RELAZIONE PROGETTO **ASP**

Università degli studi di Torino

Corso di Laurea Magistrale in Informatica

Intelligenza Artificiale e Laboratorio 2015/2016

DOCENTE:

Prof. Gian Luca Pozzato

RELAZIONE A CURA DI:

823832 **Giovanni Bonetta**
762661 **Riccardo Renzulli**
763056 **Gabriele Sartor**

ESERCIZI SUI VINCOLI

LE CINQUE CASE

Questo esercizio presenta 5 case allineate, ognuna con un colore diverso e un proprietario di nazionalità diversa. Ogni proprietario ha un lavoro, una bevanda preferita ed un animale diversi da quelli degli altri.
Dato l'insieme di valori ed una serie di vincoli, il programma trova chi possiede la zebra tra i diversi padroni di casa.
Per risolvere il problema, viene generato un solo fatto *casa*(*X,Col,Naz,Prof,Bev,Anim*) per ognuna delle cinque case nel seguente modo:

```
numeroCase(1..5).
nazionalita(italiano;giapponese;inglese;norvegese;spagnolo).
colori(rosso;giallo;blu;bianco;verde).
professione(pittore;scultore;diplomatico;violinista;dottore).
bevanda(te;caffe;latte;succo).
animali(cane;lumache;volpe;cavallo;zebra).

1{ casa(X,Col,Naz,Prof,Bev,Anim) : nazionalita(Naz), colori(Col),
professione(Prof), bevanda(Bev), animali(Anim) }1 :- numeroCase(X).
```

Dopodiché possiamo individuare nel codice tre tipologie diverse di vincoli. Il primo è di unicità di alcuni valori; per esempio abbiamo che tutte le case hanno colore diverso.

```
%Vincolo: tutte le case hanno colore diverso
:- casa(X,Col,_,_,_,_),casa(XX,Col,_,_,_,_),X!=XX.
```

In questo *integrity constraint* viene espresso il fatto che non possono esserci due case diverse con lo stesso colore.
Il secondo è stato utilizzato per la correlazione di due campi della stessa casa. Ad esempio, se l'*inglese* abita nella casa di colore *rosso*, possiamo dire che se una *casa* è *rossa* allora la *nazionalità* del proprietario non può essere diversa da *inglese*.

```
%Vincolo1: l'inglese vive nella casa rossa
:- casa(_,rosso,Naz,_,_,_), Naz!=inglese.
```

Infine, sono presenti alcuni vincoli che fanno riferimento all'adiancenza di due case. Il nostro programma prevede che il *norvegese* abiti nella casa di fianco a quella *blu*.

```
%Vincolo11: La casa del norvegese è adiacente a quella blu
:- casa(X,_,norvegese,_,_,_), casa(Y,blu,_,_,_,_), |X-Y|>1.
:- casa(X,_,norvegese,_,_,_), casa(Y,blu,_,_,_,_), X==Y.
```

Nei nostri vincoli abbiamo scritto che le due case non possono essere la stessa casa e nemmeno essere a più di una posizione di distanza.
Al termine dell'esecuzione, viene mostrato il fatto *chiHaZebra* per rispondere alla domanda posta dal problema.

IL CLUB DEI ROSICONI

In questo esercizio si vogliono scoprire delle informazioni riguardanti i due personaggi Oreste e Bruno dei quali non conosciamo il grado di onestà. In particolare siamo interessati alla loro età (anziano/giovane) e alla loro squadra del cuore (inter/torino).
Abbiamo quindi deciso di usare i seguenti aggregati per definire un fatto *ha_eta, ha_tipo* e *tifa* per ognuno di essi.

```
1{ha_eta(X,E):eta(E)}1:- tifoso(X).

1{ha_tipo(X,T):tipo(T)}1:-tifoso(X).

1{tifa(X,S):squadra(S)}1:-tifoso(X).
```

Sappiamo inoltre che:

- i tifosi dell'Inter anziani dicono sempre la verità, mentre quelli giovani mentono sempre
- i tifosi del Torino giovani dicono sempre la verità, mentre quelli anziani mentono sempre

Questa conoscenza è implementata dalle seguenti regole:

```
ha_tipo(X,onesto):-tifa(X,inter),ha_eta(X,anziano).
ha_tipo(X,bugiardo):-tifa(X,inter),ha_eta(X,giovane).
ha_tipo(X,onesto):-tifa(X,torino),ha_eta(X,giovane).
ha_tipo(X,bugiardo):-tifa(X,torino),ha_eta(X,anziano).
```

Infine Oreste e Bruno ci comunicano alcuni rispettivi fatti. Oreste dice che Bruno è anziano e tifoso del Torino, mentre Bruno ci comunica che Oreste è giovane e tifoso dell'Inter.

