Università degli Studi della Calabria

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea Specialistica in Informatica

TESI DI LAUREA

Valutazione Efficiente di Aggregati Ricorsivi in Programmazione Logica

Relatori

Prof. Nicola Leone ____________________

Prof. Wolfgang Faber ____________________

Candidato

Mario Alviano ____________________

Anno Accademico 2006-2007

Alla mia famiglia

Sommario

La programmazione logica disgiuntiva è uno strumento potente per la rappresentazione della conoscenza e del ragionamento di senso comune. Il formalismo classico non prevede le operazioni di aggregazione, sebbene siano state definite molte estensioni in questa direzione. Questa tesi analizza le proprietà dei programmi logici disgiuntivi con aggregati ricorsivi secondo la semantica dei modelli stabili.

Dopo aver presentato una nuova nozione di insieme infondato per programmi logici disgiuntivi con aggregati ricorsivi, caratterizziamo i modelli stabili attraverso questa definizione. Presentiamo un operatore per la computazione del greatest unfounded set, mostrando come calcolarne modularmente il punto fisso. Generalizziamo quindi l'operatore well-founded, definendo un algoritmo per la sua computazione.

Indichiamo inoltre come ovviare all'insufficienza delle attuali tecniche di istanziazione, che non sono adatte al trattamento degli aggregati ricorsivi.

L'efficienza delle tecniche computazionali individuate risulta dall'implementazione in DLV *e dalla conseguente sperimentazione.*

Abstract

Disjunctive logic programs are a powerful tool for knowledge representation and commonsense reasoning. While the classic formalism does not include aggregations, many extensions in this direction have been defined. This thesis analyses properties of disjunctive logic programs with recursive aggregates under stable model semantics.

After introducing a new notion of unfounded set for disjunctive logic programs with recursive aggregates, we characterize stable models by means of this definition. We introduce an operator for the computation of greatest unfounded sets, showing how to compute its fix point in a modular way. Then, we generalize the well-founded operator, defining an algorithm to compute it.

We also suggest how to remedy the actual instantiation technique which is not directly suitable for handling recursive aggregates.

We report on the implementation of the obtained techniques in DLV *and assess its efficiency by means of a thorough experimentation.*

Indice

1 Introduzione **1**

1.1 Contesto e motivazioni 1
1.2 Contributi della tesi 5
1.3 Organizzazione della tesi 6

I Proprietà dei programmi logici con aggregati **9**

2 Programmazione logica con aggregati **13**

2.1 Sintassi 13
2.2 Semantica dei modelli stabili 19
2.3 Monotonicità degli aggregati 26
2.4 Grafo delle dipendenze 29
2.5 Complessità computazionale 32
2.6 Rappresentazione della conoscenza 34

3 Programmi con modello stabile unico **41**

4 Caratterizzazione dei modelli stabili attraverso gli insiemi infondati **45**

4.1 Insiemi infondati 45
4.2 Controllo di stabilità tramite insiemi infondati 51
4.3 Taglio dello spazio di ricerca attraverso gli insiemi infondati . . . 55

II Algoritmi e aspetti computazionali 59

5 Istanziazione efficiente 63
5.1 Istanziazione di un programma . . . 63
5.2 Istanziazione degli aggregati . . . 64
5.3 Tecniche per l'istanziazione incrementale . . . 66
5.3.1 Semi-naive per programmi logici tradizionali . . . 66
5.3.2 Semi-naive per programmi logici con aggregati . . . 69

6 Operatori per il taglio dello spazio di ricerca 75
6.1 L'operatore di conseguenza logica immediata $\mathcal{T}_{\mathcal{P}}$ . . . 75
6.2 L'operatore di Fitting $\Phi_{\mathcal{P}}$ . . . 76
6.3 L'operatore well-founded $\mathcal{W}_{\mathcal{P}}$ . . . 78
6.4 Combinazione efficiente degli operatori . . . 79

7 Computazione del GUS 85
7.1 L'operatore $\mathcal{R}^{\omega}_{\mathcal{P},I}$ per il calcolo del GUS . . . 85
7.2 Valutazione modulare del GUS . . . 88

III Implementazione e sperimentazione 93

8 Architettura del prototipo 97

9 Risultati sperimentali 101
9.1 Sistemi confrontati . . . 101
9.2 Problemi di benchmark . . . 103
9.3 Risultati . . . 107

IV Lavori correlati e conclusioni 113

10 Lavori Correlati 115

11 Conclusioni 117

A Algoritmi per la computazione modulare del GUS 123

B Codifiche dei problemi di benchmark e tabelle dei risultati 127

Elenco delle tabelle

2.1 Carattere dei letterali aggregati 30
2.2 Complessità del controllo di stabilità 33
2.3 Complessità del cautious reasoning 33

9.1 Sistemi confrontati 102
9.2 Risultati suite Asparagus 111

B.1 Risultati suite Company Controls (tempo medio di esecuzione) . . 138
B.2 Risultati suite Company Controls (tempo massimo di esecuzione) 139

Elenco delle figure

1.1 Un’istanza di Company Controls 4

2.1 Intervalli e valori di verità degli aggregati 23
2.2 Grafi *(a)* $DG_{\mathcal{P}_4}$ e *(b)* $DG_{\mathcal{P}_5}$ 31
2.3 Codifica di Car Sequencing 34
2.4 Codifica di Social Golfer (con aggregati stratificati) 35
2.5 Codifica di Social Golfer (con aggregati ricorsivi) 35
2.6 Un’istanza di Knap-Sack 37
2.7 Codifica di Knap-Sack 37
2.8 Codifica di Party Invitations 38
2.9 Codifica di Seating 38
2.10 Codifica di Employee Raise 39

8.1 Architettura del sistema DLV 98

9.1 Codifica di Company Controls 103
9.2 Un’istanza di Sokoban 105
9.3 Comapny Controls: tempo medio di esecuzione 109
9.4 Comapny Controls: tempo massimo di esecuzione 110

A.1 La procedura *computeGUS* 124
A.2 La funzione *isActive* 125
A.3 La funzione *evaluateAggregate* 125

Capitolo 1

Introduzione

1.1 Contesto e motivazioni

La programmazione logica disgiuntiva (DLP) è un potente formalismo per la rappresentazione della conoscenza e il ragionamento di senso comune, che consente di formalizzare, in modo semplice e naturale, problemi complessi. La DLP, anche nota come answer set programming ($\mathcal{ASP}$), consente la disgiunzione e la negazione non stratificata: quello che ne risulta è un linguaggio la cui alta espressività ha importanti implicazioni pratiche, consentendo la modellazione di situazioni reali, quali la rappresentazione di conoscenza incompleta.

L'introduzione degli atomi aggregati [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] è una delle più importanti estensioni sintattiche degli ultimi anni alla programmazione logica. Il formalismo risultante è la programmazione logica disgiuntiva con aggregati (DLP$^{\mathcal{A}}$ o $\mathcal{ASP}^{\mathcal{A}}$). La possibilità di eseguire operazioni di aggregazione su multi-insiemi di termini, tramite un apposito costrutto, permette una formulazione concisa ed una valutazione efficiente dei programmi che ne fanno uso.

Tuttavia, nonostante la semantica e le proprietà computazionali dei programmi logici standard (ovvero privi di aggregati) siano state investigate a fondo, relativamente pochi lavori hanno focalizzato l'attenzione su programmi logici con

aggregati; alcune delle loro proprietà semantiche e dei loro aspetti computazionali sono ancora lontani dall'essere completamente chiari. In particolare, mancano studi su algoritmi e metodi di ottimizzazione per implementare in modo efficiente gli aggregati ricorsivi nella programmazione logica con la semantica dei modelli stabili.

La disabilitazione degli aggregati ricorsivi, d'altronde, impedisce una facile rappresentazione di alcuni problemi, nei quali questi costrutti trovano una naturale applicazione.

Esempio 1.1 *A titolo esemplificativo presentiamo un noto problema che può essere facilmente modellato con la programmazione logica con aggregati ricorsivi: "Company Controls". In questo problema abbiamo:*

- *un insieme di compagnie, rappresentate per mezzo del predicato unario* $company(X)$*, dove* X *indica l'identificatore della compagnia,*
- *un insieme di stock azionari, rappresentati tramite il predicato* $owns(X, Y, S)$*, dove* S *è la percentuale di azioni di* Y *possedute da* X*.*

Lo scopo è determinare tutti i controlli fra compagnie sapendo che:

- *una compagnia* ha il controllo *di un'altra compagnia se possiede, direttamente o indirettamente, più del* 50% *delle sue azioni,*
- *una compagnia possiede* indirettamente *gli stock delle compagnie che controlla.*

Possiamo osservare che esiste una relazione ricorsiva fra il controllo delle compagnie e il possesso degli stock azionari che coinvolge un'operazione di aggregazione.
L'aspetto induttivo di Company Controls può essere modellato con i predicati:

- $cv(X, Z, Y, S)$ *per indicare il fatto che* X *possiede una percentuale* S *delle azioni di* Y *tramite la compagnia* Z*,*

- $controls(X, Y)$ *per indicare che* X *controlla la compagnia* Y*, direttamente o indirettamente.*

Il programma logico che ne risulta è il seguente:

$$
\begin{aligned}
cv(X, X, Y, S) \quad &:- \quad owns(X, Y, S).\\
cv(X, Z, Y, S) \quad &:- \quad controls(X, Z), owns(Z, Y, S).\\
controls(X, Y) \quad &:- \quad company(X), company(Y),\\
& \qquad \#\texttt{sum}\{\ S, Z\ :\ cv(X, Z, Y, S)\ \} > 50.
\end{aligned}
$$

Intuitivamente, la prima regola indica che ogni compagnia possiede direttamente i propri stock azionari. La seconda modella il fatto che ogni compagnia possiede indirettamente gli stock delle compagnie che controlla. Infine, la terza regola determina il fatto che una compagnia X *controlla una compagnia* Y *se ne possiede più del* 50% *delle azioni.*
Consideriamo l'istanza riportata in Figura 1.1. I nodi indicano le compagnie a, b, c*. Gli archi rappresentano gli stock azionari posseduti dalle compagnie, ovvero il fatto che:*

- a *possiede il* 60% *delle azioni di* b*,*
- a *possiede il* 40% *delle azioni di* c*,*
- b *possiede il* 20% *delle azioni di* c*.*

Rappresentiamo questo scenario con i seguenti fatti:

$$
\begin{array}{ll}
company(a). & owns(a, b, 60).\\
company(b). & owns(a, c, 40).\\
company(c). & owns(b, c, 20).
\end{array}
$$

Osserviamo che a *ha il controllo diretto di* b *e, grazie al possesso indiretto degli stock azionari di* b*, controlla anche* c*.*

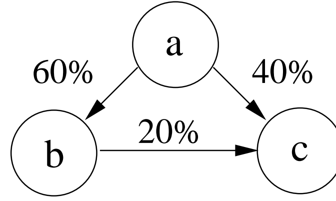


Figura 1.1: *Un'istanza di Company Controls*

Al momento nessun solver per $\mathcal{ASP}$ disgiuntiva con la semantica dei modelli stabili è in grado di processare programmi come quello mostrato nell'esempio 1.1.

Motivati da ciò, nel lavoro di tesi abbiamo cercato di rimediare a queste mancanze e fare un passo avanti verso un'implementazione efficiente degli aggregati ricorsivi in $\mathcal{ASP}$. In particolare, le nostre attenzioni sono state rivolte al ricco frammento dell'$\mathcal{ASP}^{\mathcal{A}}$ che consente la disgiunzione, la negazione nonmonotona e gli aggregati ricorsivi monotoni e antimonotoni. Denotiamo questo linguaggio con $\text{DLP}^{\mathcal{A}}_{m,a}$ (*Disjunctive Logic Programming with Monotone and Antimonotone Aggregates*). Per raggiungere questo scopo, ci siamo concentrati principalmente sulle proprietà degli insiemi infondati per programmi con aggregati, scelta motivata dal fatto che gli insiemi infondati sono alla base delle implementazioni di tutti i solver $\mathcal{ASP}$ attualmente disponibili [5, 11, 12, 13, 14, 15]: i solver $\mathcal{ASP}$ *nativi*, come per esempio DLV e *Smodels*, usano gli insiemi infondati per il pruning dello spazio di ricerca (tramite l'operatore well-founded); i solver $\mathcal{ASP}$ *SAT-based*, come per esempio *ASSAT*, *Cmodels* e *clasp*, usano il relativo concetto di loop formulas [16, 17] per il controllo di stabilità dei modelli. Per queste ragioni, ci aspettiamo che uno studio approfondito delle proprietà degli insiemi infondati per programmi con aggregati sia un contributo valido per l'implementazione di sistemi $\mathcal{ASP}^{\mathcal{A}}$ efficienti. Inoltre, allo scopo di individuare una valida strategia di istanziazione, abbiamo analizzato gli aspetti computazionali di alcuni sottoinsiemi della programmazione logica con aggregati. Nessuno dei precedenti lavori ha

curato questo aspetto, che dai nostri studi ed esperimenti è risultato di cruciale importanza.

1.2 Contributi della tesi

I principali contributi del presente lavoro di tesi sono riassumibili in tre categorie: analisi delle proprietà della $\text{DLP}^{\mathcal{A}}_{m,a}$, progettazione di algoritmi per la valutazione efficiente, implementazione e sperimentazione del prototipo.

In particolare, i contributi relativi all'analisi delle proprietà del linguaggio sono i seguenti.

- La definizione di una nuova e intuitiva nozione di insieme infondato per $\text{DLP}^{\mathcal{A}}_{m,a}$ correlata alle precedenti, le quali sono in accordo con le definizioni di insiemi infondati per programmi privi di aggregati e ne caratterizzano le proprietà.

- La dimostrazione formale di alcune proprietà che consentono di ottimizzare la computazione dei modelli stabili per un sottoinsieme della programmazione logica con aggregati.

- La prova che gli insiemi infondati possono essere utilizzati con profitto nella computazione $\text{DLP}^{\mathcal{A}}_{m,a}$ per il controllo di stabilità e per il taglio dello spazio di ricerca.

Relativamente alla progettazione di algoritmi per la valutazione efficiente, i nostri contributi sono i seguenti.

- La dimostrazione dell'insufficienza della tecnica semi-naive per istanziazione degli aggregati ricorsivi e la presentazione di una nuova tecnica che corregge questa mancanza.

- La definizione di un operatore per la computazione del greatest unfounded set e la progettazione di una tecnica di valutazione modulare per la sua computazione, che localizza il calcolo sulle singole componenti.

- La presentazione di un algoritmo per la computazione efficiente del greatest unfounded set, che localizza la computazione sulle sole componenti coinvolte nella propagazione del passo precedente.

Per quanto riguarda l'implementazione e la sperimentazione del prototipo realizzato, i nostri contributi possono essere riassunti nel seguente modo.

- L'implementazione dei precedenti risultati in DLV [1], ottenendo un sistema prototipale che supporta il linguaggio $\text{DLP}^{\mathcal{A}}_{m,a}$.

- La valutazione sperimentale delle caratteristiche dei programmi con aggregati ricorsivi e del prototipo realizzato.

Per poter raggiungere i nostri obiettivi, abbiamo adottato la semantica $\mathcal{ASP}^{\mathcal{A}}$ definita in [1], che sembra aver ricevuto un certo consenso. Lavori recenti, quali [18, 19], danno ulteriore supporto alla plausibilità di questa semantica, collegandola ai costrutti stabiliti per programmi privi di aggregati. In particolare, [18] presenta una semantica per programmi generici e mostra che questa coincide con [1] per programmi $\mathcal{ASP}^{\mathcal{A}}$.

A nostra conoscenza, il lavoro di questa tesi fornisce la prima implementazione di aggregati ricorsivi in $\mathcal{ASP}$ disgiuntiva. Le precedenti implementazioni omettevano gli aggregati ricorsivi [6] o disabilitavano la disgiunzione [20, 21, 5, 10] [2]. Osserviamo inoltre che le semantiche in [20, 21, 5, 10], che sono in generale differenti da quella in [1], coincidono con questa nel frammento della $\text{DLP}^{\mathcal{A}}_{m,a}$.

1.3 Organizzazione della tesi

Il seguito della tesi è così strutturato.

- Nella Parte I descriviamo la programmazione logica con aggregati, mostrando la sintassi, la semantica e le proprietà del linguaggio. Definiamo

[1] Prima di questa estensione DLV supportava solo gli aggregati non ricorsivi.
[2] Si noti che Cmodels [22] disabilita gli aggregati nelle regole disgiuntive.

inoltre gli insiemi infondati, con le loro proprietà utili per caratterizzare i modelli stabili.

- Nella Parte II illustriamo gli aspetti computazionali relativi alla programmazione logica con aggregati. Mostriamo prima come istanziare efficientemente i programmi, tramite una variante del metodo semi-naive. Successivamente, descriviamo un operatore per la computazione del GUS e come modularizzarne il calcolo.

- Nella Parte III presentiamo l'architettura del prototipo realizzato e i risultati ottenuti dalla sperimentazione.

- Nella Parte IV riportiamo i lavori correlati e le conclusioni.

Parte I

Proprietà dei programmi logici con aggregati

La definizione formale delle proprietà dei programmi logici con aggregati è fondamentale per la trattazione delle tecniche che illustreremo nel seguito della tesi, per le quali i concetti di modello stabile e insieme infondato ricoprono un ruolo centrale.

Nel capitolo 2 descriviamo la sintassi dei programmi logici con aggregati e la semantica dei modelli stabili. Introduciamo inoltre le nozioni di monotonicità, antimonotonicità e non-monotonicità, di cruciale importanza per questo lavoro. Facciamo una breve trattazione della complessità computazionale per le classi dei linguaggi della DLP$^{\mathcal{A}}$, mostrando che in generale coincide con quella della DLP.

Nel capitolo 3 definiamo alcuni sottoinsiemi della DLP$^{\mathcal{A}}$ per i quali è garantita l'esistenza e l'unicità del modello stabile.

Nel capitolo 4 introduciamo le nozioni di insieme infondato e greatest unfounded set per programmi logici con aggregati, che utilizziamo per caratterizzare i modelli stabili.

Capitolo 2

Programmazione logica con aggregati

In questo capitolo presentiamo la sintassi e la semantica della programmazione logica con gli aggregati, mostriamo la complessità del linguaggio e come utilizzarlo per rappresentare la conoscenza e il ragionamento di senso comune.

2.1 Sintassi

Sia $\mathcal{C}$ un insieme di *costanti*, $\mathcal{V}$ un insieme di *variabili* ed $\mathcal{S}$ un insieme di *simboli di predicato*. Ad ogni simbolo di predicato associamo un numero naturale, detto *arità*. Adottiamo la convenzione classica per cui le variabili vengono generalmente indicate con la prima lettera maiuscola (per esempio, X, Y, Z, ma anche $Company$, $\ldots$), i simboli di predicato con la prima lettera minuscola (per esempio, p, $company$, $\ldots$) e le costanti con numeri o con la prima lettera minuscola (per esempio, a, b, 1234, $\ldots$).

Un *termine* è una variabile o una costante (quindi un elemento di $\mathcal{C} \cup \mathcal{V}$).

Un *atomo standard* è un costrutto del tipo

$$p(t_1, \ldots, t_n)$$

dove

- p è un simbolo di predicato (ovvero $p \in \mathcal{S}$),
- $t_1, \ldots, t_n$ sono termini (ovvero $t_i \in \mathcal{C} \cup \mathcal{V}$, $i = 1, \ldots, n$).

L'*arità* associata al predicato p è n e, nel caso in cui questa sia 0, le parentesi tonde vengono generalmente omesse, consentendo la più semplice scrittura p. Se tutti i termini sono costanti, allora l'atomo viene detto *ground* (ovvero privo di variabili).

Multi-insiemi di termini. Un *multi-insieme di termini* può essere un multi-insieme simbolico o un multi-insieme ground.
Un *multi-insieme simbolico* è una struttura del tipo

$$\{Vars \ : \ Conj\}$$

dove

- $Vars$ è una lista di variabili,
- $Conj$ è una congiunzione di atomi standard, che costituisce la funzione di supporto del multi-insieme.

Esempio 2.1 *Il multi-insieme*

$$\{X \ : \ a(X,Y), p(Y)\}$$

rappresenta il multi-insieme dei valori X *che rendono la congiunzione* $a(X,Y) \wedge p(Y)$ *vera, ovvero il multi-insieme*

$$\{X \ : \ \exists Y\, s.t.\ a(X,Y) \wedge p(Y)\ is\ true\}$$

Un *multi-insieme ground* è un insieme di coppie del tipo

$$\langle\, \bar{t} \;:\; Conj \,\rangle$$

dove

- $\bar{t}$ è una lista di costanti,
- $Conj$ è una congiunzione di atomi standard ground, che rappresenta la funzione di supporto dell'elemento $\bar{t}$.

Esempio 2.2 *Sono multi-insiemi ground*

$$\begin{array}{l}
\{\, \langle\, a, 0 \;:\; p(a,0)\,\rangle, \langle\, a, 1 \;:\; p(a,1)\,\rangle, \langle\, b, 1 \;:\; p(b,1)\,\rangle \,\}\\
\{\, \langle\, a \;:\; p(a,0)\,\rangle, \langle\, a \;:\; p(a,1)\,\rangle, \langle\, b \;:\; p(b,1)\,\rangle \,\}\\
\{\, \langle\, 0 \;:\; p(a,0)\,\rangle, \langle\, 1 \;:\; p(a,1)\,\rangle, \langle\, 1 \;:\; p(b,1)\,\rangle \,\}
\end{array}$$

Funzioni di aggregazione. Una *funzione di aggregazione* è un costrutto del tipo

$$f(S)$$

dove

- S è un insieme di termini,
- f è un *simbolo di funzione* predefinito.

Intuitivamente, una funzione di aggregazione mappa multi-insiemi di costanti su una costante.

Esempio 2.3 Negli esempi adotteremo la sintassi di DLV per denotare gli aggregati.

Le funzioni di aggregazione che al momento sono supportate dal sistema DLV *sono:*

- #`count`: *numero di termini diversi*
- #`sum`: *somma su interi non negativi*
- #`times`: *prodotto di interi positivi*
- #`min`: *minimo termine*
- #`max`: *massimo termine*

Letterali aggregati. Un *atomo aggregato* è un costrutto del tipo

$$f(S) \prec T$$

dove

- $f(S)$ è una funzione di aggregazione,
- $\prec \in \{=, <, \leq, >, \geq\}$ è un operatore di confronto predefinito,
- T è un termine, detto *guardia*.

Inoltre, un atomo aggregato può avere la forma

$$T_1 \prec_1 f(S) \prec_2 T_2$$

dove

- $f(S)$ è una funzione di aggregazione,
- $\prec_1, \prec_2 \in \{<, \leq\}$ sono operatori di confronto predefiniti,
- T_1 e T_2 sono termini, chiamati rispettivamente *guardia inferiore* e *guardia superiore*.

Esempio 2.4 *Di seguito sono riportati due esempi di aggregati nella notazione di* DLV*; il primo è un aggregato non-ground, mentre il secondo rappresenta una possibile istanza ground del primo:*

$$\#\texttt{max}\{Z \ : \ r(Z), a(Z, V)\} > Y$$
$$\#\texttt{max}\{\langle\, 1 : r(1), a(1, k)\,\rangle, \langle\, 2 : r(2), a(2, c)\,\rangle\} > 1$$

Un *atomo* è un atomo standard o un atomo aggregato. Un *letterale* ℓ è un atomo $\mathcal{A}$ o un atomo $\mathcal{A}$ preceduto dal simbolo di negazione per fallimento `not`. Non bisogna confondere la negazione classica ($\neg$) con la negazione per fallimento: `not` $\mathcal{A}$ indica che $\mathcal{A}$ non deve essere vero, mentre $\neg\mathcal{A}$ afferma che $\mathcal{A}$ è falso. Se $\mathcal{A}$ è un atomo standard, allora ℓ è un *letterale standard*. Se $\mathcal{A}$ è un atomo aggregato, allora ℓ è un *letterale aggregato*.

Programmi DLP$^{\mathcal{A}}$. Una *regola* DLP$^{\mathcal{A}}$ r è un costrutto

$$a_1 \ \vee \ \cdots \ \vee \ a_n \ :- \ b_1, \ldots, b_k, \ \texttt{not} \ b_{k+1}, \ldots, \ \texttt{not} \ b_m.$$

dove

- $a_1, \ldots, a_n$ sono atomi standard,
- $b_1, \ldots, b_m$ sono atomi,
- $n >= 1, m >= k >= 0$.

La disgiunzione $a_1 \vee \cdots \vee a_n$ viene detta la *testa* di r, mentre la congiunzione $b_1, \ldots, b_k, \texttt{not} \ b_{k+1}, \ldots, \ \texttt{not} \ b_m$ è il *corpo* di r. Denotiamo con $H(r)$ l'insieme degli atomi nella testa di r, con $B(r)$ l'insieme $b_1, \ldots, b_k, \texttt{not} \ b_{k+1}, \ldots, \ \texttt{not} \ b_m$ dei letterali del corpo di r. $B^+(r)$ e $B^-(r)$ denotano, rispettivamente, gli insiemi dei letterali positivi e negativi in $B(r)$. Si noti che questa sintassi non consente esplicitamente l'utilizzo di vincoli di integrità (regole con testa vuota). Tuttavia,

è possibile simularli mettendo in testa un nuovo simbolo di predicato (con arità 0) e forzando il suo valore a *falso*.

Esempio 2.5 *Possiamo forzare il valore di verità di un nuovo atomo* co *ad essere falso aggiungendo al programma la regola*

p :– co, not p.

dove p *è un atomo che non compare nel programma. Quindi, un constraint*

:– $a, b, c, \ldots$

può essere scritto come

co :– $a, b, c, \ldots$

Un *programma* DLP$^{\mathcal{A}}$ è un insieme di regole DLP$^{\mathcal{A}}$.

Safety. Una variabile di una regola che compare in un atomo standard è detta *variabile globale*. Tutte le altre variabili che compaiono nella regola sono dette *variabili locali*. Una regola è *safe* (sicura) se valgono le seguenti condizioni:

(i) ogni variabile globale di r appare in almeno un letterale standard positivo nel corpo di r;

(ii) ogni variabile locale di r che compare in un insieme simbolico $\{Vars : Conj\}$ appare in un atomo di $Conj$;

(iii) ogni guardia di un atomo aggregato di r è una costante o una variabile globale.

Un programma $\mathcal{P}$ è safe se tutte le regole $r \in \mathcal{P}$ sono safe.

Esempio 2.6 *Consideriamo le seguenti regole con aggregati:*

$$p(X) :- q(X,Y,V), \#\mathtt{max}\{Z : r(Z), a(Z,V)\} > Y.$$
$$p(X) :- q(X,Y,V), \#\mathtt{sum}\{Z : a(Z,S)\} > Y.$$
$$p(X) :- q(X,Y,V), \#\mathtt{min}\{Z : r(Z), a(Z,V)\} > T.$$

La prima regola è safe, mentre la seconda non lo è, poiché la variabile locale S viola la condizione (ii). La terza regola è anch'essa non safe, perché la guardia T viola la condizione (iii).

Nel seguito assumiamo, per semplicità della trattazione, che tutti i programmi DLP$^{\mathcal{A}}$ sono safe.

2.2 Semantica dei modelli stabili

Universo e Base di Herbrand. Dato un programma DLP$^{\mathcal{A}}$ $\mathcal{P}$, definiamo:

- *Universo di Herbrand*, e lo indichiamo con $U_{\mathcal{P}}$, l'insieme delle costanti che compaiono in $\mathcal{P}$;
- *Base di Herbrand*, denotata con $B_{\mathcal{P}}$, l'insieme degli atomi *standard* costruibili a partire dai predicati di $\mathcal{P}$ con le costanti in $U_{\mathcal{P}}$.

