Answer Set Programming

João Leite

Logic Programs

- Sets of facts and rules
  - person(peter).
  - person(anna).
  - bar(hard_rock).
  - happy(P) :- person(P), bar(B), beer(C),
    goes_to(P,B), sells(B,C), likes(P,C).
  - uncle(Y,X) :- person(X), person(Y), person(Z), child_of(X,Z),
    brother(Y,Z).
  - innocent(X) :- person(X), not guilty(X).

Logic Programs

- A Normal Logic Program P is a set of rules:

  $H :- A_1, \ldots, A_n, \text{not } B_1, \ldots \text{not } B_m \ (n,m \geq 0)$

  where H, A_i and B_j are atoms
- Literals not B_j are called *default literals*
- When no rule in P has default literals, P is called *definite (or positive)*

Semantics of Positive Programs

- Program:

```
person(peter).
person(anna).
person(john).
friend_of(peter,anna).
has_friends(X) :- person(X), person(Y), friend_of(X,Y).
happy(X) :- person(X), has_friends(X).
friend_of(X,Y) :- person(X), person(Y), friend_of(Y,X).
martian(X) :- person(X), born_in_mars(X).
```

- What is true?

person(peter)	has_friends(peter)	happy(anna)
person(anna)	happy(peter)	has_friends(john) ?
person(john)	friend_of(anna,peter)	martian(peter) ?
friend_of(peter,anna)	has_friends(anna)	martian(john) ?

Semantics of Positive Programs

- Minimal Models: the model (set of true atoms) contains the minimum atoms possible, derivable from the program.
- Every definite logic program has one (and only one) least (truth ordering) model, denoted by Cn(P).

Semantics of Positive Programs

- The least model of a positive program can be computed bottom-up, via operator T_P, starting from the empty model, and adding to it all immediate consequences that can be derived from what has been previously derived using the program rules. The process ends when nothing else is derivable.
- Let I be an interpretation of definite program P.

$$T_P(I) = \{H: (H :- Body) \in P \text{ and } Body \subseteq I\}$$

- If P is definite, T_P is monotone and continuous. Its minimal fixpoint can be built by

$$I_0 = \{\} \qquad \text{and} \qquad I_n = T_P(I_{n-1})$$

- The least model of definite program P is

$$least(P) = T_P{\uparrow}\omega(\{\})$$

Semantics of Positive Programs

Program:

```
person(peter).   person(anna).   person(john).   friend_of(peter,anna).
has_friends(peter) :- person(peter), person(anna), friend_of(peter, anna).
has_friends(anna) :- person(anna), person(peter), friend_of(anna, peter).
has_friends(john) :- person(john), person(peter), friend_of(john, peter).
…
happy(peter) :- person(peter), has_friends(peter).
happy(anna) :- person(anna), has_friends(anna).
happy(john) :- person(john), has_friends(john).
friend_of(peter, peter) :- person(peter), person(peter), friend_of(peter, peter).
friend_of(anna, peter) :- person(anna), person(peter), friend_of(peter, anna).
friend_of(john, peter) :- person(john), person(peter), friend_of(peter, john).
…
martian(peter) :- person(peter), born_in_mars(peter).
martian(anna) :- person(anna), born_in_mars(anna).
martian(john) :- person(john), born_in_mars(john).
```

$I0 = \{\}$

$I1 = I0 \cup \{person(peter), person(anna), person(john), friend\_of(peter,anna)\}$

$I2 = I1 \cup \{has\_friends(peter), friend\_of(anna, peter)\}$

$I3 = I2 \cup \{happy(peter), has\_friends(anna)\}$

$I4 = I3 \cup \{happy(anna)\}$

$I5 = I4 \cup \{\} = I4$

Least Model = {person(peter),person(anna),person(john),friend_of(peter,anna), friend_of(anna,peter), has_friends(peter), has_friends(anna),happy(peter), happy(anna)}

Semantics of Programs with Negation

- How about programs with negation?
- The minimal model semantics doesn't apply anymore.
- Program:

  p :- not q.
- Has two Minimal Models:

  {p}

  {q}
- No least model!

Semantics of Programs with Negation

- In some programs with negation, it is easy to determine one intuitive model:
- Program:

  person(peter).

  person(john).

  guilty(peter).

  innocent(X) :- person(X), not guilty(X).
- Model:

  {person(peter),person(john),guilty(peter),innocent(john)}

Semantics of Programs with Negation

- In other programs, it doesn't seem possible to determine a single intuitive model:
- Program:

  person(peter).

  pacifist(X) :- person(X), not hawk(X).

  hawk(X) :- person(X), not pacifist(X).
- Model(s):

  ? {person(peter)} ×

  ? {person(peter), hawk(peter), pacifist(peter)} ×

  ? {person(peter), hawk(peter)} ✓

  ? {person(peter), pacifist(peter)} ✓

Semantics of Programs with Negation

- Stable Models
  - Stable Models are those that are somehow corroborated by the rules of the program.
  - They can be seen as possible scenarios.
  - Each program may have zero, one or more stable models.
- Determining Stable Models
  - It is possible to determine if some interpretation is a stable model through its comparison with the least model of a positive program obtained from the original program after a suitable transformation (Gelfond-Lifschitz Transformation).

Stable Models Semantics

- Gelfond-Lifschitz Transformation
  - Let P be a program and I an interpretation (set of atoms). The program P^I is obtained from P by:
    - removing each rule with some negative literal in the body “not a” such that $a \in I$;
    - removing all negative literals from all remaining rules.
  - Program P^I is definite (positive) so it has the least model $Cn(P^I)$.
- Stable Models
  - Let P be a program and I an interpretation. I is a stable model of P iff:

$$I = Cn(P^I)$$

Stable Models Semantics

- P:

  person(peter).
  pacifist(peter) :- person(peter), not hawk(peter).
  hawk(peter) :- person(peter), not pacifist(peter).

- I = {person(peter), pacifist(peter)} is Stable Model?
- P^I:

  person(peter).
  pacifist(peter) :- person(peter).

- $Cn(P^I)$ = {person(peter), pacifist(peter)}

  $Cn(P^I) = I \quad \Rightarrow \quad$ I is Stable Model.

Stable Models Semantics

- P:

  person(peter).
  pacifist(peter) :- person(peter), not hawk(peter).
  hawk(peter) :- person(peter), not pacifist(peter).

- I = {person(peter), hawk(peter)} is Stable Model?
- P^I:

  person(peter).
  hawk(peter) :- person(peter).

- $Cn(P^I)$ = {person(peter), hawk(peter)}

  $Cn(P^I) = I \quad \Rightarrow \quad$ I is Stable Model.

Stable Models Semantics

- P:

  person(peter).
  pacifist(peter) :- person(peter), not hawk(peter).
  hawk(peter) :- person(peter), not pacifist(peter).

- I = {person(peter)} is Stable Model?
- P^I:

  person(peter).
  pacifist(peter) :- person(peter).
  hawk(peter) :- person(peter).

- $Cn(P^I)$ = {person(peter),pacifist(peter),hawk(peter)}

  $Cn(P^I) \neq I \quad \Rightarrow$ I is not Stable Model.

