Knowledge Representation
Chapter 3. Relational Representation and Reasoning

Pedro Cabalar

Dept. Computer Science
University of Corunna, SPAIN

February 17, 2017

Relational Representation

- As atoms, we use now predicates like `neighbour(france,spain)` or `exports(germany,france,cars)` to allow representing relations among individuals.
- Herbrand Domain = set of individuals, each one uniquely identified by a constant name. E.g. $D = \{\texttt{germany}, \texttt{france}, \texttt{spain}, \texttt{cars}, \dots\}$.
- There are no types among individuals. We use predicates:

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

Relational Representation

- A set of facts becomes the extensional database!

```
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

Deductive Databases

- Deductive database: some predicates can be deduced from rules (intensional) instead of listing their facts.
- Example: `neighbour` should be symmetric.

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

- Capital letters `X`, `Y`, etc, denote (universally quantified) variables = arbitrary individuals.
- Ground atom = it has no variables (predicate and constants). Grounding = replacing variables by all their possible instances. In the example:

Deductive Databases

Example: grounding of program

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

yields:

```
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).
```

and so, the stable model also contains:

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

Deductive Databases

- Datalog: deductive database paradigm, allowing positive rules with predicates and variables.
- Datalog is more expressive than SQL, but less expressive than logic programs with negation.
- 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.

Answer Set Programming

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

```
bird(tweety).
bird(woody).
penguin(tweety).
flies(X) :- bird(X), not ab(X).
ab(X) :- penguin(X).
```

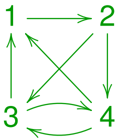
Answer set:

```
bird(tweety) bird(woody) penguin(tweety)
ab(tweety) flies(woody)
```

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

```
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). edge(4,1).
```

- 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 with an auxiliary predicate `out`.

```
in(X,Y) :- edge(X,Y), not out(X,Y).
out(X,Y) :- edge(X,Y), not in(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).
```

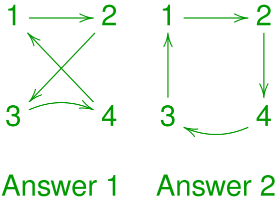
  and we forbid unreached vertices:

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

An example: Hamiltonian circuits

- Making the call:
  `clingo 0 hamilt.txt`
  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
```



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).
  ...
  ```

- Variable occurrences in a rule must be safe

Definition (Safety)

A variable is safe if it occurs in the positive body of the rule

- Example: in the rule

```
unreached(X) :- not reached(X).
```

variable `X` is unsafe. However, in rule

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

`X` becomes safe.

Cardinality constraints

- The rules:

```
in(X,Y) :- edge(X,Y), not out(X,Y).
out(X,Y) :- edge(X,Y), not in(X,Y).
```

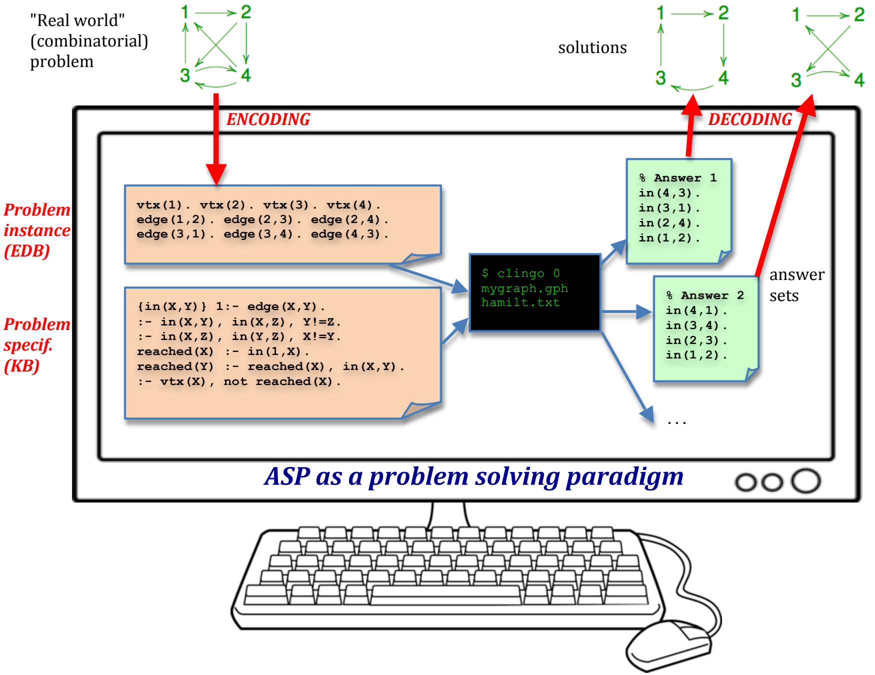
  can be replaced by

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

- Furthermore, we can use conditional literals to include several literals in the set. Example:

```
2 { friend(X,Y) : person(Y)} :- person(X).
```

  Generates at least two friends for each person `X`.



"Real world" (combinatorial) problem
1 2 3 4
solutions
1 2 3 4
1 2 3 4
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

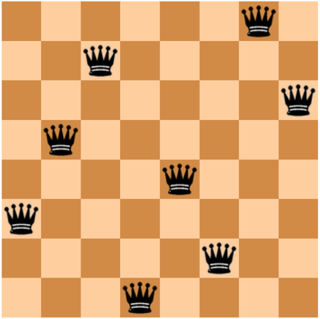
ASP vs Prolog

	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.
- Exercise: encode the problem in ASP. (Use cardinality atoms).

Strong negation

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

  ```
  cross :- not train.
  ```

  risky! we cross the railway tracks when no information on train approaching is available.

- We could use an auxiliary atom `no_train`

  ```
  cross :- no_train.
  :- train, no_train.
  ```

- Strong negation ‘–’ makes this same effect.

  ```
  cross :- -train.
  ```

  and the constraint `:- train, -train` 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.

New features

- 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
```

New features

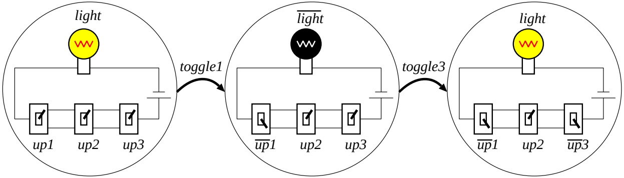
- Function symbols as constructors.

```
owner( person(bill,gates), microsoft ).
owner( person(jeff,bezos), amazon ).
owner( company(inditex), zara).
family(Y) :- owner( person(X,Y), Z).
```

Reasoning about actions with ASP

- We begin with some "type declarations".

```
time(0..pathlength).
step(0..pathlength-1).
switch(1..3).
```



Reasoning about actions with ASP

```
% Effect axioms
  up(X,I+1) :- -up(X,I),  toggle(X,I), step(I).
 -up(X,I+1) :-  up(X,I),  toggle(X,I), step(I).
 light(I+1) :- -light(I), toggle(X,I), step(I).
-light(I+1) :-  light(I), toggle(X,I), step(I).

% Executability constraints: none

% Inertia
  up(X,I+1) :-  up(X,I), not -up(X,I+1), step(I).
 -up(X,I+1) :- -up(X,I), not  up(X,I+1), step(I).
 light(I+1) :-  light(I),not -light(I+1), step(I).
-light(I+1) :- -light(I),not  light(I+1), step(I).

% Unique action
:- toggle(X,I), toggle(Y,I), X!=Y, step(I).
```

Prediction example

```
% switches-predict.txt
% Initial state
light(true,0).
up(X,true,0) :- switch(X).

% Performed actions
toggle(1,0).
```

Calling gringo/clasp with

```
$ clingo -c pathlength=1
     switches.txt switches-predict.txt

Answer: 1
... -up(1,1) up(2,1) up(3,1) -light(1)
```

