Il motore di inferenza del Prolog potrebbe non trovare soluzioni

```
/* file: simpleProblem3.pl */
/* Initial Situation   a
                       b c */
ontable(b,s0). ontable(c,s0). on(a,b,s0). clear(a,s0). clear(c,s0).

/* Goal                a
                       b
                       c */
goal(S) :- ontable(c,S), on(a,b,S), on(b,c,S), clear(a,S).
```

È necessario controllare la ricerca del piano

Pianificatori scritti in GOLOG: ricerca in profondità limitata

(simplePlanners.pl)

```
:- [golog_swi].
:- discontiguous proc/2.

%  Implementation of a simple depth-first planner.
% plandf(N): ricerca un piano di lunghezza massima N
%            la ricerca e' in profondita'
plandf(N) :- do(depthFirstPlanner(N),s0,_), askForMore.
askForMore :- write('More? '), read(n).

% procedura per la ricerca in profondita' limitata
proc(depthFirstPlanner(N),
     ?(goal) : ?(prettyPrintSituation) #
     ?(N > 0) : pi(a,?(primitive_action(a)) : a) :
         pi(n1, ?(n1 is N - 1) : depthFirstPlanner(n1))).
```

Predicati ausiliari

```
prettyPrintSituation(S) :- makeActionList(S,Alist), nl,
                           write(Alist), nl.

makeActionList(s0,[]).
makeActionList(do(A,S), L) :- makeActionList(S,L1),
                              append(L1, [A], L).

restoreSitArg(prettyPrintSituation,S,prettyPrintSituation(S)).
restoreSitArg(goal,S,goal(S)).
```

Esempi di esecuzione

```
?- plandf(3).

[moveToTable(a), move(b, c), move(a, b)]
More? n.


?- plandf(6).

[move(a, c), move(b, a), moveToTable(b), moveToTable(a),
      move(b, c), move(a, b)]
More? y.

[move(a, c), move(a, b), move(a, c), moveToTable(a), move(b, c), move(a, b)]
More? n.
```

Pianificatori scritti in GOLOG: ricerca in ampiezza

```
%   Implementation of World's Simplest Breadth-First Planner

planbf(N) :- do(breadthFirstPlanner(0,N),s0,_), askForMore.

% ricerca di un piano di lunghezza massima M
% se M non supera N
proc(breadthFirstPlanner(M,N),
     ?(M =< N) :
     (actionSequence(M) : ?(goal) : ?(prettyPrintSituation) #
      pi(m1, ?(m1 is M + 1) : ?(reportLevel(m1))
             : breadthFirstPlanner(m1,N)))).

% esecuzione di una qualunque sequenza di azioni di lunghezza N:
% do(actionSequence(N),S,S') = S' e' il risultato dell'esecuzione
% a partire da S di una qualunque sequenza di azioni di lunghezza N
proc(actionSequence(N),
     ?(N = 0) #
     ?(N > 0) : pi(a,?(primitive_action(a)) : a) :
       pi(n1, ?(n1 is N - 1) : actionSequence(n1))).

reportLevel(N) :- write('Starting level '), write(N), nl.
```

Esempi di esecuzione

```
?- do(actionSequence(2),s0,S).
S = do(move(b, a), do(move(a, c), s0)) ;
S = do(move(a, b), do(move(a, c), s0))


?- planbf(10).
Starting level 1
Starting level 2
Starting level 3

[moveToTable(a), move(b, c), move(a, b)]
More? y.
Starting level 4

[move(a, c), moveToTable(a), move(b, c), move(a, b)]
More? n.
```

Conoscenza di controllo

```
?- plandf(10).

[move(a, c), move(b, a), moveToTable(b), move(a, b), move(c, a),
  moveToTable(c), move(a, c), moveToTable(a), move(b, c), move(a, b)]
```

La qualità dei piani trovati dal pianificatore in profondità è molto scadente.

Per problemi piu' complessi i tempi di esecuzione di una ricerca in avanti diventano presto molto elevati.

Ad esempio, il pianificatore in profondità impiega più di 300 secondi per risolvere un problema con 6 blocchi (invocato con limite 12).

Approcci logici alla pianificazione: è possibile sfruttare il **potere espressivo del linguaggio** per fornire una guida al pianificatore:

- conoscenza specifica sul dominio;
- conoscenza di controllo (o conoscenza *euristica*): informazioni che riducono il numero di piani accettabili, dunque lo spazio di ricerca del piano.

Conoscenza di controllo per i pianificatori GOLOG: bad situations

Se durante la ricerca del piano si genera una "cattiva situazione", si deve abbandonare quella strada.

```
% procedura per la ricerca in profondita' limitata
% che abbandona la ricerca quando si genera una bad situation
proc(depthFirstPlanner(N),
     ?(goal) : ?(prettyPrintSituation) #
     ?(N > 0) : pi(a,?(primitive_action(a)) : a) :
         ?(-badSituation) :
         pi(n1, ?(n1 is N - 1) : depthFirstPlanner(n1))).

% per la ricerca in ampiezza:
% esecuzione di una qualunque sequenza di azioni di lunghezza N
% che non risulti in una ``bad situation''
proc(actionSequence(N),
    ?(N = 0) #
    ?(N > 0) : pi(a,?(primitive_action(a)) : a) :
      ?(-badSituation) :
      pi(n1, ?(n1 is N - 1) : actionSequence(n1))).

restoreSitArg(badSituation,S,badSituation(S)).
```

La definizione delle cattive situazioni è dipendente dal dominio.

