Answer Set Programming: A Primer

Thomas Eiter, Thomas Krennwallner (TU Wien, Austria)
Giovambattista Ianni (Università della Calabria, Italy)

Supported by Austrian Science Fund (FWF) project P20840 & P20841, the EC ICT Integrated Project Ontorule (FP7 231875), and the Italian National Project Interlink II04CG8AGG.

Unit 2 – ASP Paradigms and Solvers

Thomas Krennwallner

KBS Group, Institute of Information Systems, TU Vienna

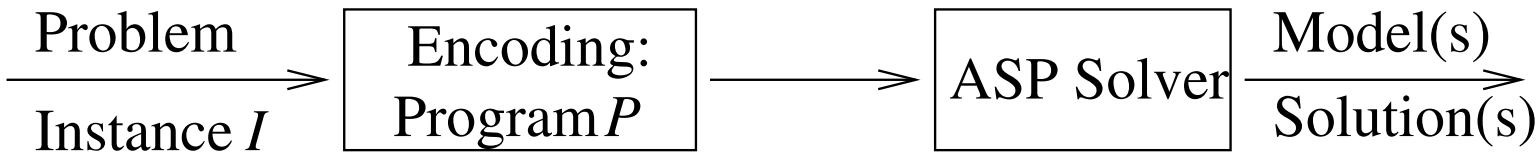
Reasoning Web Summer School 2009

Unit Outline

ASP Paradigm

General idea: stable models are solutions!

Reduce solving a problem instance I to computing stable models of a LP



1. *Encode* I as a (non-monotonic) logic program P, such that solutions of I are represented by models of P
2. *Compute* some model M of P, using an ASP solver
3. *Extract* a solution for I from M.

Variant: Compute multiple models (for multiple / all solutions)

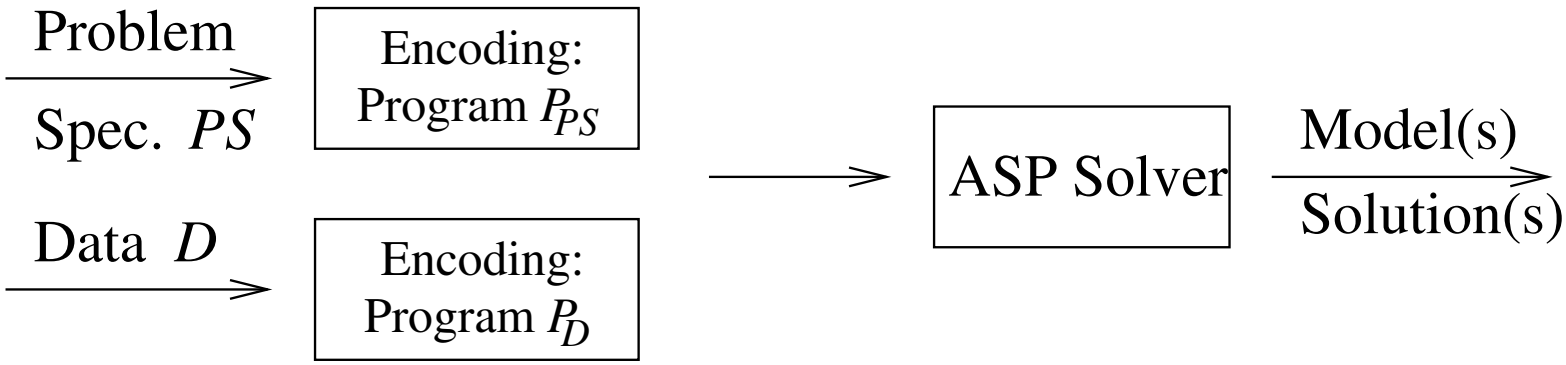
ASP Paradigm (ctd.)

Compared to SAT solving, ASP offers more features:

- transitive closure
- negation as failure
- predicates and variables

Generic problem solving by separating the

- **specification** of solutions ("logic" PS)
- **concrete instance** of the problem (data D)



Use of Double Negation

Defining a predicate p in terms of its negation $-p$

Example (Greatest Common Divisor — Euclid-style)

% base case

$gcd(X, X, X) \leftarrow int(X), X > 1.$

% subtract smaller from larger number

$gcd(D, X, Y) \leftarrow X < Y, gcd(D, X, Y_1), Y = Y_1 + X.$

$gcd(D, X, Y) \leftarrow X > Y, gcd(D, X_1, Y), X = X_1 + Y.$

This is not easy to come up with and needs more care in Prolog.

Use of Double Negation

Defining a predicate p in terms of its negation $-p$

Example (Greatest Common Divisor — ASP-style)

% Declare when D divides a number N.

$$divisor(D,N) \leftarrow int(D), int(N), int(M), N = D * M.$$

% Declare common divisors

$$cd(T,N_1,N_2) \leftarrow divisor(T,N_1), divisor(T,N_2).$$

% Single out non-maximal common divisors T

$$-gcd(T,N_1,N_2) \leftarrow cd(T,N_1,N_2), cd(T_1,N_1,N_2), T < T_1.$$

% Apply double negation: take non non-maximal divisor

$$gcd(T,N_1,N_2) \leftarrow cd(T,N_1,N_2), not\ -gcd(T,N_1,N_2).$$

☞ Run example (here, maximal integer $n = 4$)

The “Guess and Check” Methodology

Important element of ASP: Guess and Check methodology (or Generate-and-Test [Lifschitz, 2002]).