Programs with Variables

- Let P be a logic program
  - Let **T** be the set of terms (without variables)
  - Let **A** be the set of atoms (without variáveis) obtainable from **T**
    - Usually caled the Herbrand Universe
- Ground instances of r∈P: set of rules without variables obtained through the substitution of all variables in r by elements of **T**:

$$ground(r) = \left\{ r(\theta) \mid \theta : \mathrm{var}(r) \rightarrow \mathbf{T} \wedge \mathrm{var}(r\theta) = \varnothing \right\}$$

  - var(r) denotes the set of all variables of r;
  - θ is a substitution (ground)
- Ground instances of P: $ground(P) = \bigcup_{r \in P} ground(r)$

Programs with Variables

- Let P be a logic program with variables. A set of (ground) atoms X is a stable model of P if

$$Cn\left(ground(P)^X\right) = X$$

Stable Models Semantics - Examples

- The Program:

  person(peter).
  pacifist(X) :- person(X), not hawk(X).
  hawk(X) :- person(X), not pacifist(X).

  represents a disjunction where a person (peter) is either a pacifist or a hawk. The two stable models are:

  {person(peter), pacifist(peter)}
  {person(peter), hawk(peter)}

- If we add the fact person(anna), we would obtain 4 stable models, corresponding to all combinations where peter and anna are either pacifists or hwaks:

  {person(peter), person(anna), pacifist(peter), pacifist(anna)}
  {person(peter), person(anna), hawk(peter), pacifist(anna)}
  {person(peter), person(anna), pacifist(peter), hawk(anna)}
  {person(peter), person(anna), hawk(peter), hawk(anna)}

Stable Models Semantics - Examples

- If we were to represent a disjunction between 3 atoms, we could do it as follows:

```
album(takk).
tape(X) :- album(X), not cd(X), not vinyl(X).
cd(X) :- album(X), not tape(X), not vinyl(X).
vinyl(X) :- album(X), not tape(X), not cd(X).
```

- This program represents a situation where an album has one support (tape, cd or vinyl).
- It has 3 stable models:

{album(takk), tape(takk)}
{album(takk), cd(takk)}
{album(takk), vinyl(takk)}

Stable Models Semantics - Examples

- It would be possible to represent that an album may have zero, one, two or all three formats. To do that, we will introduce three new auxiliary predicates (n_tape(X), n_cd(X) e n_vinyl(X)). The program is:

```
album(takk).
tape(X) :- album(X), not n_tape(X).
n_tape(X) :- album(X), not tape(X).
cd(X) :- album(X), not n_cd(X).
n_cd(X) :- album(X), not cd(X).
vinyl(X) :- album(X), not n_vinyl (X).
n_vinyl (X) :- album(X), not vinyl (X).
```

- This program has eight stable models. They are (omitting album(takk)):

{tape(takk), cd(takk), vinyl(takk)}
{tape(takk), n_cd(takk), vinyl(takk)}
{n_tape(takk), n_cd(takk), vinyl(takk)}
{tape(takk), n_cd(takk), n_vinyl(takk)}
{n_tape(takk), cd(takk), vinyl(takk)}
{tape(takk), cd(takk), n_vinyl(takk)}
{n_tape(takk), cd(takk), n_vinyl(takk)}
{n_tape(takk), n_cd(takk), n_vinyl(takk)}

- If we also omit the new auxiliary predicates, we obtain:

{tape(takk), cd(takk), vinyl(takk)}
{tape(takk), vinyl(takk)}
{vinyl(takk)}
{tape(takk)}
{cd(takk), vinyl(takk)}
{tape(takk), cd(takk)}
{cd(takk)}
{}

- Making it clearer the correspondence between the stable models and the possible scenarios described by the problem and program.

Stable Models - Examples

- Some rules have the effect of preventing the existence of some stable models.
  - For example, if we have a logic program and we want to prevent the existence of stable models where, simultaneously, atoms a and b are true, and c and d are false, we can add the rule (where α is a new atom):

$$\alpha \text{ :- a, b, not c, not d, not } \alpha.$$

Stable Models Semantics - Examples

- Let's consider the previous program:

  person(peter).

  pacifist(X) :- person(X), not hawk(X).

  hawk(X) :- person(X), not pacifist(X).

- If we learn that Peter is not a hawk, then we must eliminate all stable models (worlds) in which Peter is a hawk.
- This can be done by adding the rule:

  α :- hawk(peter), not α.

- Let's see if it works…

Stable Models - Examples

- P:

  person(peter).
  pacifist(peter) :- person(peter), not hawk(peter).
  hawk(peter) :- person(peter), not pacifist(peter).
  α :- hawk(peter), not α.

- I = {person(peter), hawk(peter)} is Stable Model?
- P^I:

  person(peter).
  hawk(peter) :- person(peter).
  α :- hawk(peter).

- $Cn(P^I)$ = {person(peter), hawk(peter), α}

  $Cn(P^I) = I \quad \Rightarrow \quad$ I is not a Stable Model.

Stable Models Semantics

- P:

  person(peter).

  pacifist(peter) :- person(peter), not hawk(peter).

  hawk(peter) :- person(peter), not pacifist(peter).

  α :- hawk(peter), not α.

- I = {person(peter), pacifist(peter)} is Stable Model?
- P^I:

  person(peter).

  pacifist(peter) :- person(peter).

  α :- hawk(peter).

- $Cn(P^I)$ = {person(peter), pacifist(peter)}

  $Cn(P^I) = I \quad \Rightarrow \quad$ I is Stable Model.

Stable Models - Examples

- If, conversely, we just wish to allow the existence of stable models that obey a certain condition, we use a similar technique.
  - For example, if we have a logic program and we want to only allow the existence of stable models where, simultaneously, atoms a and b are true, and c and d are false, we can add the pair of rules (where α e β are new atoms):

$$\beta \text{ :- a, b, not c, not d.}$$

$$\alpha \text{ :- not } \beta\text{, not } \alpha\text{.}$$

Stable Models Semantics - Examples

- Let's consider the previous program:

  person(peter).

  pacifist(X) :- person(X), not hawk(X).

  hawk(X) :- person(X), not pacifist(X).

- But now, instead, we learn that Peter is a hawk, then we must keep all stable models (worlds) in which Peter is a hawk, eliminating all remaining.
- This can be done by adding the rules:

  β :- hawk(peter).

  α :- not β, not α.

- Let's see if it works…

Stable Models - Examples

- P:

  person(peter).
  pacifist(peter) :- person(peter), not hawk(peter).
  hawk(peter) :- person(peter), not pacifist(peter).
  β :- hawk(peter).
  α :- not β, not α.

- I = {person(peter), hawk(peter), β} is Stable Model?
- P^I:

  person(peter).
  hawk(peter) :- person(peter).
  β :- hawk(peter).

- $Cn(P^I)$ = {person(peter), hawk(peter), β}

  $Cn(P^I) = I \quad \Rightarrow \quad$ I is Stable Model.

Stable Models Semantics

- P:

  person(peter).

  pacifist(peter) :- person(peter), not hawk(peter).

  hawk(peter) :- person(peter), not pacifist(peter).

  β :- hawk(peter).

  α :- not β, not α.

- I = {person(peter), pacifist(peter)} is Stable Model?
- P^I: person(peter).

  pacifist(peter) :- person(peter).

  β :- hawk(peter).

  α.