Reasoning about actions with ASP

Postdiction example:

```
% switches-postdict.txt

% Actions generation
1 { toggle(Z,I) : switch(Z) } 1 :- step(I).
  % generate 1 toggle among all switches Z

% Completing facts about the initial situation
1 {up(X,0); -up(X,0)} 1 :- switch(X).
1 {light(0); -light(0)} 1.

% Observations
up(3,0). light(0).
-light(1). -up(1,1). up(3,1). toggle(3,1).
```

Reasoning about actions with ASP

Calling clingo

```
$ clingo -c pathlength=1 0
   switches.txt switches-postdict.txt
```

we get 6 possible explanations. One of them:

```
Answer: 1
Stable Model:
... toggle(1,0) -up(2,0) up(1,0) -up(2,1) ...
```

Planning example

```
% switches-plan.txt
% Planning problem
#show toggle/2.
% Actions generation

1 { toggle(Z,I) : switch(Z) } 1 :- step(I).

% Initial state
light(0).
up(X,0) :- switch(X).

% Goal state
goal :- light(pathlength), -up(1,pathlength),
        up(2,pathlength), -up(3,pathlength).
:- not goal.
```

Reasoning about actions with ASP

Calling clingo

```
$ clingo -c pathlength=1 0
  switches.txt switches-plan.txt
```

We don't get models. After increasing `pathlength`

```
$ clingo -c pathlength=2 0
  switches.txt switches-plan.txt
```

we get 2 possible plans

```
Answer: 1
toggle(1,0) toggle(3,1)
Answer: 2
toggle(3,0) toggle(1,1)
```

Exercise: missionaries and cannibals

- A classical example: missionaries and cannibals

  > *3 missionaries and 3 cannibals come to a river and find a boat that holds two. If the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten. How shall they cross?*

Exercise: missionaries and cannibals

We will use the following fluents:

1. `num(G,B,N,T)` = there are `N` persons of group `G` at bank `B` and time instant `T`.

   Ex.: `num(mis,left,3,0)` = "*initially, there are 3 missionaries in the left bank*"

2. `boat(B,J)` points out the boat bank. Ex. `boat(left,0)` = "*initially, the boat is at left bank*"

Exercise: missionaries and cannibals

We will use action:

- `move(M,C,T)` move `M` missionaries and `C` cannibals from situation `T-1` to situation `T`.
- For simplicity, we include two action attributes `moved(mis,N,T)` and `moved(can,N,T)` that point out separatedly how many persons of each group are moved.

Exercise: missionaries and cannibals

We will use the incremental mode for planning that varies the incremental constant `t` and allows three program parts:

```
#include <incmode>.     % incremental mode
...
#program base.
...                     % Rules for time t=0
#program step(t).
...                     % Rules for times t>0
#program check(t).
...                     % Rules for times t>=0
```

Exercise: missionaries and cannibals

We begin with the program base (`t=0`) and some type declarations

```
#include <incmode>.
#program base.              % Rules for t=0

% Type declarations
group(mis). group(can).
bank(left). bank(right).
number(0..3).

opposite(left,right). opposite(right,left).

% Initial state
num(mis,left,3,0). num(can,left,3,0). boat(left,0).
num(mis,right,0,0). num(can,right,0,0).
```

Exercise: missionaries and cannibals

Rules for transitions (`t>0`)

```
#program step(t).

% Action generation
1 {move(X,Y,t) : number(X),number(Y) } 1.

% Auxiliary (action attributes)
moved(mis,M,t) :- move(M,C,t).
moved(can,C,t) :- move(M,C,t).

% Executability axioms
:- move(M,C,t), M=0, C=0.
:- move(M,C,t), M+C>2.
:- moved(G,N,t), boat(B,t-1), num(G,B,M,t-1), N>M.
```