Istanziazione. Una *sostituzione* σ è un mapping da un insieme di variabili verso $U_{\mathcal{P}}$:

$$\sigma : \mathcal{V} \longrightarrow U_{\mathcal{P}}$$

Distinguiamo due tipi di sostituzioni:

- una *sostituzione globale* per una regola r è una sostituzione dall'insieme delle sue variabili globali verso $U_{\mathcal{P}}$;
- una *sostituzione locale* per un insieme simbolico S è una sostituzione dall'insieme delle sue variabili locali verso $U_{\mathcal{P}}$.

Esempio 2.7 *Consideriamo la sostituzione $\sigma = [X/a, Y/b, Z/c]$, che sostituisce ogni occorrenza delle variabili X, Y, Z con le costanti a, b, c, rispettivamente.*

Allora, l'applicazione di σ alla regola

$$r \quad : \quad p(X,Y) \ {:}{-}\ q(Y,Z), t(X).$$

darà come risultato la seguente sostituzione globale

$$\sigma(r) \quad : \quad p(a,b) \ {:}{-}\ q(b,c), t(a).$$

Dato un insieme simbolico senza variabili globali

$$S = \{\ Vars\ :\ Conj\ \}$$

l'*istanziazione di* S è il seguente insieme di coppie ground:

$$Inst(S) \ = \ \{\langle\gamma(Vars) : \gamma(Conj)\rangle \ : \ \gamma \textit{ is a local substitution for } S\}$$

Un'*istanza ground* di una regola r si ottiene in due passi:

1. una sostituzione globale σ viene applicata su r;
2. ogni insieme simbolico S in $\sigma(r)$ viene sostituito dalla sua istanziazione $Inst(S)$.

L'istanziazione $Ground(\mathcal{P})$ di un programma $\mathcal{P}$ è l'insieme di tutte le possibili istanze delle regole di $\mathcal{P}$.

Esempio 2.8 *Consideriamo il seguente programma:*

$$\begin{aligned} \mathcal{P}_1 \quad = \quad \{ \quad & q(1) \vee p(2,2).\,, \\ & q(2) \vee p(2,1).\,, \\ & t(X) \ {:}{-}\ q(X), \#\mathtt{sum}\{Y : p(X,Y)\} > 1. \quad \} \end{aligned}$$

L'istanziazione di $\mathcal{P}_1$ è la seguente:

$$\begin{array}{rcl} Ground(\mathcal{P}_1) & = & \{ \quad q(1) \vee p(2,2).\,, \\ & & \quad q(2) \vee p(2,1).\,, \\ & & \quad t(1) :- q(1), \#\texttt{sum}\{\langle 1\!:\!p(1,1)\rangle, \langle 2\!:\!p(1,2)\rangle\} > 1.\,, \\ & & \quad t(2) :- q(2), \#\texttt{sum}\{\langle 1\!:\!p(2,1)\rangle, \langle 2\!:\!p(2,2)\rangle\} > 1. \quad \} \end{array}$$

Nel seguito, dove non specificato diversamente, assumiamo che tutti i programmi DLP$^{\mathcal{A}}$ sono ground.

Se X è un insieme, useremo la notazione $\neg.X$ per indicare l'insieme

$$\neg.X = \{\neg x \,:\, x \in X\}$$

Interpretazioni. Un insieme $X \subseteq (B_{\mathcal{P}} \cup \neg.B_{\mathcal{P}})$ di letterali standard ground è consistente se nessun atomo compare allo stesso tempo positivo e negativo in X.

Un'*interpretazione* I per un programma DLP$^{\mathcal{A}}$ $\mathcal{P}$ è un insieme consistente di letterali standard ground.

Un letterale standard ground ℓ è vero (risp. falso) rispetto ad I se $\ell \in I$ (risp. $\ell \in \neg.I$). Se un letterale standard ground non è né vero né falso rispetto ad I, allora è indefinito rispetto ad I. Denotiamo con I^+ (risp. I^-) l'insieme di tutti gli atomi che occorrono in letterali standard positivi (risp. negativi) in I, e con $\overline{I}$ l'insieme degli atomi indefiniti rispetto ad I (ovvero $B_{\mathcal{P}} \setminus I^+ \cup I^-$).

Un'interpretazione I è *totale* se $\overline{I}$ è vuoto (ovvero $I^+ \cup I^- = B_{\mathcal{P}}$), altrimenti I è *parziale*. Una *totalizzazione* di un'interpretazione (parziale) I è un'interpretazione totale J che contiene I (ovvero J è un'interpretazione totale tale che $J \supseteq I$).

Un'interpretazione fornisce anche un significato per i letterali aggregati.

Sia I un'interpretazione totale. Una congiunzione standard ground è vera rispetto ad I se tutti i suoi letterali sono veri rispetto ad I; altrimenti è falsa (ovvero è falsa se qualcuno dei suoi letterali è falso rispetto ad I).

Il significato di un multi-insieme di termini, di una funzione di aggregazione e di un atomo aggregato rispetto ad un'interpretazione totale è, rispettivamente, un multi-insieme, un valore numerico e un valore di verità.

Sia $f(S)$ una funzione di aggregazione. La valutazione $I(S)$ di S rispetto ad I è il multi-insieme della prima costante degli elementi di S la cui congiunzione è vera rispetto ad I. Più precisamente, $I(S)$ denota il multi-insieme

$$I(S) = \{t_1 \mid \langle t_1, ..., t_n : Conj\rangle \in S \wedge Conj \text{ is true w.r.t. } I\}$$

La valutazione $I(f(S))$ di una funzione di aggregazione $f(S)$ rispetto ad I è il risultato dell'applicazione di f su $I(S)$. Se il multi-insieme $I(S)$ non è nel dominio di f, $I(f(S)) = \perp$ (dove $\perp$ è un simbolo fissato che non occorre in $\mathcal{P}$).

Un letterale aggregato istanziato $\mathcal{A}$ della forma

$$f(S) \prec k$$

è vero rispetto ad I se:

(i) $I(f(S)) \neq \perp$, e

(ii) $I(f(S)) \prec k$.

altrimenti $\mathcal{A}$ è falso. Un letterale aggregato istanziato

$$\texttt{not}\ \mathcal{A} = \texttt{not}\ f(S) \prec k$$

è vero rispetto ad I se:

(i) $I(f(S)) \neq \perp$, e

(ii) $I(f(S)) \not\prec k$.

altrimenti $\mathcal{A}$ è falso.

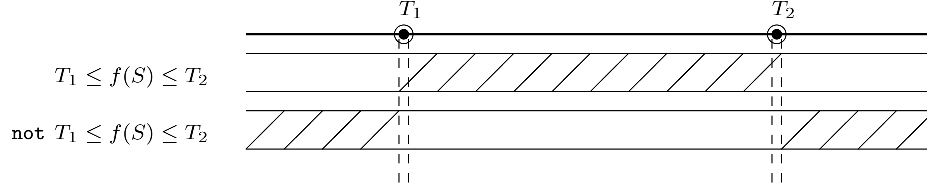


Figura 2.1: *Intervalli e valori di verità degli aggregati*

Intuitivamente, come mostrato in Figura 2.1, un letterale aggregato positivo

$$T_1 \leq f(S) \leq T_2$$

è vero rispetto ad un'interpretazione totale I se la valutazione della sua funzione di aggregazione rispetto ad I appartiene all'intervallo $[T_1, T_2]$ (ovvero $I(S) \in [T_1, T_2]$), altrimenti è falso. Un letterale aggregato negativo

$$\texttt{not}\ T_1 \leq f(S) \leq T_2$$

è vero rispetto ad un'interpretazione totale I se la valutazione della sua funzione di aggregazione rispetto ad I non appartiene all'intervallo $[T_1, T_2]$ (ovvero $I(S) \notin [T_1, T_2]$), altrimenti è falso. Se l'operatore a sinistra di $f(S)$ è $<$ possiamo sostituire T_1 con $T_1 + 1$. Se l'operatore a destra di $f(S)$ è $<$ possiamo sostituire T_2 con $T_2 - 1$. Se una delle due guardie manca possiamo sostituirla con il valore estremo dell'intervallo di definizione della funzione di aggregazione (per esempio, 0 oppure ∞). Quindi, possiamo sempre ricondurre la valutazione di un letterale aggregato ad un controllo di appartenenza (o non appartenenza) ad un intervallo.

Se I è un'interpretazione *parziale*, un letterale $\mathcal{A}$ è vero (risp. falso) rispetto ad I se è vero (risp. falso) rispetto a *tutte* le totalizzazioni J di I; altrimenti è indefinito. È da notare che, in generale, non è possibile fare una valutazione efficiente del valore di verità per gli aggregati rispetto ad un'interpretazione parziale I; infatti, il numero di totalizzazioni di I è esponenziale nella cardinalità di

$\overline{I}$. Tuttavia, come illustreremo in seguito (a pagina 26 di questo capitolo), per gli aggregati ricorsivi monotoni e antimonotoni questa valutazione può essere fatta in modo efficiente.

Esempio 2.9 *Consideriamo l'atomo* $\mathcal{A} = \#\mathtt{sum}\{\langle 1 : p(1)\rangle, \langle 2 : p(2)\rangle\} > 1$, *e sia* S *l'insieme ground in* $\mathcal{A}$.
Per l'interpretazione $I = \{p(2)\}$, *ogni interpretazione totale* J *che estenda* I *conterrà o* $p(1)$ *o* $\mathtt{not}\ p(1)$.
Perciò, per una qualsiasi totalizzazione J *di* I, *si avrà che*

- $J(S) = \{2\}$, *oppure*
- $J(S) = \{1, 2\}$.

L'applicazione di $\#\mathtt{sum}$ *, quindi, produrrà*

- $2 > 1$, *che è vero, oppure*
- $3 > 1$, *che è altrettanto vero.*

Ne deduciamo che l'aggregato è vero in tutte le totalizzazioni di I *e, quindi, possiamo affermare che* $\mathcal{A}$ *è vero anche rispetto ad* I.

Le definizioni di interpretazione e valori di verità che abbiamo dato preservano la "monotonicità della conoscenza": se un'interpretazione J estende I (ovvero $J \supseteq I$), allora ogni letterale che è vero rispetto ad I è vero rispetto a J, e ogni letterale che è falso rispetto ad I è falso rispetto a J.

Modelli Minimali. Data un'interpretazione I e una regola ground r, allora

- la testa di r è vera rispetto ad I se qualche letterale in $H(r)$ è vero rispetto ad I;
- il corpo di r è vero rispetto ad I se tutti i letterali in $B(r)$ sono veri rispetto ad I;

- la regola r è soddisfatta rispetto ad I se la sua testa è vera rispetto ad I nel caso in cui il suo corpo è vero rispetto ad I.

Un'interpretazione totale M è un *modello* di un programma DLP$^{\mathcal{A}}$ $\mathcal{P}$ se tutte le regole $r \in \mathcal{P}$ sono soddisfatte rispetto ad M.

Un modello M per $\mathcal{P}$ è minimale se non esiste nessun altro modello M' per $\mathcal{P}$ tale che $M'^{+} \subset M^{+}$.

Si noti che, sotto queste definizioni, la parola *interpretazione* può fare riferimento ad un'interpretazione anche parziale, mentre con *modello* intendiamo sempre un'interpretazione totale.

Modelli stabili (o answer sets). Di seguito riportiamo la generalizzazione della trasformazione di Gelfond-Lifschitz e la definizione di modello stabile (o answer set) per programmi DLP$^{\mathcal{A}}$ introdotta in [1].

Definizione 1 *Dato un programma* DLP$^{\mathcal{A}}$ *ground* $\mathcal{P}$ *e un'interpretazione totale* I, $\mathcal{P}^I$ *denoti il programma trasformato ottenuto da* $\mathcal{P}$ *eliminando tutte le regole nelle quali un letterale del corpo è falso rispetto ad* I.

I è un modello stabile di un programma $\mathcal{P}$ se è un modello minimale di $\mathcal{P}^I$.

Esempio 2.10 *Consideriamo le interpretazioni*

$$I_1 = \{p(a)\}$$

$$I_2 = \{\texttt{not}\ p(a)\}$$

e i due programmi

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

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

Per il primo programma avremo

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

$Ground(\mathcal{P}_2)^{I_1} = Ground(\mathcal{P}_2)$

$Ground(\mathcal{P}_2)^{I_2} = \emptyset$.

Mentre per il secondo

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

$Ground(\mathcal{P}_3)^{I_1} = \emptyset$

$Ground(\mathcal{P}_3)^{I_2} = Ground(\mathcal{P}_3)$.

I_2 è il solo modello stabile di $\mathcal{P}_2$ (poiché I_1 non è un modello minimale di $Ground(\mathcal{P}_2)^{I_1}$), mentre $\mathcal{P}_3$ non ammette modelli stabili (I_1 non è un modello minimale di $Ground(\mathcal{P}_3)^{I_1}$, e I_2 non è un modello di $Ground(\mathcal{P}_3) = Ground(\mathcal{P}_3)^{I_2}$).

Si noti che ogni modello stabile $\mathcal{M}$ di $\mathcal{P}$ è anche un modello di $\mathcal{P}$, perché $\mathcal{P}^{\mathcal{M}} \subseteq \mathcal{P}$ e le regole in $\mathcal{P} \setminus \mathcal{P}^{\mathcal{M}}$ sono soddisfatte rispetto ad $\mathcal{M}$ (in quanto il corpo di queste regole è falso).

2.3 Monotonicità degli aggregati

Monotonicità Date due interpretazioni I e J, diciamo che $I \leq J$ se valgono entrambe le condizioni

- $I^+ \subseteq J^+$
- $I^- \supseteq J^-$.

Un letterale ground ℓ è *monotono* se, per tutte le interpretazioni I, J, tali che $I \leq J$, vale che:

1. ℓ vero rispetto ad I implica ℓ vero rispetto a J,
2. ℓ falso rispetto a J implica ℓ falso rispetto ad I.

Un letterale ground ℓ è *antimonotono* se accade l'opposto, ovvero per tutte le interpretazioni I, J, tali che $I \leq J$, vale che:

1. ℓ falso rispetto ad I implica ℓ falso rispetto a J,

2. ℓ vero rispetto a J implica ℓ vero rispetto ad I.

Un letterale ground ℓ è *nonmonotono* se non è né monotono né antimonotono.

Si noti che i letterali standard positivi sono sempre monotoni, mentre i letterali standard negativi sono sempre antimonotoni. I letterali aggregati (positivi o negativi), invece, possono essere monotoni, antimonotoni o nonmonotoni. I letterali nonmonotoni includono la somma su interi (possibilmente negativi) e la media.

Esempio 2.11 *Tutte le istanze ground di*

$\#\texttt{count}\{X : p(X)\} > 1$, *e*

$\texttt{not}\ \#\texttt{count}\{X : p(X)\} < 1$

sono monotone, mentre per

$\#\texttt{count}\{X : p(X)\} < 1$, *e*

$\texttt{not}\ \#\texttt{count}\{X : p(X)\} > 1$

sono antimonotone.

Denotiamo con $\text{DLP}^{\mathcal{A}}_{m,a}$ il frammento di $\text{DLP}^{\mathcal{A}}$ nel quale possono occorrere solo aggregati monotoni e antimonotoni. Data una regola r di un programma $\text{DLP}^{\mathcal{A}}_{m,a}$, $Mon(B(r))$ e $Ant(B(r))$ denotano, rispettivamente, l'insieme dei letterali *monotoni* e *antimonotoni* in $B(r)$.

Nel seguito con la parola *programma* faremo generalmente riferimento a programmi $\text{DLP}^{\mathcal{A}}_{m,a}$.

È da notare che, come descritto in [23], molti programmi con letterali nonmonotoni possono essere riscritti polinomialmente in programmi $\text{DLP}^{\mathcal{A}}_{m,a}$. Qualche esempio importante include i programmi che contengono atomi aggregati del tipo

(i) $T_1 \prec_1 f(S) \prec_2 T_2$

(ii) $f(S) = T$

che sono nonmonotoni indipendentemente da $f(S)$. Possiamo riscrivere questi letterali nonmonotoni come congiunzione di un letterale monotono e di un letterale antimonotono:

(i) $T_1 \prec_1 f(S) \wedge f(S) \prec_2 T_2$

(ii) $f(S) \geq T \wedge f(S) \leq T$.

Esempio 2.12 *La regola*

$$p(0) :\!-\; 3 \leq \#\texttt{count}\{X : p(X)\} \leq 7.$$

può essere riscritta nella regola

$$p(0) :\!-\; \#\texttt{count}\{X : p(X)\} \geq 3, \#\texttt{count}\{X : p(X)\} \leq 7.$$

Mentre la regola

$$p(0) :\!-\; \texttt{not}\ 5 \leq \#\texttt{sum}\{X : p(X)\} \leq 21, \texttt{not}\ 4 \leq \#\texttt{times}\{Y : q(Y)\} \leq 15.$$

può essere riscritta polinomialmente tramite l'uso di un nuovo atomo per ogni letterale aggregato negativo

$$aux_1 :\!-\; \texttt{not}\ \#\texttt{sum}\{X : p(X)\} \geq 5.$$

$$aux_1 :\!-\; \texttt{not}\ \#\texttt{sum}\{X : p(X)\} \leq 21.$$

$$aux_2 :\!-\; \texttt{not}\ \#\texttt{times}\{Y : q(Y)\} \geq 4.$$

$$aux_2 :\!-\; \texttt{not}\ \#\texttt{times}\{Y : q(Y)\} \leq 15.$$

$$p(0) :\!-\; aux_1, aux_2.$$

Pertanto, nel classificare i diversi tipi di letterali aggregati, in Tabella 2.1, è conveniente considerare solo letterali nella forma

$$f(S) \prec T \quad \texttt{e} \quad \texttt{not}\ f(S) \prec T$$

con $\prec \in \{<, \leq, >, \geq\}$.

2.4 Grafo delle dipendenze

Possiamo quindi introdurre la nozione di *grafo delle dipendenze* e di componenti fortemente connesse.

Definizione 2 *Associamo ad ogni programma ground $\mathcal{P}$ un grafo diretto $DG_{\mathcal{P}} = (\mathcal{N}, E)$, chiamato il* grafo delle dipendenze *di $\mathcal{P}$, nel quale*

(i) ogni atomo standard di $\mathcal{P}$ è un nodo in $\mathcal{N}$,

(ii) c'è un arco in E diretto da un nodo a verso un nodo b se e solo se c'è una regola r in $\mathcal{P}$ tale che $b \in H(r)$ e

- *a è un atomo standard in $Mon(B(r))$ oppure*
- *a è un atomo che compare nell'insieme ground di un letterale aggregato in $Mon(B(r))$.*

Esempio 2.13 *Consideriamo i seguenti programmi:*

$$\begin{aligned} \mathcal{P}_4 \;=\; \{ \;\; & p(0) \vee q(0).\,, \\ & s(0) :- \#\texttt{count}\{ \langle 0, p(0) \rangle \} > 0.\,, \\ & s(0) :- \#\texttt{count}\{ \langle 0, q(0) \rangle \} > 0. \quad \} \end{aligned}$$

$$\begin{aligned} \mathcal{P}_5 \;=\; \{ \;\; & q(0) :- \#\texttt{count}\{ \langle 0, p(0) \rangle \} > 0.\,, \\ & p(0) :- \#\texttt{count}\{ \langle 0, q(0) \rangle \} > 0. \quad \} \end{aligned}$$

Funzione	**Dominio**	**Operatore**	**Segno**	**Carattere**
#`count`	$\mathbb{N}$	$<, \leq$		antimonotono
			`not`	monotono
		$>, \geq$		monotono
			`not`	antimonotono
#`sum`	$\mathbb{N}$	$<, \leq$		antimonotono
			`not`	monotono
		$>, \geq$		monotono
			`not`	antimonotono
	$\mathbb{Z}$	$<, \leq$		*non-monotono*
			`not`	*non-monotono*
		$>, \geq$		*non-monotono*
			`not`	*non-monotono*
#`times`	$\mathbb{N} \setminus \{0\}$	$<, \leq$		antimonotono
			`not`	monotono
		$>, \geq$		monotono
			`not`	antimonotono
	$\mathbb{N}$	$<, \leq$		*non-monotono*
			`not`	*non-monotono*
		$>, \geq$		*non-monotono*
			`not`	*non-monotono*
#`min`	$\mathbb{N}$	$<, \leq$		monotono
			`not`	antimonotono
		$>, \geq$		antimonotono
			`not`	monotono
#`max`	$\mathbb{N}$	$<, \leq$		antimonotono
			`not`	monotono
		$>, \geq$		monotono
			`not`	antimonotono

Tabella 2.1: *Carattere dei letterali aggregati*

Figura 2.2: *Grafi* (a) $DG_{\mathcal{P}_4}$ *e* (b) $DG_{\mathcal{P}_5}$

In Figura 2.2 sono riportati i grafi delle dipendenze relativi a questi programmi.

$\mathcal{P}_4$ *è stratificato, mentre* $\mathcal{P}_5$ *è ricorsivo, in quanto è presente un ciclo fra i nodi* $p(0)$ *e* $q(0)$*.*

Possiamo, quindi, introdurre una classe importante e ben nota di programmi: quella dei programmi *head-cycle-free (HCF)*.

Definizione 3 *Un programma* $\mathcal{P}$ *è HCF se e solo se non c'è nessuna regola* r *in* $\mathcal{P}$ *tale che due predicati nella testa di* r *sono nello stesso ciclo di* $DG_{\mathcal{P}}$*.*

Esempio 2.14 *Consideriamo i programmi* $\mathcal{P}_4$ *e* $\mathcal{P}_5$ *dell'esempio precedente e il programma* $\mathcal{P}_6$*:*

$$\begin{aligned} \mathcal{P}_6 \;=\; \{ \;& p(0) \vee q(0). \,, \\ & q(0) \coloneq \#\texttt{count}\{\, \langle 0, p(0) \rangle \,\} > 0. \,, \\ & p(0) \coloneq \#\texttt{count}\{\, \langle 0, q(0) \rangle \,\} > 0. \quad \} \end{aligned}$$

Si noti che il grafo associato a $\mathcal{P}_6$ *coincide con* $DG_{\mathcal{P}_5}$*. Allora, abbiamo che*

- $\mathcal{P}_4$ *è aciclico, e quindi HCF;*
- $\mathcal{P}_5$ *è ciclico ed HCF;*
- $\mathcal{P}_6$ *è ciclico e non HCF, in quanto nella regola* $p(0) \vee q(0).$ *compaiono due letterali che appartengono ad uno stesso ciclo in* $DG_{\mathcal{P}_5}$*.*

Nella nostra implementazione per $\text{DLP}^{\mathcal{A}}_{m,a}$, descritta nel capitolo 8, consideriamo solo i programmi HCF. È stato recentemente mostrato che questa è la più grande classe di programmi per la quale i processi di reasoning standard sono ancora in NP (si veda, per esempio, [24]). Il principale risultato di questo capitolo, il Teorema 7.4, viene affermato per programmi HCF.

Possiamo partizionare l'insieme di atomi ground occorrenti in $\mathcal{P}$ in componenti fortemente connesse.

> Due atomi a e b sono nella stessa componente se c'è un cammino da a a b e un cammino da b ad a in $DG_{\mathcal{P}}$.

Possiamo inoltre definire un ordine parziale $\preceq$ per le componenti.

Definizione 4 *Siano C_1, C_2 due componenti di $\mathcal{P}$. Allora, $C_1 \preceq C_2$ se e solo se esistono $a \in C_1$, $b \in C_2$ tali che c'è un cammino da a a b.*

Il sottoprogramma $\mathcal{P}_C \subseteq \mathcal{P}$, associato ad una componente C, consiste di tutte le regole di $\mathcal{P}$ che contengono un atomo di C in testa.

2.5 Complessità computazionale

Dal punto di vista della complessità della programmazione logica con aggregati sono di rilevante interesse due aspetti computazionali:

- la verifica di stabilità di un modello,
- il reasoning cauto, o scettico (*cautious reasoning* o *skeptikal reasoning*).

La verifica di stabilità di un modello corrisponde al seguente problema decisionale:

> Dato un programma $\text{DLP}^{\mathcal{A}}$ $\mathcal{P}$ e un modello M,
> M è un modello stabile di $\mathcal{P}$?

Come mostrato in [1, 19], questo problema è in generale *coNP*-completo, ma diviene trattabile (ovvero polinomiale) se vengono disabilitati i letterali aggregati non-monotoni e la disgiunzione.

Il cautious reasoning, invece, corrisponde al problema:

> Dato un programma DLP$^{\mathcal{A}}$ $\mathcal{P}$ e un atomo ground standard $\mathcal{A}$,
> $\mathcal{A}$ è vero in tutti i modelli stabili di $\mathcal{P}$?

[1] mostra che questo problema è in generale Π_2^P-completo. L'eliminazione della disgiunzione e dei letterali aggregati non-monotoni porta il problema ad essere *coNP*-completo, mentre la trattabilità viene raggiunta solo se si eliminano la negazione e tutti i tipi di letterali aggregati; in questo caso, infatti, il programma può essere riscritto come una *CNF* con sole clausole di *Horn*, che è ben noto essere *P*-completo.

I risultati di complessità computazionale descritti in [1, 19] sono riassunti in Tabella 2.2 e in Tabella 2.3.

Checking	$\emptyset$	$\{\texttt{not}\}$	$\{\vee\}$	$\{\texttt{not}, \vee\}$
$\emptyset$	*P*	*P*	*coNP*	*coNP*
$\{m, a\}$	*P*	*P*	*coNP*	*coNP*
$\{m, a, n\}$	*coNP*	*coNP*	*coNP*	*coNP*

Tabella 2.2: *Complessità del controllo di stabilità*

Cautious	$\emptyset$	$\{\texttt{not}\}$	$\{\vee\}$	$\{\texttt{not}, \vee\}$
$\emptyset$	*P*	*coNP*	Π_2^p	Π_2^p
$\{m, a\}$	*coNP*	*coNP*	Π_2^p	Π_2^p
$\{m, a, n\}$	Π_2^p	Π_2^p	Π_2^p	Π_2^p

Tabella 2.3: *Complessità del cautious reasoning*

Le righe indicano i tipi di aggregato presenti nel linguaggio: *m* - monotoni, *a* - antimonotoni, *n* - nonmonotoni. Le colonne, invece, indicano la presenza o l'assenza della negazione (`not`) o della disgiunzione ($\vee$).

2.6 Rappresentazione della conoscenza

Come già detto, la programmazione logica disgiuntiva consente la modellazione intuitiva di molti problemi complessi. L'aggiunta degli aggregati, più in generale degli aggregati ricorsivi, accresce le capacità rappresentative di questo paradigma.

Per concludere questo capitolo, mostriamo come alcuni problemi possono essere rappresentati in modo conciso ed elegante grazie agli aggregati.

Car Sequencing (CAR-SEQ)

tratto da Asparagus [1]

Un certo numero di automobili deve essere prodotto; non sono identiche, perché differenti opzioni sono disponibili come varianti del modello base. La catena di assemblaggio ha stazioni differenti che installano le varie opzioni (aria condizionata, copri-sole, ...). Queste stazioni sono state progettate per gestire una certa percentuale di automobili fra quelle che passano dalla catena di assemblaggio. Le automobili devono quindi essere sistemate in una sequenza che consenta ad ogni stazione di eseguire il proprio lavoro. Per esempio, se una particolare stazione può lavorare solo la metà delle automobili che passano dalla catena di assemblaggio, la sequenza deve essere fatta in modo che al più un'automobile ogni due richieda questa opzione.

In Figura 2.3 è riportata una codifica che fa uso di aggregati stratificati.

```
sequencing(I,C) v fail(I,C) :- classes(C), cars(I).

:- cars(I), not #count{ C : sequencing(I,C) }=1.
:- not #count{I : sequencing(I,C)}=N, carPerClass(C,N).
:- #count{S: sequencing(S,C), optInClass(C,O,1), +(I,SIZE,W), S>=I, S<W}>MAX,
   maxCarsInBlock(O,MAX), blockSize(O,SIZE), cars(I).
```

Figura 2.3: *Codifica di Car Sequencing*

Intuitivamente, la prima regola sfrutta il non-determinismo per individuare una sequenza: $sequencing(I, C)$ indica che la I-esima automobile nella catena

[1] http://asparagus.cs.uni-potsdam.de/?action=encodings&id=79

di assemblaggio è una vettura di classe C. I successivi constraint indicano che ad ogni posizione della sequenza deve essere associata una sola automobile, che per ogni classe deve essere prodotto esattamente il numero richiesto di automobili e che ogni stazione sia in grado di gestire la sequenza.

