Intelligenza Artificiale e Laboratorio

Answer Set Programming

Alberto Martelli

Answer set programming (ASP)

I modelli stabili, detti anche **Answer Set**, sono stati introdotti inizialmente per dare una semantica alla negazione per fallimento adottata dagli interpreti Prolog, anche se l'interprete del Prolog non calcola esattamente i modelli stabili.

Più di recente sono diventati il fondamento di una nuovo tecnica di programmazione basata sempre su clausole di Horn (**Answer set programming**) in cui l'inferenza non si basa, come nel Prolog, su backward chaining, ma sulla *costruzione di modelli stabili*.

Questo paradigma di programmazione di è dimostrato particolarmente utile per risolvere problemi combinatori (es. problemi di vincoli, planning) facendo uso di *answer set solvers* notevolmente efficienti: DLV, Smodels, Clingo.

ASP

La tecnica di Answer Set Programming si applica solo a programmi logici *proposizionali*.

La maggior parte dei tool per ASP consente per comodità di usare variabili, ma le clausole devono poter essere trasformate in un numero finito di clausole ground.

Ad esempio in un problema di planning posso usare il termine pickup(X), che verrà istanziato a pickup(a) e pickup(b), se a e b sono gli unici blocchi esistenti.

ASP

Ad esempio, per il seguente programma

```
vola(X):- uccello(X), not abnormal(X).
uccello(X):- pinguino(X).
uccello(tweety).
abnormal(X):- pinguino(X).
```

ottengo come risultato un unico answer set

```
{vola(tweety), uccello(tweety)}
```

Con il programma

```
p:- not q.
q:- not p.
```

si ottengono due answer set

```
{p}   e   {q}
```

Programmi ASP

I programmi ASP sono **insiemi di regole di logic programming**.

E' possibile anche usare regole senza testa (*integrity constraint*):

:- A1, A2, …, An

dal significato: è inconsistente che A1 e A2 e… An siano tutti veri. E' come se nella testa della regola ci fosse *falso*.
Sono usate come test per scartare answer set non desiderati.

Normalmente le regole devono essere **safe**, ossia tutte le variabili che compaiono in una regola devono comparire in un letterale positivo (non preceduto da *not*) del corpo.

ASP con negazione classica

Normalmente le regole usate in ASP sono estese rispetto a quelle del Prolog con la cosiddetta **negazione classica**:

attraversa :- ¬treno

"Attraversa se non arriva il treno" ha un significato diverso da

attraversa :- **not** treno

Nel secondo caso si può attraversare in assenza di informazione esplicita sul treno in arrivo. La prima regola è più forte: si attraversa solo se si può derivare che il treno non è in arrivo.

In realtà, un letterale negato ¬p non ha nessuna proprietà particolare. Viene considerato come se fosse un nuovo atomo positivo, aggiungendo il vincolo :- p, ¬p.

Esempio negazione classica

In ASP si può scrivere

```
pacifista(X):- quacchero(X), not -pacifista(X).
-pacifista(X):- repubblicano(X), not pacifista(X).
repubblicano(nixon).
quacchero(nixon).
```

Si ottengono due *answer set*: uno contiene pacifista(nixon) e l'altro ¬pacifista(nixon).

ASP con disgiunzione

In alcuni linguaggi per ASP si possono anche avere delle regole con la disgiunzione nella testa.

```
colored(X,r) v colored(X,g) v colored(X,b) :- node(X).
```

vincolo ⟶ `:- edge(X,Y), colored(X,C), colored(Y,C).`

```
node(a).
node(b).
node(c).
edge(a,b).
edge(b,c).
edge(c,a).
```

Si ottengono 6 answer set con le diverse combinazioni di colori

```
{node(b), node(a), node(c), edge(b,c), edge(a,b), edge(c,a),
colored(b,r), colored(a,g), colored(c,b)}
ecc.
```

ASP con disgiunzione

Se le regole consentono l'uso della disgiunzione, la definizione di Answer set deve essere opportunamente modificata.

Il punto principale è che gli answer set sono insiemi minimi, e quindi ogni answer set conterrà solo un disgiunto.

Se un programma contiene il fatto

a v b.

ci sono due answer set: uno contiene a e uno b.

Calcolare un answer set

Un *answer set solver* calcola un answer set (modello stabile).

Come funziona?

Per prima cosa, se il programma permette l'uso di variabili, il *solver* lo deve modificare sostituendo le regole che contengono variabili con tutte le possibili *istanziazioni* (attenzione che un programma istanziato potrebbe diventare infinito).

Successivamente si passa alla costruzione degli answer set. In questa fase si utilizzano *tecniche simili* a quelle utilizzate in logica proposizionale per verificare la ***soddifacibilità*** di un insieme di formule.

Nelle prossime tre slide diamo un'idea di come si può ottenere un modello per la logica proposizionale classica (soddisfacibilità).

Inferenza proposizionale efficiente

(Breve digressione sulla logica proposizionale)

Per verificare la **soddisfacibilità** di una formula, occorre trovare un **modello**.

Per trovare un modello in logica proposizionale si può costruire una tabella che enumera tutti i modelli (sono in numero finito) e poi valutare la formula in tutti i modelli finché se ne trova uno in cui è vera.

L'**algoritmo di Davis, Putnam, Logemann, Loveland** (DPLL 1962)

applica alcuni miglioramenti alla semplice procedura di enumerare tutti i modelli, ed è ancora alla base di molti strumenti per determinare la soddisfacibilità (**SAT**).

The DPLL algorithm

Determine if an input propositional logic sentence (in CNF) is satisfiable.

