_{ramsey} \cup \mathcal{F}$ 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 $R(3, 4)$.

The problems considered so far are at the first level of the Polynomial Hierarchy. We next show that also problems located at the second level of the Polynomial Hierarchy can be encoded by the **GCO** technique.

Quantified Boolean Formulas (2QBF) The first problem at the second level of the Polynomial Hierarchy which we consider is the canonical Σ_2^P-complete problem 2QBF [Papadimitriou, 1994]. Here, we have to decide whether a quantified Boolean formula (QBF) of the shape $\Phi = \exists X \forall Y \phi$, where X and Y are disjoint sets of propositional variables and $\phi = C_1 \vee \ldots \vee C_k$ is a 3DNF formula over $X \cup Y$, evaluates to true. Moreover, in this case, we may want to have a witnessing assignment σ to the variables X, i.e., an assignment σ such that $\phi[X/\sigma(X)]$ is a tautology, where $X/\sigma(X)$ denotes the substitution of X by $\sigma(X)$.[5] This naturally leads to a Guess & Check disjunctive logic program, in which the witness assignment σ is guessed by some rules, and the rest of the program is devoted to checking whether $\phi[X/\sigma(X)]$ is a tautology.

Our encoding is a variant of the reduction of 2QBF into a propositional DLP in [Eiter and Gottlob, 1995]. Here, a QBF Φ as above is encoded as a set of facts F_Φ, which is evaluated by a fixed program $\mathcal{P}_{2QBF}$. In detail, F_Φ contains the following facts:

- $exists(v)$, for each existential variable $v \in X$;
- $forall(v)$, for each universal variable $v \in Y$; and
- $term(p_1, p_2, p_3, q_1, q_2, q_3)$, for each disjunct $l_1 \wedge l_2 \wedge l_3$ in ϕ, where (i) if l_i is a positive atom v_i, then $p_i = v_i$, otherwise p_i= "$true$", and (ii) if l_i is a negated atom $\neg v_i$, then $q_i = v_i$, otherwise q_i="$false$". For example, $term(x_1, true, y_4, false, y_2, false)$, encodes the term $x_1 \wedge \neg y_2 \wedge y_4$.

[5]Note that such a witness does no exist for universally quantified formulas of shape $\forall X \exists Y \phi$.

The program $\mathcal{P}_{2QBF}$ is then

$$\left.\begin{array}{rl} & t(true). \quad f(false). \\ & t(X) \,\mathrm{v}\, f(X) \text{ :- } exists(X). \end{array}\right\} \textbf{Guess}$$

$$\left.\begin{array}{rl} t(Y) \,\mathrm{v}\, f(Y) & \text{:- } forall(Y). \\ w & \text{:- } term(X,Y,Z,Na,Nb,Nc), \\ & \quad t(X), t(Y), t(Z), f(Na), f(Nb), f(Nc). \\ t(Y) & \text{:- } w, forall(Y). \\ f(Y) & \text{:- } w, forall(Y). \\ & \text{:- not } w. \end{array}\right\} \textbf{Check}$$

The guessing part "initializes" the logical constants "$true$" and "$false$" and chooses a witnessing assignment σ to the variables in X, which leads to an answer set M_G for this part. The more tricky checking part then tests whether $\phi[X/\sigma(X)]$ is a tautology, using a saturation technique [Eiter and Gottlob, 1995]: The constraint :- not w. enforces that w must be true in any answer set of the program; the preceding two rules imply that such an answer set M contains both $t(y)$ and $f(y)$ for every $y \in Y$. Hence, M has a unique extension with respect to w and all $t(y)$ and $f(y)$ where $y \in Y$. By the minimality of answer sets, an extension of M_G to the (uniquely determined) answer set M of the whole program exists, if and only if for each possible assignment μ to the variables in Y, effected by the disjunctive rule in the checking part, the atom w is derived. The latter holds iff there is some disjunct in $\phi[X/\sigma(X), Y/\mu(Y)]$ which is true. Hence, M is an answer set iff the formula $\phi[X/\sigma(X)]$ is a tautology. In summary, we obtain that Φ is a Yes-instance, i.e., it evaluates to true, if and only if $\mathcal{P}_{2QBF} \cup F_\Phi$ has some answer set. Moreover, the answer sets of $\mathcal{P}_{2QBF} \cup F_\Phi$ are in one-to-one correspondence with the witnesses σ for the truth of Φ.

Since 2QBF is Σ_2^P-complete, as discussed in Section 1.3.1 the use of disjunction in the checking part is not accidental but necessary: the guessing and checking parts are layered hierarchically (and Splitting Sets [Lifschitz and Turner, 1994] do exist).

Strategic Companies A problem located at the second level of the Polynomial Hierarchy is the following, which is known under the name *Strategic Companies* [Cadoli *et al.*, 1997].

Definition 1.18 (STRATCOMP) *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 $i = 1, \dots, m$ 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 Σ_2^P-hard in general [Cadoli *et al.*, 1997]; reformulated as a decision problem ("Given a particular company c in the input, is c strategic?"), it is Σ_2^P-complete. To our knowledge, it is one of the rare KR problems from the business domain of this complexity that have been considered so far.

In the following, we adopt the setting from [Cadoli *et al.*, 1997] where each product is produced by at most four companies (for each $g \in G$ $|\{c_i \mid g \in G_i\}| \leq 4$) and each company is jointly controlled by at most four other companies, i.e. $|O_i| \leq 4$ for $i = 1, \dots, m$ (in this case, the problem is still Σ_2^P-hard). Assume that for a given instance of STRATCOMP, $\mathcal{F}$ contains the following facts:

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

We next present a program $\mathcal{P}_{strat}$, which solves this hard problem elegantly by only two rules:

$$r_{s1}: \quad strat(W) \vee strat(X) \vee strat(Y) \vee strat(Z) \text{ :- } prod\_by(W, X, Y, Z). \quad \} \textbf{ Guess}$$

$$r_{s2}: \quad \left.\begin{array}{l} strat(W) \text{ :- } contr\_by(W, X, Y, Z, T), strat(X), \\ \qquad strat(Y), strat(Z), strat(Z). \end{array}\right\} \textbf{ Check}$$

Here $strat(X)$ means that company X is a strategic company. The guessing part $\mathcal{G}$ of the program consists of the disjunctive rule r_{s1}, and the checking part $\mathcal{C}$ consists of the normal rule r_{s2}. The program $\mathcal{P}_{strat}$ is surprisingly succinct, given that STRATCOMP is a hard (Σ_2^P-hard) problem. To overcome the difficulty of the encoding, coming from the intrinsic high complexity of the STRATCOMP problem, we next explain this encoding more in-depth, compared with the previous **GCO** encodings.

The program $\mathcal{P}_{strat}$ exploits the minimization which is inherent to 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_{s1} intuitively selects one of the companies c_1, c_2, c_3 and c_4 that produce some item g, which is described by $prod\_by(g, c_1, c_2, c_3, c_4)$. If there were no company control information, minimality of answer sets would naturally ensure that the answer sets of $\mathcal{F} \cup \{r_{s1}\}$ correspond to the strategic sets; no further checking would be needed. However, in case control information is available, the rule r_{s2} 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 $\mathcal{P}_{strat} \cup \mathcal{F}$ correspond one-to-one to the strategic sets of the holding described in $\mathcal{F}$; a company c is thus strategic iff $strat(c)$ is in some answer set of $\mathcal{P}_{strat} \cup \mathcal{F}$.

An important note here is that the checking "constraint" r_{s2} interferes with the guessing rule r_{s1}: applying r_{s2} may "spoil" the minimal answer set generated by r_{s1}. For example, suppose the guessing part gives rise to a ground rule r_{sg1}

$$strat(c1) \ \texttt{v} \ strat(c2) \, \texttt{v} \ strat(c3) \ \texttt{v} \ strat(c4) \mathrel{:-} prod\_by(g, c1, c2, c3, c4).$$

and the fact $prod\_by(g, c1, c2, c3, c4)$ is given in $\mathcal{F}$. 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_{s2} is applied which derives $strat(c2)$, then the application of the rule r_{sg1} to derive $strat(c1)$ is invalidated, as the minimality of answer sets implies that $strat(c1)$ cannot be derived from the rule r_{sg1}, if another atom in its head is true.

By the complexity considerations in Subsection 1.3.1, such interference is needed to solve STRATCOMP in the above way (without disjunctive rules in the Check part), since deciding whether a particular company is strategic is Σ_2^P-

complete. If $\mathcal{P}_{strat}$ is rewritten to eliminate such interference and layer the parts hierarchically, then further disjunctive rules must be added. An encoding which expresses the strategic sets in the generic **GCO**-paradigm with clearly separated guessing and checking parts is given in [Eiter *et al.*, 2000].

Note that, the program above cannot be replaced by a simple normal (non-disjunctive) program. Intuitively, this is due to the fact that disjunction in the head of rules is not exclusive, while at the same time answer sets are subset-minimal. Using techniques like the ones in [Eiter *et al.*, 2003a], $\mathcal{P}_{strat}$ can be extended to support an arbitrary number of producers per product and controlling companies per company, respectively.

Preferred Strategic Companies

Let us consider an extension of Strategic Companies which also deals with preferences. Suppose that the president of the holding desires, in case of options given by multiple strategic sets, to discard those where certain companies are sold or kept, respectively, by expressing preferences among possible solutions. For example, the president might give highest preference to discard solutions where company a is sold; next important to him is to avoid selling company b while keeping c, and of equal importance to avoid selling company d, and so on.

In presence of such preferences, the STRATCOMP problem becomes slightly harder, as its complexity increases from Σ_2^P to Δ_3^P. Let us assume that the president's preferences are represented by a single predicate $avoid(c_{sell}, c_{keep}, pr)$, which intuitively states that selling c_{sell} while keeping c_{keep} should be avoided with priority pr; in the above example, the preferences would be $avoid(a, c_\top, top)$, $avoid(b, c, top{-}1)$, $avoid(d, c_\top, top{-}1), \ldots$, where $c_\top$ is a dummy company which belongs to every strategic set, and top is the highest priority number. Then, we can easily represent this more complicated problem, by adding the following weak constraint to the original encoding for STRATCOMP:

$$:\sim avoid(Sell, Keep, Priority), \text{not } strat(Sell), strat(Keep).\ [: Priority]$$

The (optimal) answer sets of the resulting program then correspond to the solutions of the above problem.

Chapter 2

The Disjunctive Logic Programming System DLV

In this chapter we introduce the DLV system and give an overview of its architecture and implementation. The theoretical foundations of the implementation of DLV are also briefly discussed. The main procedure for the computation of the answer set semantics is then described.

2.1 The Architecture of DLV: an Overview

The system architecture of DLV is shown in Figure 2.1. The internal system language is the one described in Chapter 1, i.e. Disjunctive Logic Programming extended by weak constraints. The DLV Core (the shaded part of the figure) is an efficient engine for computing answer sets (one, some, or all) of its input. The DLV core has three layers (see Figure 2.1), each of which is a powerful subsystem per se: The *Intelligent Grounder* (IG, also *Instantiator*) has the power of a deductive database system; the *Model Generator* (MG) is as powerful as a Satisfiability Checker; and the *Model Checker* (MC) is capable of solving co-NP-complete problems. In addition to its kernel language, DLV provides a number of application front-ends that show the suitability of our formalism for solving various problems from the areas of Artificial Intelligence, Knowledge Representation and (Deductive) Databases. Currently, the DLV system has front-ends for inheritance reasoning [Buccafurri *et al.*, 2002], model-based diagnosis [Eiter *et al.*, 1997a], planning [Eiter *et al.*, 2003b], and SQL3 query processing. Each front-end maps its problem specific input into a DLV program, invokes the DLV kernel, and then post-processes any answer set returned, extracting from it the desired solution;

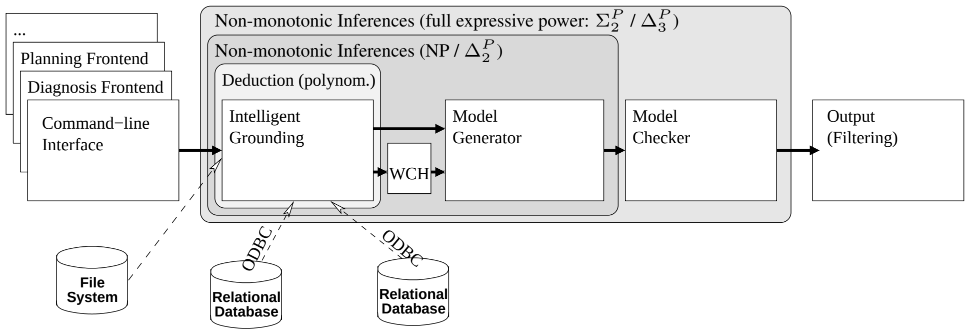


Figure 2.1: The System Architecture of DLV

furthermore, there is a Graphical User Interface (GUI) that provides convenient access to some of these front-ends as well as the system itself.

2.2 Theoretical Foundations

The implementation of the DLV system is based on very solid theoretical foundations, and exploits the results on the computational complexity discussed in section 1.2. Ideally, the performance of a system should reflect the complexity of the problem at hand, such that "easy" problems (say, those of polynomial complexity) are solved fast, while only harder problems involve methods of higher run-time cost. Indeed, the DLV system is designed according to this idea, and thrives to exploit the complexity results reported in Section 1.2.4.

For example, stratified normal programs (which have polynomial complexity, as reported in Table 1.1[1]) are evaluated solely using techniques from the field of deductive databases, without employing the more complex techniques which are needed to evaluate full DLV programs; in fact, such normal stratified programs are evaluated without generating the program instantiation at all.

The architecture of the DLV Core closely reflects complexity results for various subsets of our language. As mentioned before, the Intelligent Grounding (IG) module is able to completely solve some problems which are known to be of polynomial time complexity (like normal stratified programs); the Model Generator (together with the Grounding) is capable of solving NP-complete problems. Adding the Model Checker is needed to solve Σ_2^P-complete problems. The WCH (Weak Constraints Handler) comes into play only in presence of weak constraints.

[1]Note that the complexity of propositional DLV programs reported in Tables 1.1–1.3 coincides with the data complexity of non-ground DLV programs.

More precisely, referring to the notation of Section 1.2.2, we have the following five disjoint language classes L_1 – L_5 for evaluation:

- L_1 contains the programs included in the class $\langle\{\}, \{w, \mathrm{not}_s\}\rangle$, which all have polynomial complexity. They are completely evaluated by the IG module, which runs in polynomial time (referring to propositional complexity).

- L_2 contains the programs which are in the subclass corresponding to $\langle\{\,\mathtt{v_h}\,\}, \{\mathrm{not}\}\rangle$, but not in L_1. The complexity of this fragment is NP, and the programs are evaluated by the MG module (besides the IG) with only a call to the linear-time part of the MC module. Note that the MG implements a flat backtracking algorithm and is suitable for solving NP-complete problems.

- L_3 contains the DLV programs from $\langle\{\,\mathtt{v_h}\,\}, \{\mathrm{not}, w\}\rangle$ minus $L_1 \cup L_2$. The complexity of this fragment is Δ_2^P. Here, also the WCH module is employed, which iteratively invokes the MG. Again, only the linear-time part of the MC is invoked.

- L_4 contains the programs from the subclass corresponding to $\langle\{\,\mathrm{v}\,\}, \{\mathrm{not}\}\rangle$ minus $L_1 \cup L_2 \cup L_3$. The complexity of this fragment is Σ_2^P, and the programs are evaluated by the MG module (besides the IG) with calls to the *full* MC module. Note that a flat backtracking algorithm is not sufficient to evaluate Σ_2^P-complete problems, and such a nested evaluation scheme, with calls to MC, is needed.

- Finally, L_5 contains all other programs, i.e., those in the full class (corresponding to $\langle\{\,\mathrm{v}\,\}, \{\mathrm{not}, w\}\rangle$) which are not contained in $L_1 \cup L_2 \cup L_3 \cup L_4$, where we have the full language complexity of Δ_3^P. The evaluation proceeds as for L_4, but also the WCH module comes into play for handling the weak constraints.

The three DLV modules, MG, MC, and WCH, thus deal with the three sources of complexity denoted by (s_1), (s_2), and (s_3) in Section 1.2.4; each of them is fully activated *only if* the respective source of complexity is present in the program at hand.

2.3 DLV Computation

Next we present the evaluation flow of the DLV computation in some more detail. It is worth noting that we describe the computational engine of the DLV system

[Faber *et al.*, 1999; 2001], but also other systems (like Smodels [Niemelä and Simons, 1996; Simons, 2000], for instance) employ very similar techniques. Upon startup, the DLV Core or one of the front-ends parses the input specified by the user and transforms it into the internal data structures of DLV. In both cases, this is done efficiently (requiring only linear memory and time). The input is usually read from text files, but DLV also provides a bridge to relational databases through an ODBC interface, which allows for retrieving facts stored in relational tables.

Using differential and other advanced database techniques ([Faber *et al.*, 1999; Leone *et al.*, 2001]) together with suitable data structures, the *Intelligent Grounding* (IG) module then efficiently generates a ground instantiation $Ground(\mathcal{P})$ of the input that has the same answer sets as the full program instantiation, but is much smaller in general. For example, in case of a stratified program, the IG module already computes the single answer set, and does not produce any instantiation.

The heart of the computation is then performed by the Model Generator and the Model Checker. Roughly, the former produces some "candidate" answer sets, the stability of which is subsequently verified by the latter. In presence of weak constraints, further processing is needed, which is performed under the control of the WCH module. Since the handling of weak constraints is somehow orthogonal to the rest of the computation, we first focus on the evaluation of standard disjunctive logic programs, describing the processing of weak constraints later on.

The generation of the answer sets of a program $\mathcal{P}$ relies on a monotonic operator $\mathcal{W}_{\mathcal{P}}$ [Leone *et al.*, 1997] which extends the well-founded operator of [Gelder *et al.*, 1991] for normal programs to disjunctive programs. It is defined in terms of a suitable notion of *unfounded set*. Intuitively, an unfounded set for a disjunctive program $\mathcal{P}$ w.r.t. an interpretation I is a set of positive literals that cannot be derived from $\mathcal{P}$ assuming the facts in I [Leone *et al.*, 1997].

Briefly, the MG works as follows: First, $\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset)$ (the fixpoint of $\mathcal{W}_{\mathcal{P}}$) is computed, which is contained in every answer set. If $\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset)$ is a total model, it is returned as the (unique) answer set. Otherwise, moving from $\mathcal{W}_{\mathcal{P}}^{\omega}(\emptyset)$ towards the answer sets, a literal (called *possibly-true literal* in [Leone *et al.*, 1997]), the truth of which allows to infer new atoms, is assumed true. Clearly the choice of "good" possibly-true literals at each step (i.e., a sequence of possibly-true literals that quickly leads to an answer set) is crucial for an efficient computation, so we employ novel heuristics with extensive lookahead and also propagate knowledge about choices that lead to inconsistency [Faber *et al.*, 2001].

