Asp

Valentina Pitoni

Indice

1 Answer Set Programming **2**
1.1 Concetti base 3
1.2 Sottoclassi dell'AnsProlog* 5
1.3 Answer set dei programmi AnsProlog$^{-\text{not}}$ 7
1.4 Answer set dei programmi AnsProlog 9
1.5 Answer set in AnsProlog$^{\neg}$ 11
1.6 AND/OR Handle 15
1.6.1 Grafi ciclici 20
1.7 Risolutori: SMODELS 21
1.7.1 Lparse 21
1.7.2 Smodels 22

Capitolo 1

Answer Set Programming

L' *Answer Set Programming* (*ASP*) é una forma di programmazione dichiarativa basata sulla collezione di dichiarazioni che descrivono gli oggetti di un dominio e le loro proprietá. Inoltre é improntato su un particolare tipo di programmazione logica [1] che ha una specifica semantica conosciuta come *answer set (o stable models) semantics* e usa come linguaggio un sottoinsieme del *Prolog*, l'*AnsProlog** [2]. Grazie all'espressivitá della sintassi e all'efficienza dei risolutori, l'Answer Set Programming viene sempre piú frequentemente impiegato per codificare problemi di planning (pianificazione), diagnosi e , piú in generale, problemi di rappresentazione della conoscenza. Questo linguaggio ha le seguenti caratteristiche:

- é un formalismo basato sulla semantica, infatti l'ordine dei letterali non deve avere alcuna importanza;
- l'esecuzione di un goal avviene in bottom-up[3] quindi non si cade in loop banali, come ad esempio nel caso seguente:
  $a \leftarrow \text{not } b$.
  $b \leftarrow \text{not } a$.

[1] Questo paradigma di programmazione é dichiarativo e interattivo. Per dichiarativo intendiamo che é caratterizzato dallo specificare il problema da risolvere piuttosto che l'algoritmo per risolvere tale problema. Vi sono due vantaggi di questo approccio: in primo luogo che il programma puó essere facilmente aggiornato con nuove informazioni senza la necessitá di sviluppare algoritmi aggiuntivi, e in secondo luogo che il programma puó produrre non solo il risultato, ma una spiegazione di come siamo arrivati a tale risultato. Si osserva anche che la programmazione logica ha anche un'interpretazione procedurale.

[2] La $*$ in AnsProlog* sta ad indicare che non vi é nessuna restrizione sulle regole che possono essere rappresentate dal linguaggio.

[3] Particolare strategia di elaborazone.

- il costrutto extra-logico *cut* non é presente (al massimo c'é un pó di aritmetica e i vincoli di cardinalitá);
- viene utilizzata la semantica dell'answer sets per evitare problemi di fallimento del programma in presenza di negazioni;
- non monotonicitá (l'aggiunta di nuove conoscenze porta a ritrarre precedenti conclusioni e quindi si possono invalidare alcune delle conclusioni precedentemente derivabili);
- il codice viene scritto avendo in mente la semantica del modello stabile[4].

1.1 Concetti base

La teoria dell'answer sets é costituita da un alfabeto e un linguaggio Γ basato su tale alfabeto. Fanno parte di questo alfabeto le costanti, le variabili, i simboli che rappresentano i predicati, i simboli per la punteggiatura e i simboli matematici. Quindi possiamo dire che siamo in presenza di una four-tuple $\sigma =< O, F, P, V >$ dove O sono gli oggetti, F sono le funzioni, P sono i predicati (usati per denominare le relazioni tra gli oggetti del dominio) ed infine V sono le variabili. Ogni funzione ed ogni predicato sono associati con la loro *arietá*, ovvero il numero degli argomenti che richiede la funzione, e possiamo definire un termine *ground* se non compaiono variabili in esso. Inoltre un *atomo* (o *letterale*) risulta essere una formula del tipo $p(t_1...t_n)$ dove p é il predicato e i t_i sono termini(variabili o costanti). L'insieme dei termini gruond che possono essere formati da costanti e funzioni in Γ definisce l'*Herbrand Universe* (HU_Γ) ed invece l'insieme degli atomi che possono essere formati dai predicati in Γ e dai termini in HU_Γ mi definisce l'*Herbrand Base* (HB_Γ). Risulta inoltre che gli answer sets di uno specifico programma sono un sottoinsieme dell'HB_Γ, noto come interpretazione di Herbrand.

Esempio 1.
$naturalnumber(0)$.

[4]La definizione di Gelfond e Lifshitz tramite la trasformazione: $\Gamma(P, S)$ (P= programma, S = insiemi di atomi) é il programma ottenuto da P attraverso l'eliminazione delle clausole not A nel corpo $\forall A \in S$, ed eliminando tutti gli atomi negati dai corpi delle rimanenti clausole. Quindi S é un modello stabile per P $\Leftrightarrow$ S é un modello minimo di $\Gamma(P, S)$.

$naturalnumber(s(X)) \leftarrow naturalnumber(X).$

Risulta che:
$HU_\Gamma = \{0, s(0), s(s(0)), ...\}$
$HB_\Gamma = \{naturalnumber(0), naturalnumber(s(0)), naturalnumber(s(s(0))), ...\}$

Esempio 2.
$parent(terach, abraham).$
$parent(isaac, jacob).$
$parent(abraham, isaac).$
$parent(jacob, benjamin).$
$ancestor(X, Y) \leftarrow parent(X, Y).$
$ancestor(X, Z) \leftarrow parent(X, Y), ancestor(Y, Z).$

Risulta che:
$HU_\Gamma = \{terach, abraham, isaac, jacob, benjamin\}$
$HB_\Gamma = \{parent(terach, isaac), parent(terach.abraham), ...\}.$

Definizione 1.1.1.
Una *regola* (ASP) é una clausola della forma:

