Answer Set Programming
e la
codifica dei rompicapi

Agostino Dovier
Dip. di Matematica e Informatica
Univ. degli Studi di Udine
`www.dimi.uniud.it/dovier`

November 27, 2014

Abstract

Il materiale presentato è abbastanza generale da poter essere utilizzato con qualunque ASP solver (smodels, cmodels, DLV ecc). Si suggerisce tuttavia di scaricare `clingo` dal sito `http://potassco.sourceforge.net/` per il sistema operativo utilizzato nel proprio calcolatore. Dallo stesso sito, sotto "teaching" si può trovare del materiale didattico preparato dal gruppo dell'Università di Potsdam. La sintassi nelle ultime versioni è leggermente cambiata, in particolare si faccia attenzione al valore assoluto che ora si scrive con $|\cdot|$ mentre prima era `abs`.

1 Programmi ASP

Una *formula atomica (o atomo)* è un oggetto del tipo

$$p(s_1, \ldots, s_p)$$

ove p è un simbolo di predicato e $s_1, \ldots, s_p$ sono simboli di costante o di variabile. Per esattezza useremo nomi che iniziano con la lettera minuscola per costanti e predicati, nomi con la lettera maiuscola per le variabili. Per le costanti possiamo anche usare dei numeri interi.

Per chiarezza, questi nomi li scegliamo noi a seconda del significato che deve avere un predicato nel programma che scriviamo (non ci sono parole riservate o liste di nomi da utilizzare).

Esempi di *atomi* sono:

$$\texttt{zio}(\texttt{paperino}, \texttt{qui}, \texttt{quo}, \texttt{qua}), \texttt{fratello}(\texttt{linus}, \texttt{lucy}), \texttt{maggiore}(2, X)$$

Un *programma ASP* è un insieme di regole del tipo:

$$H :- A_1, \ldots, A_n, \textbf{not}\, B_1, \ldots, \textbf{not}\, B_m.$$

ove i vari H, A_i, B_j sono *atomi*. H è detta *testa*; $A_1, \ldots, A_n, \textbf{not}\, B_1, \ldots, \textbf{not}\, B_m$ è detto *corpo*. Se $n = m = 0$ allora la regola è detta *fatto* (e si omette $:-$). Sono ammesse anche regole senza testa, in tal caso si parla di *vincoli*. Il significato di $:-$ è quello dell'implicazione a sinistra $\leftarrow$.

Un esempio di programma è il seguente:

```
deo(zeus).
uomo(socrate).
personaggio(X) :- deo(X).
personaggio(X) :- uomo(X).
mortale(X) :- personaggio(X), not deo(X).
:- deo(X), uomo(X).
```

I primi due sono fatti. Poi ci sono tre regole. L'ultimo è un vincolo.

Il significato dei fatti è quello di asserire informazioni in modo esplicito. Si dice che `zeus` è un `deo` e che `socrate` è un `uomo`.

Analizziamo ora il ruolo delle variabili. Ogni regola ha la sola visibilità delle proprie variabili (non c'è nessuna parentela tra le variabili in regole diverse). Le variabili vanno intese come *chiuse universalmente*. Ad esempio nella prima regola (non fatto) viene detto che per ogni valore di X, se X è un `deo`, allora X sarà anche un `personaggio` della nostra storia.

Le costanti presenti nel programma (in questo caso `zeus` e `socrate`) permettono di fornire una versione equivalente, piatta (o, come viene detto in inglese, *ground*) del programma. Il programma può essere riscritto senza le variabili nel seguente modo:

```
deo(zeus).
uomo(socrate).
personaggio(zeus) :- deo(zeus).
personaggio(socrate) :- deo(socrate).
personaggio(zeus) :- uomo(zeus).
personaggio(sorate) :- uomo(socrate).
mortale(zeus) :- personaggio(zeus), not deo(zeus).
mortale(socrate) :- personaggio(socrate), not deo(socrate).
:- deo(zeus), uomo(zeus).
:- deo(socrate), uomo(socrate).
```

Ovviamente, a fronte di molte costanti e di molte variabili nella stessa clausola, questa fase, detta fase di *grounding*, può essere dispendiosa sia a livello di tempo che di spazio (generazione di un file molto grosso a fronte di un programma con le variabili di dimensioni limitate).