The computation proceeds by alternately selecting a possibly-true literal and applying the pruning operator, until either a total model of $Ground(\mathcal{P})$ is reached or two contradictory literals are derived. If a model is found, the Model Checker is called; otherwise, backtracking is performed.

The *Model Checker* (MC) verifies whether the model M at hand is an answer set for the input program $\mathcal{P}$. In particular, the MC disregards weak constraints, and verifies whether M is an answer set for $Rules(\mathcal{P})$; the optimality of the models w.r.t. the violation of weak constraints is handled by the WCH module. The task performed by MC is very hard in general, because checking the stability of a model is well-known to be co-NP-complete (cf. [Eiter *et al.*, 1997b]). However, for some relevant and frequently used classes of programs answer-set checking can be efficiently performed (see Table 1.3 in Section 1.2.4).

The MC implements novel techniques for answer-set checking [Koch and Leone, 1999], which extend and complement previous results [Ben-Eliyahu and Dechter, 1994; Ben-Eliyahu and Palopoli, 1994; Leone *et al.*, 1997]. The MC fully complies with the complexity bounds specified in section 1.2. Indeed, (a) it terminates in polynomial time on every program where answer-set checking is tractable according to Table 1.3 (including, e.g., HCF programs); and (b) it always runs in polynomial space and single exponential time. Moreover, even on general (non-HCF) programs, the MC limits the inefficient part of the computation to the subprograms that are not HCF. Note that it may well happen that only a very small part of the program is not HCF [Koch and Leone, 1999].

Finally, once an answer set has been found, the control is returned to the front-end in use, which performs post-processing and possibly invokes the MG to look for further models.

In presence of weak constraints, after the instantiation of the program, the computation is governed by the WCH and consists of two phases: (i) the first phase determines the cost of an optimal answer set[2], together with one "witnessing" optimal answer set and, (ii) the second phase computes all answer sets having that optimal cost. It is worthwhile noting that both the IG and the MG also have built-in support for weak constraints, which is activated (and therefore incurs higher computational cost) only if weak constraints are present in the input. The MC, instead, does not need to provide any support for weak constraints, since these do not affect answer-set checking at all.

[2]By *cost* of an answer set we mean the sum of the weights of the weak constraints violated by the answer set, weighted according to their priority level – see Section 1.1.2.

Chapter 3

The DLV Instantiation Module

In this chapter, we provide a short description of the overall instantiation module of DLV, the Intelligent Grounding, and focus on the “heart” procedure of this module which produces the ground instances of a given rule.

3.1 Architecture

In Figure 3.1 we have depicted the general structure of the instantiator module.

An input program $\mathcal{P}$ is first analyzed by the parser, which also builds the extensional database from the facts in the program, and encodes the rules in the intensional database in a suitable way. Then, a rewriting procedure (see [Faber *et al.*, 1999]), optimizes the rules in order to get an equivalent program $\mathcal{P}'$ that can be instantiated more efficiently and that can lead to a smaller ground program. At this point, another module of the instantiator computes the dependency graph of $\mathcal{P}'$, its connected components, and a topological ordering of these components. Finally, $\mathcal{P}'$ is instantiated one component at a time, starting from the lowest components in the topological ordering, i.e., those components that depend on no other component, according to the dependency graph.

3.2 General Instantiation Algorithm

The aim of the instantiator is mainly twofold: (i) to fully evaluate($\vee$ -free) stratified program components, and (ii) to generate the instantiation of disjunctive or unstratified components (if the input program is disjunctive or unstratified).

In order to evaluate efficiently stratified programs (components), DLV uses an

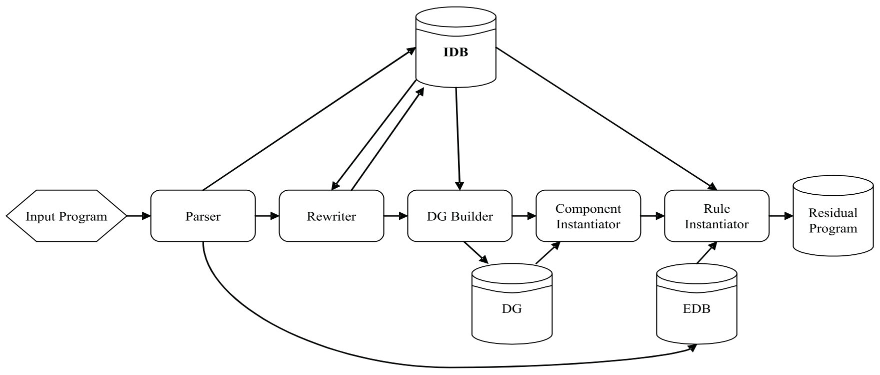


Figure 3.1: Architecture of DLV Instantiator

improved version of the generalized semi-naive technique [Ullman, 1989] implemented for the evaluation of linear and non-linear recursive rules.

If the input program is normal (i.e., v -free) and stratified, the instantiator evaluates the program completely and no further module is employed after the grounding; the program has a single answer set, namely the set of the facts and the atoms derived by the instantiation procedure. If the input program is disjunctive or unstratified, the instantiation procedure cannot evaluate the program completely. However, the optimization techniques mentioned above are useful to generate efficiently the instantiation of the non-monotonic part of the program. Two aspects are crucial for the instantiation:

(a) the number of generated ground rules,

(b) the time needed to generate such an instantiation.

The size of the generated instantiation is important because it strongly influences the computation time of the other modules of the system, running in exponential time (in the worst case) in the size of the instantiation. A slower instantiation procedure generating a smaller grounding may be preferable to a faster one generating a large grounding. However, the time needed by the former cannot be ignored otherwise we could not really have a gain in the total computation time.

The main reason of large groundings even for small input programs is that each atom of a rule in $\mathcal{P}$ may be instantiated to many atoms in $B_{\mathcal{P}}$, leading to a combinatorial explosion. However, most of these atoms may not be derivable whatsoever, and ground instances containing these atoms in the positive bodies are completely useless, they will not lead to any derivation. The instantiator module generates ground instances of rules containing only atoms which can possibly be derived from $\mathcal{P}$. As an example, consider the following program $\mathcal{P}_{Inst}$

$$\begin{array}{ll} a(1) \text{ :- } not\ a(2). & b(2) \text{ :- } not\ b(3). \\ a(2) \text{ :- } not\ a(1). & b(3) \text{ :- } not\ b(2). \\ \multicolumn{2}{c}{r:\ p(X) \text{ :- } a(X), b(X).} \end{array}$$

The full instantiation of $\mathcal{P}_{Inst}$ contains three ground instances of rule r, namely

$$\begin{array}{l} r_1:\ p(1) \text{ :- } a(1), b(1). \\ r_2:\ p(2) \text{ :- } a(2), b(2). \\ r_3:\ p(3) \text{ :- } a(3), b(3). \end{array}$$

However, r_1 and r_3 are useless, they will never be applicable because their bodies contain atoms $b(1)$ and $a(3)$, respectively, that are not derivable from $\mathcal{P}_{Inst}$ (they do not appear in the head of any rule).

The DLV instantiator, instead, first computes the instantiation of the rules defining predicates a and b (this is the starting point, because a and b do not depend on any other predicate). Since these rules are already ground, their instantiation does not add anything new to the four ground rules; but has the only effect of determining the set of atoms for a and b that are "potentially" derivable, as they appear in the head of some rule instance. These sets are $\{a(1), a(2)\}$ and $\{b(2), b(3)\}$ in the example, and play the role of *actual domains*, only elements from these domains will be taken into consideration for instantiating positive body occurrences of a and b in other rules. Thus, the instantiation of r is computed by using only such atoms, and it therefore consists only of the ground rule r_2 (r_1 and r_3 are not generated at all).

3.3 Instantiate Component

In order to generate a small ground program equivalent to $\mathcal{P}$, the DLV instantiator takes into account some structural information of the input program, concerning the dependencies among IDB predicates.

The *Dependency Graph* of $\mathcal{P}$ is now defined, which, intuitively, describes how predicates depend on each other.

Definition 3.1 Let $\mathcal{P}$ be a program. The *Dependency Graph* of $\mathcal{P}$ is a directed graph $G_{\mathcal{P}} = \langle N, E \rangle$, where N is a set of nodes and E is a set of arcs. N contains a node for each IDB predicate of $\mathcal{P}$, and E contains an arc $e = (p, q)$ if there is a

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

$G_{\mathcal{P}}$

Figure 3.2: Dependency Graph.

rule r in $\mathcal{P}$ such that q occurs in the head of r and p occurs in a positive literal of the body of r. $\Box$

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

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

Graph $G_{\mathcal{P}}$ is illustrated in Figure 3.2; the strongly connected components of $G_{\mathcal{P}}$ are $\{s\}$, $\{q\}$ and $\{p, t\}$. They correspond to the three following modules:

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

[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.

Moreover, the first and second module do not contain recursive rules, while the third one contains one exit rule, namely $p(X, Y) \lor s(Y) \text{ :- } q(X), q(Y), not\, t(X, Y)$, and two recursive rules. □

The dependency graph 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 3.3 Consider the dependency graph $G_{\mathcal{P}}$ shown in Figure 3.2; it is easy to see that component $\{p, t\}$ depends on components $\{s\}$ and $\{q\}$, while $\{s\}$ depends only on $\{q\}$. □

This ordering can be exploited to single 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 C_j in the sequence (i.e. $i < j$). Intuitively, this partial ordering allows the 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.

A description of the instantiation process based on this principle follows.
A given 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.

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. 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 $\mathcal{P}_{C_2}$. Once C has been evaluated, it is removed from $G_{\mathcal{P}}$. This runs on until all the components of $G_{\mathcal{P}}$ have been evaluated.

Example 3.4 Let $\mathcal{P}$ be the program of Example 3.2. The unique component of $G_{\mathcal{P}}$ having no incoming edges is $\{q\}$. Thus the program module $\mathcal{P}_q$ is evaluated first. Then, once $\{q\}$ 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 $G_{\mathcal{P}}$,$\{p, t\}$ is processed at last, completing the instantiation process. □

The instantiation of each component is performed according to a semi-naïve evaluation technique [Ullman, 1989]. According to this schema, first of all, exit rules are considered and instantiate once, while concerning recursive rules, they are processed several times and at each iteration n, only the significant information derived during iteration $n-1$ has to be used.

Each rule r in the program module of C is processed by calling procedure *Instantiate* described in the following Section. 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, *Instantiate* 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.

3.4 Rule Instantiation

In this section, we describe the process of rule instantiation – the "heart" of the instantiation module – as it is currently implemented in DLV.

The algorithm *Instantiate*, shown in Figure 3.3, generates the possible instantiations for a rule r of a program $\mathcal{P}$. When this procedure is called, for each predicate q occurring in the body of r we are given its extension, as a set I_q containing all the ground instances of q that have been already generated (i.e., appearing in the head of a ground instance that has already been generated; I_q is called actual domain for q in the above example). We say that a mapping $\theta : var(r) \rightarrow U_{\mathcal{P}}$ is a valid substitution for r if it is valid for every positive literal occurring in its body, i.e., for every positive literal Q in $B(r)$, $\theta Q \in I_q$ holds. In other words, we discard a priori any substitution mapping a positive body literal Q to a ground instance of Q which is not in I_q. *Instantiate* outputs all such valid substitutions for r.

Note that, since the rule is safe, each variable occurring either in a negative literal or in the head of the rule appears also in some positive body literal. For the sake of presentation, we assume that the body is ordered in a way such that any negative literal always follows the positive atoms containing its variables. Actually, DLV has a specialized module that computes a clever ordering of the body [Leone *et al.*, 2001] (e.g., exploiting the quantitative information on the size of any predicate extension) that satisfies this assumption.

Instantiate first stores the body literals $L_1, \ldots, L_n$ into an ordered list $B = (null, L_1, \cdots, L_n, last)$. Then, it starts the computation of the valid substitutions for r. To this end, it maintains a variable θ, initially set to $\emptyset$, representing, at each step, a partial substitution for $var(r)$.

Now, the computation proceeds as follows: For each literal L_i, we denote by $PreviousVars(L_i)$ the set of variables occurring in any literal that precedes L_i in

```
Algorithm Instantiate
Input  $R$: Rule, $I$: Set of instances for the predicates occurring in $B(R)$;
Output $S$: Set of Total Substitutions;
var $L$: Literal, $B$: List of Atoms, $\theta$: Substitution, MatchFound: Boolean;
begin
    $\theta = \emptyset$;
    (* returns the ordered list of the body literals (null, $L_1, \cdots, L_n$, last)  *)
    $B$ := BodyToList($R$);
    $L := L_1$;   $S := \emptyset$;
    while $L \neq null$
        Match($L$, $\theta$, MatchFound);
        if  MatchFound
            if($L \neq$ last) then
                $L$ := NextLiteral($L$);
            else (* $\theta$ is a total substitution for the variables of $R$ *)
                $S := S \cup \theta$;
                $L$ := PreviousLiteral($L$);
                (* look for another solution *)
                MatchFound := False;
                $\theta := \theta \mid_{PreviousVars(L)}$;
        else
            $L$ := PreviousLiteral($L$);
            $\theta := \theta \mid_{PreviousVars(L)}$;
    output $S$;
end;
```

Figure 3.3: Computing the instantiations of a rule

the list B, and by $FreeVars(L_i)$ the set of variables which occur for the first time in L_i, i.e., $FreeVars(L_i) = var(L_i) - PreviousVars(L_i)$.

At each iteration of the **while** loop, by using function *Match*, we try to find a match for a literal L_i with respect to θ, in other words, we apply θ to L_i and look for an instantiation of θL_i that matches an atom in I_{L_i}. More precisely, we look in I_{L_i} for a ground instance G which is consistent with the assignments for the variables in $PreviousVars(L_i)$, and then use G in order to extend θ to the variables in $FreeVars(L_i)$; note that, if $FreeVars(L_i) = \emptyset$, this task simply consists in checking whether θ is a valid substitution for L_i. This is accomplished by the procedure *Match* (figure 3.4) that, in turns, calls *FirstMatch* if this is the first attempt to find a match for L_i, or *NextMatch* if we already have a valid substitution for L_i and

```
Procedure Match ($L$:Literal, var $\theta$:Substitution, var MatchFound: Boolean)
begin
    if MatchFound then (* this is the first try on a new literal *)
        FirstMatch($L$, $\theta$, MatchFound );
    else (* the last match failed, look for another match on a previous literal *)
        NextMatch($L$, $\theta$, MatchFound);
end;
```

```
Procedure FirstMatch ($L$: Literal, var $\theta$: Substitution, var MatchFound: Boolean)

 (* Look in the extension $I_L$ for the first tuple of values matching $\theta$, and possibly update $\theta$
    accordingly. The boolean variable MatchFound is assigned True if such a matching
    tuple has been found; otherwise, it is assigned False. *)

Procedure NextMatch ($L$: Literal, var $\theta$: Substitution, var MatchFound: Boolean)

 (* Similar to FirstMatch, but finds the next matching tuple. *)
```

Figure 3.4: The matching procedures

we have to look for a further one. If there is no such a substitution (*Match* returns *MatchFound* equal to *false*), then we backtrack to the previous literal in the list; else, we consider two cases: if there are further literals to be evaluated, then we continue with the next literal in the list; otherwise, we have successfully instantiated all positive body literals, θ encodes a (total) valid substitution (because of the safety condition, the instantiation of the positive literals implies that all rule variables have been instantiated) and is thus added to the output set S. Even in this case, we backtrack for finding another solution, since we want to compute *all* instantiations of r.

Note that this kind of classical backtracking procedure works well for rules with a few literals and with a few tuples for each predicate extension. However, DLV has been designed to work even for manipulating complex knowledge on large databases, and for such applications the simple algorithm described above is not satisfactory.

Example 3.5 Suppose we want to compute all ground instantiations of the rule

$$\begin{aligned} r_1 :\ a(X,Y) \text{:-}\, & p_1(X,Y), p_2(X,Z), p_3(Z,H,T), p_4(T,W), \\ & p_5(X,V,Z), p_6(X,Y,V). \end{aligned}$$

and that we have already computed a partial substitution θ for the variables $\{X, Y, Z,\ H, T, W\}$, but we are not able to find a valid substitution for p_5 (i.e., we could not find consistent value for V in the extension of p_5 in order to extend θ). In this case, according to the algorithm in Figure 3.3, we should backtrack to the previous literal p_4. However, the failure on atom $p_5(X, V, Z)$ is independent of variables $\{H, T, W\}$, and thus, to have a chance of finding a successful match for p_5, we should change the substitution for Z or X. It follows that, intuitively, we could safely "backjump" directly to atom $p_2(X, Z)$, where variable Z has been instantiated. Thus, jumping over both $p_3(Z, H, T)$ and $p_4(T, W)$, as a new match for p_3 or p_4 would not change the assignment of Z and X and would not therefore chance the situation of p_5. It is worthwhile noting that making such a jump can allow us to save a very large amount of time, especially if the extensions of p_3 and p_4 contain many tuples.

In order to overcome such troubles, a number of extensions of the backtracking technique have been described in the literature—see Section 4.4 on related work. However, for different reasons, none of these proposals perfectly fits our needs. E.g., some of them are designed only for binary constraint satisfaction problems, and for computing any solution for a given problem instance. Rather, we need a specialized algorithm that should be able to compute efficiently *all* instantiations of a rule with predicates of *arbitrary* arity, which corresponds to finding all solutions of general (non-binary) constraint satisfaction problems.

3.5 Instances Simplification and Stratification Handling

The rule instantiation algorithm, described above, generates total substitutions of the rule, which allow to compute the rule instances. However, the generated instances can be simplified by eliminating from the body the (positive and negative) literals which are already known to be true; moreover, since the substitution is computed looking only at the positive body literals, it may happen that some negative literal is known to be false and one can delete the rule instance completely in this case, since it can never be applied.

To this end, the set I of generated extensions is actually partitioned in two subsets grouping, respectively, the true and the potentially-true instances (denoted by I^T and I^{PT}, respectively). I^T is initialized with the facts, and then filled with the head of the rules becoming facts after that the simplification below is applied;

while the heads of the remaining rules belong to I^{PT} (I^{PT} gets the atoms from the disjunctive heads and from non-disjunctive heads whose body contains some atom in I^{PT} also after the simplification). We say that a literal with predicate q is *solved* if: (i) q is defined solely by non-disjunctive rules (i.e., all rules with q in the head are non-disjunctive), and (ii) q does not depend (even transitively) by any unstratified predicate [Apt *et al.*, 1988] or disjunctive predicate (i.e., a predicate defined by a disjunctive rule).

In order to exploit possible stratifications and to compute efficiently such extensions, the DLV instantiator evaluates predicates according to a topological ordering of their dependency graph. The instantiation starts by evaluating first the rules defining predicates P_0 that depend on no other predicates (that is, only defined by facts), then the predicates P_1 that only depend on predicates in P_0, and so on. Note that this instantiation ordering, together with the simplification below, guarantees that the extensions of solved predicates belong completely to the I^T component of I. That is, the truth values of all ground literals that are instances of solved predicates are fully determined by the instantiator. In particular, observe that, if the input program is non-disjunctive and stratified, all predicates are solved and the instantiator evaluates the program completely (that is, no further module is employed after the grounding). In fact, in this case, the program has a single answer set, coinciding with the set of atoms computed in I^T by the instantiation procedure.

