Answer Set Programming (ASP) e la codifica dei rompicapi

Agostino Dovier
Dip. di Scienze Matematiche, Informatiche e Fisiche
Univ. degli Studi di Udine

Maggio 2016

Contents

1 Installazione

Il materiale presentato è abbastanza generale da poter essere utilizzato con qualunque ASP solver (smodels, cmodels, DLV ecc) incluso l'ambiente di sviluppo ASPIDE.

Per uniformità 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`.

Aggiungiamo qualche dettaglio in più per facilitare l'installazione.

Una volta scaricato, ed eventualmente decompresso (unzippato) posizionate la cartella nella "zona" dove avete installato gli altri linguaggi di programmazione (o i programmi in genere). A quel punto dobbiamo far conoscere quel PATH (tipo `C:\ProgramFiles\clingo-v4.4\`) al vostro computer.

Da sistemi windows, pannello di controllo, sistema e sicurezza, sistema, impostazioni di sistema avanzate, cliccate su "variabili d'ambiente", selezionate "PATH", cliccate modifica, a quel punto aggiungete (scrivendo o copia incollandolo) il path alla fine del path esistente, concludendo coon un ";" alla fine. Dunque salvate. Ora, ogni volta che aprite una finestra "terminale" (com "cmd") cliccando clingo sarà correttamente lanciata l'esecuzione del solver.

Da sistemi MACOS, nella "home" si può aggiungere l'alias `clingo="PATH/clingo"` nel file ".profile". Se non c'e' tale file, va prima creato.

Per scrivere i programmi ci serve un editor. In linea di principio ogni editor va bene, ma sono da preferire quelli "snelli" per scrivere file puramente testuali (emacs, aquamacs, winedt, Geany, etc.). In particolare Geany funziona in tutti i sistemi operativi ed scaricabile da `http://www.geany.org/`

Per verificare che tutto funzioni, create con l'editor un file dal nome `pippo.lp` e scriveteci dentro il programma

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

(tulle le lettere sono minuscole. Non ci sono spazi tra i due punti ed il meno). Lo salvate dove preferite (ad esempio in `C://ASP`). Ora aprite un terminale. Con una serie opportuna di comandi "CD" (change directory) posizionatevi nella cartella dove avete salvato il file (ad esempio `cd C:`, seguita da `cd ASP`). Dunque scrivete:

```
clingo pippo.lp
```

(e battete return). Dovrebbero apparire alcune righe tra cui quelle che scopriremo essere le più interessanti per noi, ovvero

```
Answer: 1
p(a)
```

Se ora invece scrivete

```
clingo pippo.lp 0
```

Dovrebbero apparire due righe più di prima; in particolare:

```
Answer: 2
p(b)
```

Se succede, siete pronti a lavorare con Answer Set Programming.

2 Programmi ASP

Iniziamo lo studio del linguaggio che sarà utilizzato fornendone la sintassi rigorosa che deriva dalla logica matematica.

2.1 Definizioni principali

Una *formula atomica (o, brevemente, 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. 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 chiarire, 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*; $A_1, \ldots, A_n$, **not** $B_1, \ldots,$ **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.

2.2 Significato di fatti, regole, e vincoli

Il significato dei fatti è quello di asserire informazioni in modo esplicito. Nell'esempio 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). Non si devono dichiarare i tipi delle variabili.

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. In logica si scriverebbe:

$$(\forall X)(\texttt{deo}(X) \rightarrow \texttt{personaggio}(X))$$

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). E, in ogni modo, non vogliamo che se ne debba curare il programmatore.

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 (e non solo, ci sarebbe anche qualche problema di correttezza sul significato del `not`, ma non è il momento di entrare in questi ulteriori dettagli) 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 regole (implicazioni). L'obiettivo è calcolare l'insieme delle *conseguenze logiche*, ovvero degli atomi che sono veri in qualunque "modello logico ragionevole" del programma.[1]

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*

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

(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.

Vale la pena ricordare che l'implicazione

$$A \leftarrow B$$

è equivalente alla disgiunzione $A \vee \neg B$ ovvero l'unico caso che la rende falsa è quello in cui B è vera ed A è falsa.

Inoltre si ricorda che $\neg\exists X\, \neg\Phi(X)$ equivale a $\forall X \Phi(X)$. Sfrutteremo questa equivalenza tra formule quantificate per codificarle usando i vincoli. Ad esempio per dire che per ogni coppia di coordinate (X_1, Y_1), (X_2, Y_2) vale la proprietà $\texttt{vicino}(X_1, Y_1, X_2, Y_2)$ metteremo il vincolo:
$\texttt{:- coordinata}(X_1, Y_1)\texttt{, coordinata}(X_2, Y_2)\texttt{, not vicino}(X_1, Y_1, X_2, Y_2)\texttt{.}$

2.3 Abbreviazioni utili

Completiamo questa breve introduzione con alcune comode abbreviazioni sintattiche. In realtà tutte si possono realizzare con i programmi visti finora, ma il loro uso semplifica (abbrevia) il codice.