Social Golfer (SOC-GOLF)

tratto da Asparagus [2]

La coordinatrice di un golf club ha il seguente problema. Nel suo club, ci sono $m \times n$ golfisti, ognuno dei quali gioca a golf una volta a settimana, e sempre in gruppi di n. Le piacerebbe stilare un orario di gioco per questi golfisti, per p settimane, tale che nessun golfista giochi più di una volta con un altro golfista.

In Figura 2.4 è riportata una codifica per questo problema che fa uso di aggregati stratificati, mentre in Figura 2.5 abbiamo una codifica con aggregati ricorsivi.

```
plays(Golfer,Week,Group) v fails(Golfer,Week,Group) :-
                       groups(Group), players(Golfer), weeks(Week).

:- not #count{G :  plays(Golfer,W,G)}=1, weeks(W), players(Golfer).
:- not #count{Golfer : plays(Golfer,W,Group)} = SIZE,
   groups(Group), weeks(W), groupSize(SIZE).
:- #count{W: plays(G1,W,G), plays(G2,W,G) } > 1,
   players(G1), players(G2), G1 < G2.
```

Figura 2.4: *Codifica di Social Golfer (con aggregati stratificati)*

```
plays(Golfer,Week,Group) :-
   groups(Group), players(Golfer), weeks(Week),
   #count{ G : plays(Golfer,Week,G), G != Group } < 1,
   #count{ G : plays(G,Week,Group), G != Golfer } < SIZE, groupSize(SIZE).

:-  #count{W: plays(G1,W,G), plays(G2,W,G) } > 1,
    players(G1), players(G2), G1 < G2.
```

Figura 2.5: *Codifica di Social Golfer (con aggregati ricorsivi)*

Nella codifica con aggregati stratificati, la prima regola individua in modo non-deterministico un orario di gioco: $plays(Golfer, Week, Group)$ indica che il

[2] http://asparagus.cs.uni-potsdam.de/?action=encodings&id=78

golfista *Golfer*, nella settimana *Week*, giocherà nel gruppo *Group*. I constraint che seguono modellano i vincoli imposti dal problema, ovvero che ogni golfista giochi esattamente una volta a settimana, che ogni gruppo sia della dimensione giusta e che nessuna coppia di golfisti giochi più di una volta insieme.

La codifica con aggregati ricorsivi è più concisa. La prima regola afferma che ogni settimana un golfista può giocare in un certo gruppo se in quella stessa settimana non gioca già in un altro gruppo e se il gruppo non è già completo. Il successivo constraint, invece, impedisce che una coppia di golfisti giochi più di una volta insieme.

Knap-Sack (KNAP-SACK)

Il problema Knap-Sack è un problema combinatorio di ottimizzazione. Problemi simili si trovano spesso in teoria della complessità, crittografia, finanza e matematica applicata.

> Dato un insieme di oggetti, ognuno con un peso e un valore, determinare quali oggetti includere in una collezione, in modo tale che il peso totale non sia superiore ad un peso dato e il valore totale sia il più grande possibile.

In Figura 2.6 mostriamo una possibile istanza di Knap-Sack, in cui bisogna scegliere fra 5 oggetti diversi. Il peso degli oggetti è riportato in chilogrammi (kg), mentre il valore in dollari ($). Il peso massimo sostenibile è di 15 kg.

La variante decisionale di Knap-Sack corrisponde alla domanda

> "Possiamo selezionare oggetti per un valore di almeno V, senza eccedere il peso C?"

In Figura 2.7 riportiamo una codifica per questo problema che fa uso di aggregati ricorsivi antimonotoni. Con la prima regola indichiamo che un oggetto X può essere selezionato se il peso degli oggetti presenti nella bisaccia, con l'aggiunta di X, non eccede il massimo peso sostenibile. Il constraint verifica che il valore degli oggetti selezionati sia almeno pari al minimo valore voluto.

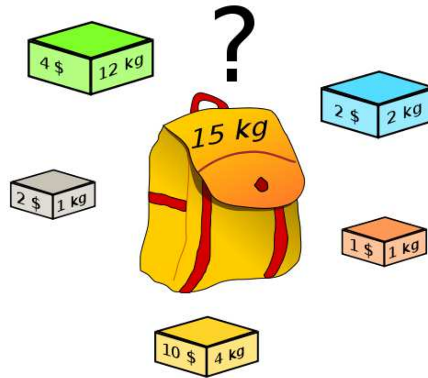


Figura 2.6: *Un'istanza di Knap-Sack*

```
select( X ) :- cost( X, C ), #sum{ C1, Y : sel( Y ), cost( Y, C1 ), X != Y } <= C2,
               MC = C2 + C, maxCost( MC ).

:- minValue( MV ), not #sum{ C, X : select( X ), value( X, C ) } >= MV.
```

Figura 2.7: *Codifica di Knap-Sack*

Party Invitatons (PARTY)

Si vuole organizzare un party e bisogna decidere a quali persone spedire gli inviti. È noto che ogni persona accetterà l'invito solo se almeno k dei propri amici lo accetteranno ($k \geq 0$). L'obiettivo è determinare le persone che verranno al party.

In Figura 2.8 riportiamo una codifica che fa uso di aggregati ricorsivi monotoni.

La prima regola definisce la relazione *friend* come simmetrica. La regola successiva determina le persone che verranno sicuramente alla festa. Si noti che questa è una regola ricorsiva, in quanto il predicato *coming* appare sia nella testa che nel corpo della regola (in particolare in un letterale aggregato).

```
friend(X,Y) :- friend(Y,X).
coming(X) :- requires(X,K), #count{ Y : friend(X,Y), coming(Y) } >= K.
```

Figura 2.8: *Codifica di Party Invitations*

Group Seating (SEATING)

Vogliamo disporre un gruppo di n persone in un ristorante, essendo a conoscenze che il numero di tavoli per il numero di posti a sedere di ogni tavolo è uguale ad n. Le persone che "si piacciono" devono sedersi allo stesso tavolo e quelle che "non si piacciono" devono sedersi in tavoli diversi.

In Figura 2.9 mostriamo una codifica che fa uso di aggregati ricorsivi antimonotoni.

```
at(P,T) :- person(P), table(T),
           #count{ T1 : at(P,T1), T1 != T } < 1,
           #count{ P1 : at(P1,T), P1 != P } < C, nchairs(C).

:- table(T), like(P1,P2), at(P1,T), not at(P2,T).
:- table(T), dislike(P1,P2), at(P1,T), at(P2,T).
```

Figura 2.9: *Codifica di Seating*

Con la prima regola determiniamo la disposizione delle persone nei tavoli: una persona può sedersi ad un tavolo se non si è già seduto da qualche altra parte e il tavolo non è ancora pieno. I constraint verificano che le persone che si piacciono siano sedute allo stesso tavolo e che quelle che non si piacciono siano sedute a tavoli diversi.

Employee Raise (EMPRAISE)

Un manager decide di selezionare un gruppo di N impiegati a cui dare un aumento. Un impiegato è un buon candidato per l'aumento se ha lavorato per almeno K ore a settimana. L'obiettivo è determinare tutti i possibili insiemi di N impiegati candidati.

In Figura 2.10 riportiamo una codifica per questo problema.

```
raised(X) :- empName(X), #sum{ H : emp(X,H) } >= K, nHours(K),
             #count{ Y : raised(Y), X != Y } < N, maxRaised(N).
```

Figura 2.10: *Codifica di Employee Raise*

Si noti che il programma è composto da una sola regola, e che in questa compaiono due aggregati: il primo stratificato e il secondo ricorsivo. Con questa regola affermiamo che l'aumento viene dato ad un impiegato se questo ha lavorato un numero sufficiente di ore e il numero previsto di aumenti non è stato già superato.

Capitolo 3

Programmi con modello stabile unico

In generale, un programma logico con aggregati può avere uno, nessuno o molti modelli stabili. Tuttavia, per un sottoinsieme della programmazione logica con aggregati è possibile dimostrare che esiste sempre un unico modello stabile.

In questo capitolo mostriamo due classi di problemi che godono di questa interessante proprietà.

Teorema 3.1 *Un programma $\mathcal{P}$ positivo e privo di disgiunzione, con soli aggregati monotoni, ha sempre un unico modello stabile, che coincide con il suo unico modello minimale.*

Dimostrazione. L'intersezione di due modelli è sempre un modello.

Siano M_1 ed M_2 due modelli di $\mathcal{P}$. Supponiamo, per assurdo, che $M := M_1 \cap M_2$ non è un modello. Allora esisterà una regola $r \in \mathcal{P}$ tale che

- $B(r)$ è vero rispetto ad M, e
- $H(r)$ è falso rispetto ad M.

Da $M \leq M_1$ e $M \leq M_2$ segue che $B(r) = Mon(B(r))$ è vero anche rispetto ad M_1 ed M_2. Inoltre, poiché $H(r)$ è vero rispetto ad M_1 ed M_2 (in quanto modelli) e $|H(r)| = 1$ (essendo $\mathcal{P}$ privo di disgiunzione), avremo che $H(r) \subseteq M_1$

e $H(r) \subseteq M_2$, il che implica $H(r) \subseteq M = M_1 \cap M_2$. Questa è una contraddizione con l'assunzione che $H(r)$ è falso rispetto ad M. □

Esempio 3.2 *La codifica del problema "Company Controls", mostrata nell'esempio 1.1, è positiva, priva di disgiunzione e con soli aggregati monotoni.*

Qualsiasi istanza di company controls, allora, avrà un unico modello stabile. Per l'istanza riportata in Figura 1.1 abbiamo che

- a *controlla direttamente* b*,*
- a *controlla indirettamente* c*; infatti,* a *possiede direttamente il* 40% *delle azioni di* c *e, indirettamente tramite* b*, il* 20%*, per un totale del* 60%*.*

Quindi, l'unico modello stabile contiene

$$controls(a, b), controls(a, c).$$

Teorema 3.3 *Un programma* $\mathcal{P}$ *privo di disgiunzione, con negazione stratificata, aggregati monotoni arbitrari, e aggregati antimonotoni e non-monotoni non-ricorsivi, ha sempre un unico modello stabile.*

Dimostrazione. Sia $C_1, C_2, ..., C_n$ un ordine totale (fra i possibili) per le componenti di $\mathcal{P}$ tale che $C_i \preceq C_j$ implica $i \leq j$.

Sia $\mathcal{P}_{C_i}$ il sottoprogramma relativo a C_i. Computiamo

$$\mathcal{P}_1 := \mathcal{P}_{C_1}$$

$$MM_1 = \text{ un modello minimale di } \mathcal{P}_1$$

e, per $i = 2, ..., n$

$$\mathcal{P}_i := MM_{i-1} \cup P_{C_i}$$

$$MM_i = \text{ un modello minimale per } \mathcal{P}_i.$$

Si noti che con $MM_{i-1} \cup P_{C_i}$ vogliamo indicare il programma ottenuto aggiungendo a P_{C_i} un fatto $\ell.$ per ogni letterale $\ell \in MM_{i-1}$.

Proviamo, per induzione su i, che MM_i è il solo modello minimale di $\mathcal{P}_i$.

Base. $\mathcal{P}_1$ è positivo e privo di disgiunzione, con soli aggregati monotoni. Dal Teorema 3.1 segue che MM_1 è il solo modello minimale di $\mathcal{P}_1$.

Supponiamo che MM_i è il solo modello minimale di $\mathcal{P}_i$, e proviamo che MM_{i+1} è il solo modello minimale di $\mathcal{P}_{i+1}$.

Siano M_1 ed M_2 due modelli di $\mathcal{P}_{i+1}$. Supponiamo, per assurdo, che $M = M_1 \cap M_2$ non è un modello di $\mathcal{P}_{i+1}$. Allora esisterà una regola $r \in \mathcal{P}$ tale che

- $B(r)$ è vero rispetto ad M,
- $H(r)$ è falso rispetto ad M.

Poiché $M \leq M_1$ e $M \leq M_2$, $Mon(B(r))$ è vero anche rispetto ad M_1 ed M_2.

Si noti, inoltre, che il valore di verità di ogni letterale standard negativo e di ogni letterale aggregato antimonotono e non-monotono appartenenti a $B(r)$ è determinato dai letterali in MM_i (perché $\mathcal{P}$ ha solo negazione stratificata e aggregati antimonotoni e non-monotoni non-ricorsivi). Quindi questi valori sono gli stessi per ogni modello di $\mathcal{P}_{i+1}$.

Segue che $B(r)$ è vero anche rispetto ad M_1 ed M_2 e, poiché questi sono modelli, anche $H(r)$ sarà vero rispetto ad M_1 ed M_2. Questo implica che $H(r)$ è vero anche rispetto ad M, poiché $|H(r)| = 1$, che contraddice l'assunzione.

Chiaramente P_n ha gli stessi modelli di $\mathcal{P}$. Quindi MM_n è l'unico modello stabile di $\mathcal{P}$. □

Esempio 3.4 *Consideriamo una variante di company controls, che chiameremo "Undirect Company Controls", in cui siamo interessati a determinare le compagnie che controllano indirettamente altre compagnie.*

Estendiamo la codifica per company controls presentata nell'Esempio 1.1 con il predicato $undirect_controls(X,Y)$ *e la regola*

$$undirect_controls(X,Y) \quad :- \quad controls(X,Y), \\ \texttt{not}\ \#\texttt{sum}\{\ S\ :\ owns(X,Y,S)\ \} > 50.$$

Intuitivamente, determiniamo prima tutte le istanze del predicato $controls(X,Y)$ *e, successivamente, scartiamo quelle per cui la compagnia* X *controlla direttamente la compagnia* Y.

È da notare che quest'ultima regola aggiunta al programma contiene un letterale aggregato antimonotono, ma che questo non è ricorsivo. Dal teorema precedente, quindi, ogni istanza di undirect company controls avrà un unico modello stabile.

Se riprendiamo l'esempio in Figura 1.1, possiamo osservare che l'unico controllo indiretto è quello di a *su* b*. Infatti, l'unico modello stabile contiene*

$$undirect_controls(a,c).$$

Capitolo 4

Caratterizzazione dei modelli stabili attraverso gli insiemi infondati

In questo capitolo presentiamo una definizione di *insieme infondato* (o *unfounded set*) per programmi DLP$^{\mathcal{A}}$ con aggregati monotoni e antimonotoni, estendendo quella presentata in [19]. Successivamente mostriamo come utilizzare gli insiemi infondati per caratterizzare i modelli stabili.

4.1 Insiemi infondati

Nel seguito denotiamo con $S_1 \dot{\cup} \neg.S_2$ l'insieme $(S_1 \setminus S_2) \cup \neg.S_2$, dove S_1 ed S_2 sono insiemi di letterali ground standard.

Definizione 5 (Insieme infondato) *Un insieme X di atomi ground è un insieme infondato, o unfounded set, per un programma* DLP$^{\mathcal{A}}_{m,a}$ $\mathcal{P}$ *rispetto ad un'interpretazione I se, per ogni regola $r \in \mathcal{P}$ tale che $H(r) \cap X \neq \emptyset$, almeno una delle seguenti condizioni vale:*

(1) $Ant(B(r))$ è falso rispetto ad I,

(2) $Mon(B(r))$ è falso rispetto ad $I \dot{\cup} \neg.X$,

(3) $H(r)$ *è vero rispetto ad* $I \,\dot{\cup}\, \neg.X$.

La condizione (1) dichiara che la soddisfazione della regola non dipende dagli atomi in X, mentre le condizioni (2) e (3) assicurano che la regola è soddisfatta anche se agli atomi in X viene associato il valore falso. Si noti che la condizione (3) è equivalente a $(H(r) \setminus X) \cap I \neq \emptyset$, e che l'insieme vuoto $\emptyset$ è sempre un insieme infondato, indipendentemente dall'interpretazione e dal programma.

Esempio 4.1 *Consideriamo il seguente programma:*

$$\begin{array}{rcll} \mathcal{P}_7 & = & \{ & a(1) \vee a(2).\,, \\ & & & a(1) \mathrel{:-} \#\texttt{count}\{\langle 1 \!:\! a(2)\rangle\} > 0.\,, \\ & & & a(2) \mathrel{:-} \#\texttt{count}\{\langle 1 \!:\! a(1)\rangle\} > 0. \quad \} \end{array}$$

e l'interpretazione $I_3 = \{a(1), a(2)\}$. *Allora,* $\{a(1)\}$ *e* $\{a(2)\}$ *sono insiemi infondati per* $\mathcal{P}_7$ *rispetto ad* I_3*, mentre* $\{a(1), a(2)\}$ *non è un unfounded set per* $\mathcal{P}_7$ *rispetto ad* I_3*, poiché per la prima regola non si verifica nessuna delle tre condizioni di infondatezza rispetto a questa interpretazione.*

Teorema 4.2 *Per un programma privo di aggregati e un'interpretazione* I*, gli insiemi infondati rispetto alla Definizione 5 corrispondono agli insiemi infondati definiti in [25].*

Dimostrazione. Un insieme X è un insieme infondato rispetto alla Definizione 3.1 di [25] se, per ogni regola r tale che $H(r) \cap X \neq \emptyset$, vale almeno una delle seguenti condizioni:

(a) $B(r) \cap \neg.I \neq \emptyset$,

(b) $B^+(r) \cap X \neq \emptyset$,

(c) $(H(r) \setminus X) \cap I \neq \emptyset$.

La Definizione 5, nel caso privo di aggregati (ovvero se $Mon(B(r)) = B^+(r)$ e $Ant(B(r)) = B^-(r)$ per ogni regola $r \in \mathcal{P}$) è equivalente a:

Per ogni regola $r \in \mathcal{P}$ tale che $H(r) \cap X \neq \emptyset$

(1) $B^-(r) \cap \neg.I \neq \emptyset$ oppure

(2) $B^+(r) \cap \neg.(I \dot{\cup} \neg.X) \neq \emptyset$ oppure

(3) $H(r) \cap (I \dot{\cup} \neg.X) \neq \emptyset$.

(2) è equivalente a $B^+(r) \cap (\neg.(I \setminus X) \cup \neg.\neg.X) \neq \emptyset$, che vale se e solo se vale $B^+(r) \cap \neg.(I \setminus X) \neq \emptyset$ oppure $B^+(r) \cap X \neq \emptyset$. Poiché X contiene solo atomi, $\neg.X$ conterrà solo letterali negativi, perciò $B^+(r) \cap \neg.X = \emptyset$. Quindi (2) equivale a (2.a) $B^+(r) \cap \neg.I \neq \emptyset$ o (2.b) $B^+(r) \cap X \neq \emptyset$. Osserviamo che "(1) o (2.a)" è equivalente ad (a), mentre (2.b) è equivalente a (b).

Infine, (3) equivale a $H(r) \cap ((I \setminus X) \cup \neg.X) \neq \emptyset$ e, poiché $H(r)$ e X contengono solo letterali positivi, questo è equivalente a (c). □

La Proposizione 3.3 in [25] afferma che gli insiemi infondati di [25] generalizzano gli insiemi infondati "originali" di [26], che erano definiti per programmi standard non-disgiuntivi. Perciò, dal Teorema 4.2 e dalla Proposizione 3.3 in [25], segue che anche gli insiemi infondati della Definizione 5 generalizzano quelli di [26] nel caso non-disgiuntivo privo di aggregati.

Corollario 4.3 *Per un programma non-disgiuntivo e privo di aggregati $\mathcal{P}$ e un'interpretazione I, ogni insieme infondato rispetto alla Definizione 5 è un insieme infondato standard (secondo la definizione in [26]).*

La Definizione 5 di insieme infondato generalizza anche quella data in [19] per programmi non-disgiuntivi con aggregati monotoni e antimonotoni.

Teorema 4.4 *Per un programma non-disgiuntivo $\text{LP}^{\mathcal{A}}_{m,a}$ $\mathcal{P}$ e un'interpretazione I, la Definizione 5 è equivalente alla Definizione 2 in [19].*

Dimostrazione. Un insieme X è un insieme infondato rispetto a [19] se, per ogni regola $r \in \mathcal{P}$ con $H(r) \in X$, almeno una delle seguenti condizioni vale:

(a) qualche letterale antimonotono del corpo di r è falso rispetto a I,

(b) qualche letterale monotono del corpo di r è falso rispetto a $I \dot{\cup} \neg.X$.

Chiaramente (a) e (b) sono equivalenti a (1) e (2), rispettivamente. Inoltre, la condizione (3) è sempre falsa per un programma non-disgiuntivo, poiché da $H(r) \cap X \neq \emptyset$ e $|H(r)| = 1$ segue che $H(r) \setminus X = \emptyset$. □

Possiamo infine dimostrare che essa coincide, nel frammento $\text{DLP}^{\mathcal{A}}_{m,a}$, con quella più generale presentata in [27].

Teorema 4.5 *Un insieme X di atomi ground è un insieme infondato per un programma $\text{DLP}^{\mathcal{A}}_{m,a}$ $\mathcal{P}$ rispetto ad un'interpretazione I in accordo con la Definizione 5 se e solo se è un insieme infondato per $\mathcal{P}$ rispetto ad I in accordo con la Definizione 1 di [27].*

Dimostrazione. In accordo con la Definizione 1 di [27], un insieme X di atomi ground è un insieme infondato per un programma $\mathcal{P}$ rispetto ad un'interpretazione I se, per ogni regola r in $\mathcal{P}$ che ha qualche atomo di X in testa, almeno una delle seguenti condizioni è verificata:

(a) qualche letterale in $B(r)$ è falso rispetto ad I,

(b) qualche letterale in $B(r)$ è falso rispetto ad $I \dot{\cup} \neg.X$,

(c) qualche atomo in $H(r) \setminus X$ è vero rispetto ad I.

Osserviamo che le condizioni (1) e (2) della Definizione 5 implicano rispettivamente le condizioni (a) e (b), e che, come già osservato in precedenza, la condizione (3) della Definizione 5 è quivalente alla condizione (c).

Ora, si osservi che se un letterale monotono del corpo è falso rispetto ad I, sarà falso anche rispetto a $I \dot{\cup} \neg.X$. Similmente, se un letterale antimonotono di r è falso rispetto a $I \dot{\cup} \neg.X$, allora sarà falso anche rispetto a I. Pertanto

- se la condizione (a) vale per un letterale monotono ℓ, la condizione (2) varrà per ℓ;

- se la condizione (a) vale per un letterale antimonotono ℓ, varrà la condizione (1) per ℓ;
- se la condizione (b) vale per un letterale monotono ℓ, varrà la condizione (2) per ℓ;
- se la condizione (b) vale per un letterale antimonotono ℓ, varrà la condizione (1) per ℓ.

□

La prossima proposizione mostra una proprietà di monotonicità molto importante della nostra definizione di insieme infondato.

Proposizione 1 *Sia I un'interpretazione parziale per un programma* $\text{DLP}^{\mathcal{A}}_{m,a}$ $\mathcal{P}$ *e X un insieme infondato per $\mathcal{P}$ rispetto ad I.*

Allora, per ogni $J \supseteq I$, X è un insieme infondato per $\mathcal{P}$ anche rispetto a J.

Dimostrazione. Se X è un insieme infondato per $\mathcal{P}$ rispetto ad I, allora per ogni $a \in X$ e per ogni $r \in \mathcal{P}$ con $a \in H(r)$ vale

(1) $Ant(B(r))$ è falso rispetto ad I,

(2) $Mon(B(r))$ è falso rispetto ad $I \dot{\cup} \neg.X$,

(3) $H(r)$ è vero rispetto ad $I \dot{\cup} \neg.X$.

Si noti che, poiché $I \subseteq J$, varrà anche che $I \dot{\cup} \neg.X \subseteq J \dot{\cup} \neg.X$. Perciò, se vale (1) deve valere anche per J, e se valgono (2) o (3) allora devono valere anche per $J \dot{\cup} \neg.X$. □

Di seguito definiamo la nozione centrale di *Greatest Unfounded Set* (GUS).

Definizione 6 *Sia I un'interpretazione per un programma $\mathcal{P}$.*

Allora, $GUS_{\mathcal{P}}(I)$ (il GUS per $\mathcal{P}$ rispetto ad I) denota l'unione di tutti gli insiemi infondati per $\mathcal{P}$ rispetto ad I.

Dalla Proposizione 1 segue che il GUS di un'interpretazione I è sempre contenuto nel GUS di un superset (o sovrainsieme) di I.

Corollario 4.6 *Sia I un'interpretazione per un programma $\mathcal{P}$.*
Allora $GUS_{\mathcal{P}}(I) \subseteq GUS_{\mathcal{P}}(J)$, per ogni J tale che $I \supseteq J$.

È da notare che, a discapito del suo nome, non è garantito che il GUS sia sempre un insieme infondato. Nel caso non-disgiuntivo, l'unione di due insiemi infondati è anch'esso un insieme infondato, anche in presenza di aggregati monotoni e antimonotoni [19]. Per questi programmi il GUS è necessariamente un insieme infondato. Tuttavia, in presenza di regole disgiuntive questa proprietà viene persa, come mostrato in [25]. Per i programmi $\mathrm{DLP}^{\mathcal{A}}_{m,a}$ quindi il GUS non è garantito essere un insieme infondato.

Esempio 4.7 *Si consideri l'esempio 4.1. $GUS_{\mathcal{P}_7}(I_3) = \{a(1), a(2)\}$ che, come già detto, non è un insieme infondato.*

Osservazione 4.8 *Se X_1 e X_2 sono insiemi infondati per un programma $\mathrm{DLP}^{\mathcal{A}}_{m,a}$ $\mathcal{P}$ rispetto ad un'interpretazione I, allora $X_1 \cup X_2$ non è necessariamente un insieme infondato.*

Recentemente, in [27] è stato mostrato che per un programma $\mathrm{DLP}^{\mathcal{A}}$ $\mathcal{P}$ (anche con aggregati non-monotoni) e un'interpretazione I, se due insiemi infondati sono disgiunti da I, allora la loro unione è anch'essa un insieme infondato. Ovviamente, questo risultato vale anche per $\mathrm{DLP}^{\mathcal{A}}_{m,a}$.

Proposizione 2 *Se X_1 e X_2 sono insiemi infondati per un programma $\mathcal{P}$ rispetto ad un'interpretazione I, e vale $X_1 \cap I = \emptyset$ e $X_2 \cap I = \emptyset$, allora $X_1 \cup X_2$ è un insieme infondato per $\mathcal{P}$ rispetto ad I.*

Questo ci consente di definire la classe di interpretazioni unfounded-free, per le quali è garantito che il GUS sia un insieme infondato.

Definizione 7 (Interpretazioni Unfounded-free) *Sia I un'interpretazione per un programma $\mathcal{P}$.*

I è unfounfed-free (priva di infondatezza) se $I \cap X = \emptyset$ per ogni insieme infondato X per $\mathcal{P}$ rispetto ad I.

Come una facile conseguenza abbiamo che per le interpretazioni unfounded-free il GUS è sempre un insieme infondato.

Proposizione 3 *Sia I un'interpretazione unfounded-free per un programma $\mathcal{P}$.*

Allora $GUS_{\mathcal{P}}(I)$ è un insieme infondato.

Possiamo inoltre mostrare che le interpretazioni totali possiedono un'interessante, quanto importante, proprietà.

Proposizione 4 *Sia I un'interpretazione totale per un programma $\mathcal{P}$.*

Allora I è unfounded-free se e solo se nessun insieme non-vuoto $X \subseteq I^+$ è un insieme infondato per $\mathcal{P}$ rispetto ad I.

Dimostrazione. ($\Rightarrow$) Se un insieme non-vuoto Y di I^+ è un insieme infondato per $\mathcal{P}$ rispetto ad I, allora I non è unfounded-free.

