Knowledge Representation for the Semantic Web

Lecture 7: Answer Set Programming II

Daria Stepanova

partially based on slides by Thomas Eiter

D5: Databases and Information Systems
Max Planck Institute for Informatics

WS 2017/18
Unit Outline

More about Logic Programs

ASP Paradigm

Programming Techniques

Answer Set Solvers
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 $\lnot a$) 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:

- $\text{walk} \leftarrow \text{at}(X), \text{crossing}(X), \neg \text{train approaches}(X)$.

- $\text{walk} \leftarrow \text{at}(X), \text{crossing}(X), \neg a$.

$r_1$ fires if there is no information that a train approaches.

$r_2$ fires if it is explicitly known that no train approaches.
Strong Negation

- **Default negation** “\textit{not }a\text{"} means “\textit{a cannot be proved (derived) using rules,"} and that \textit{a} is false by default (believed to be false).

- **Strong negation** $\neg a$ (also $\bar{a}$) means that \textit{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:

\begin{align*}
\text{walk} & \leftarrow \text{at}(X), \text{crossing}(X), \text{not train\textunderscore approaches}(X). & (r_1) \\
\text{walk} & \leftarrow \text{at}(X), \text{crossing}(X), \neg \text{train\textunderscore approaches}(X). & (r_2)
\end{align*}

- $r_1$ fires if there is no information that a train approaches.
- $r_2$ fires if it is explicitly known that no train approaches.
Constraints

- **Constraints** are rules with empty head which exclude invalid models.
  \[ \leftarrow q_1, \ldots, q_m, \text{not } r_1, \ldots, \text{not } r_n. \]

  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.

\[ \leftarrow q_1, \ldots, q_m, \text{not } r_1, \ldots, \text{not } r_n. \]

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 \( p \):

\[ p \leftarrow q_1, \ldots, q_m, \text{not } r_1, \ldots, \text{not } r_n, \text{not } p. \]
Constraints

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

\[ \leftarrow q_1, \ldots, q_m, \text{not } r_1, \ldots, \text{not } r_n. \]

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 \( p \):

\[ p \leftarrow q_1, \ldots, q_m, \text{not } r_1, \ldots, \text{not } r_n, \text{not } p. \]

**Example:** adjacent nodes cannot be colored with the same color

\[ \bot \leftarrow \text{edge}(X, Y), \text{colored}(X, Z), \text{colored}(Y, Z). \]
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:

\[
female(X) \lor male(X) \leftarrow person(X).
\]

\[
broken(left\_arm, robot1) \lor broken(right\_arm, robot1).
\]

- to express a “guess” and to create non-determinism.

\[
ok(C) \lor \neg ok(C) \leftarrow component(C).
\]
Minimality

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

\[ a \lor b \lor c. \]
Minimality

- Semantics: disjunction is **minimal** (different from classical logic):
  \[ a \lor b \lor c. \]

  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:

\[ a \lor b. \quad a \lor c. \]
Minimality

- Semantics: disjunction is minimal (different from classical logic):
  \[ a \lor b \lor c. \]

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

- Actually subset minimal:
  \[ a \lor b. \quad a \lor c. \]

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

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

\[ a \lor b \lor c. \]

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

- Actually subset minimal:

\[ a \lor b. \quad a \lor c. \]

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

\[ a \lor b. \quad a \leftarrow b. \]
Minimality

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

  \[ a \lor b \lor c. \]

  Minimal models: \( \{a\}, \{b\}, \text{ and } \{c\} \).

- Actually **subset minimal**:

  \[ a \lor b. \quad a \lor c. \]

  Minimal models: \( \{a\} \text{ and } \{b, c\} \).

- But minimality is not necessarily exclusive:

  \[ a \lor b. \quad b \lor c. \quad a \lor c. \]

  Minimal models: \( \{a, b\}, \{a, c\}, \text{ and } \{b, c\} \).