L_0or ... or$L_k \leftarrow L_{k+1}$&...&L_m& notL_{m+1}... & not L_n

dove gli L_i sono delle formule atomiche con $k, m, n \geq 0$, e *not* é un connettivo chiamato *negazione da fallimento*. Tale regola é detta *ground* se tutti gli atomi sono ground, inoltre la parte alla sinistra della $\leftarrow$ viene detta *testa* della regola ed invece quella alla destra *corpo* della regola. Una regola col corpo privo di atomi viene detta *fatto*, d'altro canto una regola priva di testa viene detta *costrizione* o piú comunemente vincolo. Il vincolo ha l'effetto di invalidare ogni answer set che ne soddisfi il corpo, ovvero ogni insieme di atomi che contenga tutti gli atomi L_1, ...,L_m e nessuno degli L_{m+1},...,L_n. Inoltre possiamo dire che sono una estensione comoda ma puramente sintattica. Tale regola puó anche essere scritta come:

L_0or ... or$L_k \leftarrow L_{k+1}$,...,L_m ,not L_{m+1}...,not L_n

Possiamo dire quindi che un programma in AnsProlog* é un insieme di regole del tipo appena descritto sopra.

Un'interpretazione di un programma é un assegnamento di valori agli atomi ground nella Herbrand Base (interpretazione di Herbrand) in modo che nell'interpretazione alcuni atomi sono veri ed altri sono falsi. Nel Programma una regola (tipo 1.1.1) puó essere soddisfatta dall'interpretazione, se soddisfa una delle seguenti condizioni:

- se tutti i letterali positivi nel corpo sono nell'interpretazione, e nessuno dei letterali negativi nel corpo sono nell'interpretazione, allora il letterale nella testa é in tale interpretazione;

- se non c'é nessun letterale nella testa di una regola (cioé un vincolo), allora l'insieme di tutti i letterali positivi del corpo non sono nell'interpretazione, o esiste un letterale negativo del corpo che é nell'interpretazione

Se una interpretazione di Herbrand soddisfa tutte le regole in un programma, allora viene detta *modello di Herbrand* del programma.

1.2 Sottoclassi dell'AnsProlog*

Esistono differenti sottoclassi dell'AnsProlog*:

1. AnsProlog: un insieme di regole 1.1.1 dove L_i sono atomi e $k = 0$;

   Esempio 3.
   $jumps(X) \leftarrow mammal(X), \text{not} ab(X).$
   $ab(X) \leftarrow elephant(X).$
   $mammal(X) \leftarrow elephant(X).$
   $mammal(larry) \leftarrow .$
   $elephant(titus) \leftarrow .$

   Da questo esempio possiamo concludere che larry salta ed invece titus no (perché é un elefante quindi gli elefanti sono anormali in questo senso)

2. AnsProlog$^{-\text{not}}$: un insieme di regole 1.1.1 dove L_i sono atomi, $k = 0$ e $m = n$, questi programmi sono molto simili ai programmi in Prolog

che non fanno uso di negazioni o del cut. Inoltre un'altra differenza tra Prolog e AnsProlog$^{-\text{not}}$ (AnsProlog* in generale) é che la semantica del Prolog é definita rispetto a un meccanismo di deduzione fissa, mentre l'AnsProlog* no.

Esempio 4.
$flight(X, Z) \leftarrow flight(X, Y), flight(Y, Z).$
$flight(roma, juventus).$
$flight(juventus, inter).$

Questo risulta essere anche un esempio della chiusura transitiva di una relazione data. Possiamo concludere che c' é un combattimento tra roma e inter (inoltre i fatti esprimono che vi é un combattimento anche tra roma e juventus e tra juventus e inter).

3. AnsProlog$^{\neg}$: un insieme di regole 1.1.1 dove $k = 0$;

   Esempio 5.
   $jumps(X) \leftarrow mammal(X), \text{not}\neg jumps(X).$
   $\neg jumps(X) \leftarrow elephant(X).$
   $mammal(X) \leftarrow elephant(X).$
   $mammal(larry) \leftarrow .$
   $elephant(titus) \leftarrow .$

   Questo esempio, come il primo, esprime sempre il concetto che larry salta ed invece titus no.

4. AnsProlog^{or}: un insieme di regole 1.1.1 dove L_i sono atomi. L'operatore or é differente dall'operatore $\bigvee$ infatti é una disgiunzione epistemica e quindi se abbiamo L_1 or L_2 significa che viene ritenuta vera o L_1 o L_2 pertanto $L_1 \text{or} \neg L_1$ non é sempre vera.

   Esempio 6.
   $insect(X) \text{or} bird(X) \leftarrow flies(X).$
   $flies(larry) \leftarrow .$

   Questo é un semplice esempio che ci fa capire che larry é un insetto o un uccello ma non possiamo concludere che sia entrambi.

5. se ammettiamo la presenza di vincoli avremo allora AnsProlog$^{\perp}$, AnsProlog$^{-\text{not},\perp}$, AnsProlog$^{\neg,\perp}$ e AnsProlog$^{or,\perp}$;

6. AnsProlog*(n): equivale all'AnsProlog* con al piú n letterali nel corpo delle regole.

1.3 Answer set dei programmi AnsProlog$^{-\text{not}}$

L'interpretazione di Herbrand di un programma AnsProlog$^{\perp}$ Γ é un sottoinsieme $I \subseteq HB_{\Gamma}$. Un answer set é definito come una particolare interpretazione di Herbrand che soddisfa le proprietá del programma ed inoltre é minimale ovvero significa che preso un'interpretazione I non ne esiste un'altra che sia un suo sottoinsieme proprio. Inoltre l'introduzione di questi answer set riducono notevolmente i problemi computazionali soprattutto quando siamo in presenza di negazione.
Diciamo che un'interpretazione di Herbrand S di Π soddisfa la regola 1.1.1 dell'AnsProlog$^{\perp}$ se:

- $L_0 \neq 0 : \{L_1...L_m\} \subseteq S$ e $\{L_{m+1}...L_n\} \cap S = \emptyset \Rightarrow L_0 \in S$;
- $L_0 = 0 : \{L_1...L_m\} \nsubseteq S$ e $\{L_{m+1}...L_n\} \cap S \neq \emptyset$.