It follows that none of the solved predicates should occur in the rules of the ground instantiation P' produced by the instantiator. All the predicates occurring in the rules of P' should be unsolved, and will be evaluated by the answer set solver in the second phase of the computation taking the ground instantiation as the input. Therefore, once a rule instance R is generated, the following actions are carried out for simplifying the program:

1. If a positive body literal Q is in $B(R)$ and $Q \in I^T$, then delete Q from $B(R)$.

2. If a solved negative body literal $\text{not} Q$ is in $B(R)$ and $Q \notin I$,[3] then delete $\text{not} Q$ from $B(R)$.

3. If a negative body literal $\text{not} Q$ is in $B(R)$ and $Q \in I^T$, then remove the ground instance R.

[3]Note that a solved literal is also stratified and, actually, condition $Q \notin I^T$ would be equivalent.

As far as the negative literals are concerned, it is worth noting that the safety condition ensures that each negative literal is made ground by the computed rule substitutions. Moreover, the component ordering obtained by the dependency graph ensures that when a rule with a stratified body predicate q is instantiated, the rules for predicate q have been instantiated previously and its true instances are already in I. In particular, in the case of disjunction-free and stratified programs, all generated rule instances are either simplified to facts or deleted.

Example 3.6 *Let $\mathcal{P}$ be the following disjunction-free stratified program*

$a(1).\ a(2).\ b(1).$
$r:\ p(X) \text{ :- } a(X),\ not\ b(X).$
$s:\ q(X,X) \text{ :- } p(X).$

Initially, I^T contains the facts appearing in $\mathcal{P}$, that is $I^T = \{a(1),\ a(2),\ b(1)\}$, while I^{PT} is empty. Then, the instantiation process starts by evaluating rule r, which depends only on facts, and the following two ground instances are computed

$r_1:\ p(1) \text{ :- } a(1),\ not\ b(1).$
$r_2:\ p(2) \text{ :- } a(2),\ not\ b(2).$

However, by applying the simplification described above, r_1 is removed because the negative body literal not $b(1)$ is known to be false ($b(1) \in I^T$), and r_2 becomes a fact $p(2)$ because the body literals $a(2)$ and not $b(2)$ are known to be true and are therefore eliminated (note that not $b(2)$ is a solved literal and $b(2) \notin I$). At this point, $p(2)$ is added to I^T, whence $I^T = \{a(1),\ a(2),\ b(1),\ p(2)\}$, and the instantiation continues by evaluating rule s (I^{PT} is still empty). The body of s contains a literal with predicate p, whose actual domain has been already computed and consists only of atom $p(2)$. Thus s has only one ground instance, namely

$s_1:\ q(2,2) \text{ :- } p(2).$

After the simplification, s_1 becomes a fact $q(2,2)$, which is added to I^T. Therefore, at the end of the instantiation, we have $I^{PT} = \emptyset$ and $I^T = \{a(1),\ a(2),\ b(1),\ p(2),\ q(2,2)\}$, which is in fact the unique answer set of $\mathcal{P}$.

Consider now the program $\mathcal{P}'$ obtained from $\mathcal{P}$ by replacing rule r by

$r':p(X) \vee s(X) \text{ :- } a(X),\ not\ b(X).$

As for program $\mathcal{P}$, we initially have $I^T = \{a(1),\ a(2),\ b(1)\}$ and $I^{PT} = \emptyset$. Then, rule r' is instantiated and the following two ground instances are computed

$$\begin{array}{l} r'_1 :\ p(1) \,\mathtt{v}\, s(1) \text{ :- } a(1),\ not\ b(1). \\ r'_2 :\ p(2) \,\mathtt{v}\, s(2) \text{ :- } a(2),\ not\ b(2). \end{array}$$

Similarly as before, by applying the simplification, r'_1 is removed, and $a(2)$ and $not\ b(2)$ are eliminated from the body of r'_2. However, we can not add the head of r'_2 to I^T, because its body is still not empty. Rather, the head atoms $p(2)$ and $s(2)$ are added to I^{PT}. Thus, we get $I^T = \{a(1),\ a(2),\ b(1)\}$ and $I^{PT} = \{p(2),\ s(2)\}$. At this point, rule s is instantiated and, as before, the only generated ground instance is $s_1 :\ q(2,2) \text{ :- } p(2)$. However, in this case the body literal $p(2)$ belongs to I^{PT}, and thus s_1 cannot be simplified and its head atom $q(2,2)$ is added to I^{PT}. Thus, the instantiation of $\mathcal{P}'$ consists of the ground rules

$$\begin{array}{l} p(2) \,\mathtt{v}\, s(2). \\ q(2,2) \text{ :- } p(2). \end{array}$$

Moreover, $I^T = \{a(1),\ a(2),\ b(1)\}$ and $I^{PT} = \{p(2),\ s(2),\ q(2,2)\}$.

Chapter 4

A BackJumping Technique for DLP Programs Instantiation

In chapter 3, we have described the instantiation process of DLV and we have seen that the efficiency of the instantiation procedure can be measured in terms of the size of its output and the time needed to generate this instantiation. In this chapter, we present a new kind of structure-based backjumping algorithm for rule instantiation that can be used in order to improve the efficiency of the instantiation procedure of DLV. In particular, this algorithm allows to reduce the size of the generated ground instantiation and optimize the execution time which is needed to generate it.

4.1 Some Motivations

As observed in chapter 3, the rule instances of a program $\mathcal{P}$ may contain many atoms that are not derivable whatsoever, and hence such instantiations do not render applicable rules. A good instantiator should generate ground instances of rules containing only atoms that can potentially be derived from $\mathcal{P}$.

Moreover, as described in the previous chapter, at each step of the instantiation process, the truth values of the ground instances of solved predicates are already fully determined by the instantiator (each of these ground literals is already known to be true or to be false, none is undecided - undefined). We have seen that the program may be simplified in such a way that all predicates occurring in the rules of the instantiation are unsolved. However, this process may be expensive for very large programs. Rather, we would like to have a grounding algorithm that directly avoids the generation of useless rule instances, as well as the generation of useless literals in useful rules—i.e., rules or literals, respectively, to be deleted according to the simplification procedure.

Example 4.1 Consider the following rule

$$r_2 : a(X,Z) \text{ :- } q_1(X,Z,Y), q_2(W,T,S), q_3(V,T,H), \\ q_4(Z,H), q_5(T,S,V).$$

Suppose we know that predicates q_3, q_4, and q_5 are solved, and consider the following ground instances for r_2:

$$a(x_1,z_1) \text{ :- } q_1(x_1,z_1,y_1), q_2(w_1,t_1,s_1), q_3(v_1,t_1,h_1), \\ q_4(z_1,h_1), q_5(t_1,s_1,v_1).$$

$$a(x_1,z_1) \text{ :- } q_1(x_1,z_1,y_1), q_2(w_1,t_1,s_1), q_3(v_2,t_1,h_1), \\ q_4(z_1,h_1), q_5(t_1,s_1,v_2).$$

$$\vdots$$

$$a(x_1,z_1) \text{ :- } q_1(x_1,z_1,y_1), q_2(w_1,t_1,s_1), q_3(v_{99},t_1,h_{100}), \\ q_4(z_1,h_{100}), q_5(t_1,s_1,v_{99}).$$

$$a(x_1,z_1) \text{ :- } q_1(x_1,z_1,y_1), q_2(w_1,t_1,s_1), q_3(v_{100},t_1,h_{100}), \\ q_4(z_1,h_{100}), q_5(t_1,s_1,v_{100}).$$

Now, assume that all these instances are applicable, that is, all instances of the atoms over the solved predicates q_3, q_4 and q_5 are true, and all instances of the atoms over unsolved predicates (i.e. atoms $q_1(x_1,z_1,y_1), q_2(w_1,t_1,s_1)$) could be true (i.e., they are not provably false, at this point as, for instance, are in some disjunctive rule head of the instantiation). Then, it is easy to see that all these 10000 rules are semantically equivalent to the single instance

$$a(x_1,z_1) \text{ :- } q_1(x_1,z_1,y_1), q_2(w_1,t_1,s_1).$$

A key observation here is that we are not interested in finding all the ”consistent” substitutions for all variables of a rule r. Rather, we have to find just their restrictions to a set of variables that we call *relevant variables* of r. This set contains all the variables occurring in literals over unsolved predicates, together with the variables occurring in the head of r. For instance, for rule r_2 in the above example, the relevant variables are X, Z, Y, W, T, and S.

Thus, we only need all the (applicable) instantiations of literals binding relevant variables, while the remaining ones are just used to validate such instances. For instance, to get the one—simplified—rule

$$sr: \quad a(x_1, z_1) :\!-\, q_1(x_1, z_1, y_1), q_2(w_1, t_1, s_1)$$

it is sufficient to compute just the one total substitution leading to the first rule

$$a(x_1, z_1) :\!-\, q_1(x_1, z_1, y_1), q_2(w_1, t_1, s_1), q_3(v_1, t_1, h_1),\ q_4(z_1, h_1), q_5(t_1, s_1, v_1)$$

because, after the simplification of the solved literals, all other rules would yield precisely the same ground instance sr. Equivalently, the projection of all the total substitutions over the relevant variables is always $X = x_1$, $Z = z_1$, $Y = y_1$, $W = w_1, T = t_1$, and $S = s_1$.

Recall that the ground rules corresponding to such substitutions projected onto the unsolved predicates (as the one shown in the above example) are called *relevant instances* for r.

4.2 The BJ_Instantiate Algorithm

In this section, we describe the Algorithm *BJ_Instantiate*, that given a rule r, a set I of ground instances for the predicates occurring in the body of r and a set of relevant variables *OutputVars*, returns the relevant instances of r with respect to I. Formally, *BJ_Instantiate* returns the projections on *OutputVars* of all the valid substitutions for r, that we call the *relevant solutions* of our problem.

The basic schema of this algorithm is no more the classical backtracking paradigm, but rather a structure-based backjumping paradigm, well studied in the constraint satisfaction area (see., e.g., [Dechter, 1990; Tsang, 1993; Prosser, 1993]). In algorithms of this kind, when a backtrack step is necessary, it is possible to jump back many elements, rather than just one as in the standard chronological algorithm. Of course, such jumps should be designed carefully, in order to avoid that some solution is missed, especially in our case, where we have to compute all (relevant) solutions.

Let r be a rule and B the ordered list of its body literals $(null, L_1, \cdots, L_n, last)$. We say that L_i $(1 \leq i \leq n)$ is a *binder* for a variable X if there is no literal L_j, with $1 \leq j < i$ such that $X \in var(L_j)$. Moreover, for a set of variables V and a literal L_k, let *ClosestBinder*(L_k,V) denote the greatest literal L_i among the binders of the variables in V. A crucial notion in our algorithm is the *Closest Successful Binder* (CSB), which represents, intuitively, the greatest literal that is a binder of some output variable X whose current assigned value belongs to the last computed solution. The CSB acts as a barrier for some kind of jumps, as described later in this section. Another important concept is a set of variables called *Failure Set*. This set is dynamically computed during the instantiation of a rule and, roughly, it keeps track of the failures of the matchings occurred on the way. In particular, when a match of a literal fails, its variables are added to the *Failure Set*, meaning that the reason of the failure may be the assignment made for one of these variables. On the other hand, when the match of a literal L succeeds, the variables occurring for the first time in L (that is *FreeVars(L)*) are removed from the set, meaning that the assignments for these variables are changed, and thus they are no more cause of failure.

Example 4.2 As a running example in this section, consider the following rule

$$\begin{aligned} r_3 : a(L,S) \coloneq\, & q_1(X,H), q_2(T,S), q_3(H,X,T), q_4(H,L), \\ & q_5(Z,X,V), q_6(T,Z,H). \end{aligned}$$

In order to instantiate r_3, our algorithm needs the additional information on the relevant variables and the already known instances for the predicates occurring in the body. Then, assume that q_1 is the only unsolved predicate and thus *OutputVars* $= \{L, S, X, H\}$ (L and S are head variables, and X and H occur in a literal over an unsolved predicate). Moreover, assume that we are given the following extensions for the predicates occurring in $B(r_3)$:

$$\begin{aligned} & \{q_1(x_1,h_1), q_1(x_2,h_1)\},\ \{q_2(t_1,s_1), q_2(t_2,s_1), q_2(t_2,s_2)\}, \\ & \{q_3(h_1,x_1,t_1), q_3(h_1,x_2,t_1), q_3(h_1,x_2,t_2)\},\ \{q_4(h_1,l_1)\}, \\ & \{q_5(z_1,x_2,v_1), q_5(z_1,x_2,v_2)\},\ \{q_6(t_2,z_1,h_1)\} \end{aligned}$$

Figure 4.1 shows the algorithm *BJ_Instantiate*. As for Algorithm *Instantiate*, at each iteration of the **while** loop, the procedure *Match* tries to find a match for a literal L_i with respect to the current partial substitution θ. If it succeeds and L_i is not the last literal, then we can proceed with the next literal L_{i+1}. Otherwise, we have to backtrack, and thus we have to decide where to jump and, possibly, update the current CSB. Now, we have a number of different cases to be handled, depending on the outcome *Status* of the procedure *Match*.

1. **Success, and θ encodes a total substitution**. Since also the match on the last literal is successful, θ encodes a valid substitution for the variables in r, and its restriction to *OutputVars* is therefore added to the set of solutions. Then, in order to look for further solutions, we have to backtrack. However, in this algorithm, we are not forced to go back to the previous literal. Rather, we can jump to the closest literal L_j binding a relevant variable, that is, jump to *ClosestBinder*($last$,*OutputVars*). In this case the CSB is set to closest literal to L_j binding a relevant variable, that is, CSB = *ClosestBinder*(L_j,*OutputVars*).

   Example 4.3 In our running example, the algorithm is able to find the total substitution $\theta(X) = x_2$, $\theta(H) = h_1$, $\theta(T) = t_2$, $\theta(S) = s_1$, $\theta(L) = l_1$, $\theta(Z) = z_1$, and $\theta(V) = v_1$. That is, we have a match for all the literals in B and we are at $last$. Then, the restriction of θ to the set of relevant variables is added to S. In our case, this solution corresponds to the following instance of r_3:

   $$\begin{aligned} a(l_1, s_1) \text{ :- } & q_1(x_2, h_1), q_2(t_2, s_1), q_3(h_1, x_2, t_2), \\ & q_4(h_1, l_1), q_5(z_1, x_2, v_1), q_6(t_2, z_1, h_1). \end{aligned}$$

   which is semantically equivalent to the following rule:

   $$a(l_1, s_1) \text{ :- } q_1(x_2, h_1).$$

   Now, according to the algorithm, we jump back to $q_4(H, L)$ for finding other solutions. Note that we do not look for further consistent tuples in the extensions of q_5 and q_6 because they do not bind any relevant variable. Indeed, possible solutions coming from other instances of these predicates (e.g., the solution with $\theta(V) = v_2$) would just lead to useless rules in the instantiation of the program at hand. Finally, the CSB is set to q_2 because it is the closest literal to q_4 binding a relevant variable (in our case S).

2. **Failure at the first attempt to find a match for a literal** L_i. We jump back to the closest literal L_j binding any of the variables in L_i, that is, jump to *ClosestBinder*(L_i,$var(L_i)$). Indeed, in this case, the only way for finding a match for L_i is to change the assignment for some of its bound variables.

   Example 4.4 In our running example, the first time that we try to find a match for q_5, we have computed the partial substitution $\theta(X) = x_1, \theta(H) = h_1, \theta(T) = t_1, \theta(S) = s_1$, and $\theta(L) = l_1$. In this case, we are not able to find any matching instance in the extension of q_5. Indeed, none of its instances has a value x_1 for variable X. Then, we have to change the value assigned to one of the variables occurring in q_5, and thus, we can safely jump over q_4, q_3, and q_2, and try to match again $q_1(X, H)$. Indeed, q_1 is the closest binder for $var(q_5)$, as it determines the value for variable X.

3. **Failure while looking for another match for a literal** L_i. In this case, L_i is a binder of some set of variables $\bar{X}$, and we fail in finding a different consistent substitution for these variables. Since we were successful on our first attempt to deal with L_i, this means that, for some reason, we jumped back to L_i from some later item, say L_j, of the list B. Now, we have to decide where to jump after the current failure, and this time the variables occurring in L_i are not the only candidates to be changed. Rather, we have to look at the *Failure Set* and, in particular, try to jump back to the closest literal to L_i binding one of the variables of the *Failure Set* (that is, jump to *ClosestBinder*(L_i,*FailureSet*)).

   Example 4.5 Assume that, in our running example, we are looking for another match for $q_5(Z, X, V)$ and that the *Failure Set* is $\{X, H, T, Z, V\}$. According to the algorithm, we have to jump to q_2, even if it is not a binder for any variable occurring in q_5. The reason is that q_2 is a binder for T, which belongs to the *Failure Set*, and changing its value may lead to some new solution (possibly comprising values already considered for the variables occurring in q_5).

   Another important issue concerns the management of the CSB. Indeed, if the *ClosestBinder*(L_i,*FailureSet*) precedes the CSB, we can not jump to it otherwise we could miss some relevant solutions; rather, we must jump to the CSB which acts as a barrier. Note that we eventually update the CSB, by pushing it back to the *ClosestBinder*(CSB,*OutputVars*).

Example 4.6 Let us continue from the execution step described at point 1 above, where we have found our first solution. Recall that we jumped back to $q_4(H, L)$ and the CSB is set to $q_2(T, S)$. Now, assume that the match of q_4 fails and that the current *Failure Set* is $\{L, X, H\}$. In this case, according to the *Failure Set* we could jump to q_1, but we are forced to stop our jumping back to literal $q_2(T, S)$, because of the CSB limit. It is worthwhile noting that, if we go directly to $q_1(X, H)$, we miss the solution obtainable by assigning s_2 to variable S and corresponding to the following instance of r_3:

$$\begin{aligned} a(l_1, s_2) :- \; & q_1(x_2, h_1), q_2(t_2, s_2), q_3(h_1, x_2, t_2), \\ & q_4(h_1, l_1), q_5(z_1, x_2, v_1), q_6(t_2, z_1, h_1). \end{aligned}$$

which is semantically equivalent to the rule

$$a(l_1, s_2) :- q_1(x_2, h_1).$$

Theorem 4.7 *Algorithm* BJ_Instantiate *is sound and complete. That is, given a rule r, the set I of the ground instances for the predicates occurring in its body, and the set of its relevant variables OutputVars,* BJ_Instantiate *computes the set containing all and only the projections over OutputVars of the valid substitutions for r w.r.t. I.*

Proof. Looking at the management of the set of substitutions S in Figure 4.1, it is evident that this set contains *only* the projections over *OutputVars* of the valid substitutions for r.

