Programming Paradigms: Logic Programming

Modelling, Part 1

Víctor Gutiérrez Basulto

Language constructs

- Intervals `M .. N`
  - `grid(1..S,1..S):- size(S).`
  - `grid(X,Y):- X=1..S,Y=1..S, X-Y!=0,X+Y-1!=S.`
- Conditional literals `p(X) :- q(X) : r(X)`
  - `meet:- available(X) : person(X)`
  - `on(X) : day(X):-meet.`

Language constructs

- Choice (Cardinality Constraints)
  - `1{p(1 .. 3)}2.`
  - `1 { has_property(X,C) : property(C)} 1 :- item(X).`
- Aggregates
  - `20 <= # sum {4 : course (db) ; 6 course(ai); 8 : course(project); 3 : course(xml)}`
  - `#sum { 3 : bananas; 25 : cigars; 10 : broom } <= 30`
  - `within_budget :- # sum 10 {Amount : paid(Amount)} 100.`

Language constructs

- Aggregates `#count{ ··· }`
  - `many_neighbors(X):-vertex(X), #count{Y : adjacent (X,Y)} >3.`
- Disjunction `p(X) ; q(X) :- r(X)`
  - if `r(X)` then `p(X)` **or** `q(X)`
- Integrity constraints `:- q(X), p(X)`
  - `:- in_clique(X), in_clique(Y), not edge(X,Y).`
  - this constraint says: it **cannot** be the case that nodes X and Y are in a clique, *and* there is no edge between X and Y.

Language constructs

- Multi-objective oOptimization
  - Weak constraints
  - Statements

```
:~ q(X), p(X,C) [C]
#minimize { C : q(X), p(X,C) }
#maximize { 1,X:in_clique(X), node(X) }.
```

- `noisy :- hotel(X), main_street(X).`
- `#maximize { Y@1,X : hotel(X), star(X,Y) }.`
- `#minimize { Y / Z@2,X : hotel(X), cost(X,Y), star(X,Z) }.`
- `noisy. [ 1@3 ]`

Language constructs

- Multi-objective oOptimization
  - Weak constraints `:~ q(X), p(X,C) [C@42]`
  - Statements `#minimize { C@42 : q(X), p(X,C) }`

```
#maximize { 1,X:in_clique(X), node(X) }.
```

- `noisy :- hotel(X), main_street(X).`
- `#maximize { Y@1,X : hotel(X), star(X,Y) }.`
- `#minimize { Y / Z@2,X : hotel(X), cost(X,Y), star(X,Z) }.`
- `noisy.   [ 1@3 ]`

Language constructs

- Arithmetic Functions
  - $+$ (addition), $-$ (subtraction), $*$ (multiplication), $/$ (integer division), $\backslash$ (modulo), $**$ (exponentiation), $|\cdot|$ (absolute value).
- Comparison Predicates
  - $=$ (equal), $!=$ (not equal), $<$ (less than), $<=$ (less than or equal), $>$ (greater than), and $>=$ (greater than or equal).

Language constructs: Lab 2

Marina, Willem, Bob, Tina, Bert, Jane, and Alyssa are invited at a wedding.

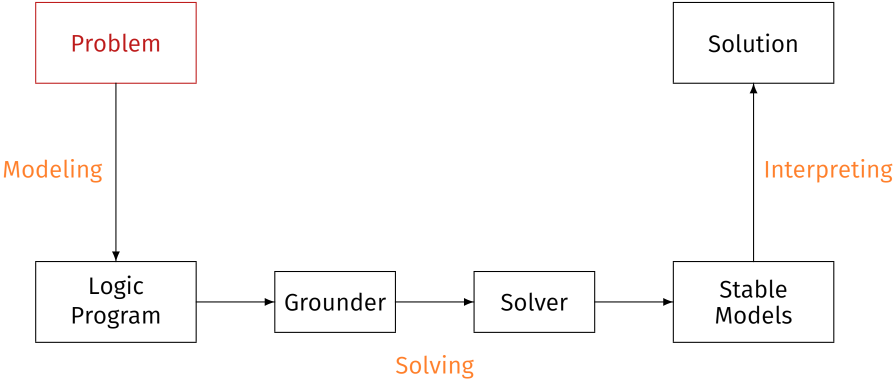
Write a program to identify the possible allocations for them knowing that:

- there are three tables;
- each table should have at least two people;
- each table cannot have more than three people;
- Marina does not want to seat with Willem.