1. Guess: use unstratified negation or disjunctive heads to create candidate solutions to a problem (program part $\mathcal{G}$), and
2. Check: use further rules and/or constraints to test candidate solution if it is a proper solution for our problem (program part $\mathcal{C}$).

The "Guess and Check" Methodology

Important element of ASP: Guess and Check methodology (or Generate-and-Test [Lifschitz, 2002]).

1. Guess: use unstratified negation or disjunctive heads to create candidate solutions to a problem (program part $\mathcal{G}$), and
2. Check: use further rules and/or constraints to test candidate solution if it is a proper solution for our problem (program part $\mathcal{C}$).

From another perspective:

- $\mathcal{G}$: defines the search space
- $\mathcal{C}$: prunes illegal branches.

Further discussion in [Eiter *et al.*, 2000], [Leone *et al.*, 2006] (+ additional component for computing optimal solutions).

Example: 3-Coloring

Problem specification PS: 3-Colorability condition

Problem specification P_{PS}

$$\left.\begin{array}{l} g(X) \vee r(X) \vee b(X) \leftarrow node(X) \end{array}\right\} \textbf{Guess}$$

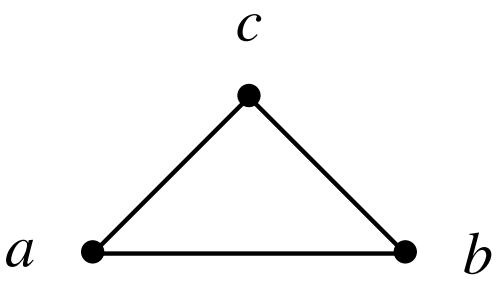
$$\left.\begin{array}{l} \leftarrow b(X), b(Y), edge(X,Y) \\ \leftarrow r(X), r(Y), edge(X,Y) \\ \leftarrow g(X), g(Y), edge(X,Y) \end{array}\right\} \textbf{Check}$$

Data P_D: Graph $G = (V, E)$

$$P_D = \{node(v) \mid v \in V\} \cup \{edge(v,w) \mid (v,w) \in E\}.$$

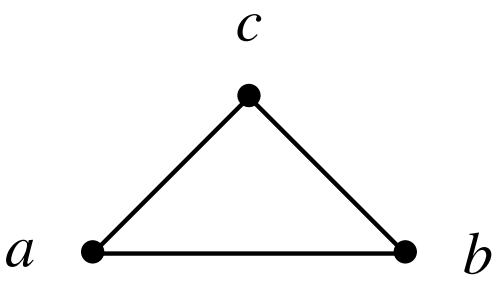
Correspondence 3-colorings $\rightleftharpoons$ models:
$v \in V$ is colored with $c \in \{r, g, b\}$ iff $c(v)$ is in the model of $P_{PS} \cup P_D$.

Example: 3-Coloring (ctd.)

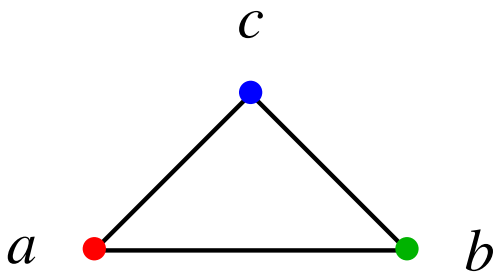


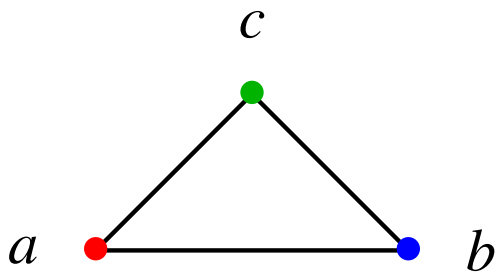
$$P_D = \{node(a), node(b), node(c), edge(a,b), edge(b,c), edge(a,c)\}$$

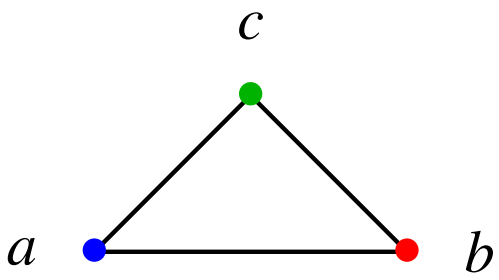
Example: 3-Coloring (ctd.)