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:

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

dove `dominio` 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 $\texttt{dominio}(X, Y)$) 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:

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

```
1 {fun(X,Y): cod(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:

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

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

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

E in luogo dei fatti (numerici o meno)

```
wal(a).   wal(5).   wal(r).
```

Possiamo scrivere

```
wal(a;5;r).
```

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.

2.4 Aggregati

ASP supporta l'utilizzo dei cosiddetti *aggregati* che permettono di definire in modo intensionale funzioni associate ai valori degli argomenti degli atomi nell'answer set. Ci sono due principali aggregati: `count` e `sum`. Gli aggregati sono uno strumento di programmazione molto espressivo; la loro sintassi non è ancora "in evoluzione" (si consiglia di leggere il manuale più recente).

Introduciamo gli aggregati con degli esempi:

```
dom(1..3).
p(1,1).  p(2,2).   p(3,3).
somma(S) :- S = #sum { Y : dom(X), p(X,Y) }.
```

L'output è `somma(6)`. Invece in

```
dom(1..3).
p(1,1).  p(2,2).   p(3,2).
somma(S) :- S = #sum { Y : dom(X), p(X,Y) }.
```

L'output è `somma(3)` (ci aspetteremmo 5 invece, vero?)

L'attuale interpretazione di questi aggregati considera insiemi di valori, non multi-insiemi, ovvero le ripetizioni non contano. Per avere quello che immaginiamo dobbiamo scrivere:

```
dom(1..3).
p(1,1).  p(2,2).   p(3,2).
somma(S) :- S = #sum { Y,X : dom(X), p(X,Y) }.
```

L'output è `somma(5)`.

`count` ha sintassi (e problemi annessi) simili, ma conta il numero di atomi che soddisfano una condizione. Ad esempio:

```
dom(1..3).
p(1,1).  p(2,2).   p(3,3).
conta(S) :- S = #count{ p(X,Y):dom(X),p(X,Y),Y > 2 }.
```

L'output è `conta(1)`.

3 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.

3.1 ASP come Database deduttivo

Iniziamo la nostra avventura nell'ASP codificando un albero genealogico e definendo alcuni predicati su di esso. Immaginate la situazione di una famiglia sia la seguente:

```
padre(alberto,bruno).
padre(alberto,camilla).
padre(alberto,dorothy).
padre(bruno,emma).
padre(bruno,francesco).
padre(francesco,giacomo).
padre(francesco,herbert).
madre(irene,bruno).
madre(irene,camilla).
madre(jennifer,dorothy).
madre(katya,emma).
madre(katya,francesco).
madre(laura,giacomo).
madre(maria,herbert).
```

Iniziamo a definire su questo database alcune relazioni, su cui ragionare per comprendere la definizione logica:

```
genitore(X,Y) :- padre(X,Y).
genitore(X,Y) :- madre(X,Y).
nonno(X,Y) :- padre(X,Z), genitore(Z,Y).
nonna(X,Y) :- madre(X,Z), genitore(Z,Y).
fratello(X,Y) :- padre(P,X), padre(P,Y), madre(M,X), madre(M,Y), X != Y.
fratellastro(X,Y) :- genitore(A,X),genitore(A,Y), X != Y, not fratello(X,Y).
cambiocompagna(X) :- padre(X,Y), padre(X,Z),  madre(A,Y), not madre(A,Z).
cambiocompagno(X) :- madre(X,Y), madre(X,Z),  padre(X,Y), not padre(X,Z).
```

In particolare, le prime due dicono che `X` è genitore di `Y` se (e solo se, in un certo senso) `X` è padre di `Y` oppure `X` è madre di `Y`. Meditate sulle altre come esercizio.

Possiamo anche definire relazioni in modo ricorsivo (`antenat` non distingue il genere):

```
antenat(X,Y) :- genitore(X,Y).
antenat(X,Y) :- genitore(X,Z), antenat(Z,Y).
```

e per finire cercare il capostipite maschio.