Possiamo affermare che, vista l'assenza di negazione, i programmi in AnsProlog$^{-\text{not}}$ hanno un unico answer set, che é sempre un modello minimo[5], e tali programmi vengono detti *categorici*; inoltre l'intersezione di modelli di Herbrand ci dá proprio il modello minimo di interesse (la dimostrazione della prima affermazione la riprenderemo dopo aver fornito altre nozioni basilari). Si puó presentare anche il caso in cui non vi é alcun answer set. D'altra parte peró non tutti i modelli minimi sono degli answer set infatti per esserlo nessun atomo che é vero nel modello dipende (direttamente o indirettamente) dalla negazione di un altro atomo sempre vero nel modello.

Esempio 7.
$p \leftarrow a.$
$q \leftarrow b.$
$a \leftarrow .$

L'insieme $\{a, b, p, q\}$ é un modello del programma e soddisfa tutte le regole ma notiamo che anche $\{a, p, q\}$ e $\{a, p\}$ sono dei modelli. Ad ogni modo l'insieme $\{a, p, q\}$ non é un answer set infatti: se b non compare nell'answer

[5]Fanno parte dei modelli minimi i fatti e tutto ció che deriva in modo aciclico da essi (not A vale solo se non vale A).

set lo stiamo assumendo falso quindi di conseguenza, per la regola $q \leftarrow b.$, abbiamo che anche p é falso e non dove comparire nel nostro answer set come invece succede. Considerando inoltre che l'insieme $\{a, p\}$ risulta essere un sottoinsieme di $\{a, b, p, q\}$ abbiamo il modello minimo cercato ovvero l'answer set di questo programma é proprio $\{a, p\}$.

Un altro metodo molto importante per trovare l'answer set é quello del *punto fisso.*
Denotiamo con 2^{HB_Γ} l'insieme di tutte le interpretazioni di Herbrand in Γ e definiamo un operatore lineare ${T_\Gamma}^0 : 2^{HB_\Gamma} \longmapsto 2^{HB_\Gamma}$ come segue:

$${T_\Gamma}^0 = \{L_0 \in HB_\Gamma | \Gamma \text{ contiene una regola } L_0 \leftarrow L_1 ... L_m \text{ tale che } \{L_1 ... L_m\} \subseteq I\}$$

Intuitivamente vediamo che questo operatore é un insieme di atomi che possono essere derivati da una singola applicazione di Γ dando gli atomi in I ed inoltre é un operatore monotono ovvero se $I \subseteq I^1 \Rightarrow {T_\Gamma}^0(I) \subseteq {T_\Gamma}^0(I^1)$. Denotiamo ${T_\Gamma}^0 \uparrow 0$ l'insieme vuoto ed inoltre ${T_\Gamma}^0 \uparrow (i+1) = {T_\Gamma}^0({T_\Gamma}^0 \uparrow i)$. Chiaramente ${T_\Gamma}^0 \uparrow 0 \subseteq {T_\Gamma}^0 \uparrow 1$ e per monotonicitá di ${T_\Gamma}^0$ e transitivitá dell'operazione $\subseteq$ abbiamo che ${T_\Gamma}^0 \uparrow i \subseteq {T_\Gamma}^0 \uparrow (i+1)$. Nel caso di un Herbrand Base finita é possibile vedere che ripetendo l'operatore ${T_\Gamma}^0$ partendo dall'insieme vuoto arriveremo al punto fisso di ${T_\Gamma}^0$. Questo punto fisso sará l'answer set del nostro programma.

Esempio 8.
$p \leftarrow a.$
$q \leftarrow b.$
$a \leftarrow .$

Per definizione, ${T_\Gamma}^0 \uparrow 0 = \emptyset$
${T_\Gamma}^0 \uparrow 1 = {T_\Gamma}^0({T_\Gamma}^0 \uparrow 0) = \{a\}$
${T_\Gamma}^0 \uparrow 2 = {T_\Gamma}^0({T_\Gamma}^0 \uparrow 1) = \{a, p\}$
${T_\Gamma}^0 \uparrow 3 = {T_\Gamma}^0({T_\Gamma}^0 \uparrow 2) = \{a, p\} = {T_\Gamma}^0 \uparrow 2$
Quindi l'anwer set cercato é proprio $\{a, p\}$ (come avevamo visto nell'esempio precedente).

1.4 Answer set dei programmi AnsProlog

I programmi AnsProlog sono delle superclassi di quelli AnsProlog$^{-\text{not}}$ dove é permesso l'utilizzo della negazione nel corpo della regola. L'answer set nel programmi AnsProlog$^{-\text{not}}$ possono avere piú modelli minimali e logicamente non tutti hanno senso. Potremo utilizzare sempre il metodo del punto fisso ma come vedremo l'estensione all'AnsProlog$^{-\text{not}}$ dell'operatore ${T_\Gamma}^0$ non é monotona. Infatti:

$${T_\Gamma}^1(I) = \{L_0 \in HB_\Gamma | \Gamma \text{ contiene una regola } L_0 \leftarrow L_1 ... L_m, \text{ not } L_{m+1} ..., \text{ not } L_n \text{ tale che } \{L_1 ... L_m\} \subseteq I, \{ \text{ not } L_{m+1} ..., \text{ not } L_n\} \cap I = \emptyset\}$$

Che ha lo stesso comportamento dell'operatore non esteso. Adesso vediamo tramite un esempio quello che abbiamo giá detto precedentemente, ovvero che ${T_\Gamma}^1$ non é monotono.