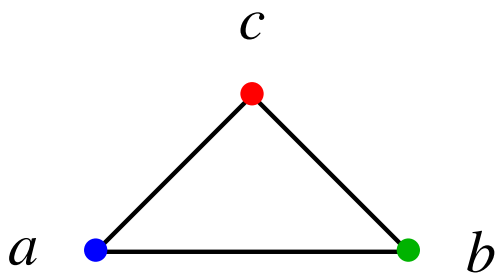


$$P_D = \{node(a), node(b), node(c), edge(a,b), edge(b,c), edge(a,c)\}$$

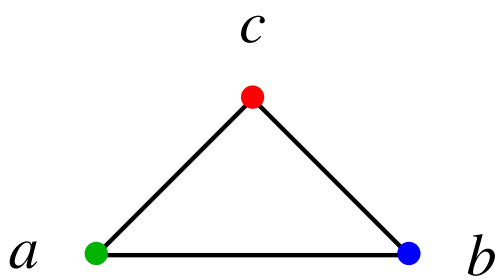


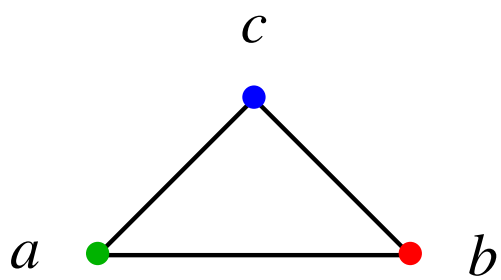






☞ Run example





Example: Hamiltonian Path/Cycle

Input: A directed graph represented by $node(\_)$ and $edge(\_,\_)$ and a starting node $start(\_)$.

Problem: Find a path/cycle beginning at the starting node which contains all nodes of the graph.

$$inPath(X,Y) \vee outPath(X,Y) \leftarrow edge(X,Y).$$ } **Guess**

$$\leftarrow inPath(X,Y),\ inPath(X,Y_1),\ Y \neq Y_1.$$
$$\leftarrow inPath(X,Y),\ inPath(X_1,Y),\ X \neq X_1.$$
$$\leftarrow node(X),\ not\ reached(X).$$
} **Check**

$$reached(X) \leftarrow start(X).$$
$$reached(X) \leftarrow reached(Y),\ inPath(Y,X).$$
} **Auxiliary Predicate**

Example: Hamiltonian Path/Cycle

Input: A directed graph represented by $node(\_)$ and $edge(\_,\_)$ and a starting node $start(\_)$.

Problem: Find a path/cycle beginning at the starting node which contains all nodes of the graph.

$$inPath(X,Y) \vee outPath(X,Y) \leftarrow edge(X,Y).$$ **Guess**

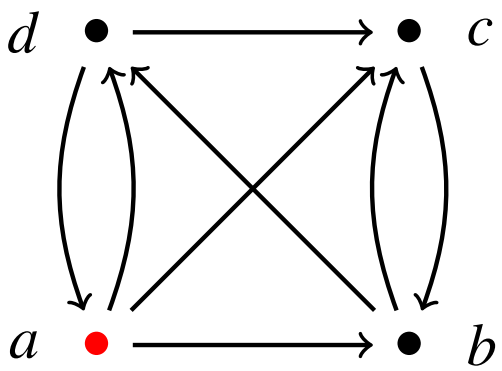
$$\leftarrow inPath(X,Y),\ inPath(X,Y_1),\ Y \neq Y_1.$$
$$\leftarrow inPath(X,Y),\ inPath(X_1,Y),\ X \neq X_1.$$
$$\leftarrow node(X),\ not\ reached(X).$$
$$\leftarrow not\ start\_reached.$$

Check

$$reached(X) \leftarrow start(X).$$
$$reached(X) \leftarrow reached(Y),\ inPath(Y,X).$$
$$start\_reached \leftarrow start(Y), inPath(X,Y).$$

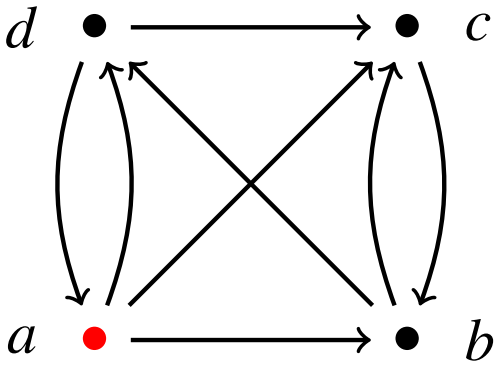
Auxiliary Predicate

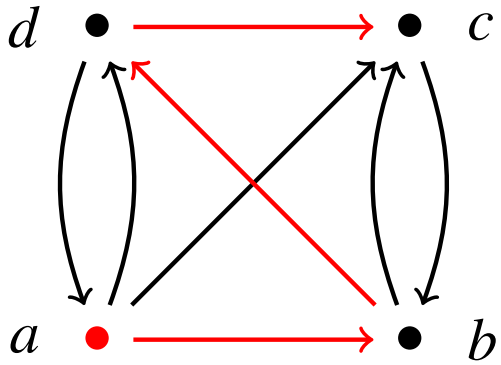
Example: Hamiltonian Path/Cycle (ctd.)

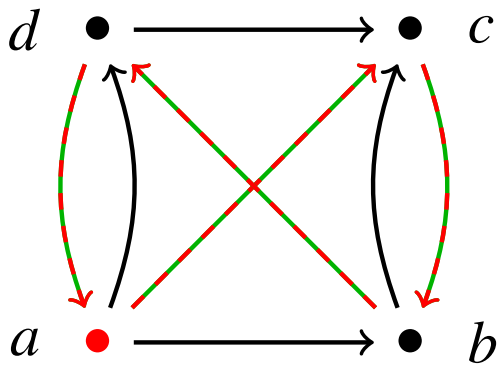


$$
\begin{aligned}
P_D = \{&node(a), node(b),\\
&node(c), node(d),\\
&edge(a,b), edge(a,c)\\
&edge(c,b), edge(b,c)\\
&edge(b,d), edge(d,c)\\
&edge(d,a), edge(a,d)\\
&start(a)\}
\end{aligned}
$$

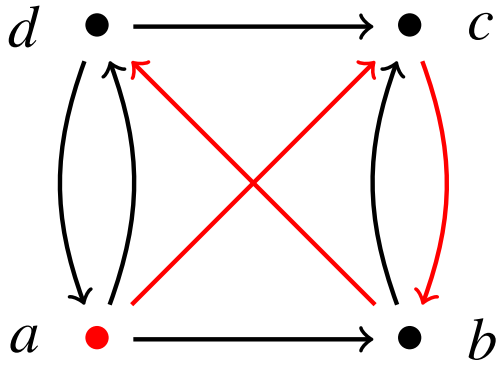
Example: Hamiltonian Path/Cycle (ctd.)

