Knowledge Representation for the Semantic Web

Lecture 7: Answer Set Programming II

Daria Stepanova

partially based on slides by Thomas Eiter

max planck institut informatik

D5: Databases and Information Systems
Max Planck Institute for Informatics

WS 2017/18

Unit Outline

Strong Negation

- Default negation "$not\, a$" means "a cannot be proved (derived) using rules," and that a is false by default (believed to be false).

Strong Negation

- Default negation "$not\, a$" means "a cannot be proved (derived) using rules," and that a is false by default (believed to be false).
- Strong negation $\neg a$ (also $-a$) means that a is (provably) false
- Both default and strong negation can be used in ASP

Strong Negation

- Default negation "$not\, a$" means "a cannot be proved (derived) using rules," and that a is false by default (believed to be false).
- Strong negation $\neg a$ (also $-a$) means that a is (provably) false
- Both default and strong negation can be used in ASP

"At a railroad crossing, cross the rails if no train approaches."

We may encode this scenario using one of the following two rules:

$$walk \leftarrow at(X), crossing(X), not\ train\_approaches(X). \qquad (r_1)$$

$$walk \leftarrow at(X), crossing(X), -train\_approaches(X). \qquad (r_2)$$

- r_1 fires if there is no information that a train approaches.
- r_2 fires if it is explictly <u>known</u> that no train approaches.

Constraints

- Constraints are rules with empty head which exclude invalid models.

$$\leftarrow q_1, \ldots, q_m, not\ r_1, \ldots, not\ r_n.$$

kills answer sets that

- contain $q_1, \ldots, q_m$, and
- do not contain $r_1, \ldots, r_n$.

Constraints

- Constraints are rules with empty head which exclude invalid models.

$$\leftarrow q_1, \ldots, q_m, not\ r_1, \ldots, not\ r_n.$$

kills answer sets that

- contain $q_1, \ldots, q_m$, and
- do not contain $r_1, \ldots, r_n$.

An equivalent version of the above rule is with a fresh predicate p:

$$p \leftarrow q_1, \ldots, q_m, not\ r_1, \ldots, not\ r_n, not\ p.$$

Constraints

- Constraints are rules with empty head which exclude invalid models.

$$\leftarrow q_1, \ldots, q_m, not\ r_1, \ldots, not\ r_n.$$

kills answer sets that

- contain $q_1, \ldots, q_m$, and
- do not contain $r_1, \ldots, r_n$.

An equivalent version of the above rule is with a fresh predicate p:

$$p \leftarrow q_1, \ldots, q_m, not\ r_1, \ldots, not\ r_n, not\ p.$$

Example: adjacent nodes cannot be colored with the same color

$\bot \leftarrow edge(X, Y), colored(X, Z), colored(Y, Z).$

Disjunction

The use of disjunction is natural

- to express indefinite knowledge:

$$female(X) \lor male(X) \leftarrow person(X).$$

$$broken(left\_arm, robot1) \lor broken(right\_arm, robot1).$$

Disjunction

The use of disjunction is natural

- to express indefinite knowledge:

$$female(X) \vee male(X) \leftarrow person(X).$$

$$broken(left\_arm, robot1) \vee broken(right\_arm, robot1).$$

- to express a "guess" and to create non-determinism.

$$ok(C) \vee -ok(C) \leftarrow component(C).$$

Minimality

- Semantics: disjunction is minimal (different from classical logic):

$$a \vee b \vee c.$$

Minimality

- Semantics: disjunction is minimal (different from classical logic):

$$a \vee b \vee c.$$

Minimal models: $\{a\}$, $\{b\}$, and $\{c\}$.

Minimality

- Semantics: disjunction is minimal (different from classical logic):

$$a \lor b \lor c.$$

  Minimal models: $\{a\}$, $\{b\}$, and $\{c\}$.

- Actually subset minimal:

$$a \lor b. \qquad a \lor c.$$

Minimality

- Semantics: disjunction is minimal (different from classical logic):

$$a \vee b \vee c.$$

  Minimal models: $\{a\}$, $\{b\}$, and $\{c\}$.

- Actually subset minimal:

$$a \vee b. \qquad a \vee c.$$

  Minimal models: $\{a\}$ and $\{b, c\}$.

Minimality

- Semantics: disjunction is minimal (different from classical logic):

$$a \vee b \vee c.$$

  Minimal models: $\{a\}$, $\{b\}$, and $\{c\}$.

- Actually subset minimal:

$$a \vee b. \qquad a \vee c.$$

  Minimal models: $\{a\}$ and $\{b, c\}$.

$$a \vee b. \qquad a \leftarrow b.$$

Minimality

- Semantics: disjunction is minimal (different from classical logic):

$$a \lor b \lor c.$$

  Minimal models: $\{a\}$, $\{b\}$, and $\{c\}$.

- Actually subset minimal:

$$a \lor b. \qquad a \lor c.$$

  Minimal models: $\{a\}$ and $\{b, c\}$.

$$a \lor b. \qquad a \leftarrow b.$$

  Models: $\{a\}$ and $\{a, b\}$, but only $\{a\}$ is minimal.

Minimality

- Semantics: disjunction is minimal (different from classical logic):

$$a \lor b \lor c.$$

  Minimal models: $\{a\}$, $\{b\}$, and $\{c\}$.

