Reasoning and Planning
Unit 4. Relational Reasoning

Pedro Cabalar

Dept. Computer Science
University of Corunna, SPAIN

October 31, 2022

Relational Representation

- Atoms = instead of propositions, we have now predicates.
  They represent relations among entities:

```
neighbour(france,spain).    exports(germany,france,cars).
```

- Herbrand Domain = set of individuals, each one uniquely identified by a (lowercase) constant name. E.g.
  $D = \{\mathtt{germany}, \mathtt{france}, \mathtt{spain}, \mathtt{cars}, \dots\}$
  $\cup \{0, 1, 2, \dots, -1, -2, \dots\}$ plus finite subsets of integer numbers.
- Unique Names Assumption (UNA) =
  different terms represent different individuals.
  $\mathtt{spain} \neq \mathtt{france}$, $\mathtt{spain} \neq \mathtt{cars}$, $\mathtt{spain} \neq \mathtt{españa}$, $0 \neq 1$,
  $0 \neq \mathtt{cars}$
- We can use unary predicates to represent types:

```
country(spain).  country(france).  country(germany).
tradegood(cars).  tradegood(food).
```

```
country(spain;france;germany).
```

Relational Representation

- A set of facts becomes the extensional database (EDB)!

```
neighbour(spain,france).
neighbour(france,germany).
exports(spain,germany,food).
exports(spain,france,food).
exports(germany,france,cars).
exports(france,spain,cars).
```

Table `neighbour`

C1	C2
spain	france
france	germany

Table `exports`

FROM	TO	GOOD
spain	germany	food
spain	france	food
germany	france	cars
france	spain	cars

Relational Representation

- A query to the EDB becomes a rule with variables.
  Variable = name with upcase initial (`X`, `Y`, `Country`, ...)
  universally quantified and denoting arbitrary individuals.
  '_' = anonymous variable (different each time it occurs)

```
exgood(G) :- exports(_,_,G). exgood(G) :- exports(X1,X2,G).
```

$$\forall X1, X2, G\ (exports(X1, X2, G) \rightarrow exgood(G))$$

- Ex.: "neighbours of France and goods she imports from them"

```
answer(N,G) :- neighbour(france,N),exports(N,france,G).
```

  SQL equivalent is more verbose

```
SELECT neighbour.C2, exports.GOOD FROM neighbour
INNER JOIN exports ON neighbour.C2=exports.FROM
WHERE neighbour.C1=france AND exports.TO=france;
```

  Problem: we get no goods from Spain using our previous data!
  We had `neighbour(spain,france)` but not the opposite!

Deductive Databases

- Predicate `neighbour` should be symmetric! We add a rule

```
neighbour(X,Y) :- neighbour(Y,X).
```

- Deductive database: some predicates are intensional or (partially) deduced from rules, rather than extensional (list of facts).

- Ground atom = predicate + constants, no variables.
  Grounding = replacing variables by all their possible instances.
  (although it is actually more intelligent than that)

Example: the grounding of program

```
neighbour(spain,france). neighbour(france,germany).
neighbour(X,Y) :- neighbour(Y,X).
```

would potentially yield the rules

```
neighbour(spain,france). neighbour(france,germany).
neighbour(spain,france)   :- neighbour(france,spain).
neighbour(spain,germany)  :- neighbour(germany,spain).
neighbour(france,spain)   :- neighbour(spain,france).
neighbour(france,germany) :- neighbour(germany,france).
neighbour(germany,spain)  :- neighbour(spain,germany).
neighbour(germany,france) :- neighbour(france,germany).
```

Example: the grounding of program

```
neighbour(spain,france). neighbour(france,germany).
neighbour(X,Y) :- neighbour(Y,X).
```

would potentially yield the rules, but in practice ...

```
neighbour(spain,france). neighbour(france,germany).
neighbour(spain,france)   :- neighbour(france,spain).
neighbour(spain,germany)  :- neighbour(germany,spain).
neighbour(france,spain)   :- neighbour(spain,france).
neighbour(france,germany) :- neighbour(germany,france).
neighbour(germany,spain)  :- neighbour(spain,germany).
neighbour(germany,france) :- neighbour(france,germany).
```

Deductive Databases

Example: the grounding of program

```
neighbour(spain,france). neighbour(france,germany).
neighbour(X,Y) :- neighbour(Y,X).
```

would potentially yield the rules, but in practice . . .

```
neighbour(spain,france). neighbour(france,germany).


neighbour(france,spain).


neighbour(germany,france).
```

Deductive Databases

- Datalog: deductive database paradigm using normal logic programs (under stratified negation) with predicates and variables.