Tuttavia, per comprendere cosa stiamo facendo (e cosa calcolerà poi il risolutore ASP) è bene ragionare sulla versione ground del programma. Per aiutare il risolutore nel grounding è necessario che tutte le variabili presenti in una regola siano anche presenti in un predicato di dominio usato senza il not nel corpo.

Rimane da illustrare brevemente il ruolo dei vincoli. Cosa impone un vincolo? In sunto va letto come: "Non è possibile che: ..." dunque ad esempio nel programma ground sopra si dice che non è possibile che socrate sia (contemporaneamente) uomo e deo. Lo stesso per zeus. Guardando al programma iniziale (non ground) si dice che: "Non è possibile che ci sia un X che sia contemporaneamente deo e uomo".

Vediamo un altro esempio di vincolo.

Supponete di aver codificato una relazione binaria `genitore` tra coppie di individui e una relazione binaria `eta` (immaginatevi l'accento) che dato un individuo mi fornisce la sua età. Il seguente vincolo

```
:- genitore(X,Y), eta(X,EX), eta(Y,EY), EX < EY.
```

dice che non è possibile che ci siano due individui, poniamo X e Y, tali che X è genitore di Y, e l'età di X è minore dell'età di Y. Vedremo che i vincoli saranno importantissimi per le nostre codifiche.

Un programma ASP è un insieme di fatti, di vincoli e di implicazioni. L'obiettivo è calcolare l'insieme delle *conseguenze logiche*, ovvero degli atomi che sono veri in qualunque "modello logico ragionevole" del programma.[1]

[1] In queste note omettiamo la definizione formale di modello e la lasciamo a un livello intuitivo.

Nel programma appena visto, ad esempio, ci aspetteremo di dedurre che `mortale(socrate)`.

Sfortunatamente ci sono programmi che non hanno modelli o che ne hanno più di uno e del tutto indipendenti tra loro. Si osservi il seguente programma ground basato sull'unica costante `a`:

```
p(a) :- not q(a).
q(a) :- not p(a).
```

Se `p(a)` è falso, allora `q(a)` deve essere vero (e viceversa). Dunque ci sono due "modelli" che possiamo identificare con l'insieme degli atomi veri:

1. {`p(a)`}
2. {`q(a)`}

In realtà vi sarebbe pure il modello con entrambi gli atomi veri. In generale, modelli che sono sovrainsiemi propri di altri modelli non sono interessanti. Si cercano sempre modelli *minimali*.

La nozione precisa di modello "ragionevole" e che abbia anche la proprietà di minimalità esiste ed è la nozione dovuta a Gelfond-Lifschitz di *modello stabile* o *answer set* (1988). Qui confidiamo che il risolutore ASP (che è basato su tale nozione) calcoli delle conseguenze opportune: lo useremo come una *scatola nera* senza guardarci dentro.

Completiamo questa breve introduzione con alcune comode abbreviazioni sintattiche.

Un *ground cardinality constraint*[2] si usa come fosse un atomo ed ha la forma

$$n\{L_1, \ldots, L_h\}m$$

dove $L_1, \ldots, L_h$ sono atomi e n e m sono numeri interi (uno o entrambi possono essere assenti).

Se usata come testa di un *fatto* questa primitiva fa sí che vengano cercate le soluzioni/modelli in cui un numero di atomi compreso tra n ed m (tra gli h atomi $L_1, \ldots, L_h$) sia vero.

Come caso particolare, il fatto

$$1\{L_1, \ldots, L_h\}1.$$

fa sí che vengano cercate le soluzioni/modelli in cui esattamente un atomo tra $L_1, \ldots, L_h$.