We next show that Algorithm *BJ_Instantiate* computes *all* such projections. The only potential source of incompleteness is jumping upon backtracking. We prove that in all the three cases where this may occur in the algorithm no relevant solution is missed:

(*SuccessfulMatch* **and** L **=** $last$.) In this case, we have just found a total substitution θ, and we backtrack looking for another solution. According to our algorithm, we backtrack to the closest literal L_j binding a variable of interest; that is, we jump to *ClosestBinder*($last$,*OutputVars*). In general, such a jump may avoid the generation of some valid substitution θ'. This means that, to compute θ', one should rather backtrack to a literal closer than L_j to $last$. Let L_k be this literal, with $k \geq j + 1$. Then, $\theta \mid_{PreviousVars(L_{j+1})} = \theta' \mid_{PreviousVars(L_{j+1})}$–possible further jumps before j are covered by the other cases below. Moreover, since L_j is the greatest literal in the body binding a relevant variable, *OutputVars* $\subseteq$ *PreviousVars*(L_{j+1}). Therefore, $\theta \mid_{OutputVars} = \theta' \mid_{OutputVars}$, and thus the substitution θ' is equal to θ, as far as relevant variables are concerned.

(*FailureOnFirstMatch*.) In this case, we have just failed in finding a first match for a certain literal L_i, $1 < i \leq n$. According to our algorithm, we backtrack to the closest literal L_j binding some variable of L_i. A solution may be lost only if the change of assignment of a jumped literal (i.e., a literal between L_j and L_i), would solve the conflict on L_i. However, this cannot be the case. Indeed, let L_k, $j < k < i$, be a jumped literal: whatever change we perform to the substitutions of the variables in $var(L_k)$, it does not affect the reason of the failure, because L_k does not bind any variables contained in $var(L_i)$, by the construction of L_j and the fact that $k > j$.

(*FailureOnNextMatch*.) In this case, we have just failed in finding another match for a literal L_i, $1 < i \leq n$. Then, according to the algorithm, we determine the closest literal to L_i binding a variable in the *Failure Set*, say L_j, and jump back either to it or to the CSB, depending on which is the closest to L_i. We next consider both cases: $L_j \geq CSB$ and $L_j < CSB$.

If $L_j \geq CSB$, it is easy to see that we cannot miss any solution. The proof is similar to the previous one: a valid substitution is missed only if there is a literal L_k, with $j < k < i$, such that, by changing values to its variables, the match of some literal L, which previously failed, now succeeds. Therefore, the variables of L should belong to the *Failure Set*, and thus L_k should bind some variables in the *Failure Set*. However, this contradicts the definition of L_j.

Let us consider now the case $L_j < CSB$, where we jump back to the CSB, according to the algorithm. Assume by contradiction that we miss some relevant solution, that is, there is a literal $L_k > CSB$ such that, by jumping back to L_k, we eventually get a relevant solution θ' that is not generated otherwise. Note that L_k does not bind any variable in the *Failure Set*, because $L_k > L_j$, and L_j is the closest to L_i with this property. It follows that L_k binds some relevant variables $\bar{X}$, and hence θ' can be obtained by keeping the same assignments of the last computed solution for *PreviousVars*(L_k), changing the assignment to $\bar{X}$, and continue with the construction of the substitution, as for the last computed solution. However, this is a contradiction with the definition of CSB, which is precisely defined as the last literal in the body whose assignment to some relevant variable belongs to the last computed solution.

4.3 Experiments and Benchmarks

In order to check the validity of the proposed method, we have implemented our technique in the grounding engine of the DLV system, and we have run the enhanced system on a collection of benchmark programs taken from different domains. We selected programs where the instantiation process is hard, and it takes a relevant part of the entire computation. In particular, we evaluated the method on the following problems:

- Strategic Companies (*StratComp*)
- 3-Colorability (*3-Col*)
- Constraint 3-Colorability (*Constraint-3-Col*)
- Ramsey Numbers (*Ramsey*)
- Timetabling
    - University Timetabling (*Timet_University*)
    - School Timetabling (*Timet_School*)
- Grammar Based Information Extraction (*GrammarBased_IE*)

All benchmark instances and encodings as well as the binaries used for our experiments can be retrieved at the Web page `http://www.mat.unical.it/catalano/thesisexperiments.zip`.

4.3.1 Benchmark Problems and Data

We next provide a short description of the benchmark problems we considered for the experiments and, for each problem, we specify the data and the encodings which we have used.

Strategic Companies (*StratComp*) 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 produced by the holding, 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. Each good is produced by at most four companies and each company is jointly controlled by at most four other companies.

A subset-minimal set $C' \subseteq C$ is called *strategic* 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 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 known to be Σ_2^P-hard in general [Cadoli *et al.*, 1997].
Encoding The encoding considered for this problem is the one described in Section 1.3.2.
Data We generated tests with instances for n companies ($200 \leq n \leq 8000$), $3n$ products, 10 uniform randomly chosen $contr\_by$ relations per company, and uniform randomly chosen $prod\_by$ relations. For each problem size we generated 10 random instances.

3-Colorability (*3-Col*) The 3-Colorability problem consists in the assignment of three colors to the nodes of a graph in such a way that adjacent nodes have different colors. This problem is known to be NP-complete.
Encoding Let us suppose that nodes and edges are represented by a set $\mathcal{F}$ of facts with predicates $node$ (unary) and $edge$ (binary), respectively. Then, the following program allows us to determine the admissible colorings for the given graph.

$$\begin{array}{ll} r_1: & color(X, red) \vee color(X, yellow) \vee color(X, green) \text{ :- } node(X) \\ c_1: & \text{:- } edge(X, Y), color(X, C), color(Y, C) \end{array}$$

Rule r_1 above states that every node is colored red or yellow or green; while the constraint c_1 forbids the assignment of the same color to two adjacent nodes. The answer sets of $F \cup \{r_1\}$ are all the possible colorings of the graph. If an answer set of $\mathcal{F} \cup \{r_1\}$ satisfies the constraint c_1, then it represents an admissible 3-coloring for the graph. Thus, there is a one-to-one correspondence between the

	%solved instances		average time	
Constraint-3-Col	*dlv*	*dlvBJ*	*dlv*	*dlvBJ*
35 nodes, 35 edges	100%	100%	9.41	0.01
35 nodes, 45 edges	100%	100%	11.99	0.01
40 nodes, 40 edges	100%	100%	36.83	0.01
40 nodes, 50 edges	93%	100%	56.58	0.02
45 nodes, 45 edges	83%	100%	77.76	0.02
45 nodes, 55 edges	63%	100%	72.60	0.02
50 nodes, 50 edges	50%	100%	150.66	0.02
50 nodes, 60 edges	43%	100%	199.87	0.02
55 nodes, 55 edges	17%	100%	143.31	0.01
55 nodes, 65 edges	7%	100%	89.59	0.01
60 nodes, 60 edges	10%	100%	371.50	0.03
60 nodes, 70 edges	0%	100%	-	0.02

Table 4.1: Results for *Constraint-3-Col* (times in seconds)

solutions of the 3-coloring problem and the answer sets of $\mathcal{F} \cup \{r_1\}$ satisfying c_1. The graph is 3-colorable if and only if there exists one of such answer set.
Data We used ladder graphs with n levels, i.e., $2 * n$ nodes ($100 \leq n \leq 7500$ and simplex graphs generated with the Stanford GraphBase [Knuth, 1994], using the function $simplex(n, n, -2, 0, 0, 0, 0)$, $10 \leq n \leq 160$ (i.e. $(n + 1) * (n + 2)/2$ nodes).

Constraint 3-Colorability (*Constraint-3-Col*) In addition to the encoding described above, we considered for the 3-Colorability problem an alternative representation in a constraint satisfaction style.
Encoding Let us consider the graph G shown below and let $\mathcal{F} = \{edge(g, r), edge(g, b), edge(r, b), edge(r, g), edge(b, g), edge(b, r)\}$ be the collection of input facts representing all the possible colorings with 3 colors (green, red and blue) of adjacent nodes. The following one-rule program $\mathcal{P}$ allows us to decide if G is 3-colorable or not.

Intuitively, the body of r encodes the connections between the nodes of the graph G. $\mathcal{P} \cup \mathcal{F}$ has a single answer set; G is 3-colorable if and only if the atom *colorable* belongs to the answer set.

It is worthwhile noting that for this problem we do not use a single, uniform encoding to solve all instances, rather, we have one program for each instance. In particular, the length of the body of r changes with the number of edges of the graph. More precisely, if m is the number of edges, r will have exactly m literals in its body.

Data We considered random graphs generated with the Stanford GraphBase, using the function $ramdom\_graph(\#nodes, \#edges, 0, 0, 1, 0, 0, 0, 0, seed)$ for different values of $\#nodes$ and $\#edges$. For each problem size we generated 30 random instances.

Ramsey Numbers (*Ramsey*) The Ramsey number $ramsey(k, m)$ is the least integer n such that, no matter how we color the edges of the complete undirected graph (clique) with n nodes 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). Ramsey numbers exist for all pairs of positive integers k and m [Radziszowski, 1994].

Encoding The encoding considered for this problem is the one described in Section 1.3.2.

Note that, as for the Constraint 3-Colorability problem, we do not have a single, uniform encoding to solve all instances, rather we have one program for each instance. In particular, for checking that n is the Ramsey number $ramsey(k, m)$, the first constraint contains $\binom{k}{2}$ atoms with predicate red and the second constraint contains $\binom{m}{2}$ atoms with predicate $blue$.

Data We considered the problem of deciding, for varying k, m, and n, if n is the Ramsey number $ramsey(k, m)$.

Timetabling We considered two different versions of the timetabling problem. The first one (*Timet_University*) consists in determining a timetable for some university lectures that 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. This problem can be made harder by changing the number of students groups. For our experiments we considered 4, 5, and 7 groups. The second one is a school timetabling problem (*Timet_School*) where we look for an assignment of rooms and time-slots to a given set of lessons such that there are no conflicting assignments for classes, teachers and rooms The encodings of these problems are quite long and, thus, they are not reported here.

Grammar Based Information Extraction (*GrammarBased_IE*) This problem has been used at the First Answer Set Programming System Competition `http://asparagus.cs.uni-potsdam.de/contest/`. It constitutes a part of a more complex application for recognizing and extracting meaningful information from unstructured Web documents. In particular, given a context free grammar, which specifies arithmetic equations, and a string, the problem is to determine whether the input string is an equation belonging to the language defined by the grammar and whether the equation holds. The encoding of this problem is quite long and involved, thus we do not report it here. Concerning the data, we have used three different instances $in\_008.asp$, $in\_024.asp$, $in\_032.asp$. Both the encoding and the instances can be retrieved from the web page of the competition.

4.3.2 Results and Discussion

We have implemented the Algorithm *BJ_Instantiate* in C++, and we have integrated our implementation in the instantiator module of the DLV system. Then, we have run a number of experiments by using the above described benchmark problems, in order to compare the performance of the previous backtracking-based rule instantiator of DLV with the new method proposed in this paper. All binaries have been produced by the GNU compiler GCC 4.1.2, and the experiments have been performed on a Intel Pentium M 1.3 GHz with 768 Mbytes of main memory.

Figure 4.6 shows the results obtained for *3-Col* on ladder graphs (top) and simplex graphs (bottom), while Figure 4.5 displays the results for *StratComp*. The results for *Ramsey*, *Timetabling* and *GrammarBased_IE* are shown in Tables 4.2, 5.2, and 4.4, respectively. Table 4.1, finally, contains the results for *Constraint-3-Col*. In the graphs and the tables, *dlv* (resp. *dlvBJ)* denotes the instantiator obtained using Algorithm *Instantiate* (resp., Algorithm *BJ_Instantiate*) in the rule instantiator module of DLV. For every instance, we allowed a maximum running time of 600 seconds (ten minutes). In the graphs, the line corresponding to an instantiator stops whenever some problem instance was not solved within the allowed time limit; in the tables, this information is represented by means of the symbol '$-$'.

In general, the experimental results confirm the intuition that the new backjumping-based procedure outperforms the previous one in many cases, and can be very useful for improving the efficiency of DLV (and of any other DLP system that could exploit its instantiator).

Ramsey	*dlv*	*dlvBJ*
ramsey(3,7) $\neq$ 21	74.03	49.74
ramsey(3,7) $\neq$ 23	129.98	75.63
ramsey(4,6) $\neq$ 26	81.38	54.98
ramsey(4,6) $\neq$ 31	320.29	208.64
ramsey(4,7) $\neq$ 23	148.46	97.35
ramsey(4,7) $\neq$ 25	429.89	249.54
ramsey(5,6) $\neq$ 27	167.88	125.45
ramsey(5,6) $\neq$ 30	293.59	194.45

Table 4.2: Results for *Ramsey* (times in seconds)

Timetabling	*dlv*	*dlvBJ*
University, 4 groups	85.95	36.57
University, 5 groups	181.25	73.90
University, 7 groups	472.37	198.67
School	347.53	184.70

Table 4.3: Results for *Timetabling* (times in seconds)

GrammarBased_IE	*dlv*	*dlvBJ*
GrammarBased_IE_1	82.82	74.74
GrammarBased_IE_2	63.25	56.07
GrammarBased_IE_3	46.84	40.11

Table 4.4: Results for *GrammarBased_IE* (times in seconds)

Of course, the speed-up is not that high on input programs where rules are very short or where body literals have many variables in common, as the two procedures have a similar behavior in these cases. This is witnessed, for instance, by *GrammarBased_IE* which exhibits however a little performance gain (about 12%).

We observe an impressive speed-up when programs contain some rules with many literals in their bodies and/or when such rules have a few relevant variables (i.e., many solved predicates occur in their bodies). For instance, *Constraint-3-Col* consists of a single long rule where all predicates are solved. In fact, in this extreme case, the old instantiator is not able to solve many of the instances within the allowed time limit, while the new instantiator has solved all the instances almost instantaneously. It is worthwhile noting that, even when the old backtracking procedure is able to compute the instantiation, it generates thousands of redundant rule instances for this problem. As an example, on a graph with 35 nodes and 35 edges, the instantiation generated by the old procedure, consists of 767829 rules[1], while the new procedure generates an instantiation containing just one rule.

Interestingly, we observe a very nice speed-up even in some cases where all variables are relevant, like *Ramsey*, *3-Col*, *StratComp*, and *Timetabling*. In particular, the pictures for *3-Col* and *StratComp*, which are similar, show that *dlvBJ* clearly outperforms the old instantiator with a gain of about 50%. Furthermore, for *3-Col*, the old DLV instantiator was not able to solve all the instances within the time limit. More precisely, for ladder graphs, it solved instances up to size 12300 (i.e., graphs with 12300 nodes) and for simplex graphs up to 8385 (i.e, graphs with 8385 nodes), while the *dlvBJ* solved all considered instances, that is, up to 15000 (resp. 13041) nodes for ladder (resp. simplex) graphs. Tables 4.2 and 5.2 show that also in *Ramsey* and *Timetabling* the usage of *dlvBJ* provides a nice performance gain (about 35% for *Ramsey* and 57% for *Timetabling*). In all these cases, since all variables are relevant, the instantiations generated by the two procedures are identical, but the *dlvBJ* is faster since it "jumps" several useless steps during the computation.

[1]Actually, DLV does not materialize the duplicated rules, but it does generate them many times. To check the number of duplicates in the actual execution, one can modify the head to $colorable \text{ v } noncolorable$ and check the size of instantiation.

4.4 Related Work

In order to overcome the troubles of traditional backtracking, many extensions and improvements of this technique have been described in the literature, both in the logic programming and in the constraint satisfaction communities.

For instance, we recall the *intelligent backtracking* technique developed in [Bruynooghe and Pereira, 1984] for evaluating logic programs, and the *intelligent backtracking* technique developed in [Shen, 1996] for a parallel implementation of Prolog. In particular, the latter paper has some similarity with our approach, as the author defines the notion of groups, which are clusters of atoms that are independent of other clusters of atoms in a rule. Exploiting groups, it is possible to jump back in a clever way. However, inside groups, his approach works as the sequential backtracking, apart for some special features dealing with parallelism issues, and completely unrelated to our work. Moreover, there is no notion of relevant variables (according to our meaning of "relevance"). This is a crucial feature of our algorithm. Indeed, whenever some predicates are solved and not all variables occur in the rule head (which is often the case), we focus only on a subset of the substitutions, and we can get rid of a large number of solutions that we do not generate at all, with a clear computational advantage (witnessed by our experiments, see the previous section).

Our algorithm is also strongly related to the various backjumping techniques proposed for solving constraint satisfaction problems (CSPs) [Tsang, 1993; Dechter, 1990; Prosser, 1993]. Indeed, the rule instantiation problem can be viewed also as a CSP. However, most algorithms for solving CSPs focused on problems with binary constraints only, and compute only one solution (if any). On the contrary, in our context, we have to compute efficiently *all* instantiations of a rule with predicates of *arbitrary* arity, which corresponds to the problem of finding all solutions of general (non-binary) constraint satisfaction problems.

In the CSP community, a proposals in this direction is described in [Chen and van Beek, 2001]. In this paper, the authors provide a revised version of the conflict-directed backjumping algorithm, with a variation that allows the algorithm to compute all solutions of a CSP, without completely degenerating to the chronological backtracking. Their approach also works for non-binary CSPs. However, their algorithm is quite different from ours for the following reasons: (i) The way variables are made bound is the same as the usual algorithms proposed for binary CSPs. That is, they consider a variable at a time, while our technique is based on the instantiation of an atom at a time. In fact, we introduced the notion of closest successful binder (an atom), and we guarantee the completeness of the algorithm in a different way. (ii) They have no notion of relevant variables, and thus their algorithm misses one of the distinguishing features of our proposal, as discussed above.

Concerning other instantiators, to our knowledge, no one exploits backjumping techinique, except the recent istantiator *GrinGo* [Gebser *et al.*, 2007b]. In particular, *GrinGo* already exploits the backjumping technique described in this chapter with very good results, and performances similar to DLV. For instance, the instantiation generated by *GrinGo* for the ground program for the Constraint 3-Colorability problem consists of only one rule, as for DLV.

Interestingly, *GrinGo* also introduces a little modification to our algorithm, in order to avoid the re-generation of some solutions that we may obtain in case of literals binding both relevant and non relevant variables. For instance, consider the program

$$c(X) \mathrel{:-} a(X,Y), f(X).$$
$$a(1,1). \quad a(1,2). \quad f(1).$$

and assume that f is an unsolved predicate. Here, according to Algorithm *BJ_Instantiate*, after we found the first solution $c(1) \mathrel{:-} f(1)$, we jump back to $a(X,Y)$ because it instantiates the relevant variable X, and find a new assignment for this literal by using the fact $a(1,2)$. In this way, we explore the substitution with $X = 1$ and $Y = 2$, whose projection on the relevant variables is identical to the previous one, and thus adds nothing to the result set S.