Queste affermazioni sono modellate come segue:

```
o_dice:-tifa(bruno,torino),ha_eta(bruno,anziano).
b_dice:-tifa(oreste,inter),ha_eta(oreste,giovane).
```

Risolviamo quindi il problema affermando di non volere come soluzione i modelli contenenti fatti discordanti in cui un personaggio è onesto e dice il falso oppure bugiardo e dice il vero.

```
:-o_dice, ha_tipo(oreste,bugiardo). %se oreste è bugiardo non dice il vero
:-b_dice, ha_tipo(bruno,bugiardo). %se bruno è bugiardo non dice il vero
:-not o_dice, ha_tipo(oreste,onesto). %se oreste è onesto dice il vero
:-not b_dice, ha_tipo(bruno,onesto). %se bruno è onesto dice il vero
```

Al termine dell'esecuzione vengono mostrati i fatti soluzione: *tifa/2* e *ha_eta/2*.

IL QUADRATO MAGICO

Lo scopo di questo esercizio è quello di inserire in un quadrato *N x N* tutti gli interi da 1 a N^2, in modo da avere la stessa somma su tutte le righe, tutte le colonne e sulle due diagonali principali.

Per realizzare un'implementazione più concisa, abbiamo sfruttato il fatto che un quadrato *N x N* ha come costante "magica" un numero ben definito. Grazie ad un aggregato è stata generata una *magic_cell* per ogni coppia *(i, j)* della matrice mentre un *integrity constraint* è stato usato per assicurarci che ogni *magic_cell* avesse un valore distinto preso dal range $[1..N^2]$.

Per ogni riga, colonna e diagonale è stata calcolata la somma degli elementi controllando che fosse proprio uguale alla *magic_constant*. Affinchè il programma generasse un modello corretto, sono stati definiti dei vincoli che impongono la presenza di esattamente *N* righe, *N* colonne e due diagonali corrette.

Nel nostro caso abbiamo specificato *N* uguale a 3 ma potrebbe assumere valori anche più grandi.

```
#const n=3.

row(1..n).
col(1..n).
value(1..n*n).
magic_constant((n * (n * n + 1)) / 2).

% Every cell must be filled with a positive number.
1 {magic_cell(R,C,V) : value(V)} 1 :- col(C), row(R).

% Every cell must be filled with a distinct number.
:- magic_cell(R1,C1,V), magic_cell(R2,C2,V), 1 {R1 != R2; C1 != C2} 2.

% Given a row, the sum of its integers must be equal to the magic constant
row_OK(R) :- S = #sum { V : magic_cell(R,C,V), col(C) }, row(R), magic_constant(S).
% Given a column, the sum of its integers must be equal to the magic constant
column_OK(C) :- S = #sum { V: magic_cell(R,C,V), row(R) }, col(C), magic_constant(S).
% Given the primary diagonal, the sum of its integers must be equal to the magic constant
primarydiagonal_OK :- S = #sum { V : magic_cell(R,R,V), row(R) }, magic_constant(S).
% Given the secondary diagonal, the sum of its integers must be equal to the magic constant
secondarydiagonal_OK :- S = #sum { V : magic_cell(R,n + 1 - R,V), row(R) }, magic_constant(S).

% There must be exactly n rows in which the sum of their integers is equal to the magic constant.
:- not n {row_OK(R)} n.
% There must be exactly n columns in which the sum of their integers is equal to the magic constant.
:- not n {column_OK(C)} n.
% There must be exactly one primary diagonal in which the sum of its integers is equal to the magic constant.
:- not 1 {primarydiagonal_OK} 1.
% There must be exactly one secondary diagonal in which the sum of its integers is equal to the magic constant.
:- not 1 {secondarydiagonal_OK} 1.
```