($\Leftarrow$) Se I non è unfounded-free, allora esiste un sottoinsieme non-vuoto di I^+ che è un insieme infondato per $\mathcal{P}$ rispetto ad I. Sia X un insieme infondato per $\mathcal{P}$ rispetto ad I tale che $Y = X \cap I \neq \emptyset$. Si noti che $I \,\dot{\cup}\, \neg.X = I \,\dot{\cup}\, \neg.Y$. Quindi Y è anche un insieme infondato per $\mathcal{P}$ rispetto ad I. □

Gli insiemi infondati trovano diverse applicazioni nella computazione dei modelli stabili per un programma logico con aggregati: è possibile utilizzarli per eseguire il controllo di stabilità di un'interpretazione totale e per eliminare rami inutili della computazione.

4.2 Controllo di stabilità tramite insiemi infondati

In questa sezione mostriamo alcune importanti proprietà degli insiemi infondati della Definizione 5 relative ad interpretazioni totali, grazie alle quali è possibile

utilizzare gli insiemi infondati per caratterizzare i modelli e gli answer set di un programma DLP$^{\mathcal{A}}_{m,a}$. Mostriamo, inoltre, come queste caratterizzazioni possano essere elegantemente utilizzate per la verifica degli answer set, operazione che va sotto il nome di *answer set checking* o *controllo di stabilità dei modelli*.

Proposizione 5 *Sia M un'interpretazione totale per un programma $\mathcal{P}$.*

Allora M è un modello per $\mathcal{P}$ se e solo se M^- è un insieme infondato per $\mathcal{P}$ rispetto ad M.

Dimostrazione. ($\Leftarrow$) Assumiano che M non sia un modello per $\mathcal{P}$. Allora esisterà una regola $r \in \mathcal{P}$ tale che $B(r)$ è vero rispetto ad M e $H(r)$ è falso rispetto ad M. Perciò, $H(r) \subseteq M^-$ poiché M è totale. Quindi la condizione (3) della Definizione 5 non vale per M^-. Si noti che

$$M \dot{\cup} \neg.M^- = (M \setminus M^-) \cup \neg.M^- = M \cup \neg.M^- = M$$

Quindi, né la condizione (1) né la (2) valgono per M^-. Segue che M^- non è un insieme infondato per $\mathcal{P}$ rispetto ad M.

($\Rightarrow$) Assumiamo che M^- non sia un insieme infondato per $\mathcal{P}$ rispetto ad M. Allora c'è un $a \in M^-$ tale che esiste $r \in \mathcal{P}$, con $a \in H(r)$, per la quale

(1) $Ant(B(r))$ non è falso rispetto ad M,

(2) $Mon(B(r))$ non è falso rispetto ad $M \dot{\cup} \neg.M^- = M$,

(3) $H(r)$ non è vero rispetto ad $M \dot{\cup} \neg.M^- = M$.

Poiché M è totale, da (1) e (2) segue che $B(r)$ è vero rispetto ad M, e da (3) segue che $H(r)$ è falso rispetto a M. Quindi M non è un modello per $\mathcal{P}$. □

Possiamo mostrare che gli insiemi infondati caratterizzano anche i modelli minimali.

Proposizione 6 *Sia M un modello per un programma positivo $\mathcal{P}$.*

M è un modello minimale per $\mathcal{P}$ se e solo se è un'interpretazione unfounded-free.

Dimostrazione. ($\Leftarrow$) Se M non è minimale allora esiste un altro modello M_1 tale che $M_1^+ \subset M^+$, e quindi $X = M^+ \setminus M_1^+ \neq \emptyset$. Allora, per ogni $r \in \mathcal{P}$ tale che $H(r) \cap X \neq \emptyset$

(i) $H(r) \cap M_1^+ \neq \emptyset$ oppure

(ii) $Ant(B(r))$ è falso rispetto ad M_1 oppure

(iii) $Mon(B(r))$ è falso rispetto ad M_1.

Si noti che $M_1 = (M \setminus X) \cup \neg.X = M \dot{\cup} \neg.X$ e quindi:

- da (i) segue che $H(r)$ è vero rispetto ad $M \dot{\cup} \neg.X$,
- da (ii) segue che $Ant(B(r))$ è falso rispetto ad M (perchè $M_1 \leq M$),
- da (iii) segue che $Mon(B(r))$ è falso rispetto ad $M \dot{\cup} \neg.X$.

Perciò X è un insieme infondato per $\mathcal{P}$ rispetto ad M e quindi M non è unfounded-free.

($\Rightarrow$) Assumiamo, per contraddizione, che M non è unfounded-free. Allora, dalla Proposizione 4, esisterà un insieme non-vuoto $X \subseteq M^+$ che è un insieme infondato per $\mathcal{P}$ rispetto ad M. Mostriamo che l'interpretazione totale $M_1 = M \dot{\cup} \neg.X$ è un modello per $\mathcal{P}$ (contraddicendo la minimalità di M).

Sia r una regola di $\mathcal{P}$ tale che $H(r)$ è vera rispetto ad M e $H(r)$ è falsa rispetta ad M_1. Allora $H(r) \cap X \neq \emptyset$. Ma X è un insieme infondato per $\mathcal{P}$ rispetto ad M, quindi:

(1) $Ant(B(r))$ è falso rispetto ad M (e quindi è falso rispetto ad M_1, perché $M_1 \leq M$) oppure

(2) $Mon(B(r))$ è falso rispetto ad $M \dot{\cup} \neg.X = M_1$ oppure

(3) $H(r)$ è vero rispetto ad $M \dot{\cup} \neg.X = M_1$.

Si noti che (3) non può valere per assunzione. Allora r è soddisfatta rispetto ad M_1 dal corpo, contraddicendo la minimalità di M. □

È possibile dimostrare che un'interpretazione unfounded-free per un programma rimane tale anche per il suo ridotto.

Lemma 4.9 *Sia M un'interpretazione totale per un programma $\mathcal{P}$.*
M è unfounded-free per $\mathcal{P}$ se e solo se è unfounded-free per $\mathcal{P}^M$.

Dimostrazione. $(\Rightarrow)$ Se X non è un unfounded set per $\mathcal{P}$ rispetto ad M, allora per qualche $a \in X$ esiste una regola $r \in \mathcal{P}$ tale che r viola tutte le condizioni della Definizione 5. Dalle condizioni (1) e (2) $B(r)$ è vero rispetto ad M. Pertanto l'immagine r' di r è in $\mathcal{P}^M$. Chiaramente r' viola tutte le condizioni della Definizione 5 per $\mathcal{P}^M$ rispetto ad M.

Se M è unfounded-free per $\mathcal{P}$, allora, dalla Proposizione 4, ogni sottoinsieme non-vuoto $X \subseteq M^+$ non è un insieme infondato per $\mathcal{P}$ rispetto ad M, e quindi non è un insieme infondato per $\mathcal{P}^M$ rispetto ad M. Pertanto M è unfounded-free per $\mathcal{P}^M$.

$(\Leftarrow)$ Sia X un insieme infondato per $\mathcal{P}$ rispetto ad M. Allora per ogni $a \in X$ e per ogni $r \in \mathcal{P}$ con $a \in H(r)$

(1) $Ant(B(r))$ è falso rispetto ad M oppure

(2) $Mon(B(r))$ è falso rispetto ad $M \dot{\cup} \neg.X$ oppure

(3) $H(r)$ è vero rispetto ad $M \dot{\cup} \neg.X$.

I casi (1) e (2) implicano che o r non ha un'immagine in $\mathcal{P}^M$ oppure la condizione (2) vale per la sua immagine r'. Il caso (3) implica che questa valga anche per r'. Perciò X è un insieme infondato anche per $\mathcal{P}^M$ rispetto ad M. Quindi se M non è unfounded-free per $\mathcal{P}$, allora non è unfounded-free per $\mathcal{P}^M$. Segue che M unfounded-free per $\mathcal{P}^M$ implica M unfounded-free per $\mathcal{P}$. □

Questi risultati consentono di verificare se un modello è stabile semplicemente utilizzando la nozione di insieme infondato.

Proposizione 7 *Sia M un modello per $\mathcal{P}$.*

M è un modello stabile per $\mathcal{P}$ se e solo se M è unfounded-free per $\mathcal{P}$.

Dimostrazione. ($\Rightarrow$) Se M è un modello stabile per $\mathcal{P}$, allora M è un modello minimale per $\mathcal{P}^M$. Dalla Proposizione 6 segue che M è unfouded-free per $\mathcal{P}^M$. Quindi, dal Lemma 4.9, M è unfounded-free per $\mathcal{P}$.

($\Leftarrow$) Sia M un modello unfounded-free per $\mathcal{P}$. Chiaramente M è un modello per $\mathcal{P}^M$ e quindi, dal Lemma 4.9, M è unfounded-free per $\mathcal{P}^M$. Perciò, dal Lemma 4.9, M è un modello minimale per $\mathcal{P}^M$ (quindi un modello stabile per $\mathcal{P}$).
□

4.3 Taglio dello spazio di ricerca attraverso gli insiemi infondati

In questa sezione mostriamo alcune proprietà del GUS di un programma $\mathcal{P}$ rispetto ad un'interpretazione I, $GUS_{\mathcal{P}}(I)$. Queste proprietà possono essere usate durante la computazione dei modelli stabili per determinare rami della computazione che non portano a nessuna soluzione. È quindi possibile eseguire il *pruning* (letteralmente *taglio*) di questi rami, riducendo notevolmente lo spazio di ricerca.

Teorema 4.10 *Sia I un'interpretazione per un programma $\mathcal{P}$ tale che $I \cap GUS_{\mathcal{P}}(I) \neq \emptyset$.*

Allora nessuna totalizzazione di I è un modello stabile per $\mathcal{P}$.

Dimostrazione. Se $I \cap GUS_{\mathcal{P}}(I) \neq \emptyset$, allora esisterà un insieme infondato X per $\mathcal{P}$ rispetto ad I tale che $I \cap X \neq \emptyset$. Sia J una totalizzazione di I. Allora, dalla Proposizione 1, X è un insieme infondato per $\mathcal{P}$ rispetto a J. Chiaramente $J \cap X \neq \emptyset$, perciò J non è unfounded-free. Dalla Proposizione 7 segue che J non è un modello stabile per $\mathcal{P}$. □

Perciò, durante la costruzione dei modelli stabili, possiamo computare il GUS rispetto all'interpretazione corrente e testare se contiene qualche elemento dell'interpretazione stessa. Se questo avviene possiamo abbandonare la costruzione e fare backtracing, poiché nessun modello stabile può essere trovato nel ramo corrente.

Il GUS può essere utilizzato con profitto per il pruning dello spazio di ricerca anche come operatore di inferenza.

Teorema 4.11 *Data un'interpretazione I per un programma $\mathcal{P}$, se J è un modello stabile contenente I, allora J contiene anche $I \dot{\cup} \neg.GUS_{\mathcal{P}}(I)$.*

Dimostrazione. Assumiamo che $J \not\supseteq I \dot{\cup} \neg.GUS_{\mathcal{P}}(I)$, allora $J \cap GUS_{\mathcal{P}}(I) \neq \emptyset$. Dal Corollario 4.6 segue che $J \cap GUS_{\mathcal{P}}(J) \neq \emptyset$. Quindi, in virtù del Teorema 4.10, J non è un modello stabile per $\mathcal{P}$. □

In altre parole $\neg.GUS_{\mathcal{P}}(I)$ è contenuto in tutti i modelli stabili che estendono I; perciò, durante la costruzione dei modelli stabili candidati, è possibile aggiungere in modo sicuro questi letterali.

Il GUS può essere utilizzato anche per verificare se un'interpretazione totale è unfounded-free e, quindi, un modello stabile.

Teorema 4.12 *Sia I un modello per un programma $\mathcal{P}$.*

I è unfounded-free se e solo se $I^- = GUS_{\mathcal{P}}(I)$.

Dimostrazione. ($\Leftarrow$) È facile vedere che ogni insieme infondato X per $\mathcal{P}$ rispetto ad I è un sottoinsieme di I^-. Quindi vale che $I \cap X = \emptyset$.

($\Rightarrow$) Per ogni insieme infondato X per $\mathcal{P}$ rispetto I, vale $I \cap X = \emptyset$. Poiché I è totale, questo è equivalente a $X \subseteq I^-$, e quindi $GUS_{\mathcal{P}}(I) \subseteq I^-$. Dalla Proposizione 5, segue che $I^- \subseteq GUS_{\mathcal{P}}(I)$, e quindi $I^- = GUS_{\mathcal{P}}(I)$. □

Come conseguenza della Proposizione 7 e del Teorema 4.12, possiamo enunciare il seguente corollario.

Corollario 4.13 *Data un'interpretazione totale I per un programma $\mathcal{P}$, I è un modello stabile se e solo se $I^- = GUS_{\mathcal{P}}(I)$.*

Questi risultati consentono un notevole taglio dello spazio di ricerca inferendo deterministicamente nuovi letterali negativi e individuando interpretazioni per le quali nessuna totalizzazione è un modello stabile.

Parte II

Algoritmi e aspetti computazionali

In questa parte della tesi descriviamo delle tecniche di computazione per la determinazione dei modelli stabili di un programma logico con aggregati ricorsivi.

Mostriamo prima, nel capitolo 5, come ottenere in modo efficiente l'istanziazione di un programma logico con aggregati ricorsivi, tramite delle ottimizzazioni al metodo semi-naive.

Nel capitolo 6 adattiamo alla programmazione logica con aggregati gli operatori per il taglio dello spazio di ricerca utilizzati nella programmazione logica standard e mostriamo come combinarli in modo efficiente.

Nel capitolo 7 presentiamo un algoritmo per computare modularmente il GUS per un programma rispetto ad un'interpretazione.

Capitolo 5

Istanziazione efficiente

In questo capitolo riprendiamo i concetti di istanziazione presentati nel capitolo 2 e mostriamo una strategia efficiente per la computazione della versione ground di un programma logico con aggregati. In questo capitolo, quindi, consideriamo programmi non-ground.

5.1 Istanziazione di un programma

Data una regola non-ground r (ovvero con variabili), definiamo le istanze di r come tutte le regole ground ottenibili con una sostituzione applicata ad r.

Il grounding è il processo di istanziazione di un programma logico, ovvero si occupa di sostituire ogni regola con tutte le sue istanze. Lo scopo è di ottenere un programma logico ground i cui modelli stabili coincidano con quelli del programma in input.

Questo può essere ottenuto applicando ogni possibile sostituzione (con codominio nell'Universo di Herbrand) alle variabili di una regola, ma spesso non tutte queste istanze sono utili ai fini della computazione dei modelli stabili. Per esempio, molte di queste conterranno letterali che non potranno essere veri in alcun modello stabile.

Quindi una buona strategia di istanziazione è necessaria per ottenere un solver efficiente. DLV è forte di un ottimo modulo di grounding, che implementa un algoritmo di back-jumping e una tecnica semi-naive per la computazione dei join fra predicati [**?**].

Nel modulo di grounding vengono distinti due tipi di regole:

- le regole *ricorsive* e
- le regole di *exit* (o non ricorsive).

Durante l'istanziazione vengono prima processate le regole di exit. Si procede quindi al processamento iterato delle sole regole ricorsive, fino al raggiungimento di un punto fisso (ovvero finché non viene generata nessuna regola nuova).

5.2 Istanziazione degli aggregati

Per facilitare l'istanziazione degli aggregati sostituiamo la congiunzione *Conj* del multi-insieme simbolico con un atomo ausiliario $aux(\overline{X})$, dove $\overline{X}$ sono i termini che compaiono in *Conj*, e aggiungiamo la regola

$$aux(\overline{X}) :- Conj.$$

Il grounding degli aggregati non ricorsivi è facilitato dal fatto che all'atto dell'istanziazione tutte le istanze dell'atomo ausiliario sono note. Pertanto è possibile eseguire l'istanziazione tramite un matching delle istanze dell'atomo ausiliario associato all'aggregato. Proprio per la loro natura, questa proprietà viene persa negli aggregati ricorsivi. Infatti è possibile che nel corso delle iterazioni successive venga generata qualche altra istanza dell'atomo ausiliario.

Una prima soluzione a questo problema può essere ignorare gli aggregati ricorsivi durante il grounding dell'intera componente e processarli una volta che sono stati generati tutti gli atomi appartenenti alla componente corrente: è chia-

ro che a questo punto tutte le possibili istanze dell'atomo ausiliario saranno state istanziate e quindi il matching dell'aggregato può essere fatto in modo sicuro.

Va detto che questa strategia, seppur corretta, non è adatta ad un'implementazione efficiente degli aggregati ricorsivi nella programmazione logica. Procedendo in questo modo, infatti, vengono generati "tutti" gli atomi che sono coinvolti in un ciclo di ricorsione. Per esempio, nel caso di company controls (si veda l'Esempio 1.1) per n compagnie si otterrebbero ben n^2 istanze della regola

$$\begin{aligned} controls(X,Y) \quad :- \quad & company(X), company(Y), \\ & \#\texttt{sum}\{\ S, Z\ :\ cv(X,Y,Z,S)\ \} > 50. \end{aligned}$$

Per gli aggregati monotoni (come per esempio quelli coinvolti in company controls) è possibile adottare una strategia più restrittiva e più efficiente, che ne sfrutti la proprietà di monotonicità. Ricordiamo che un letterale monotono vero rispetto ad un'interpretazione I sarà vero anche in tutte le estensioni di I. Possiamo quindi eseguire il grounding degli aggregati monotoni durante l'istanziazione della regola.

(i) Se il letterale aggregato risulta essere vero, allora è possibile aggiungere la regola senza l'aggregato (l'aggiunta di un letterale vero nel corpo è superflua).

(ii) Se il letterale aggregato risulta essere falso, la regola non deve essere aggiunta in quanto soddisfatta dal corpo. La regola sarà riprocessata alla prossima iterazione, in quanto è possibile che la valutazione dell'aggregato cambi in seguito alla determinazione di nuove istanze dell'atomo ausiliario.

(iii) Se il letterale aggregato è indefinito, bisogna aggiungere la regola con l'aggregato.

Si noti che ulteriori miglioramenti a questa strategia sarebbero possibili.

- Nel caso in cui un aggregato sia stato già processato nell'iterazione precedente (e non sia risultato vero) sarebbe possibile eseguire il matching dell'atomo ausiliario sulle sole nuove istanze, aggiungendo queste a quelle determinate precedentemente.

- Se l'aggregato è risultato vero sarebbe utile evitare l'intera istanziazione.

- In caso di aggregati indefiniti è probabile che molte istanze diverse dello stesso aggregato (e quindi della stessa regola) siano generate: per esempio, potremmo avere la stessa regola in cui l'aggregato viene istanziato in iterazioni successive (potenzialmente su insiemi differenti). Solo l'ultima di queste istanze è quella realmente corretta, ma la presenza delle altre regole non compromette il buon funzionamento del sistema.

Realizzare queste modifiche tuttavia ha un costo. Occorrerebbe infatti ideare opportune strutture dati per ritrovare efficientemente le istanze di letterali aggregati generati ai passi precedenti.

Possiamo applicare questa tecnica anche ai letterali aggregati nonmonotoni equivalenti ad una disgiunzione di un letterale monotono e di uno antimonotono, considerando la sola parte monotona.

Bisogna inoltre osservare che questa strategia non è applicabile ai letterali aggregati antimonotoni (e in generale a quelli nonmonotoni); per questi viene mantenuta la strategia di posticipazione dell'istanziazione al termine del grounding della componente.

5.3 Tecniche per l'istanziazione incrementale

5.3.1 Semi-naive per programmi logici tradizionali

Per eseguire il grounding di una regola, un solver per programmi logici deve sostituire le variabili globali che occorrono nella regola con, virtualmente, tutte le possibili costanti presenti nell'universo di Herbrand $B_{\mathcal{P}}$.

Ricordando il concetto di *safety* per una regola, ovvero che tutte le variabili che compaiono in una regola devono comparire in almeno un atomo positivo del corpo, viene naturale eseguire la sostituzione sui soli atomi determinati (come veri o indefiniti) nei passi precedenti.

Una tecnica semi-naive [28, **?**] può essere applicata ad ogni componente. Questa può essere vista come un algoritmo a due fasi:

- la prima processa le regole non-ricorsive (o di exit), che possono essere completamente valutate in un solo passo;
- la seconda processa le regole ricorsive, che necessitano della computazione iterativa di un punto fisso per la loro completa valutazione.

Ad ogni passo della computazione ci sono un certo numero di predicati le cui estensioni sono state già completamente determinate (ovvero i predicati appartenenti a qualche componente precedente, che sono quindi già completamente determinati), e un certo numero di predicati ricorsivi (che quindi appartengono alla componente) per i quali un nuovo insieme di valori di verità può essere determinato da quelli correntemente determinati.

Sia p_i uno di questi predicati ricorsivi. Indichiamo con Δp_i^k (e lo chiamiamo la *differenza* di p_i) l'insieme dei nuovi atomi determinati per p_i al passo k, mentre indichiamo con p_i^k gli atomi relativi a p_i determinati fino al passo k escluso.

Sia

$$p :- p_1, \ldots, p_n, q_1, \ldots, q_m.$$

una regola ricorsiva tale che

- $p_1, \ldots, p_n$ sono mutuamente ricorsivi con p,
- $q_1, \ldots, q_m$ non sono mutuamente ricorsivi con p.

Il metodo semi-naive mira a ridurre il più possibile il numero di atomi computati al passo k che sono già stati determinati al passo $k - 1$. Per poter raggiungere

questo scopo, il semi-naive valuta, ad ogni iterazione del punto fisso, la seguente formula:

$$\Delta p^{k+1} = \bigcup_{t=1,\ldots,n} \left(\Delta p_t^k \wedge \bigwedge_{i=1,\ldots,n \wedge i \neq t} p_i^k \wedge \bigwedge_{j=1,\ldots m} q_j^k \right)$$

Intuitivamente, il parametro t rappresenta un *token* che viene assegnato di volta in volta ai predicati ricorsivi: il predicato con il token viene valutato solo sulla sua differenza. Questo consente di minimizzare la taglia della prima relazione in join, che comporta un notevole vantaggio prestazionale.

In [**?**] viene fatta un'interessante osservazione su come questa tecnica possa essere ulteriormente migliorata, al fine di eseguire operazioni di join più leggere. Sempre in [**?**] viene presentato l'algoritmo *Differential semi-naive Evaluation*, che è attualmente implementato nel sistema DLV.

Nella prossima sezione presentiamo questa variante del metodo semi-naive. Mostriamo inoltre che non è direttamente applicabile al caso degli aggregati ricorsivi, ma che sono necessarie alcune piccole modifiche.

Ottimizzazione del semi-naive

Durante il grounding di una regola DLV associa ad ogni predicato un *range*, che indica al sistema su quali insiemi deve essere eseguito il matching dei predicati. Sono presenti tre tipi differenti di range.

- I indica che al passo $k+1$ il predicato deve essere matchato sugli atomi determinati fino al passo $k-1$, senza considerare gli atomi determinati al passo precedente k.

- ΔI indica che al passo $k+1$ il predicato deve essere matchato solo sugli atomi determinati al passo precedente k.

- $I \cup \Delta I$ indica che al passo $k+1$ il predicato deve essere matchato su tutti gli atomi determinati fino al passo k.

La differenza fra l'algoritmo semi-naive standard e quello implementato in DLV è principalmente incentrata su un'ottimizzazione nella gestione del token che viene assegnato ai diversi predicati:

- al predicato al quale è assegnato il token viene dato range ΔI;
- ai predicati ricorsivi che hanno già ricevuto il token (e a quelli non ricorsivi) viene assegnato range I;
- ai predicati ricorsivi che ancora non hanno ricevuto il token viene dato range $I \cup \Delta I$.

5.3.2 Semi-naive per programmi logici con aggregati

Nel caso di aggregati non ricorsivi questi avranno sempre associato range I. Pertanto saranno sempre istanziati sui soli atomi determinati al passo precedente.

In presenza di aggregati ricorsivi (monotoni) l'applicazione del semi-naive attualmente implementato in DLV non è sufficiente. In particolare non è sufficiente assegnare al letterale aggregato un range diverso da $I \cup \Delta I$. Infatti è possibile che un letterale aggregato che è falso rispetto ad I sia vero rispetto ad $I \cup \Delta I$. Chiariamo meglio questo concetto con un esempio.

Esempio 5.1 *Si consideri l'istanza $\mathcal{P}$ di Company Controls riportata in Figura 1.1.*

$$\begin{array}{ll} company(a). & owns(a, b, 60). \\ company(b). & owns(a, c, 40). \\ company(c). & owns(b, c, 20). \end{array}$$

La compagnia a controlla direttamente la compagnia b e, tramite questa, ha il controllo di c.

Il modulo di grounding di DLV *processerebbe prima la regola di exit*

$$cv(X, X, Y, S) :- owns(X, Y, S).$$

ottenendo le seguenti istanze di cv:

$$
\begin{array}{l}
cv(a,a,b,60). \\
cv(a,a,c,40). \\
cv(b,b,c,20).
\end{array}
$$

Verrebbero quindi processate le regole ricorsive

$$
\begin{array}{rcl}
cv(X,Z,Y,S) & :- & controls(X,Z), owns(Z,Y,S). \\
controls(X,Y) & :- & company(X), company(Y), \\
 & & \#\mathtt{sum}\{\, S,Z \;:\; cv(X,Z,Y,S)\} > 50.
\end{array}
$$

con

$$
\begin{array}{rcll}
I & = & \{ & company(a), company(b), company(c), \\
 & & & owns(a,b,60), owns(a,c,40), owns(b,c,20), \\
 & & & cv(a,a,b,60), cv(a,a,c,40), cv(b,b,c,20) \quad \} \\
\Delta I & = & \{ & \}
\end{array}
$$

dalle quali verrebbe istanziato

$$controls(a,b).$$

Poiché è stata derivata una nuova istanza di predicato, le regole saranno riprocessate con

$$
\begin{array}{rcll}
I & = & \{ & company(a), company(b), company(c), \\
 & & & owns(a,b,60), owns(a,c,40), owns(b,c,20), \\
 & & & cv(a,a,b,60), cv(a,a,c,40), cv(b,b,c,20) \quad \} \\
\Delta I & = & \{ & controls(a,b) \quad \}
\end{array}
$$

producendo

$$cv(a, b, c, 30).$$

É facile vedere che alla successiva iterazione, con

$$\begin{aligned}
I &= \{ \quad company(a), company(b), company(c), \\
&\qquad owns(a, b, 60), owns(a, c, 40), owns(b, c, 20), \\
&\qquad cv(a, a, b, 60), cv(a, a, c, 40), cv(b, b, c, 20), \\
&\qquad controls(a, b) \quad \} \\
\Delta I &= \{ \quad cv(a, b, c, 20) \quad \}
\end{aligned}$$

non verranno prodotte nuove istanze, in quanto né gli atomi in I né quelli in ΔI sono sufficienti ad istanziare l'aggregato

$$\#\mathtt{sum}\{\, S, Z \;:\; cv(a, Z, c, S)\} > 50\,.$$

Verrebbero prodotte, infatti, le seguenti istanze ground:

$$\begin{aligned}
&\#\mathtt{sum}\{\, \langle\, 40 \,:\, cv(a, a, c, 40)\,\rangle\} > 50\,, \\
&\#\mathtt{sum}\{\, \langle\, 20 \,:\, cv(a, b, c, 20)\,\rangle\} > 50\,,
\end{aligned}$$

mentre l'unica istanza corretta sarebbe

$$\#\mathtt{sum}\{\, \langle\, 40 \,:\, cv(a, a, c, 40)\,\rangle, \langle\, 20 \,:\, cv(a, b, c, 20)\,\rangle\} > 50\,.$$

Pertanto, occorre assegnare sempre range $I \cup \Delta I$ ai letterali aggregati ricorsivi. Abbiamo quindi una strategia di semi-naive leggermente diversa, che descriviamo brevemente nel prossimo paragrafo.

