AUTOMATED REASONING

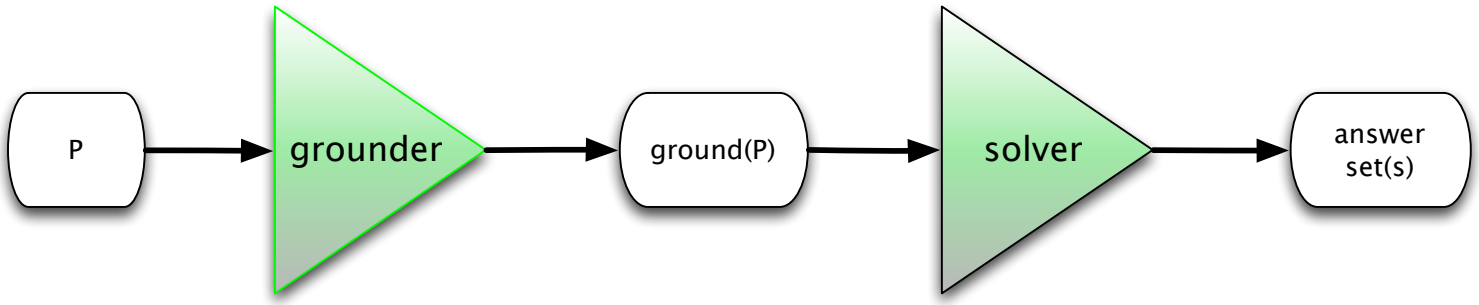
Agostino Dovier

Università di Udine
CLPLAB

Udine, November 2016

ASP SOLVERS

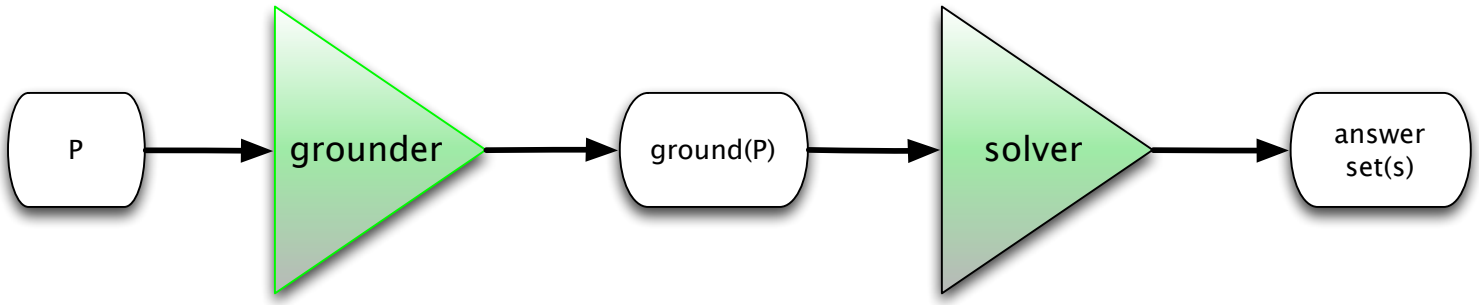
- We have seen that detecting if P, ground, has a stable model is NP complete
- What happens if P is not ground?



ground(P) should be finite.
To ensure that, P must fulfill a set of requirements (range restrictions).

This reminds the Minizinc $\mapsto$ Flatzinc $\mapsto$ Constraint Solving pipeline.

There are two exceptions (Asperix and GASP [DDPR09]) that mix grounding and search with some advantages and many drawbacks.



ground(P) should be finite.
To ensure that, P must fulfill a set of requirements (range restrictions).

This reminds the Minizinc $\mapsto$ Flatzinc $\mapsto$ Constraint Solving pipeline.

There are two exceptions (Asperix and GASP [DDPR09]) that mix grounding and search with some advantages and many drawbacks.