- Actually subset minimal:

$$a \lor b. \qquad a \lor c.$$

  Minimal models: $\{a\}$ and $\{b, c\}$.

$$a \lor b. \qquad a \leftarrow b.$$

  Models: $\{a\}$ and $\{a, b\}$, but only $\{a\}$ is minimal.

- But minimality is not necessarily exclusive:

$$a \lor b. \qquad b \lor c. \qquad a \lor c.$$

Minimality

- Semantics: disjunction is minimal (different from classical logic):

$$a \vee b \vee c.$$

  Minimal models: $\{a\}$, $\{b\}$, and $\{c\}$.

- Actually subset minimal:

$$a \vee b. \qquad a \vee c.$$

  Minimal models: $\{a\}$ and $\{b, c\}$.

$$a \vee b. \qquad a \leftarrow b.$$

  Models: $\{a\}$ and $\{a, b\}$, but only $\{a\}$ is minimal.

- But minimality is not necessarily exclusive:

$$a \vee b. \qquad b \vee c. \qquad a \vee c.$$

  Minimal models: $\{a, b\}$, $\{a, c\}$, and $\{b, c\}$.

Extended Logic Programs with Disjunctions

Extended Logic Programs

An extended disjunctive logic program (EDLP) is a finite set of rules

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, not\, c_1, \ldots, not\, c_n \qquad (k, m, n \geq 0) \tag{1}$$

where all a_i, b_j, c_l are atoms or strongly negated atoms.

Extended Logic Programs with Disjunctions

Extended Logic Programs

An extended disjunctive logic program (EDLP) is a finite set of rules

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, not\, c_1, \ldots, not\, c_n \qquad (k, m, n \geq 0) \tag{1}$$

where all a_i, b_j, c_l are atoms or strongly negated atoms.

Semantics:

- Stable models (answer sets) of EDLPs are defined similarly as for LPs, viewing $-p$ as a new predicate.

Extended Logic Programs with Disjunctions

Extended Logic Programs

An extended disjunctive logic program (EDLP) is a finite set of rules

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, not\, c_1, \ldots, not\, c_n \qquad (k, m, n \geq 0) \qquad (1)$$

where all a_i, b_j, c_l are atoms or strongly negated atoms.

Semantics:

- Stable models (answer sets) of EDLPs are defined similarly as for LPs, viewing $-p$ as a new predicate.
- Differences:
  - I must not contain atoms $p(c_1, \ldots, c_n)$, $-p(c_1, \ldots, c_n)$ (consistency)
  - I is a model of ground rule (1), if either $\{b_1, \ldots, b_m\} \not\subseteq I$ or $\{a_1, \ldots, a_k, c_1, \ldots, c_n\} \cap I \neq \emptyset$.

Extended Logic Programs with Disjunctions

Extended Logic Programs

An extended disjunctive logic program (EDLP) is a finite set of rules

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, not\, c_1, \ldots, not\, c_n \qquad (k, m, n \geq 0) \tag{1}$$

where all a_i, b_j, c_l are atoms or strongly negated atoms.

Semantics:

- Stable models (answer sets) of EDLPs are defined similarly as for LPs, viewing $-p$ as a new predicate.
- Differences:
  - I must not contain atoms $p(c_1, \ldots, c_n)$, $-p(c_1, \ldots, c_n)$ (consistency)
  - I is a model of ground rule (1), if either $\{b_1, \ldots, b_m\} \not\subseteq I$ or $\{a_1, \ldots, a_k, c_1, \ldots, c_n\} \cap I \neq \emptyset$.
  - Condition "M is the least model of P^M" is replaced by "M is a minimal model of P^M" (P^M may have multiple minimal models).

Example

Let P be the following program:

$$\begin{aligned} man(dilbert). \\ single(X) \lor husband(X) \leftarrow man(X). \end{aligned}$$

Example

Let P be the following program:

$$man(dilbert).$$
$$single(X) \vee husband(X) \leftarrow man(X).$$

- As P is "not"-free, $grnd(P)^M = grnd(P)$ for every M.

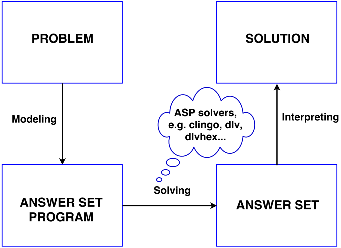
Example

Let P be the following program:

$$man(dilbert).$$
$$single(X) \vee husband(X) \leftarrow man(X).$$

- As P is "not"-free, $grnd(P)^M = grnd(P)$ for every M.
- Answer sets:
  - $M_1 = \{man(dilbert), single(dilbert)\}$, and
  - $M_2 = \{man(dilbert), husband(dilbert)\}$.

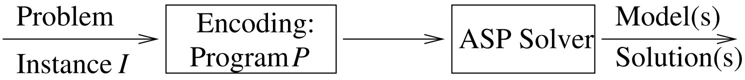
ASP Paradigm



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 (cont'd)

Compared to SAT solving, ASP offers more features:

- transitive closure
- negation as failure
- predicates and variables

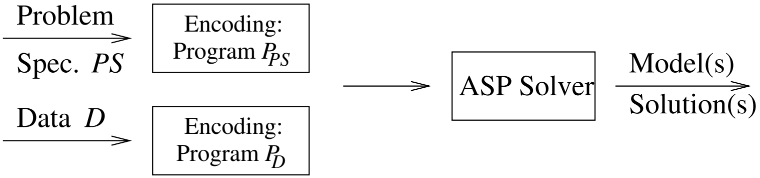
ASP Paradigm (cont'd)

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)



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], [Janhunen and Niemelä, 2016], [Gebser and Schaub, 2016] (+ additional component for computing optimal solutions).