Esempio 9.
$a \leftarrow$ not b
$b \leftarrow$ not a

Per definizione, ${T_\Gamma}^1 \uparrow 0 = \emptyset$
${T_\Gamma}^1 \uparrow 1 = {T_\Gamma}^1({T_\Gamma}^1 \uparrow 0) = \{a, b\}$
${T_\Gamma}^1 \uparrow 2 = {T_\Gamma}^1({T_\Gamma}^1 \uparrow 1) = \emptyset$
${T_\Gamma}^1 \uparrow 3 = {T_\Gamma}^1({T_\Gamma}^1 \uparrow 2) = \{a, b\}$
Si oscilla tra $\emptyset$ e $\{a, b\}$ quindi non abbiamo un punto fisso che possiamo prendere come answer set del nostro programma.

Considerando un programma AnsProlog prendiamo l'insieme S degli atomi e chiamiamo Γ^S il programma ottenuto da Γ eliminando:

- le regole che hanno not L nel corpo con $L \in S$;
- tutte le formule dei tipo not L nel corpo delle restanti regole.

Chiaramente Γ^S non contiene negazioni, quindi siamo in presenza di un programma AnsProlog$^{-\text{not}}$ e avremo un unico answer set. Se questo answer set coinciderá con S potremo dire che S é l'answer set di Γ. Inoltre abbiamo la seguente proprietá:
Sia M un answer set di Γ allora é un modello di Γ e per ogni M^1, dove M^1 é un modello di Γ^M, abbiamo $M \subseteq M^1$.

Esempio 10.
$p \leftarrow a.$
$a \leftarrow$ not $b.$
$b \leftarrow$ not $a.$

Mostriamo che $S_1 = \{p, a\}$ e $S_2 = \{b\}$ sono answer set di Γ.
$\Gamma^{S_1} = \{p \leftarrow a., a \leftarrow .\}$ e l'answer set di Γ^{S_1} é proprio S_1 quindi S_1 é answer set di Γ.
$\Gamma^{S_2} = \{p \leftarrow a., b \leftarrow .\}$ e l'answer set di Γ^{S_2} é proprio S_2 quindi S_2 é answer set di Γ.
Per mostrare, per esempio, che l'insieme $S = \{a, b\}$ non é un answer set di Γ computiamo Γ^S. In questo caso $\Gamma^S = \{p \leftarrow a.\}$ e il suo answer set é $\emptyset$ che é differente da S, quindi S non é un answer set di Γ.

Esempio 11.
Prendiamo di nuovo il programma Γ_1 :

$a \leftarrow$ not $b.$
$b \leftarrow$ not $a.$

Γ_1 ha 2 answer set: $\{a\}$ e $\{b\}$. Supponiamo ora di aggiungere a Γ_1 il seguente vincolo v:

$\leftarrow$ not $a, c.$

Dal momento che c risulta sempre falso in Γ_1, il vincolo sará sempre soddisfatto e non porterá all'invalidazione di nessun answer set.
Sia invece $\Gamma_2 = \Gamma_1 \cup \{c\}$; l'applicazione del vincolo v a π_2 provoca la generazione di un solo modello stabile, $\{a, c\}$, mentre scompare l'answer set $\{b\}$.
Infatti not$a$ e c non possono essere veri nello stesso answer set; ció implica che not$a$ deve risultare falso dal momento che c é un fatto in Γ_2 quindi é sempre vero, e dalla sua falsitá, per la struttura di Γ_2, discende l'impossibilitá di derivare b.
Questo esempio ci mostra anche in pratica il problema della non monotonicitá del nostro linguaggio ovvero l'aggiunta di nuove informazioni al programma ci modifica gli answer set risultanti.

1.5 Answer set in AnsProlog$^{\neg}$

Sapendo che il not non preclude la negazione del valore di un oggetto, bensí semplicemente il valore opposto, necessitiamo quindi dell'operatore $\neg$ che si comporta da complementare. Inoltre se un answer set contiene una coppia di complementari $(p, \neg p)$ ci restituisce l'intero insieme dei letterali (Lit).
Definiamo adesso gli answer set di un programma AnsProlog$^{\neg}$ Γ: per prima cosa prendiamo in esame un programma AnsProlog$^{\neg,-\text{ not}}$.
Un'interpretazione parziale di Herbrand di un programma in AnsProlog$^{\neg}$ Γ é un sottoinsieme $I \subset Lit$.
Un interpretazione parziale S di Γ soddisfa le regole 1.1.1 dell'AnsProlog$^{\neg}$ se:

- $L_0 \neq 0 : \{L_1...L_m\} \subseteq S$ e $\{L_{m+1}...L_n\} \cap S = \emptyset \Rightarrow L_0 \in S$;
- $L_0 = 0 : \{L_1...L_m\} \nsubseteq S$ e $\{L_{m+1}...L_n\} \cap S \neq \emptyset$.

Un modello parziale di Herbrand A di Γ é un'interpretazione parziale di Herbrand S di Γ che soddisfa tutte le regole in Γ e tale che se S contiene una coppia di letterali complementari allora $S = Lit$.
Quindi un answer set di un programma AnsProlog$^{\neg,-\text{ not}}$ Γ é un modello parziale di Herbrand di Γ, che é quello minimo tra i modelli parziali di Herbrand di Γ, ed inoltre questo answer set é unico (come ne caso dell'AnsProlog$^{-\text{ not}}$).

Esempio 12.
$p \leftarrow q.$
$\neg p \leftarrow r.$
$q \leftarrow .$

Come si puó vedere l'unico answer set é l'insieme $\{q, p\}$.