- What is the size of $ground(P)$? What is the complexity for computing it?
- If H_P is infinite, $ground(P)$ will be infinite. No way
- Let us assume that H_P be finite (let $|H_P| = c$).
- Let $r_1, \ldots, r_n$ be the clauses of P ($n \leq |P|$), and let $\alpha_1, \ldots, \alpha_n$ be the number of the variables occurring in them (α_i is the number of different variables in the clause i)
- Then

$$|ground(P)| = \sum_{i=1}^{n} c^{\alpha_i}$$

- If $k = \max_i\{\alpha_i\}$, then

$$|ground(P)| \leq nc^k$$

- What is the size of *ground*(P)? What is the complexity for computing it?
- If H_P is infinite, *ground*(P) will be infinite. No way
- Let us assume that H_P be finite (let $|H_P| = c$).
- Let $r_1, \ldots, r_n$ be the clauses of P ($n \leq |P|$), and let $\alpha_1, \ldots, \alpha_n$ be the number of the variables occurring in them (α_i is the number of different variables in the clause i)
- Then

$$|ground(P)| = \sum_{i=1}^{n} c^{\alpha_i}$$

- If $k = \max_i\{\alpha_i\}$, then

$$|ground(P)| \leq nc^k$$

- What is the size of *ground*(P)? What is the complexity for computing it?
- If H_P is infinite, *ground*(P) will be infinite. No way
- Let us assume that H_P be finite (let $|H_P| = c$).
- Let $r_1, \ldots, r_n$ be the clauses of P ($n \leq |P|$), and let $\alpha_1, \ldots, \alpha_n$ be the number of the variables occurring in them (α_i is the number of different variables in the clause i)
- Then

$$|ground(P)| = \sum_{i=1}^{n} c^{\alpha_i}$$

- If $k = \max_i\{\alpha_i\}$, then

$$|ground(P)| \leq nc^k$$

- What is the size of *ground*(P)? What is the complexity for computing it?
- If H_P is infinite, *ground*(P) will be infinite. No way
- Let us assume that H_P be finite (let $|H_P| = c$).
- Let $r_1, \ldots, r_n$ be the clauses of P ($n \leq |P|$), and let $\alpha_1, \ldots, \alpha_n$ be the number of the variables occurring in them (α_i is the number of different variables in the clause i)
- Then

$$|ground(P)| = \sum_{i=1}^{n} c^{\alpha_i}$$

- If $k = \max_i\{\alpha_i\}$, then

$$|ground(P)| \leq nc^k$$

- If $k = \max_i\{\alpha_i\}$, then

$$|ground(P)| \leq nc^k$$

- Since c can be defined implicitly (e.g., $p(1..1000)$), grounding might require exponential time w.r.t. $|P|$.
- These are not bad news . . . we can also encode problems outside NP using intervals.

- We must help the grounder in its process by limiting the scope of the variables.
- The general rule is that any variable occurring in a rule must occur as argument of a domain predicate that occur positively in its body.
- If this is the case the program is said strongly range restricted
- For instance, the following rules

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

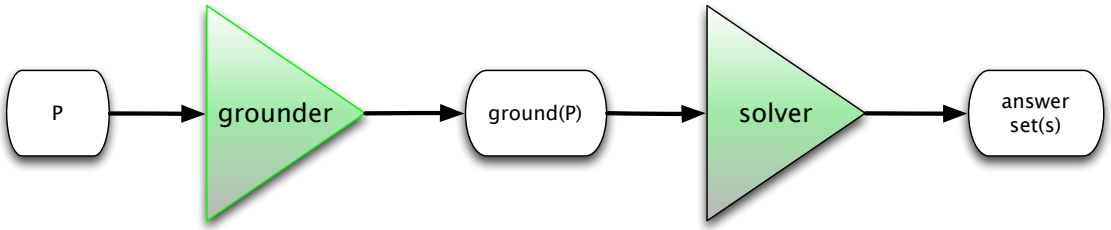
lead to not strongly range restricted programs

- If a predicate is defined extensionally, using a set of ground facts, it is a domain predicate
- If a predicate is defined by strongly range restricted definite clauses, it is a domain predicate
- I suggest to stop here with the notion of domain predicate.
- (in literature you can find a “wider” notion)

- If a predicate is defined extensionally, using a set of ground facts, it is a domain predicate
- If a predicate is defined by strongly range restricted definite clauses, it is a domain predicate
- I suggest to stop here with the notion of domain predicate.
- (in literature you can find a “wider” notion)