Semi-naive per programmi logici con aggregati ricorsivi

Se il predicato che riceve il token è un aggregato (monotono) gli assegnamo range $I \cup \Delta I$, altrimenti range ΔI. Ai predicati ricorsivi che hanno già ricevuto il token diamo range I, mentre a quelli che ancora non l'hanno ricevuto assegnamo range $I \cup \Delta I$. Naturalmente ai predicati non-ricorsivi diamo sempre range I. Dopodiché, se il predicato con il token non è un letterale aggregato ricorsivo, il token viene spostato al predicato successivo; altrimenti tutte le nuove tuple sono state già derivate e possiamo interrompere il processamento di questa regola.

Ottimizzazione per aggregati falsi su insieme vuoto

Per molti letterali aggregati è possibile eseguire un'ulteriore ottimizzazione, in modo da sfruttare maggiormente la tecnica semi-naive. Se il letterale aggregato necessita di almeno un'istanza del suo predicato ausiliario per essere valutato come vero (come per esempio $\#$`min` e $\#$`max`, o $\#$`count` e $\#$`sum` con guardia inferiore maggiore di zero), allora è possibile e conveniente aggiungere al corpo della regola il predicato ausiliario, proiettandone le variabili libere. In questo modo l'algoritmo semi-naive riconoscerà il predicato ausiliario come ricorsivo, limitando l'istanziazione della regola alle sole istanze del predicato ausiliario determinate al passo precedente.

Esempio 5.2 *Il letterale aggregato* $\#\mathtt{sum}\{\ S, Z\ :\ cv(X, Z, Y, S)\} > 50$ *nella codifica di Company Controls dell'esempio 1.1 è falso se non esistono istanze di* cv. *Quindi, possiamo aggiungere alla regola*

$$\begin{aligned} controls(X, Y) \quad :- \quad & company(X), company(Y), \\ & \#\mathtt{sum}\{\ S, Z\ :\ cv(X, Z, Y, S)\ \} > 50. \end{aligned}$$

l'atomo $cv(X, _, Y, _)$. *Il programma risultante è il seguente*

$$
\begin{aligned}
cv(X, X, Y, S) &\ \text{:–}\ \ owns(X, Y, S).\\
cv(X, Z, Y, S) &\ \text{:–}\ \ controls(X, Z), owns(Z, Y, S).\\
controls(X, Y) &\ \text{:–}\ \ company(X), company(Y), cv(X, _, Y, _),\\
&\qquad \#\texttt{sum}\{\ S, Z\ :\ cv(X, Z, Y, S)\ \} > 50.
\end{aligned}
$$

Pertanto, durante l'istanziazione verranno matchate solo le regole per cui esiste una nuova istanza dell'atomo $cv(X, _, Y, _)$.

Capitolo 6

Operatori per il taglio dello spazio di ricerca

In questo capitolo generalizziamo alcuni operatori monotoni utilizzati dal sistema DLV per il taglio dello spazio di ricerca. In particolare l'operatore well-founded, il quale riveste un ruolo importante nella computazione dei modelli stabili. Infatti, grazie all'uso del greatest unfounded set descritto nel capitolo 4, questo operatore riesce ad effettuare un taglio migliore rispetto agli altri, determinando un numero maggiore di letterali sicuramente falsi.

6.1 L'operatore di conseguenza logica immediata $\mathcal{T}_{\mathcal{P}}$

L'operatore di *conseguenza logica immediata* consente di derivare come veri atomi che sono conseguenza diretta di altri.

Definizione 8 *Sia $\mathcal{P}$ un programma e I un'interpretazione.*

Definiamo l'operatore di conseguenza immediata $\mathcal{T}_\mathcal{P}$ come segue:

$$\begin{array}{rcll} \mathcal{T}_\mathcal{P}(I) & = & \{a \in B_\mathcal{P} \mid & \exists r \in \mathcal{P}\ s.t.\ a \in H(r) : \\ & & & B(r) \textit{ is true w.r.t. } I, \\ & & and & H(r) \setminus \{a\} \subseteq I^-\}. \end{array}$$

Intuitivamente, dato un programma $\mathcal{P}$ e un'interpretazione I, l'operatore $\mathcal{T}_\mathcal{P}$ deriva come veri un insieme di atomi che sono conseguenza diretta dell'interpretazione applicata al programma.

Esempio 6.1 *Consideriamo l'Esempio 2.13. Per il programma $\mathcal{P}_4$ e l'interpretazione $I_4 = \{p(0)\}$, l'applicazione dell'operatore $\mathcal{T}_\mathcal{P}$ produrrà*

$$\mathcal{T}_{\mathcal{P}_4}(I_4) = \{p(0), s(0)\}\,.$$

Per il programma $\mathcal{P}_5$ dello stesso esempio, e l'interpretazione $I_5 = \emptyset$, avremo

$$\mathcal{T}_{\mathcal{P}_5}(I_5) = \emptyset\,.$$

Si noti che l'operatore di conseguenza immediata non è riuscito a derivare niente, mentre è facile vedere che sia $p(0)$ che $q(0)$ sono infondati.

Questo operatore è sicuramente insufficiente per programmi con negazione non stratificata e/o disgiunzione. Gli altri operatori che presentiamo in questo capitolo estendono $\mathcal{T}_\mathcal{P}$, aggiungendo a questo la determinazione di atomi falsi in tutti i modelli stabili che totalizzano l'interpretazione.

6.2 L'operatore di Fitting $\Phi_\mathcal{P}$

L'operatore di *Fitting*, oltre ad applicare l'operatore $\mathcal{T}_\mathcal{P}$, determina come falsi tutti quegli atomi che non hanno il supporto di alcuna regola (dove per supporto intendiamo una regola non già soddisfatta che abbia in testa l'atomo).

Definizione 9 *Sia $\mathcal{P}$ un programma, e I un'interpretazione.*

Definiamo $\gamma_{\mathcal{P}}(I)$ come l'insieme

$$\begin{aligned}\gamma_{\mathcal{P}}(I) \;=\; \{a \in B_{\mathcal{P}} \mid\; & \forall r \in \mathcal{P}\ s.t.\ a \in H(r): \\ & (a)\ B(r)\ is\ false\ w.r.t.\ I, \\ or\;\; & (b)\ H(r)\ is\ true\ w.r.t.\ I \,\dot{\cup}\, \neg.\{a\}\}.\end{aligned}$$

Definizione 10 *Sia $\mathcal{P}$ un programma, e I un'interpretazione.*

Definiamo l'operatore di Fitting $\Phi_{\mathcal{P}}$ come segue:

$$\Phi_{\mathcal{P}}(I) = \mathcal{T}_{\mathcal{P}}(I) \cup \neg.\gamma_{\mathcal{P}}(I).$$

Partendo da I definiamo la sequenza F_k:

$$\begin{aligned}&F_0 = I \\ &F_k = F_{k-1} \cup \Phi_{\mathcal{P}}(F_{k-1}), \quad k > 0.\end{aligned}$$

La sequenza incrementa monotonicamente e converge finitamente ad un limite che denotiamo con $\Phi_{P}^{\infty}(I)$.

Esempio 6.2 *Consideriamo ancora l'Esempio 2.13. Per il programma $\mathcal{P}_4$ e l'interpretazione $I_4 = \{p(0)\}$ l'applicazione dell'operatore $\Phi_{\mathcal{P}}^{\infty}(I)$ produrrà*

$$\Phi_{\mathcal{P}_4}^{\infty}(I_4) = \{p(0), s(0), \texttt{not}\ q(0)\}\,.$$

Per il programma $\mathcal{P}_5$ dello stesso esempio e l'interpretazione $I_5 = \emptyset$ avremo

$$\Phi_{\mathcal{P}_5}^{\infty}(I_5) = \emptyset\,.$$

Si noti che anche l'operatore di Fitting non è riuscito a derivare come falsi i letterali $p(0)$ e $q(0)$ dall'interpretazione I_5.

Per l'operatore di Fitting e per il suo punto fisso vale la proprietà di monotonicità.

Proposizione 8 *Dato un programma $\mathcal{P}$ e due interpretazioni I, J tali che $I \subseteq J$, vale che*

$$\Phi_{\mathcal{P}}(I) \subseteq \Phi_{\mathcal{P}}(J)\,.$$

Dimostrazione. Banalmente, poichè $\mathcal{T}_{\mathcal{P}}(I) \subseteq \mathcal{T}_{\mathcal{P}}(J)$ per definizione e la condizione "(a) o (b)" vale anche per $J \supseteq I$. □

Proposizione 9 *Dato un programma $\mathcal{P}$ e due interpretazioni I, J tali che $I \subseteq J$, vale che*

$$\Phi^{\infty}_{\mathcal{P}}(I) \subseteq \Phi^{\infty}_{\mathcal{P}}(J)\,.$$

Dimostrazione. Base. $\Phi_{\mathcal{P}}(I) \subseteq \Phi_{\mathcal{P}}(J)$ dalla Proposizione 8.

Supponiamo $\Phi^{i}_{\mathcal{P}}(I) \subseteq \Phi^{i}_{\mathcal{P}}(J)$.

Allora $\Phi^{i+1}_{\mathcal{P}}(I) = \Phi_{\mathcal{P}}(\Phi^{i}_{\mathcal{P}}(I)) \subseteq \Phi_{\mathcal{P}}(\Phi^{i}_{\mathcal{P}}(J)) = \Phi^{i+1}_{\mathcal{P}}(J)$. □

6.3 L'operatore well-founded $\mathcal{W}_{\mathcal{P}}$

L'operatore *well-founded* si distingue dall'operatore di Fitting per l'applicazione degli insiemi infondati in sostituzione dell'insieme γ.

Definizione 11 *Sia $\mathcal{P}$ un programma e I un'interpretazione unfounded-free.*

Definiamo l'operatore well-founded $\mathcal{W}_{\mathcal{P}}$ come segue:

$$\mathcal{W}_{\mathcal{P}}(I) = \mathcal{T}_{\mathcal{P}}(I) \cup \neg.GUS_{\mathcal{P}}(I).$$

Definiamo inoltre la sequenza W_k:

$$\begin{aligned} &W_0 = I \\ &W_k = W_{k-1} \cup \mathcal{W}_{\mathcal{P}}(W_{k-1}), \quad k > 0. \end{aligned}$$

La sequenza incrementa monotonicamente e converge finitamente ad un limite che denotiamo con $\mathcal{W}^{\infty}_{\mathcal{P}}(I)$.

Esempio 6.3 *Si consideri ancora una volta l'Esempio 2.13. Per il programma* $\mathcal{P}_4$ *e l'interpretazione* $I_4 = \{p(0)\}$ *l'applicazione dell'operatore* $\mathcal{W}_{\mathcal{P}}^{\infty}(I)$ *produrrà*

$$\mathcal{W}_{\mathcal{P}_4}^{\infty}(I_4) = \{p(0), s(0), \texttt{not}\ q(0)\}\,.$$

Per il programma $\mathcal{P}_5$ *dello stesso esempio e l'interpretazione* $I_5 = \emptyset$ *avremo*

$$\mathcal{W}_{\mathcal{P}_5}^{\infty}(I_5) = \{\texttt{not}\ p(0), \texttt{not}\ q(0))\}\,.$$

L'operatore well-founded, a differenza dei precedenti, riesce a derivare la falsità dei letterali $p(0)$ *e* $q(0)$.

Anche l'operatore well-founded e il suo punto fisso godono della proprietà di monotonicità.

Proposizione 10 *Sia* I *un'interpretazione per un programma* $\mathcal{P}$.
Allora $\mathcal{W}_{\mathcal{P}}(I) \subseteq \mathcal{W}_{\mathcal{P}}(J)$, *per ogni* $J \supseteq I$.

Dimostrazione. Dalla definizione di $\mathcal{T}_{\mathcal{P}}(I)$ e dal Corollario 4.6 □

Proposizione 11 *Sia* I *un'interpretazione per un programma* $\mathcal{P}$. *Allora*

$$\mathcal{W}_{\mathcal{P}}^{\infty}(I) \subseteq \mathcal{W}_{\mathcal{P}}^{\infty}(J)\,.$$

Dimostrazione. Base. $\mathcal{W}_{\mathcal{P}}(I) \subseteq \mathcal{W}_{\mathcal{P}}(J)$ dalla Proposizione 10.
Supponiamo $\mathcal{W}_{\mathcal{P}}^{i}(I) \subseteq \mathcal{W}_{\mathcal{P}}^{i}(J)$.
Allora $\mathcal{W}_{\mathcal{P}}^{i+1}(I) = \mathcal{W}_{\mathcal{P}}(\mathcal{W}_{\mathcal{P}}^{i}(I)) \subseteq \mathcal{W}_{\mathcal{P}}(\mathcal{W}_{\mathcal{P}}^{i}(J)) = \mathcal{W}_{\mathcal{P}}^{i+1}(J)$. □

6.4 Combinazione efficiente degli operatori

La computazione del GUS è in generale più costosa della computazione di γ, ma consente un taglio maggiore dello spazio di ricerca.

Esiste tuttavia un insieme di programmi per i quali il GUS non determina nulla di più rispetto a quanto determinato da γ. In questo caso, pertanto, sarà più conveniente applicare l'operatore di Fitting e non quello well-founded.

Proviamo innanzitutto che per programmi HCF l'operatore di Fitting determina un sottoinsieme di quanto computato dall'operatore well-founded.

Lemma 6.4 *Sia I un'interpretazione per un programma HCF $\mathcal{P}$. Allora*

$$\Phi_{\mathcal{P}}(I) \subseteq \mathcal{W}_{\mathcal{P}}(I)\,.$$

Dimostrazione. Dobbiamo provare $\gamma_{\mathcal{P}}(I) \subseteq GUS_{\mathcal{P}}(I)$.

Sia C una componente di $\mathcal{P}$. Per ogni $a \in \gamma_{\mathcal{P}}(I) \cap C$ e per ogni $r \in \mathcal{P}$ tale che $a \in H(r)$, vale che (a) $B(r)$ è falso rispetto ad I, oppure (b) $H(r)$ è vero rispetto ad $I \,\dot{\cup}\, \neg.\{a\}$. Poiché $I \,\dot{\cup}\, \neg.(\gamma_{\mathcal{P}}(I) \cap C) \leq I$, da (a) segue che $Ant(B(r))$ è falso rispetto ad I oppure $Mon(B(r))$ è falso rispetto ad $I \,\dot{\cup}\, \neg.(\gamma_{\mathcal{P}}(I) \cap C)$. Da (b) segue che $H(r)$ è vero rispetto ad $I \,\dot{\cup}\, \neg.(\gamma_{P}(I) \cap C)$, poiché a è il solo atomo di $H(r)$ in $\gamma_{\mathcal{P}}(I) \cap C$.

Allora, $\gamma_{\mathcal{P}}(I) \cap C$ è un insieme infondato per $\mathcal{P}$ rispetto ad I, e quindi è contenuto in $GUS_{\mathcal{P}}(I)$. Poiché

$$\gamma_{\mathcal{P}}(I) = \bigcup_{C \in comp(\mathcal{P})} \gamma_{\mathcal{P}}(I) \cap C$$

dove $comp(\mathcal{P})$ denota l'insieme delle componenti di $\mathcal{P}$, anche $\gamma_{\mathcal{P}}(I)$ è un sottoinsieme di $GUS_{\mathcal{P}}(I)$. □

Possiamo quindi dimostrare che per programmi aciclici vale anche il contrario, ovvero che il well-founded non determina niente di più rispetto all'operatore di Fitting.

Lemma 6.5 *Sia I un'interpretazione per un programma aciclico $\mathcal{P}$. Allora*

$$GUS_{\mathcal{P}}(I) \subseteq \Phi^{\infty}_{\mathcal{P}}(I)^{-}\,.$$

Dimostrazione. Mostriamo che ogni insieme infondato X per $\mathcal{P}$ rispetto ad I è un sottoinsieme di $\Phi_{\mathcal{P}}^{\infty}(I)^-$.

Sia $C_1, C_2, ..., C_n$ un ordine totale (fra i possibili) per le componenti di $\mathcal{P}$, tale che $C_i \preceq C_j$ se e solo se $i \leq j$. Poiché

$$X = \bigcup_{i=1}^{n} X \cap C_i$$

possiamo provare, per induzione su i, che $X \cap C_i$ è un sottoinsieme di $\Phi_{\mathcal{P}}^{\infty}(I)^-$.

Base. Per ogni $a \in X \cap C_1$ e per ogni $r \in \mathcal{P}$ con $a \in H(r)$, vale che (3) $H(r)$ è vero rispetto ad $I \dot{\cup} \neg.X$ (si noti che le condizioni (1) e (2) di infondatezza non possono valere perché $B(r)$ è vuoto). Da (3) segue che $H(r)$ è vero anche rispetto ad $I \dot{\cup} \neg.\{a\}$. Allora $a \in \gamma_P(I) \subseteq \Phi_{\mathcal{P}}^{\infty}(I)^-$.

Supponiamo $X \cap C_i \subseteq \Phi_{\mathcal{P}}^{\infty}(I)^-$.

Per ogni $a \in X \cap C_{i+1}$ e per ogni $r \in \mathcal{P}$ con $a \in H(r)$, vale che (1) $Ant(B(r))$ è falso rispetto ad I oppure (2) $Mon(B(r))$ è falso rispetto ad $I \dot{\cup} \neg.X$ oppure (3) $H(r)$ è vero rispetto ad $I \dot{\cup} \neg.X$. Chiaramente (1) o (3) implicano (a) o (b) della Definizione 9. Se vale (2), poiché $\mathcal{P}$ è aciclico, $Mon(B(r))$ dipende solo dagli atomi in

$$C_1 \cup C_2 \cup ... \cup C_i$$

e quindi è falso anche rispetto ad

$$I \dot{\cup} \neg.(X \cap \bigcup_{j=1}^{i} C_j)\,.$$

Poiché $X \cap C_j \subseteq \Phi_{\mathcal{P}}^{\infty}(I)^-$ vale per $j = 1, ..., i$, varrà anche

$$X \cap \bigcup_{j=1}^{i} C_j \subseteq \Phi_{\mathcal{P}}^{\infty}(I)^-\,.$$

Come conseguenza $I \dot{\cup} \neg.(X \cap \bigcup_{j=1}^{i} C_j) \subseteq \Phi_{\mathcal{P}}^{\infty}(I)$. Quindi $Mon(B(r))$ è falso

anche rispetto a $\Phi^{\infty}_{\mathcal{P}}(I)$ e $a \in \Phi^{\infty}_{\mathcal{P}}(I)$. □

Da questi due lemmi è possibile concludere che per programmi aciclici l'applicazione dell'uno o dell'altro operatore produce esattamente lo stesso risultato.

Teorema 6.6 *Dato un programma* DLP$^{\mathcal{A}}$ *aciclico* $\mathcal{P}$ *(ovvero* $\mathcal{P}$ *è non-ricorsivo, anche rispetto agli aggregati) e un'interpretazione* I*, vale che*

$$\Phi^{\infty}_{\mathcal{P}}(I) = \mathcal{W}^{\infty}_{\mathcal{P}}(I)\,.$$

Dimostrazione. In due passi

1. $\Phi^{i}_{\mathcal{P}}(I) \subseteq \mathcal{W}^{\infty}_{\mathcal{P}}(I)$, per ogni i.
2. $\mathcal{W}^{i}_{\mathcal{P}}(I) \subseteq \Phi^{\infty}_{\mathcal{P}}(I)$, per ogni i.

1. Base. $\Phi_{\mathcal{P}}(I) \subseteq \mathcal{W}_{\mathcal{P}}(I) \subseteq \mathcal{W}^{\infty}_{\mathcal{P}}(I)$ dal Lemma 6.4 e dalla definizione di $\mathcal{W}^{\infty}_{\mathcal{P}}(I)$.

   Si supponga che $\Phi^{i}_{\mathcal{P}}(I) \subseteq \mathcal{W}^{\infty}_{\mathcal{P}}(I)$.

   Allora
   $$\Phi^{i+1}_{\mathcal{P}}(I) = \Phi_{\mathcal{P}}(\Phi^{i}_{\mathcal{P}}(I)) \subseteq \mathcal{W}_{\mathcal{P}}(\mathcal{W}^{i}_{\mathcal{P}}(I)) \subseteq \mathcal{W}^{\infty}_{\mathcal{P}}(I)$$

   dal Lemma 6.4 e dalla definizione di $\mathcal{W}^{\infty}_{\mathcal{P}}(I)$.

2. Base. $\mathcal{W}_{\mathcal{P}}(I) \subseteq \Phi^{\infty}_{\mathcal{P}}(I)$ dal Lemma 6.5.

   Supponiamo che $\mathcal{W}^{i}_{\mathcal{P}}(I) \subseteq \Phi^{\infty}_{\mathcal{P}}(I)$.

   Allora
   $$\mathcal{W}^{i+1}_{\mathcal{P}}(I) = \mathcal{W}_{\mathcal{P}}(\mathcal{W}^{i}_{\mathcal{P}}(I)) \subseteq \Phi^{\infty}_{\mathcal{P}}(\mathcal{W}^{i}_{\mathcal{P}}(I))$$

   dal Lemma 6.5, e
   $$\Phi^{\infty}_{\mathcal{P}}(\mathcal{W}^{i}_{\mathcal{P}}(I)) \subseteq \Phi^{\infty}_{\mathcal{P}}(\Phi^{\infty}_{\mathcal{P}}(I))$$

   dall'ipotesi di induzione. Chiaramente

   $$\Phi^{\infty}_{\mathcal{P}}(\Phi^{\infty}_{\mathcal{P}}(I)) = \Phi^{\infty}_{\mathcal{P}}(I)$$

dalla definizione di $\Phi_{\mathcal{P}}^{\infty}(I)$.

□

Esempio 6.7 *Il programma $\mathcal{P}_4$ dell'Esempio 2.13 è aciclico, mentre il programma $\mathcal{P}_5$ dello stesso esempio non lo è.*

Come si può vedere dall'Esempio 6.2 e dall'Esempio 6.3, l'applicazione dell'operatore $\Phi_{\mathcal{P}}^{\infty}$ e dell'operatore $\mathcal{W}_{\mathcal{P}}^{\infty}$ per il programma $\mathcal{P}_4$ produce lo stesso risultato, mentre per il programma $\mathcal{P}_5$ vengono restituiti insiemi differenti.

Capitolo 7

Computazione del GUS

Nei capitoli precedenti abbiamo mostrato le proprietà degli insiemi infondati e come questi possano essere utilizzati durante la computazione dei modelli stabili. In questo capitolo, invece, illustriamo delle tecniche per la determinazione del GUS e mostriamo come questo processo possa essere modularizzato, localizzando il calcolo sulle singole componenti del programma logico.

7.1 L'operatore $\mathcal{R}^{\omega}_{\mathcal{P},I}$ per il calcolo del GUS

In questa sezione definiamo un operatore per la computazione del greatest unfounded set di un programma $DLP^{\mathcal{A}}_{m,a}$ $\mathcal{P}$ rispetto ad un'interpretazione I: l'operatore $\mathcal{R}_{\mathcal{P},I}$ che, dato un insieme X di atomi ground, scarta gli elementi in X che non soddisfano nessuna delle condizioni di infondatezza della Definizione 5.

Definizione 12 *Sia $\mathcal{P}$ un programma $DLP^{\mathcal{A}}_{m,a}$ e I un'interpretazione.*

Definiamo l'operatore $\mathcal{R}_{\mathcal{P},I}$ come il mapping $2^{B_{\mathcal{P}}} \rightarrow 2^{B_{\mathcal{P}}}$ seguente:

$$\begin{array}{rcll} \mathcal{R}_{\mathcal{P},I}(X) & = & \{a \in X \mid & \forall r \in ground(\mathcal{P}) \; with \; a \in H(r) \\ & & & Ant(B(r)) \textit{ is false w.r.t. } I, \\ & & \textit{or} & Mon(B(r)) \textit{ is false w.r.t. } I \,\dot{\cup}\, \neg.X, \\ & & \textit{or} & H(r) \textit{ is true w.r.t. } I \,\dot{\cup}\, \neg.\{a\}\} \end{array}$$

Dato un insieme $X \subseteq B_{\mathcal{P}}$, la sequenza

$$\begin{array}{rcll} R_0 & = & X & \\ R_n & = & \mathcal{R}_{\mathcal{P},I}(R_{n-1}) & \quad n > 0 \end{array}$$

decresce monotonicamente e converge in modo finito ad un limite che denotiamo con $\mathcal{R}^{\omega}_{\mathcal{P},I}(X)$.

Nel seguito mostriamo che $\mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I)$ è un insieme infondato. Quindi è possibile utilizzare questo operatore per identificare atomi infondati, riducendo lo spazio di ricerca.

Proposizione 12 *Sia $\mathcal{P}$ un programma $DLP^{\mathcal{A}}_{m,a}$ e I un'interpretazione. Allora, $\mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I)$ è un insieme infondato per $\mathcal{P}$ rispetto ad I.*

Dimostrazione. Sia $X = \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I)$.

Dalla definizione di $\mathcal{R}_{\mathcal{P},I}$, abbiamo che $X \subseteq B_{\mathcal{P}} \setminus I$, e quindi $X \cap I = \emptyset$. Per tutti gli $a \in X$ e per tutte le regole $r \in \mathcal{P}$, $a \in H(r)$ implica che $Ant(B(r))$ è falso rispetto ad I oppure $Mon(B(r))$ è falso rispetto ad $I \,\dot{\cup}\, \neg.X$ oppure $H(r)$ è vero rispetto ad $I \,\dot{\cup}\, \neg.\{a\}$. Se vale l'ultimo, poiché $X \cap I = \emptyset$, $H(r)$ è vero anche rispetto ad $I \,\dot{\cup}\, \neg.X$.

Segue che X è un insieme infondato per $\mathcal{P}$ rispetto ad I. □

Il Lemma 6.4 in [11] mostra un'importante proprietà di $\mathcal{R}_{\mathcal{P},I}$ nel caso privo di aggregati: esso non scarta nessun atomo appartenente ad un insieme infondato contenuto nell'insieme di partenza. È possibile estendere questo risultato anche ai programmi con aggregati.

Proposizione 13 *Sia $\mathcal{P}$ un programma $DLP^{\mathcal{A}}_{m,a}$, I un'interpretazione per $\mathcal{P}$ e $J \subseteq B_{\mathcal{P}}$ un insieme di atomi.*

Ogni insieme infondato per $\mathcal{P}$ rispetto ad I che è contenuto in J è anche contenuto in $\mathcal{R}^{\omega}_{\mathcal{P},I}(J)$.

Dimostrazione. Sia $X \subseteq J$ un insieme infondato per $\mathcal{P}$ rispetto ad I.

Per ogni $a \in X$ e per ogni regola $r \in \mathcal{P}$ tale che $a \in H(r)$, $Ant(B(r))$ è falso rispetto ad I oppure $Mon(B(r))$ è falso rispetto ad $I \dot{\cup} \neg.X$ oppure $H(r)$ è vero rispetto ad $I \dot{\cup} \neg.X$. Se vale l'ultimo, poiché $\{a\} \subseteq X$, $H(r)$ è vero anche rispetto ad $I \dot{\cup} \neg.\{a\}$. Dalla definizione di $\mathcal{R}_{\mathcal{P},I}$ segue che $\mathcal{R}_{\mathcal{P},I}(X) = X$ e, poiché $X \subseteq J$, $\mathcal{R}^{\omega}_{\mathcal{P},I}(J) \supseteq X$. □

Utilizzando le precedenti proposizioni possiamo dimostrare che $\mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I)$ computa il greatest unfounded set per $\mathcal{P}$ rispetto ad I.

Teorema 7.1 *Sia $\mathcal{P}$ un programma $DLP^{\mathcal{A}}_{m,a}$ e I un'interpretazione unfounded-free. Allora*

$$\mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I) = GUS_{\mathcal{P}}(I)\,.$$

Dimostrazione. ($\supseteq$) Poiché I è unfounded-free, $I \cap X = \emptyset$ vale per ogni insieme infondato per $\mathcal{P}$ rispetto ad I, e quindi $X \subseteq B_{\mathcal{P}} \setminus I$. Perciò, dalla Proposizione 13, vale anche che $X \subseteq \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I)$, e quindi $GUS_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I)$ è contenuto in $\mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I)$.