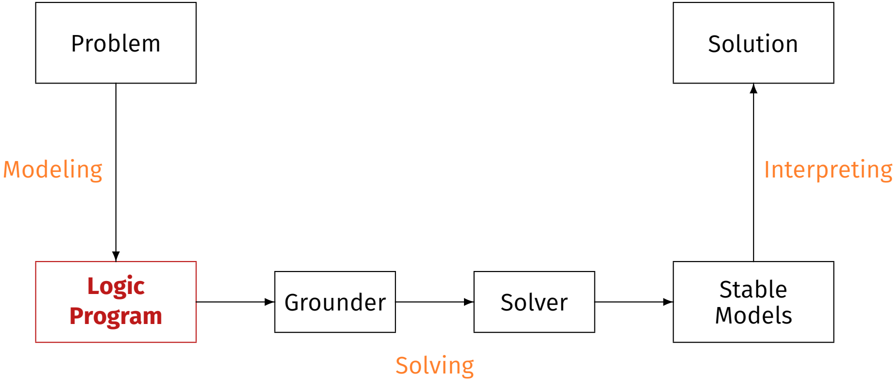
Language constructs: Lab 2

```
person(marina;willem;bob;tina;bert;jane;alyssa).

table(1..3).

1{seating(P,T) : table(T)}1 :- person(P).

2{seating(P,T) : person(P)}3 :- table(T).

:- seating(marina, T), seating(willem, T).
```