ASP SOLVERS

MAIN TOOLS



- Grounders (front-end) are LPARSE, DLV GROUNDER, GRINGO
- The solvers are SMODELS (the first one), CMODELS, DLV SOLVER, CLASP.
- DLV [Vienna and Rende] is equipped with an IDE (ASPIDE)
- We'll use CLINGO (= gringo+clasp) [developed in Potsdam University: the fastest tool]
- We'll analyze later how (clever) the solvers are implemented

- Assume P is written in the file `asp.lp`.
- You can run it by invoking the command

  `clingo asp.lp` n

  (or simply `clingo asp.lp` — default case)
- n indicates how many stable models you want it computes. $n = 1$ is the default. $n = 0$ means "all"
- As we'll see you can assign values to constants (e.g. `clingo -c n=15 asp.lp`)
- At the end of the file you can ask which predicate are produced explicitly in the output. E.g. `#show p/2.` requires to print the atoms based on the predicate `p` of arity 2 that holds in the answer set(s). Without this declaration all atoms are printed.

- It is rather common using intervals of (integer) values when encoding
- For instance a predicate that holds on the integer coordinates of a square of size 4 can be defined extensionally:

```
lato(1).   lato(2).   lato(3).   lato(4).
```

- Or intensionally

```
lato(1..4).
```

- You can also use a *constant* that will be instantiated at runtime (for parametric programs):

```
lato(1..n).
```

(For instance, if you want to run with `n=4`, just call:
`clingo -c n=4 nomefile.lp`)

- It is rather common using intervals of (integer) values when encoding
- For instance a predicate that holds on the integer coordinates of a square of size 4 can be defined extensionally:

```
lato(1).    lato(2).    lato(3).    lato(4).
```

- Or intensionally

```
lato(1..4).
```

- You can also use a *constant* that will be instantiated at runtime (for parametric programs):

```
lato(1..n).
```

(For instance, if you want to run with `n=4`, just call:
`clingo -c n=4 nomefile.lp`)

- It is rather common using intervals of (integer) values when encoding
- For instance a predicate that holds on the integer coordinates of a square of size 4 can be defined extensionally:

```
lato(1).   lato(2).   lato(3).   lato(4).
```

- Or intensionally

```
lato(1..4).
```

- You can also use a *constant* that will be instantiated at runtime (for parametric programs):

```
lato(1..n).
```

(For instance, if you want to run with `n=4`, just call:
`clingo -c n=4 nomefile.lp`)

- If instead you have a set of values which is not an interval, e.g.

```
primo(2).   primo(3).   primo(5).   primo(7).
```

- you can use the syntactic sugar

```
primo(2;3;5;7).
```

- Use this possibility with parsimony. I'd suggest to use it only in facts (the ";" in bodies can have unexpected results)

A *Ground cardinality constraint* can be used as a positive atom:

$$n\{\mathtt{L}_1\mathtt{;}\ldots\mathtt{;}\mathtt{L}_h\mathtt{;}\ \mathtt{not}\ \mathtt{H}_1\mathtt{;}\ldots\mathtt{;}\ \mathtt{not}\ \mathtt{H}_k\}m$$

where $\mathtt{L}_1\mathtt{,}\ldots\mathtt{,}\mathtt{L}_h\mathtt{,}\mathtt{H}_1\mathtt{,}\ldots\mathtt{,}\mathtt{H}_k$ are atoms and n and m are integer numbers (one or both can be omitted).
Assume to have a set of atoms S and a cardinality constraint C, then

$$val(C,S) = \mid S \cap \{\mathtt{L}_1\mathtt{,}\ldots\mathtt{,}\mathtt{L}_h\} \mid + (k - \mid S \cap \{\mathtt{H}_1\mathtt{,}\ldots\mathtt{,}\mathtt{H}_k\} \mid).$$

C is true in S if $n \leqslant val(C,S) \leqslant m$.

A *cardinality constraint* can be also not ground (for understanding its meaning we might think to its grounding).