Example: 3-Coloring

Problem specification P_{PS} (compact encoding)

$$g(X) \vee r(X) \vee b(X) \leftarrow node(X) \quad \Big\} \textbf{Guess}$$

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

Example: 3-Coloring

Problem specification P_{PS} (compact encoding)

$$g(X) \vee r(X) \vee b(X) \leftarrow node(X) \quad \} \textbf{ Guess}$$

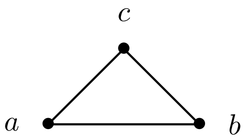
$$\left.\begin{array}{l} \leftarrow b(X), b(Y), edge(X,Y) \\ \leftarrow r(X), r(Y), edge(X,Y) \\ \leftarrow g(X), g(Y), edge(X,Y) \end{array}\right\} \textbf{ Check}$$

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

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

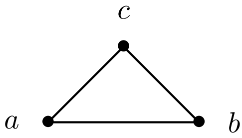
Correspondence: 3-colorings $\rightleftharpoons$ models:
$v \in V$ is colored with $c \in \{r, g, b\}$ iff $c(v)$ is in the model of $P_{PS} \cup P_D$.

Example: 3-Coloring (cont'd)

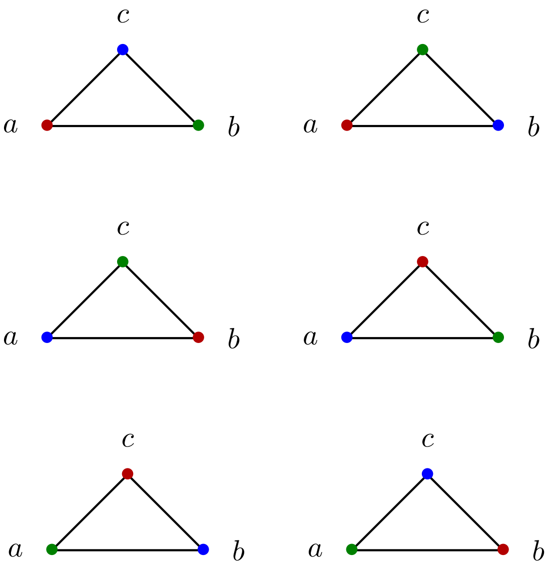


$$P_D = \{node(a), node(b), \\ node(c), edge(a, b), \\ edge(b, c), edge(a, c)\}$$

Example: 3-Coloring (cont'd)



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



Example: Hamiltonian Path/Cycle

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

Problem: find a path beginning at the starting node which visits all nodes of the graph exactly once.

Example: Hamiltonian Path/Cycle

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

Problem: find a path beginning at the starting node which visits all nodes of the graph exactly once.

$$inPath(X,Y) \vee outPath(X,Y) \leftarrow edge(X,Y). \quad \Big\} \textbf{Guess}$$

$$\left.\begin{array}{l} \leftarrow inPath(X,Y),\ inPath(X,Y_1),\ Y \neq Y_1. \\ \leftarrow inPath(X,Y),\ inPath(X_1,Y),\ X \neq X_1. \\ \leftarrow node(X),\ not\ reached(X). \end{array}\right\} \textbf{Check}$$

$$\left.\begin{array}{l} reached(X) \leftarrow start(X). \\ reached(X) \leftarrow reached(Y),\ inPath(Y,X). \end{array}\right\} \textbf{Auxiliary Predicate}$$

Example: Hamiltonian Path/Cycle

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

Problem: find a path/cycle[1] beginning at the starting node which visits all nodes of the graph exactly once.

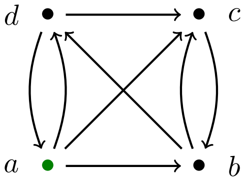
$$inPath(X,Y) \vee outPath(X,Y) \leftarrow edge(X,Y). \quad \Big\} \textbf{ Guess}$$

$$\left.\begin{array}{l} \leftarrow inPath(X,Y),\ inPath(X,Y_1),\ Y \neq Y_1. \\ \leftarrow inPath(X,Y),\ inPath(X_1,Y),\ X \neq X_1. \\ \leftarrow node(X),\ not\ reached(X). \\ \leftarrow not\ start\_reached. \end{array}\right\} \textbf{ Check}$$

$$\left.\begin{array}{l} reached(X) \leftarrow start(X). \\ reached(X) \leftarrow reached(Y),\ inPath(Y,X). \\ start\_reached \leftarrow start(Y), inPath(X,Y). \end{array}\right\} \textbf{ Auxiliary Predicate}$$

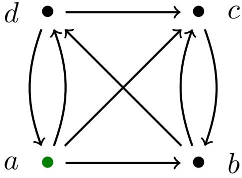
[1] The encoding for the Hamilthonian cycle contains red lines along with green ones.