- $Cn(P^I)$ = {person(peter), pacifist(peter), α}

  $Cn(P^I) = I \quad \Rightarrow \quad$ I is not Stable Model.

Stable Models - Examples

- To simplify the writing of rules that prevent the existence of stable models (usually referred to as integrity constraints), we omit all occurrences of the new atom α.
- To prevent the existence of stable models where CONDITION is true, we add the rule:

  :- CONDITION.

  - For example, to prevent the existence of stable models where, simultaneously, atoms a and b are true, and c and d are false, we add the rule :

    :- a, b, not c, not d.

  - For example, for only allowing the existence of stable models where, simultaneously, atoms a and b are true, and c and d are false, we can add the pair of rules (where β is a new atom):

    β :- a, b, not c, not d.

    :- not β.

Stable Models Semantics - Examples

- If we know that Peter is not a hawk, the program is:

```
person(peter).
pacifist(X) :- person(X), not hawk(X).
hawk(X) :- person(X), not pacifist(X).
:- hawk(peter).
```

- If we know that Peter is a hawk, the program is:

```
person(peter).
pacifist(X) :- person(X), not hawk(X).
hawk(X) :- person(X), not pacifist(X).
β :- hawk(peter).
:- not β.
```

Encoding of Closed Conditions

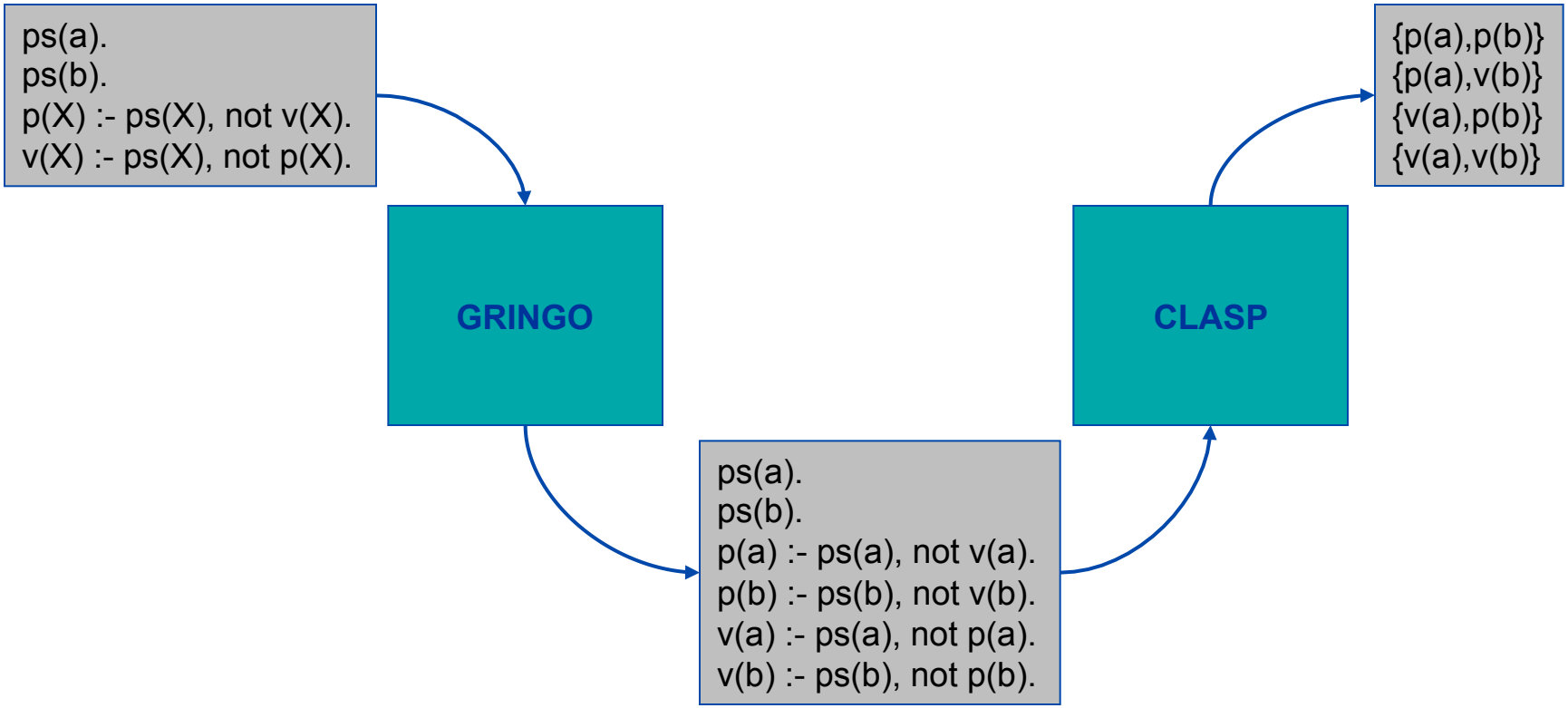
- The formula $F_1 \wedge F_2$ is encoded as
  - $p\_F \leftarrow p\_F_1, p\_F_2.$
- The formula $F_1 \vee F_2$ is encoded as
  - $p\_F \leftarrow p\_F_1.$
  - $p\_F \leftarrow p\_F_2.$
- The formula $\neg F_1$ is encoded as
  - $p\_F \leftarrow$ not $p\_F_1.$
- The limited existential quantifier $\exists X{:}domain.F_1$ is encoded as
  - $p\_F \leftarrow domain(X), p\_F_1(X)$
- The limited universal quantifier $\forall X{:}domain.F_1$
  - $p\_F \leftarrow$ not $n\_p\_F.$
  - $n\_p\_F \leftarrow domain(X),$ not $p\_F_1(X).$

Answer-Set Programming

- The expressiveness and the declarative character of logic programming under the stable models semantics…
- The existence of software that allows for the fast computation of stable models:
  - CLINGO: https://potassco.org/
  - and many more...
- …led to the application of this paradigm in solving certain problems, leading to the introduction of …

Answer Set Programming

CLINGO = GRINGO + CLASP



- Command line:

  clingo filename 0

  - 0 – Returns all answer sets.

Answer-Set Programming

- Concept:
  - Represent the problem as a logic program such that its stable models (answer-sets) correspond to the possible solutions (answers).

one stable model = one solution

- 3 steps:
  - determine the format of the solutions to the problem. normally, there will be one (or more) predicates that will act as “containers” to the solution;
  - generate all possible models, where each model represents one hypothetical solution to the problem;
  - eliminate the models that do not obey the problem specification.

Stable Models - Examples

- Let's consider a scenario where we have a music collection composed of albums. We may have each album in cd, tape or vinyl, but we don't remember which album exists in each format. For now, to keep things simple, let's assume we only have one album (takk).

  album(takk). format(cd). format(tape). format(vinyl).

  in_format(A,F) :- album(A), format(F), not n_in_format(A,F).

  n_in_format(A,F) :- album(A), format(F), not in_format(A,F).