☞ Remember: stratified implies a unique stable model.

- Datalog is more expressive than SQL, but less expressive than logic programs without the stratification limitation.

- It allows, for instance, defining recursive relations, such as:

```
connected(X,Y) :- neighbour(X,Y).
connected(X,Z) :- neighbour(X,Y), connected(Y,Z).
```

  so that we would get `connected(spain,germany)` even though they are not neighbours.

- Bodies can add conditions on variables `X!=Z`, `X>Z*(Y+1)`, etc.

```
connected(X,Z) :- neighbour(X,Y), connected(Y,Z), X!=Z.
```

Deductive Databases

- Domain independence: answers shouldn't change if we just augment the Herbrand Domain

```
switch(1..3).
p(X,Y) :- X<Y.   % ordered pairs of different switches
```

returns `p(1,2), p(1,3), p(2,3)` if $D = \{1, 2, 3\}$
but for $D = \{1, 2, 3, 4\}$ we miss `p(1,4), p(2,4), p(3,4)`.
The set of possible pairs of integers is infinite!

```
p(X) :- not switch(X).   % anything that is not a switch
```

The potential D with non-switches is even worse!

- All variable occurrences in a rule must be safe

Definition (Safety: guarantees domain independence)

A variable is safe if it occurs in a non-negated predicate in the body.

```
p(X,Y) :- X<Y, switch(X), switch(Y).
q(X) :- object(X), not switch(X).  % define valid objects!
```

Answer Set Programming

- Answer Set Programming (ASP) = we allow normal logic programs (unstratified negation) with predicates and variables.
- In ASP, the stable models are called answer sets.
- Example:

```
pacifist(X)   :- quaker(X), not bellicous(X).
bellicous(X)  :- republican(X), not pacifist(X).
quaker(nixon). republican(nixon).
republican(reagan).
```

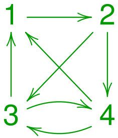
Two answer sets:

```
Answer: 1
... bellicous(reagan) bellicous(nixon)
Answer: 2
... bellicous(reagan) pacifist(nixon)
```

An example: Hamiltonian circuits

Definition (*HAMILT*)

The Hamiltonian Cycle problem, *HAMILT*, consists in deciding whether a graph contains a cyclic path in a graph that visits each vertex exactly once. *HAMILT* is an **NP**-complete problem.



- extensional database `mygraph.gph` with the graph
- Examples of medium sized graphs (200 nodes, 1250 edges): `http://www.cs.uky.edu/ai/benchmark-suite/hamiltonian-cycle.html`

An example: Hamiltonian circuits

- Predicate `in(X,Y)` points out that an edge `X` $\rightarrow$ `Y` is in the cycle. We generate arbitrary choices

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

- Only one outgoing vertex, only one incoming vertex:

```
:- in(X,Y), in(X,Z), Y!=Z.
:- in(X,Z), in(Y,Z), X!=Y.
```

- Disregard disconnected cycles. We use `reached(X)` meaning that `X` can be reached from an arbitrary fixed vertex, say 1.

```
reached(X) :- in(1,X).
reached(Y) :- reached(X), in(X,Y).
```

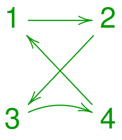
and we forbid unreached vertices:

```
:- vtx(X), not reached(X).
```

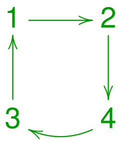
An example: Hamiltonian circuits

- Making the call:
  `clingo 0 hamilt.lp`
  We obtain two answers:

```
Answer: 1
in(4,3) in(3,1) in(2,4) in(1,2)
Answer: 2
in(4,1) in(3,4) in(2,3) in(1,2)
SATISFIABLE
```