Un *cardinality constraint* è la generalizzazione con variabili. Ad esempio con due variabili:

$$n\{p(X, Y) : range(X, Y)\}m$$

dove `range` è un predicato che fissa dei valori alle coppie. Vengono cercati i modelli in cui un numero di atomi della forma $p(X, Y)$ (tali che $range(X, Y)$) è compreso tra n e m.

Useremo molto la possibilità sopra nei nostri modelli. In generale nei problemi servirà calcolare una funzione (chiamiamola `fun`) tra un `dominio` (un insieme di valori definito da un'altra relazione, di solito data mediante dei fatti: ad esempio: `dom(1). dom(2) dom(3).`) e un codominio (similmente, ad esempio: `cod(3). cod(4)`. La seguente regola, basata su un cardinality constraint:

```
1 {fun(X,Y): cod(X,Y) } 1 :- dom(X).
```

dice di assegnare mediante la funzione `fun` ad ogni elemento X del dominio esattamente un elemento Y del codominio. Equivale alla versione ground:

```
1 {fun(1,3),fun(1,4) } 1.
1 {fun(2,3),fun(2,4) } 1.
1 {fun(3,3),fun(3,4) } 1.
```

Un'altra abbreviazione utilizzata è la seguente: in luogo dei fatti:

[2] Qui ne mostriamo una variante semplificata rispetto alle possibilità permesse

```
val(4).    val(5).    val(6).    val(7).
```

possiamo scrivere (si noti che i punti sono due):

```
val(4..7).
```

Al posto della regola

```
p(X):- dominio(X), dominio(Y), dominio(Z), q(X,Y), not r(Y,Z).
```

possiamo scrivere (si notino i ";"):

```
p(X):- dominio(X;Y;Z), q(X,Y), not r(Y,Z).
```

Inoltre possiamo usare le tipiche funzioni matematiche: $+$, $-$, $*$, $/$, `mod`, $<$,$>$, $=$, $!=$, ecc.

Il simbolo % si usa per i *commenti*. Tutto il testo dopo quel simbolo fino alla fine della riga non viene letto dal solver ASP.

2 Modellare con ASP

In questa sezione introduco alcuni esempi di modelli di giochi/ropicapi che ci danno una base per progettare autonomamente delle codifiche per altri problemi.

2.1 Le n regine

Consideriamo il seguente problema: vogliamo trovare un modo per posizionare 4 *regine* (degli scacchi) in una scacchiera 4×4 in modo tale che nessuna sia attaccata dalle altre (si veda Fig. 1).

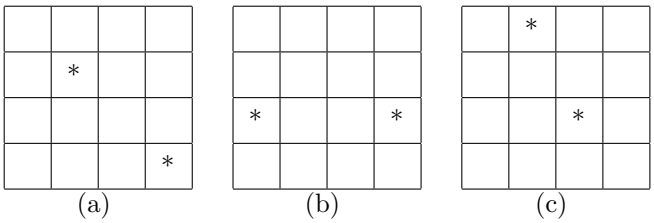

Figure 1: Regine che si attaccano (a), (b), due regine che non si attaccano (c).

Il problema potrebbe essere codificato nel seguente modo. Innanzitutto inseriamo un predicato che mi dice che le coordinate (sia sull'asse x che sull'asse y) sono i punti dall'1 al 4. Questo può essere modellato dal predicato `coord` definito da un unico fatto:

```
coord(1..4).
```

A questo punto dobbiamo definire un predicato che dica che per ogni colonna dall'1 al 4 assegnamo la regina in una riga dall'1 al 4. Definiamo pertanto il predicato `queen` come segue:

```
1 {queen(1,1), queen(2,1), queen(3,1), queen(4,1) } 1.
1 {queen(1,2), queen(2,2), queen(3,2), queen(4,2) } 1.
1 {queen(1,3), queen(2,3), queen(3,3), queen(4,3) } 1.
1 {queen(1,4), queen(2,4), queen(3,4), queen(4,4) } 1.
```

Sebbene corretta questa definizione sarebbe poi poco generalizzabile per scacchiere più grandi. Sostituiamola pertanto con la versione con variabili del cardinality constraint:

```
1{queen(X,Y) : coord(X)}1 :- coord(Y).
```

La seconda definizione, dopo il grounding, diventa esattamente la prima.