- This program has the following stable models (omitting the atoms album/1, format/1 e n_in_format/2):

  {in_format(takk,tape), in_format(takk,cd), in_format(takk,vinyl)}

  {in_format(takk,cd), in_format (takk,vinyl)}

  {in_format(takk,tape), in_format(takk,vinyl)}

  {in_format(takk,tape), in_format(takk,cd)}

  {in_format(takk,tape)}

  {in_format(takk,vinyl)}

  {in_format(takk,cd)}

  {}

Stable Models - Examples

- Going back to the previous program:
  album(takk). format(cd). format(tape). format(vinyl).
  in_format(A,F) :- album(A), format(F), not n_in_format(A,F).
  n_in_format(A,F) :- album(A), format(F), not in_format(A,F).
- Whose stable models are (showing only the atoms in_format/2):
  {in_format(takk,tape), in_format(takk,cd), in_format(takk,vinyl)}
  {in_format(takk,cd), in_format (takk,vinyl)} {in_format(takk,cd)}
  {in_format(takk,tape), in_format(takk,vinyl)} {in_format(takk,vinyl)}
  {in_format(takk,tape), in_format(takk,cd)} {in_format(takk,tape)} {}
- If we know that there is no album simultaneously in tape and CD, we can add the following rule (integrity constraint):
  :- album(A), in_format(A,tape), in_format(A,cd).
- After the introduction of the new rule, the stable models are:
  {in_format(takk,cd), in_format (takk,vinyl)} {in_format(takk,cd)}
  {in_format(takk,tape)} {in_format(takk,tape), in_format(takk,vinyl)}
  {in_format(takk,vinyl)} {}

Stable Models - Examples

- Continuing with the same program:
  album(takk). format(cd). format(tape). format(vinyl).
  in_format(A,F) :- album(A), format(F), not n_in_format(A,F).
  n_in_format(A,F) :- album(A), format(F), not in_format(A,F).
  :- album(A), in_format(A,tape), in_format(A,cd).
- Whose stable models are (showing only the atoms in_format/2):
  {in_format(takk,cd), in_format (takk,vinyl)} {in_format(takk,cd)}
  {in_format(takk,tape)} {in_format(takk,tape), in_format(takk,vinyl)}
  {in_format(takk,vinyl)} {}
- If we know that each album exists in at least one format, we can add the following rules:
  with_format(A) :- album(A), format(F), in_format(A,F).
  :- album(A), not with_format(A).
- After the introduction of the new rules, the stable models are:
  {in_format(takk,cd), in_format (takk,vinyl)} {in_format(takk,cd)}
  {in_format(takk,tape)} {in_format(takk,tape), in_format(takk,vinyl)}
  {in_format(takk,vinyl)}

Stable Models - Examples

- Continuing with the same program:
  album(takk). format(cd). format(tape). format(vinyl).
  in_format(A,F) :- album(A), format(F), not n_in_format(A,F).
  n_in_format(A,F) :- album(A), format(F), not in_format(A,F).
  :- album(A), in_format(A,tape), in_format(A,cd).
  com_format(A) :- album(A), format(F), in_format(A,F).
  :- album(A), not com_format(A).
- Whose stable models are (showing only the atoms in_format/2):
  {in_format(takk,cd), in_format (takk,vinyl)} {in_format(takk,cd)}
  {in_format(takk,tape)} {in_format(takk,tape), in_format(takk,vinyl)}
  {in_format(takk,vinyl)}
- If we know the album “Takk” exists in vinyl, we can simply add the following fact:
  in_format(takk,vinyl).
- After the introduction of the new fact, the stable models are :
  {in_format(takk,cd), in_format (takk,vinyl)}
  {in_format(takk,tape), in_format(takk,vinyl)} {in_format(takk,vinyl)}

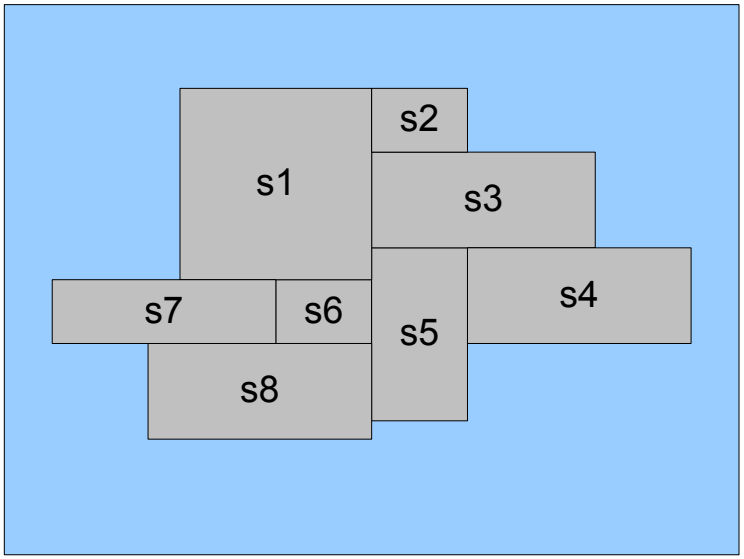
Stable Models - Examples

- Continuing with the same program, to which we add the new album “Boy”:
  album(takk). album(boy) format(cd). format(tape). format(vinyl).
  in_format(A,F) :- album(A), format(F), not n_in_format(A,F).
  n_in_format(A,F) :- album(A), format(F), not in_format(A,F).
  :- album(A), in_format(A,tape), in_format(A,cd).
  com_format(A) :- album(A), format(F), in_format(A,F).
  :- album(A), not com_format(A).
  in_format(takk,vinyl).
- The stable models are (where in_format was replaced by f, and omitting the remaining atoms):
  {f(takk,vinyl),f(boy,vinyl),f(boy,cd),f(takk,cd)} {f(takk,vinyl),f(boy,cd),f(takk,cd)}
  {f(takk,vinyl),f(boy,tape)} {f(takk,vinyl),f(boy,tape),f(takk,tape)}
  {f(takk,vinyl),f(boy,tape),f(takk,cd)} {f(takk,vinyl),f(boy,vinyl),f(boy,tape),f(takk,cd)}
  {f(takk,vinyl),f(boy,vinyl),f(takk,cd)} {f(takk,vinyl),f(boy,vinyl),f(boy,tape)}
  {f(takk,vinyl),f(boy,vinyl),f(boy,tape),f(takk,tape)} {f(takk,vinyl),f(boy,cd),f(takk,tape)}
  {f(takk,vinyl),f(boy,cd)} {f(takk,vinyl),f(boy,vinyl)} {f(takk,vinyl),f(boy,vinyl),f(boy,cd)}
  {f(takk,vinyl),f(boy,vinyl),f(boy,cd),f(takk,tape)} {f(takk,vinyl),f(boy,vinyl),f(takk,tape)}
- If we know that there are no two albums with the same format, we can add the following rule (where neq(A1, A2) is true if A1≠A2):
  :- album(A1), album(A2), format(F), in_format(A1,F), in_format(A2,F), neq(A1,A2).
- After the introduction of the new fact, the stable models are:
  {f(takk,vinyl),f(boy,cd)} {f(takk,vinyl),f(boy,cd),f(takk,tape)}
  {f(takk,vinyl),f(boy,tape),f(takk,cd)} {f(takk,vinyl),f(boy,tape)}