Answer 1



Answer 2

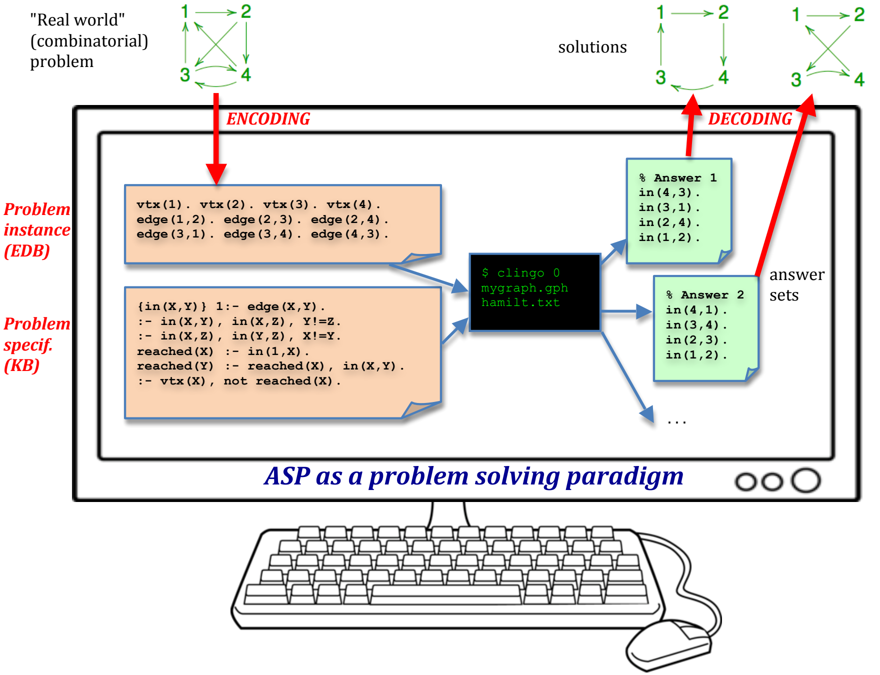
An example: Hamiltonian circuits

- We can split `clingo` in two steps:
  grounder `gringo` + propositional solver `clasp`.
- Download `gringo` from `potassco.org` and make the call

```
$ gringo hamilt.txt | clasp 0
```

- To display the ground program, try the following

```
$ gringo -t hamilt.txt
...
:-in(1,2),in(1,3).
:-in(1,3),in(1,2).
:-in(2,1),in(2,3).
...
reached(2):-in(1,2).
reached(3):-in(2,3),reached(2).
reached(3):-in(1,3),reached(1).
...
```


"Real world"
(combinatorial)
problem
1
2
3
4
solutions
ENCODING
DECODING
Problem
instance
(EDB)
vtx(1). vtx(2). vtx(3). vtx(4).
edge(1,2). edge(2,3). edge(2,4).
edge(3,1). edge(3,4). edge(4,3).
Problem
specif.
(KB)
{in(X,Y)} 1:- edge(X,Y).
:- in(X,Y), in(X,Z), Y!=Z.
:- in(X,Z), in(Y,Z), X!=Y.
reached(X) :- in(1,X).
reached(Y) :- reached(X), in(X,Y).
:- vtx(X), not reached(X).
$ clingo 0
mygraph.gph
hamilt.txt
% Answer 1
in(4,3).
in(3,1).
in(2,4).
in(1,2).
% Answer 2
in(4,1).
in(3,4).
in(2,3).
in(1,2).
answer
sets
...
ASP as a problem solving paradigm

Generate-Define-Test (GDT)

Many problem specifications follow the:
Generate, Define and Test (GDT) methodology

G: Generate candidate solutions with choice rules

```
{in(X,T)} :- edge(X,Y).
```

D: Define: auxiliary predicates when needed for G or T

```
reached(X) :- in(1,X).
reached(Y) :- reached(X), in(X,Y).
```