Exercise: missionaries and cannibals

Rules for any situation $\texttt{t} \geq 0$

```
#program check(t).
% Constraints: unique value
:- num(G,B,N,t), num(G,B,M,t), M!=N.
:- boat(left,t), boat(right,t).

% Missionaries not outnumbered by canibals
:- num(mis,B,M,t), num(can,B,C,t), C>M, M>0.

:- query(t), not goal(t).
goal(t) :- num(mis,right,3,t), num(can,right,3,t).

#show move/3.     % We only show performed actions
```

Exercise: missionaries and cannibals

We execute `clingo mc.pl` and it will increase $t = 1, 2, \ldots$ until a solution is found. The first solution is obtained with $t = 12$.

$\vdots$

```
Solving...
Solving...
Solving...
Solving...
Answer: 1
move(1,1,1) move(1,0,2) move(0,2,3) move(0,1,4)
move(2,0,5) move(1,1,6) move(2,0,7) move(0,1,8)
move(0,2,9) move(1,0,10) move(1,1,11)
```

Option `-n 0` provides all solutions (there are 4 plans with $t = 12$).

Inertia

- For a boolean fluent `F`, inertia is just the pair of rules

```
 h(F,t) :-  h(F,t-1), not -h(F,t), fluent(F).
-h(F,t) :- -h(F,t-1), not  h(F,t), fluent(F).
```

  where `h(F,t)` means that fluent `F` holds at time `t`

- When the fluent `F` has a range of more than 2 values, this is generalised as:

```
 h(F,V,t) :- h(F,V,t-1), not -h(F,V,t), fluent(F).
-h(F,V,t) :- h(F,W,t), V!=W, fluent(F), range(F,V).
```

- In the Mis&Can problem, inertia was not needed. If we had to add it, it would look like

```
 num(G,B,N,t) :- num(G,B,N,t-1), not -num(G,B,N,t).
-num(G,B,N,t) :- num(G,B,M,t), N!=M, number(N).
```

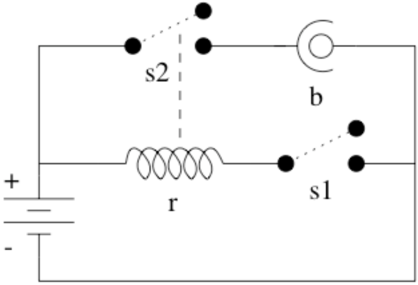
Diagnosis

- An agent acts in a dynamic environment and observes the results of her actions.
- Sometimes she gets discrepancies: observations $\neq$ expected result

Diagnosis

- Example [Balduccini & Gelfond 03]

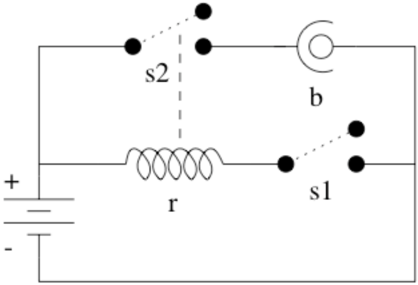
  We have a circuit with lightbulb b and a relay r. The agent can close s1 causing s2 to close (if r is not damaged). The bulb emits light if s2 is closed and b is not damaged.



Diagnosis example

- Example [Balduccini & Gelfond 03]

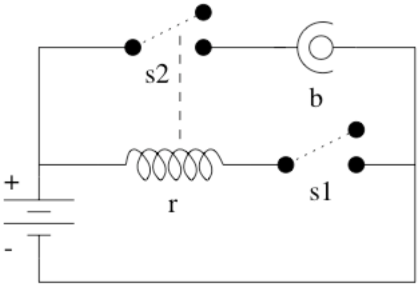
  Exogenous action break damages the relay. Action power-surge damages r, and b too, if the latter is not protected (prot).



Diagnosis example

- Example [Balduccini & Gelfond 03]

  We close s1 but b does not emit light: what has happened?