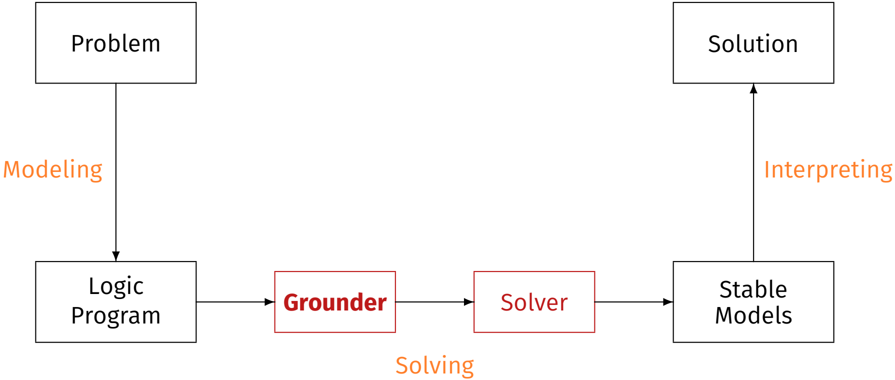
ASP solving process



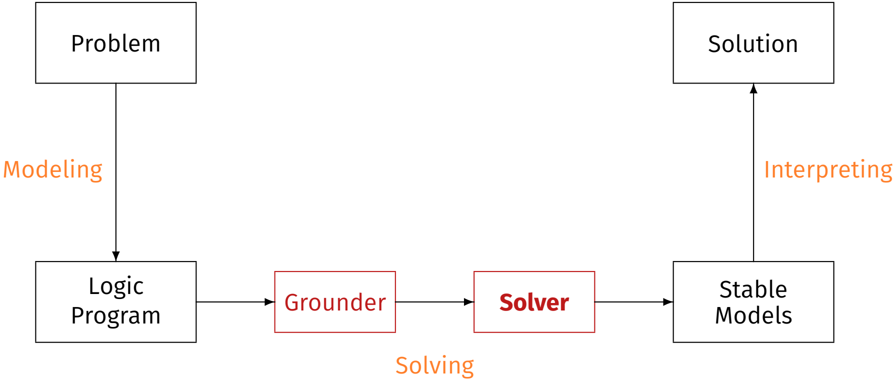
ASP solving process



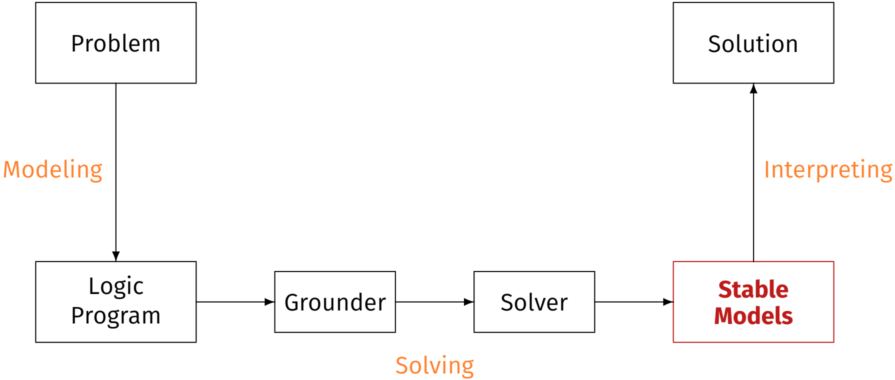
ASP solving process



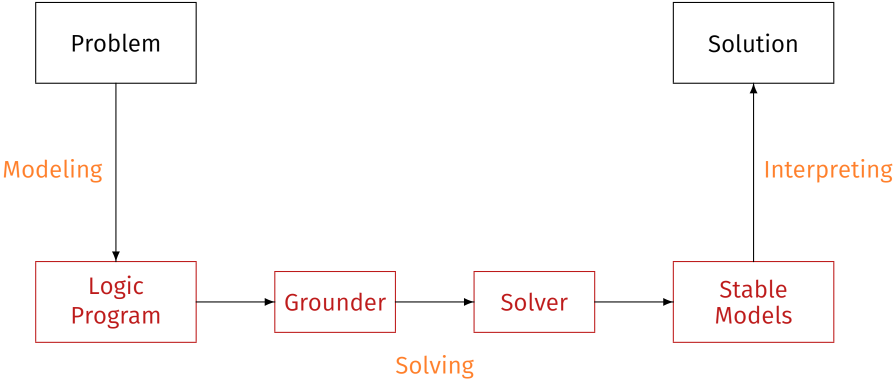
ASP solving process



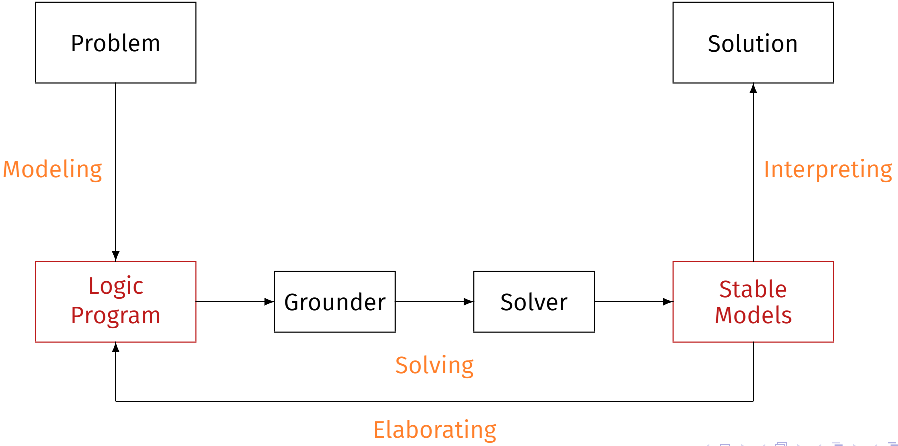
ASP solving process



ASP solving process



ASP solving process



Basic methodology for writing your program

Methodology

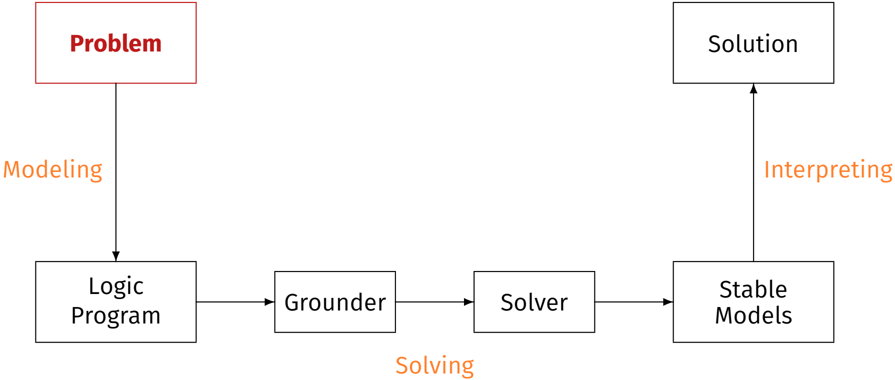
Generate and Test (or: Guess and Check)

Generator: Generate potential stable model candidates (typically through non-deterministic constructs)