($\subseteq$) Dalla Proposizione 12 $\mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I)$ è un insieme infondato per $\mathcal{P}$ rispetto ad I. Quindi, dalla definizione di GUS, segue che $\mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I)$ è contenuto in $GUS_{\mathcal{P}}(I)$. □

Il punto fisso dell'operatore $\mathcal{R}_{\mathcal{P},I}$ è efficientemente computabile. Perciò, dal teorema precedente, è possibile utilizzare $\mathcal{R}_{\mathcal{P},I}$ come un potente ed efficiente operatore di pruning per interpretazioni unfounded-free.

Nella classe dei programmi head-cycle-free [29] l'operatore $\mathcal{R}_{\mathcal{P},I}$ consente di computare sempre il greatest unfounded set, anche se l'interpretazione non è unfounded-free. Può quindi essere utilizzato sia per il pruning dello spazio di

ricerca che per la verifica di stabilità dei modelli. Nella prossima sezione mostriamo come questo può essere fatto in modo efficiente, fornendo un algoritmo per la computazione modulare del GUS tramite l'operatore $\mathcal{R}_{\mathcal{P},I}$.

7.2 Valutazione modulare del GUS

Prima di introdurre l'algoritmo è bene mostrare qualche proprietà dell'operatore $\mathcal{R}_{\mathcal{P},I}$.

Lemma 7.2 *Sia $\mathcal{P}$ un programma e I un'interpretazione.*

Per ogni coppia di insiemi di atomi X e Y, tali che $X \subseteq Y$, vale che

$$\mathcal{R}^{\omega}_{\mathcal{P},I}(X) \subseteq \mathcal{R}^{\omega}_{\mathcal{P},I}(Y)\,.$$

Dimostrazione. Per induzione.
La sola condizione della Definizione 12 che dipende dall'insieme iniziale X è "$Mon(B(r))$ è falso rispetto ad $I \dot{\cup} \neg.X$". Se questo vale per quache atomo in $\mathcal{R}_{\mathcal{P},I}(X)$ e qualche regola regola r in $\mathcal{P}$, allora $Mon(B(r))$ è falso anche rispetto ad $I \dot{\cup} \neg.Y$, perché $I \dot{\cup} \neg.Y \leq I \dot{\cup} \neg.X$. Segue che $\mathcal{R}_{\mathcal{P},I}(X) \subseteq \mathcal{R}_{\mathcal{P},I}(Y)$.
Assumiamo $\mathcal{R}^{(i)}_{\mathcal{P},I}(X) \subseteq \mathcal{R}^{(i)}_{\mathcal{P},I}(Y)$.
Allora $\mathcal{R}^{(i+1)}_{\mathcal{P},I}(X) = \mathcal{R}_{\mathcal{P},I}(\mathcal{R}^{(i)}_{\mathcal{P},I}(X)) \subseteq \mathcal{R}_{\mathcal{P},I}(\mathcal{R}^{(i)}_{\mathcal{P},I}(Y)) = \mathcal{R}^{(i+1)}_{\mathcal{P},I}(Y)$. □

Lemma 7.3 *Sia $\mathcal{P}$ un programma e I un'interpretazione, C una componente di $\mathcal{P}$ e $\mathcal{P}_C$ il sottoprogramma associato a C. Allora, per ogni $X \subseteq C$*

$$\mathcal{R}_{\mathcal{P}_C,I}(X) = \mathcal{R}_{\mathcal{P},I}(X)\,.$$

Dimostrazione. Chiaramente ogni regola r di $\mathcal{P}$ con $H(r) \cap X \neq \emptyset$ è anche in $\mathcal{P}_C$ (e soddisfa almeno una delle condizioni della Definizione 12). □

Teorema 7.4 *Sia $C_1, ..., C_n$ un ordine totale (fra i possibili) per le componenti di un programma HCF $\mathcal{P}$ tale che $C_i \preceq C_j$ implica $i \leq j$.*
A partire da

$I_0 := I$

e quindi, per ogni $i = 1, ..., n$, *computando*

$X_i := \mathcal{R}^{\omega}_{\mathcal{P}_{C_i}, I_{i-1}}(C_i \setminus I)$

$I_i := I_{i-1} \cup \neg.X_i$

vale che I_n *è uguale a* $I \cup \neg.GUS_{\mathcal{P}}(I)$.

Dimostrazione. Dimostriamo che ad ogni passo della computazione vale che $X_i = \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I) \cap C_i$.

Base ($\subseteq$). Dal Lemma 7.3 e dal Lemma 7.2 segue che

$$X_1 = \mathcal{R}^{\omega}_{\mathcal{P}_{C_1}, I}(C_1 \setminus I) = \mathcal{R}^{\omega}_{\mathcal{P},I}(C_1 \setminus I) \subseteq \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I)\,.$$

Perciò,

$$X_1 \subseteq \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I) \cap C_1$$

perché $X_1 \subseteq C_1$.

Base ($\supseteq$). Per ogni $a \in \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I) \cap C_1$ e per ogni regola $r \in P$ con $a \in H(r)$

(1) $Ant(B(r))$ è falso rispetto ad I oppure

(2) $Mon(B(r))$ è falso rispetto ad $I \dot{\cup} \neg.\mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I)$ oppure

(3) $H(r)$ è vero rispetto ad $I \dot{\cup} \neg.\{a\}$.

Si noti che, poiché $\mathcal{P}$ è HCF, a è il solo atomo in $H(r)$ appartenente a C_1; perciò, da (3) segue che $H(r)$ è vero rispetto a $I \dot{\cup} \neg.(\mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I) \cap C_1)$. Inoltre, si noti che $Mon(B(r))$ dipende solo dagli atomi in C_1, quindi da (2) segue che $Mon(B(r))$ è falso rispetto a $\mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I) \cap C_1$. Perciò $\mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I) \cap C_1$ è un unfounded set per $\mathcal{P}_{C_1}$ rispetto ad I e, dalla Definizione 12, è un sottoinsieme di X_1.

Supponiamo che $X_i = \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I) \cap C_i$.

($\subseteq$) Per ogni $a \in X_{i+1}$ e per ogni $r \in \mathcal{P}_{C_{i+1}}$ con $a \in H(r)$

- $Ant(B(r))$ è falso rispetto ad I_i (e quindi rispetto ad I perché $I_i \leq I$) oppure
- $Mon(B(r))$ è falso rispetto ad $I_i \dot{\cup} \neg.X_{i+1}$ $(= I \dot{\cup} \neg.(X_{i+1} \cup (I_i^- \setminus I^-)))$ oppure
- $H(r)$ è vero rispetto ad $I_i \dot{\cup} \neg.\{a\}$ (e quindi anche rispetto ad $I \dot{\cup} \neg.(X_{i+1} \cup (I_i^- \setminus I^-))$ perché $I_i^+ = I^+$ e a è il solo atomo di $H(r)$ appartenente a C_j, for $j = 1, ..., i+1$).

Nessun'altra regola in $\mathcal{P} \setminus \mathcal{P}_{C_{i+1}}$ ha a in testa, quindi $X_{i+1} \cup (I_i^- \setminus I^-)$ è un unfounded set per $\mathcal{P}$ rispetto ad I. Perciò dalla Proposizione 13, X_{i+1} è un sottoinsieme di $\mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I)$.

($\supseteq$) Per ogni $a \in \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I) \cap C_{i+1}$ e per ogni $r \in \mathcal{P}$ con $a \in H(r)$

(1) $Ant(B(r))$ è falso rispetto ad I (e quindi anche rispetto ad I_i perché $I_i \supseteq I$) oppure

(2) $Mon(B(r))$ è falso rispetto ad $I \dot{\cup} \neg.\mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I)$ oppure

(3) $H(r)$ è vero rispetto ad I (e perciò rispetto ad I_i).

Da (2) segue che $Mon(B(r))$ è falso rispetto a

$$Y = I \dot{\cup} \neg.(\mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I) \cap (C_1 \cup C_2 \cup ... \cup C_{i+1})$$

perché $Mon(B(r))$ dipende solo dagli atomi in $C_1, \ldots, C_{i+1}$. Ma

$$I \dot{\cup} \neg.(\mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I) \cap (C_1 \cup C_2 \cup ... \cup C_i) = I_i$$

e

$$X_{i+1} \subseteq \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}} \setminus I) \cap C_{i+1}$$

dalle quali segue che

$$Y^+ = (I_i \dot{\cup} \neg.X_{i+1})^+$$

e

$$Y^- \supseteq (I_i \dot{\cup} \neg.X_{i+1})^- \,.$$

Perciò $Y \leq I_i \dot{\cup} \neg.X_{i+1}$ e $Mon(B(r))$ è falso rispetto ad $I_i \dot{\cup} \neg.X_{i+1}$. □

Quindi, per computare il greatest unfounded set possiamo agire sulle singole componenti e sui relativi sottoprogrammi associati, senza considerare il programma nella sua interezza. I risultati precedenti garantiscono la correttezza di questo approccio.

Parte III

Implementazione e sperimentazione

I risultati teorici ottenuti sono stati implementati in DLV, realizzando un prototipo la cui architettura è descritta nel capitolo 8.

Il prototipo implementato è stato testato sia con programmi logici con aggregati ricorsivi che con programmi logici con aggregati stratificati. Il primo gruppo di test è stato eseguito per valutare l'efficienza delle tecniche sviluppate rispetto a quelle già esistenti implementate in altri solver. Il secondo gruppo di test ha lo scopo di misurare l'overhead introdotto dalle modifiche applicate a DLV per poter supportare la programmazione logica con aggregati ricorsivi. I risultati ottenuti, insieme alla descrizione dei problemi utilizzati, sono riportati nel capitolo 9.

Capitolo 8

Architettura del prototipo

Per poter sperimentare i risultati teorici ottenuti abbiamo esteso il sistema DLV, che già processava gli aggregati non ricorsivi, con la tecnica di istanziazione presentata nel capitolo 5 e con l'algoritmo per la computazione del greatest unfounded set descritto nel capitolo 7. Il prototipo ottenuto è un sistema in grado di computare i modelli stabili di programmi logici con aggregati ricorsivi.

La struttura principale del sistema è riportata in Figura 8.1, mentre per una descrizione dettagliata dell'architettura di DLV è possibile fare riferimento a [30].

L'input, dopo essere stato trattato da qualche eventuale frontend, viene processato dal core di DLV. In particolare il modulo di grounding produce una versione ground del programma in input, per la quale è garantito che i modelli stabili coincidano.

Il controllo passa quindi al *model generator*, il quale esegue un algoritmo di backjumping [31] che sfrutta un'euristica allo scopo di individuare possibili modelli stabili. Durante questa ricerca sono applicate diverse tecniche di pruning dello spazio di ricerca, fra le quali anche la computazione degli insiemi infondati (si vedano, per esempio, [32, 33]).

Ognuno dei modelli stabili candidati individuati viene quindi sottomesso al model checker, che ne verifica la stabilità (si veda, per esempio, [13]). Quan-

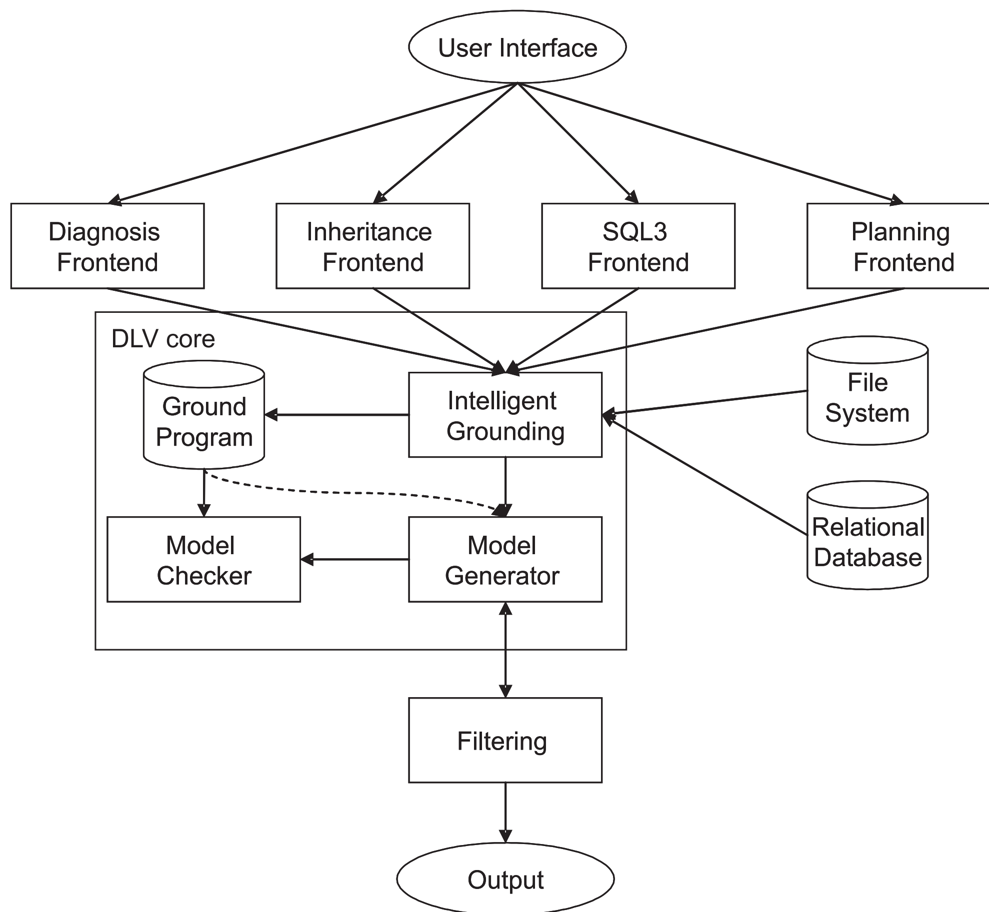


Figura 8.1: *Architettura del sistema DLV*

do l'input è un programma HCF questo controllo non è necessario poiché ogni candidato individuato è noto essere un modello stabile.

Pertanto, per il nostro prototipo abbiamo dovuto modificare i moduli del grounding e del model generator. Nella fase di grounding, in caso di aggregati non-ricorsivi, tutte le istanze dei predicati appartenenti all'insieme di aggregazione sono note nel momento in cui l'aggregato viene istanziato. In presenza di aggregati ricorsivi questa assunzione non è più valida. Abbiamo quindi dovuto sviluppare una strategia più complessa di grounding, descritta nel capitolo 5. Per quanto riguarda il model generator una larga parte dell'implementazione esistente per gli aggregati stratificati può essere riutilizzata. Per poter trattare gli aggregati ricorsivi correttamente è necessaria la computazione degli insiemi infondati descritta nel capitolo 7, che non era ancora presente in DLV. Abbiamo quindi implementato la computazione degli insiemi infondati utilizzando una versione ottimizzata del metodo descritto nella sezione 7.2, che localizza maggiormente la computazione concentrandosi solo sulle componenti che sono state coinvolte nell'ultimo passo di propagazione. L'algoritmo utilizzato è riportato in appendice A.

Il prototipo del sistema è disponibile in rete all'indirizzo

`http://www.dlvsystem.com/dlvRecAggr`

e supporta un sovrainsieme dei programmi HCF, richiedendo l'assenza di cicli in testa solo nelle componenti con aggregati ricorsivi.

Capitolo 9

Risultati sperimentali

Abbiamo eseguito una serie di benchmark con lo scopo di valutare l'efficienza del sistema realizzato con la presenza o l'assenza dei diversi aspetti computazionali presentati nei capitoli precedenti. I test eseguiti possono essere suddivisi in due suite. Nella prima sono presenti istanze di taglia crescente di *Company Controls*, problema che abbiamo già presentato nei capitoli precedenti. Con questa suite vogliamo misurare la scalabilità delle tecniche sviluppate al crescere delle dimensioni dei dati in input. La seconda suite di benchmark prende in considerazione esclusivamente programmi privi di aggregati ricorsivi tratti da *Asparagus* [1], l'ultima competition $\mathcal{ASP}$. Lo scopo è di verificare se l'introduzione di queste nuove strategie comporta un degrado delle prestazioni del sistema DLV nel processamento di programmi logici standard.

Nel seguito descriviamo brevemente i sistemi messi a confronto, i problemi utilizzati per il benchmark e i risultati ottenuti.

9.1 Sistemi confrontati

I sistemi che abbiamo confrontato sperimentalmente sono riportati in Tabella 9.1.

[1] http://asparagus.cs.uni-potsdam.de/contest/index.php

DLV	Il sistema DLV senza alcuna modifica.
$\mathrm{DLV}^{\mathcal{A}}$	Il sistema DLV con il supporto agli aggregati ricorsivi, la cui l'istanziazione viene posticipata al termine del grounding della componente a cui appartiene.
$\mathrm{DLV}^{\mathcal{A}}_{\mathcal{I}}$	Il sistema DLV con il supporto agli aggregati ricorsivi e la tecnica di istanziazione presentata nel capitolo 5.
$\mathrm{DLV}^{\mathcal{A}}_{\mathcal{I}+\mathcal{A}}$	Il sistema DLV con il supporto agli aggregati ricorsivi e la tecnica di istanziazione presentata nel capitolo 5, con l'aggiunta dell'atomo ausiliario al corpo della regola.
SMODELS	Il sistema *Smodels* + *lparse*.
CMODELS	Il sistema *Cmodels* + *lparse*.
CLASP	Il sistema *clasp* + *lparse*.

Tabella 9.1: *Sistemi confrontati*

Il confronto più interessante è sicuramente quello di DLV con la presenza o l'assenza delle strategie presentate nei capitoli 5 e 7. In particolare abbiamo confrontato tre estensioni differenti del sistema DLV, che fanno tutte uso della definizione di insieme infondato presentato nel capitolo 4. La prima di queste estensioni, cui faremo riferimento come $\mathrm{DLV}^{\mathcal{A}}$, esegue l'istanziazione degli aggregati ricorsivi al termine della componente cui l'aggregato appartiene. La seconda, che chiameremo $\mathrm{DLV}^{\mathcal{A}}_{\mathcal{I}}$, implementa la strategia per l'istanziazione anticipata degli aggregati ricorsivi monotoni descritta nel capitolo 5. La terza estensione, $\mathrm{DLV}^{\mathcal{A}}_{\mathcal{I}+\mathcal{A}}$, oltre all'istanziazione anticipata dei monotoni, implementa l'*ottimizzazione per aggregati falsi su insieme vuoto* descritta nel capitolo 5.

Di rilevante interesse è anche il confronto dei sistemi da noi sviluppati con altri solver che risolvono il problema Company Controls, quali *Smodels* [2], *Cmodels* [3] e *clasp* [4]. Questi sistemi hanno bisogno di un tool esterno per l'istanziazione del programma logico. Fra i software adatti a questo scopo abbiamo scelto *lparse* [5],

[2] `http://www.tcs.hut.fi/Software/smodels/`
[3] `http://www.cs.utexas.edu/users/tag/cmodels.html`
[4] `http://www.cs.uni-potsdam.de/clasp/`
[5] `http://www.tcs.hut.fi/Software/smodels/lparse`

che è quello più comunemente utilizzato.

9.2 Problemi di benchmark

In questa sezione presentiamo i problemi utilizzati per la sperimentazione. La suite Company Controls ha lo scopo di misurare le prestazioni delle strategie sviluppate, mentre la suite Asparagus può essere vista come un test di regressione per le performance, ovvero per la valutazione del costo aggiunto al sistema.

Suite Company Controls

Company Controls costituisce l'esempio di maggiore rilevanza, in quanto con questo possiamo testare i diversi aspetti teorici e computazionali sviluppati durante la tesi.

Per questa suite abbiamo generato numerose istanze di taglia crescente, in modo da valutare la scalabilità dei sistemi realizzati. Per ogni taglia abbiamo generato 5 istanze diverse, in modo da avere una misura più accurata dei tempi di risposta dei sistemi.

La codifica utilizzata è riportata in Figura 9.1 e fa uso di aggregati ricorsivi monotoni.

```
cv(X,X,Y,S) :- owns(X,Y,S).
cv(X,Z,Y,S) :- controls(X,Z),owns(Z,Y,S).

controls(X,Y) :-
    50 < #sum{S,Z : cv(X,Z,Y,S) }, company(X),company(Y).
```

Figura 9.1: *Codifica di Company Controls*

Suite Asparagus

Dall'ultima competizione Asparagus abbiamo tratto una serie di test per valutare il costo aggiunto a DLV in programmi logici privi di aggregati ricorsivi.

Di seguito riportiamo una breve descrizione dei problemi che costituiscono la suite, mentre rimandiamo al sito ufficiale della competizione per una descrizione dettagliata [6].

Bounded Spanning Tree (BST)

Sia G un grafo diretto. Uno spanning tree (albero ricoprente) è d-bounded se per ogni vertice il numero di archi uscenti è al più d.

Nel problema *bounded spanning tree* sono dati un grafo diretto $G = (V, E)$, dove V è l'insieme di vertici ed E l'insieme di archi, e un bound intero d.

Lo scopo è di trovare un d-bounded spanning tree in G.

La codifica utilizzata è riportata in appendice B.

Hamiltonian Cycle (HAMILT-CYCLE)

Sia G un grafo non orientato. Un ciclo Hamiltoniano è un ciclo in G che visita tutti i vertici di G esattamente una volta.

Nel problema Hamiltonian cycle viene dato un grafo non orientato $G = (V, E)$, dove V è l'insieme dei vertici di G ed E è l'insieme degli archi.

Lo scopo è trovare un ciclo Hamiltoniano in G.

La codifica utilizzata è riportata in appendice B.

Weighted Spanning Tree (WST)

Sia G un grafo diretto. Ad ogni arco del grafo è assegnato un peso. Uno spanning tree (albero ricoprente) è w*-bounded* se per ogni vertice la somma dei pesi degli archi uscenti è al più w.

Nel problema *weighted spanning tree* è dato un grafo diretto $G = (V, E, W)$, dove V è l'insieme di vertici, E è l'insieme di archi e W è una funzione che mappa ogni arco nel grafo verso un intero nel range $1, \ldots, |V|$ che rappresenta il peso associato all'arco. Viene inoltre dato un bound intero w.

[6] `http://asparagus.cs.uni-potsdam.de/contest/index.php`

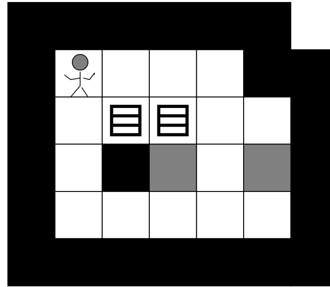


Figura 9.2: *Un'istanza di Sokoban*

Lo scopo è trovare un w-bounded spanning tree in G.

La codifica utilizzata è riportata in appendice B.

Sokoban (SOKOBAN)

Sokoban è un puzzle sviluppato dalla compagnia Giapponese *Thinking Rabbit, Inc.* nel 1982. "Sokoban" in giapponese significa "magazziniere".

Ogni puzzle consiste di una stanza (un numero di quadrati che rappresentano muri o parti di pavimento, alcuni dei quali sono marcati come aree di deposito) e una situazione iniziale (un sokoban ed un certo numero di casse, ciascuna delle quali risiede su qualche area del pavimento). L'obiettivo del gioco è muovere tutte le scatole sulle aree di deposito. Per raggiungere questo scopo, il sokoban può muoversi lungo le aree libere del pavimento e spingere singole casse da un'area del pavimento ad una contigua libera.

La Figura 9.2 mostra una tipica configurazione che comprende due casse, in cui le aree grigie sono quelle di deposito e quelle nere sono tratti di muri.

La codifica utilizzata è riportata in appendice B.

Fastfood (FASTFOOD)

La catena di fastfood McBurger possiede diversi ristoranti lungo un'autostrada. Hanno recentemente deciso di costruire diversi depositi. Ogni deposito deve essere collocato presso un ristorante e rifornirne diversi. Ogni ristorante dovrà essere

rifornito dal deposito più vicino. Se due o più depositi sono equidistanti da un ristorante, questo sarà rifornito esattamente da uno dei due depositi (non importa quale). I depositi devono essere posizionati in modo da minimizzare la somma delle distanze da ogni ristorante al suo deposito più vicino.

Dato uno schema di costruzione dei depositi, lo scopo è verificare se questo è ottimo o, se non lo è, trovare uno schema migliore.

La codifica utilizzata è riportata in appendice B.

Traveling Salesperson Suite (TSS)

Sia G un grafo diretto ai cui vertici è associato un costo. L'obiettivo del commesso viaggiatore è di trovare un ciclo Hamiltoniano con il minor costo possibile. Siamo interessati alla versione decisionale del problema.

> Dato un grafo diretto G e un intero k, esiste un ciclo Hamiltoniano di costo non superiore a k?

La codifica utilizzata è riportata in appendice B.

Social Golfer (SOC-GOLF)

Il problema Social Golfer è stato già presentato nella sezione 2.6. La codifica utilizzata è quella in Figura 2.4.

Car Sequencing (CAR-SEQ)

Abbiamo già presentato il problema Car Sequencing nella sezione 2.6. La codifica utilizzata è quella in Figura 2.3.

Towers Of Hanoi Competition (HANOI)

Il problema della *Torre di Hanoi* classico ha tre pali e n dischi. Inizialmente, tutti i dischi sono sul palo a sinistra. Lo scopo è di muovere tutti i dischi sul palo di destra con l'aiuto del palo centrale. Le regole sono:

1. spostare un disco per volta,
2. spostare solo il disco più in alto in un certo palo,
3. non poggiare un disco più grande su un disco più piccolo.

La codifica utilizzata è riportata in appendice B.

9.3 Risultati

Gli esperimenti sono stati condotti su un computer dedicato con

- processore Intel dual Xeon a 3 GHz e con 2 MB di memoria cache,
- 3 GB di memoria RAM,
- sistema operativo Debian GNU/Linux 4.0 (etch),
- kernel Linux 2.6.18,
- compilatore GCC 4.1.2.

Ogni test è stato eseguito per 3 volte, consentendo l'allocazione di non più di 1 GB di memoria centrale e un tempo massimo di esecuzione pari a 300 secondi.

Suite Company Controls

Dai risultati ottenuti per la suite Company Controls abbiamo realizzato i grafici riportati in Figura 9.3 e in Figura 9.4, nei quali DLV [1] è $\text{DLV}^{\mathcal{A}}$, DLV [2] è $\text{DLV}^{\mathcal{A}}_{\mathcal{I}}$ e DLV [3] è $\text{DLV}^{\mathcal{A}}_{\mathcal{I}+\mathcal{A}}$. Le ascisse rappresentano il numero di compagnie presenti nell'istanza, ovvero la taglia. Le ordinate, in scala logaritmica, corrispondono ai tempi medi di esecuzione per la Figura 9.3 e ai tempi massimi di esecuzione per la Figura 9.4. I grafici in alto mostrano una visione d'insieme dei risultati della

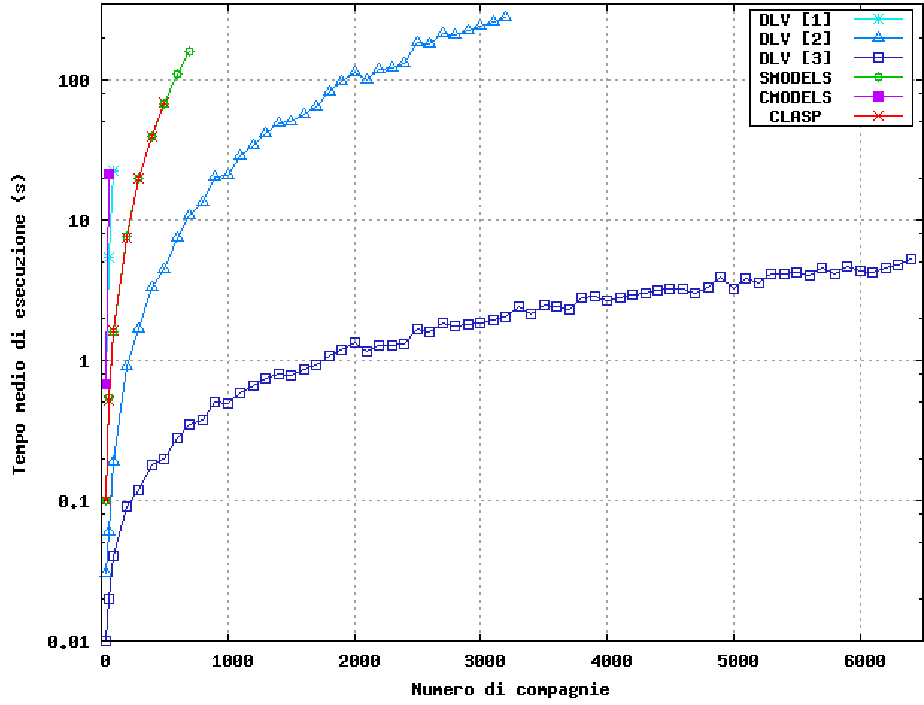
sperimentazione con i diversi sistemi. In basso, invece, abbiamo riportato gli stessi grafici limitati alle istanze fino a 1000 compagnie. Le tabelle dalle quali sono stati estratti questi grafici sono riportate in Appendice B.