A *Ground cardinality constraint* can be used as a positive atom:

$$n\{\mathtt{L}_1\mathtt{;}\ldots\mathtt{;}\mathtt{L}_h\mathtt{;}\ \mathtt{not}\ \mathtt{H}_1\mathtt{;}\ldots\mathtt{;}\ \mathtt{not}\ \mathtt{H}_k\}m$$

where $\mathtt{L}_1\mathtt{,}\ldots\mathtt{,}\mathtt{L}_h\mathtt{,}\mathtt{H}_1\mathtt{,}\ldots\mathtt{,}\mathtt{H}_k$ are atoms and n and m are integer numbers (one or both can be omitted).
Assume to have a set of atoms S and a cardinality constraint C, then

$$val(C,S) = \mid S \cap \{\mathtt{L}_1\mathtt{,}\ldots\mathtt{,}\mathtt{L}_h\} \mid + (k - \mid S \cap \{\mathtt{H}_1\mathtt{,}\ldots\mathtt{,}\mathtt{H}_k\} \mid).$$

C is true in S if $n \leqslant val(C,S) \leqslant m$.

A *cardinality constraint* can be also not ground (for understanding its meaning we might think to its grounding).

```
coord(1..2).
valore(1..3).
1 {assegna(X,Y,V) : valore(V) } 1 :- coord(X),coord(Y).
```

It states that for each point identified by its *x* and *Y* coordinates, one and only one value (valore) *V* is assigned.

`assegna` is a function. It is the same as:

```
1 {assegna(X,Y,1);assegna(X,Y,2);assegna(X,Y,3)} 1 :-
      coord(X),coord(Y).
```

which in turn it is equivalent to:

```
1 {assegna(1,1,1);assegna(1,1,2);assegna(1,1,3)} 1.
1 {assegna(1,2,1);assegna(1,2,2);assegna(1,2,3)} 1.
1 {assegna(2,1,1);assegna(2,1,2);assegna(2,1,3)} 1.
1 {assegna(2,2,1);assegna(2,2,2);assegna(2,2,3)} 1.
```

```
coord(1..2).
valore(1..3).
1 {assegna(X,Y,V) : valore(V) } 1 :- coord(X),coord(Y)
```

It states that for each point identified by its *x* and *Y* coordinates, one and only one value (valore) *V* is assigned.
`assegna` is a function. It is the same as:

```
1 {assegna(X,Y,1);assegna(X,Y,2);assegna(X,Y,3)} 1 :-
        coord(X),coord(Y).
```

which in turn it is equivalent to:

```
1 {assegna(1,1,1);assegna(1,1,2);assegna(1,1,3)} 1.
1 {assegna(1,2,1);assegna(1,2,2);assegna(1,2,3)} 1.
1 {assegna(2,1,1);assegna(2,1,2);assegna(2,1,3)} 1.
1 {assegna(2,2,1);assegna(2,2,2);assegna(2,2,3)} 1.
```

```
1{ p; q; r } 1.
```

can be transformed in:

```
p :- not np. np :- not p.
q :- not nq. nq :- not q.
r :- not nr. nr :- not r.
```

(non deterministic choice viewed in the last lesson) plus

```
:- p, q.
:- p, r.
:- q, r
```

Constraint. It is not possible that both p and q holds, it is not possible that both p and r holds, It is not possible that both q and r holds.
Exercise Try a similar rewriting of `2 { p; q; r; not s } 3.`
Verify your encoding using `clingo filename.pl 0`

The cardinality constraint used as 0 . . . 1

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

introduces the nondeterministic choice
(`good(X,Y)` can either hold or not).

You can also write it this way:

```
{ good(X,Y) } :- lato(X), lato(Y).
```

(and of course you can define it defining a new predicate nogood as not good, and vice versa)

The cardinality constraint used as 0 . . . 1

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

introduces the nondeterministic choice
(`good(X,Y)` can either hold or not).
You can also write it this way:

```
{ good(X,Y) } :- lato(X), lato(Y).
```

(and of course you can define it defining a new predicate nogood as not good, and vice versa)

http://sourceforge.net/projects/potassco/files/guide/
There are the following built ins:

```
plus      L + R          minus     L - R
uminus    - R            times     L * R
divide    L / R          modulo    L \ R
absolute  |  R|          power     L ** R
bitand    L & R          bitor     L ? R
bitxor    L ^ R          bitneg    ~ R
```

- They are evaluated during the grounding process.
- You can also use your own functions using external (C/C++) pieces of code

- ASP supports *aggregates*
- aggregates allows to defined intensionally numerical functions to sets of values
- `count` and `sum` are the two main aggregates.
- They are rather expressive. The syntax is not yet stable. We use the one of GRINGO4 (different from previous versions).