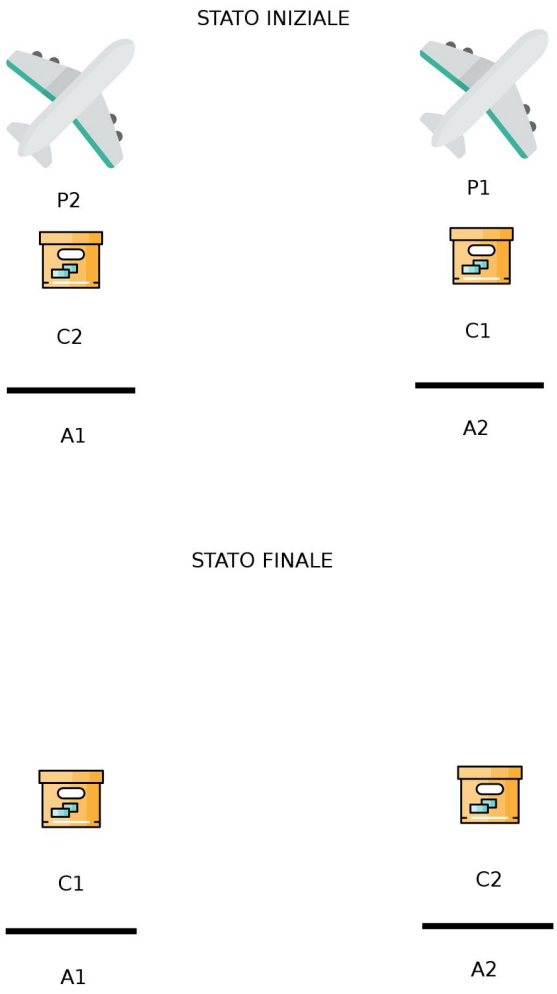
Una possibile soluzione è la seguente:

6	1	8
7	5	3
2	9	4

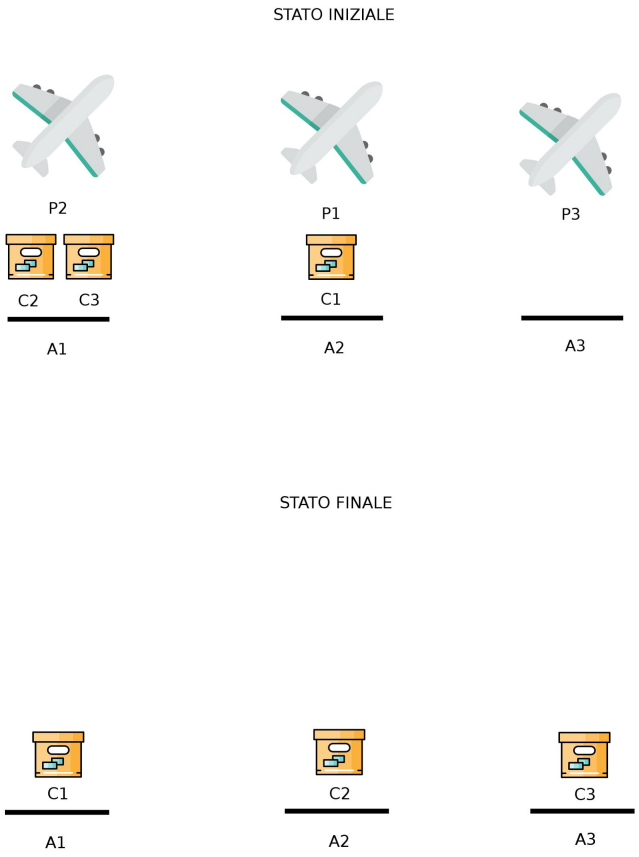
Da notare come il programma non sia soddisfacibile, come appunto accade correttamente, quando *N* è uguale a 2 in quanto non esiste una soluzione al problema.

PIANIFICAZIONE

Come problema di pianificazione, abbiamo optato per quello del trasporto aereo di merci.
Abbiamo formulato sia la versione sequenziale sia la versione parallela e le abbiamo testate su due domini diversi, uno più semplice e uno leggermente più complesso.
In entrambi i domini non è importante la posizione finale degli aerei ma solamente quella dei carichi.
Il primo dominio, raffigurato nell'immagine sottostante, consiste di due aerei, due carichi e due aeroporti.



Il secondo dominio invece consiste di tre aerei, tre carichi e tre aeroporti.