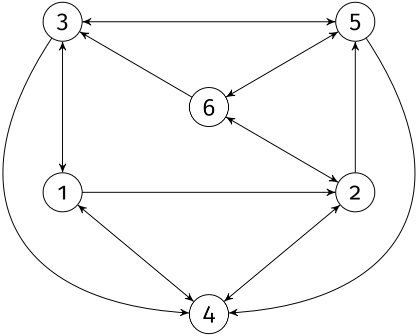
Tester: Eliminate invalid candidates (typically through integrity constraints)

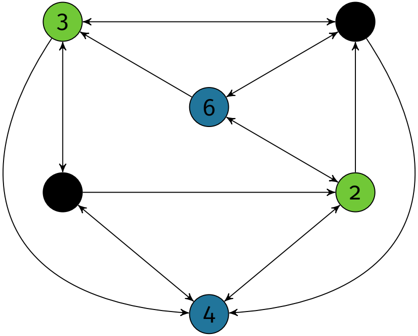
Logic program = Data + Generator + Tester (+ Optimizer)

A case-study: Graph colouring



Graph colouring



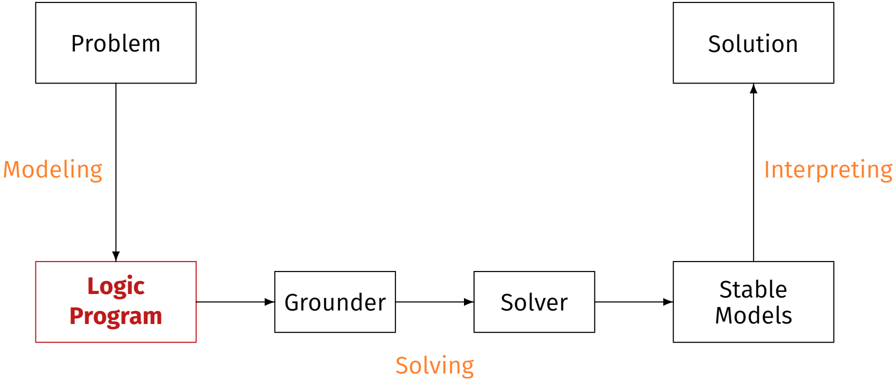


Graph colouring

- Problem instance A graph consisting of nodes and edges
  - facts formed by predicates `node/1` and `edge/2`
  - facts formed by predicate `colour/1`
- Problem class Assign each node one colour such that no two nodes connected by an edge have the same colour

  In other words,
  1. Each node has exactly one colour
  2. Two connected nodes must not have the same colour

ASP solving process



Graph colouring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).
```

Problem instance

```
1 { colour(N,C) : col(C) } 1 :- node(N).

:- edge(N,M), colour(N,C), colour(M,C).
```

Problem encoding

Graph colouring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).
```

Problem instance

```
1 { colour(N,C) : col(C) } 1 :- node(N).

:- edge(N,M), colour(N,C), colour(M,C).
```

Problem encoding

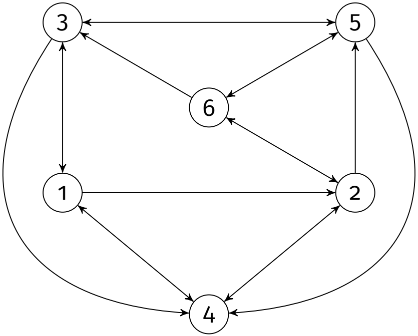
Choice rule

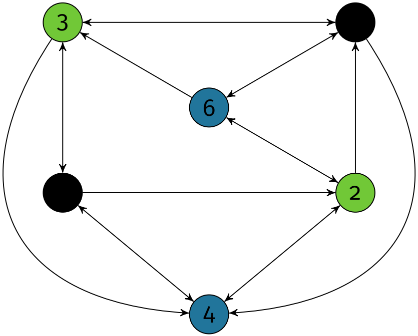
- Idea Choices over subsets
- Syntax A choice rule is of the form

$$\{a_1, \dots, a_m\} \leftarrow a_{m+1}, \dots, a_n, {\sim}a_{n+1}, \dots, {\sim}a_o$$

  where $0 \leq m \leq n \leq o$ and each a_i is an atom for $1 \leq i \leq o$

- Informal meaning If the body is satisfied by the stable model at hand, then any subset of $\{a_1, \dots, a_m\}$ can be included in the stable model
- Example `{ buy(pizza), buy(wine), buy(corn) } :- at(grocery).`
- Another example $P = \{\{a\} \leftarrow b,\ b \leftarrow\}$ has two stable models: $\{b\}$ and $\{a, b\}$


3
5
6
1
2
4


3
6
2
4

Graph colouring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).
```

Problem instance

```
1 { colour(N,C) : col(C) } 1 :- node(N).