```
figlio(Y) :- genitore(X,Y).
capostipitemaschio(X) :- padre(X,Y), not figlio(X).
```

Esercizio 3.1 *1. Si provi il programma sopra usando clingo.*

2. *Si scriva poi su un file diverso il proprio albero genealogico e si definisca il predicato che calcola la relazione cugino (di primo grado) e cugino di secondo grado e si determinino (lanciando l'esecuzione del programma) tutti i propri cugini di primo grado e di secondo. Cosa succede se c'è una omonimia (per esempio se Mario Rossi aveva come bisonno Mario Rossi)? Provate a metterna una "finta" se non ci fosse già nel vostro albero genealogico (spesso c'è).*

3. *Si scriva un predicato che dice in che classe è un individuo (immaginatevi 15 classi, dalla primaA alla quintaC, mettete pure nomi di fantasia). Dunque avrete dei fatti del tipo* `inclasse(mario,primaA).` `inclasse(berto,secondaC).` *Immaginate che alcune classi siano vicine (per esempio:*
   `vicino(primaA,primaB)`, `vicino(primaB,primaC)` *—fatevi un disegno per copiare la topologia. Potete immaginare che la seconda dell'argomento sia a destra della prima). Definite un predicato che dice se due classi sono o vicine o a distanza 2 tra loro (ovvero ce n'è una in mezzo). Definite un predicato che dice se due studenti sono in classi vicine o a distanza 2 tra loro. Trovate le coppie di suddette classi e studenti lanciando l'esecuzione.*

Elaborate l'esempio usando dei fatti del tipo: `prima(primaA). prima(primaB). seconda(secondaA).` *etc Definite il predicato* `inprima` *che dovrà essere vero se e solo se un alunno è in prima o meno (similmente per le seconde, terze etc). Definite ora il predicato che dati due alunni deve stabilire se hanno esattamente un anno di differenza (supponiamo non ci siano bocciature).*

3.2 Un quadrato magico di lato 3

Vogliamo codificare il problema del quadrato magico, ovvero riempire un quadrato (per ora) 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. Introdurremo poi ulteriori restrizioni.

riga1	riga1	riga1
riga2	riga2	riga2
riga3	riga3	riga3

col1	col2	col3
col1	col2	col3
col1	col2	col3

diag1		diag2
	diag1/2	
diag2		diag1

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 anticipato in Sezione 2.3 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).
```

Per ogni valore di X (colonne) e per ogni valore di Y (righe), nella cella (X, Y) metto esattamente un valore, che è un numero dall'1 al 9.

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. Nella prossima sezione generalizzeremo la cosa a quadrati di dimensione maggiore (e otterremo stranamente, un programma più semplice):

```
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), lato(X2), X1 < X2,  sum_col(X1,V1), sum_col(X2,V2), V1 != V2.
% Tutte le righe uguali
diff_sum :- lato(X1), lato(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 1 (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), lato(X2), lato(Y1), X1<X2, valore(V), magic(X1,Y1,V), magic(X2,Y1,V).
% diagonale 1
:- lato(X1), lato(X2), X1<X2, valore(V), magic(X1,X1,V), magic(X2,X2,V).
% diagonale 2
:- lato(X1), lato(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), lato(X2), lato(Y1), lato(Y2), Y1<Y2,
               valore(V), magic(X1,Y1,V), magic(X2,Y2,V).
uguali :-  lato(X1), lato(X2), lato(Y1), lato(Y2), X1<X2,
               valore(V), magic(X1,Y1,V), magic(X2,Y2,V).
:- uguali.
```

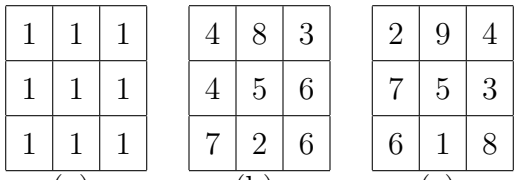

Figure 1: Tre soluzioni per il quadrato magico: senza vincoli di differenza (a), con vincoli su ogni riga e diagonale (b), con il vincolo di differenza di ogni coppia di valori (c)

3.3 Un quadrato magico di lato n generico

```
lato(1..n).
valore(1..n*n).
diag(1..2).
magicval(((n*n)*(n*n+1))/(2*n)).

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

% Use aggregates for computing the sum in a compact way
sum_cols(X, S ) :- S = #sum{ V : magic(X,L,V), lato(L)}, lato(X).
sum_rows(Y, S ) :- S = #sum{ V : magic(L,Y,V), lato(L)}, lato(Y).
sum_diag(1, S ) :- S = #sum{ V : magic(L,L,V), lato(L)}.
sum_diag(2, S ) :- S = #sum{ V : magic(L,n-L+1,V), lato(L)}.

% It cannot happen that for one column the sum is wrong (for all lato(X))
:- lato(X), sum_cols(X,V), magicval(T), V != T.
% It cannot happen that for one row the sum is wrong (for all lato(X))
  :- lato(X), sum_rows(X,V), magicval(T), V != T.
% It cannot happen that for one diagonal the sum is wrong (for all diag(D))
:- diag(D), sum_diag(D,V), magicval(T), V != T.

#show magic/3.
```

Nel caso uno immaginasse che un programma così elegante sia inefficiente rispetto alla programmazione "tradizionale" si veda nella figura 2 i confronti con i tempi della computazione in ASP con un programma in C per lo stesso problema. Il sorgente del programma C si trova in Appendice.