In order to overcome this, *GrinGo* introduced the notion of *binder splitting*: in our example, the instantiator internally replaces the literal $a(X,Y)$ with two literals $a(X,\_)$ and $a(X,Y)$, one binding the relevant variable X and the other one binding the non relevant variable Y.

We are aware of such cases where we may try some more substitutions that are unnecessary. Of course, we may avoid them by exploiting a suitable "informed matching procedure," which distinguishes between relevant and non relevant variables. However, we did not implement this variant in the current version for two reasons: first, in our benchmark programs and our applications, the overhead for dealing with the informed matching slows down the system; and second—strongly related to the previous one, many "pathological" cases as the one described above do not really occur in DLV, because of its rewriting module (see Section 3.2). Indeed, rules are automatically rewritten by exploiting well known database optimization techniques, consisting in "pushing down" projections and selections. For instance, in the program of the previous example, the Rewriter module replaces the rule $c(X) \text{:-}\, a(X,Y), f(X)$ by the rules $c(X) \text{:-}\, aux(X), f(X)$, and $aux(X) \text{:-}\, a(X,Y)$. In sum, DLV instantiator and *GrinGo* use very similar backjumping techniques, but *GrinGo* requires more severe syntactic restrictions which limit its effectiveness on data-intensive applications (as deductive databases and many real-world applications).

```
**Algorithm** *BJ_Instantiate*
**Input** $R$: Rule, $I$: Set of instances for the predicates occurring in $B(R)$,
        *OutputVars*: Set of Variables;
**Output** $S$: Set of Substitutions;
**var** $L$: Literal, $B$: List of Atoms, $\theta$: Substitution, $CSB$: Literal,
    *Status*: MATCH_STATUS, *FailureSet*: Set of Variables;
**begin**
    $\theta = \emptyset$;
    (* returns the ordered list of the body literals $(null, L_1, \cdots, L_n, last)$ *)
    $B$ := *BodyToList*($R$);
    $L := L_1$;   *Status* := SuccessfulMatch;
    $CSB := null$;   $S := \emptyset$;   *FailureSet* := $\emptyset$
    **while** $L \neq null$
        *Match*($L$, $\theta$, *Status*);
        **switch** (*Status*)
            **case** SuccessfulMatch
                *FailureSet* := *FailureSet* - *FreeVars*($L$);
                **if**($L \neq last$) **then**
                    $L$ := *NextLiteral*($L$);
                **else** (* $\theta$ is a total substitution for the variables of $R$ *)
                    $S := S \cup \theta \mid_{OutputVars}$;
                    $L$ := *BackFromSolutionFound*($L$, $CSB$, *Status*);
                    $\theta := \theta \mid_{PreviousVars(L)}$;
                **break;**
            **case** FailureOnFirstMatch
                *FailureSet* := *FailureSet* $\cup$ *var*($L$);
                $L$ := *BackFromFailureOnFirstMatch*($L$, $CSB$);
                $\theta := \theta \mid_{PreviousVars(L)}$;
                **break;**
            **case** FailureOnNextMatch
                *FailureSet* := *FailureSet* $\cup$ *var*($L$);
                $L$ := *BackFromFailureOnNextMatch*($L$, *FailureSet*, $CSB$);
                $\theta := \theta \mid_{PreviousVars(L)}$;
                **break;**
    **output** $S$;
**end;**
```

Figure 4.1: The BJ_Instantiate algorithm

```
enum MATCH_STATUS = { SuccessfulMatch, FailureOnFirstMatch, FailureOn-
NextMatch};

Procedure Match ($L$:Literal, var $\theta$:Substitution, var Status: MATCH_STATUS)
begin
    if Status = SuccessfulMatch then
        (* the last match was successful, this is the first try on a new literal *)
        FirstMatch($L$, $\theta$, Status);
    else (* the last match failed, look for another match on a previous literal *)
        NextMatch($L$, $\theta$, Status);
end;


Procedure FirstMatch ($L$: Literal, var $\theta$: Substitution, var Status: MATCH_STATUS)

 (* Look in the extension $I_L$ for the first tuple of values matching $\theta$, and possibly update
      $\theta$ accordingly. Status is assigned SuccessfulMatch if such a matching tuple exists;
      otherwise, it is assigned FailureOnFirstMatch *)

Procedure NextMatch ($L$: Literal, var $\theta$: Substitution, var Status: MATCH_STATUS)

 (* Similar to FirstMatch, but finds the next matching tuple. In case of failure, Status is
      set to FailureOnNextMatch *)
```

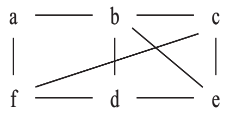
Figure 4.2: Matching procedures for *BJ_Instantiate*

```
**Function** *BackFromFailureOnFirstMatch* ($L$: literal, var $CSB$: Literal): Literal;
**begin** (* the first match on a new literal failed *)
    return *ClosestBinder*($L$, *var*($L$));
**end;**
```

```
**Function** *BackFromFailureOnNextMatch*($L$: Literal, *FailureSet* : Set of variables,
        var $CSB$: Literal): Literal;
**begin** (* failure looking for another match for $L$ *)
    $L'$ := *ClosestBinder*($L$,*FailureSet*);
    $L''$ := $\max_<\{L',CSB\}$;
    **if** $L'' = CSB$ **then**
        $CSB$ := *ClosestBinder*($L''$,*OutputVars*);
    return $L''$;
**end;**
```

```
**Function** *BackFromSolutionFound*($L$: Literal, var $CSB$: Literal,
        var *Status*: MATCH_STATUS) : Literal;
**begin**
    *Status* := FailureOnNextMatch; (* look for another solution *)
    $L'$ := *ClosestBinder*($L$,*OutputVars*);
    $CSB$ := *ClosestBinder*($L'$,*OutputVars*);
    return $L'$;
**end;**
```

Figure 4.3: Backjumping procedures for *BJ_Instantiate*



$$r: \quad colorable :\!-\ edge(A,B), edge(A,F), edge(B,D), edge(B,C), edge(B,E), edge(C,E), edge(C,F), edge(E,D), edge(D,F).$$

Figure 4.4: Encoding for *Constraint-3-Col*

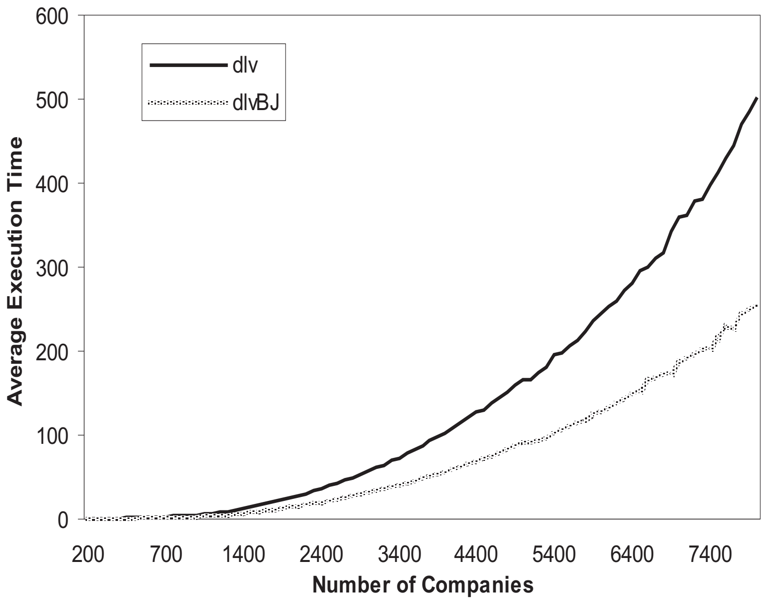
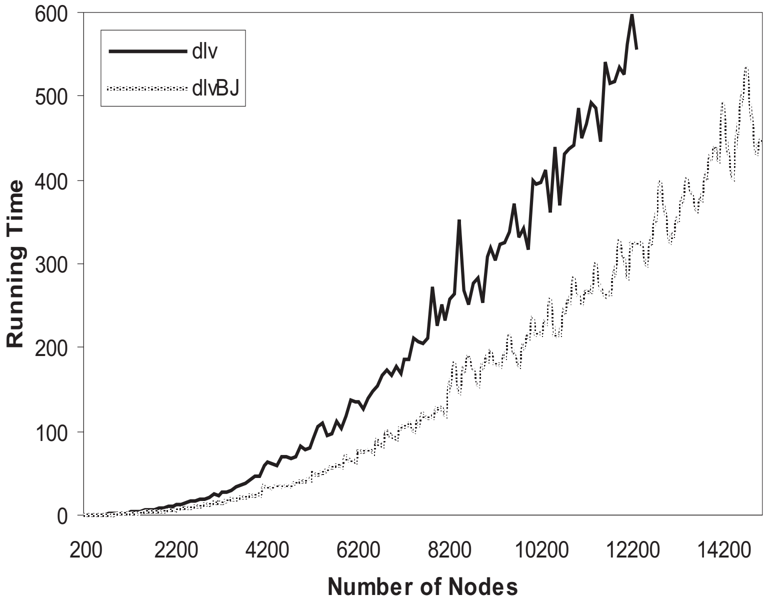


Figure 4.5: Results for Strategic Companies



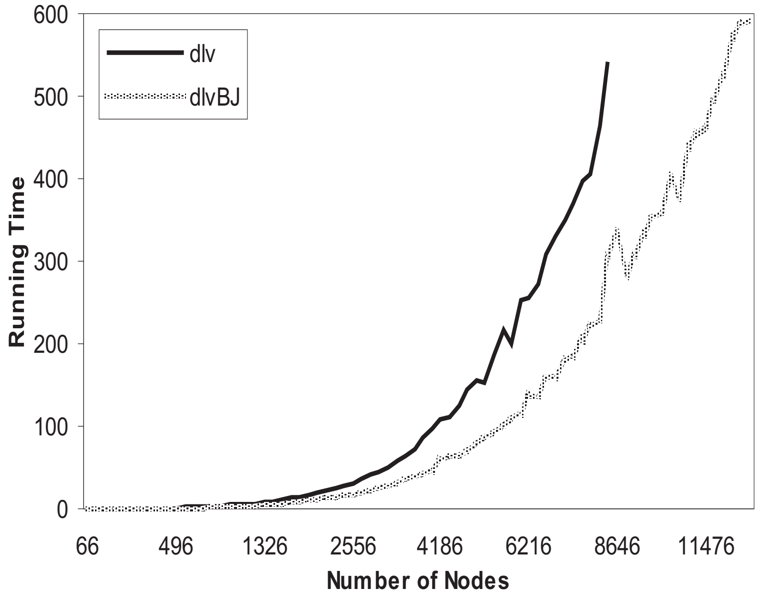


Figure 4.6: Results for 3-Colorability on ladder graphs (top) and simplex graphs (bottom)

Chapter 5

On Demand Indexing for the DLV Instantiator

In this chapter we propose to employ main-memory indexing techniques for enhancing the performance of the instantiation procedure of the ASP system DLV. In particular, we adapt a classical first argument indexing schema to our context, and propose an on demand indexing strategy where indexes are computed during the evaluation (and only if exploitable). Moreover, we define two heuristics which can be used for determining the most appropriate argument to be indexed, when more than one possibility exists. We discuss some key issues for its implementation into the DLV system and we then report the results of our experimentation activity on a number of benchmark problems.

5.1 Motivations

In chapter 3 we have seen that in ASP systems, the instantiation phase may be computationally expensive in some cases. Thus, having a good instantiator is crucial for the efficiency of the entire ASP system.

Moreover, some emerging application areas of ASP, like knowledge management and information integration, [1] where large amount of data are to be processed, make very evident the need of improving ASP instantiators significantly.

The DLV instantiator already incorporates a number of optimization techniques [Faber *et al.*, 1999; Leone *et al.*, 2001; Perri *et al.*, 2007] but, since ASP

[1]The application of ASP in these areas has been investigated also in the EU projects INFOMIX IST-2001-33570, and ICONS IST-2001-32429, and is profitably exploited by Exeura s.r.l., a spin-off of University of Calabria having precisely this mission.

applications grow in size, there is the need to efficiently handle larger and larger amount of data.

A critical issue for the efficiency of the instantiator is the retrieval of ground instances from the extensions of the predicates. Indeed, rule instantiation is essentially performed by evaluating the relational join of the positive body literals [2], and, as for join computation, in the absence of techniques for speeding-up the retrieval, the time spent in identifying candidate instances can dramatically affect the performances.

In this chapter, we face this issue and we propose the use of indexing techniques, that is techniques for the design and the implementation of suitable data structures that allow to efficiently access to large datasets.

5.2 Indexing Techniques for Rule Instantiation

A critical issue for the efficiency of the DLV instantiator is the task accomplished by function *Match* shown in Figure 3.3. As said in section 3.4, this function takes as input a literal L, its extension I_L and a partial substitution θ and tries to find a ground instance in I_L matching θ. This task, in the absence of techniques for speeding-up the retrieval of candidate instances, may be very expensive. Indeed, the size of I_L can be very large and thus, a simple approach based on linear search trough I_L leads to a drop in performance of the instantiator (also because *Match* is invoked very frequently during the instantiation).

In this Section, we describe two indexing strategies which can be exploited in order to facilitate the retrieval of instances thus allowing for a more efficient matching function.

5.2.1 First Argument

In the following we describe how the classical first argument indexing schema can be adapted to our context. Such indexing allows for efficiently performing the match of literals L whose first argument is *indexable*; an argument is said to be indexable if it is either a constant or a variable X such that $X \in \mathit{PreviousVars}(L)$ (thus, θ already contains a substitution for X).

Suitable hash structures are used for boosting the retrieval of ground instances according to values of their first term.

[2]Note that, since rules are safe [Ullman, 1989], the join of the positive body literals allows for instantiating all rule variables.

For each predicate p, its extension I_p is implemented by means of a list storing, according to lexicographical order, the ground instances of p. We associate to each extension I_p a sparse secondary index implemented by means of an hash map. More in detail, let $C \subseteq U_{\mathcal{P}}$ be the set of all the distinct constants appearing as first argument of some instance in I_p. An index is an hash map M_p that associates to each $c \in C$ (the key of the map) a pointer pt to an instance in I_p. In particular, pt identifies the first ground instance in I_p having c as first argument and thus, due to the lexicographical order of I_p, facilitates the retrieval of all ground instances in I_p with the same characteristic.

By using these structures, the match of a literal L whose first term is instantiated with a constant c can be performed as follows: first of all, we access to the index corresponding to L using c as key. Then, we simply follow the pointer associated to c in order to directly access the instances of I_p having c as first term, and try to extend θ by using, one after the other, such instances. Note that, since the index is implemented by means of an hash map, looking up the pointer pt by its key is efficient. In particular, the average case complexity of this operation is constant time.

Such indexing schema takes advantage from the lexicographical order of the predicate extensions for creating indexes whose size is in general smaller than the corresponding extensions thus limiting the space required to store the index. However, as the following example shows, it is not general, and it allows the use of indexes only in a small range of cases.

Example 5.1 *Consider the following program*

$$r_1: \ a(Z) \text{ :- } p(X,1,Y), q(X,Y,Z).$$
$$r_2: \ b(T,V) \text{ :- } r(U), q(T,U,V).$$

During the instantiation, the first argument indexing schema is exploited only once. Indeed, for r_1, the evaluation proceeds by matching first the literal $p(X,1,Y)$ whose first argument is not indexable, since $X \notin \mathit{PreviousVars}(p(X,1,Y))$. Then, the literal $q(X,Y,Z)$ is matched and the first argument index can be used, indeed $X \in \mathit{PreviousVars}(q(X,Y,Z))$. For the evaluation of r_2, it is easy to see that indexes can never be exploited.

5.2.2 On Demand Indexing

The first argument indexing schema described above allows for using sparse indexes with the advantage of limiting the memory consumption. However, it has the disadvantage of being not general and hence exploitable only in few cases.

In the following the on demand indexing strategy is described which allows for the efficient match of literals where a generic argument (not only the first one) is indexable. Such strategy is more general than the first argument indexing but requires more space for storing indexes, since the lexicographical order of the extensions can not be exploited and, an index must contain a pointer for each instance in the extension.

Let p be a predicate, I_p be the extension of p, a an indexable argument and x_a its position in the parameter list of p. Moreover, let $C \subseteq U_{\mathcal{P}}$ be the set of all the distinct constants appearing in some instance of I_p in position x_a. An index to I_p for the argument a is an hash multi map $MM_{p,a}$ that associates with each $c \in C$ (the key of the map) a number of pointers to I_p, one for each instance having c in position x.

In our on demand indexing strategy, the argument to be indexed is not predetermined but is established during the computation. More in detail, during the evaluation of a rule, when a match of a literal L has to be performed, an argument a is chosen among all the indexable arguments of L and the index for L corresponding to a is created (if it does not exist yet).

Thus, indexes are created only if really exploitable and, in two different moments of the evaluation, a predicate can be associated to two different indexes, depending on the argument which is more appropriate to be indexed. For instance, a predicate can appear in the body of two different rules, and the most convenient index to use could be not the same in the two cases.

Example 5.2 *Consider the program of example 5.1. While the first argument indexing schema allows for using of indexes only for the match of literal $q(X,Y,Z)$ in rule r_1, the on demand indexing schema has a better behavior, since indexes can be exploited in three cases. More in detail, the matches of $p(X,1,Y)$ in rule r_1 and of $q(T,U,V)$ of rule r_2 can be performed by using indexes on the second argument. Moreover, for the match of literal $q(X,Y,Z)$ of rule r_1, two indexes could be associated to q, for the first argument or the second one. The choice of the more appropriate index to be used can be made according to an heuristic, as described below.*

In the case of a literal L having more than one indexable argument, an heuristic is used in order to choose which is the one to be indexed. In this work, we experiment with two heuristics. The first one ($\mathbf{H_1}$) is very simple and consists in the selection of the first indexable argument (in a left to right order). The second one ($\mathbf{H_2}$) allows for a more refined choice and tries to select the indexable argument where it is more likely that few candidate instances will be retrieved.

More in detail, such a choice is made by taking into account the selectivity of each indexable argument a of L, that is the number of distinct constants for a in I_L. The heuristic selects the argument with the greatest selectivity, that is, the one whose selectivity better approximates the size of I_L.