Ora dobbiamo dire che due regine non si possono attaccare in orizzontale, e poi in diagonale.

Per l'attacco in orizzontale uso un vincolo

```
:- coord(X;Y1;Y2), queen(X,Y1), queen(X,Y2),  Y1 != Y2.
```

che dice: "Non è possibile che esistano delle coordinate X, Y_1, Y_2, con $Y_1 \neq Y_2$ tali che la regina in colonna Y_2 sta in riga X e la regina in colonna Y_1 sta nella (stessa) riga X."

Per l'attacco in orizzontale uso un vincolo

```
:- coord(X1;X2,Y1,Y2), queen(X1,Y1), queen(X2,Y2), X1 != X2,  | X1-X2 | = | Y1-Y2 |.
```

che dice: "Non è possibile che esistano delle coordinate (X_1, Y_1) e (X_2, Y_2), con $X_1 \neq X_2$ tali che $|X_1 - X_2| = |Y_1 - Y2|$. Quest'ultima, se ci pensate, è proprio la condizione di attacco in diagonale. Come accennato nell'Abstract, in vecchie versioni di clingo il valore assoluto andava indicato con `abs` o `#abs` (c'è stata evoluzione sulla sintassi di questa funzione).

Supponete di aver salvato il file come `regine.lp`.

Con il comando `clingo regine.lp` viene trovata una soluzione. Con il comando `clingo regine.lp 0` vengono trovate tutte le soluzioni (quante?).

Il programma scritto si può generalizzare con un semplice cambiamento. Se sostituite il primo fatto con:

```
coord(1..n).
```

il programma si generalizza per risolvere il problema delle n regine. E' sufficiente fornire il valore di n al momento dell'esecuzione. Ad esempio, per lanciare il problema con 8 regine, si deve digitare il comando `clingo -c n=8 regine.lp`

Una nota sul formato di output (anche questo modificato leggermente nelle ultime versioni di clingo). Se volete vedere soltanto alcuni predicati (ad esempio `queen`) dovreste scrivere alla fine del vostro file le direttive `#show` con i predicati e il loro numero di argomenti che volete visualizzare (ad esempio `#show queen/2.`)

2.2 Un quadrato magico

Vogliamo codificare il problema del quadrato magico, ovvero riempire un quadrato di lato 3, in modo tale che la somma dei numeri presenti in ogni riga, colonna, e diagonale sia la stessa. In ogni cella ammettiamo numeri dall'1 al 9. Aggiungeremo poi ulteriori restrizioni.

Dovremo definire un predicato `magic(X,Y,V)` che assegna a ogni coppia (X, Y) un valore V.

Innanzitutto definiamo dei predicati di dominio e stabiliamo, come fatto nell'esempio precedente, che `magic` è una funzione:

```
lato(1..3).
valore(1..9).
diag(1..2).

1 { magic(X,Y,V) : valore(V) } 1 :- lato(X),lato(Y).
```

Ora si tratta di calcolare le somme su righe colonne e diagonali. Trattandosi di un quadrato di dimensioni 3 possiamo pensare di espandere la somma nelle tre componenti. Pensate a come potreste generalizzare la cosa a quadrati di dimensione maggiore:

```
sum_col(X,V1+V2+V3)  :- magic(X,1,V1), magic(X,2,V2), magic(X,3,V3).

sum_row(Y,V1+V2+V3)  :- magic(1,Y,V1), magic(2,Y,V2), magic(3,Y,V3).

sum_diag(1,V1+V2+V3) :- magic(1,1,V1), magic(2,2,V2), magic(3,3,V3).

sum_diag(2,V1+V2+V3) :- magic(1,3,V1), magic(2,2,V2), magic(3,1,V3).
```