Diagnosis example

- All fluents are Boolean: we will use $h(F, J)$ and $-h(F, J)$ to represent that fluent F is true or false at time J, respectively.
- This allows a common pair of inertia rules

```
 h(F,I+1) :-  h(F,I), not -h(F,I+1),
              step(I), fluent(F).
-h(F,I+1) :- -h(F,I), not  h(F,I+1),
              step(I), fluent(F).
```

Diagnosis example

- Types and domains

```
time(0..pathlength).
step(0..pathlength-1).
switch(s1;s2).
component(r;b).
#show o/2.
```

Diagnosis example

- Fluents *ab*(*C*) point out that a component is damaged

```
fluent(r_active).
fluent(b_on).
fluent(b_prot).
fluent(closed(S)) :- switch(S).
fluent(ab(C)):-component(C).
```

Diagnosis example

- Actions are exogenous *exog* or agent's *agent*:

```
agent(close(s1)).
exog(break).
exog(surge).
action(Y):-exog(Y).
action(Y):-agent(Y).
```

Diagnosis example

- Predicate $o(A, I)$ means that A occurred at instant I.
- Effect axioms:

```
% Normal functioning
h(closed(s1),I+1) :- o(close(s1),I), step(I).

% Malfunctioning
h(ab(b),I+1) :- o(break,I), step(I).
h(ab(r),I+1) :- o(surge,I), step(I).
h(ab(b),I+1) :- o(surge,I), not h(b_prot,I),
                  step(I).
```

Diagnosis example

- Indirect effects:

```
 h(r_active,J) :- h(closed(s1),J),
                   -h(ab(r),J), time(J).
-h(r_active,J) :- -h(closed(s1),J), time(J).
-h(r_active,J) :- h(ab(r),J), time(J).
 h(closed(s2),J) :- h(r_active,J), time(J).
 h(b_on,J) :-  h(closed(s2),J),
                   -h(ab(b),J), time(J).
-h(b_on,J) :- -h(closed(s2),J), time(J).
-h(b_on,J) :- h(ab(b),J), time(J).
```

- Executability:

```
:- o(close(S),J), h(closed(S),J), time(J).
```

Diagnosis example

- We use predicates *obs* (observed) hand *hpd* (happened) to distinguish between observed facts and actions from real facts and performed actions.

```
% Something happening actually occurs
o(A,I) :- hpd(A,I), step(I).

% Check that observations hold
:- obs(F,J), not h(F,J), time(J).
:- -obs(F,J), not -h(F,J), time(J).

% Completing the initial state
1 {h(F,0); -h(F,0)} 1 :- fluent(F).
```

Diagnosis example

- These are the observations:

```
% A history
hpd(close(s1),0).
-obs(closed(s1),0).
-obs(closed(s2),0).
obs(b_prot,0).
-obs(ab(b),0).
-obs(ab(r),0).
% Something went wrong
-obs(b_on,1).

% Diagnostic module: generate exogenous actions
{ o(Z,I):exog(Z) } :- step(I).
```

Diagnosis example

- This will provide all possible explanations, but not minimal diagnoses.

```
$ clingo -c pathlength=1 -n 0 diag.txt
clingo version 4.5.0
Reading from diag.txt
Solving...
Answer: 1
o(close(s1),0) o(break,0)
Answer: 2
o(close(s1),0) o(surge,0)
Answer: 3
o(close(s1),0) o(surge,0) o(break,0)
```

Diagnosis example

- Optimization problems: we can use weak constraints

  ```
  :~ body   [ weight@priority ]
  ```

- Example

  ```
  :~ o(Z,I), exog(Z), step(I). [1]
  ```

  means “try to avoid occurrences of exogenous actions”

- This time we get the optimum diagnosis

  ```
  $ clingo -c pathlength=1 diag.txt
  clingo version 4.5.0
  Reading from diag.txt
  Solving...
  Answer: 1
  o(close(s1),0) o(break,0)
  Optimization: 1
  OPTIMUM FOUND
  ```