Example 5.3 *In the previous example, we have seen that, for the match of literal* $q(X, Y, Z)$ *of rule* r_1*, the on demand indexing technique can choose among two indexes corresponding to two different arguments of* q *(first and second one). Suppose now that the size of the extensions of* p *and* q *are* 100 *and* 600*, respectively. Moreover, assume that the instances in* I_q *are the following:*

$$\begin{array}{l} q(a,1,1), q(a,2,2), \ldots q(a,100,100), \\ q(b,1,101), q(b,2,102), \ldots q(b,100,200), \\ \qquad\qquad\qquad\qquad \vdots \\ q(f,1,501), q(f,2,502), \ldots q(f,100,600) \end{array}$$

It is easy to see that the selectivities of the first and the second argument of q *are* 6 *and* 100*, respectively. Given these values, heuristic* H_2 *chooses the second one as argument to be indexed, thus suggesting a different choice w.r.t* H_1 *(which selects the first one). Importantly, according to such different choices, the cost of the matching operation of* q *varies notably. Indeed, using the index on the first argument (as* H_1 *suggests), we have that, for each instance of* p*,* 100 *candidate instances have to be considered for the match of* q*; thus, to compute the join among* p *and* q*,* 10000 *possible matchings have to be performed. On the contrary, the index on the second argument (as suggested by* H_2*) identifies, for each instance of* p*,* 6 *possible candidate instances for the match of* q*, thus the join among* p *and* q *is computed by considering only* 600 *possible matchings.*

Note that, in order to limit the memory usage, beside avoiding the creation of useless indexes, a structural analysis of the input program is done to identify possible indexes previously created but not exploitable in the next steps of the computation.

More in detail, since the instantiation of the program is performed according to the dependencies among components given by the dependency graph, for the evaluation of each single component, only a subset of the predicates occurring in the program is involved. Thus, if a predicate p is involved in the instantiation of a component C and it is not necessary for the evaluation of the components following C in the topological ordering, the eventual indexes associated with p can be destroyed as soon as C has been processed. Hence, only indexes which can be possibly exploited again during the computation are maintained.

5.3 Experimental Results

In order to check the impact of the proposed indexing techniques on the DLV instantiator, we carried out an experimentation activity on a number of benchmark problems, taken from different domains. For space limitation, we do not include the code of benchmark programs; however they can be retrieved, together with the binaries used for the experiments, from our web page: `http://www.mat.unical.it/catalano/thesisexperiments.zip`. Moreover, we give below a very short description of the problems.

5.3.1 Benchmark Programs

We have considered several problems whose encodings are significantly hard to instantiate. Some of them are known programs which have been already used in the evaluation of ASP instantiators ([Leone *et al.*, 2006; Gebser *et al.*, 2007a; Perri *et al.*, 2007]), some others are programs arising in practical applications of ASP.

InsuranceWorkflow. The goal is to emulate, by means of an ASP program, the execution of a workflow, in which each step constitutes a transformation to be applied to some data (in order to query for and/or extract implicit knowledge). Two problem instances were provided by the company EXEURA s.r.l. [exe,], which have been automatically generated by a software working on several American insurance data.

Scheduling. A scheduling problem for determining shift rotation of employees, ensuring appropriate days off for each employee and respecting other given constraints on the availability of some workers.

Cristal. Cristal (Cooperative Repositories & Information System for Tracking Assembly Lifecycle) is a deductive databases application that involves complex knowledge manipulations. The main purpose is to manage the gathering of production data during the ongoing construction of the Electromagnetic Calorimeter of the Compact Muon Solenoid, at the European Centre for Nuclear Research (CERN) [cri,].

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.

DocClass. The problem is to assign a document to one or more categories, based on its contents. In particular, the input data represent words or sequence of words appearing in the document (ngrams) and the document is classified according to the presence or the absence of given ngrams. The single problem instance was provided by EXEURA s.r.l. [exe,].

DataIntegration. A data integration problem. Given some tables containing discording data, find a repair where some key constraints are satisfied. The single problem instance used for these tests was originally defined within the EU project INFOMIX [Leone *et al.*, 2005].

Hilex. The problem consists in recognizing and extracting meaningful information from unstructured web documents. This is done by combining both syntactic and semantic information, through the use of domain ontologies. A preprocessor transforms the input documents into ASP facts, extraction rules are translated into ASP, and information extraction amounts to reasoning on an ASP program, which is executed by DLV. The single problem instance was provided by the company EXEURA s.r.l. [exe,].

Timetabling. The problem was considered of determining a timetable for some university lectures that 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. The five instances we considered were provided by the University of Calabria; they refer to different numbers of student groups.

GrammarBasedIE. This problem has been used at the First Answer Set Programming System Competition [Gebser *et al.*, 2007a]. It constitutes a part of a more complex application for recognizing and extracting meaningful information from unstructured Web documents. In particular, given a context free grammar, which specifies arithmetic equations, and a string, the problem is to determine whether the input string is an equation belonging to the language defined by the grammar and whether the equation holds. For the experiments, we used five different instances taken from the web page of the competition.

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.
Encoding The encoding considered for this problem is the one described in Section 4.3.
Data We considered five instances representing ladder graphs with increasing number of nodes.

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.
Encoding The encoding of this problem consists of one exit rule and a recursive one. We assume that A is represented by the binary relation arc(X, Y), where a fact arc(a, b) means that G contains an arc from a to b, i.e., $(a, b) \in A$; the set of nodes N is not explicitly represented, since the nodes appearing in the transitive closure are implicitly given by these facts. The following program then computes a relation reachable(X, Y) containing all facts reachable(a, b) such that b is reachable from a through the arcs of the input graph G:

$$reachable(X, Y) \text{ :- } arc(X, Y).$$
$$reachable(X, Y) \text{ :- } arc(X, U), reachable(U, Y).$$

Data We considered five different instances which have been used at the First Answer Set Programming System Competition [Gebser *et al.*, 2007a].

5.3.2 Compared Methods

We implemented the indexing strategies described in the previous Section in the DLV instantiator and we compared the resulting prototypes by using the above benchmark problems. In particular, the following instantiators were compared:

Program	noIndexes	1^{st}Arg	onDemand-H_1	onDemand-H_2	% gain
InsuranceWorkflow1	21.58	12.96	8.15	0.51	97%
InsuranceWorkflow2	102.47	22.97	10.34	2.61	97%
Scheduling	114.54	114.52	25.65	12.52	89%
Cristal	3.65	0.67	0.45	0.26	93%
Food	135.04	115.43	57.47	49.37	63%
DocClass	–	4.48	2.40	2.42	–
DataIntegration	293.68	293.75	3.71	3.72	99%
Hilex	16.54	8.23	3.66	3.52	79%

Table 5.1: Results for real problems (times in seconds)

Timetabling	noIndexes	1^{st}Arg	onDemand-H_1	onDemand-H_2	% gain
17 groups	187.28	187.75	35.94	14.39	92%
19 groups	287.18	288.46	37.19	15.11	95%
21 groups	319.66	318.98	51.51	16.54	95%
23 groups	334.06	334.02	73.85	18.45	94%
25 groups	431.99	409.61	97.14	20.15	95%

Table 5.2: Results for Timetabling (times in seconds)

- **noIndexes**: the DLV instantiator without any indexing technique;
- **1st Arg**: the DLV instantiator enhanced with first argument indexing; [3]
- **onDemand-H_1**: the DLV instantiator with on demand indexing where the first indexable argument is chosen;
- **onDemand-H_2**: the DLV instantiator with on demand indexing where the indexable argument with greatest selectivity is chosen.

All binaries were produced by using the GNU compiler GCC 4.1.2, and the experiments were performed on a dual processor Intel Xeon HT (single core) 3.60GHz machine, equipped with 3GB of RAM and running Debian Gnu Linux 2.6.

GrammarBased_IE	noIndexes	1^{st}Arg	onDemand-H_1	onDemand-H_2	% gain
GrammarBased_IE_1	4.49	2.98	0.98	0.98	78%
GrammarBased_IE_2	7.57	3.41	0.95	0.96	87%
GrammarBased_IE_3	7.89	4.09	1.16	1.15	85%
GrammarBased_IE_4	8.84	6.22	1.74	1.74	80%
GrammarBased_IE_5	9.50	7.26	2.15	2.14	77%

Table 5.3: Results for GrammarBased_IE (times in seconds)

3-Colorability	noIndexes	1^{st}Arg	onDemand-H_1	onDemand-H_2	% gain
15000 nodes	79.13	0.98	0.97	0.97	98%
20120 nodes	222.84	0.98	0.98	0.99	99%
32300 nodes	–	3.29	3.24	3.22	–
39900 nodes	–	2.13	2.11	2.12	–
40120 nodes	–	2.11	2.08	2.08	–

Table 5.4: Results for 3-Colorability (times in seconds)

5.3.3 Results and Discussion

Tables 5.1– 5.3 shows the results of our experiments. In each table, for each benchmark program P described in column 1, columns 2 –5 report the times employed to instantiate P by using the above binaries; column 6 reports the percentage gain obtained by onDemand-H_2 w.r.t noIndexes. All running times are expressed in seconds. The symbol '$-$' means that the instantiator did not terminate within 10 minutes.

The results confirm the intuition that the indexing techniques can be very useful for improving the efficiency of the DLV instantiator. Indeed, it is clear from the tables that, even a simple strategy, like the one exploited by 1^{st}Arg, allows for outperforming the instantiator noIndexes in many cases. In particular, the first argument indexing gives very relevant improvements in some benchmarks, as, for instance, DocClass and three of the instances of 3-Colorability, where noIndexes does not terminate within ten minutes, while 1^{st}Arg takes few seconds. However, there are also some cases in which the speed-up introduced by 1^{st}Arg is not so considerable, or it is not present at all. Consider for instance, Scheduling and Data Integration and the instances of Timetabling where noIndexes and 1^{st}Arg

[3]This version of the DLV instantiator coincides with the one of the official release October 11th 2007.

Reachability	noIndexes	1^{st}Arg	onDemand-H_1	onDemand-H_2	% gain
Reachability_13	71.83	66.39	0.95	0.98	99%
Reachability_14	367.60	339.63	2.33	2.31	99%
Reachability_15	–	–	5.47	5.49	–
Reachability_16	–	–	12.61	12.58	–
Reachability_18	–	–	68.05	68.68	–

Table 5.5: Results for Reachability (times in seconds)

perform very similarly, or the two solved instances of Reachability where the gain is about 7%, and Food where the gain is about 15%. The reason of such different behavior of 1^{st}Arg on these problems is that the speed-up reflects how intensively first argument indexing can be used for a given benchmark. More precisely, the performance gain is low when the encodings are such that first argument indexing is exploitable only for the match of few literals.

The situation changes when looking at the results of the on demand indexing technique. Indeed, for all the tested benchmarks, the speed-up introduced by this technique is really impressive. Moreover, it is clear from the tables that the instantiator exploiting H_2 as heuristic behaves quite better than the one exploiting H_1. Indeed, OnDemand-H_1 allows for notable improvements in many cases with difference of even 2 orders of magnitude w.r.t. noIndexes (as, for instance, DataIntegration, Reachability). Moreover, it is able to solve all the problem instances within the allowed time limit. However, these considerations hold also for OnDemand-H_2; indeed, either it exhibits the same behavior of OnDemand-H_1 or perform better, as, for instance, in Timetabling, InsuranceWorkflow and other real problems. This shows that performance improvements strongly depend on the "quality" of the used index. Intuitively, when to an entry in the index corresponds an high number of candidate instances (close to the size of the extension), then indexing may not bring great benefits.

Summarizing, the tested indexing techniques allow to improve performance of the instantiator but the on demand strategy is applicable in a wider range of cases w.r.t the first argument one and gives relevant speed-ups especially when combined with an accurate choice of the argument to be indexed.

5.4 Related Work

Indexing methods have been originally introduced in the database field for improving the speed of the operations in a table Indeed in this field many structures have been already proposed. For instance, a common and important kind of index is the B-tree. However they are designed to be stored in mass-memory, whereas the DLV instantiator works in main-memory, thus indexing has to be designed according to more strict memory limits.

Moreover, indexing techniques are now profitably used also in the logic programming area [Demoen *et al.*, 1989; Carlsson, 1987]. Indeed, effective indexing has become an integral component of high performance declarative programming systems. Almost all the Prolog implementations support indexing on the main functor symbol of the first argument of predicates. Several systems have generalized the first argument indexing. For example, BIM_Prolog [BIM, 1990] can index on any argument when given appropriate declarations. SEPIA [SEP, 1990] incorporates heuristics to decide which predicate arguments are important for deterministic selection. Other system, like XSB [Rao *et al.*, 1997], SWI-Prolog [Wielemaker, 1997 2003], support more sophisticated indexing schemata.

The reason for developing several different indexing techniques is that the conditions under which data have to be retrieved differ from context to context. In addition, if on the one hand indexing allows to significantly speedup the retrieval, on the other hand it could lead to a considerable memory consumption and hence a compromise between these two factors has to be made. Thus, there is no an optimal indexing technique for all the applications, rather each application may require to develop its own specialized technique.

Chapter 6

DLV Instantiator vs. Other Instantiators

To check the strength of the enhanced DLV instantiator, which includes the methods described in the previous Chapters, we carried out a deep experimental analysis for comparing it w.r.t. others two popular instantiators, namely Lparse [Niemelä and Simons, 1997; Syrjänen, 2002] and *GrinGo* [Gebser *et al.*, 2007b]. In this Chapter, we first briefly describe the compared instantiators, and then discuss the results of the experiments.

6.1 Overview of the Compared Instantiators

To assess the effectiveness of all our proposed methods, we have compared the enhanced DLV instantiator also with Lparse, and *GrinGo*. These instantiators accept different classes of input programs, and follow different strategies for the computation. As a consequence, the size of the respective instantiations, as well as the time needed for generating them, may differ significantly.

6.1.1 Lparse

The instantiation module of the Smodels system is a separate application called Lparse, which preprocesses the programs which are then evaluated by Smodels [Simons *et al.*, 2002]. Lparse accepts logic programs respecting *domain restrictions*. This condition enforces each rule's variable to occur in a positive body literal, called domain literal, which (i) is not mutually recursive with the head, and (ii) is not unstratified nor (transitively) depends on an unstratified literal (see

[Syrjänen, 2002] for details). For instance, the simple program $\mathcal{P}_{Inst}$

$$\begin{array}{ll} a(1) \mathrel{:-} not\, a(2). & b(2) \mathrel{:-} not\, b(3). \\ a(2) \mathrel{:-} not\, a(1). & b(3) \mathrel{:-} not\, b(2). \\ \multicolumn{2}{c}{r:\ p(X) \mathrel{:-} a(X), b(X).} \end{array}$$

is not accepted by Lparse. To instantiate a rule r, Lparse employs a nested loop that scans the extensions of the domain predicates occurring in the body of r, and generates its ground instances, accordingly. Thus, it is a simple instantiation method, and runs very fast, at least for applications where there are few domains to scan or they have small extensions (like, e.g., 3-Colorability). However, Lparse may generate several useless rules inasmuch as they may contain non-domain body literals that are not derivable by the program. Our instantiator, instead, incorporates several database optimization techniques, and builds the domains dynamically. Consequently, the instantiation generated by DLV is generally a subset of that generated by Lparse.

Example 6.1 *Consider for instance the following program:*

$$\begin{array}{ll} f_1: & f(1). \\ f_2: & h(1..10). \\ r_1: & a(X) \mathrel{:-} not\, b(X), f(X). \\ r_2: & b(X) \mathrel{:-} a(X), h(X). \end{array}$$

The istantiation produced by Lparse is:

$$\begin{array}{l} f(1).\quad h(1).\quad h(2).\quad \ldots\quad h(9).\quad h(10). \\ a(1) \mathrel{:-} not\, b(1). \\ b(1) \mathrel{:-} a(1). \\ b(2) \mathrel{:-} a(2). \\ \quad\vdots \\ b(9) \mathrel{:-} a(9). \\ b(10) \mathrel{:-} a(10). \end{array}$$

The DLV *instantiator produces a subset of this, that considerably smaller:*

$$\begin{array}{l} f(1).\qquad h(1).\quad h(2).\quad \ldots\quad h(9).\quad h(10). \\ a(1) \mathrel{:-} not\, b(1). \\ b(1) \mathrel{:-} a(1). \end{array}$$

Note that for applications with few domains with small extensions the strategy of Lparse may be computationally less expensive than the instantiation method adopted by DLV.

On the contrary, in case of applications where the size of domains extensions are very large (for instance, for real world applications like Grammar Based Information Extraction (see Section 4.3), or Reachability (see Section 5.3 and other Deductive Database Applications), Lparse may take significantly more time and produce a (uselessly) larger instantiation than DLV.

Notably, the backjumping technique presented in Chapter 4 allows in some cases to further reduce the size of the instantiation generated by DLV (as well as the execution time) with respect to that of Lparse. Indeed, even in case of applications where all the body literals are in fact domain literals (like, e.g., Constraint 3-Colorability) DLV may generate an instantiation considerably smaller than Lparse (e.g., only one rule versus thousand of rules for Constraint 3-Colorability) because the new backjumping algorithm avoids the generation of redundant rules.

6.1.2 *GrinGo*

The recent instantiator *GrinGo* combines the instantiation techniques of Lparse and DLV. Similarly to Lparse, it accepts domain restricted programs. However, their notion of domain literal is an extension of that of Lparse (see [Gebser *et al.*, 2007b], for details). In particular, *GrinGo* can handle all the programs accepted by Lparse while the vice versa does not hold. For instance, the program

$$\begin{array}{l} a(X) \,\text{:-}\, b(X), c(X). \\ b(X) \,\text{:-}\, a(X). \\ c(1). \quad b(1). \end{array}$$

is accepted by *GrinGo* while it is not handled by Lparse. The same holds for the program $\mathcal{P}_{Inst}$ in the previous Section. However, the following program, encoding the reachability on a graph, is not accepted by *GrinGo*

$$\begin{array}{l} reachable(X,Y) \,\text{:-}\, arc(X,Y). \\ reachable(X,Y) \,\text{:-}\, arc(X,U), reachable(U,Y). \end{array}$$

The extended definition of domain predicates allows *GrinGo* to have, in general, a better behavior than Lparse, but does not solve the above mentioned problems, related to the use of domains for the instantiation. Indeed, similarly to Lparse, *GrinGo* may not behave well in case of real world applications, like Grammar Based Information Extraction, where many domains are required and their extensions may be huge.

Notably, the rule instantiation procedure of *GrinGo* already exploits the backjumping technique described in Chapter 4 with very good results, and performances similar to DLV. The good performance of *GrinGo* in some of the conducted experiments are, indeed, due to the use of our technique.

6.2 Benchmark Problems and Data

We have selected programs where the instantiation process is hard, and it takes a relevant part of the entire computation. In particular, we evaluated the systems on the following problems:

- Ramsey Numbers (*Ramsey*)
- Reachability (*Reachability*)
- 3-Colorability (*3-Col*)
- Grammar Based Information Extraction (*GrammarBased_IE*)
- Constraint 3-Colorability (*Constraint-3-Col*)

We next provide a short description of the problems and the data used for the experiments. All benchmark instances and encodings as well as the DLV binaries used for our experiments are available at the Web page `http://www.mat.unical.it/catalano/thesisexperiments.zip`.

Note that, since the considered grounders take different input languages, it is not possible to compare their performance using the same encodings. The encodings are similar i.e., as far as possible, they are straightforward translations of each other, except that domain atoms are added for Lparse and *GrinGo*, when needed.