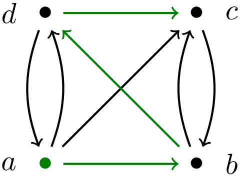
Example: Hamiltonian Path/Cycle (cont'd)

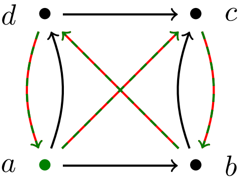


$$
\begin{aligned}
P_D = \{ & node(a), node(b), \\
& node(c), node(d), \\
& edge(a,b), edge(a,c), \\
& edge(c,b), edge(b,c), \\
& edge(b,d), edge(d,c), \\
& edge(d,a), edge(a,d), \\
& start(a)\}
\end{aligned}
$$

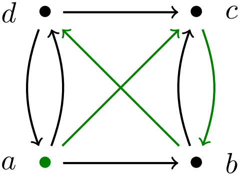
Example: Hamiltonian Path/Cycle (cont'd)

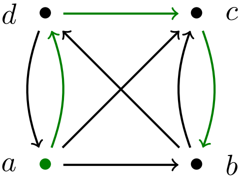






$$P_D = \{node(a), node(b), node(c), node(d), edge(a,b), edge(a,c), edge(c,b), edge(b,c), edge(b,d), edge(d,c), edge(d,a), edge(a,d), start(a)\}$$





Example: Course Assignment

Input: information about members and courses of a computer science (CS) dept cs

Problem:

- assign courses to members of the CS dept
- teachers must like the assigned course
- each member must teach 1-2 courses

Example: Course Assignment

Input: information about members and courses of a computer science (CS) dept cs

Problem:

- assign courses to members of the CS dept
- teachers must like the assigned course
- each member must teach 1-2 courses

P_D:

$member(sam, cs).$ $member(bob, cs).$ $member(tom, cs).$

Example: Course Assignment

Input: information about members and courses of a computer science (CS) dept cs

Problem:

- assign courses to members of the CS dept
- teachers must like the assigned course
- each member must teach 1-2 courses

P_D:

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

Example: Course Assignment

Input: information about members and courses of a computer science (CS) dept cs

Problem:

- assign courses to members of the CS dept
- teachers must like the assigned course
- each member must teach 1-2 courses

P_D:

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

Example: Course Assignment (cont'd)

Problem specification P_{PS}:

% assign a course to a teacher who likes it, by default

$$teach(X,Y) \leftarrow member(X,cs), course(Y,cs), likes(X,Y), not\ -teach(X,Y).$$

% determine when a course should not be assigned to a teacher

$$-teach(X,Y) \leftarrow member(X,cs), course(Y,cs), teach(X_1,Y), X_1 \neq X.$$

Example: Course Assignment (cont'd)

Problem specification P_{PS}:

% assign a course to a teacher who likes it, by default

$$teach(X,Y) \leftarrow member(X,cs), course(Y,cs), likes(X,Y), not\ -teach(X,Y).$$

% determine when a course should not be assigned to a teacher

$$-teach(X,Y) \leftarrow member(X,cs), course(Y,cs), teach(X_1,Y), X_1 \neq X.$$

% check each cs member has some course

$$has\_course(X) \leftarrow member(X,cs), teach(X,Y).$$

$$\leftarrow member(X,cs), not\ has\_course(X).$$

Example: Course Assignment (cont'd)

Problem specification P_{PS}:

% assign a course to a teacher who likes it, by default

$$teach(X,Y) \leftarrow member(X,cs), course(Y,cs), likes(X,Y), not\ -teach(X,Y).$$

% determine when a course should not be assigned to a teacher

$$-teach(X,Y) \leftarrow member(X,cs), course(Y,cs), teach(X_1,Y), X_1 \neq X.$$

% check each cs member has some course

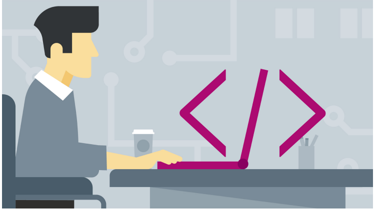
$$has\_course(X) \leftarrow member(X,cs), teach(X,Y).$$

$$\leftarrow member(X,cs), not\ has\_course(X).$$

% check each cs member has at most 2 courses

$$\leftarrow teach(X,Y_1), teach(X,Y_2), teach(X,Y_3), Y_1 \neq Y_2, Y_1 \neq Y_3, Y_2 \neq Y_3.$$

Programming Techniques



Programming Techniqes

- With the "guess and check paradigm", one may use different techniques to solve particular tasks

  E.g.,

  - choice of exactly one element from a set
  - computing maximum / minimum values (use double negation)
  - testing a property for all elements in a set (iteration over a set)
  - testing a co-NP hard property (saturation)
  - modularization

- We do not discuss here saturation (see [Eiter *et al.*, 2009])

Note: extensions of ASP allow to test properties of a set / choose elements elegantly

Selecting One Element from a Set

- Task: given a set, defined by predicate $p(X)$, select exactly one element from it (if nonempty).
- More general variant: $p(\vec{X}, \vec{Y})$, where $\vec{X} = X_1, \ldots, X_n$, $\vec{Y} = Y_1, \ldots, Y_m$, select for each $\vec{X}$ exactly one $\vec{Y}$ (if possible)
  - Implicitly, already done in the above course assignment problem