:- edge(N,M), colour(N,C), colour(M,C).
```

Problem encoding

Cardinality rule

- Idea Control (lower) cardinality of subsets
- Syntax A cardinality rule is the form

$$a_0 \leftarrow l \; \{ \; a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \; \}$$

where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$; l is a non-negative integer.

- Informal meaning The head atom belongs to the stable model, if at least l elements of the body are included in the stable model
- Example `pass(c42) :- 2 { pass(a1); pass(a2); pass(a3) }.`
- Another example $P = \{a \leftarrow 1\{b, c\}, \; b \leftarrow\}$ has stable model $\{a, b\}$

Cardinality rules with upper bounds

- A rule of the form

$$a_0 \leftarrow l \; \{ \; a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n \; \} \; u \tag{1}$$

  where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$;
  l and u are non-negative integers

- Note The single constraint in the body of the cardinality rule (1) is referred to as a cardinality constraint

Cardinality constraints

- Syntax A cardinality constraint is of the form

$$l \ \{ \ a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n \ \} \ u$$

  where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$;
  l and u are non-negative integers

- Informal meaning A cardinality constraint is satisfied by a stable model X, if the number of its contained literals satisfied by X is between l and u (inclusive)
- In other words, if

$$l \leq |\,(\{a_1, \dots, a_m\} \cap X) \cup (\{a_{m+1}, \dots, a_n\} \setminus X)\,| \leq u$$

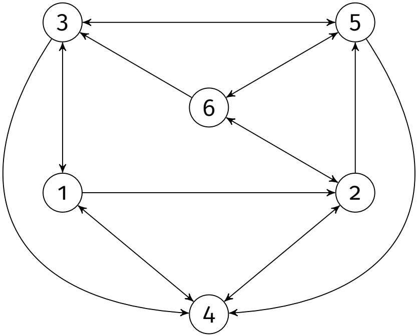
Cardinality constraints as heads

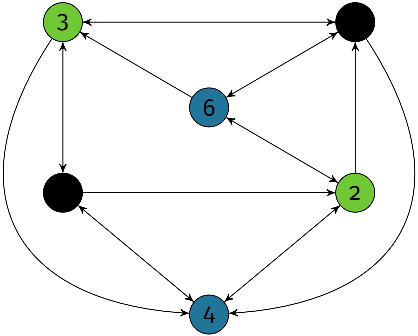
- A rule of the form

$$l \; \{a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n\} \; u \leftarrow a_{n+1}, \dots, a_o, \sim a_{o+1}, \dots, \sim a_p$$

  where $0 \leq m \leq n \leq o \leq p$ and each a_i is an atom for $1 \leq i \leq p$;
  l and u are non-negative integers

- Example `1{ color(v42,red); color(v42,green); color(v42,blue) }1.`


3
5
6
1
2
4


3
6
2
4

Graph colouring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).
```

Problem instance

```
1 { colour(N,C) : col(C) } 1 :- node(N).

:- edge(N,M), colour(N,C), colour(M,C).
```

Problem encoding

Conditional literals

- Syntax A conditional literal is of the form

$$l : l_1, \dots, l_n$$

  where l and l_i are literals for $0 \leq i \leq n$

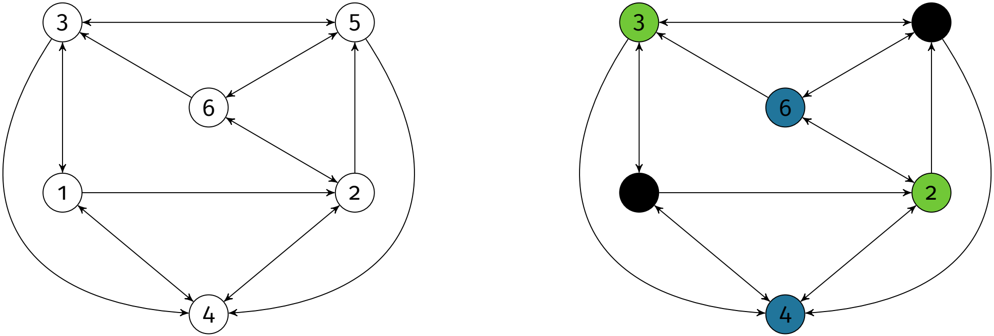
- Informal meaning A conditional literal can be regarded as the list of elements in the set $\{l \mid l_1, \dots, l_n\}$

- Note The expansion of conditional literals is context dependent
- Example Given '`color(red). color(green). color(blue)`'

```
:- color(v42,C) : color(C).
```

  is instantiated to

```
:- color(v42,red), color(v42,green), color(v42,blue).
```


3
5
6
1
2
4
3
6
2
4

Graph colouring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).
```

Problem instance

```
1 { colour(N,C) : col(C) } 1 :- node(N).