A questo punto poniamo i vincoli. L'evento "somma sbagliata" (che modelliamo con un predicato dal nome `diff_sum`) può essere scatenato da due colonne con differente somma, da due righe, da due diagonali, o dalla somma differente tra una colonna e una riga o tra una riga e una diagonale o tra una colonna e una diagonale. Se ci pensate un attimo quest'ultimo test è superfluo se fisso i precedenti e dunque non lo codifico:

```
% Tutte le colonne uguali
diff_sum :- lato(X1;X2), X1 < X2,  sum_col(X1,V1), sum_col(X2,V2), V1 != V2.
% Tutte le righe uguali
diff_sum :- lato(X1;X2), X1 < X2,  sum_row(X1,V1), sum_row(X2,V2), V1 != V2.
% Tutte le diagonali uguali
diff_sum :- sum_diag(1,V1), sum_diag(2,V2), V1 != V2.
% riga-colonna uguale (basta la prima)
diff_sum :- sum_col(1,V1), sum_row(1,V2), V1 != V2.
% riga-diag uguale (basta la prima)
diff_sum :- sum_row(1,V1), sum_diag(1,V2), V1 != V2.
% colonna-diag uguale e' superfluo (se fossero diversi, lo sarebbe uno dei due sopra)
diff_sum :- sum_col(1,V1), sum_diag(1,V2), V1 != V2.
```

A questo punto è sufficiente porre il vincolo che dice che non è possibile che sia vero il predicato `diff_sum`, ovvero:

```
:- diff_sum.
```

Lanciate l'esecuzione e vedrete una soluzione (ad esempio, quella in Figura 2 (a)).

Il problema diventa più interessante se chiediamo che in ogni riga e in ogni diagonale i numeri siano diversi. Questo può essere imposto con:

```
% righe
:- lato(X1;X2;Y1), X1<X2, valore(V), magic(X1,Y1,V), magic(X2,Y1,V).
% diagonale 1
:- lato(X1;X2), X1<X2, valore(V), magic(X1,X1,V), magic(X2,X2,V).
% diagonale 2
:- lato(X1;X2), X1<X2, valore(V), magic(X1,4-X1,V), magic(X2,4-X2,V).
```

Perchè ho scritto $4 - X_1, 4 - X_2$?

Infine si può richiedere che le 9 celle siano tutte differenti:

```
uguali :- lato(X1;X2;Y1;Y2), Y1<Y2, valore(V), magic(X1,Y1,V), magic(X2,Y2,V).
uguali :- lato(X1;X2;Y1;Y2), X1<X2, valore(V), magic(X1,Y1,V), magic(X2,Y2,V).
:- uguali.
```

1	1	1
1	1	1
1	1	1

(a)

4	8	3
4	5	6
7	2	6

(b)

2	9	4
7	5	3
6	1	8

(c)

Figure 2: Tre soluzioni per il quadrato magico: senza vincoli di differenza (a), con vincoli su ogni riga e diagonale (b), con il vincolo di differenza di ogni coppia di valori (c)

1	10	3	6
4	7	12	9
11	2	5	
6		8	13

Figure 3: Visitare un quadrato "a cavallo". La passeggiata si è interrotta al passo 13: non ho più nuove celle raggiungibili. Non è una soluzione.

2.3 Cavalcata in una griglia

Quando ero alle superiori mi divertivo a provare a riempire un quadrato $n \times n$ partendo da una cella qualunque e muovendomi poi come il cavallo negli scacchi senza mai ripassare per la stessa cella. Se non ci avete mai giocato, provateci: può essere divertente (si veda Fig 3).

Cerchiamo di modellarlo in modo parametrico (ovvero che funzioni con quadrati di lato arbitrario). Abbiamo già visto che un parametro (o più) può essere fissato al momento dell'esecuzione. Il risolutore farà l'opportuno grounding del programma basandosi su quel dato. Se il lato del quadrato è n, vorremmo fare una cavalcata di n^2 passi. Definiamo pertanto i predicati di dominio dipendenti da n:

```
lato(1..n).
passi(1..n*n).
coppia(X,Y) :- lato(X), lato(Y).
```

Al solito stabiliamo che al passo I ci può essere una ed una sola posizione:

```
1 { posizione(I,X,Y) : coppia(X,Y) } 1 :- passi(I).
```