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

The output is `somma(6).`

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

The output is `somma(3).`

Anything strange? Semantics is based on sets, not on multi-sets.

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

The output is `somma(6).`

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

The output is `somma(3).`

Anything strange? Semantics is based on sets, not on multi-sets.

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

The output is `somma(6).`

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

The output is `somma(3).`

Anything strange? Semantics is based on sets, not on multi-sets.

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

The output is `somma(6).`

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

The output is `somma(3).`

Anything strange? Semantics is based on sets, not on multi-sets.

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

The output is `somma(6).`

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

The output is `somma(3).`

Anything strange? Semantics is based on sets, not on multi-sets.

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

The output is `somma(5).`

We have fixed collecting “pairs” and counting pairs.
Not all of us is happy with this new semantics. In particular Vladimir Lifschitz whose (previous year) programs did not run correctly in front of his students . . . very embarrassing for the “boss” of the area.

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

The output is `somma(5).`

We have fixed collecting “pairs” and counting pairs.
Not all of us is happy with this new semantics. In particular Vladimir Lifschitz whose (previous year) programs did not run correctly in front of his students . . . very embarrassing for the “boss” of the area.

`count` has similar syntax and problems. It counts the number of atoms that satisfy a condition.

```
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 }.
```

The output is `conta(1).`

Modeling CSP with ASP

Magic Square

Problem: Fill an $n \times n$ matrix (square) with the numbers $1, \ldots, n^2$ in such a way that each number is used once and the sum of each row, of each column, and of each of the two diagonals is the same.
Since the global sum is

$$\sum_{i=1}^{n^2} i = \frac{n^2(n^2+1)}{2}$$

each one of these sums amounts at $S = \frac{n^2(n^2+1)}{2n}$.
E.g., with $n = 3$, $S = 15$.

2	7	6
9	5	1
4	3	8

Magic Square

Problem: Fill an $n \times n$ matrix (square) with the numbers $1, \ldots, n^2$ in such a way that each number is used once and the sum of each row, of each column, and of each of the two diagonals is the same.
Since the global sum is

$$\sum_{i=1}^{n^2} i = \frac{n^2(n^2+1)}{2}$$

each one of these sums amounts at $S = \frac{n^2(n^2+1)}{2n}$.
E.g., with $n = 3$, $S = 15$.

2	7	6
9	5	1
4	3	8

AN IMPERATIVE SOLUTION

SCAN PERMUTATIONS

```
int main(){

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

  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)
     stampamat(quad);
  else printf("No solution \n");
}
```

An imperative solution

Check

```
const int n=3;
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;
}
```

An imperative solution

Dijkstra, A Discipline of Programming, Prentice-Hall, 1997, pp. 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--;
    }
}
```

An imperative solution

Auxiliary procedures

```
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");
   }
}
```

ASP MODELING

JUST A QUICK VIEW — WE'LL SEE DETAILS LATER

```
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).


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)}.

:- lato(X), sum_cols(X,V), magicval(T), V != T.
:- lato(X), sum_rows(X,V), magicval(T), V != T.
:- diag(D), sum_diag(D,V), magicval(T), V != T.

#show magic/3.
```

EFFICIENCY

```
Agostino@ACUFENE /cygdrive/c/Documents and Settings/Agostino/Dropbox/VARIE
$ time ./a.exe
Tentativi massimi: 362880
HAI VINTO
|  2 | 7 | 6 |
 ----
|  9 | 5 | 1 |
 ----
|  4 | 3 | 8 |
 ----

real    0m0.094s
user    0m0.015s
sys     0m0.046s
Agostino@ACUFENE /cygdrive/c/Documents and Settings/Agostino/Dropbox/VARIE
$ time ./a.exe
Tentativi massimi: 20922789888000
HAI VINTO
|  1 | 2 | 15 | 16 |
 ----
|  12 | 14 | 3 | 5 |
 ----
|  13 | 7 | 10 | 4 |
 ----
|  8 | 11 | 6 | 9 |
 ----

real    74m16.214s
user    74m13.376s
sys     0m0.077s
```

EFFICIENCY

```
$ clingo -c n=3 quadrato.lp
clingo version 4.4.0
Reading from quadrato.lp
Solving...
Answer: 1
magic(1,1,2) magic(1,3,6) magic(1,2,7) magic(2,3,1) magic(2,2,5) magic(2,1,9) m
agic(3,2,3) magic(3,1,4) magic(3,3,8)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 0.077s (Solving: 0.06s 1st Model: 0.06s Unsat: 0.00s)
CPU Time     : 0.078s