Passiamo ora a definire gli answer set dei programmi in AnsProlog$^{\neg}$: consideriamo Γ un programma AnsProlog$^{\neg}$ senza variabili. Per qualche insieme S di letterali, prendiamo Γ^S che risulta essere un programma in AnsProlog$^{\neg,-\text{ not}}$ ottenuto da Γ eliminando:

- le regole che hanno not L nel corpo con $L \in S$;
- tutte le formule dei tipo not L nel corpo delle restanti regole.

Come abbiamo detto in precedenza per l'AnsProlog avremo che Γ^S non contiene negazioni, quindi siamo in presenza di un programma AnsProlog$^{-\text{not}}$, e che questo answer set é unico. Se l'answer set trovato coinciderá con S potremo dire che S é l'answer set di Γ. Adesso siamo in grado di dimostrare la seguente affermazione:

Proposizione 1.
Ogni answer set di Γ *é un modello minimale di Herbrand di* Γ

Dimostrazione. Consideriamo un answer set M, facciamo vedere che M é un modello di Γ. Imponiamo che R sia una regola di Γ, se il corpo di R contiene un letterale $\neg B$ tale che $B \in M$, allora R é vera in M. Se non fosse cosí consideriamo la regola R^1 ottenuta da R cancellando i letterali negativi dal suo corpo. Poiché R^1 é una regola di Γ^M ed M é un modello minimo di Γ^M , é certo che R^1 é vera in M. D'altra parte logicamente R deriva da R^1 e di conseguenza R é vera in M. Per mostrare che M é minimo, assumiamo che esista sottoinsieme M^1 di M che é un modello di Γ; faremo vedere che M^1 é anche un modello per Γ^M. Consideriamo una regola R^1 di Γ^M ottenuta come in precedenza da R cancellando tutti i letterali negativi dal corpo e, in tutti questi letterali $\neg B$, $B \notin M$. Per mostrare che R^1 é vero in M^1 osserviamo che R é vera in M^1 (poiché M^1 é un modello di Γ), che ogni letterale negativo $\neg B$ nel corpo di R é vero in M^1 (poiché $B \notin M$ e $M \subset M^1$) e che R^1 puó essere ottenuto da R togliendo questi letterali. Poiché M é un modello minimo di Γ^M allora abbiamo che $M = M^1$. □

Esempio 13.
Consideriamo il seguente programma:

$p(1, 2).$
$q(x) \leftarrow p(x, y), \neg q(y).$

Imponiamo che Γ sia questo programma con la seconda regola sostituita da istanze ground:

$q(1) \leftarrow p(1, 1), \neg q(1)$
$q(1) \leftarrow p(1, 2), \neg q(2)$
$q(2) \leftarrow p(2, 1), \neg q(1)$
$q(2) \leftarrow p(2, 2), \neg q(2)$

Abbiamo $M = \{q(2)\}$, quindi in questo caso il nostro Γ^M sará:

$p(1, 2).$
$q(1) \leftarrow p(1, 1).$
$q(2) \leftarrow p(2, 1).$

Quindi il modello minimale di Herbrand di questo programma é $\{p(1, 2)\}$, che é differente da M quindi M non é stabile (questo risultato poteva essere previsto applicando la proposizione precedente in quanto M non é un modello di Γ). Adesso proviamo $M = \{p(1, 2), q(1)\}$, in questo caso Γ^M é:

$p(1, 2).$
$q(1) \leftarrow p(1, 2).$
$q(2) \leftarrow p(2, 2).$

Il modello minimo é $\{p(1, 2), q(1)\}$ quindi in questo caso M é stabile.

Esempio 14.
$fly(tweety) \leftarrow bird(tweety),\ \text{not}\ \neg fly(tweety).$
$\neg fly(tweety) \leftarrow penguin(tweety).$
$fly(rocky) \leftarrow bird(rocky),\ \text{not}\ \neg fly(rocky).$
$\neg fly(rocky) \leftarrow penguin(rocky).$
$bird(tweety) \leftarrow .$
$bird(rocky) \leftarrow .$
$penguin(rocky) \leftarrow .$

L'answer del programma é l'insieme $\{bird(tweety), bird(rocky), penguin(rocky), fly(tweety), \neg fly(rocky)\}$. L'importante aspetto del programma Γ é che se troviamo che tweety non vola possiamo aggiungere direttamente $\neg fly(tweety)$ al programma Γ e il programma rimane comunque consistente ma cambia l'answer set che risulta essere l'insieme $\{bird(tweety), bird(rocky), penguin(rocky), \neg fly(tweety), \neg fly(rocky)\}$.

Esempio 15.
$a \leftarrow\ \text{not}\ b.$
$b \leftarrow\ \text{not}\ a.$
$q \leftarrow a.$

$\neg q \leftarrow a.$

Mostriamo che l'unico answer set é $\{b\}$ ed invece non possiamo prendere né Lit, né $\{a, q, \neg q\}$.
$\Gamma^{\{b\}}$ é il programma $\{b \leftarrow .\}$ il cui answer set é $\{b\}$, quindi stando a quello che abbiamo detto precedentemente, $\{b\}$ é un answer set di Γ.
Adesso consideriamo $\Gamma^{\{a,q,\neg q\}}$: esso é il programma $\{a \leftarrow ., q \leftarrow a., \neg q \leftarrow a.\}$. Ma in base alla definizione di answer set, l'answer set di $\Gamma^{\{a,q,\neg q\}}$ é Lit e non $\{a, q, \neg q\}$, quindi $\{a, q, \neg q\}$ non é un answer set di Γ.
In modo simile consideriamo Γ^{Lit} che ha come programma $\{q \leftarrow a., \neg q \leftarrow a.\}$ il quale ha a sua volta come answer set $\emptyset$, quindi anche Lit non é un answer set di Γ.