T: Test: constraints remove unwanted combinations

```
:- in(X,Y), in(X,Z), Y!=Z.
:- in(X,Z), in(Y,Z), X!=Y.
:- vtx(X), not reached(X).
```

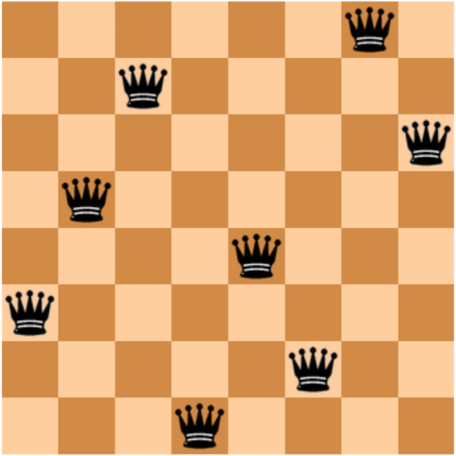
ASP vs Prolog

	ASP	Prolog
semantics	several $n \geq 0$ answer sets	unique (canonical) model

problem solving	1 answer set = 1 solution	1 var. instantiation = 1 solution
		`?- graph(G), hamilt(G,X).`
		`X=[(4,3),(3,1),(2,4),(1,2)];`
		`X=[(4,1),(3,4),(2,3),(1,2)]`

computational power	**NP**-complete	**Turing**-complete
language type	specification (~~execution~~)	programming (flow control: ordering, cut,...)

8 Queens revisited



Example (8-queens problem)

- Arrange 8 queens in a 8×8 chessboard so they do not attack one each other.

Explicit negation

- We can sometimes be interested in a second negation, strong or explicit negation (originally called "classical"). Example:

```
fill :- empty, not fire.
```

risky! we fill when no information on fire, but no guarantee.

- We could use auxiliary atom `no_fire` ("I'm sure there is no fire")

```
fill :- empty, no_fire.
:- fire, no_fire.
no_fire :- wet.
```

- Explicit negation '–' makes this same effect.

```
fill :- empty, -fire.
-fire :- wet.
```

and the constraint `:- fire, -fire` is implicit.

Einstein's 5 houses riddle: who keeps fishes as pets?

1. The Brit lives in the red house.
2. The Swede keeps dogs as pets.
3. The Dane drinks tea.
4. The green house is on the immediate left of the white house.
5. The green house's owner drinks coffee.
6. The owner who smokes Pall Mall rears birds.
7. The owner of the yellow house smokes Dunhill.
8. The owner living in the center house drinks milk.
9. The Norwegian lives in the first house.
10. The Blends smoker is neighbor of the one who keeps cats.
11. The horse keeper is neighbor of the one who smokes Dunhill.
12. The owner who smokes Bluemasters drinks beer.
13. The German smokes Prince.
14. The Norwegian lives next to the blue house.
15. The Blends smoker lives next to the one who drinks water.

Pooling and constants

- Pooling: abbreviate several facts in a same atom

```
house(1..5).
color(red;green;blue;white;yellow).
```

is the same than

```
house(1). house(2). house(3). house(4).house(5).
color(red). color(green). color(blue).
color(white). color(yellow).
```

- Constants: can be defined in the file

```
#const numhouses=5.
house(1..numhouses).
```

or passed as arguments in command line

```
$ clingo -c numhouses=5 einstein.txt
```

Compound terms

- Domain elements can be compound terms using tuples

```
birthdate( (10,july,1980) ).
independence( (4,july,1776) ).
important(D) :- birthdate(D).
important(D) :- independence(D).
birthday( (D,M) ) :- birthdate( (D,M,_) ).
julyevent(D) :- important( (D,july,_) ).
```

- We can also use function symbols as “tuple names”:

```
birthdate( date(10,july,1980) ).
independence( date(4,july,1776) ).
important(D) :- birthdate(D).
important(D) :- independence(D).
birthday( day(D,M) ) :- birthdate( date(D,M,_) ).
julyevent(D) :- important( date(D,july,_) ).
```