Improvements over truth table enumeration:

1. Early termination
   - A clause is true if any literal is true.
   - A sentence is false if any clause is false.

2. Pure symbol heuristic
   - Pure symbol: always appears with the same "sign" in all clauses.
   - e.g., In the three clauses $(A \vee \neg B)$, $(\neg B \vee \neg C)$, $(C \vee A)$, A and B are pure, C is impure.
   - Make a pure symbol literal true.

3. Unit clause heuristic
   - Unit clause: only one literal in the clause
   - The only literal in a unit clause must be true.

The DPLL algorithm

function DPLL-SATISFIABLE?(s) **returns** *true* or *false*

inputs: s, a sentence in propositional logic

clauses $\leftarrow$ the set of clauses in the CNF representation of s

symbols $\leftarrow$ a list of the proposition symbols in s

return DPLL(*clauses*, *symbols*, [])

---

function DPLL(*clauses*, *symbols*, *model*) **returns** *true* or *false*

if every clause in *clauses* is true in *model* **then return** *true*

if some clause in *clauses* is false in *model* **then return** *false*

P, *value* $\leftarrow$ FIND-PURE-SYMBOL(*symbols*, *clauses*, *model*)

if P is non-null **then return** DPLL(*clauses*, *symbols*–P, $[P = value | model]$)

P, *value* $\leftarrow$ FIND-UNIT-CLAUSE(*clauses*, *model*)

if P is non-null **then return** DPLL(*clauses*, *symbols*–P, $[P = value | model]$)

$P \leftarrow$ FIRST(*symbols*); *rest* $\leftarrow$ REST(*symbols*)

return DPLL(*clauses*, *rest*, $[P = true | model]$) **or**
DPLL(*clauses*, *rest*, $[P = false | model]$)

CLINGO

Gli esercizi ASP proposti nel corso possono essere svolti con **Clingo**, disponibile al sito http://potassco.sourceforge.net/.

Il sito fornisce diversi tool, fra cui:

Gringo: trasforma tutti i termini in termini ground

Clasp: un Answer Set Solver

Clingo: combina i due precedenti

Iclingo: versione incrementale di Clingo.

Le caratteristiche di questi tool sono descritte in dettaglio nella Guida Utente fornita negli allegati.

Uso di clingo

Clingo viene invocato da linea di comando dando i nomi dei file che contengono il programma ed eventuali opzioni descritte nella guida.
Ad esempio:

```
clingo bird.cl
```

dà come risultato:

```
Answer: 1
bird(tux) penguin(tux) bird(tweety) -flies(tux) flies(tweety)
SATISFIABLE

Models       : 1
Time         : 0.000
  Prepare    : 0.000
  Prepro.    : 0.000
  Solving    : 0.000
```

Caratteristiche di clingo

Oltre alle normali regole, clingo fornisce diversi costrutti che danno la possibilità di definire **aggregati** (operazioni su multiinsiemi) e di trovare **soluzioni ottime**.

Gli aggregati consentono di esprimere vincoli, come ad esempio:

```
3 { scelto(C) : course(C,_) } 6.
```

Dati dei fatti `corso(prog,9), corso(logica,6),` ecc., che descrivono dei corsi con nome e numero CFU, la soluzione deve contenere da 3 a 6 corsi scelti.

Problemi di vincoli

ASP è particolarmente utile per risolvere problemi di vincoli.
Ad esempio, il problema della **colorazione dei nodi di un grafo** può essere formulato con le seguenti due regole:

```
#const n = 3.

1{ color(X,1..n) }1 :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
```

La prima regola stabilisce che ogni nodo ha esattamente un colore.
La seconda crea il vincolo che i nodi collegati da un arco devono avere colori diversi.

Ogni answer set calcolato per un particolare grafo costituisce una soluzione del problema.

Criptoaritmatica

Il file *cripto.cl* contiene il programma per risolvere il problema TWO + TWO = FOUR dal Russell e Norvig, usando i riporti per ogni colonna della somma.
Ad esempio

```
1{val(C,0..9)}1:- cifra(C).
1{val(X,0..1)}1:- rip(X).
:- val(C1,V), val(C2,V), cifra(C1), cifra(C2), C1!=C2.
```

stabiliscono che ogni lettera e riporto ha esattamente un valore e che le lettere hanno valori diversi fra loro, mentre il vincolo

```
:- val(o,Vo), val(r,Vr), val(x1,Vx1), Vo+Vo != Vr+10*Vx1.
```

definisce la somma sulla prima colonna di destra.

N regine

Il problema delle n regine può essere formulato come segue (v. *regine.cl*):

```
% guess horizontal position for each row

1 {q(R,C): col(C)} 1:- row(R).


% assert that each column may only contain (at most) one queen
:- q(R1,C), q(R2,C), R1 != R2.

% assert that no two queens are in a diagonal from top left to
bottom right
:- q(R1,C1), q(R2,C2), R2=R1+N, C2=C1+N, diag(N).

% assert that no two queens are in a diagonal from top right to
bottom left
:- q(R1,C1), q(R2,C2), R2=R1+N, C2=C1-N, diag(N).
```

Ottimizzazione

Un esempio di problema di ottimizzazione è dato dal **minimum spanning tree**.
Assumiamo per semplicità che il grafo sia orientato e aciclico e che sia nota la radice dell'albero.
Il programma è il seguente, dove **in_tree** rappresenta gli archi che costituiscono lo spanning tree:

```
% un solo arco entrante per ogni nodo, tranne la radice
1{in_tree(X,Y,C) : edge(X,Y,C)}1:- node(Y), not root(Y).

% minimizare la somma dei costi
#minimize [in_tree(X,Y,C) = C].
```