Selecting One Element from a Set

- Task: given a set, defined by predicate $p(X)$, select exactly one element from it (if nonempty).
- More general variant: $p(\vec{X}, \vec{Y})$, where $\vec{X} = X_1, \ldots, X_n$, $\vec{Y} = Y_1, \ldots, Y_m$, select for each $\vec{X}$ exactly one $\vec{Y}$ (if possible)
  - Implicitly, already done in the above course assignment problem

Select one element from a set: Normal rule encoding

$$sel(\vec{X}, \vec{Y}) \leftarrow p(\vec{X}, \vec{Y}), not\ -sel(\vec{X}, \vec{Y}).$$

$$-sel(\vec{X}, \vec{Y}) \leftarrow p(\vec{X}, \vec{Y}), sel(\vec{X}, \vec{Z}), Y_i \neq Z_i. \qquad i = 1, \ldots, m$$

where $\vec{Z} = Z_1, \ldots, Z_m$,

Selecting One Element from a Set (cont'd)

Example: Course assignment

- $p(X, Y)$ is $member(Y, cs), course(X, cs), likes(Y, X)$ and $sel(X, Y)$ is $teach(Y, X)$.
- could define an auxiliary rule

$$p(X, Y) \leftarrow member(Y, cs), course(X, cs), likes(Y, X)$$

Select one element from a set: Disjunctive rule encoding

$$sel(\vec{X}, \vec{Y}) \leftarrow p(\vec{X}, \vec{Y}), not\ -sel(\vec{X}, \vec{Y}).$$

$$-sel(\vec{X}, \vec{Y}) \vee -sel(\vec{X}, \vec{Z}) \leftarrow p(\vec{X}, \vec{Y}), p(\vec{X}, \vec{Z}), Y_i \neq Z_i. \qquad i = 1, \ldots, m$$

Selecting One Element from a Set (cont'd)

Example: Course assignment

- $p(X,Y)$ is $member(Y,cs), course(X,cs), likes(Y,X)$ and $sel(X,Y)$ is $teach(Y,X)$.
- could define an auxiliary rule

$$p(X,Y) \leftarrow member(Y,cs), course(X,cs), likes(Y,X)$$

Select one element from a set: Disjunctive rule encoding

$$sel(\vec{X},\vec{Y}) \leftarrow p(\vec{X},\vec{Y}), not\ -sel(\vec{X},\vec{Y}).$$

$$-sel(\vec{X},\vec{Y}) \vee -sel(\vec{X},\vec{Z}) \leftarrow p(\vec{X},\vec{Y}), p(\vec{X},\vec{Z}), Y_i \neq Z_i. \qquad i = 1,\ldots,m$$

In some answer set solvers, special syntax is available (see ASP-Core2):

$$1\{sel(\vec{X},\vec{Y}) : p(\vec{X},\vec{Y})\}1 \leftarrow p(\vec{X},\vec{Z})$$

Use of Double Negation

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

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 (cont'd)

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

Iteration over a Set

- Testing a property, say $Prop$, for all elements of a set S without negation
- This may be needed in some contexts:
  - in combination with other techniques, e.g., saturation (see [Eiter *et al.*, 2009]), or
  - if negation could lead to undesired behavior (e.g., cyclic negation).

Iteration over a Set

- Testing a property, say $Prop$, for all elements of a set S without negation
- This may be needed in some contexts:
  - in combination with other techniques, e.g., saturation (see [Eiter *et al.*, 2009]), or
  - if negation could lead to undesired behavior (e.g., cyclic negation).

- walk through all elements of set S, from the first to the last element;
- check whether property $Prop$ holds up to the current element y
  $\Leftrightarrow$ holds for y and holds up to for x, where y is the successor of x;
- when $Prop$ holds up to the last element, it holds for all elements of S

Iteration over a Set

- Testing a property, say $Prop$, for all elements of a set S without negation
- This may be needed in some contexts:
  - in combination with other techniques, e.g., saturation (see [Eiter *et al.*, 2009]), or
  - if negation could lead to undesired behavior (e.g., cyclic negation).

- walk through all elements of set S, from the first to the last element;
- check whether property $Prop$ holds up to the current element y
  $\Leftrightarrow$ holds for y and holds up to for x, where y is the successor of x;
- when $Prop$ holds up to the last element, it holds for all elements of S

- Note: this is a form of *finite induction*
- Use an enumeration of S with predicates $first(\_)$, $succ(\_,\_)$, $last(\_)$
  - Easy for static S, more involved for dynamically computed S

Example: Hamiltonian Path 2/Reachability

- Variant: no use of negation in checking that all nodes are reached (do not immediately kill stable model candidate):

$$\leftarrow node(X),\ not\ reached(X).$$

- Check that all nodes of the graph are reached via the selected edges ($inPath(X,Y)$) by iteration ($S =$ nodes of the graph)
- Use
  - $all\_reached\_upto(\_)$
  - $all\_reached$
- Supply in the input an enumeration of the nodes via $first(\_), succ(\_,\_), last(\_)$
  - Alternative: build the enumeration *dynamically* in the progam, using e.g. string comparison.