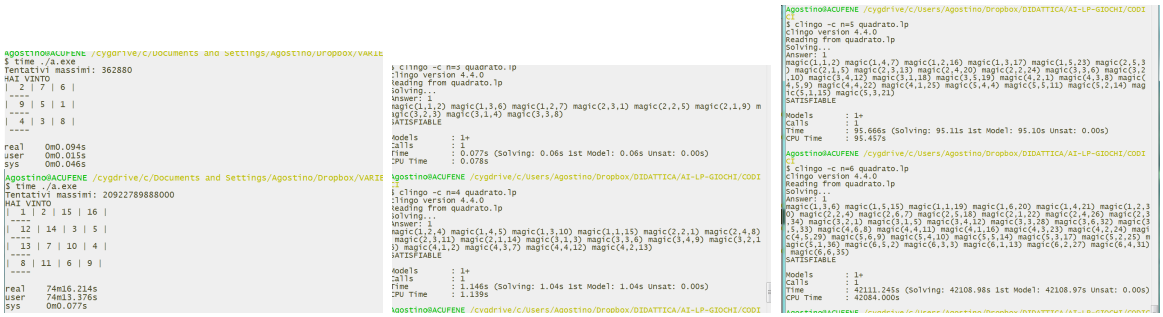

Figure 2: Da sinistra a destra, i tempi del programma C per risolvere le 3 regine, e i tempi dell'esecuzione di clingo sul programma ASP per $n = 3$ (in centro) e $n = 4$ (a destra)

3.4 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. 3).

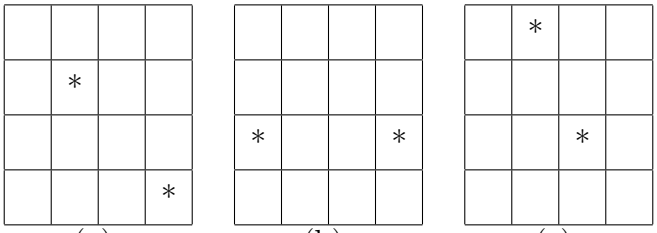

Figure 3: Regine che si attaccano (a), (b), due regine che non si attaccano (c).

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), coord(Y1), coord(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 diagonale uso un vincolo

```
:-  coord(X1), coord(X2),coord(Y1), coord(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.`)

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

Figure 4: Visitare un quadrato "a cavallo". La passeggiata si è interrotta al passo 13: non ho più nuove celle raggiungibili. Non è una soluzione.

3.5 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 4).

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), passi(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), passi(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 $n = 4$?

3.6 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), coord(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), coord(Y), coord(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!

3.7 Hidato

Lo scopo del gioco Hidato (anche chiamato Hidoku) è quello di riempire una griglia di n celle con numeri compresi fra 1 ed n, in modo che numeri consecutivi si trovino in celle adiacenti orizzontalmente, verticalmente o diagonalmente. Inizialmente alcune delle celle sono già riempite con dei numeri pre-inseriti. Inoltre, il valore massimo e valore minimo sono sempre presenti ed evidenziati (Figura 5).[3]

Rappresentiamo l'istanza con una matrice rettangolare che contenga tutte le celle; quelle non ammesse nel gioco vengono fornite in input col valore 0:

```
number(1..40).
cell(1..8,1..8).

%%% Celle con dati

matrix(1, 2, 33). matrix(1, 3, 35).
matrix(2, 3, 24). matrix(2, 4, 22).
matrix(3, 4, 21).
matrix(4, 2, 26). matrix(4, 4, 13). matrix(4, 5, 40). matrix(4, 6, 11).
matrix(5, 1, 27). matrix(5, 5, 9).  matrix(5, 7, 1).
```

[3]Ringrazio lo studente Alessio Gonella per le figure e una prima codifica del gioco.

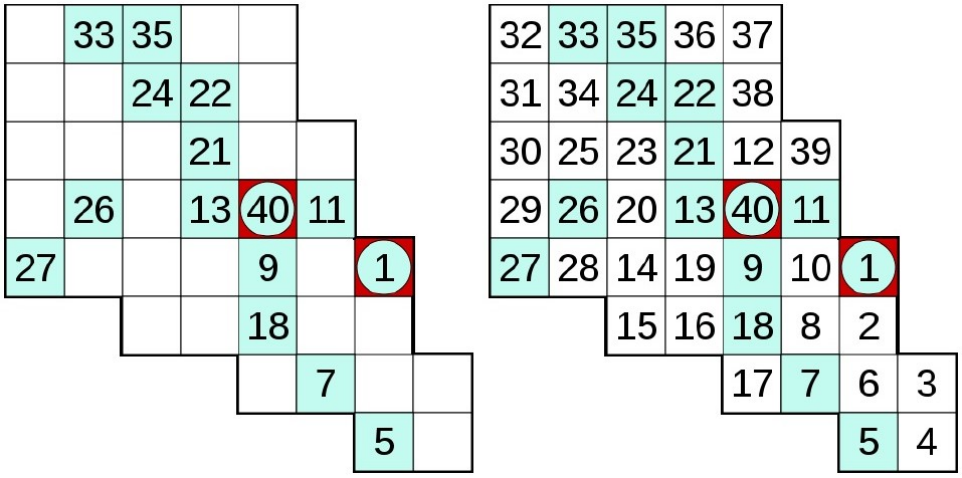

Figure 5: Un'istanza di Hidato. A sinistra lo schema iniziale, a destra la sua soluzione.