Stable Models - Examples

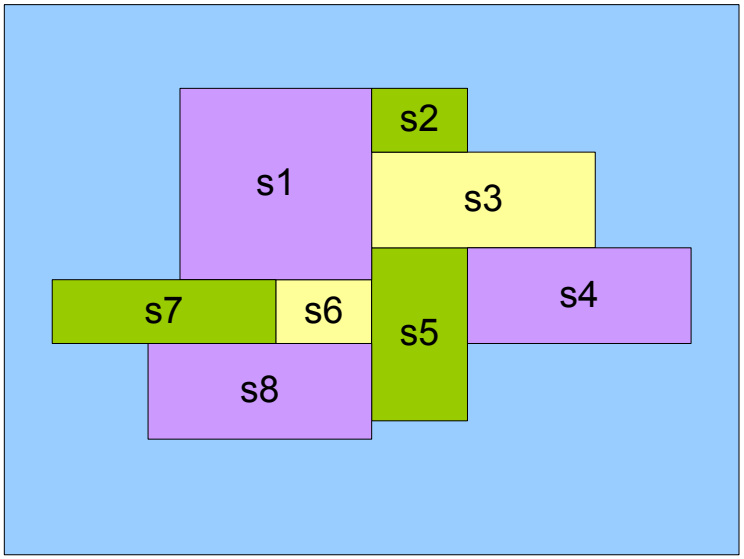
- Continuing with the same program:
  album(takk). album(boy) format(cd). format(tape). format(vinyl).
  in_format(A,F) :- album(A), format(F), not n_in_format(A,F).
  n_in_format(A,F) :- album(A), format(F), not in_format(A,F).
  :- album(A), in_format(A,tape), in_format(A,cd).
  com_format(A) :- album(A), format(F), in_format(A,F).
  :- album(A), not com_format(A).
  in_format(takk,vinyl).
  :- album(A1), album(A2), format(F), in_format(A1,F), in_format(A2,F), neq(A1,A2).
- Whose stable models are (showing only the atoms in_format/2):
  {in_format(takk,vinyl), in_format(boy,cd)} {in_format (takk,vinyl), in_format(boy,tape)}
  {in_format (takk,vinyl), in_format(boy,cd), in_format(takk,tape)}
  {in_format (takk,vinyl), in_format(boy,tape), in_format(takk,cd)}
- If we know that there is at least one tape and one cd, we can add the following rules:
  ok :- album(A1), album(A2), in_format(A1,cd), in_format(A2,tape).
  :- not ok.
- After the introduction of the new fact, the stable models are:
  {in_format (takk,vinyl), in_format(boy,cd), in_format(takk,tape)}
  {in_format (takk,vinyl), in_format(boy,tape), in_format(takk,cd)}

Map Colouring



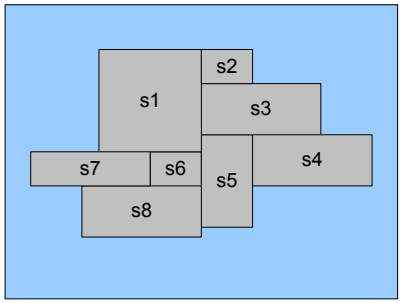
- Problem: find all colourings of a map of countries using not more than 3 colours, such that neighbouring countries are not given the same colour.

Map Colouring



- Problem: find all colourings of a map of countries using not more than 3 colours, such that neighbouring countries are not given the same colour.

ASP – Map Colouring



- Problem: find all colourings of a map using not more than 3 colours, such that neighbouring states are not given the same colour. **Solution 1**:

state(s1). … state(s8).

border(s1,s2). border(s1,s7). … border(s5,s4).

these facts encode the given map (graph).

colour(red). colour(green). colour(blue).

these facts encode the three colours.

painted(S,C) :- state(S), colour(C), not n_painted(S,C).

n_painted(S,C) :- state(S), colour(C), not painted(S,C).

these two rules generate all models resulting from the possible combinations where each state (S) either has a colour (C) (painted(S,C) is true) or doesn't have a colour (painted(S,C) is false, thus n_painted(S,C) is true).

is_painted(S) :- state(S), colour(C), painted(S,C).

:- state(S), not is_painted(S).

predicate is_painted(X) is defined such that it belongs to an answer-set every time state S has at least one colour. Therefore, the two rules eliminate all answer-sets where there is a state without a colour.

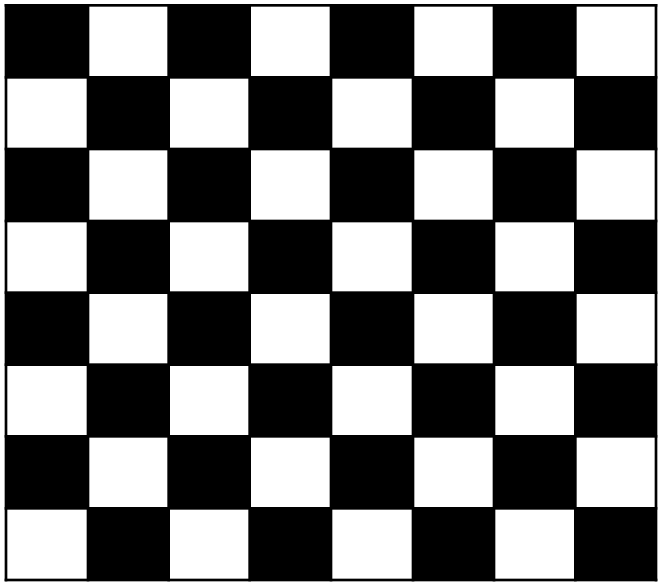
:- state(S), colour(C1), colour(C2), neq(C1,C2), painted(S,C1), painted(S,C2).

this rule eliminates all answer-sets with one state painted with two different colours.

:- state(S1), state(S2), colour(C), border(S1,S2), painted(S1,C), painted(S2,C).

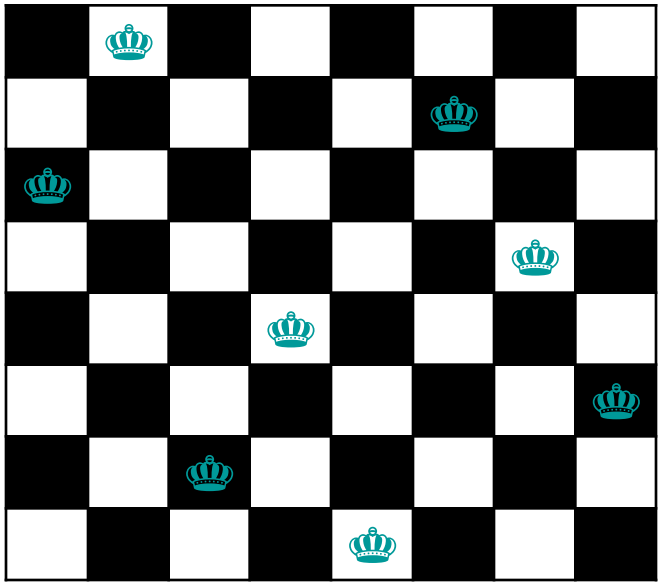
this rule eliminates all answer-sets with two neighbor states painted with the same colour.

8 Queens



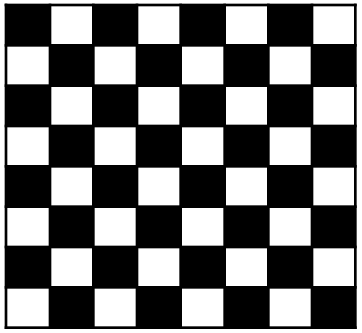
- The 8 queens puzzle is the problem of placing 8 chess queens on an 8×8 chessboard such that none of them is able to capture any other using the standard chess queen's moves. Thus, a solution requires that no two queens share the same row, column, or diagonal.