Ramsey Numbers (*Ramsey*) The Ramsey number $ramsey(k, m)$ is the least integer n such that, no matter how we color the edges of the complete undirected graph (clique) with n nodes 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). Ramsey numbers exist for all pairs of positive integers k and m [Radziszowski, 1994].
Encoding The encoding considered for this problem is the one described in Section 1.3.2.
Data We considered the problem of deciding, for varying k, m, and n, if n is the Ramsey number $ramsey(k, m)$.

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.
Encoding The encoding considered for this problem is the one described in Section 5.3.
Data The input graphs for Reachability were generated by means of the Stanford GraphBase [Knuth, 1994], using the function *random_graph(#nodes, #arcs, 0,0,0,0,0,0,0,0)* with a ratio of 3:1 between #arcs and #nodes (nodes ($50 \leq n \leq 12000$).

3-Colorability (*3-Col*) The 3-Colorability problem consists in the assignment of three colors to the nodes of a graph in such a way that adjacent nodes have different colors. This problem is known to be NP-complete.
Encoding The encoding considered for this problem is the one described in Section 4.3.
Data We used ladder graphs with n levels, i.e., $2 * n$ nodes ($100 \leq n \leq 30000$) and simplex graphs generated with the Stanford GraphBase [Knuth, 1994], using the function $simplex(n, n, -2, 0, 0, 0, 0)$, $10 \leq n \leq 400$ (i.e. $(n + 1) * (n + 2)/2$ nodes).

Grammar Based Information Extraction (*GrammarBased_IE*) This problem has been used at the First Answer Set Programming System Competition `http://asparagus.cs.uni-potsdam.de/contest/`. It constitutes a part of a more complex application for recognizing and extracting meaningful information from unstructured Web documents. In particular, given a context free grammar, which specifies arithmetic equations, and a string, the problem is to determine whether the input string is an equation belonging to the language defined by the grammar and whether the equation holds. The encoding of this problem is quite long and involved, thus we do not report it here. Concerning the data, we have used three different instances $in\_001.asp$, $in\_020.asp$, $in\_047.asp$, $in\_071.asp$, and $in\_082.asp$. Both the encoding and the instances can be retrieved from the web page of the competition.

Constraint 3-Colorability (*Constraint-3-Col*) In addition to the encoding described above, we considered for the 3-Colorability problem an alternative representation in a constraint satisfaction style.

Encoding The encoding considered for this problem is the one described in Section 4.3.

Data We considered random graphs generated with the Stanford GraphBase, using the function $ramdom\_graph(\#nodes, \#edges, 0, 0, 1, 0, 0, 0, 0, seed)$ for different values of $\#nodes$ and $\#edges$.

6.3 Results and Discussions

The experimental analysis was performed by comparing the following instantiators:

- *dlvOld*
- *dlvNew*
- Lparse
- *GrinGo*.

dlvOld is the DLV instantiator without the techniques described in this thesis while *dlvNew* is the DLV instantiator which integrates the backjumping algorithm and the on demand indexing technique described in Chapters 4 and 5, respectively. Both the binaries, *dlvOld* and *dlvNew*, were produced by using the GNU compiler GCC 4.1.2, and the experiments were performed on a dual processor Intel Xeon HT (single core) 3.60GHz machine, equipped with 3GB of RAM and running Debian Gnu Linux 2.6. For the other instantiators we use in our tests, their current versions at the time of this writing, Lparse 1.0.1 and *GrinGo* 1.0.0.

The results of the experimental activities on the benchmark problems presented above are summarized in Figures 6.1, 6.2 and 6.3 and in Tables 6.1, 6.2 and 6.3. The tables report in seconds the instantiation times of the four tested instantiators. For every instance, we allowed a maximum running (real-)time of 600 seconds. In the figures, the line corresponding to an instantiator stops whenever a problem instance was not solved within the allowed time limit or when a system is not able to solve it because of errors during the execution; in the tables the information is represented by means of the symbol '-'.

On the overall, the results show that the performance of the old DLV instantiator are in some cases comparable to those of Lparse and *GrinGo* and in some cases even worst. On the contrary, the enhanced instantiator *dlvNew* outperforms *dlvOld* in all considered programs and, in addition, behaves much better than Lparse and

Ramsey	dlvOld	dlvNew	*Lparse*	GrinGo
ramsey(3,3)$\neq$ 5	0,00	0,00	0,00	0,00
ramsey(3,3)$\neq$ 6	0,00	0,00	0,00	0,00
ramsey(3,4)$\neq$8	0,00	0,00	0,05	0,00
ramsey(3,4)$\neq$9	0,00	0,00	0,08	0,00
ramsey(3,5)$\neq$12	0,07	0,02	4,12	0,04
ramsey(3,5)$\neq$13	0,13	0,04	6,29	0,06
ramsey(3,5)$\neq$14	0,19	0,06	9,17	0,10
ramsey(3,6)$\neq$16	1,06	0,37	-	0,63
ramsey(3,6)$\neq$17	2,72	0,57	-	0,97
ramsey(3,6)$\neq$18	4,51	0,80	-	1,44
ramsey(3,7)$\neq$15	2,91	0,53	-	0,65
ramsey(3,7)$\neq$16	3,43	0,86	-	1,15
ramsey(3,7)$\neq$17	6,36	1,41	-	1,92
ramsey(3,7)$\neq$18	12,21	1,88	-	3,20
ramsey(3,7)$\neq$19	19,68	2,93	-	5,03
ramsey(3,7)$\neq$20	27,34	5,32	-	7,52
ramsey(3,7)$\neq$21	50,68	6,56	-	11,54
ramsey(3,7)$\neq$22	67,86	10,45	-	16,81
ramsey(3,7)$\neq$/23	164,68	17,00	-	24,11
ramsey(4,4)$\neq$13	0,08	0,02	0,62	0,04
ramsey(4,4)$\neq$14	0,12	0,03	0,83	0,06
ramsey(4,4)$\neq$15	0,18	0,05	1,11	0,08
ramsey(4,4)$\neq$16	0,29	0,06	1,40	0,11
ramsey(4,4)$\neq$17	0,40	0,08	1,80	0,14
ramsey(4,4)$\neq$18	0,48	0,09	2,30	0,18
ramsey(4,5)$\neq$17	1,13	0,24	25,58	0,36
ramsey(4,5)$\neq$18	1,15	0,28	33,64	0,50
ramsey(4,5)$\neq$19	1,65	0,37	44,17	0,67
ramsey(4,5)$\neq$20	3,05	0,56	57,54	0,88
ramsey(4,5)$\neq$21	3,28	0,64	73,20	1,14
ramsey(4,5)$\neq$22	6,67	0,92	92,86	1,47
ramsey(4,5)$\neq$23	6,89	1,08	114,17	1,86
ramsey(4,5)$\neq$24	9,59	1,38	139,86	2,45

Table 6.1: Results for *Ramsey* (times in seconds)

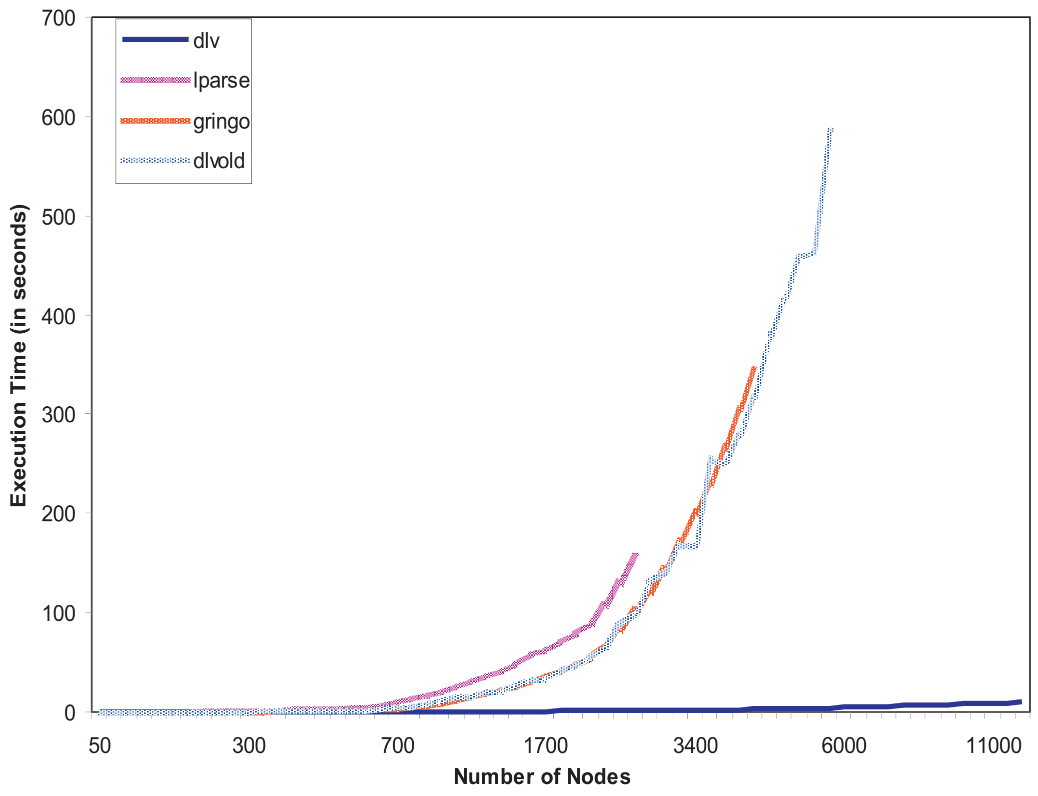


Figure 6.1: Results for Reachability

GrinGo. More detailed considerations follow for each tested programs.

Ramsey Numbers (Table 6.1). In this problem, the instantiator with the worst behaviour is Lparse which takes more time of *dlvOld* and in many cases cannot solve the instance at hand because of some memory problem. The instantiator *GrinGo* instead exhibits good performance on all the instances, outperforming the old DLV instantiator. However, the proposed techniques in this thesis allow to *dlvNew* for showing the best performance, with very low execution times even for the hardest instances (Ramsey 3,7).

Reachability (Figure 6.1). The results of Reachability show similar performance for Lparse, *GrinGo* and *dlvOld*, but *dlvOld* is able to solve more instances (up to graphs with 5500 nodes) than the other two systems which stop to graphs with

GrammarBased_IE	*dlvOld*	*dlvNew*	Lparse	*GrinGo*
GrammarBased_IE_1	36,77	2,35	–	–
GrammarBased_IE_2	16,82	1,09	–	–
GrammarBased_IE_3	35,09	2,01	–	–
GrammarBased_IE_4	36,13	1,15	–	–
GrammarBased_IE_5	35,2	1,33	–	–

Table 6.2: Results for GrammarBased_IE (times in seconds)

Constraint-3-Col	*dlvOld*	*dlvNew*	Lparse	*GrinGo*
35 nodes, 35 edges	0,01	0,01	–	0.02
35 nodes, 45 edges	0,01	0,01	449,54	0.03
40 nodes, 40 edges	0,02	0,02	319,4	0.01
40 nodes, 50 edges	0,01	0,01	–	0.01
45 nodes, 45 edges	0,01	0,01	–	0.01
45 nodes, 55 edges	0,02	0,02	–	0.02
50 nodes, 50 edges	0,01	0,01	–	0.01
50 nodes, 60 edges	0,01	0,01	–	0.16
60 nodes, 60 edges	0,02	0,02	–	0.01
60 nodes, 70 edges	0,01	0,01	–	0.02

Table 6.3: Results for *Constraint-3-Col* (times in seconds)

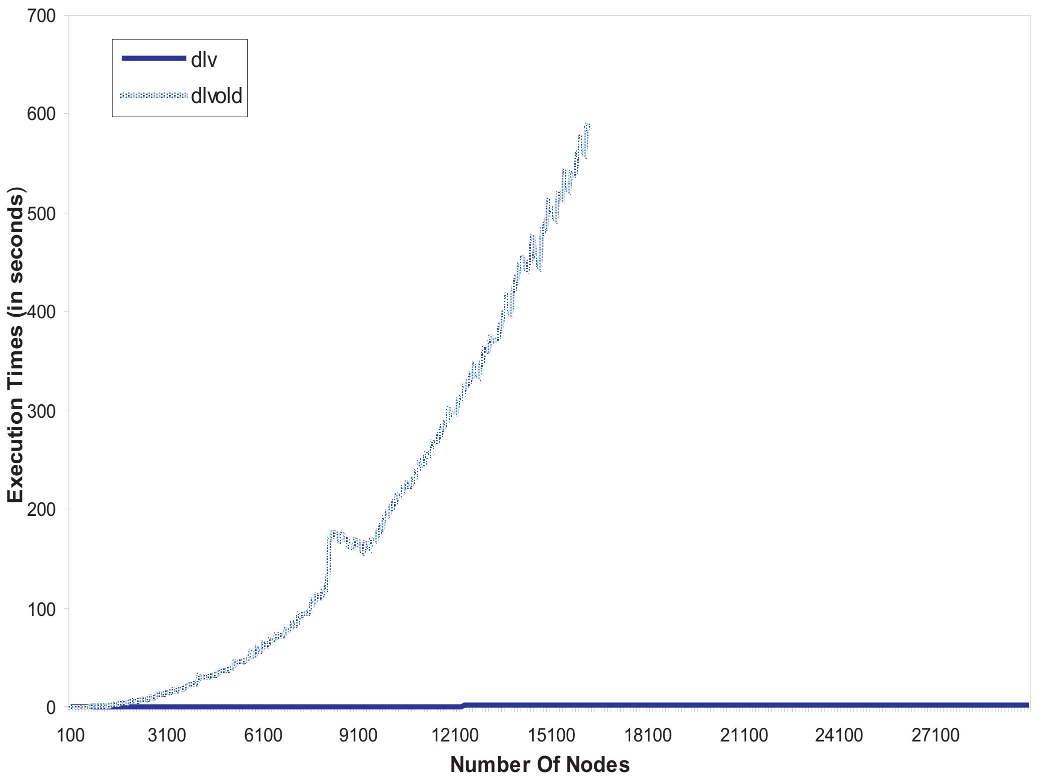
2600 and 4200 nodes, respectively. The negative performances of Lparse and *GrinGo* are due to the domain predicates that have to be added to the encoding of this problem. *dlvNew*, even in this case, gives the best performance, allowing for solving all the considered instances with execution times smaller than 10 seconds.

Constraint 3-Colorability (Table 6.3). This problem highlights the power of the backjumping technique we introduced. Indeed, while *dlvOld* and Lparse are not able to solve all the instances within the allowed time limit, *dlvNew* and *GrinGo* solve all the instances in less than 1 second. The reason is that, for this problem, *dlvOld* and Lparse generate an instantiation consisting of thousands of useless rules while, the other two instantiators which both exploits our backjumping technique, generate only one ground rule.

Grammar Based Information Extraction (Table 6.2). For this real problem, involving an huge input, predicates domain have a tremendous impact on the performance of Lparse and *GrinGo* that are not able to solve any instances. On the contrary both *dlvOld* and *dlvNew* solve all instances and *dlvNew* shows a very nice speed-up. Thus, the DLV instantiator seems to be the most appropriate to handle real-world applications and techniques we proposed make it even more competitive.

3-Colorability (Tables 6.2 and 6.3). The results of the experiments on this problem (both for ladder and simplex graphs) are reported in two pictures, one comparing *dlvOld* w.r.t. *dlvNew* and one comparing *dlvNew* w.r.t. Lparse and *GrinGo*. This is because, *dlvOld* shows performance much worst than the other three systems, while the enhanced DLV instantiator, behave significantly better than it. Moreover, *dlvNew* and Lparse exhibit better performance than *GrinGo* on both ladder and simplex graphs. However, pictures 6.2 and 6.3 show that the line of Lparse stops before that of the other systems; the reason is that it gives a segmentation fault on the unsolved instances.

In summary, the performance of the DLV instantiator have been notably improved by the introduction of our techniques, which make it much more competitive than the other instantiators both on classical and real-world problems.



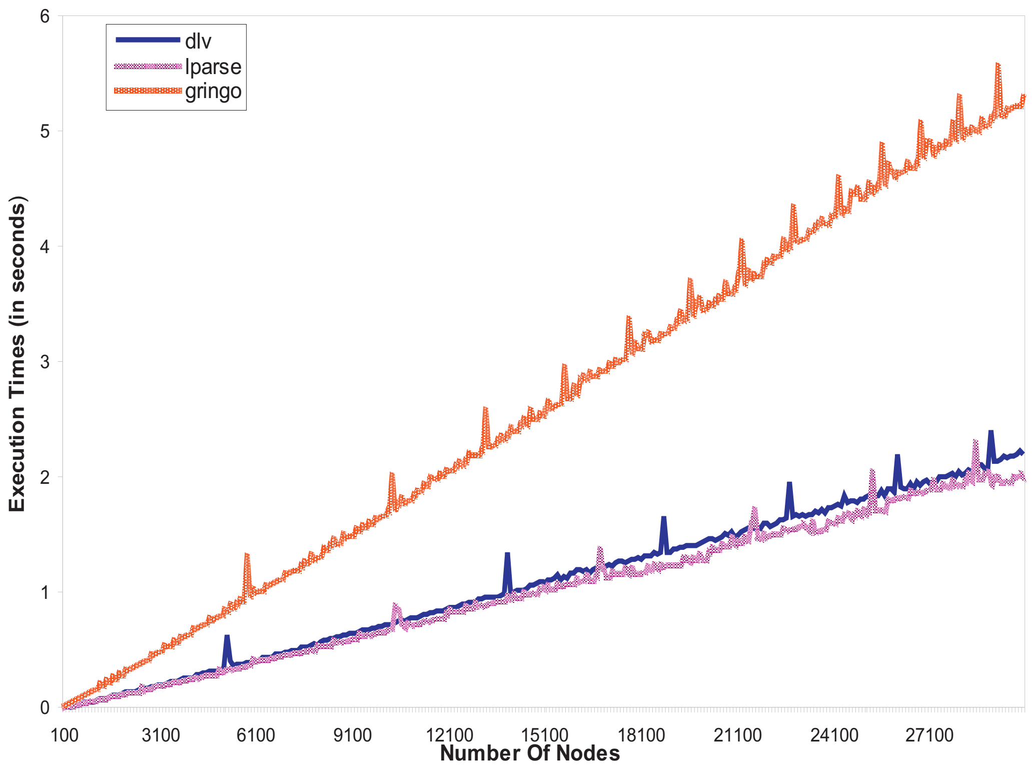
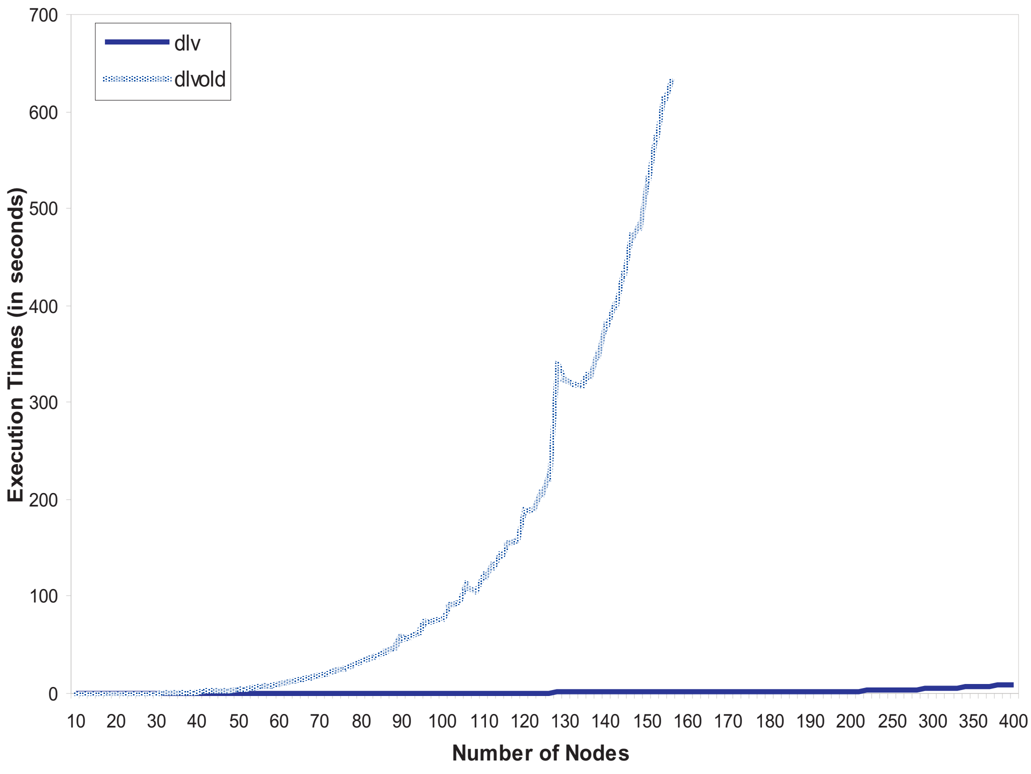