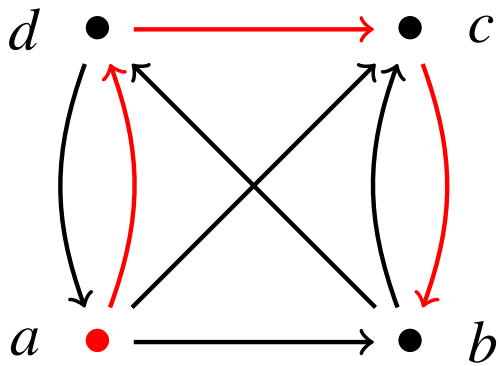






$$P_D = \{node(a), node(b),$$
$$node(c), node(d),$$
$$edge(a,b), edge(a,c)$$
$$edge(c,b), edge(b,c)$$
$$edge(b,d), edge(d,c)$$
$$edge(d,a), edge(a,d)$$
$$start(a)\}$$





☞ Run Hamiltonian Path

☞ Run Hamiltonian Cycle

Example: Course Assignment

Information about members and courses of a computer science dept. cs:

$member(sam, cs).$ $course(java, cs).$ $course(ai, cs).$
$member(bob, cs).$ $course(c, cs).$ $course(logic, cs).$
$member(tom, cs).$
$likes(sam, java).$ $likes(sam, c).$ $likes(tom, ai).$
$likes(bob, java).$ $likes(bob, ai).$ $likes(tom, logic).$

$$teach(X, Y) \leftarrow member(X, cs), course(Y, cs), likes(X, Y), not - teach(X, Y).$$
$$-teach(X, Y) \leftarrow member(X, cs), course(Y, cs), teach(X_1, Y), X_1 \neq X.$$
$$has\_course(X) \leftarrow member(X, cs), teach(X, Y).$$
$$\leftarrow member(X, cs), not\ has\_course(X).$$
$$\leftarrow teach(X, Y_1), teach(X, Y_2), teach(X, Y_3),$$
$$Y_1 \neq Y_2, Y_1 \neq Y_3, Y_2 \neq Y_3.$$

☞ Run example

Saturation Technique

Saturation technique: check whether all possible guesses satisfy a certain property Pr, like not being a solution to a problem (e.g., 3-uncolorability: co-NP-hard)

To test a property Pr we

- design a program P and an answer set candidate M_{sat} such that M_{sat} is the single answer set of P if the property Pr holds, and
- P has other answer sets (excluding M_{sat}) otherwise.

The construction is such that

- any answer set of P is a subset of M_{sat}, and
- whenever the property is found to hold, any candidate answer set is "saturated" to M_{sat}.

Example: 3-uncolorability

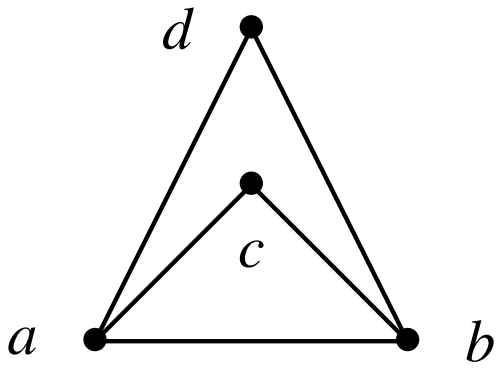
The constraints in the checking part of the 3-colorability program can be replaced by "saturation rules:"

$$b(X) \vee r(X) \vee g(X) \leftarrow node(X).$$ **Guess**

$$\left.\begin{array}{l} non\_col \leftarrow r(X), r(Y), edge(X,Y). \\ non\_col \leftarrow g(X), g(Y), edge(X,Y). \\ non\_col \leftarrow b(X), b(Y), edge(X,Y). \end{array}\right\}$$ **Check**

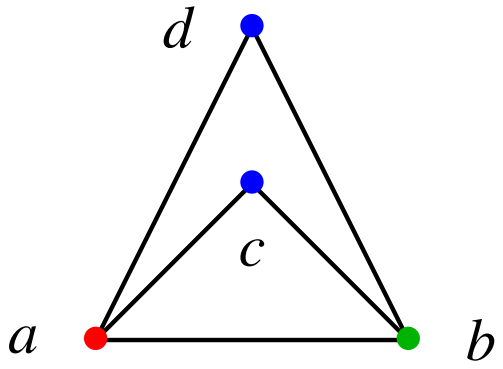
$$\left.\begin{array}{l} r(X) \leftarrow non\_col, node(X). \\ g(X) \leftarrow non\_col, node(X). \\ b(X) \leftarrow non\_col, node(X). \end{array}\right\}$$ **Saturize**

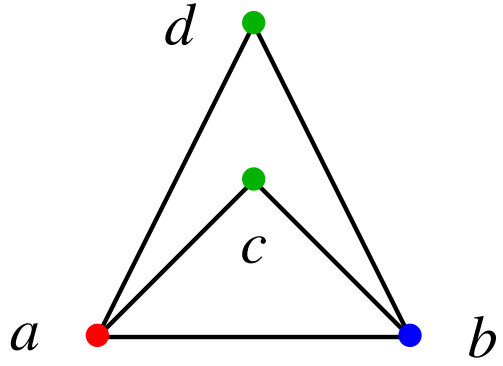
Example: 3-uncolorability (ctd.)