:- edge(N,M), colour(N,C), colour(M,C).
```

Problem encoding

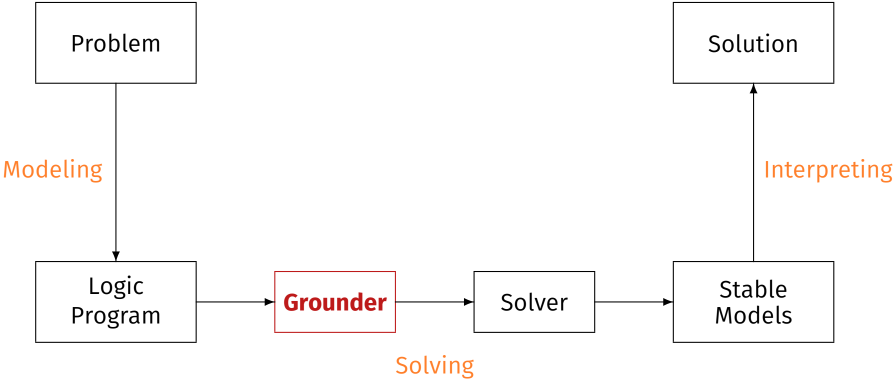
Integrity constraint

- Idea Eliminate unwanted solution candidates
- Syntax An integrity constraint is of the form

$$\leftarrow a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n$$

  where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$
- Example `:- edge(3,7), color(3,red), color(7,red).`

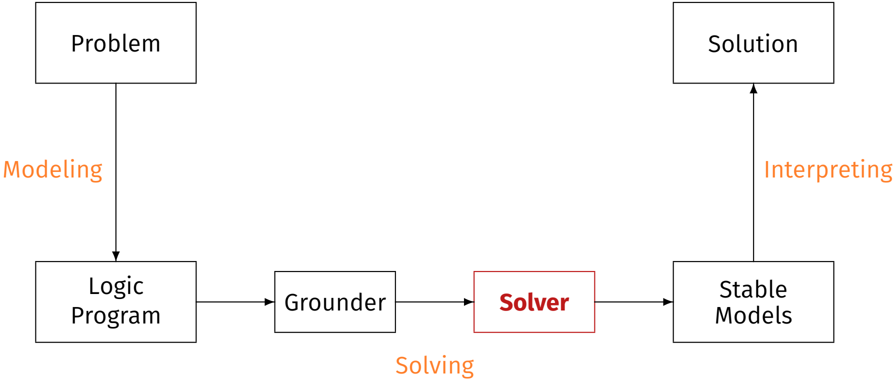
ASP solving process



Graph colouring: Grounding

```
$ clingo --mode=gringo --text graphcolour.lp

node(1).  node(2).  node(3).  node(4).  node(5).  node(6).

edge(1,2).  edge(2,4).  edge(3,1).  edge(4,1).  edge(5,3).  edge(6,2).
edge(1,3).  edge(2,5).  edge(3,4).  edge(4,2).  edge(5,4).  edge(6,3).
edge(1,4).  edge(2,6).  edge(3,5).              edge(5,6).  edge(6,5).

col(r).  col(b).  col(g).

#delayed(1).
#delayed(1) <=> 1<=#count0,colour(1,r):colour(1,r);0,colour(1,g):colour(1,g);0,colour(1,b):colour(1,b)<=1
[...]