Per forzare i passi successivi ad eseguire la regola del cavallo prima diamo la definizione di (buon) successivo, poi diciamo che non ci possono essere passi senza che sia verificata la regola. Infine diciamo che non si può tornare sulla stessa cella due volte.

```
successivo(X1,Y1,X2,Y2) :- coppia(X1,Y1), coppia(X2,Y2), |X1-X2| = 1, |Y1-Y2| = 2.
successivo(X1,Y1,X2,Y2) :- coppia(X1,Y1), coppia(X2,Y2), |X1-X2| = 2, |Y1-Y2| = 1.

%%% passi successivi sono legati alla regola del cavallo
:- passi(I;I+1), coppia(X1,Y1), coppia(X2,Y2),
   posizione(I,X1,Y1), posizione(I+1,X2,Y2), not successivo(X1,Y1,X2,Y2).

%%% non ritorno mai sulla stessa cella
:- passi(I1;I2), I1 < I2, coppia(X,Y),
   posizione(I1,X,Y), posizione(I2,X,Y).
```

Volendo possiamo aggiungere una euristica sulla cella iniziale; ad esempio con

```
:- posizione(1,X,Y), X+Y>3.
```

permettiamo di partire dalle celle $(1,1),(1,2)$, e $(2,1)$. Se il file viene salvato in `cavallo.lp` allora possiamo chiamare l'esecuzione (ad sempio con $n = 5$) con: `clingo -c n=5 cavallo.lp` Al solito aggiungendo uno 0 alla fine vengono mostrate tutte le soluzioni. Ci sono soluzioni con $n = 4$?

2.4 Il Sudoku

Il Sudoku è un gioco molto diffuso di cui non riportiamo le regole (note a tutti). Iniziamo a fornire una rappresentazione dell'input (instanze in rete si trovano facilmente, ad esempio in `http://www.tellmehowto.net/sudoku/veryhardsudoku.html`. L'istanza commentata può essere rappresentata mediante un predicato ternario che chiamiamo `x` (come incognita):

```
% _,_,6, _,_,_, _,9,_,
% _,_,_, 5,_,1, 7,_,_,
% 2,_,_, 9,_,_, 3,_,_,
% _,7,_, _,3,_, _,5,_,
% _,2,_, _,9,_, _,6,_,
% _,4,_, _,8,_, _,2,_,
% _,_,1, _,_,3, _,_,4,
% _,_,5, 2,_,7, _,_,_,
% _,3,_, _,_,_, 8,_,_
x(1, 3, 6).  x(1, 8, 9).
x(2, 4, 5). x(2, 6, 1). x(2, 7, 7).
x(3, 1, 2). x(3, 4, 9). x(3, 7, 3).
x(4, 2, 7). x(4, 5, 3). x(4, 8, 5).
x(5, 2, 2). x(5, 5, 9). x(5, 8, 6).
x(6, 2, 4). x(6, 5, 8). x(6, 8, 2).
x(7, 3, 1). x(7, 6, 3). x(7, 9, 4).
x(8, 3, 5). x(8, 4, 2). x(8, 6, 7).
x(9, 2, 3). x(9, 7, 8).
```

A questo punto definiamo i domini (coordinate dall'1 al 9, valori dall'1 al 9; potremmo usare lo stesso predicato ma per chiarezza nelle definizioni successive ho preferito metterne due), e definiamo al solito il predicato che richiede il calcolo di una funzione da una coppia di coordinate ad un valore:

```
coord(1..9).
val(1..9).
% Per ogni cella si assegna esattamente un valore
1 { x(X,Y,N) : val(N) } 1   :- coord(X;Y).
```

Avremo bisogno della nozione di sotto quadrato: il seguente predicato partiziona in 9 sottoquadrati indirizzati con un numero dall'1 al 9 le varie celle:

```
square(I,X,Y) :-
   coord(X;Y;I),
   I == (X-1) / 3 + 3*((Y-1) / 3) + 1.
```

Come/perchè funziona? fatevi degli esempi!!!