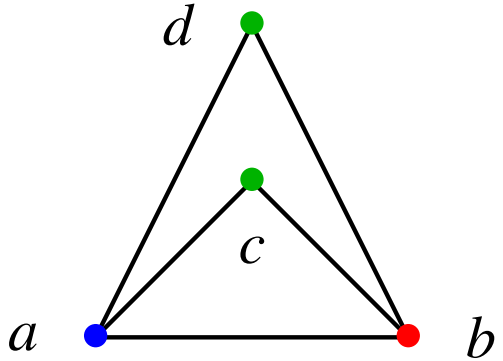


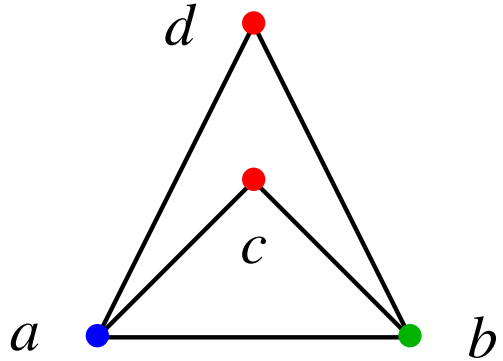
$$P_D = \{node(a), node(b), node(c), node(d), edge(a,b), edge(a,c), edge(b,c), edge(a,d), edge(b,d)\}$$

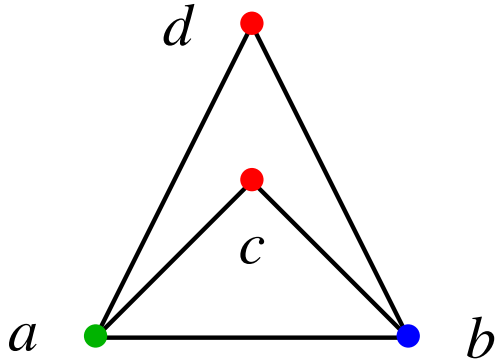
☞ Run example

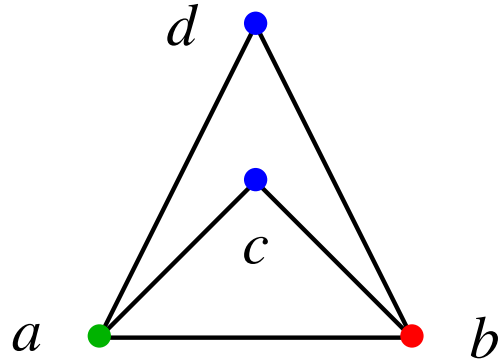




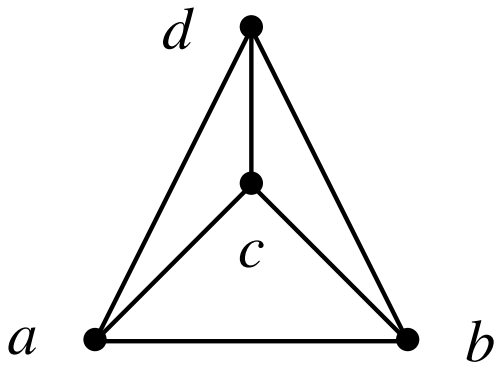




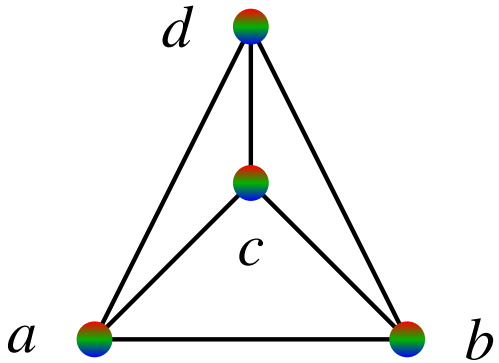




Example: 3-uncolorability (ctd.)



$$P_D = \{node(a), node(b), node(c), node(d), edge(a,b), edge(a,c), edge(b,c), edge(a,d), edge(b,d), edge(c,d)\}$$



☞ Run example

"Guess and Saturation Check" Paradigm

General design rule:
if we want to check that a property Pr holds for all guesses, we can

1. define the search space of guesses through a subprogram P_{guess}, using disjunctive rules, and
2. define a subprogram P_{check}, which checks Pr for a guess M_g.
3. If Pr holds for M_g, an appropriate set of saturation rules P_{sat} generates the special candidate answer set M_{sat}, otherwise
4. if Pr does not hold for M_g, an answer set results which is a strict subset of M_{sat} (thus preventing that M_{sat} is an answer set).