```
matrix(6, 5, 18).
matrix(7, 6, 7).
matrix(8, 7, 5).

%%% Celle non ammesse

matrix(1,6,0). matrix(1,7,0). matrix(1,8,0).
matrix(2,6,0). matrix(2,7,0). matrix(2,8,0).
matrix(3,7,0). matrix(3,8,0).
matrix(4,7,0). matrix(4,8,0).
matrix(5,8,0).
matrix(6,1,0). matrix(6,2,0). matrix(6,8,0).
matrix(7,1,0). matrix(7,2,0). matrix(7,3,0). matrix(7,4,0).
matrix(8,1,0). matrix(8,2,0). matrix(8,3,0).
matrix(8,4,0). matrix(8,5,0). matrix(8,6,0).
```

Il codice ASP che modella il gioco è dunque il seguente:

```
% Celle vere del gioco

goodcell(X,Y) :- cell(X,Y), not matrix(X,Y,0).

% ogni casella "buona" deve contenere esattamente un numero

1 { fin_matrix(X, Y, Num) : number(Num) } 1 :- goodcell(X,Y).
```

```
% Tutti i numeri (nelle caselle "buone") devono essere diversi

1 { fin_matrix(X, Y, Num) : goodcell(X,Y) } 1 :- number(Num).

% L'input e' preservato

fin_matrix(X, Y, Num):- goodcell(X,Y),matrix(X,Y,Num).

% (X1, Y1) e (X2, Y2) sono vicine:

neighbour(X1, Y1, X2, Y2) :-
        goodcell(X,Y),goodcell(X1,Y1),
        offset(DX), offset(DY),
        X2 = X1 + DX,
        Y2 = Y1 + DY.

% Offset di +/-1 per il vicinato

offset(-1;0;1).

% Non e' possibile che due numeri successivi si trovino in due caselle non adiacenti.
% L'unica eccezione alla regola e' il massimo (che non ha successori).

:- number(Num), number(Num+1), goodcell(X1,Y1), goodcell(X2,Y2),
   fin_matrix(X1, Y1, Num), fin_matrix(X2, Y2, Num+1),
   not neighbour(X1, Y1, X2, Y2).

#show fin_matrix/3.
```

3.8 Quadrato "Talisman"

Il problema è il seguente: dati due numeri n e k, inserire una e una sola volta i numeri da 1 a n^2 in un quadrato di lato n in modo tale che, per ogni cella, la differenza tra il numero presente in quella cella e ognuno dei numeri presenti nelle celle adiacenti sia sempre maggiore o uguale a k (in valore assoluto).

Ad esempio, con $n = 4$ e $k = 3$ una soluzione può essere:

6	3	15	2
13	10	7	11
16	4	1	14
9	12	8	5

La codifica in ASP del problema è molto semplice:[4] L'esecuzione va effettuata assegnando i valori alle costanti n e k.

```
size(1..n).
val(1..n*n).

% per ogni riga e colonna (quindi cella) esiste un unico valore n
1 { x(Row, Col, N) : val(N) } 1 :- size(Row), size(Col).

% vincolo di alldifferent per i valori
1 { x(Row, Col, N) : size(Row), size(Col) } 1 :- val(N).

:- x(Row, Col, Val), x(Row, Col+1, Val1), |Val-Val1|<k.
:- x(Row, Col, Val), x(Row+1, Col+1, Val1), |Val-Val1|<k.
:- x(Row, Col, Val), x(Row+1, Col, Val1), |Val-Val1|<k.
:- x(Row, Col, Val), x(Row+1, Col-1, Val1), |Val-Val1|<k.
:- x(Row, Col, Val), x(Row, Col-1, Val1), |Val-Val1|<k.
:- x(Row, Col, Val), x(Row-1, Col-1, Val1), |Val-Val1|<k.
:- x(Row, Col, Val), x(Row-1, Col, Val1), |Val-Val1|<k.
:- x(Row, Col, Val), x(Row-1, Col+1, Val1), |Val-Val1|<k.

#show x/3.
```

Sono necessaro tutti gli 8 vincoli scritti sopra?

3.9 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).
```

[4] Ringrazio lo studente Riccardo Zucchetto per la codifica del gioco.

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?