Per capire invece gli answer set di un programma in AnsProlog$^{\neg}$, possiamo ridurre quest'ultimo ad un programma AnsProlog sostituendo gli $\neg L$ con letterali positivi. La forma positiva di un letterale L viene denotata con L^+ ed inoltre Γ^+ sta per un programma in AnsProlog ottenuto da Γ rimpiazzando la regola 1.1.1(con $k = 0$) con:

$$L_0{}^+ \leftarrow L_1{}^+ ... L_m{}^+, \text{not } L_{m+1}{}^+ ... L_n{}^+$$

Per qualche $S \subseteq Lit$, S^+ é l'insieme S con i letterati tutti in forma positiva. Possiamo affermare che S é un answer set di $\Gamma \Leftrightarrow S^+$ é un answer set di Γ^+.

Esempio 16.
$a \leftarrow$ not $b.$
$b \leftarrow$ not $a.$
$q \leftarrow a.$
$\neg q \leftarrow a.$

passando al Γ^+ abbiamo:

$a \leftarrow$ not $b.$
$b \leftarrow$ not $a.$
$q \leftarrow a.$
$q^1 \leftarrow a.$

É semplice osservare che Γ^+ ha 2 answer sets $S_1{}^+ = \{a, q, q^1\}$ ed $S_2{}^+ = \{b\}$.

In base a quello che abbiamo detto precedentemente S_2 é un answer set di Γ al contrario di $S_1 = \{a, q, \neg q\}$ che invece non lo soddisfa quindi non risulta essere un answer set.

Osservazione 1 (Cardinal Constraints).
Vale in generale che le regole della forma 1.1.1 devono essere scelte in modo tale che:

$$n\{p(X,Y) : q(Y)\}m \leftarrow r(X).$$

ovvero: per ogni X tale che r(X) vale, ogni answer set deve contenere da n a m letterali della forma p(X,Y), dato che vale anche q(Y). Inoltre r e q definiscono i domini delle variabili X e Y rispettivamente.
Molto importante é il caso $n = m = 1$, quindi quando é permessa una sola p(X,Y) di conseguenza é ammessa un'unica soluzione. Per esempio:

$$1\{residence(X;Y) : place(Y)\}1 \leftarrow person(X).$$

Infatti una persona puó avere la residenza in un unico posto.

Osservazione 2.
In conclusione diversamente dalla programmazione logica tradizionale, le soluzioni di un problema non sono ottenute per sostituzioni di variabili in risposta ad una query, ma vengono espresse come insiemi di risposte. Quindi il programma potrá avere un solo answer set, molteplici answer set o nessun answer set; in quest'ultimo caso tale programma verrá detto *inconsistente*.

1.6 AND/OR Handle

I modelli stabili di un generico programma logico Γ coincidono, senza perdita di generalitá, con quelli della corrispondente *forma canonica* in cui in seguito identifichiamo i cicli contenuti nel programma e mostriamo che i modelli stabili dell'intero programma coincidono con la composizione dei modelli stabili di ogni singolo sotto-programma formato da tali cicli.
I cicli possono essere di due tipi:

- *ciclo pari*: dipendenze cicliche che coinvolgono un numero pari di atomi; ogni ciclo pari ha due answer set, uno costituito dagli atomi "pari"

del ciclo, l'altro da quelli "dispari"; inoltre, se abbiamo diversi cicli pari, i loro answer set si combinano.
Prendiamo il seguente esempio:

$a \leftarrow$ not b.
$b \leftarrow$ not a.
$e \leftarrow$ not f.
$f \leftarrow$ not g.
$g \leftarrow$ not r.
$r \leftarrow$ not e.

Quindi per quello che abbiamo detto in precedenza i nostri answer set sono : $\{a, e, g\}$, $\{b, e, g\}$, $\{a, f, r\}$ e $\{b, f, r\}$ (considerando sempre il metodo risolutivo spiegato nella sezione 1.4).

- *ciclo dispari*: dipendenze cicliche che coinvolgono un numero dispari di atomi; i programmi con cicli dispari possono avere answer set solo se sono presenti cicli pari e connessioni tra tali cicli pari e quelli dispari che rimuovono le contraddizioni causate dalle negazioni.
  Prendiamo il seguente esempio:

  $a \leftarrow$ not b.
  $b \leftarrow$ not c.
  $c \leftarrow$ not a.

  Vediamo che utilizzando il metodo della sezione 1.4 non riusciamo a trovare nessun answer set: infatti prendendo come answer set $S = \{b, c\}$, abbiamo $\Gamma^S = \{c \leftarrow .\}$ che ha invece come answer set $S^1 = \{c\}$, allora $S \neq S^1$ quindi S non é un answer set del nostro programma Γ. Questo succede anche con $\{a, b\}$ e $\{a, c\}$.

Le connessioni di cui stavamo parlando nel punto precedente prendono il nome di **Handle** ed hanno la proprietá di eliminare le inconsistenze nei programmi. Abbiamo due diversi tipi di handle:

- *OR Handle*: consiste nell'inserire, all'interno del programma, una regola alternativa che abbia nella testa un atomo presente nel ciclo dispari ed invece nel corpo un atomo coinvolto nel ciclo pari. Vediamo

il seguente esempio:

$a \leftarrow$ not b.
$b \leftarrow$ not a.
$q \leftarrow$ not q.

L'ultima regola ci dà una contraddizione, in quanto un atomo non puó essere allo stesso tempo vero e falso, in quantoció ci rende il programma Γ inconsistente, quindi per evitare questo aggiungiamo al programma stesso la regola:

$q \leftarrow b$. (la nostra OR Handle)

Possiamo verificare che $S = \{b, q\}$ é un answer set: infatti $\Gamma^S = \{b \leftarrow .,\ q \leftarrow b.\}$, l'answer set di Γ^S é proprio S quindi S é answer set di Γ. Vincolo molto importante per evitare l'inconsistenza del programma é che il corpo della regola aggiunta sia vero, forzando cosí anche l'atomo della testa ad essere vero.