A volte è necessaria anche conoscenza specifica sul problema particolare

Esempio nel mondo dei blocchi: per ogni problema, si definiscono le "buone torri": quelle che non dovranno mai essere smontate per raggiungere il goal.

Ad esempio, se l'obiettivo è quello di costruire la torre con `a` sopra `b`, `b` sopra `c` e `c` sul tavolo:

```
goal(S) :- ontable(c,S), on(a,b,S), on(b,c,S), clear(a,S).
```

```
%%% goodTower(X,S) = nella situazione S c'e' una buona torre
%                    con X in cima
goodTower(a,S) :- on(a,b,S), goodTower(b,S).
goodTower(b,S) :- on(b,c,S), goodTower(c,S).
goodTower(c,S) :- ontable(c,S).
```

Definizione di bad situations nel mondo dei blocchi

- Se si sposta un blocco sopra un altro creando una torre che non è buona, allora si ottiene una cattiva situazione

```
%  Don't create a bad tower by a move action.
badSituation(do(move(X,Y),S)) :- not(goodTower(X,do(move(X,Y),S))).
```

- Se si distrugge una buona torre, allora si ottiene una cattiva situazione

```
%  Don't move anything from a good tower to the table.
badSituation(do(moveToTable(X),S)) :- goodTower(X,S).
```

- Se si ha l'opportunità di creare una buona torre, e invece si costruisce una torre che non è buona, allora si ottiene una cattiva situazione

```
/*  Opportunistic rule: If an action can create a good tower, don't
    do a bad-tower-creating moveToTable action.  */

badSituation(do(moveToTable(X),S)) :-
                                 not(goodTower(X,do(moveToTable(X),S))),
                                 existsActionThatCreatesGoodTower(S).

existsActionThatCreatesGoodTower(S) :-
                           (A = move(Y,_) ; A = moveToTable(Y)),
                           poss(A,S), goodTower(Y,do(A,S)).
```

Tempi di esecuzione dei pianificatori GOLOG con conoscenza euristica per alcune istanze del mondo dei blocchi

su un PC con P4, 3.00GHz e 1GB RAM, sotto Linux

nome del problema	numero di blocchi	GOLOG			
		depth first		breadth first	
		tempo	lunghezza del piano	tempo	lunghezza del piano
Prob4-1	4	0.07	5	0.08	5
Prob6-1	6	0.07	5	0.08	5
Prob8-1	8	0.07	10	0.23	10
Prob10-1	10	0.08	17	11.66	16
Prob12-1	12	0.11	21	23.39	17
Prob14-1	14	0.13	22	–	–
Prob16-1	16	0.22	28	–	–
Prob18-1	18	0.39	32	–	–

I pianificatori sono stati sempre richiamati con un limite di profondità sufficiente a trovare una soluzione

Il tempo concesso è stato limitato a 300 secondi

Lunghezza del piano = numero di azioni

Problemi dei pianificatori GOLOG

- I pianificatori non accettano un vero e proprio linguaggio di pianificazione (tipo STRIPS): la specifica di un problema è codice Prolog.

  La formulazione di conoscenza di controllo in termini di "bad situations" è spesso piuttosto complicata.

  Di conseguenza scrivere (e fare debugging di) un dominio di pianificazione richiede una buona conoscenza del Prolog.

- Il pianificatore in ampiezza genera piani ottimi (di lunghezza minima), ma è piuttosto inefficiente relativamente ai tempi di esecuzione.

- Il pianificatore in profondità, quando gli è fornita una buona conoscenza di controllo, è piuttosto efficiente, sia rispetto al tempo di esecuzione che alla qualità dei piani.

  Ma deve essere necessariamente invocato con un limite massimo di profondità.

  Se il limite non è sufficiente a trovare una soluzione, può impiegare molto tempo per scoprirlo.

I file disponibili in rete

Dalla pagina del corso

`http://cialdea.dia.uniroma3.it/teaching/lsi/slides/planning/golog/`

si possono scaricare:

- `golog_swi.pl`: interprete GOLOG in SWI-Prolog
- `simpleElevator.pl` e `moveObjects.pl`: i due esempi di programmi GOLOG per il controllo di un'ascensore e di un agente che deve spostare oggetti;
- `simplePlanners.pl`: versione semplificata dei pianificatori GOLOG, che utilizzano conoscenza di controllo;
- `simplePlanners_noControl.pl`: versione semplificata dei pianificatori GOLOG, che non utilizzano conoscenza di controllo;
- `simpleProblem*.pl`: esempi semplici di problemi di pianificazione nel mondo dei blocchi (ciascun file include la definizione del dominio e del problema);
- `planners.pl`: versione più elaborata dei pianificatori (breadth-first e depth-first) scritti in GOLOG, che danno anche informazione sui tempi di esecuzione (versione in SWI-Prolog dei pianificatori disponibili in rete, scritti in Eclipse Prolog);
- `blocks_domain.pl`: definizione del dominio nel mondo dei blocchi;

  `blocksWorldInstance*.pl`: istanze di problemi nel mondo dei blocchi (il numero nel nome del file indica il numero di blocchi).