A questo punto dobbiamo modellare che in ogni riga, colonna, sottoquadrato ci sia una ed una sola volta ogni valore:

```
% Ogni valore viene usato esattamente una volta in una colonna
1 { x(X,Y,N) : coord(X) } 1 :- coord(Y), val(N).

% Ogni valore viene usato esattamente una volta in una riga
1 { x(X,Y,N) : coord(Y) } 1 :- coord(X), val(N).

% Ogni valore viene usato esattamente una volta in un sottoquadrato
1 { x(X,Y,N) : square(I,X,Y) } 1  :- val(N), coord(I).
```

Buon divertimento! Avete notato i tempi di esecuzione? Provate a risolverlo a mano!

2.5 La capra e il cavolo

L'ultimo esempio che vediamo riguarda il problema della capra e cavolo. Una capra, un lupo, un cavolo, e il proprietario di capra e cavolo devono attraversare il fiume su di una barca. Ci sono dei vincoli. La barca non può trasportare più di un "personaggio" contemporaneamente oltre all'uomo, non possiamo lasciare da soli su un lato del fiume capra e cavolo oppure lupo e capra.

Il problema è un tipico problema di *planning*, bisogna trovare una sequenza di azioni elementari da eseguirsi una per ogni istante di tempo che permettano di raggiungere lo scopo.

Iniziamo con i predicati di dominio:

```
time(1..n).
place(left;right;boat).
object(man;goat;cabbage;wolf).
```

Dobbiamo fornire un concetto di *stato* (pensiamo ad una fotografia di una determinata situazione: dobbiamo dire se i nostri personaggi stanno sulla riva sinistra, destra, o sulla barca). Lo stato cambia a seconda del tempo (time) e verrà descritto da un predicato `on` con tre argomenti: il tempo corrente, il personaggio e la sua posizione.

All'inizio ci sarà lo stato:

```
on(1,goat,left).
on(1,cabbage,left).
on(1,wolf,left).
on(1,man,left).
```

Alla fine vorremmo invece avere tutti e quattro a destra. In generale, ci sarà il predicato (funzione) che stabilisce che in ogni tempo ogni personaggio sta in esattamente un posto.

```
1 { on(T,O,P) : place(P) } 1 :- time(T), object(O).
```

Per ogni tempo T poniamo un po' di regole e vincoli:

```
%%% SE (goat e cabbage) OR (goat e wolf) sono nel posto P, allora  man sta in P
on(T,man,P) :- on(T,goat,P), on(T,cabbage,P), time(T), place(P).
on(T,man,P) :- on(T,goat,P), on(T,wolf,P), time(T), place(P).

%%% Effetto del muoversi in barca
on(T+2,O,right):- on(T+1,O,boat), on(T,O,left),  time(T), object(O).
on(T+2,O,left) :- on(T+1,O,boat), on(T,O,right), time(T), object(O).

%%% Se qualcuno e' in barca, ci deve essere l'uomo.
on(T,man,boat) :- on(T,O,boat), time(T), object(O).
```

```
%%% La barca contiene da 0 a 2 personaggi
0 { on(T,O,boat) : object(O) } 2 :- time(T).
```

In questo tipo di problemi giocano un ruolo chiave le cosiddetta codifica del *frame problem* (anche detto, con abuso di notaizone, "principio d'inerzia"). Ovvero, se nessuno muove un oggetto, sarà nella stessa posizione in cui era al tempo precedente:

```
:- on(T+1,O,left), on(T,O,right), time(T), object(O).
:- on(T+1,O,right), on(T,O,left), time(T), object(O).
```

Alla fine definiamo cosa sia per noi l'obiettivo (goal) e diciamo che l'obiettivo non può essere falso.

```
goal :- on(n,goat,right), on(n,cabbage,right), on(n,wolf,right), on(n,man,right).
:- not goal.
```

Eseguite il programma facendo crescere n nella chiamata e vedete quando (e quante) le soluzioni vengono calcolate. Le avevate calcolate a mente?