Example: Hamiltonian Path 2 (cont'd)

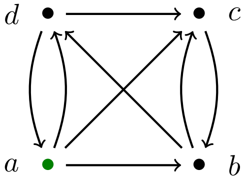
$$inPath(X,Y) \vee outPath(X,Y) \leftarrow edge(X,Y).$$ **Guess**

$$\leftarrow inPath(X,Y),\ inPath(X,Y_1),\ Y \neq Y_1.$$
$$\leftarrow inPath(X,Y),\ inPath(X_1,Y),\ X \neq X_1.$$ **Check**

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

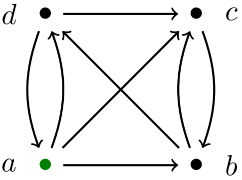
$$all\_reached\_upto(X) \leftarrow first(X), reached(X).$$
$$all\_reached\_upto(X) \leftarrow all\_reached\_upto(Y), succ(Y,X), reached(X).$$
$$all\_reached \leftarrow last(X), all\_reached\_upto(X).$$ **reached = nodes**

Example: Hamiltonian Path 2 (cont'd)

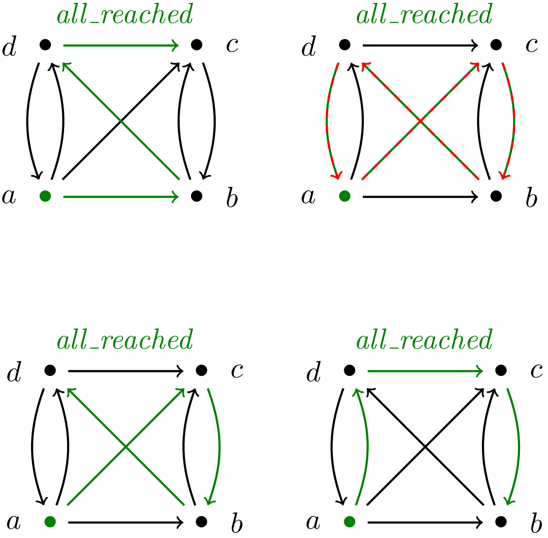


$$P_D = \{edge(a,b), edge(a,c),$$
$$edge(c,b), edge(b,c),$$
$$edge(b,d), edge(d,c),$$
$$edge(d,a), edge(a,d),$$
$$first(a), succ(a,b),$$
$$succ(b,c), succ(c,d),$$
$$last(d), start(a)\}$$

Example: Hamiltonian Path 2 (cont'd)



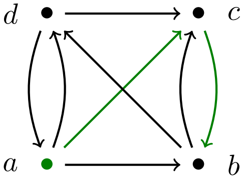
$$P_D = \{edge(a,b), edge(a,c), edge(c,b), edge(b,c), edge(b,d), edge(d,c), edge(d,a), edge(a,d), first(a), succ(a,b), succ(b,c), succ(c,d), last(d), start(a)\}$$



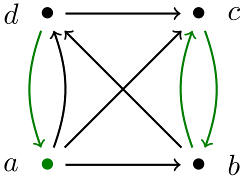
Example: Hamiltonian Path 2 (cont'd)

Some path guesses not reaching all nodes from a:

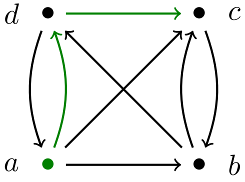
$all\_reached\_upto(c)$



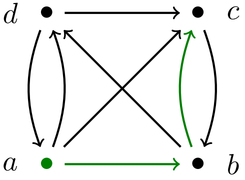
$all\_reached\_upto(a)$



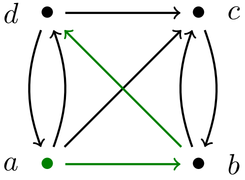
$all\_reached\_upto(a)$



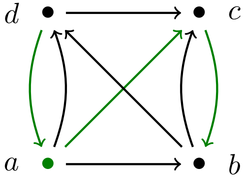
$all\_reached\_upto(c)$



$all\_reached\_upto(b)$



$all\_reached\_upto(c)$



Modularization

- Do not reinvent the wheel: reuse solutions to basic problems.
- Program Splitting: syntactic means to
  - develop larger programs by combining parts, and to
  - compute answer sets layer by layer (by composition).

Modularization

- Do not reinvent the wheel: reuse solutions to basic problems.
- Program Splitting: syntactic means to
  - develop larger programs by combining parts, and to
  - compute answer sets layer by layer (by composition).

Program splitting

Suppose (ground) P can be split into $P = P_1 \cup P_2$, such that every atom A that occurs in P_1 ("bottom part") occurs in P_2 ("top part") only in rule bodies (i.e., A is "defined" entirely in P_1). Then

$$AS(P) = \bigcup_{M \in AS(P_1)} AS(P_2 \cup M).$$

$AS(P)$ = set of answer sets of P

- **Examples**: "stratified" programs, like GCD; guess&check
- Versions of ASP with modules, macros etc. are available

Answer Set Solvers

Answer Set Solvers

(see also http://en.wikipedia.org/wiki/Answer_set_programming)

ASPERIX	www.info.univ-angers.fr/pub/claire/asperix/
ASSAT	assat.cs.ust.hk/
CLASP [1]	potassco.sourceforge.net/#clasp/
CMODELS	www.cs.utexas.edu/users/tag/cmodels/
DLV [2]	www.dbai.tuwien.ac.at/proj/dlv/
ASPTOOLS	research.ics.aalto.fi/software/asp/
ME-ASP	www.mat.unical.it/ricca/me-asp/
OMIGA	www.kr.tuwien.ac.at/research/systems/omiga
SMODELS	www.tcs.hut.fi/Software/smodels/
WASP	www.mat.unical.it/ricca/wasp/
XASP	xsb.sourceforge.net/, distributed with XSB
	....