Agostino@ACUFENE /cygdrive/c/Users/Agostino/Dropbox/DIDATTICA/AI-LP-GIOCHI/CODI
CI
$ clingo -c n=4 quadrato.lp
clingo version 4.4.0
Reading from quadrato.lp
Solving...
Answer: 1
magic(1,2,4) magic(1,4,5) magic(1,3,10) magic(1,1,15) magic(2,2,1) magic(2,4,8)
 magic(2,3,11) magic(2,1,14) magic(3,1,3) magic(3,3,6) magic(3,4,9) magic(3,2,1
6) magic(4,1,2) magic(4,3,7) magic(4,4,12) magic(4,2,13)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 1.146s (Solving: 1.04s 1st Model: 1.04s Unsat: 0.00s)
CPU Time     : 1.139s

Agostino@ACUFENE /cygdrive/c/Users/Agostino/Dropbox/DIDATTICA/AI-LP-GIOCHI/CODI
```

EFFICIENCY

```
Agostino@ACUFENE /cygdrive/c/Users/Agostino/Dropbox/DIDATTICA/AI-LP-GIOCHI/CODI
CI
$ clingo -c n=5 quadrato.lp
clingo version 4.4.0
Reading from quadrato.lp
Solving...
Answer: 1
magic(1,1,2) magic(1,4,7) magic(1,2,16) magic(1,3,17) magic(1,5,23) magic(2,5,3
) magic(2,1,5) magic(2,3,13) magic(2,4,20) magic(2,2,24) magic(3,3,6) magic(3,2
,10) magic(3,4,12) magic(3,1,18) magic(3,5,19) magic(4,2,1) magic(4,3,8) magic(
4,5,9) magic(4,4,22) magic(4,1,25) magic(5,4,4) magic(5,5,11) magic(5,2,14) mag
ic(5,1,15) magic(5,3,21)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 95.666s (Solving: 95.11s 1st Model: 95.10s Unsat: 0.00s)
CPU Time     : 95.457s

Agostino@ACUFENE /cygdrive/c/Users/Agostino/Dropbox/DIDATTICA/AI-LP-GIOCHI/CODI
CI
$ clingo -c n=6 quadrato.lp
clingo version 4.4.0
Reading from quadrato.lp
Solving...
Answer: 1
magic(1,3,6) magic(1,5,15) magic(1,1,19) magic(1,6,20) magic(1,4,21) magic(1,2,3
0) magic(2,2,4) magic(2,6,7) magic(2,5,18) magic(2,1,22) magic(2,4,26) magic(2,3
,34) magic(3,2,1) magic(3,1,5) magic(3,4,12) magic(3,3,28) magic(3,6,32) magic(3
,5,33) magic(4,6,8) magic(4,4,11) magic(4,1,16) magic(4,3,23) magic(4,2,24) magi
c(4,5,29) magic(5,6,9) magic(5,4,10) magic(5,5,14) magic(5,3,17) magic(5,2,25) m
agic(5,1,36) magic(6,5,2) magic(6,3,3) magic(6,1,13) magic(6,2,27) magic(6,4,31)
 magic(6,6,35)
SATISFIABLE

Models       : 1+
Calls        : 1
Time         : 42111.245s (Solving: 42108.98s 1st Model: 42108.97s Unsat: 0.00s)
CPU Time     : 42084.000s

Agostino@ACUFENE /cygdrive/c/Users/Agostino/Dropbox/DIDATTICA/AI-LP-GIOCHI/CODIC
```

LESSON LEARNT

- If you don't have a great idea, direct encoding in C/JAVA etc is useless
- This applies to all NP complete problems
- ASP solvers (and SAT solvers and CP solvers) are nowadays much more clever than any heuristic we can develop, implement and test in a reasonable time

MAGIC SQUARE MODELING

THE CASE $n = 3$

"domain" predicates:

```
lato(1..3).
valore(1..9).
diag(1..2).
magicval(15).
```

We define the predicate `magic`. Intuitively, `magic`(X, Y, V) should be true if in cell (X, Y) the value is V.
For each cell, assign one and only one value:

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

For each value, assign it to one and only one cell:

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

MAGIC SQUARE MODELING

THE CASE $n = 3$

"domain" predicates:

```
lato(1..3).
valore(1..9).
diag(1..2).
magicval(15).
```

We define the predicate `magic`. Intuitively, `magic`(X, Y, V) should be true if in cell (X, Y) the value is V.
For each cell, assign one and only one value:

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

For each value, assign it to one and only one cell:

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

MAGIC SQUARE MODELING

THE CASE $n = 3$

Compute the sum on rows, columns, diagonals.

```
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 flag `diff_sum` capturing a wrong sum is introduced.

```
diff_sum :- lato(X),  sum_col(X,V), magicval(S), V != S.
diff_sum :- lato(X),  sum_row(X), magicval(S), V != S.
diff_sum :- sum_diag(1,V),magicval(S), V != S.
diff_sum :- sum_diag(2,V),magicval(S), V != S.
```

Last: `diff_sum` cannot be true!

```
:- diff_sum.
```

MAGIC SQUARE MODELING

THE CASE $n = 3$

Compute the sum on rows, columns, diagonals.

```
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 flag `diff_sum` capturing a wrong sum is introduced.

```
diff_sum :- lato(X),  sum_col(X,V), magicval(S), V != S.
diff_sum :- lato(X),  sum_row(X), magicval(S), V != S.
diff_sum :- sum_diag(1,V),magicval(S), V != S.
diff_sum :- sum_diag(2,V),magicval(S), V != S.
```

Last: `diff_sum` cannot be true!

```
:- diff_sum.
```

MAGIC SQUARE MODELING

THE CASE $n = 3$

The `diff_sum` can be expressed by using constraints:

```
:- lato(X),  sum_col(X,V), magicval(S), V != S.
```

It states that *It cannot happen that there is a column X such that its sum is V and V is different from the magic sum.*
Constraints allow to state universal quantification:

$$(\forall X)(\forall V)\neg(\texttt{lato}(X) \wedge \texttt{sum\_col}(X, V) \wedge \texttt{magicval}(S) \wedge V \neq S)$$

Namely,

$$(\forall X)(\forall V)((\texttt{lato}(X) \wedge \texttt{sum\_col}(X, V) \wedge \texttt{magicval}(S)) \rightarrow V \neq S)$$

Similarly for rows and diagonals:

```
:- lato(X),  sum_row(X,V), magicval(S), V != S.
:- sum_diag(1,V),magicval(S), V != S.
:- sum_diag(2,V),magicval(S), V != S.
```

MAGIC SQUARE MODELING

THE CASE $n = 3$

The `diff_sum` can be expressed by using constraints:

```
:- lato(X),  sum_col(X,V), magicval(S), V != S.
```

It states that *It cannot happen that there is a column X such that its sum is V and V is different from the magic sum.*
Constraints allow to state universal quantification:

$$(\forall X)(\forall V)\neg(\texttt{lato}(X) \wedge \texttt{sum\_col}(X, V) \wedge \texttt{magicval}(S) \wedge V \neq S)$$

Namely,

$$(\forall X)(\forall V)((\texttt{lato}(X) \wedge \texttt{sum\_col}(X, V) \wedge \texttt{magicval}(S)) \rightarrow V \neq S)$$

Similarly for rows and diagonals:

```
:- lato(X),  sum_row(X,V), magicval(S), V != S.
:- sum_diag(1,V),magicval(S), V != S.
:- sum_diag(2,V),magicval(S), V != S.
```

MAGIC SQUARE MODELING

THE GENERAL CASE

```
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.
```

LESSON LEARNT

- Use of cardinality constraints $1\{\cdots\}1$ for forcing relations to be functions
- Use of cardinality constraints $1\{\cdots\}1$ for forcing injectivity.
- Use of aggregates (sum) for compact encoding of properties
- Use of constraints for removing wrong solutions. They allow us to express *universal quantification*.