With additional guessing rules that are not involved in the saturation, we can express Σ_2^p-hard problems, like the strategic companies problem [Leone *et al.*, 2006], [Eiter *et al.*, 2000].

Iteration over a Set

Testing a property for all elements of a set without the use of negation.

This may be needed in some contexts:

- in combination with the saturation technique, or
- when the use of negation could lead to undesired behavior (e.g., in case of cyclic negation).

Example: Reachability for subgraphs

% Guess a subgraph for testing

$edge_1(X,Y) \vee edge_1(Y,X) \leftarrow edge(X,Y), edge(Y,X).$

$edge_1(X,Y) \leftarrow edge(X,Y), not\ edge(Y,X).$

% Compute all reachable nodes

$reached(X) \leftarrow start(X).$

$reached(X) \leftarrow reached(Y),\ edge_1(Y,X).$

% iterate to check if all nodes are reached

$all\_reached \leftarrow last(X), all\_reached\_upto(X).$

$all\_reached\_upto(X) \leftarrow all\_reached\_upto(Y), succ(Y,X), reached(X).$

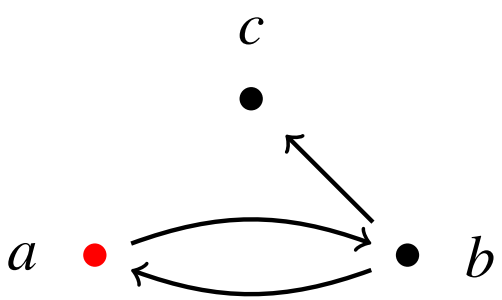
$all\_reached\_upto(X) \leftarrow first(X), reached(X).$

% Saturation rule

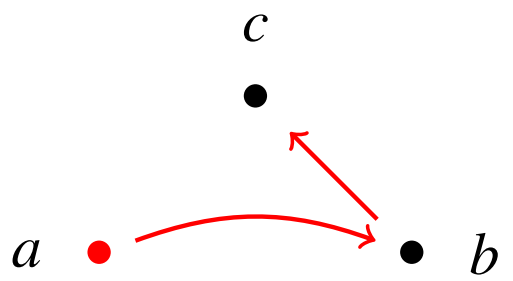
$edge_1(X,Y) \leftarrow all\_reached, edge(X,Y).$

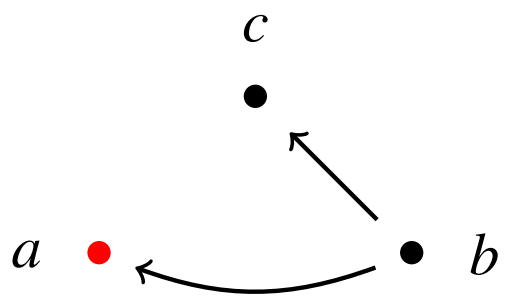
$succ$: (user-) defined predicate

Example: Reachability for subgraphs (ctd.)



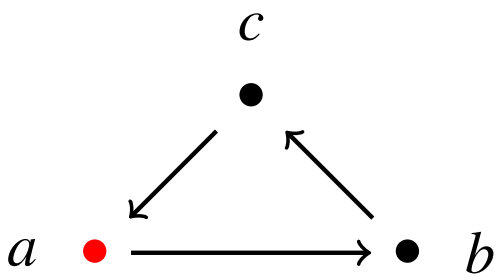
$$P_D = \{node(a), node(b), node(c), edge(a,b), edge(b,c), edge(b,a), start(a)\}$$



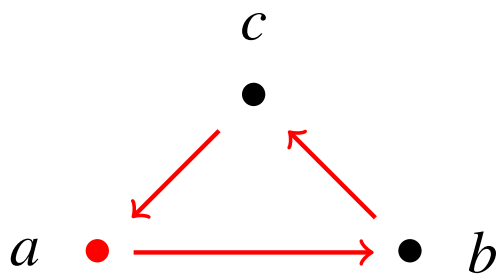


☞ Run example

Example: Reachability for subgraphs (ctd.)



$$P_D = \{node(a), node(b), node(c), edge(a,b), edge(b,c), edge(c,a), start(a)\}$$



☞ Run example

Answer Set Solvers

- NP-/Σ_2^p-completeness: Efficient answer set computation is not easy!
- Need to handle, for applications
  1. complex data (large data volumes)
  2. search
- Efforts to realize tractable fragments
- Many ASP solvers are available (function-free programs)

Approach

- Logic programming and deductive database techniques (for (1))
- SAT/Constraint Programming techniques for (2)

Different sophisticated algorithms have been developed
(like for SAT solving)

Answer Set Solvers

DLV[1]	http://www.dbai.tuwien.ac.at/proj/dlv/
Smodels[2]	http://www.tcs.hut.fi/Software/smodels/
GnT	http://www.tcs.hut.fi/Software/gnt/
Cmodels	http://www.cs.utexas.edu/users/tag/cmodels/
ASSAT	http://assat.cs.ust.hk/
NoMore(++)	http://www.cs.uni-potsdam.de/~linke/nomore/
Platypus	http://www.cs.uni-potsdam.de/platypus/
clasp	http://www.cs.uni-potsdam.de/clasp/
XASP	http://xsb.sourceforge.net/, distributed with XSB
aspps	http://www.cs.engr.uky.edu/ai/aspps/
ccalc	http://www.cs.utexas.edu/users/tag/cc/