Esercizi

Considerare i seguenti domini di pianificazione e rappresentarli in Prolog in modo che siano utilizzabili dai pianificatori GOLOG. Per ciascun dominio, definire anche un'opportuna conoscenza di controllo che aiuti il pianficatore. Nella descrizione che segue, per ciascun dominio è definita un'istanza di problema; definirne comunque anche altre e provare il comportamento dei pianificatori sulle diverse istanze.

1. **Dominio:** una scimmia è in una stanza, dove delle banane sono appese al soffitto, fuori della sua portata. Nella stanza c'è anche un panchetto, sul quale la scimmia potrebbe salire per raggiungere le banane. Le azioni possibili per la scimmia sono: andare da un posto all'altro nella stanza, spingere un oggetto da un posto all'altro, salire su un oggetto e afferrare un oggetto.

   Problema: le posizioni della stanza sono A, B, C e D; inizialmente, la scimmia è in A, le banane sono appese in B e il panchetto è in C. Gli altri oggetti presenti nella stanza sono una palla, inizialmente in D, e un libro, inizialmente in B. L'obiettivo è quello di avere le banane.

2. **Dominio:** Un robot con un solo braccio agisce in un ambiente costituito da una sola stanza e un corridoio. Le azioni che può compiere sono entrare e uscire dalla stanza, aprire borse, chiudere borse, mettere oggetti piccoli dentro borse (solo se queste sono aperte), prendere in mano borse (solo se queste sono chiuse).

   Problema: nello stato iniziale il robot è fuori della stanza. Nella stanza ci sono un tavolo (grande), una borsa chiusa e un libro (piccolo) appoggiato sul tavolo. L'obiettivo del robot è quello di trovarsi fuori della stanza, con il libro dentro la borsa.

3. **Dominio:** un robot si può muovere tra diversi posti e comprare oggetti. Per comprare un oggetto deve essere in un posto in cui si vende tale oggetto. Dopo aver comprato un oggetto, il robot lo possiede.

   Problema: i posti dell'ambiente sono *casa*, *super* (negozio di alimentari) e *hs* (hardware store, che vende attrezzi da bricolage). Inizialmente il robot è a casa e non possiede alcun oggetto. L'obiettivo è quello di essere a casa con un litro di latte, una banana e un trapano.

4. **Dominio (Gripper):** Un robot con due braccia agisce in un ambiente costituito da due stanze, A e B. Ogni braccio del robot può tenere un solo oggetto alla volta. Le azioni che il robot può compiere sono "prendere la palla X con la mano Y nella stanza Z", "lasciare la palla X tenuta con la mano Y nella stanza X", andare dalla stanza X alla stanza Y".

   Problema: Inizialmente la stanza A contiene 4 palle. L'obiettivo è quello di avere le 4 palle nella stanza B.

5. **Dominio (Briefcase):** Un robot agisce in un ambiente costituito da un certo numero di locazioni, e dispone di una borsa che può trasportare oggetti. Si assume che la borsa sia di capienza illimitata. Le azioni che può compiere sono andare da una locazione all'altra, mettere un oggetto nella borsa, togliere un oggetto dalla borsa.

   Problema: nell'ambiente ci sono 4 locazioni (*casa*, A. B, C) e 3 oggetti. Inizialmente il robot è a *casa*, l'oggetto 1 è in A, l'oggetto 2 è in B e l'oggetto 3 è in C. L'obiettivo è quello di avere tutti e 3 gli oggetti a *casa* (fuori della borsa).

6. **Dominio (Teatime):** Un robot agisce in un ambiente costituito da un certo numero di stanze e un corridoio. Da qualche parte ci sono una macchina del tè e una pila di tazze. Il robot può andare da una locazione all'altra (stanze o corridoio), purché origine e destinazione siano connesse, può prendere una tazza vuota se non ha oggetti in mano, può riempire di tè la tazza vuota che ha in mano, può consegnare il tè a un abitante di una stanza che lo abbia ordinato.

   Problema: L'ambiente è costituito da 4 stanze, ciascuna con un abitante che ha ordinato il tè. Le stanze sono tutte connesse al corridoio; inoltre la stanza 1 è connessa alla stanza 2, e la stanza 3 alla 4. La macchina del tè è nella stanza 1 e la pila di tazze nella stanza 2. L'obiettivo è di consegnare il tè a tutti e 4 gli abitanti delle stanze.