3.10 Qualche ulteriore esercizio

Per alcuni di questi esercizi trovate soluzioni ed esempi nei lucidi delle lezioni di autunno 2015, disponibili da `https://users.dimi.uniud.it/~agostino.dovier/DID/lpandgames.html`.

1. Le tre botti. Ci sono tre botti di capacit 12 (in generale n pari), 7 e 5 ($\frac{n}{2}+1$ e $\frac{n}{2}-1$). Allinizio la botte pi capace piena di vino e le altre sono vuote. Si vuol raggiungere una conformazione in cui la pi piccola vuota e le altre due contengono esattamente la stessa quantit. Le sole azioni permesse sono di svuotare una botte in unaltra (non permesso di misurare i flussi o laltezza del vino o pesare le botti).

2. Le due taniche (dal film Die Hard). Riportiamo la definizione originale: *You have a 3 and a 5 litre water container, each container has no markings except for that which gives you it's total volume. You also have a running tap. You must use the containers and the tap in such away as to exactly measure out 4 litres of water. How is this done? Can you generalise the form of your answer?*

3. La torre di Hanoi. Si tratta di un gioco molto famoso. Ci sono tre pioli e n dischi di raggio (proporzionale a) $1, 2, \ldots, n$. La mossa ammessa è quella di spostare il disco che sta sopra a una pila di dischi in un altro piolo, a patto che venga appoggiato o sul pavimento o sopra un disco più grande. L'obiettivo classico è dato lo stato iniziale in cui tutti i dischi stanno nel piolo 1, il più grande sotto e via via a calare gli altri (il più piccolo sopra), riuscire a spostare tutti i dischi dal piolo 1 al piolo 2 (usando anche il piolo 3 "per appoggio"). Nella codifica siate generali nel senso che il programma dovrebbe saper portare una generica situazione (stato) in un altra generica situazione.

4. II Sam Lloyd's puzzle. Si tratta del classico giochino che (di solito) consta di un quadrato di lato 4 con sopra 15 tesserine numerate dall'1 al 15 (o rappresentanti disegni da ricostruire) e la mossa possibile è quella di muovere le tessere vicine allo spazio vuoto.

5. I codici di Hamming. La distanza di Hamming tra due n-uple (di 0 e 1) si ottiene mettendo le due n-uple una sotto l'altra e contando i punti in cui sono diverse. Ad esempio, la distanza tra $(0,0,0,1,1,1)$ e $(0,0,1,0,1,0)$ è 3. Il problema è il seguente, dati n, k, d (dall'input) trovare k n-uple distinte tali che prese a due a due la loro distanza di Hamming è almeno d.

6. Scheduling di un torneo di calcetto. Si tratta di organizzare un torneo tra un certo numero di squadre con una serie di vincoli (vincoli di lavoro dei giocatori, vincoli della struttura ospitante, distanza minima rta una partita e l'altra etc etc). Immaginatevi la situazione e provate a codificare. Nel caso, trovate una soluzione sufficientemente generale nei lucidi nel sito suddetto.

7. Ci sono 2013 carte che possono essere a faccia in su o a faccia in giù. All'inizio tutte le carte sono a faccia in giù. Una mossa consiste nel prendere una carta a faccia in giù, girarla e girare le 50 carte a destra (indipendetemente dalla posizione—non è possibile effettuare una mossa se non ci sono 50 carte a destra). Quando tutte le carte saranno a faccia in su, si vince la partita Stabilire una strategia vincente. [Suggerimento: si scriva un problema parametrico su n carte. Potrebbe darsi che 50 sia un numero troppo grande per i l grounding, nel caso risolvetelo fino all'n massimo che riuscite]

8. Ci sono 12 monete uguali in tutto ma una di esse sappiamo essere falsa. Quella falsa pesa differentemente dalle altre (ma non sappiamo se sia più leggera o più pesante). Abbiamo a disposizione una bilancia a due piatti (che non misura il peso ma ci fa capire in quale dei due piatti c'è un peso maggiore e in quale minore). Come facciamo ad identificare univocamente la moneta falsa in 3 pesate?

4 Osservazioni generali

In questa sezione collezioniamo alcune osservazioni generali circa la riproducibilità in ASP di esercizi di programmazione che vengono usualmente spiegati con altri linguaggi di programmazione. Ricordiamo prima che, tecnicamente, ASP non è un linguaggio di programmazione nel senso esteso del termine. C'è prima una fase detta di grounding che produce un file talvolta volte enorme ma di lunghezza finita e poi un ragionamento su questo file, ragionamento che conduce sempre a terminazione. Per dirla tutta, un programmino C/Java che entra in loop in esecuzione non sarà riproducibile in ASP. Tuttavia, possiamo riprodurre agevolmente la gran parte degli algoritmi "classici". E sfruttare la "bidirezionalità" delle definizioni dei predicati per avere in autoimatico le inverse delle funzioni date.