- Several provide a number of extensions to the language described here.
- Answer Set Solver Implementation: see [Niemelä, 2004] tutorial
- ASP Solver competition
- ASPARAGUS Benchmark platform
  http://asparagus.cs.uni-potsdam.de/

[1] + many extensions, e.g., DLVEX, DLVHEX, DLV^{DB}, DLT, DLV-Complex

[2] + Smodels_cc

ASP Competition 2009

- ASP competition at the biannual LPNMR conference (started 2007)
- This year, the following systems ranked best:
  1. Potassco `http://potassco.sourceforge.net/`
  2. Claspfolio (Potassco + best options prediction)
  3. DLV `http://www.dbai.tuwien.ac.at/proj/dlv/`

`http://www.cs.kuleuven.be/~dtai/events/ASP-competition/`

SAT Competition 2009

- SAT competition at the annual SAT conference
  `http://www.satisfiability.org/`
- Clasp is an ASP solver from the Potassco suite and performed surprisingly well! `http://www.cs.uni-potsdam.de/clasp/`
- This year, the following systems ranked best in the *crafted instances* category (SAT+UNSAT instances):
  1. Clasp
  2. SATzilla2009_I
  3. SATzilla2009_C
- and for the *crafted instances* category (UNSAT instances):
  1. Clasp
  2. SATzilla2009_C
  3. IUT_BMB_SAT

`http://www.satcompetition.org/`

Architecture of ASP Solvers

Typically, a two level architecture

1. **Grounding Step**
   Given a program P with variables, generate a (subset) of its grounding which has the same models

Architecture of ASP Solvers

Typically, a two level architecture

1. **Grounding Step**
   Given a program P with variables, generate a (subset) of its grounding which has the same models

2. **Model Search**
   More complicated than in SAT/CSP Solving:
   - Candidate generation (classical model)
   - model checking (stability!)

Architecture of ASP Solvers

Typically, a two level architecture

1. **Grounding Step**
   Given a program P with variables, generate a (subset) of its grounding which has the same models

2. **Model Search**
   More complicated than in SAT/CSP Solving:
   - Candidate generation (classical model)
   - model checking (stability!)
     - for SAT, model checking is in ALOGTIME
     - for normal propositional programs, model checking is P-complete
     - for disjunctive propositional programs, model checking is co-NP-complete

Grounding Step

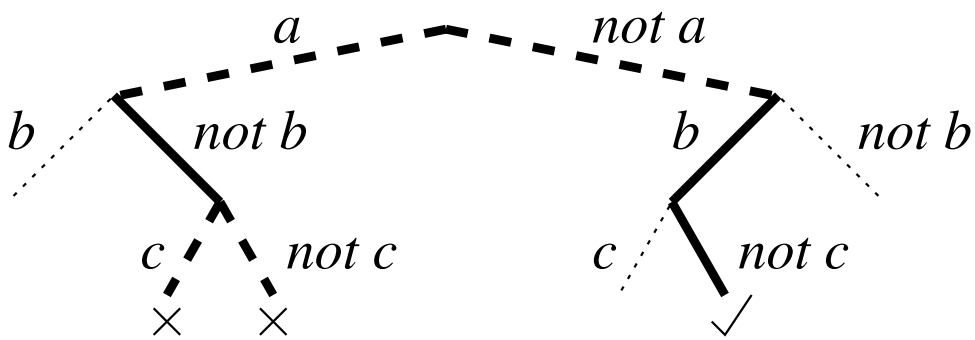
- Efficient grounding is at the heart of current systems
- Sophisticated techniques
  - DLV's grounder (built-in);
  - lparse (Smodels), gringo (clasp)
  - XASP, aspps
- Special techniques used:
  - "*Safe rules*" (DLV): every variable in a rule must occur in an unnegated atom in the body, whose predicate is not "$=$" or any another built-in predicate
  - *domain-restriction* (Smodels)
- Problem: Grounding bottleneck [Eiter *et al.*, 2007]

  Research on nonground evaluation (e.g., [Brüning and Schaub, 1999], [Leone *et al.*, 2006], [Calimeri *et al.*, 2008], [Lin and You, 2008], [Gebser *et al.*, 2008], [Palù *et al.*, 2008]);
  XASP (XSB Extensions)

Model search

- Applied for ground programs.
- Different Techniques:
  - Translations to SAT (e.g. Cmodels, ASSAT)
  - tailored search procedures (Smodels, DLV, NoMore, aspps, clasp)

a :– *not* b.
b :– *not* a.
c :– *not* c, a.



  - Backtracking procedures for assigning truth value to atoms
  - Similar to DPPL algorithm for SAT
  - Important: Heuristics (which atom/rule to consider next); involved
- Stability check: unfounded sets, reductions to UNSAT

The DLV System

`http://www.dbai.tuwien.ac.at/proj/dlv/`

- DLV is a state-of-the-art disjunctive answer set solver
- Developed at TU Wien / University of Calabria (since 1996)
- Possesses richer syntax than normal logic programs, resulting in **higher expressiveness**!
- Offers front-ends for specific KR-tasks (diagnosis, planning, etc.).