⚠ warning: `day` and `date` above are not predicates!

```
date(2,july,2010).  % No connection with "day" as function!
```

Care with infinite grounding

- Use function symbols carefully!

```
person(mary).
person(father(X)) :- person(X).
```

The grounding for this program never stops!
In this case, better use a predicate and name each person

```
person(mary).
father(mary,peter).
person(Y) :- person(X), father(X,Y).
```

- A similar care must be taken with arithmetics

```
house(1..5).
greater(X+1,X) :- house(X).
greater(X+1,Y) :- greater(X,Y).
```

causes an infinite grounding, but can add a limit

```
house(1..5).
greater(X+1,X) :- house(X), house(X+1).
greater(X+1,Y) :- greater(X,Y), house(X+1).
```

Reification

- Sometimes, different predicates follow a same pattern

```
person(brit;swede;dane;norw;german).
1 { guest(H,X) : person(X) } 1 :- house(H).
:- guest(H,X), guest(H',X), H!=H'.

color(red;green;white;blue;yellow).
1 { paint(H,X) : color(X) } 1 :- house(H).
:- paint(H,X), paint(H',X), H!=H'.

pet(fish;horse;dog;bird;cat).
1 { grows(H,X) : pet(X) } 1 :- house(H).
:- grows(H,X), grows(H',X), H!=H'.
```

`person/guest`, `color/paint`, `pet/grows` have the same roles

Reification

- Reification = *res*-(thing)-*fication*-(make)
- Convert predicate name into new object (thing) as argument:
  "types" = `type(person)`, `type(color)`, `type(pet)`
  "values" = `person(dane)` $\rightarrow$ `value(person,dane)`,
  `color(red)` $\rightarrow$ `value(color,red)`
  "assignments" = `guest(H,X)` $\rightarrow$ `at(H,person,X)`,
  `paint(H,X)` $\rightarrow$ `at(H,color,X)`

```
value(person,(brit;swede;dane;norw;german)).
value(color(red;green;white;blue;yellow)).
value(pet,(fish;horse;dog;bird;cat)).
type(T) :- value(T,_).
1 { at(H,T,X) : value(T,X) } 1 :- house(H),type(T).
:- at(H,T,X), at(H',T,X), H!=H'.
```

New features

- Aggregate = function on sets of values.
- We may have `#sum`, `#max`, `#min`, `#avg`, `#count`. Example:

```
income(jan,5).   income(feb,3).
income(mar,-2).  income(apr,10).
total(S) :- #sum{X: income(M,X)} = S.
```

`{X: income(M,X)}` = `{5,3,-2,10}` the sum is `S=16`

warning: sets have no repetitions (repeated values count once)

```
income(may,10). income(jun,10).
```

the set is still `{5,3,-2,10}` and the sum is `S=16`

- We use tuples (the sum applies to the first component):

```
total(S) :- #sum{X,M: income(M,X)} = S.
```

`{X,M: income(M,X)} =`

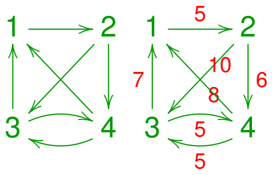
`{(5,jan), (3,feb), (-2,mar), (10,apr), (10,may), (10,jun)}`

Optimization

- ASP problem solving: 1 answer set = 1 solution
- Sometimes we are interested in preferred or optimal solutions
- Preferred/optimal answer sets
  we are going to select only some answer set(s)
- Depending on how we conceive the problem, two methods:
  - `#minimize/maximize`: conceived for optimization
  - Weak constraints: conceived for preferences

Optimization

Example of optimization: Travelling Salesman Problem = find Hamiltonian cycle with shorter distance

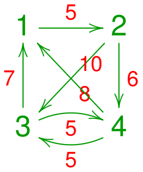


Reuse `hamilt.lp`† and adapt the problem instance as follows:

```
vtx(1..4).
edge(1,2,5). edge(2,3,10). edge(2,4,6).
edge(3,1,7). edge(3,4,5). edge(4,3,5). edge(4,1,8).
edge(X,Y) :- edge(X,Y,_).
```