4.1 Algoritmo di Euclide

Nel seguente codice è stato definito l'algoritmo di Euclide che calcola il massimo comun divisore tra due numeri. Come dicevamo, c'è bisogno di "limitare" i numeri per il grounding e dunque, dati i due valori di input (a e b) diciamo che ci interessano solo i numeri naturali da 0 al massimo tra a e b. A questo punto c'è la definizione classica (ricorsiva) dell'algortimo, ma in ogni regola "limitiamo" i valori delle variabili di input con quelle ammesse.

Infine, il predicato `result` ci selezionerà il valore di `eucl`(A, B, M) desiderato.

```
num(0..A) :- A = #max{a;b}.

eucl(A,B,M) :-
    num(A), num(B),
```

```
    A < B,
    eucl(B,A,M).

eucl(A,0,A) :-
    num(A).

eucl(A,B,M) :-
    num(A),num(B),
    A > B, B > 0,
    eucl(B,(A \ B),M).

result(M) :- eucl(a,b,M).

pairprime(A,B) :- num(A), num(B), A > 1, B > 1, eucl(A,B,1).
```

Il predicato `pairprime` definito sulla base di `eucl` ci mostra tutte le coppie di numeri (tra quelli calcolati) che sono primi tra loro (ho escluso quelle con dentro degli 1, che ci sarebbero tutte e sono poco interessanti).

Esercizio 4.1 *Basandosi sull'algoritmo appena visto, provate a definire il predicato che calcola il fattoriale (ad esempio il predicato* `fatt` *di due argomenti, nel primo n e nel secondo ci dovrà essere n!). Si definisca dunque un predicato che, dato un numero b che sia un fattoriale (1, 2, 6, 24, 120, ...) restituisca il numero n tale che* $n! = b$. *O lo avevamo già definito?*

Si faccia lo stesso con la funzione di Fibonacci.

4.2 Il lancio di dadi

Immaginiamo di voler modellare il lancio di due dadi ed ottenere tutte le combinazioni che danno 7. Iniziamo con questo programma.

```
dado(1..6).
lancio(X,Y):- dado(X), dado(Y).
```

Eseguendo clingo su di lui si ottiene:

```
Answer: 1
dado(1) dado(2) dado(3) dado(4) dado(5) dado(6)
lancio(1,1) lancio(2,1) lancio(3,1) lancio(4,1) lancio(5,1) lancio(6,1)
lancio(1,2) lancio(2,2) lancio(3,2) lancio(4,2) lancio(5,2) lancio(6,2)
lancio(1,3) lancio(2,3) lancio(3,3) lancio(4,3) lancio(5,3) lancio(6,3)
lancio(1,4) lancio(2,4) lancio(3,4) lancio(4,4) lancio(5,4) lancio(6,4)
lancio(1,5) lancio(2,5) lancio(3,5) lancio(4,5) lancio(5,5) lancio(6,5)
lancio(1,6) lancio(2,6) lancio(3,6) lancio(4,6) lancio(5,6) lancio(6,6)
SATISFIABLE
```

A questo punto proviamo ad *aggiungere* il vincolo:

```
:- lancio(X,Y),dado(X), dado(Y), X + Y != 7.
```

Ovvero ad escludere tutte le combinazioni di lanci che non danno 7.

Anzichè ottenere il risultato voluto il risolutore fornisce:

```
UNSATISFIABLE
```

Cos'è successo? Abbiamo sottovalutato la forza del sillogismo. La regola:

```
lancio(X,Y):- dado(X), dado(Y).
```

ci dice che per ogni valore possibile per il dado X e per ogni balore possibile per il dado Y il predicato `lancio` deve essere vero per la coppia (X, Y). E dunque se con il successivo vincolo vogliamo limitare questo comportamento raggiungiamo una situazione di insoddisfacibilità.

La soluzione di non usare il vincolo, ma una regola più "mirata".

```
dado(1..6).
lancio(X,Y):- dado(X), dado(Y), X + Y = 7.
```

che restituisce:

```
Answer: 1
lancio(6,1) lancio(5,2) lancio(4,3) lancio(3,4) lancio(2,5) lancio(1,6)
SATISFIABLE
```

Per vedere un altro esempio dello stesso tipo, consideriamo una variante del problema, ovvero il lancio di una moneta (che è come un dado a due facce)

```
lato(0).  lato(1).
```

Se scriviamo:

```
lancio(X) :- lato(X).  (*)
```

Allora in ogni modello del programma devono valere sia `lancio(0)` che `lancio(1)`.

Se aggiungo

```
:- lancio(0).
```

(ovvero non e' possibile che il lancio faccia 0, che equivarrebbe a lasciare le soluzioni in cui il lancio e' 1) la risposta sara' "insoddisfacibile."

Se guardiamo la teoria nel suo insieme, "lancio(1)" deve essere vero per le implicazioni (*) sopra; ma "lancio(0)" non puo' essere vero per il vincolo e dunque il tutto e' insoddisfacibile.