- *AND Handle*: consiste nell'aggiungere un atomo, che faccia parte di un ciclo pari, ad una delle regole che intervengono nel ciclo dispari. Vediamo il seguente esempio:

  $a \leftarrow$ not b.
  $b \leftarrow$ not a.
  $q \leftarrow$ not q.

  Stavolta aggiungiamo l'atomo a (AND Handle) all'ultima regola ed abbiamo:

  $a \leftarrow$ not b.
  $b \leftarrow$ not a.
  $q \leftarrow$ not q, a.

  Quindi l'answer set é $S = \{b\}$ infatti: $\Gamma^S = \{p \leftarrow a.,\ b \leftarrow .\}$ ha come answer set proprio S.
  In questo caso il vincolo da soddisfare é che l'atomo aggiunto L (che

nel caso precedente é a) deve essere falso per aprire il ciclo dispari rendendo falso uno dei suoi atomi.

Prendiamo il seguente programma, Γ, e vediamo come utilizzare entrambi i tipi di Handle:

$p \leftarrow$ not p.
$a \leftarrow$ not b.
$b \leftarrow$ not a.
$q \leftarrow$ not q.
$e \leftarrow$ not f.
$f \leftarrow$ not e.

Aggiungiamo come AND Handle l'atomo $\{$not $b\}$ per disambiguare la prima regola ed inoltre aggiungiamo $\{q \leftarrow e\}$, quindi il nostro programma diventa:

$p \leftarrow$ not p, not b.
$a \leftarrow$ not b.
$b \leftarrow$ not a.
$q \leftarrow$ not q.
$q \leftarrow e$.
$e \leftarrow$ not f.
$f \leftarrow$ not e.

Verifichiamo che $S = \{b, e, q\}$ é un aswer set: utilizzando sempre le regole della sezione 1.4 abbiamo che $\Gamma^S = \{b \leftarrow .,\ q \leftarrow\ e., e \leftarrow .\}$ ha come answer set proprio S e quindi S é answer set di Γ.
Quindi possiamo dire che: *le condizioni necessarie e sufficienti per l'esistenza di modelli stabili possono essere ottenute prendendo la composizione di cicli, trovandone i modelli stabili, e verificando che le Handle non si contraddicano a vicenda.*
Per capire il significato dell'ultima affermazione consideriamo il seguente esempio:

$p \leftarrow$ not p, not a.
$q \leftarrow$ not q, not b.
$a \leftarrow$ not b.

$b \leftarrow$ not a.

Non abbiamo le AND Handle che ci aprono i due cicli dispari in quanto dal ciclo pari abbiamo come anwer set $\{a\}$ e $\{b\}$: nessuno dei due riesce a soddisfare entrambe le OR Handle e quindi non possiamo trovare l'answer set.
Riprendendo il discorso iniziale dobbiamo spiegare come si arriva alla forma canonica di un programma, quindi diamo la seguente definizione:

Definizione 1.6.1. La forma canonica é costruita mediante la semantica del *well-founded model* (partizionare, in base alle regole presenti nel programma, gli atomi ground in tre set: true, false e unknown) e soddisfa le seguenti condizioni sintattiche:

- ogni atomo di Γ é presente sia nella testa che nel corpo di qualche regola;
- ogni atomo in Γ é coinvolto in qualche ciclo;
- ogni regola di Γ é coinvolta in un ciclo oppure in una regola aggiuntiva (OR Handle);
- ogni Handle di un ciclo é costituita da un unico letterale, a o not a, dove l'atomo a non é coinvolto nel medesimo ciclo.

Da questa definizione deduciamo che un programma Γ che é in forma canonica ha le seguenti proprietá: il corpo di ogni regola, coinvolta in un ciclo, é formato da massimo due atomi, invece il corpo di una regola aggiuntiva (OR Handle) é formato esclusivamente da un unico atomo.
Concludiamo quindi elencando i passi da fare per trovare gli answer set nel caso di un programma in forma canonica (presenza di cicli):

- cercare gli answer set della parte aciclica del programma;
- cercare gli answer set dei cicli pari;
- combinare gli answer set dei punti precedenti;
- estendere tali answer set alla parte aciclica intermedia;
- selezionare fra gli answer set risultanti quelli che forniscono Handle ai cicli dispari;

- considerare la parte aciclica rimanente.

Ovviamente non in tutti i programmi si presentano tali situazioni, infatti potrebbero mancare la parte aciclica intermedia, o quella finale o anche degli answer set dei cicli pari che riescano ad essere utilizzati come Handle di quelli dispari, quindi in quest'ultimo caso non riusciremo a trovare gli answer set cercati.

1.6.1 Grafi ciclici

Avendo un programma in forma canonica possiamo costruire un grafo i cui nodi sono i cicli presenti nel programma e i collegamenti sono le Handle. Una Handle é considerata come la connessione tra il ciclo da cui proviene e il ciclo dove la Handle appare, inoltre se queste Handle formano un percorso (chiamato *Handle Paths*) che riesce ad unire tutti i cicli vuol dire che tali Handle non si contraddicono a vicenda e che quindi siamo in grado di trovare l'answer set cercato.