1 + CLASPD, CLINGO, CLINGCON etc. (http://potassco.sourceforge.net/)

2 + DLVHEX, DLV^{DB}, DLT, DLV-COMPLEX, ONTO-DLV etc.

Answer Set Solvers

(see also http://en.wikipedia.org/wiki/Answer_set_programming)

ASPERIX	www.info.univ-angers.fr/pub/claire/asperix/
ASSAT	assat.cs.ust.hk/
CLASP [1]	potassco.sourceforge.net/#clasp/
CMODELS	www.cs.utexas.edu/users/tag/cmodels/
DLV [2]	www.dbai.tuwien.ac.at/proj/dlv/
ASPTOOLS	research.ics.aalto.fi/software/asp/
ME-ASP	www.mat.unical.it/ricca/me-asp/
OMIGA	www.kr.tuwien.ac.at/research/systems/omiga
SMODELS	www.tcs.hut.fi/Software/smodels/
WASP	www.mat.unical.it/ricca/wasp/
XASP	xsb.sourceforge.net/, distributed with XSB
	....

[1] + CLASPD, CLINGO, CLINGCON etc. (http://potassco.sourceforge.net/)
[2] + DLVHEX, DLV^{DB}, DLT, DLV-COMPLEX, ONTO-DLV etc.

- Many ASP solvers are available (mostly function-free programs)
- clasp was first ASP solver competitive to top SAT solvers
- another state-of-the-art solver is dlv

Evaluation Approaches

- Different methods and evaluation approaches:
  - resolution-based
  - forward chaining
  - lazy grounding AsperiX, Omiga
  - translation-based (see below)
  - meta-interpretation

Evaluation Approaches

- Different methods and evaluation approaches:
  - resolution-based
  - forward chaining
  - lazy grounding AsperiX, Omiga
  - translation-based (see below)
  - meta-interpretation

Predominant solver approach

intelligent grounding + model search (solving)

2-Level Architecture

1. **Intelligent grounding**

   Given a program P, generate a (subset) of $grnd(P)$ that has the same models

2-Level Architecture

1. **Intelligent grounding**

   Given a program P, generate a (subset) of $grnd(P)$ that has the same models

2. **Solving: Model search**

   More complicated than in SAT/CSP Solving:

   - candidate generation (classical model)
   - model checking (stability, foundedness!)

2-Level Architecture

1. **Intelligent grounding**

   Given a program P, generate a (subset) of $grnd(P)$ that has the same models

2. **Solving: Model search**

   More complicated than in SAT/CSP Solving:

   - candidate generation (classical model)
   - model checking (stability, foundedness!)
     - for SAT, model checking is feasible in logarithmic space
     - for normal propositional programs, model checking is PTime-complete
     - for disjunctive propositional programs, model checking is $\mathrm{co\text{-}NP}$-complete

Intelligent Grounding

- Grounding is a hard problem

$$bit(0). \quad bit(1). \tag{2}$$

$$p(X1, ..., Xn) \leftarrow bit(X1), ..., bit(Xn). \tag{3}$$

Intelligent Grounding

- Grounding is a hard problem

$$bit(0).\quad bit(1). \tag{2}$$

$$p(X1, ..., Xn) \leftarrow bit(X1), ..., bit(Xn). \tag{3}$$

- In the worst case, grounding time is exponential in the input size

Intelligent Grounding

- Grounding is a hard problem

$$bit(0).\ \ bit(1). \tag{2}$$

$$p(X1, ..., Xn) \leftarrow bit(X1), ..., bit(Xn). \tag{3}$$

- In the worst case, grounding time is exponential in the input size
- Getting the “right” rules is difficult, already for positive programs

Intelligent Grounding

- Grounding is a hard problem

$$bit(0). \quad bit(1). \tag{2}$$

$$p(X1, ..., Xn) \leftarrow bit(X1), ..., bit(Xn). \tag{3}$$

- In the worst case, grounding time is exponential in the input size
- Getting the "right" rules is difficult, already for positive programs
- Efficient grounding is at the heart of current systems
  - dlv's grounder (built-in);
  - lparse (smodels), gringo (clasp)

Intelligent Grounding

- Grounding is a hard problem

$$bit(0).\quad bit(1). \tag{2}$$

$$p(X1, ..., Xn) \leftarrow bit(X1), ..., bit(Xn). \tag{3}$$

- In the worst case, grounding time is exponential in the input size
- Getting the "right" rules is difficult, already for positive programs
- Efficient grounding is at the heart of current systems
  - dlv's grounder (built-in);
  - lparse (smodels), gringo (clasp)
- Special techniques used:

Intelligent Grounding

- Grounding is a hard problem

$$bit(0).\;\; bit(1). \tag{2}$$
$$p(X1,...,Xn) \leftarrow bit(X1),...,bit(Xn). \tag{3}$$

- In the worst case, grounding time is exponential in the input size
- Getting the "right" rules is difficult, already for positive programs
- Efficient grounding is at the heart of current systems
  - dlv's grounder (built-in);
  - lparse (smodels), gringo (clasp)

- Special techniques used:
  - "safe rules" (dlv): each variable in a rule occurs in the body in an unnegated atom with non-built-in predicate (exception: $X = c$)