Se omettiamo la regola (*) dal programma? Diventa soddisfacibile, ma non ci da' ancora `lancio(1)`. Nell'idea del modello stabile, il significato formale dell'operato del solver clingo, una cosa che non sia dimostrabile e' falsa. E dunque, poichè non c'è nessuna ragione affinchè `lancio(1)` sia vera, viene considerata falsa. Per poter essere vera deve quantomeno coimparire nella testa di una regola.

La soluzione nondeterministica al problema è la seguente.

```
{lancio(X)} :- lato(X)   (*1)
```

del tutto equivalente a:

```
0{lancio(X)}1 :- lato(X).  (*2)
```

Dice che per ogni X per cui vale lato X (dunque per $X = 0$ e $X = 1$) il programma è giustificato sia ad avere che a non avere una conseguenza del tipo `lancio`(X).

Se mettiamo al posto di (*) una qualunque tra (*1) o (*2) ed eseguiamo diventa soddisfacibile. Ma ancora non da' lancio(1). Se lanciamo con l'opzione "0" (tutte le soluzioni) vediamo che la seconda soluzione e' quella che vogliamo. Come dicevo sopra, e' del tutto giustificato che dato lato(X), lancio(X) ci sia oppure non ci sia. Lui prova prima la soluzione in cui non c'e'. Con l'opzione 0 trova tutte le soluzioni. Se vogliamo trovare la soluzione con piu' "lanci" allora usiamo il massimizzatore.

```
nlanci(N) :- N = #count{X:lancio(X)}.
#maximize {N:nlanci(N)}.
```

Ringraziamenti

Ringrazio i proff Paolo Benoli, Fabio Bove, Lura Candotti, Maria-Concetta Brocato, Luciano Dereani, Annalisa Nocino, Federica Tabacco, per la collaborazione alla preparazione di questo materiale.

A Il quadrato magico in C

```
#include <stdio.h>

const int n=3;
int MAGIC=15; // 15 con 3, 34 con 4
```

```
// WORKS ON MAC not on CYGWIN:
// int MAGIC = ((n*n)*((n*n)+1))/(2*n);

int check_sum(int M [n*n]){
  int i,j, temp;
  int correct=1;

  // ROWS
  for(i=0;i<n;i++){
    temp=0;
    for(j=0;j<n;j++)
          temp=temp+M[n*i+j];
    if(temp != MAGIC)
       correct=0;
  }

  // COLS
  if (correct){
    for(j=0;j<n;j++){
      temp=0;
      for(i=0;i<n;i++)
          temp=temp+ M[n*i+j];
      if(temp != MAGIC)
correct=0;
    }
  }

  //DIAG 1
  if (correct){
      temp=0;
      for(i=0;i<n;i++)
          temp=temp+ M[n*i+i];
      if(temp != MAGIC)
correct=0;
  }

   // DIAG2
  if (correct){
      temp=0;
      for(i=0;i<n;i++)
          temp=temp+ M[n*i+n-i-1];
      if(temp != MAGIC)
```

```
correct=0;
  }

return correct;
}


// Dijkstra
// A Discipline of Programming, Prentice-Hall, 1997
// PAG 71

void increment(int M [n*n]){
    int i,j,t;

    i = n*n - 1;
    while (M[i-1] > M[i]) i--;

    j = n*n;
    while (M[j-1] <= M[i-1]) j--;

    // swap values at positions (i-1) and (j-1)
    t = M[i-1];
    M[i-1] = M[j-1];
    M[j-1] = t;

    i++;
    j = n*n;
    while (i < j) {
       // swap values at positions (i-1) and (j-1)
       t = M[i-1];
       M[i-1] = M[j-1];
       M[j-1] = t;
       i++;
       j--;
    }
}


long int fact(int m){
  int i;
  long int t=1;
  for(i=1;i <=m; i++) t=t*i ;
```

```
  return t;
}

void stampamat(int quad [n*n]){
   int i, j;

   for(i=0;i<n;i++){
     printf("| ");
     for(j=0;j<n;j++)
          printf(" %d |",quad[i*n+j]);
     printf("\n ---- \n");
   }
}


int main(){

  int i=0, j=0, found;
  long int max;
  int quad [n*n];

// INIT
for(i=0;i<n;i++)
  for(j=0;j<n;j++)
     quad[i*n+j] = n*i+j+1;

 found = 0;
 max   = fact(n*n);
 printf("Tentativi massimi: %ld\n",max);

 while( !found &&  max>0) {
    if (check_sum(quad))
       found = 1;
    else {
      increment(quad);
      max--;
    }
 }

 if( found) {
   printf("HAI VINTO \n");
   stampamat(quad);
```

```
 }
 else printf("HAI PERSO \n");

return 0;
}
```