8 Queens



- The 8 queens puzzle is the problem of placing 8 chess queens on an 8×8 chessboard such that none of them is able to capture any other using the standard chess queen's moves. Thus, a solution requires that no two queens share the same row, column, or diagonal.

ASP – 8 Queens



The 8 queens puzzle is the problem of placing 8 chess queens on an 8×8 chessboard such that none of them is able to capture any other using the standard chess queen's moves. Thus, a solution requires that no two queens share the same row, column, or diagonal. **Solution 1**:

column(1). … column(8). row(1). … row(8).

these facts define the domain of column and row.

in(X,Y) :- column(X), row(Y), not n_in(X,Y).

n_in(X,Y) :- column(X), row(Y), not in(X,Y).

these two rules generate all models resulting from the possible combinations where each cell (X,Y) either has a queen (in(X,Y) is true) or is empty (in(X,Y) is false, thus n_in(X,Y) is true).

has_queen(X) :- column(X), row(Y), in(X,Y).

:- column(X), not has_queen(X).

predicate has_queen(X) is defined such that it belongs to an answer-set every time column X has at least one queen. Therefore, the two rules eliminate all answer-sets with a column without a queen.

:- column(X), row(Y), column(XX), not eq(X,XX), in(X,Y), in(XX,Y).

this rule eliminates all answer-sets with more than one queen in the same row (Y).

:- column(X), row(Y), row(YY), not eq(Y,YY), in(X,Y), in(X,YY).

this rule eliminates all answer-sets with more than one queen in the same column (X).

:- column(X), row(Y), column(XX), row(YY), not eq(X,XX), not eq(Y,YY), in(X,Y), in(XX,YY), eq(abs(X-XX),abs(Y-YY)).

this rule eliminates all answer-sets with more than one queen in the same diagonal.

CLINGO = GRINGO + CLASP

- n(a;b;c).
  - is equivalent to n(a), n(b), n(c).
- const max = 10
  - defines the value of the constant max as being equal to 10.
- s(1..max).
  - is equivalent to s(1), s(2), …, s(10).
- eq(X,Y) ⇔ X=Y; lt(X,Y) ⇔ X<Y; ge(X,Y) ⇔ X>=Y; …
- hide p(_,_).
  - hides atoms of the form p(_,_) from the answer-sets generated by SMODELS;
- hide.
  - hides all atoms from the answer-sets generated by SMODELS;;
- show q(_,_).
  - invalidates a previous hide instruction for atoms of the form q(_,_);

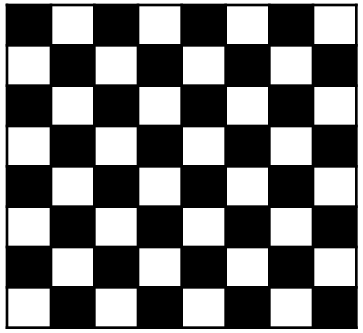
CLINGO = GRINGO + CLASP

- CLLINGO offers a special syntax.
- This syntax allows the selective generation of models, eliminating the need to use:
  - auxiliary atoms,
  - cycles to generate models,
  - some integrity constraints.
- Its use increases efficiency of Smodels, and is strongly advised.
- The general syntax is:

$$\mathbf{min\{p(X_1,\ldots,X_n,Y_1,\ldots,Y_m):q(X_1,\ldots,X_n)\}max :- s(Y_1,\ldots,Y_m).}$$

- Meaning: for each $s(Y_1,\ldots,Y_m)$, generate all possible models with a minimum min and a maximum max of atoms $p(X_1,\ldots,X_n,Y_1,\ldots,Y_m)$, where the domain of $X_1,\ldots,X_n$ is given by $q(X_1,\ldots,X_n)$.
- More details can be found in Lparse manual.

ASP – Rainhas



The 8 queens puzzle is the problem of placing 8 chess queens on an 8×8 chessboard such that none of them is able to capture any other using the standard chess queen's moves. Thus, a solution requires that no two queens share the same row, column, or diagonal. **Solution 2**:

```
column(1..8).
```

defines the domain of column. Equivalent to the facts column(1), column(2),…, column(8).

```
row(1..8).
```

defines the domain of row. Equivalent to the facts row(1), row(2),…, row(8).

```
1{in(X,Y):column(X)}1 :- row(Y).
```

this rule generates all models where each row Y has a queen in exactly one column X, eliminating all other models.

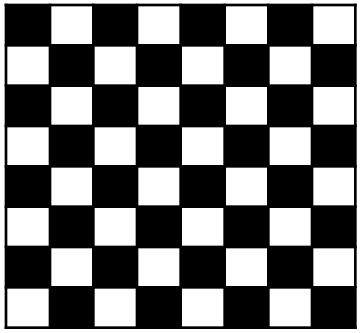
```
:- column(X), row(Y), row(YY), not eq(Y,YY), in(X,Y), in(X,YY).
```

this rule eliminates all answer-sets with more than one queen in the same column (X).

```
:- column(X), row(Y), column(XX), row(YY), not eq(X,XX), not
      eq(Y,YY), in(X,Y), in(XX,YY), eq(abs(X-XX),abs(Y-YY)).
```

this rule eliminates all answer-sets with more than one queen in the same diagonal.

ASP – Rainhas



The 8 queens puzzle is the problem of placing 8 chess queens on an 8×8 chessboard such that none of them is able to capture any other using the standard chess queen's moves. Thus, a solution requires that no two queens share the same row, column, or diagonal. **Solution 3**:

```
column(1..8).
```

defines the domain of column. Equivalent to the facts column(1), column(2),…, column(8).

```
row(1..8).
```

defines the domain of row. Equivalent to the facts row(1), row(2),…, row(8).

```
1{in(X,Y):column(X)}1 :- row(Y).
```

this rule generates all models where each row Y has a queen in exactly one column X, eliminating all other models.

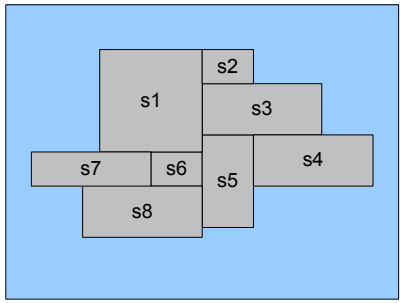
```
1{in(X,Y):row(Y)}1 :- column(X).
```

this rule generates all models where each column X has a queen in exactly one row Y, eliminating all other models.

```
:- column(X), row(Y), column(XX), row(YY), not eq(X,XX), not
      eq(Y,YY), in(X,Y), in(XX,YY), eq(abs(X-XX),abs(Y-YY)).
```

this rule eliminates all answer-sets with more than one queen in the same diagonal.

ASP – Map Colouring



- Problem: find all colourings of a map using not more than 3 colours, such that neighbouring states are not given the same colour. **Solution 2**:

```
state(s1). … state(s8).
border(s1,s2). border(s1,s7). … border(s5,s4).
```

these facts encode the given map (graph).