Optimization

Example of optimization: Travelling Salesman Problem = find Hamiltonian cycle with shorter distance



In `hamilt.lp` we can get the total distance of the path adding:

```
distance(S) :- #sum{C,X,Y:in(X,Y),edge(X,Y,C)}=S.
#show distance/1.
```

Running `clingo 0 hamilt.lp graph1.lp` we get 2 solutions

```
Answer:  1
in(1,2) in(2,4) in(3,1) in(4,3) distance(23)
```
minimal

```
Answer:  2
in(1,2) in(2,3) in(3,4) in(4,1) distance(28)
```

Optimization

- Getting minimal solution by hand is unfeasible:
  Easy optimization problems may have millions of (non-optimal) solutions. To guarantee optimality, we should generate all!

- `#minimize` declaration = works like a `#sum{ ...}` aggregate, but will choose answer sets with a minimum sum

  ```
  #minimize{C,X,Y:in(X,Y),edge(X,Y,C)}.
  ```

  We can also use `#maximize` instead.

- The call `clingo hamilt.lp graph1.lp` will start a loop:
  (1) find a solution S_0; (2) find S_{i+1} better than S_i until no one found

- By default, only one optimum is shown. To show all optima, use

  ```
  clingo --opt-mode=optN -n0 hamilt.lp graph1.lp
  ```

  Example: try changing fact `edge(2,3,10)` by `edge(2,3,5)`

Preferences as weak constraints

- Weak constraints = alternative way of selecting answer sets. Equivalent to `#minimize`.
- Constraints that we prefer to satisfy

Example (Dinner tables)

- Sit 5 people in 2 tables (with capacities 2 and 3).
- Avoid sitting a person with anybody she hates
- Prefer sitting a person with anybody she likes

Preferences as weak constraints

```
table(t1,2). table(t2,3).
person(a;b;c;d;e).
hates(a,c). hates(d,e). likes(a,d). likes(c,e).
1 {sit(X,T): table(T,_)} 1:- person(X).
:- table(T,N), #count{X:sit(X,T)}>N.
:- hates(X,Y), sit(X,T), sit(Y,T).
```

`clingo 0 dinner.lp` = we get 4 solutions

Table `t1`	Table `t2`
a d	*b c e*
a e	*b c d*
c d	*a b e*
c e	*a b d*

Strong constraint: they must like each other

`:- sit(X,T), sit(Y,T), not likes(X,Y).` unsatisfiable!

Preferences as weak constraints

```
table(t1,2). table(t2,3).
person(a;b;c;d;e).
hates(a,c). hates(d,e). likes(a,d). likes(c,e).
1 {sit(X,T): table(T,_)} 1:- person(X).
:- table(T,N), #count{X:sit(X,T)}>N.
:- hates(X,Y), sit(X,T), sit(Y,T).
```

`clingo 0 dinner.lp` = we get 4 solutions

Table `t1`	Table `t2`	Cost
a d	*b c e*	3+8=11 ☜ min
a e	*b c d*	4+9=13
c d	*a b e*	4+9=13
c e	*a b d*	3+8=11 ☜ min

Weak constraint: we prefer when they like each other
We pay a cost of 1 per each `X,Y` that dislikes (minimize the cost)

```
:~ sit(X,T), sit(Y,T), not likes(X,Y). [1,X,Y]
```

Preferences as weak constraints

```
table(t1,2). table(t2,3).
person(a;b;c;d;e).
hates(a,c). hates(d,e). likes(a,d). likes(c,e).
1 {sit(X,T): table(T,_)} 1:- person(X).
:- table(T,N), #count{X:sit(X,T)}>N.
:- hates(X,Y), sit(X,T), sit(Y,T).
```

`clingo 0 dinner.lp` = we get 4 solutions

Table `t1`	Table `t2`	Cost
a d	*b c e*	(-1)+(-1) = -2 min
a e	*b c d*	0+0=0
c d	*a b e*	0+0=0
c e	*a b d*	(-1)+(-1) = -2 min