Figure 6.2: Results for 3-Colorability on ladder graphs. *dlvNew* vs. *dlvOld* (top) and *dlvNew* vs. Lparse and *GrinGo* (bottom)



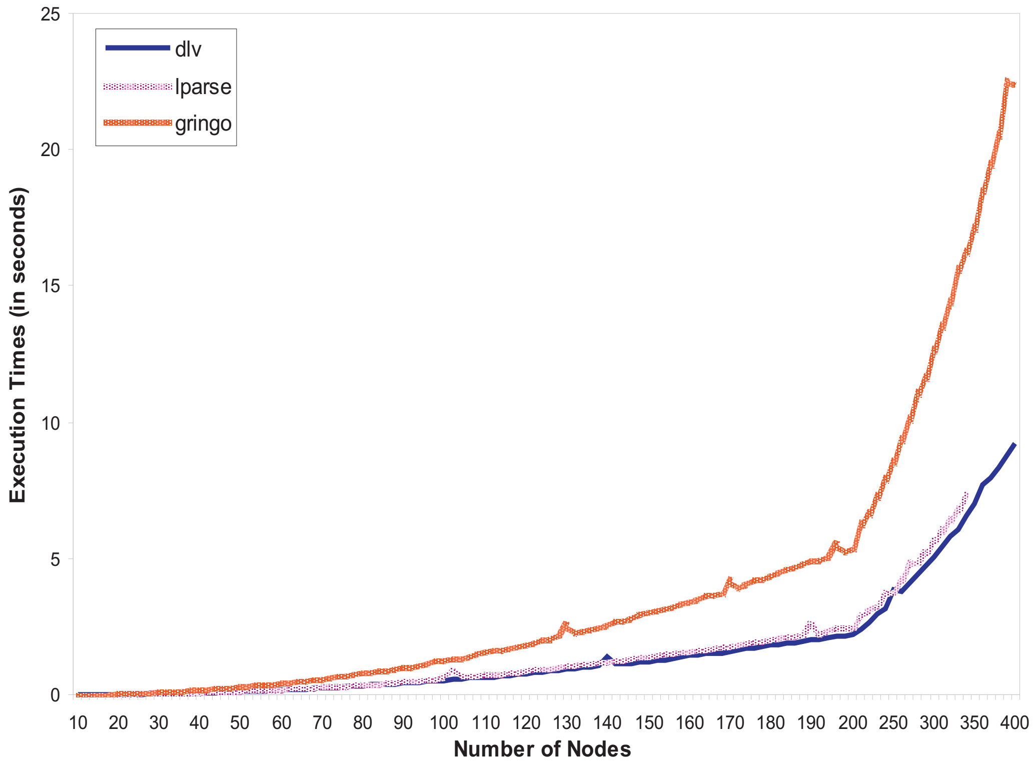


Figure 6.3: Results for 3-Colorability on simplex graphs. (*dlvNew* vs. *dlvOld* (top) and *dlvNew* vs. Lparse and *GrinGo* (bottom)

Conclusions and Ongoing Work

Disjunctive Logic Programming, (first proposed by Jack Minker at the beginning of the Eighties), is nowadays widely recognized as a valuable tool for knowledge representation and commonsense reasoning. Moreover, the recent availability of some efficient implementations, allowed for the application of DLP to real-world problems in emerging areas like knowledge management and information extraction/integration. The application of DLP to real-world problems, where large amount of input data have often to be processed, has evidenced the strong need to improve DLP instantiators.

In this thesis, we have presented methods for improving the performance of the instantiator of DLV – the state-of-the-art DLP system. In particular, we have designed a structure-based backjumping algorithm for rule instantiation which reduces the instantiation time, and avoids the generation of many superfluous instances (thus reducing also the size of the generated instantiation), while fully preserving the semantics of the program. We have implemented the proposed method in DLV and we have carried out an experimental activity on an ample collection of benchmark programs, which has fully confirmed the effectiveness of our method.

Moreover, we have investigated the use of indexes for optimizing the rule instantiation process of DLV. In particular, we have experimented with a classical first argument indexing schema adapted to our context and proposed an on demand indexing strategy where indexes are computed during the evaluation and the argument to be indexed is chosen according to a heuristic.We define two heuristics which can be used for determining the most appropriate argument to be indexed, when more than one possibility exists.

We have implemented such strategies in the instantiator of DLV and we performed a deep experimental analysis. The results of the experiments are very positive and confirm that the use of indexes causes the instantiation stage to achieve noticeable improvements. Moreover, the on demand indexing schema gives better results w.r.t the classical first argument schema in a wider range of cases and performance improve notably when a good choice of the argument to be indexed is made.

Currently, we are investigating the relations among the body ordering criterion exploited in DLV and the use of indexes. Indeed, it is easy to see that, the body ordering may have a strong impact on the use of indexes; each of the techniques described above chooses for a literal L the argument to be indexed among a set of indexable arguments, and such set depends on the position that L has in the body and, thus, by the ordering algorithm used. Hence, a different ordering criterion may lead to make different choices and, so, to considerably influence the execution times. Hence, the ordering criterion could be modified in order to take into account indexes availability. However, it is well known that the instantiation time of a rule strongly depends on the order of evaluation of literals [Leone *et al.*, 2001; Garcia-Molina *et al.*, 2000], thus a naive ordering could have a negative impact on the instantiator performance, also overshadowing the gains brought by the indexes usage. Therefore, a clever ordering has to be conceived which allows a better use of indexes but without ignoring the principles which the current method is based on and whose effectiveness has already been assessed[Leone *et al.*, 2001]. The design of the new ordering criterion is the subject of a future work.

Part of the work presented in this thesis is already included in the current DLV distribution, and can be retrieved from DLV homepage `www.dlvsystem.com`. Part of results presented in this thesis has been published in [Perri *et al.*, 2007; Catalano *et al.*, 2008; 2006].

Bibliography

[Apt *et al.*, 1988] 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.

[Arieli *et al.*, 2004] Ofer Arieli, Marc Denecker, Bert Van Nuffelen, and Maurice Bruynooghe. Database repair by signed formulae. In Dietmar Seipel and Jose Maria Turull Torres, editors, *Foundations of Information and Knowledge Systems, Third International Symposium (FoIKS 2004)*, volume 2942 of *LNCS*, pages 14–30. Springer, February 2004.

[Babovich, since 2002] Yulia Babovich. Cmodels homepage, since 2002. `http://www.cs.utexas.edu/users/tag/cmodels.html`.

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

[Baral, 2002] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2002.

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

[Ben-Eliyahu and Palopoli, 1994] Rachel Ben-Eliyahu and Luigi Palopoli. Reasoning with Minimal Models: Efficient Algorithms and Applications. In *Proceedings Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR-94)*, pages 39–50, 1994.

[BIM, 1990] BIM_Prolog Version 2.5, 1990. BIM, Everberg Belgium.

[Bruynooghe and Pereira, 1984] Maurice Bruynooghe and Luis Moniz Pereira. Deduction revision by intelligent backtracking. In J. Campbell, editor, *Implementations of Prolog*, pages 194–215. Ellis Horwood, 1984.

[Buccafurri *et al.*, 2000] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Enhancing Disjunctive Datalog by Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):845–860, 2000.

[Buccafurri *et al.*, 2002] Francesco Buccafurri, Wolfgang Faber, and Nicola Leone. Disjunctive Logic Programs with Inheritance. *Journal of the Theory and Practice of Logic Programming*, 2(3), May 2002.

[Cadoli *et al.*, 1997] 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.

[Carlsson, 1987] M. Carlsson. Freeze, Indexing, and other implementation issues in the WAM. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 40–58. MIT Press, 1987.

[Catalano *et al.*, 2006] Gelsomina Catalano, Nicola Leone, and Simona Perri. Indexing techniques for the dlv instantiator. In *Proceedings of the 22th Convegno Italiano di Logica Computazionale (CILC '07)*, Messina, Italy, 2006.

[Catalano *et al.*, 2008] Gelsomina Catalano, Nicola Leone, and Simona Perri. On demand indexing techniques for the dlv instantiator. In To appear, editor, *Proceedings of the Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'08)*, Udine, Italy, 2008.

[Chen and van Beek, 2001] Xinguang Chen and Peter van Beek. Conflict-Directed Backjumping Revisited. *Journal of Artificial Intelligence Research*, 14:53–81, 2001.

[cri,] `CRISTAL` project, homepage `http://proj-cristal.web.cern.ch/`.

[Dechter, 1990] Rina Dechter. Enhancement schemes fo constraint processing:backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41, 1990.

[Demoen *et al.*, 1989] B. Demoen, A. Mariën, and A. Callebaut. Indexing in Prolog. In *Proceedings of the North American Conference on Logic Programming*, pages 1001–1012. MIT Press, 1989.

[Dix *et al.*, 2003] Jürgen Dix, Thomas Eiter, Michael Fink, Axel Polleres, and Yingqian Zhang. Monitoring Agents using Declarative Planning. In *Proceedings of the 26th German Conference on Artificial Intelligence (KI2003)*, number 2821 in Lecture Notes in Computer Science, pages 646–660. Springer, September 2003.

[Eiter and Gottlob, 1995] Thomas Eiter and Georg Gottlob. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Annals of Mathematics and Artificial Intelligence*, 15(3/4):289–323, 1995.

[Eiter *et al.*, 1997a] Thomas Eiter, Georg Gottlob, and Nicola Leone. Abduction from Logic Programs: Semantics and Complexity. *Theoretical Computer Science*, 189(1–2):129–177, December 1997.

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

[Eiter *et al.*, 1997c] Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. A Deductive System for Nonmonotonic Reasoning. In Jürgen Dix and Ulrich Furbach and Anil Nerode, editor, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, number 1265 in Lecture Notes in AI (LNAI), pages 363–374, Dagstuhl, Germany, July 1997. Springer.

[Eiter *et al.*, 1998] Thomas Eiter, Nicola Leone, and Domenico Saccá. Expressive Power and Complexity of Partial Models for Disjunctive Deductive Databases. *Theoretical Computer Science*, 206(1–2):181–218, October 1998.

[Eiter *et al.*, 1999] Thomas Eiter, Wolfgang Faber, Georg Gottlob, Christoph Koch, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and F. Scarcello. The DLV System. In Jack Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC*, College Park, Maryland, June 1999. Computer Science Department, University of Maryland. Workshop Notes.

[Eiter *et al.*, 2000] 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.

[Eiter *et al.*, 2003a] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Computing Preferred Answer Sets by Meta-Interpretation in Answer Set Programming. *Journal of the Theory and Practice of Logic Programming*, 3:463–498, July/September 2003.

[Eiter *et al.*, 2003b] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A Logic Programming Approach to Knowledge-State Planning, II: the DLV$^{\mathcal{K}}$ System. *Artificial Intelligence*, 144(1–2):157–211, March 2003.

[exe,] Exeura s.r.l., Homepage `http://www.exeura.it/`.

[Faber *et al.*, 1999] 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.

[Faber *et al.*, 2001] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Experimenting with Heuristics for Answer Set Programming. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*, pages 635–640, Seattle, WA, USA, August 2001. Morgan Kaufmann Publishers.

[Faber *et al.*, 2004] 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)*, number 3229 in Lecture Notes in AI (LNAI), pages 200–212. Springer Verlag, September 2004.

[Garcia-Molina *et al.*, 2000] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer D. Widom. *Database System Implementation*. Prentice Hall, 2000.

[Gebser *et al.*, 2007a] 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.

[Gebser *et al.*, 2007b] 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, *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR '07)*, volume 4483, pages 266–271, Tempe, AZ, USA, 2007.

[Gelder *et al.*, 1991] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.

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

[Gottlob *et al.*, 1999] Georg Gottlob, Nicola Leone, and Helmut Veith. Succinctness as a Source of Expression Complexity. *Annals of Pure and Applied Logic*, 97(1–3):231–260, 1999.

[Gottlob, 1994] Georg Gottlob. Complexity and Expressive Power of Disjunctive Logic Programming. In M. Bruynooghe, editor, *Proceedings of the International Logic Programming Symposium (ILPS '94)*, pages 23–42, Ithaca NY, 1994. MIT Press.

[Janhunen *et al.*, 2003] Tomi Janhunen, Ilkka Niemelä, Dietmar Seipel, Patrik Simons, and Jia-Huai You. Unfolding Partiality and Disjunctions in Stable Model Semantics. Technical Report cs.AI/0303009, arXiv.org, March 2003.

[Johnson, 1990] David S. Johnson. A Catalog of Complexity Classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 2. Elsevier Science Pub., 1990.

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

[Koch and Leone, 1999] Christoph Koch and Nicola Leone. Stable Model Checking Made Easy. In Thomas Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI) 1999*, pages 70–75, Stockholm, Sweden, August 1999. Morgan Kaufmann Publishers.

[Leone *et al.*, 1997] 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.

[Leone *et al.*, 2001] 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, September 2001, Proceedings*, number 2173 in Lecture Notes in AI (LNAI). Springer Verlag, September 2001.

[Leone *et al.*, 2005] Nicola Leone, Georg Gottlob, Riccardo Rosati, Thomas Eiter, Wolfgang Faber, Michael Fink, Gianluigi Greco, Giovambattista Ianni, Edyta Kałka, Domenico Lembo, Maurizio Lenzerini, Vincenzino Lio, Bartosz Nowicki, Marco Ruzzi, Witold Staniszkis, and Giorgio Terracina. The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In *Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005)*, pages 915–917, Baltimore, Maryland, USA, June 2005. ACM Press.

[Leone *et al.*, 2006] 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.

[Lierler and Maratea, 2004] 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.

[Lierler, 2005] 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.

[Lifschitz and Turner, 1994] Vladimir Lifschitz and Hudson Turner. Splitting a Logic Program. In Pascal Van Hentenryck, editor, *Proceedings of the 11th In-*

ternational Conference on Logic Programming (ICLP'94), pages 23–37, Santa Margherita Ligure, Italy, June 1994. MIT Press.

[Lifschitz, 1996] Vladimir Lifschitz. Foundations of Logic Programming. In G. Brewka, editor, *Principles of Knowledge Representation*, pages 69–127. CSLI Publications, Stanford, 1996.

[Lin and Zhao, 2002] 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.

[Lobo *et al.*, 1992] Jorge Lobo, Jack Minker, and Arcot Rajasekar. *Foundations of Disjunctive Logic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.

[Minker, 1982] Jack Minker. On Indefinite Data Bases and the Closed World Assumption. In D.W. Loveland, editor, *Proceedings* 6^{th} *Conference on Automated Deduction (CADE '82)*, number 138 in Lecture Notes in Computer Science, pages 292–308, New York, 1982. Springer.

[Minker, 1994] Jack Minker. Overview of Disjunctive Logic Programming. *Annals of Mathematics and Artificial Intelligence*, 12:1–24, 1994.

[Niemelä and Simons, 1996] 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.

[Niemelä and Simons, 1997] 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.

[Papadimitriou, 1984] Christos H. Papadimitriou. The Complexity of Unique Solutions. *Journal of the ACM*, 31:492–500, 1984.

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

[Perri *et al.*, 2007] Simona Perri, Francesco Scarcello, Gelsomina Catalano, and Nicola Leone. Enhancing DLV instantiator by backjumping techniques. *Annals of Mathematics and Artificial Intelligence*, 51(2–4):195–228, 2007.

[Perri, 2004] Simona Perri. *Disjunctive Logic Programming: Efficient Evaluation and Language Extensions*. PhD thesis, University of Calabria, 2004.

[Prosser, 1993] Patrick Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268–299, 1993.

[Przymusinski, 1988] Theodor 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.

[Przymusinski, 1991] Theodor C. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.

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

[Rao *et al.*, 1997] Prasad Rao, Konstantinos F. Sagonas, Terrance Swift, David S. Warren, and Juliana Freire. XSB: A System for Efficiently Computing Well-Founded Semantics. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, number 1265 in Lecture Notes in AI (LNAI), pages 2–17, Dagstuhl, Germany, July 1997. Springer Verlag.

[SEP, 1990] SEPIA, Standard ECRC Prolog Integrating Advanced Features) Version 3.0, 1990. ECRC (European Computer-Industry Research Centre), Munich Germany, `http://www.clps.de/`.

[Shen, 1996] Kish Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *Journal of Logic Programming*, 29(1–3):245–293, 1996.

[Simons *et al.*, 2002] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138(1–2):181–234, June 2002.

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

[Syrjänen, 2002] Tommi Syrjänen. Lparse 1.0 User's Manual, 2002. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

[Tsang, 1993] Edward P.K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

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

[Wielemaker, 1997 2003] J. Wielemaker. SWI-Prolog 5.1: Reference Manual. University of Amsterdam, 1997-2003.

[Wittocx *et al.*, 2008] J. Wittocx, M. Mariën, and M. Denecker. Indexing in Prolog. In *Proceedings of the Twelfth International Workshop on Non-Monotonic Reasoning*, pages 189–198, 2008.

[Wolfinger, 1994] B. Wolfinger, editor. Workshop: *Disjunctive Logic Programming and Disjunctive Databases*, Berlin, August 1994. German Society for Computer Science (GI), Springer. 13^{th} IFIP World Computer Congress, Hamburg, Germany.