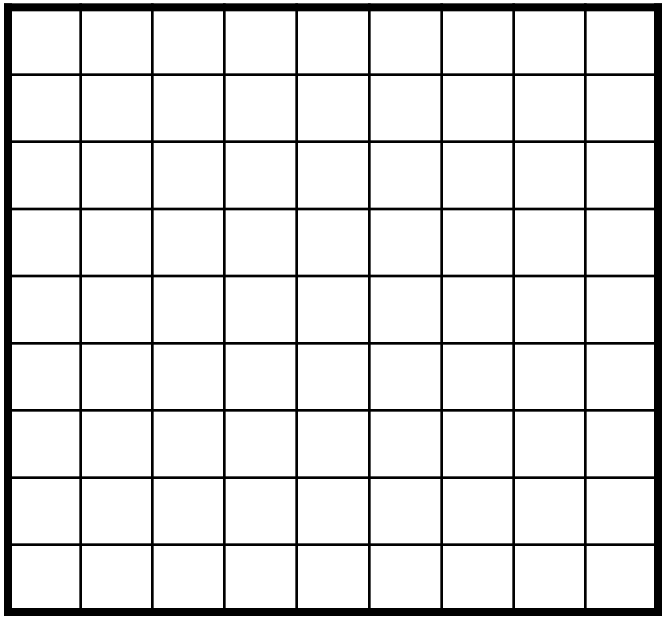
```
1{painted(S,C):colour(C)}1 :- state(S).
```

this rule generates all models where each state has exactly one colour.

```
:- state(S1), state(S2), colour(C), border(S1,S2),
   painted(S1,C), painted(S2,C).
```

this rule eliminates all answer-sets with two neighbor states painted with the same colour.

Latin Squares



- The Latin square puzzle is the problem of placing n different symbols (e.g. numbers), is an n × n table, in such a way that each symbol occurs exactly once in each row and exactly once in each column.

Latin Squares

8	1	6	3	7	4	5	9	2
3	9	4	5	2	1	7	8	6
2	5	7	9	6	8	3	4	1
6	7	9	4	3	2	1	5	8
1	2	8	7	9	5	4	6	3
5	4	3	1	8	6	9	2	7
9	6	1	8	5	3	2	7	4
4	8	5	2	1	7	6	3	9
7	3	2	6	4	9	8	1	5

- The Latin square puzzle is the problem of placing n different symbols (e.g. numbers), is an n × n table, in such a way that each symbol occurs exactly once in each row and exactly once in each column.

ASP – Latin Squares

- The Latin square puzzle is the problem of placing n different symbols (e.g. numbers), is an n × n table, in such a way that each symbol occurs exactly once in each row and exactly once in each column.
- Solution 1:

```
const size = 10.
```

  - defines the constant size = 10

```
n(1.. size).
```

  - defines the domain of n. Equivalent to the facts n(1),n(2),…,n(size).

```
1{in(X,Y,N):n(N)}1:- n(Y),n(X).
```

  - this rule generates all models where each cell (X,Y) is filled with exactly one number N, eliminating all other models.

```
:- n(X;XX;Y;N), in(X,Y,N), in(XX,Y,N), not eq(X,XX).
```

  - this rule eliminates all models where the same number N is placed in more than one column (X and XX) in the same row Y.

```
:- n(X;YY;Y;N), in(X,Y,N), in(X,YY,N), not eq(Y,YY).
```

  - this rule eliminates all models where the same number N is placed in more than one row (Y and YY) in the same column Y.

ASP – Latin Squares

- The Latin square puzzle is the problem of placing n different symbols (e.g. numbers), is an n × n table, in such a way that each symbol occurs exactly once in each row and exactly once in each column.
- Solution 2:

```
const size = 10.
```

  - defines the constant size = 10

```
n(1.. size).
```

  - defines the domain of n. Equivalent to the facts n(1),n(2),…,n(size).

```
1{in(X,Y,N):n(N)}1:- n(Y),n(X).
```

  - this rule generates all models where each cell (X,Y) is filled with exactly one number N, eliminating all other models.

```
1{in(X,Y,N):n(Y)}1:- n(X),n(N).
```

  - this rule generates all models where in each column (X), each number N is placed in exactly in one row (Y), eliminating all other models.

```
1{in(X,Y,N):n(X)}1:- n(Y),n(N).
```

  - this rule generates all models where in each row (Y), each number N is placed in exactly in one column (X), eliminating all other models.

Sudoku

		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		

- The Sudoku puzzle is the problem of filling a partially completed 9x9 grid so that each column, each row, and each of the nine 3x3 boxes (also called blocks or regions) contains the digits from 1 to 9.

Sudoku

8	1	6	3	7	4	5	9	2
3	9	4	5	2	1	7	8	6
2	5	7	9	6	8	3	4	1
6	7	9	4	3	2	1	5	8
1	2	8	7	9	5	4	6	3
5	4	3	1	8	6	9	2	7
9	6	1	8	5	3	2	7	4
4	8	5	2	1	7	6	3	9
7	3	2	6	4	9	8	1	5

- The Sudoku puzzle is the problem of filling a partially completed 9x9 grid so that each column, each row, and each of the nine 3x3 boxes (also called blocks or regions) contains the digits from 1 to 9.

ASP – Sudoku

- The Sudoku puzzle is the problem of filling a partially completed 9x9 grid so that each column, each row, and each of the nine 3x3 boxes (also called blocks or regions) contains the digits from 1 to 9.
- Sudoku is a particular case of the Latin Squares with n=9, to which we add the regions constraint. We can start with the Latin Square specification and simply add the new constraint. Solution:

```
n(1.. 9).
1{in(X,Y,N):n(N)}1:- n(Y),n(X).
1{in(X,Y,N):n(Y)}1:- n(X),n(N).
1{in(X,Y,N):n(X)}1:- n(Y),n(N).
v(1;4;7).
```

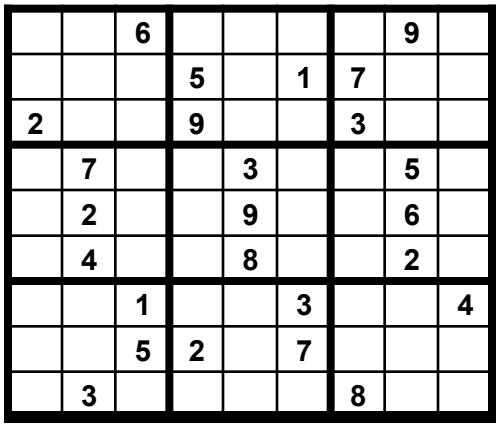
this fact defines the coordinates of the lower left cell of each 3x3 region. Equivalent to the facts v(1),v(4), e v(7).

```
:- n(X;Y;X1;Y1;N), v(VX;VY), neq(X,X1), neq(Y,Y1), in(X,Y,N), in(X1,Y1,N),
   X-VX < 3, X1-VX < 3, Y-VY < 3, Y1-VY < 3,
   X >= VX, X1 >= VX, Y >= VY, Y1 >= VY.
```

This rule eliminates all models where the same number N is placed in more than one cell in the same region.

Note that this rule only verifies pairs of cells where both the column and row are different, since other cases are already treated by the rules above.