Features of DLV

- Language: **disjunctive extended logic programs**, no function symbols
- Additionally:
  - bounded integer arithmetic, and comparison built-ins
  - integrity constraints
  - weak constraints
  - aggregates
  - most recently: function symbols (DLV-Complex, r.e.-complete)
- Support for
  - answer set generation
  - *brave* and *cautious* reasoning
  - many extensions: DLVEX, DLVHEX, DLV^{DB}, DLT, DLV-Complex

DLV Syntax

- **Rules**

$$a_1 \ v \ \cdots \ v \ a_n \ :- \ b_1, \ldots, b_k, \ \texttt{not} \ b_{k+1}, \ldots, \ \texttt{not} \ b_m.$$

where $n \geq 1$, $m \geq 0$ and all a_i, b_j are atoms or strongly negated atoms (e.g. $-a$); no function symbols.

- **Integrity Constraints**

$$:- \ b_1, \ldots, b_k, \ \texttt{not} \ b_{k+1}, \ldots, \ \texttt{not} \ b_m.$$

can be regarded as rules with an empty (false) head.

- **Queries**

$$b_1, \ldots, b_k, \ \texttt{not} \ b_{k+1}, \ldots, \ \texttt{not} \ b_m?$$

Built-in Predicates

- Comparison Predicates:

$$<,\ >,\ <=,\ >=,\ ==,\ !=$$

- Arithmetic Predicates:

$$\#\texttt{int},\ \#\texttt{succ},\ +,\ *$$

$\#\texttt{int}(\texttt{X})$:	$\texttt{X}$ is known integer $(1 \leq \texttt{X} \leq \texttt{N})$.
$\#\texttt{succ}(\texttt{X}, \texttt{Y})$:	$\texttt{Y}$ is successor of $\texttt{X}$, i.e., $\texttt{Y} = \texttt{X} + 1$.
$+(\texttt{X}, \texttt{Y}, \texttt{Z})$:	$\texttt{Z} = \texttt{X} + \texttt{Y}$.
$*(\texttt{X}, \texttt{Y}, \texttt{Z})$:	$\texttt{Z} = \texttt{X} * \texttt{Y}$.

N.B. An upper bound for integers has to be specified when `dlv` is invoked.

Safety

Each variable occurring in a rule (resp. constraint) r in either

- the head,
- a default literal `not` b, or
- a built-in comparsion predicate,

must occur in at least 1 non-comparison `not`-free literal in the body of r.

Safe rules

```
a(X) :- not b(X), c(X).
a(X) :- X > Y, node(X), node(Y).
```

Unsafe rules

```
a(X) v -a(X).
a(X) :- not b(X).
:- X <= Y, node(X).
```

Declarative Problem Solving in DLV

Solve problems using disjunction/negation.

Maximum

Input: Employees and their salaries, represented by `empl(_,_)`.

Problem: Determine maximum salary of employees.

Solve Problem using projection and double negation!

```
-max(S) :- empl(N,S), empl(N1,S1), S < S1.
 max(S) :- empl(N,S), not -max(S).
```

Front-ends

- Besides the answer set semantics core, DLV offers front-ends for particular KR tasks:
  - diagnosis
  - inheritance reasoning
  - knowledge-based planning ($\mathcal{K}$ language)
- Also:
  - built-in front-end to SQL3
- Many external front ends to DLV exist (e.g., updates, preferences, plan diagnosis, execution monitoring, etc.)

Using DLV

- DLV is command-line oriented . . .
- . . . but there is also a simple GUI.

- Input is read from files whose names are passed on the command-line.
- If the command-line option "`--`" has been specified, input is also read from standard input (stdin).
- Output is printed to standard output (stdout), one line per model / answer set.

- Detailed documentation is at
  `http://www.dbai.tuwien.ac.at/proj/dlv/`

References I

Stefan Brüning and Torsten Schaub.

Avoiding non-ground variables.

In Anthony Hunter and Simon Parsons, editors, *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty (ECSQARU'99)*, volume 1638 of *LNAI*, pages 92–103. Springer, 1999.

Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone.

Computable Functions in ASP: Theory and Implementation.

In *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, volume 5366 of *LNCS*, pages 407–424. Springer, 2008.

T. Eiter, W. Faber, N. Leone, and G. Pfeifer.

Declarative problem-solving using the DLV system.

In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

References II

Thomas Eiter, Wolfgang Faber, Michael Fink, and Stefan Woltran.

Complexity results for answer set programming with bounded predicate arities and implications.

Annals of Mathematics and Artificial Intelligence, 51(2-4):123–165, 2007.

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele.

Engineering an Incremental ASP Solver.

In M.G. de La Banda and E. Pontelli, editors, *Proceedings 24th International Conference on Logic Programming (ICLP 2008)*, number 5366 in LNCS, pages 190–205. Springer, 2008.

Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello.

The DLV System for Knowledge Representation and Reasoning.

ACM Transactions on Computational Logic, 7(3):499–562, July 2006.

References III

Vladimir Lifschitz.

Answer Set Programming and Plan Generation.

Artificial Intelligence, 138:39–54, 2002.

F. Lin and J. You.

Abductive logic programming by nonground rewrite systems.

In *AAAI-08*, pages 480–485, 2008.

Ilkka Niemelä.

The implementation of answer set solvers.

Tutorial at ICLP'04: `http://www.tcs.hut.fi/~ini/papers/niemela-iclp04-tutorial.ps.gz`, 2004.

A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi.

Gasp: Answer set programming with lazy grounding.

In *LaSh 2008: LOGIC AND SEARCH - Computation of structures from declarative descriptions*, 2008.