Weak constraint: we prefer when they like each other

Or we pay a cost of -1 per each `X,Y` that likes (minimize the cost)

```
:~ sit(X,T), sit(Y,T), likes(X,Y). [-1,X,Y]
```

Preferences as weak constraints

- We can always use `#minimize` or `#maximize` instead. Example:

```
#maximize{1,X,Y: sit(X,T), sit(Y,T), likes(X,Y)}.
```

- Preference levels `@`*p* specifies a priority (higher = more important). Example: add a second level to dinner problem
  - Maximize the likes always
  - Likes being equal, I prefer sitting *c* in `t2`

```
#maximize{1@2,X,Y: sit(X,T), sit(Y,T), likes(X,Y)}.
:~ sit(c,T), T!=t2. [1@1]
```

ASP solvers

- ASP competition: 7 editions
  Last edition (2019): 4 tracks depending on language features
- Most solvers were based on the ASP solver clasp/clingo by the Potassco group (University of Potsdam, Germany)
  on which professional applications were built

☞ Potassco branch in A Coruña!

- DLV, WASP (Univ. della Calabria, Italy):
  the other main solver with many professional applications.
- Both clingo and DLV are two-phase (ground & solve) native ASP solvers

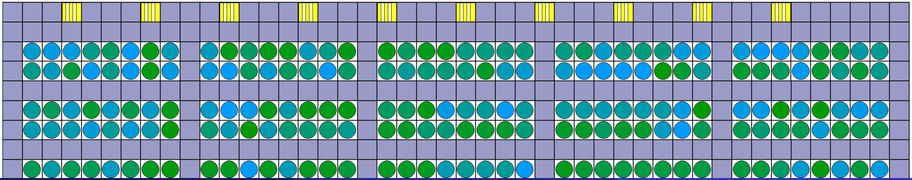
ASP solvers

Solvers using other strategies:

- Lazy grounding:
  `ASPeRIX` (Univ. of Angers, France);
  `Alpha` (TUWien, Austria)
- Top-down evaluation (a la Prolog):
  `s(ASP)` (Univ. of Texas at Dallas, USA)
- Translation to SAT:
  `ASSAT` (Univ. of Science and Tech., Hong Kong, China);
  `Cmodels` (Univ. of Texas at Austin, USA);
  Univ. of Tampere, Finland [Rankooh, Janhunen 2022]

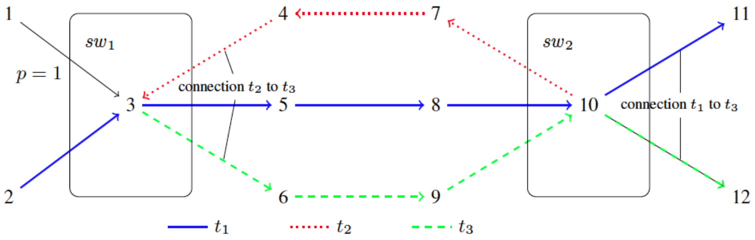
Outstanding ASP applications (Potassco)

Multi-robot path finding in automated warehouses



Outstanding ASP applications (Potassco)

SBB (Swiss Federal Railways). Solving train scheduling problems



Uses clingo[dl] = clingo + difference logic (integer constraints)

ASP applications: other examples

- Workforce and resource management. Many examples: Swiss Railway SBB, Cargo Ship Port, Hospitals (nurse shifts, room assignment, . . .)
- Telecom Italy: Intelligent phone call routing (DLV)
- Phylogenetic networks, Haplotype inference
- Repairing Large Scale Biological Networks
- Explaining and reasoning on natural language, Facebook bAbI challenge (Univ. of Nebraska at Omaha)
- Music composition
- Diagnosis for the Space Shuttle (NASA + Univ. of Lubbock, TX)
- Data integration: INFOMIX (DLV)
- Videogame scenario generation
- Robotics (combination with Robot Operating System, ROS)
- Product Configuration . . .

ASP applications: other examples

- See more at
  E. Erdem, M. Gelfond and N. Leone:
  Applications of Answer Set Programming
  AI Magazine 37(3): 53-68 (2016)

- And who knows what else soon . . .

We want you!