Come si può notare, i sistemi *CMODELS*, DLV$^{\mathcal{A}}$, *CLASP* ed *SMODELS* riescono a terminare, nel tempo e/o nello spazio previsti, solo con le istanze più piccole. Questi solver necessitano di una grande quantità di memoria per eseguire l'istanziazione del programma. I sistemi DLV$^{\mathcal{A}}_{\mathcal{I}}$ e DLV$^{\mathcal{A}}_{\mathcal{I}+\mathcal{A}}$, che implementano la nostra strategia di istanziazione per i monotoni, mostrano invece un'ottima gestione delle risorse. Infatti, DLV$^{\mathcal{A}}_{\mathcal{I}}$ riesce a computare metà delle istanze generate, mentre DLV$^{\mathcal{A}}_{\mathcal{I}+\mathcal{A}}$, che sfrutta appieno la nostra tecnica semi-naive grazie all'aggiunta dell'atomo ausiliario al corpo della regola, è in grado di computare facilmente anche le istanze più grandi.

Suite Asparagus

I risultati della suite Asparagus sono riportati in Tabella 9.2. La prima colonna riporta i nomi identificativi dei test, mentre le successive colonne rappresentano i tempi medi di esecuzione delle diverse versioni di DLV confrontate.

I risultati mostrano che il costo aggiunto per la trattazione degli aggregati ricorsivi non ha degradato le prestazioni di DLV. Più rilevante invece il fatto che in alcuni casi le versioni sviluppate nell'ambito del lavoro di tesi sono decisamente più efficienti della versione originale. Nelle istanze testate di Hamiltonian Cycle, Fastfood e Torre di Hanoi, per esempio, abbiamo un miglioramento di circa il 10%, mentre è notevole il guadagno per Social Golfer e Car Sequencing.



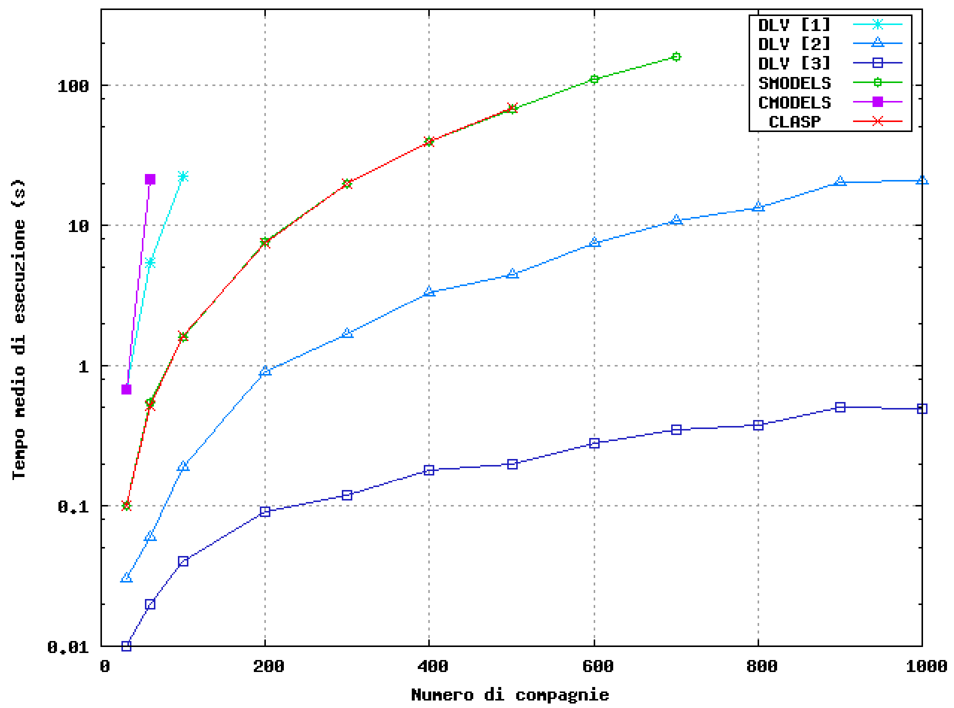


Figura 9.3: *Comapny Controls: tempo medio di esecuzione*

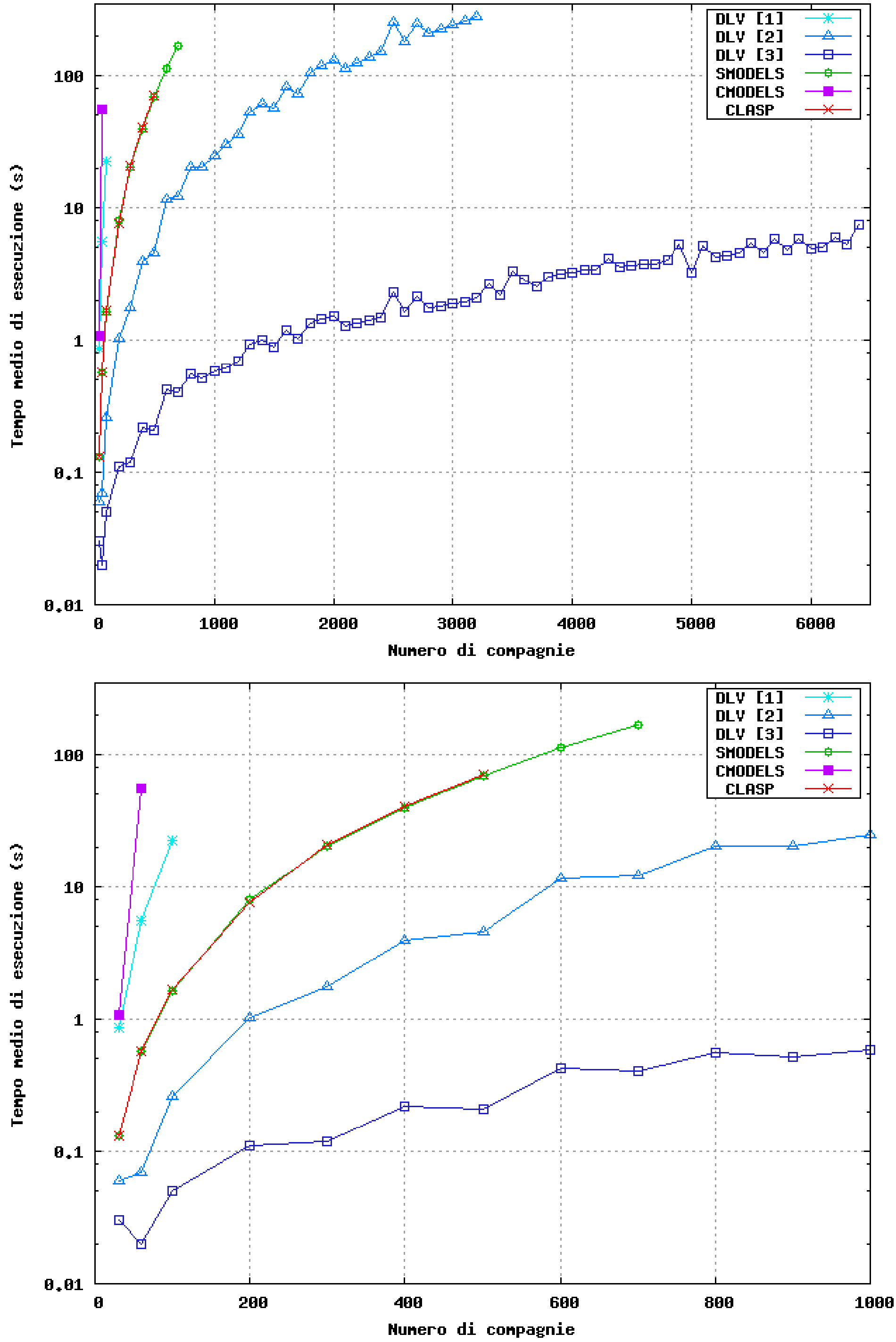


Figura 9.4: *Comapny Controls: tempo massimo di esecuzione*

	DLV	$\text{DLV}^{\mathcal{A}}$	$\text{DLV}^{\mathcal{A}}_{\mathcal{I}}$	$\text{DLV}^{\mathcal{A}}_{\mathcal{I}+\mathcal{A}}$
BST-5	0.11	0.11	0.10	0.11
BST-7	0.12	0.12	0.12	0.12
BST-8	0.12	0.12	0.11	0.11
HAMILT-CYCLE-2	2.30	2.15	2.14	2.11
HAMILT-CYCLE-3	2.32	2.21	2.17	2.16
HAMILT-CYCLE-4	2.47	2.32	2.30	2.34
WST-1	0.06	0.06	0.06	0.06
WST-2	0.03	0.03	0.02	0.03
WST-3	0.03	0.03	0.03	0.03
SOKOBAN-4	4.27	4.28	4.25	4.27
SOKOBAN-7	3.55	3.48	3.43	3.44
SOKOBAN-8	3.79	3.82	3.81	3.81
FASTFOOD-3	2.37	2.28	2.32	2.25
FASTFOOD-4	2.50	2.36	2.43	2.29
FASTFOOD-6	2.49	2.33	2.42	2.27
TSS-1	0.18	0.18	0.17	0.17
TSS-5	0.20	0.20	0.20	0.20
TSS-8	0.21	0.21	0.21	0.21
SOC-GOLF-5	13.72	0.44	1.56	0.43
SOC-GOLF-7	0.99	0.71	6.73	0.70
CAR-SEQ-6	-	0.48	0.38	0.42
HANOI-1	0.70	0.68	0.65	0.66
HANOI-2	0.70	0.67	0.66	0.66
HANOI-3	0.81	0.79	0.76	0.76

Tabella 9.2: *Risultati suite Asparagus*

Parte IV

Lavori correlati e conclusioni

Capitolo 10

Lavori Correlati

L'introduzione delle funzioni di aggregazione nei linguaggi di programmazione logica è un argomento di rilevante interesse da quando, negli anni '80, se ne avvertì la necessità nei database deduttivi quali LDL [34]. Da allora, diversi studi si sono interessati a questo argomento (si vedano, per esempio, [35, 36]). Nel seguito ci concentriamo sui lavori relativi alla programmazione logicia disgiuntiva, rimandando a [37] per una digressione storica più completa.

Diverse semantiche sono state definite per i pogrammi logici con aggregati ricorsivi (si vedano, per esempio, [38, 4, 39, 40, 37, 8, 10, 1]), tutte diffenti in qualche frammento del linguaggio (per una discussione dei pro e dei contro dei diversi approcci si faccia riferimento a [37, 41, 10]). La semantica che sembra aver avuto maggiore consenso è quella definita in [1], per la quale sono state proposte caratterizzazioni alternative in [42, 19, 27]. Per raggiungere i nostri obiettivi, abbiamo adottato la semantica di [1] e adattato alla DLP$^{\mathcal{A}}_{m,a}$ le definizioni di insieme infondato di [19, 27].

Nessuno dei lavori precedenti nell'ambito della programmazione logica con aggregati mostra come utilizzare gli insiemi infondati per la computazione dei modelli stabili, similmente a come fatto in [43] e [33] per la programmazione logica standard. In particolare, [43] definisce un operatore per la computazione del greatest unfounded set e [33] un algoritmo in grado di computarne il punto fisso;

abbiamo generalizzato questi risultati alla $\mathrm{DLP}^{\mathcal{A}}_{m,a}$ nei capitoli 6 e 7. Non sono presenti, inoltre, studi su tecniche di istanziazione che tengano in considerazione gli aggregati ricorsivi. Le strategie precedenti (si vedano, per esempio, [**?**, 28, **?**]) non sono sufficienti ad istanziare correttamente gli aggregati ricorsivi, che necessitano di un trattamento differente rispetto ai letterali standard o agli aggregati stratificati, come mostrato nel capitolo 5.

La prima implementazione di funzioni di aggregazione in answer set programming [5] è apparsa nel linguaggio di *lparse* [44], che è un frontend per l'istanziazione utilizzato dai sistemi *Smodels* [5], *Cmodels* [45] e *clasp* [20]. Questo linguaggio consente l'uso di aggregazioni ricorsive in accordo con la semantica di [40], per la quale sono state mosse diverse critiche in merito ai risultati non sempre intuitivi [46]. L'approccio di [40] è definito per programmi non-disgiuntivi con un tipo particolare di aggregati chiamati cardinality e weight constraint, che corrispondono alle funzioni *count* e *sum*. Anche il sistema DLV implementa funzioni di aggregazione [6], fra le quali *times*, *min* e *max* che non sono presenti in *lparse*. Prima di questa tesi, tuttavia, il framework DLV consentiva solo aggregazioni non-ricorsive.

Capitolo 11

Conclusioni

Le attività svolte nel corso del lavoro di tesi possono essere complessivamente suddivise in tre categorie:

- analisi delle proprietà del linguaggio,
- progettazione di algoritmi per la valutazione efficiente,
- implementazione e sperimentazione del prototipo.

Per quanto riguarda la prima parte del lavoro, abbiamo analizzato a fondo le proprietà della programmazione logica con gli aggregati ricorsivi monotoni e antimonotoni. In particolare, abbiamo mostrato alcune sottoclassi di problemi per i quali è garantita l'esistenza e l'unicità del modello stabile. Inoltre, abbiamo definito una nozione di insieme infondato che generalizza quella data in [43] e meglio si adatta alla $\text{DLP}^{\mathcal{A}}_{m,a}$ rispetto a quella più generica fornita in [27]. Attraverso questa definizione abbiamo caratterizzato i modelli stabili, mostrando come eseguire il controllo di stabilità con l'uso degli insiemi infondati.

Relativamente alla seconda categoria, abbiamo indagato su alcune strategie di istanziazione che meglio si adattano a questo linguaggio, mostrando che quelle attualmente implementate in DLV non sono sufficienti a trattare gli aggregati ricorsivi. Inoltre, abbiamo definito un operatore in grado di computare il greatest

unfounded set e abbiamo dimostrato che il punto fisso di questo operatore può essere computato modularmente, restringendo il calcolo alle singole componenti. Per quanto riguarda il taglio dello spazio di ricerca, abbiamo definito diversi operatori, fra cui l'operatore well-founded che fa uso degli insiemi infondati secondo la nostra definizione.

Abbiamo concluso estendendo DLV con le tecniche sviluppate nel corso del lavoro di tesi. Il sistema risultante è in grado di computare i modelli stabili di programmi logici con aggregati ricorsivi monotoni e antimonotoni. Le qualità del sistema sono state valutate in termini di correttezza ed efficienza da una serie di benchmark, che ne hanno mostrato le capacità e i limiti. Al momento, la nostra implementazione migliore, ovvero quella che fa uso di tutte le tecniche descritte nella tesi, è in grado di processare programmi con aggregati ricorsivi monotoni di taglia medio-grande e con aggregati antimonotoni di taglia medio-piccola. Particolare attenzione va rivolta ad un problema che spesso abbiamo preso ad esempio: *Company Controls*. Per risolvere questo problema, che è di estrema utilità per la valutazione dell'efficienza delle operazioni di aggregazione ricorsiva, il nostro solver raggiunge prestazioni notevoli. Infatti, la codifica che abbiamo utilizzato per *Company Controls* ha la proprietà di essere monotona e priva di disgiunzione. Queste caratteristiche consentono al nostro prototipo l'intero processamento in fase di grounding, sfruttando appieno la strategia semi-naive presentata in questa tesi.

Dunque, i risultati che abbiamo ottenuto possono essere utilizzati concretamente per risolvere problemi di natura applicativa. A prova della validità dei nostri risultati, questi sono stati precedentemente presentati in forma ridotta presso il *Convegno Italiano di Logica Computazionale* del 21-22 giugno 2007 (*CILC 2007*), tenutosi presso l'Università degli Studi di Messina, nell'articolo *Using Unfounded Sets for Computing Answer Sets of Programming with Recursive Aggregates* [**?**]. Inoltre, è prevista l'inclusione di queste nuove tecniche di reasoning in una prossima release ufficiale del sistema DLV, con ovvi vantaggi per la sua vasta comunità di utilizzatori.

Alcune delle caratteristiche teoriche del ricco linguaggio che abbiamo presentato e indagato a fondo nel corso della tesi sembrano promettere ulteriori e rilevanti miglioramenti per il prototipo sviluppato. Questi sviluppi consentirebbero il trattamento di istanze di taglia superiore rispetto a quelle attualmente processabili dal nostro sistema.

Appendice

Appendice A

Algoritmi per la computazione modulare del GUS

L'algoritmo di computazione del greatest unfounded set, implementato nel prototipo realizzato, è una versione ottimizzata di quella presentata nel capitolo 7.

La procedura principale è riportata in Figura A.1, e fa uso delle funzioni *isActive* e *evaluateAggregate* riportate in Figura A.2 e in Figuara A.3, rispettivamente.

Una regola r è attiva per una certa componente C rispetto ad un'interpretazione I se e solo se

(1) $H(r) \cap C \neq \emptyset$, e

(2) $B(r)$ non è falso rispetto ad I, e

(3) $H(r) \setminus (C \cap H(r)$ non è vero rispetto ad I.

La funzione *evaluateAggregate* restituisce *true* se l'aggregato, valutato rispetto all'interpretazione corrente e considerando come veri gli atomi indefiniti non appartenenti alla componente o appartenenti alla componente ma non-infondati, risulta essere vero.

```
Procedure (var C: Component, var I: Interpretation, var inconsistency: Boolean)
var a,b : Atom;
var FoundedAtoms : Interpretation;  % Stores the set of atoms of C which are proven
                                    % to be "founded" (not unfounded).
var GUSqueue : Queue;               % Stores the atoms whose "foundedness" is to be propagated;
                                    % controls the fixpoint computation.
var r.counter : Integer; (∀r)       % Stores the number of atoms of C in B+(r)
                                    % which are not proved to be founded.
                                    % if r.counter becomes zero, then the head of r gets founded.
inconsistency := false;
% Initialize the rules counters and the queue.
for each atom a ∈ C do
   if a is a recursive aggreggate then
      if evaluateAggregate( a, C, I, FoundedAtoms ) = True then
         FoundedAtoms.Add(a);
         GUSqueue.Push(a);
      end if;
   else if a is a stadard atom then
      for each rule r such that (isActive(r, C, I) and a ∈ H(()r)) do
         r.counter := |{b : b ∈ Mon(B(r)) ∩ C}|;
         if r.counter = 0 then
            FoundedAtoms.Add(a);
            GUSqueue.Push(a);
         end if;
      end for;
   end if;
end for;
% Fixpoint Computation.
while not GUSqueue.empty() do
   a := GUSqueue.Pop();
   for each rule r such that (isActive(r, C, I) and a ∈ Mon(B(r))) do
      r.counter := r.counter - 1;
      if r.counter = 0 then
         Let b be the atom of C in H(()r);
         FoundedAtoms.Add(b);
         GUSqueue.Push(b);
      end if;
   end for;
   for each recursive aggregate A in C such that (a ∈ AggregateSet(A)) do
      if evaluateAggregate( A, C, I, FoundedAtoms ) = True then
         FoundedAtoms.Add(A);
         GUSqueue.Push(A);
      end if;
   end for;
end while;
% Set to false all atoms of C which are not in FoundedAtoms.
for each atom a ∈ C do
   if a ∉ FoundedAtoms then
      if a ∈ I then
         inconsistency := true;
         return;
      else
         I := I ∪ {not a};
      end if;
   end if;
end for;
End Procedure;
```

Figura A.1: *La procedura* computeGUS

```
Function isActive(r: Rule, C: Component, I: Interpretation): Boolean
   if (H(r) ∩ C ≠ ∅ and
      B(r) is not false w.r.t. I and
      H(r) \ (C ∩ H(r)) is not true w.r.t. I ) then
         return true;
   else
         return false;
   end if;
End Function;
```

Figura A.2: *La funzione* isActive

```
Function evaluateAggregate(A: Aggregate, C: Component,
                           I: Interpretation, FoundendAtoms: Set): Boolean
   var A1: Aggregate;
   A1 := A;   % Make a copy of the aggregate
   for each atom a ∈ AggregateSet(A1) do
      if ( a is undefined w.r.t. I and
         ( a ∉ C or (a ∈ FoundedAtoms and not already propagated) ) )
            updateBounds(A1, a, true);
      end if;
   end for;
   return A1 is true;
End Function;
```

Figura A.3: *La funzione* evaluateAggregate

Appendice B

Codifiche dei problemi di benchmark e tabelle dei risultati

Codifiche

Di seguito sono riportate le codifiche utilizzate per l'esecuzione dei benchmark che non sono state mostrate nei capitoli precedenti.

Suite Asparagus

Bounded Spanning Tree (BST)

```
% by Wolfgang Faber
% based on a spanning tree encoding in my diploma thesis

bstedge(X,Y) :- reached(X), not differently_reached(Y,X), edge(X,Y).

vtx_outgoing(X) :- edge(X,_).
root(X) v nroot(X) :- vtx_outgoing(X).

:- #count{X:root(X)} < 1.
:- #count{X:root(X)} > 1.

:- vtx(X), not reached(X).
```

```
reached(X) :- root(X).
reached(X) :- bstedge(Y,X).

differently_reached(Y,X) :- root(Y), edge(X,Y).
differently_reached(Y,X) :- bstedge(Z,Y), Z != X, edge(X,Y).

:- vtx(X), #count{ Y: bstedge(X,Y) } > D, bound(D).
```

Hamiltonian Cycle (HAMILT-CYCLE)

```
% A DLV encoding for the Hamiltonian Cycle problem.
% Input graph is undirected.
% By Wolfgang Faber <wf@wfaber.com>

% Work on the directed representation. A directed HC of this exists iff an
% undirected HC exists in the undirected graph. It is just easier to deal with.
arc(X,Y) :- edge(X,Y).
arc(Y,X) :- edge(X,Y).

% Take the vertex identified by bound() as arbitrary starting point.
% Choose an outgoing arc and work the way along the thus constructed path.
in_hm(X,Y) v out_hm(X,Y) :- bound(X), arc(X,Y).
in_hm(X,Y) v out_hm(X,Y) :- reached(X), arc(X,Y).

reached(Y) :- in_hm(_,Y).

% Each node must have exactly one incoming and one outgoing arc of the HC.
:- vtx(X), not #count{ Y: in_hm(X,Y)} = 1.
:- vtx(Y), not #count{ X: in_hm(X,Y)} = 1.

% The HC must be connected.
:- vtx(X), not reached(X).

% Finally express the HC using edge representations of the input graph.
hc(X,Y) :- in_hm(X,Y), edge(X,Y).
hc(X,Y) :- in_hm(Y,X), edge(X,Y).
```

Weighted Spanning Tree (WST)

```
% by Wolfgang Faber
% based on a spanning tree encoding in my diploma thesis

wstedge(X,Y) :- reached(X), not differently_reached(Y,X), edge(X,Y).
```

```
vtx_outgoing(X) :- edge(X,_).
root(X) v nroot(X) :- vtx_outgoing(X).

:- #count{X:root(X)} < 1.
:- #count{X:root(X)} > 1.

:- vtx(X), not reached(X).

reached(X) :- root(X).
reached(X) :- wstedge(Y,X).

differently_reached(Y,X) :- root(Y), edge(X,Y).
differently_reached(Y,X) :- wstedge(Z,Y), Z != X, edge(X,Y).

:- vtx(X), #sum{ W,Y: wstedge(X,Y), wtedge(X,Y,W) } > D, bound(D).
```

Sokoban (SOKOBAN)

```
% Written by Wolfgang Faber <wf@wfaber.com>
%
% Input:
% right(L1,L2) : location L2 is right of location L1
% top(L1,L2): location L2 is on top of location L1
% box(L): location L initially holds a box
% solution(L): location L is a target for a box
% sokoban(L): the sokoban is at location L initially
% step(S): S is a step
% next(S1,S2): step S2 is the successor of step S1

% actionstep: can push at any step but the final one
actionstep(S) :- next(S,S1).

% utility predicates: left and bottom
left(L1,L2) :- right(L2,L1).
bottom(L1,L2) :- top(L2,L1).

% utility predicates: adjacent
adj(L1,L2) :- right(L1,L2).
adj(L1,L2) :- left(L1,L2).
adj(L1,L2) :- top(L1,L2).
adj(L1,L2) :- bottom(L1,L2).

% identify locations
location(L) :- adj(L,_).
```

```
% Initial configuration.
box_step(B,1) :- box(B).
sokoban_step(S,1) :- sokoban(S).

% push(B,D,B1,S):
% At actionstep S push box at location B in direction D (right, left, up, down)
% until location B1.
% The sokoban must be able to get to the location "before" B in order to push
% the box, also there should bot be any boxes between B and B1 (and also not
% on B1 itself at step S.
push(B,right,B1,S) v -push(B,right,B1,S) :- reachable(L,S), right(L,B), box_step(B,S),
        pushable_right(B,B1,S), good_pushlocation(B1), actionstep(S).
push(B,left,B1,S) v -push(B,left,B1,S) :- reachable(L,S), left(L,B), box_step(B,S),
        pushable_left(B,B1,S), good_pushlocation(B1), actionstep(S).
push(B,up,B1,S) v -push(B,up,B1,S) :- reachable(L,S), top(L,B), box_step(B,S),
        pushable_top(B,B1,S), good_pushlocation(B1), actionstep(S).
push(B,down,B1,S) v -push(B,down,B1,S) :- reachable(L,S), bottom(L,B), box_step(B,S),
        pushable_bottom(B,B1,S), good_pushlocation(B1), actionstep(S).

% reachable(L,S):
% Identifies locations L which are reachable by the sokoban at step S.
reachable(L,S) :- sokoban_step(L,S).
reachable(L,S) :- reachable(L1,S), adj(L1,L), not box_step(L,S).

% pushable_right(B,D,S):
% Box at B can be pushed right until D at step S.
% Analogous for left, top, bottom.
pushable_right(B,D,S) :- box_step(B,S), right(B,D), not box_step(D,S), actionstep(S).
pushable_right(B,D,S) :- pushable_right(B,D1,S), right(D1,D), not box_step(D,S).
pushable_left(B,D,S) :- box_step(B,S), left(B,D), not box_step(D,S), actionstep(S).
pushable_left(B,D,S) :- pushable_left(B,D1,S), left(D1,D), not box_step(D,S).
pushable_top(B,D,S) :- box_step(B,S), top(B,D), not box_step(D,S), actionstep(S).
pushable_top(B,D,S) :- pushable_top(B,D1,S), top(D1,D), not box_step(D,S).
pushable_bottom(B,D,S) :- box_step(B,S), bottom(B,D), not box_step(D,S), actionstep(S).
pushable_bottom(B,D,S) :- pushable_bottom(B,D1,S), bottom(D1,D), not box_step(D,S).

% The sokoban is at a new location after a push and no longer at its original
% position.
sokoban_step(L,S1) :- push(_,right,B1,S), next(S,S1), right(L,B1).
sokoban_step(L,S1) :- push(_,left,B1,S), next(S,S1), left(L,B1).
sokoban_step(L,S1) :- push(_,up,B1,S), next(S,S1), top(L,B1).
sokoban_step(L,S1) :- push(_,down,B1,S), next(S,S1), bottom(L,B1).
-sokoban_step(L,S1) :- push(_,_,_,S), next(S,S1), sokoban_step(L,S).

% Also the box_step has moved after having been pushed.
box_step(B,S1) :- push(_,_,B,S), next(S,S1).
```

```
-box_step(B,S1) :- push(B,_,_,S), next(S,S1).

% Inertia: Boxes and the sokoban usually remain where they are.
box_step(LB,S1) :- box_step(LB,S), next(S,S1), not -box_step(LB,S1).
sokoban_step(LS,S) :- sokoban_step(LS,S), next(S,S1), not -sokoban_step(LS,S1).

% Don't push two different boxes in one step.
:- push(B,_,_,S), push(B1,_,_,S), B != B1.
% Don't push a box in different directions in one step.
:- push(B,D,_,S), push(B,D1,_,S), D != D1.
% Don't push a box onto different locations in one step.
:- push(B,D,B1,S), push(B,D,B11,S), B1 != B11.

% Avoid pushing boxes into dead ends. There should be a location to the left
% and right or to top and bottom. Otherwise the box cannot be taken out again,
% for instance from corners. Obviously if the location is a target, these
% restrictions do not apply, as the box may remain there forever.
good_pushlocation(L) :- right(L,_), left(L,_).
good_pushlocation(L) :- top(L,_), bottom(L,_).
good_pushlocation(L) :- solution(L).

final_step(S) :- step(S), not no_final_step(S).
no_final_step(S) :- next(S,S1).

solution_found :- solution(L), final_step(S), not box_step(L,S).
:- solution_found.
```