Intelligent Grounding

- Grounding is a hard problem

$$bit(0). \;\; bit(1). \tag{2}$$

$$p(X1, ..., Xn) \leftarrow bit(X1), ..., bit(Xn). \tag{3}$$

- In the worst case, grounding time is exponential in the input size
- Getting the "right" rules is difficult, already for positive programs
- Efficient grounding is at the heart of current systems
  - dlv's grounder (built-in);
  - lparse (smodels), gringo (clasp)

- Special techniques used:
  - "safe rules" (dlv): each variable in a rule occurs in the body in an unnegated atom with non-built-in predicate (exception: $X = c$)
  - domain-restriction (smodels)

Intelligent Grounding

- Grounding is a hard problem

$$bit(0). \;\; bit(1). \tag{2}$$
$$p(X1, ..., Xn) \leftarrow bit(X1), ..., bit(Xn). \tag{3}$$

- In the worst case, grounding time is exponential in the input size
- Getting the "right" rules is difficult, already for positive programs
- Efficient grounding is at the heart of current systems
  - dlv's grounder (built-in);
  - lparse (smodels), gringo (clasp)

- Special techniques used:
  - "safe rules" (dlv): each variable in a rule occurs in the body in an unnegated atom with non-built-in predicate (exception: $X = c$)
  - domain-restriction (smodels)
  - deductive db methods: semi-naive evaluation, magic sets, ...

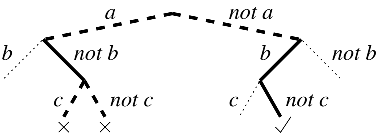
Solving: Model Search

- Applied to ground programs.

Solving: Model Search

- Applied to ground programs.
- Early solvers (e.g. smodels, dlv): native methods
  - inspired by Davis-Putnam-Logemann Loveland (DPLL) for SAT
    - 3 basic operations: decision, propagate, backtrack
  - special propagation for ASP, e.g.,
    - dlv: *must-be-true* propagation (supportedness), . . .

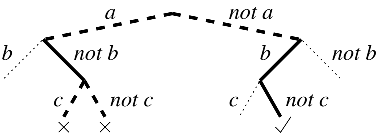
a:− *not b*.
b:− *not a*.
c:− *not c*, *a*.



Solving: Model Search

- Applied to ground programs.
- Early solvers (e.g. smodels, dlv): native methods
  - inspired by Davis-Putnam-Logemann Loveland (DPLL) for SAT
    - 3 basic operations: decision, propagate, backtrack
  - special propagation for ASP, e.g.,
    - dlv: *must-be-true* propagation (supportedness), . . .

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

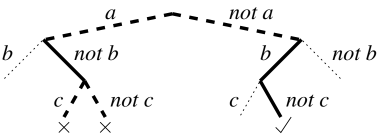


  - important: heuristics (which atom/rule is next?)
  - chronological backtrack-search improved by backjumping and look-back heuristics

Solving: Model Search

- Applied to ground programs.
- Early solvers (e.g. smodels, dlv): native methods
  - inspired by Davis-Putnam-Logemann Loveland (DPLL) for SAT
    - 3 basic operations: decision, propagate, backtrack
  - special propagation for ASP, e.g.,
    - dlv: *must-be-true* propagation (supportedness), ...

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



    - important: heuristics (which atom/rule is next?)
    - chronological backtrack-search improved by backjumping and look-back heuristics
- Stability check: unfounded sets, reductions to UNSAT (disj. ASP)

ASP Solving Approaches

- Predominant to date: modern SAT techniques (clause driven conflict learning, CDCL)
- Export of techniques from ASP to SAT (optimization issues)
- **Genuine conflict-driven ASP solvers**
  - clasp, wasp
- **Translation based solving:** to
  - SAT: assat, cmodels, lp2sat (multiple SAT solver calls)
  - SAT modulo theories (SMT) aspmt
  - Mixed Integer Programming (CPLEX backend)
- **Cross translation:** intermediate format to ease cross translation
  - SAT modulo acyclicity
    - interconnect graph based constraints with clausal constraints
    - can postpone choice of the target format to last step solver).
- **Portfolio solvers**
  - claspfolio: combines variants of clasp
  - ME-ASP: multi-engine portfolio ASP solver

Summary

1. More about logic programs
   - Strong negation, disjunction
2. The answer set programming paradigm
   - The guess and check methodology
3. Programming techniques
   - Element selection
   - Use of double negation
   - Iteration over a set
   - Modularization
4. Answer set solvers
   - Intelligent grounding and solving

References I

- 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.
- Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner.
  Answer set programming: A primer.
  In Sergio Tessaris and Enrico Franconi et al., editors, *Reasoning Web, Fifth International Summer School 2008, Bressanone Italy, August 30–September 4, 2009, Tutorial Lectures*, number 5689 in LNCS, pages 40–110. Springer, 2009.
- Martin Gebser and Torsten Schaub.
  Modeling and language extensions.
  AI Magazine, 37(3):33–44, 2016.

References II

Tomi Janhunen and Ilkka Niemelä.
The answer set programming paradigm.
AI Magazine, 37(3):13–24, 2016.

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.

Vladimir Lifschitz.
Answer Set Programming and Plan Generation.
Artificial Intelligence, 138:39–54, 2002.