ASP – Sudoku



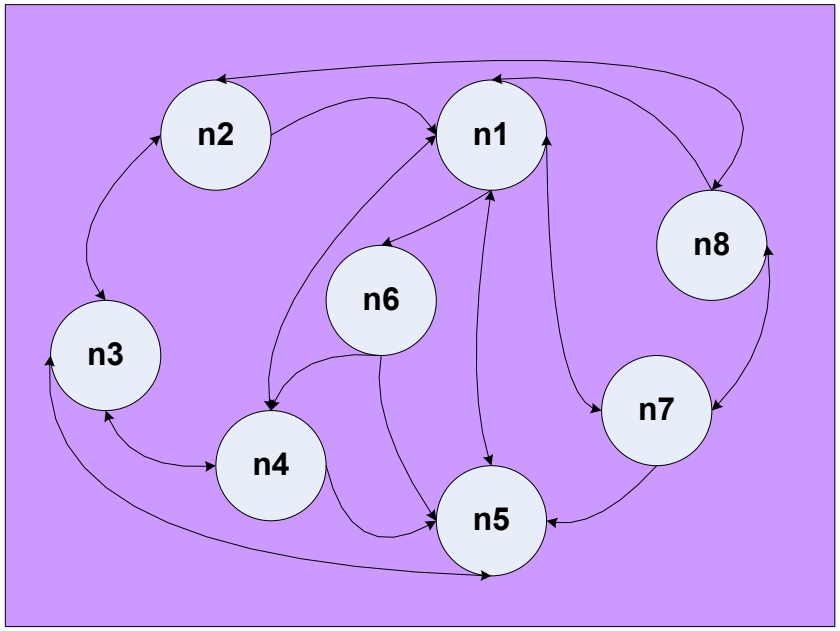
		6					9	
			5		1	7		
2			9			3		
	7			3			5	
	2			9			6	
	4			8			2	
		1			3			4
		5	2		7			
	3					8		

- Now, we only have to add facts for the cells that are already filled.

```
n(1.. 9).
1{in(X,Y,N):n(N)}1:- n(Y),n(X).
1{in(X,Y,N):n(Y)}1:- n(X),n(N).
1{in(X,Y,N):n(X)}1:- n(Y),n(N).
v(1;4;7).
:- n(X;Y;X1;Y1;N), v(VX;VY), neq(X,X1), neq(Y,Y1), in(X,Y,N), in(X1,Y1,N),
    X-VX < 3, X1-VX < 3, Y-VY < 3, Y1-VY < 3,
    X >= VX, X1 >= VX, Y >= VY, Y1 >= VY.
```

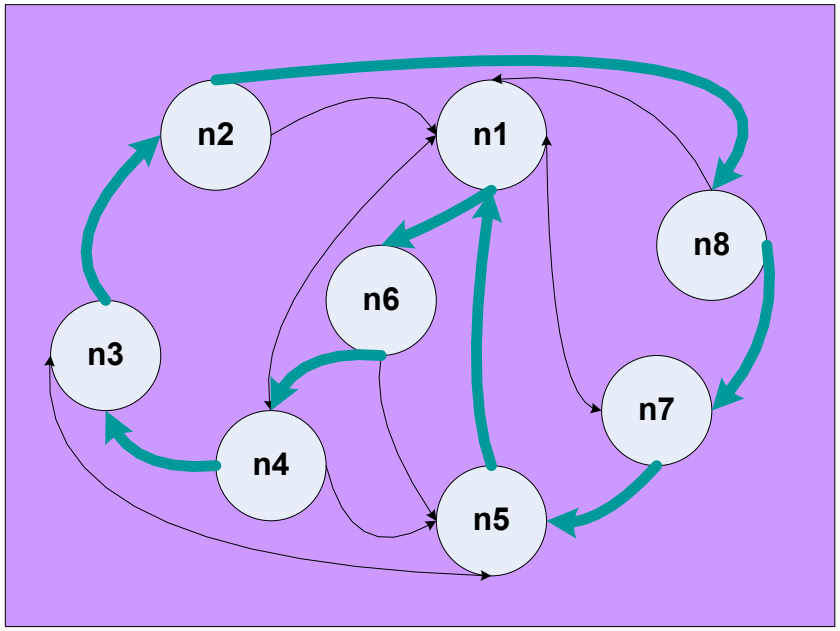
```
in(1,7,2).  in(2,1,3).  in(2,4,4).  in(2,5,2).  in(2,6,7).
in(3,2,5).  in(3,3,1).  in(3,9,6).  in(4,2,2).  in(4,7,9).
in(4,8,5).  in(5,4,8).  in(5,5,9).  in(5,6,3).  in(6,2,7).
in(6,3,3).  in(6,8,1).  in(7,1,8).  in(7,7,3).  in(7,8,7).
in(8,4,2).  in(8,5,6).  in(8,6,5).  in(8,9,9).  in(9,3,4).
```

Hamiltonian Cycles



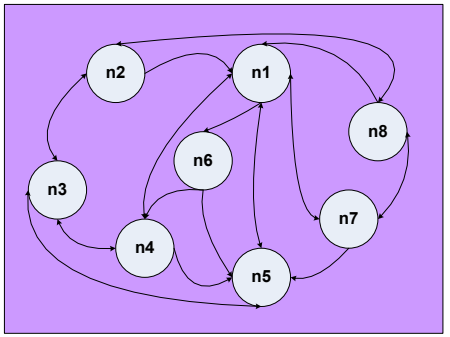
- Problem: given a directed graph, find all Hamiltonian Cycles i.e. all cycles that touch each node exactly once.

Hamiltonian Cycles



- Problem: given a directed graph, find all Hamiltonian Cycles i.e. all cycles that touch each node exactly once.

ASP – Hamiltonian Cycles



- Problem: given a directed graph, find all Hamiltonian Cycles i.e. all cycles that touch each node exactly once.

node(n1). … node(n8).

edge(n1,n5). edge(n4,n3). … edge(n7,n8).

these facts encode the given graph.

1 {in(X,Y) : edge(X,Y)} 1 :- node(X).

this rule generates all models where the cycle contains, for each node, exactly one edge going out.

1 {in(X,Y) : edge(X,Y)} 1 :- node(Y).

this rule generates all models where the cycle contains, for each node, exactly one edge going in.

reachable(X) :- node(X), in(n1,X).

reachable(Y) :- node(X), node(Y), reachable(X), in(X,Y).

these rules define the notion of reachability, starting from node n1.

:- not reachable(X), node(X).

this rule eliminates all answer-sets where some node is not reachable.

Stable Models - Example

- The albums program, using the syntax of Lparse:

  album(takk;boy).

  format(cd;tape;vinyl).

  1{in_format(A,F):format(F)} :- album(A).

  - This rule generates all models where each album has one or more formats (absence of a max value indicates that there is no upper limit for the number of atoms in_format/2), eliminating all models where an album does not have any format.

  :- album(A), in_format(A,tape), in_format(A,cd).

  in_format(takk,vinyl).

  :- album(A1), album(A2), format(F), in_format(A1,F), in_format(A2,F), neq(A1,A2).

  ok :- album(A1), album(A2), in_format(A1,cd), in_format(A2,tape).

  :- not ok.

- The stable models are (omitting atoms album/1, format/1):

  {in_format (takk,vinyl), in_format(boy,cd), in_format(takk,tape)}

  {in_format (takk,vinyl), in_format(boy,tape), in_format(takk,cd)}