Fastfood (FASTFOOD)

```
% This program checks whether a depot allocation has minimal supply
% costs among all depot allocations of the same cardinality.
%
% Input predicates:
% restaurant(Name,Km), depot(Name,Km)
% Output predicates:
% altdepot(Name,Km)
%
% Author: Wolfgang Faber <wf@wfaber.com>
% License: GNU Public License, http://www.gnu.org/licenses/gpl.html

% Maximum distance from a depot to a restaurant.
#maxint=910.

% Auxiliary predicate: All distances between restaurant locations.
distance(X,L,Y):- restaurant(_,X), restaurant(_,L), X>L,  X=L+Y.
```

```
distance(X,L,Y):- restaurant(_,X), restaurant(_,L), X<=L, L=X+Y.

% The supply distance for each restaurant for the candidate solution
% in the input.
serves(Rname,Dist) :- restaurant(Rname,RK), depot(Dname,DK),
        distance(RK,DK,Dist)
        Dist = #min {Y : depot(Dname1,DK1), distance(DK1,RK,Y) }.

% Each restaurant may be an alternative depot or not.
altdepot(R,K) v naltdepot(R,K) :- restaurant(R,K).

% The number of alternative depots must be equal to the number of depots in
% the input.
:- #count{D,K: depot(D,K)} = N, #count{D,K: altdepot(D,K)} > N.
:- #count{D,K: depot(D,K)} = N, #count{D,K: altdepot(D,K)} < N.

% The supply distance for each restaurant for the alternative solution.
altserves(Rname,Dist) :- restaurant(Rname,RK), altdepot(Dname,DK),
        distance(RK,DK,Dist),
        Dist = #min {Y : altdepot(Dname1,DK1), distance(DK1,RK,Y) }.

% Accept an alternative solution only if its supply costs are less
% than the supply costs for the input candidate.
:- #sum{Dist,R : serves(R,Dist)} = Cost,
   #sum{Dist,R : altserves(R,Dist)} >= Cost.
```

Traveling Salesperson Suite (TSS)

```
% A DLV encoding for the Hamiltonian Cycle problem.
% Input graph is undirected.
% By Wolfgang Faber <wf@wfaber.com>

% Work on the directed representation. A directed HC of this exists iff an
% undirected HC exists in the undirected graph. It is just easier to deal with.
arc(X,Y) :- edge(X,Y).
arc(Y,X) :- edge(X,Y).

% Take the vertex identified by bound() as arbitrary starting point.
% Choose an outgoing arc and work the way along the thus constructed path.
in_hm(X,Y) v out_hm(X,Y) :- bound(X), arc(X,Y).
in_hm(X,Y) v out_hm(X,Y) :- reached(X), arc(X,Y).

reached(Y) :- in_hm(_,Y).

% Each node must have exactly one incoming and one outgoing arc of the HC.
```

```
:- vtx(X), not #count{ Y: in_hm(X,Y)} = 1.
:- vtx(Y), not #count{ X: in_hm(X,Y)} = 1.

% The HC must be connected.
:- vtx(X), not reached(X).

% Finally express the HC using edge representations of the input graph.
hc(X,Y) :- in_hm(X,Y), edge(X,Y).
hc(X,Y) :- in_hm(Y,X), edge(X,Y).

:- maxweight(M), #sum{ W,X,Y: hc(X,Y), edgewt(X,Y,W) } > M.
```

Towers Of Hanoi Competition (HANOI)

```
#maxint=987654321.

number_of_moves(X):- steps(X).

largest_disc(X) :- #max{D:disk(D)}=X.

trasd(N,1):- largest_disc(N).
trasd(X,Y):- trasd(X1,Y1), #succ(X,X1), #succ(Y1,Y), X>3.

disc(X) :- trasd(_,X).

init_state1(0,1).
init_state1(V,H) :- init_state1(V1,H1), on0(H1,H), trasd(H,D),
                    AUX1=V1*10, V=AUX1+D.
init_state2(0,2).
init_state2(V,H) :- init_state2(V1,H1), on0(H1,H), trasd(H,D),
                    AUX1=V1*10, V=AUX1+D.
init_state3(0,3).
init_state3(V,H) :- init_state3(V1,H1), on0(H1,H), trasd(H,D),
                    AUX1=V1*10, V=AUX1+D.

initial_state(V1,V2,V3) :- #max{V:init_state1(V,_)}=V1,
        #max{V:init_state2(V,_)}=V2, #max{V:init_state3(V,_)}=V3.

goal_conf1(0,1).
goal_conf1(V,H) :- goal_conf1(V1,H1), ongoal(H1,H), trasd(H,D),
                   AUX1=V1*10, V=AUX1+D.
goal_conf2(0,2).
goal_conf2(V,H) :- goal_conf2(V1,H1), ongoal(H1,H), trasd(H,D),
                   AUX1=V1*10, V=AUX1+D.
goal_conf3(0,3).
```

```
goal_conf3(V,H) :- goal_conf3(V1,H1), ongoal(H1,H), trasd(H,D),
                   AUX1=V1*10, V=AUX1+D.

goal(V1,V2,V3) :- #max{V:goal_conf1(V,_)}=V1,
        #max{V:goal_conf2(V,_)}=V2, #max{V:goal_conf3(V,_)}=V3.


% from P1 to P2
put(S,D1,D2) :- move(S,V1,V2,V3), move(S1,W1,W2,W3),V1>W1,W2>V2,
                legalstack_top_rest(V1,T2,W1), legalstack_top_rest(V2,T1,_),
                trasd(D1,T1), trasd(D2,T2), V2!=0, S1=S+1.
put(S,2,D2) :- move(S,V1,0,V3), move(S1,W1,W2,W3),V1>W1,W2>0,
               legalstack_top_rest(V1,T2,W1),
               trasd(D2,T2), S1=S+1.
% from P1 to P3
put(S,D1,D2) :- move(S,V1,V2,V3), move(S1,W1,W2,W3),V1>W1,W3>V3,
                legalstack_top_rest(V1,T2,W1), legalstack_top_rest(V3,T1,_),
                trasd(D1,T1), trasd(D2,T2), V3!=0, S1=S+1.
put(S,3,D2) :- move(S,V1,V2,0), move(S1,W1,W2,W3),V1>W1,W3>0,
               legalstack_top_rest(V1,T2,W1),
               trasd(D2,T2), S1=S+1.


% from P2 to P1
put(S,D1,D2) :- move(S,V1,V2,V3), move(S1,W1,W2,W3),V2>W2,W1>V1,
                legalstack_top_rest(V2,T2,W2), legalstack_top_rest(V1,T1,_),
                trasd(D1,T1), trasd(D2,T2), V1!=0, S1=S+1.
put(S,1,D2) :- move(S,0,V2,V3), move(S1,W1,W2,W3),V2>W2,W1>0,
               legalstack_top_rest(V2,T2,W2),
               trasd(D2,T2), S1=S+1.
% from P2 to P3
put(S,D1,D2) :- move(S,V1,V2,V3), move(S1,W1,W2,W3),V2>W2,W3>V3,
                legalstack_top_rest(V2,T2,W2), legalstack_top_rest(V3,T1,_),
                trasd(D1,T1), trasd(D2,T2), V3!=0, S1=S+1.
put(S,3,D2) :- move(S,V1,V2,0), move(S1,W1,W2,W3),V2>W2,W3>0,
               legalstack_top_rest(V2,T2,W2),
               trasd(D2,T2), S1=S+1.


% from P3 to P1
put(S,D1,D2) :- move(S,V1,V2,V3), move(S1,W1,W2,W3),V3>W3,W1>V1,
                legalstack_top_rest(V3,T2,W3), legalstack_top_rest(V1,T1,_),
                trasd(D1,T1), trasd(D2,T2), V1!=0, S1=S+1.
put(S,1,D2) :- move(S,0,V2,V3), move(S1,W1,W2,W3),V3>W3,W1>0,
               legalstack_top_rest(V3,T2,W3),
               trasd(D2,T2), S1=S+1.
```

```
% from P3 to P2
put(S,D1,D2) :- move(S,V1,V2,V3), move(S1,W1,W2,W3),V3>W3,W2>V2,
                legalstack_top_rest(V3,T2,W3), legalstack_top_rest(V2,T1,_),
                trasd(D1,T1), trasd(D2,T2), V2!=0, S1=S+1.
put(S,2,D2) :- move(S,V1,0,V3), move(S1,W1,W2,W3),V3>W3,W2>0,
               legalstack_top_rest(V3,T2,W3),
               trasd(D2,T2), S1=S+1.

% Employ the notion of legal stacks, which carry information of the top
% element and the rest of the stack in addition to the entire stack itself.

% ------ legal non-empty stacks, their top element and the rest stack ------

legalstack_top_rest(D,D,0) :- disc(D).
legalstack_top_rest(S,T,B) :- legalstack_top_rest(B,T1,R1), disc(T), T < T1,
                              AUX = B * 10, S = AUX + T.

% ------ possible moves ------

possible_state(0,S1,S2,S3) :- initial_state(S1,S2,S3).
possible_state(I,S1,S2,S3) :- possible_move(I,_,_,_,S1,S2,S3).

% From stack one to stack two.

possible_move(I1,L1,L2,L3,S1,S2,L3) :-  possible_state(I,L1,L2,L3),
                               number_of_moves(J), I < J, #succ(I,I1),
                               legalstack_top_rest(L1,X,S1),
                               legalstack_top_rest(S2,X,L2).

% From stack one to stack three.

possible_move(I1,L1,L2,L3,S1,L2,S3) :-  possible_state(I,L1,L2,L3),
                               number_of_moves(J), I < J, #succ(I,I1),
                               legalstack_top_rest(L1,X,S1),
                               legalstack_top_rest(S3,X,L3).

% From stack two to stack one.

possible_move(I1,L1,L2,L3,S1,S2,L3) :-  possible_state(I,L1,L2,L3),
                               number_of_moves(J), I < J, #succ(I,I1),
                               legalstack_top_rest(L2,X,S2),
                               legalstack_top_rest(S1,X,L1).

% From stack two to stack three.

possible_move(I1,L1,L2,L3,L1,S2,S3) :-  possible_state(I,L1,L2,L3),
```

```
                               number_of_moves(J), I < J, #succ(I,I1),
                               legalstack_top_rest(L2,X,S2),
                               legalstack_top_rest(S3,X,L3).

% From stack three to stack one.

possible_move(I1,L1,L2,L3,S1,L2,S3) :-  possible_state(I,L1,L2,L3),
                               number_of_moves(J), I < J, #succ(I,I1),
                               legalstack_top_rest(L3,X,S3),
                               legalstack_top_rest(S1,X,L1).

% From stack three to stack two.

possible_move(I1,L1,L2,L3,L1,S2,S3) :-  possible_state(I,L1,L2,L3),
                               number_of_moves(J), I < J, #succ(I,I1),
                               legalstack_top_rest(L3,X,S3),
                               legalstack_top_rest(S2,X,L2).

%------ actual moves ------
% a solution exists if and only if there is a "possible_move"
% leading to the goal.
% in this case, starting from the goal, we proceed backward
% to the initial state to single out the full set of moves.

% Choose from the possible moves.

move(I,A1,A2,A3) :- goal(A1,A2,A3), possible_state(I,A1,A2,A3).

move(I,A1,A2,A3) v nomove(I,A1,A2,A3) :-
               move(J,S1,S2,S3), #succ(I,J),
               possible_move(J,A1,A2,A3,S1,S2,S3).

%------ at most one move at each step ------

:- move(I,L1,L2,L3), move(I,M1,M2,M3), L1 < M1.
:- move(I,L1,L2,L3), move(I,M1,M2,M3), L2 < M2.
:- move(I,L1,L2,L3), move(I,M1,M2,M3), L3 < M3.

%------ at least one move at each step ------

:- time(I), not moved(I).

moved(I) :- move(I,_,_,_).
```

Risultati

Suite Company Controls

In Tabella 9.3 e in Tabella 9.4 abbiamo riportato i risultati ottenuti per la suite Company Controls. La prima colonna riporta il numero di compagnie presenti nell'istanza. Il numero di predicati *owns* è sempre pari a 5 volte il numero di compagnie. Le successive colonne della Tabella 9.3 rappresentano il tempo medio di esecuzione dei diversi sistemi testati, mentre in Tabella 9.4 sono riportati i tempi massimi di esecuzione.

Compagnie	DLV$^{\mathcal{A}}$	DLV$^{\mathcal{A}}_{\mathcal{I}}$	DLV$^{\mathcal{A}}_{\mathcal{I}+\mathcal{A}}$	*SMODELS*	*CMODELS*	*CLASP*
30	0.67	0.03	0.01	0.10	0.68	0.10
60	5.44	0.06	0.02	0.54	21.24	0.52
100	22.26	0.19	0.04	1.59	-	1.63
200	-	0.91	0.09	7.62	-	7.53
300	-	1.66	0.12	19.69	-	19.71
400	-	3.33	0.18	39.21	-	39.52
500	-	4.52	0.20	68.58	-	69.48
600	-	7.49	0.28	110.20	-	-
700	-	10.82	0.35	159.64	-	-
800	-	13.30	0.38	-	-	-
900	-	20.16	0.51	-	-	-
1000	-	20.69	0.49	-	-	-
1100	-	28.42	0.58	-	-	-
1200	-	33.88	0.66	-	-	-
1300	-	41.86	0.75	-	-	-
1400	-	48.99	0.80	-	-	-
1500	-	50.05	0.79	-	-	-
1600	-	56.94	0.86	-	-	-
1700	-	64.65	0.92	-	-	-
1800	-	81.60	1.08	-	-	-
1900	-	96.76	1.20	-	-	-
2000	-	113.61	1.33	-	-	-
2100	-	100.51	1.15	-	-	-
2200	-	118.18	1.29	-	-	-
2300	-	121.37	1.27	-	-	-
2400	-	132.69	1.31	-	-	-
2500	-	183.17	1.69	-	-	-
2600	-	179.03	1.61	-	-	-
2700	-	215.71	1.85	-	-	-
2800	-	208.77	1.75	-	-	-
2900	-	225.53	1.80	-	-	-
3000	-	242.87	1.87	-	-	-
3100	-	260.35	1.95	-	-	-
3200	-	279.10	2.03	-	-	-
3300	-	-	2.42	-	-	-
3400	-	-	2.16	-	-	-
3500	-	-	2.45	-	-	-
3600	-	-	2.41	-	-	-
3700	-	-	2.28	-	-	-
3800	-	-	2.79	-	-	-
3900	-	-	2.88	-	-	-
4000	-	-	2.69	-	-	-
4100	-	-	2.79	-	-	-
4200	-	-	2.98	-	-	-
4300	-	-	3.05	-	-	-
4400	-	-	3.14	-	-	-
4500	-	-	3.22	-	-	-
4600	-	-	3.28	-	-	-
4700	-	-	3.03	-	-	-
4800	-	-	3.30	-	-	-
4900	-	-	3.91	-	-	-
5000	-	-	3.25	-	-	-
5100	-	-	3.87	-	-	-
5200	-	-	3.55	-	-	-
5300	-	-	4.14	-	-	-
5400	-	-	4.10	-	-	-
5500	-	-	4.29	-	-	-
5600	-	-	4.03	-	-	-
5700	-	-	4.53	-	-	-
5800	-	-	4.17	-	-	-
5900	-	-	4.69	-	-	-
6000	-	-	4.35	-	-	-
6100	-	-	4.21	-	-	-
6200	-	-	4.63	-	-	-
6300	-	-	4.80	-	-	-
6400	-	-	5.33	-	-	-

Tabella B.1: *Risultati suite Company Controls (tempo medio di esecuzione)*

Compagnie	$\text{DLV}^{\mathcal{A}}$	$\text{DLV}^{\mathcal{A}}_{\mathcal{I}}$	$\text{DLV}^{\mathcal{A}}_{\mathcal{I}+\mathcal{A}}$	*SMODELS*	*CMODELS*	*CLASP*
30	0.86	0.06	0.03	0.13	1.09	0.13
60	5.56	0.07	0.02	0.57	55.11	0.57
100	22.48	0.26	0.05	1.63	-	1.67
200	-	1.02	0.11	7.99	-	7.70
300	-	1.76	0.12	20.48	-	20.92
400	-	3.97	0.22	39.80	-	40.43
500	-	4.57	0.21	70.15	-	70.75
600	-	11.52	0.42	114.08	-	-
700	-	12.20	0.40	165.99	-	-
800	-	20.26	0.56	-	-	-
900	-	20.29	0.52	-	-	-
1000	-	24.85	0.58	-	-	-
1100	-	30.19	0.62	-	-	-
1200	-	36.00	0.70	-	-	-
1300	-	53.47	0.94	-	-	-
1400	-	62.11	1.01	-	-	-
1500	-	56.75	0.89	-	-	-
1600	-	82.31	1.18	-	-	-
1700	-	72.97	1.02	-	-	-
1800	-	104.17	1.34	-	-	-
1900	-	117.72	1.43	-	-	-
2000	-	131.11	1.52	-	-	-
2100	-	113.56	1.28	-	-	-
2200	-	126.19	1.36	-	-	-
2300	-	137.01	1.41	-	-	-
2400	-	150.32	1.48	-	-	-
2500	-	257.03	2.29	-	-	-
2600	-	180.90	1.62	-	-	-
2700	-	249.15	2.13	-	-	-
2800	-	209.42	1.76	-	-	-
2900	-	226.81	1.81	-	-	-
3000	-	243.69	1.88	-	-	-
3100	-	261.40	1.96	-	-	-
3200	-	280.85	2.08	-	-	-
3300	-	-	2.70	-	-	-
3400	-	-	2.17	-	-	-
3500	-	-	3.33	-	-	-
3600	-	-	2.86	-	-	-
3700	-	-	2.52	-	-	-
3800	-	-	3.04	-	-	-
3900	-	-	3.13	-	-	-
4000	-	-	3.23	-	-	-
4100	-	-	3.45	-	-	-
4200	-	-	3.39	-	-	-
4300	-	-	4.15	-	-	-
4400	-	-	3.58	-	-	-
4500	-	-	3.63	-	-	-
4600	-	-	3.73	-	-	-
4700	-	-	3.76	-	-	-
4800	-	-	4.05	-	-	-
4900	-	-	5.32	-	-	-
5000	-	-	3.27	-	-	-
5100	-	-	5.19	-	-	-
5200	-	-	4.21	-	-	-
5300	-	-	4.33	-	-	-
5400	-	-	4.60	-	-	-
5500	-	-	5.41	-	-	-
5600	-	-	4.61	-	-	-
5700	-	-	5.79	-	-	-
5800	-	-	4.75	-	-	-
5900	-	-	5.82	-	-	-
6000	-	-	4.92	-	-	-
6100	-	-	5.05	-	-	-
6200	-	-	5.97	-	-	-
6300	-	-	5.25	-	-	-
6400	-	-	7.43	-	-	-

Tabella B.2: *Risultati suite Company Controls (tempo massimo di esecuzione)*

Bibliografia

[1] Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: JELIA 2004. LNCS 3229, (2004) 200–212

[2] Kemp, D.B., Stuckey, P.J.: Semantics of Logic Programs with Aggregates. In: ISLP'91, MIT Press (1991) 387–401

[3] Denecker, M., Pelov, N., Bruynooghe, M.: Ultimate Well-Founded and Stable Model Semantics for Logic Programs with Aggregates. In: ICLP-2001, (2001) 212–226

[4] Gelfond, M.: Representing Knowledge in A-Prolog. In: Computational Logic. Logic Programming and Beyond. LNCS 2408 (2002) 413–451

[5] Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. AI **138** (2002) 181–234

[6] Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In: IJCAI 2003, Acapulco, Mexico,(2003) 847–852

[7] Pelov, N., Truszczyński, M.: Semantics of disjunctive programs with monotone aggregates - an operator-based approach. In: NMR 2004. (2004) 327–334

[8] Pelov, N., Denecker, M., Bruynooghe, M.: Partial stable models for logic programs with aggregates. In: LPNMR-7. LNCS 2923, (2004) 207–219

[9] Son, T.C., Pontelli, E.: A Constructive Semantic Characterization of Aggregates in ASP. TPLP **7** (2007) 355–375

[10] Son, T.C., Pontelli, E., Elkabani, I.: On Logic Programming with Aggregates. Tech. Report NMSU-CS-2005-006, New Mexico State University (2005)

[11] Leone, N., Rullo, P., Scarcello, F.: Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. Inf.Comp. **135**(2) (1997) 69–112

[12] Calimeri, F., Faber, W., Leone, N., Pfeifer, G.: Pruning Operators for Answer Set Programming Systems. In: NMR'2002. (2002) 200–209

[13] Koch, C., Leone, N., Pfeifer, G.: Enhancing Disjunctive Logic Programming Systems by SAT Checkers. AI **15**(1–2) (2003) 177–212

[14] Pfeifer, G.: Improving the Model Generation/Checking Interplay to Enhance the Evaluation of Disjunctive Programs. In: LPNMR-7. LNCS 2923, (2004) 220–233

[15] Lee, J.: A Model-Theoretic Counterpart of Loop Formulas. `http://www.cs.utexas.edu/users/appsmurf/papers/mtclf.pdf` (2004)

[16] Lin, F., Zhao, Y.: ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In: AAAI-2002, Edmonton, Alberta, Canada, AAAI Press / MIT Press (2002)

[17] Lee, J., Lifschitz, V.: Loop Formulas for Disjunctive Logic Programs. In: Proceedings of the Nineteenth International Conference on Logic Programming (ICLP-03), (2003) 451–465

[18] Ferraris, P.: Answer Sets for Propositional Theories. `http://www.cs.utexas.edu/users/otto/papers/proptheories.ps` (2004)

[19] Calimeri, F., Faber, W., Leone, N., Perri, S.: Declarative and Computational Properties of Logic Programs with Aggregates. In: IJCAI 2005. (2005) 406–411

[20] Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: IJCAI 2007,(2007) 386–392

[21] Lierler, Y., Maratea, M.: Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In: LPNMR-7. LNCS 2923, (2004) 346–350

[22] Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: LPNMR'05. LNCS 3662, (2005) 447–451

[23] Faber, W.: Decomposition of Nonmonotone Aggregates in Logic Programming. WLP 2006 164–171

[24] Faber, W., Leone, N.: On the Complexity of Answer Set Programming with Aggregates. In: LPNMR'07. LNCS 4483, (2007) 97–109

[25] Leone, N., Rullo, P., Scarcello, F.: Declarative and Fixpoint Characterizations of Disjunctive Stable Models. In: ILPS'95, Portland, Oregon, MIT Press (1995) 399–413

[26] Van Gelder, A., Ross, K.A., Schlipf, J.S.: The Well-Founded Semantics for General Logic Programs. JACM **38**(3) (1991) 620–650

[27] Faber, W.: Unfounded Sets for Disjunctive Logic Programs with Arbitrary Aggregates. In: LPNMR'05. LNCS 3662, (2005) 40–52

[28] Ullman, J.D.: Principles of Database and Knowledge Base Systems. Computer Science Press (1989)

[29] Ben-Eliyahu, R., Dechter, R.: Propositional Semantics for Disjunctive Logic Programs. AMAI **12** (1994) 53–87

[30] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL **7**(3) (2006) 499–562

[31] Ricca, F., Faber, W., Leone, N.: A Backjumping Technique for Disjunctive Logic Programming. AI Communications **19**(2) (2006) 155–172

[32] Faber, W.: Enhancing Efficiency and Expressiveness in Answer Set Programming Systems. PhD thesis, TU Wien (2002)

[33] Calimeri, F., Faber, W., Leone, N., Pfeifer, G.: Pruning Operators for Disjunctive Logic Programming Systems. FI **71**(2–3) (2006) 183–214

[34] Chimenti, D., Gamboa, R., Krishnamurthy, R., Naqvi, S.A., Tsur, S., Zaniolo, C.: The LDL System Prototype. IEEE TKDE **2**(1) (1990)

[35] Ross, K.A., Sagiv, Y.: Monotonic Aggregation in Deductive Databases. JCSS **54**(1) (1997) 79–97

[36] Kemp, D.B., Ramamohanarao, K.: Efficient Recursive Aggregation and Negation in Deductive Databases. IEEE TKDE **10** (1998) 727–745

[37] Pelov, N.: Semantics of Logic Programs with Aggregates. PhD thesis, Katholieke Universiteit Leuven, Leuven, Belgium (2004)

[38] Eiter, T., Gottlob, G., Veith, H.: Modular Logic Programming and Generalized Quantifiers. In: LPNMR'97. LNCS 1265, (1997) 290–309

[39] Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate Functions in DLV. In: ASP'03, Messina, Italy (2003) 274–288 Online at `http://CEUR-WS.org/Vol-78/`.

[40] Niemelä, I., Simons, P., Soininen, T.: Stable Model Semantics of Weight Constraint Rules. In: LPNMR'99. LNCS 1730, (1999) 107–116

[41] Pelov, N., Denecker, M., Bruynooghe, M.: Well-founded and Stable Semantics of Logic Programs with Aggregates. TPLP **7**(3) (2007) 301–353

[42] Ferraris, P.: Answer Sets for Propositional Theories. In: LPNMR'05. LNCS 3662, (2005) 119–131

[43] Leone, N., Rullo, P., Scarcello, F.: Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. Tech. Report CD-TR 96/98, Christian Doppler Laboratory for Expert Systems, TU Vienna, Austria, Portland, Oregon (1996) (This is an enhanced version of [25].).

[44] Syrjänen, T.: Lparse 1.0 User's Manual (2002) `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

[45] Lierler, Y.: Cmodels for Tight Disjunctive Logic Programs. In: W(C)LP 19th Workshop on (Constraint) Logic Programming, Ulm, Germany. Ulmer Informatik-Berichte, Universität Ulm, Germany (2005) 163–166

[46] Ferraris, P., Lifschitz, V.: Weight constraints as nested expressions. TPLP **5**(1–2) (2005) 45–74

Ringraziamenti

Ringrazio il gruppo di sviluppo di DLV *e in particolare: Wolfgang Faber e Nicola Leone, senza i quali non avrei raggiunto nessuno dei risultati presentati; Francesco Calimeri, Tina Dell'Armi, Giovambattista Ianni, Simona Perri e Francesco Ricca, per l'indispensabile aiuto che mi hanno offerto.*

Vorrei infine ringraziare Gisella per i suggerimenti su come esporre i concetti presenti nella tesi e per l'affetto dimostrato nei miei confronti.