:- colour(1,r),colour(2,r).  :- colour(2,r),colour(4,r). [...] :- colour(6,r),colour(2,r).
:- colour(1,b),colour(2,b).  :- colour(2,b),colour(4,b).       :- colour(6,b),colour(2,b).
:- colour(1,g),colour(2,g).  :- colour(2,g),colour(4,g).       :- colour(6,g),colour(2,g).
:- colour(1,r),colour(3,r).  :- colour(2,r),colour(5,r).       :- colour(6,r),colour(3,r).
:- colour(1,b),colour(3,b).  :- colour(2,b),colour(5,b).       :- colour(6,b),colour(3,b).
:- colour(1,g),colour(3,g).  :- colour(2,g),colour(5,g).       :- colour(6,g),colour(3,g).
:- colour(1,r),colour(4,r).  :- colour(2,r),colour(6,r).       :- colour(6,r),colour(5,r).
:- colour(1,b),colour(4,b).  :- colour(2,b),colour(6,b).       :- colour(6,b),colour(5,b).
:- colour(1,g),colour(4,g).  :- colour(2,g),colour(6,g).       :- colour(6,g),colour(5,g).
```

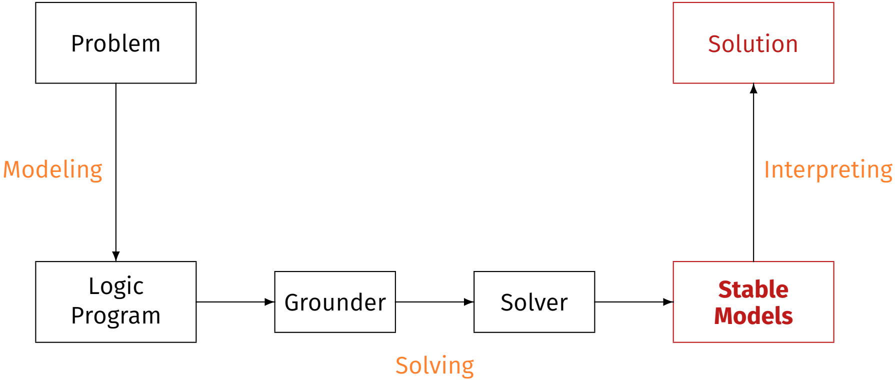
ASP solving process



Graph colouring: Solving

```
$ clingo -n 0 graphcolour.lp

Solving...
Answer: 1
node(1) [...] colour(6,b) colour(5,g) colour(4,b) colour(3,r) colour(2,r) colour(1,g)
Answer: 2
node(1) [...] colour(6,r) colour(5,g) colour(4,r) colour(3,b) colour(2,b) colour(1,g)
Answer: 3
node(1) [...] colour(6,g) colour(5,b) colour(4,g) colour(3,r) colour(2,r) colour(1,b)
Answer: 4
node(1) [...] colour(6,r) colour(5,b) colour(4,r) colour(3,g) colour(2,g) colour(1,b)
Answer: 5
node(1) [...] colour(6,g) colour(5,r) colour(4,g) colour(3,b) colour(2,b) colour(1,r)
Answer: 6
node(1) [...] colour(6,b) colour(5,r) colour(4,b) colour(3,g) colour(2,g) colour(1,r)
SATISFIABLE

Models      : 6
```

ASP solving process

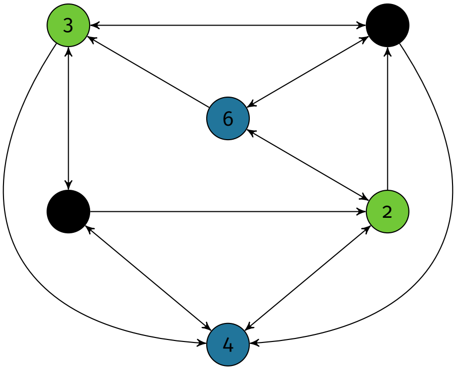


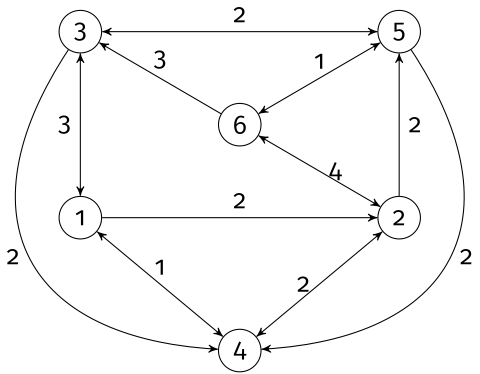
A colouring

```
Answer: 6
node(1)   [...]    \
assign(6,b) assign(5,r) assign(4,b) assign(3,g) assign(2,g) assign(1,r)
```

A colouring