SEQUENZIALE

Per definizione di pianificazione sequenziale, in ogni stato è possibile applicare solamente un'azione alla volta.

```
1 {occurs(A,S) : action(A)} 1 :- level(S).
```

Abbiamo modellato il problema di planning in modo generale attraverso predicati come *action* per le azioni, *occurs* per l'esecuzione di una certa azione in un certo stato, *fluent* per i fluenti e *holds* per gli effetti delle azioni eseguite in un certo stato. Inoltre le precondizioni sono esplicitate sotto forma di *integrity constraints* e i fluenti negativi sono derivati grazie a delle regole causali.
Nella figura sottostante è possibile vedere come abbiamo rappresentato le precondizioni delle regole.

```
% PRECONDITIONS
:- occurs(load(C,P,A),S), not holds(at(P,A),S).
:- occurs(load(C,P,A),S), not holds(at(C,A),S).

:- occurs(load(C,_,_),S), holds(in(C,_),S).
:- occurs(load(_,P,_),S), holds(in(_,P),S).

:- occurs(unload(C,P,A2),S), holds(in(C,P),S), holds(at(P,A1),S), A1 != A2.
:- occurs(unload(C,P,_),S), not holds(in(C,P),S).
:- occurs(fly(P,From,To),S), not holds(at(P,From),S).
```

Per il primo dominio esistono 22 modelli stabili (ciascuno di lunghezza minima 6). Uno di questi è ad esempio il seguente:

1. *occurs(load(c1,p1,a2),0)*
2. *occurs(load(c2,p2,a1),1)*
3. *occurs(fly(p1,a2,a1),2)*
4. *occurs(unload(c1,p1,a1),3)*
5. *occurs(fly(p2,a1,a2),4)*
6. *occurs(unload(c2,p2,a2),5)*

Per il secondo dominio le soluzioni possibili sono molto più numerose (di lunghezza minima 9). Ad esempio:

1. *occurs(load(c2,p2,a1),0)*
2. *occurs(load(c1,p1,a2),1)*
3. *occurs(fly(p1,a2,a1),2)*
4. *occurs(unload(c1,p1,a1),3)*
5. *occurs(load(c3,p1,a1),4)*
6. *occurs(fly(p1,a1,a3),5)*
7. *occurs(fly(p2,a1,a2),6)*
8. *occurs(unload(c3,p1,a3),7)*
9. *occurs(unload(c2,p2,a2),8)*

PARALLELA

Nella pianificazione parallela invece è consentita l'esecuzione di più azioni ad ogni passo.

```
1 {occurs(A,S) : action(A)} :- level(S).
```

È stato perciò necessario aggiungere le seguenti precondizioni a quelle già esistenti in modo da ottenere soluzioni corrette: non consentiamo che due azioni dello stesso aereo vengano eseguite entrambe nello stesso stato. Per semplicità abbiamo anche deciso che gli aerei hanno una capienza singola quindi non possono effettuare una nuova operazione di *load* senza prima aver depositato il carico.

```
% parallel
:- occurs(load(C,P1,_),S), occurs(load(C,P2,_),S), P1 != P2.

:- occurs(load(_,P,_),S), occurs(unload(_,P,_),S).
:- occurs(load(_,P,_),S), occurs(fly(P,_,_),S).
:- occurs(unload(_,P,_),S), occurs(fly(P,_,_),S).
:- occurs(fly(P,_,To1),S), occurs(fly(P,_,To2),S), To1 != To2. % useful when there more then two airports
```

Per il primo dominio esiste un solo modello stabile di lunghezza minima 3:

1. *occurs(load(c2,p2,a1),0)*, *occurs(load(c1,p1,a2),0)*
2. *occurs(fly(p1,a2,a1),1)*, *occurs(fly(p2,a1,a2),1)*
3. *occurs(unload(c1,p1,a1),2)*, *occurs(unload(c2,p2,a2),2)*

Per il secondo dominio le soluzioni possibili invece sono 114 (di lunghezza minima 5). Ad esempio:

1. *occurs(fly(p3,a3,a1),0)*
2. *occurs(load(c2,p2,a1),1)* , *occurs(load(c3,p3,a1),1)* , *occurs(load(c1,p1,a2),1)*
3. *occurs(fly(p1,a2,a1),2)*, *occurs(fly(p2,a1,a2),2)*, *occurs(fly(p3,a1,a3),2)*
4. *occurs(unload(c1,p1,a1),3)*, *occurs(unload(c2,p2,a2),3)*
5. *occurs(unload(c3,p3,a3),3)*