Esempio 17.
Analizziamo il seguente programma Γ:

$p \leftarrow \text{ not } s.$
$q \leftarrow \text{ not } t.$
$t \leftarrow \text{ not } p.$
$a \leftarrow \text{ not } c.$
$c \leftarrow \text{ not } a.$

Per aprire il ciclo inseriamo l'atomo $\{\text{not } c\}$ come AND Handle alla prima riga, quindi avremo:

$p \leftarrow \text{ not } s, \text{ not } c.$
$q \leftarrow \text{ not } t.$
$t \leftarrow \text{ not } p.$
$a \leftarrow \text{ not } c.$
$c \leftarrow \text{ not } a.$

che ha come answer set $S = \{c, t\}$ infatti: $\Gamma^S = \{t \leftarrow \ ., c \leftarrow \ .\}$ che ha

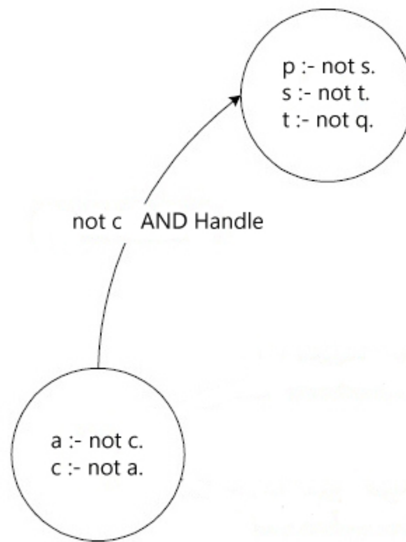

Figura 1.1: grafo

come aswer set proprio S quindi S é answer set di Γ.
In figura 1.1 vediamo la stessa situazione rappresentata con il grafo.

1.7 Risolutori: SMODELS

Il calcolo dei modelli stabili é oggi un'area di fervida ricerca; sono attualmente disponibili implementazioni piuttosto efficienti che sono in grado di risolvere la versione esatta del problema calcolando tutti i modelli stabili del programma dato: sono cioé orientate all'esplorazione dell'intero spazio delle soluzioni (albero di decisione), anche se é spesso sufficiente la verifica dell'esistenza di un modello.
Uno dei piú noti per l'Answer set Programming é *SMODELS*, costituito da **smodels**, cioé l'implementazione vera e propria della semantica dei modelli stabili per programmi logici normali, e **lparse**, un front-end che ha il compito di trasformare i programmi in una forma comprensibile per **smodels**.

1.7.1 Lparse

I sistemi per la programmazione logica tradizionale sono query-driven quindi l'utente pone una domanda e il sistema tenta di darvi una risposta. In ogni istante del calcolo viene associato un valore solo alle variabili che sono

coinvolte in qualche modo nelle query. Invece nell'Answer Set Programming tutte le variabili vengono rimosse dalle regole del programma sostituendovi valori costanti ed é proprio questo il compito di **lparse** (quindi viene utilizzato come *grounder*). L'output prodotto da lparse codifica, tramite interi, le regole e gli atomi del programma: é su questo formato che il motore di inferenza di *Smodels* lavora.

1.7.2 Smodels

Smodels é il motore di programmazione logica che si occupa di calcolare i modelli stabili di un programma e la sua funzione principale si chiama proprio *smodels*; questa funzione prende in ingresso un programma ground Γ e un insieme di letterati L e restituisce *true* se esiste un answer set di Γ che sia consistente con L, oppure *false* se tale answer set non esiste.
Smodels viene invocato nel Prompt dei comandi con la seguente formula:

$$lparse \text{ nomedoc.txt } | \; smodels \text{ } 0$$

Dove *nomedoc* deve essere sostituito dal nome del documento che contiene il programma, invece lo "0" é un tipo di opzione (in base alle esigenze lo "0" puó essere sostituito da altri numeri). Infatti se utilizziamo l'opzione "0" verranno stampati tutti i modelli stabili del programma e, immediatamente dopo, la parola *False* serve ad indicare che non vi sono altri modelli oltre a quelli visualizzati; se invece l'utente richiede un numero preciso di modelli (quindi utilizziamo numeri diversi da 0), viene stampata la parola *True* che sta a significare che potrebbero esistere altri modelli.
L'output prodotto da **Smodels** fornisce informazioni sul processo di calcolo degli answer set.
Vediamone un esempio:

```
smodels version 2.27.  Reading...done
Answer set:  1
...
True
Duration:  1800
Number of choice points:  7
Number of wrong choices:  0
```

```
Number of atoms:  610
Number of picked atoms:  1952
Number of forced atoms:  76
Number of truth assignment:  94039
Size of searchspace (removed):  240 (0)
```

La prima linea di output riporta la versione del risolutore, mentre la linea successiva contiene il primo modello calcolato. Il *True*, come abbiamo detto in precedenza, sta ad indicare che tutti i modelli stabili possibili sono stati elencati; viene poi indicata la durata del processo di ricerca. Il numero di *choice points* indica quante volte il sistema ha dovuto "indovinare" il valore di veritá per un atomo ground; il numero di *wrong choices* indica invece quante volte tale scelta si é rivelata scorretta, causando il ricorso al *backtracking* che consiste nel tornare indietro per modificare quelle scelte che hanno portato ad un fallimento. Il numero di choice point puó essere utilizzato come misura per la complessitá[6] di un problema. Le due righe successive danno informazioni sulla dimensione del programma in input (numero di atomi e di regole). Il numero di *picked atoms* indica quante volte Smodels é stato in grado di assegnare un valore di veritá ad un atomo; il numero di *forced atoms* rappresenta il numero di atomi aggiunti al modello in quanto la loro negazione avrebbe causato una contraddizione; il numero di *truth assignment* indica quante volte Smodels ha assegnato un valore di veritá ad un atomo. In ultimo la dimensione dello spazio di ricerca rappresenta il numero massimo di scelte necessario per essere certi che un modello esista (oppure non esista).

[6]La complessitá indica la difficoltá di risoluzione di un problema fornendoci le risorse minime necessarie per tale risoluzione; inoltre é valutata tramite lo spazio di memoria e il tempo di calcolo.