```
Answer: 6
node(1)   [...]    \
assign(6,b) assign(5,r) assign(4,b) assign(3,g) assign(2,g) assign(1,r)
```





Decide the round trip visiting each node in a graph exactly once (aka Hamiltonian cycle) such that accumulated edge costs is minimal.
Travelling Salesperson

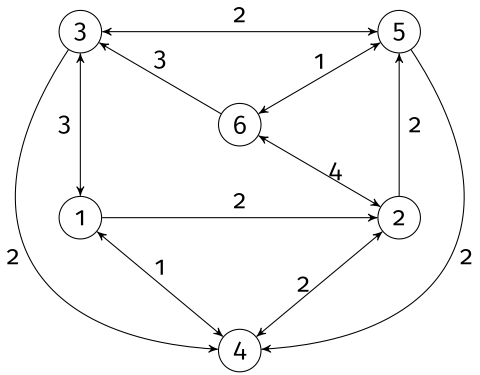
Travelling Salesperson

```
cost(1,2,2).  cost(1,3,3).  cost(1,4,1).
cost(2,4,2).  cost(2,5,2).  cost(2,6,4).
cost(3,1,3).  cost(3,4,2).  cost(3,5,2).
cost(4,1,1).  cost(4,2,2).
cost(5,3,2).  cost(5,4,2).  cost(5,6,1).
cost(6,2,4).  cost(6,3,3).  cost(6,5,1).

edge(X,Y) :- cost(X,Y,_).
node(X) :- cost(X,_,_).   node(Y) :- cost(_,Y,_).
```

Travelling Salesperson

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

reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y), reached(X).

:- node(Y), not reached(Y).

#minimize { C,X,Y : cycle(X,Y), cost(X,Y,C) }.

#show cycle/2.
```

```
Answer: 1
cycle(1,4) cycle(4,2) cycle(3,1) cycle(2,6) cycle(6,5) cycle(5,3)
Optimization: 13
Answer: 2
cycle(1,4) cycle(4,2) cycle(3,1) cycle(2,5) cycle(6,3) cycle(5,6)
Optimization: 12
Answer: 3
cycle(1,2) cycle(4,1) cycle(3,4) cycle(2,5) cycle(6,3) cycle(5,6)
Optimization: 11
OPTIMUM FOUND
```


2
3
5
3
1
3
6
2
4
1
2
2
2
2
1
2
2
4

Sum-Free

- Problem: A set X of numbers is called *sum-free* if the sum of two elements of X never belongs to X.

  For instance, the set $\{5, \dots, 9\}$ is sum-free; the set $\{4, \dots, 9\}$ is not ($4 + 4 = 8$, $4 + 5 = 9$).

- Can we partition the set $\{1, \dots, n\}$ into 2 sum-free subsets? This is possible if n = 4: both $\{1, 4\}$ and $\{2,3\}$ are sum-free. But if n = 5 then such a partition does not exist.

Exercises

```
% Partition { 1,..., n } into r sum-free sets
% Input: in/2 representing partitions, pos. integers n, r

1{in(I, 1..r)}1 :- I = 1..n.
% achieved: set { 1,...,n} partitioned into subsets
{I:in(I,1)}, ..., { I:in(I,r)}
```

TO DO

```
% Achieve these subsets are sum-free
```

- Say we save the solution in `solution.pl`

```
clingo -c r= somenumber1  -c n = somenumber2  solution.pl
```

Independent Sets

- Def. A set S of vertices in a graph is independent if no two vertices from S are adjacent.

```
% Find independent sets of vertices of size n
% Input: set node/1 of vertices of a graph G;
% set edge/2 of edges of G, positive integer n.

n {in(X) : node(X)}n.
% achieved : in/1 is a set consisting of n vertices
```

TO DO

```
% achieved: in/1 has no pairs of adjacent vertices
# show in/1.
```

Clique

- Def. A set S of vertices in a graph is called a clique if every two distinct vertices in it are adjacent.

TO DO Modify the (completed) program for independent sets to describe cliques of size n.

Number of Vertices and Edges

- Def. The degree of a node X is the number of nodes adjacent to X.

```
% Find the number of edges and degrees of vertices
% Input: set of nodes/1 of vertices of a graph G; set
edge/2 of edges of G.

adj(X,Y) :- edge(X,Y).
adj(X,Y) :- edge(Y,X).
% achieved: adjacent (X,Y) iff X,Y are adjacent in the
graph.
```

- HINT: use `#count`

Mandatory reading: Sections 3.1 and 3.3 of of Answer Set Solving in Practice, by Gebser, Kaminski, Kaufmann, Schaub