Models: \( \{a\} \text{ and } \{a, b\} \), but only \( \{a\} \) is minimal.
Minimality

- Semantics: disjunction is \textbf{minimal} (different from classical logic):
  \[ a \vee b \vee c. \]

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

- Actually \textbf{subset minimal}:
  \[ a \vee b. \quad a \vee c. \]

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

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

- But minimality is \textbf{not necessarily exclusive}:
  \[ a \vee b. \quad b \vee c. \quad a \vee c. \]
Minimality

- Semantics: disjunction is minimal (different from classical logic):
  \[ a \lor b \lor c. \]

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

- Actually subset minimal:
  \[ a \lor b. \quad a \lor c. \]

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

- But minimality is not necessarily exclusive:
  \[ a \lor b. \quad b \lor c. \quad a \lor c. \]

  Minimal models: \{a, b\}, \{a, c\}, and \{b, c\}. 
Extended Logic Programs

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

\[ a_1 \lor \cdots \lor a_k \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n \quad (k, m, n \geq 0) \quad (1) \]

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

\[ a_1 \lor \cdots \lor a_k \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n \quad (k, m, n \geq 0) \quad (1) \]

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 \( \neg p \) as a new predicate.
Extended Logic Programs with Disjunctions

Extended Logic Programs

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

\[ a_1 \lor \cdots \lor a_k \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n \quad (k, m, n \geq 0) \quad (1) \]

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 \( \lnot p \) as a new predicate.
- Differences:
  - \( I \) must not contain atoms \( p(c_1, \ldots, c_n), \lnot p(c_1, \ldots, c_n) \) (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

\[ a_1 \lor \cdots \lor a_k \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n \]  \hspace{1cm} (k, m, n \geq 0) \hspace{1cm} (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:

$$\begin{align*}
\text{man}(\text{dilbert}). \\
\text{single}(X) \lor \text{husband}(X) &\leftarrow \text{man}(X).
\end{align*}$$
Example

Let $P$ be the following program:

\[
\begin{align*}
\text{man}(\text{dilbert}). \\
\text{single}(X) \lor \text{husband}(X) & \leftarrow \text{man}(X).
\end{align*}
\]

- As $P$ is "not"-free, $\text{grnd}(P)^M = \text{grnd}(P)$ for every $M$. 
Example

Let $P$ be the following program:

\[
\begin{align*}
\text{man}(\text{dilbert}). \\
\text{single}(X) \lor \text{husband}(X) &\leftarrow \text{man}(X).
\end{align*}
\]

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

- Answer sets:
  - $M_1 = \{\text{man}(\text{dilbert}), \text{single}(\text{dilbert})\}$, and
  - $M_2 = \{\text{man}(\text{dilbert}), \text{husband}(\text{dilbert})\}$. 

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 $G$), and

2. **Check**: use further rules and/or constraints to test candidate solution if it is a proper solution for our problem (program part $C$).
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 $G$), and

2. **Check**: use further rules and/or constraints to test candidate solution if it is a proper solution for our problem (program part $C$).

From another perspective:

- $G$: defines the search space
- $C$: prunes illegal branches.

Example: 3-Coloring

Problem specification $P_{PS}$ (compact encoding)

\[
g(X) \lor r(X) \lor b(X) \leftarrow node(X) \} \text{ Guess}
\]
\[
\leftarrow b(X), b(Y), edge(X, Y)
\leftarrow r(X), r(Y), edge(X, Y)
\leftarrow g(X), g(Y), edge(X, Y) \] \text{ Check}

Data

$P_{PD}$: Graph $G = (V, E)$

$P_{PD} = \{\text{node}(v) \mid v \in V\} \cup \{\text{edge}(v, w) \mid (v, w) \in E\}$. Correspondence: 3-colorings $\leftrightarrow$ models: $v \in V$ is colored with $c \in \{r, g, b\}$ iff $c(v)$ is in the model of $P_{PS} \cup P_{PD}$. 
**Example: 3-Coloring**

**Problem specification** $P_{PS}$ (compact encoding)

$$
\begin{align*}
g(X) \lor r(X) \lor b(X) & \leftarrow \text{node}(X) \quad \text{\textbf{Guess}} \\
\leftarrow b(X), b(Y), \text{edge}(X, Y) \quad \text{\textbf{Check}} \\
\leftarrow r(X), r(Y), \text{edge}(X, Y) \\
\leftarrow g(X), g(Y), \text{edge}(X, Y)
\end{align*}
$$

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

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

Correspondence: 3-colorings $\Leftrightarrow$ models:

$v \in V$ is colored with $c \in \{r, g, b\}$ iff $c(v)$ is in the model of $P_{PS} \cup P_D$. 

---

More about Logic Programs
ASP Paradigm
Programming Techniques
Answer Set Solvers
Example: 3-Coloring (cont’d)

\[ PD = \{ \text{node}(a), \text{node}(b), \text{node}(c), \text{edge}(a, b), \text{edge}(b, c), \text{edge}(a, c) \} \]
Example: 3-Coloring (cont’d)

\[ \mathcal{P}_D = \{ \text{node}(a), \text{node}(b), \text{node}(c), \text{edge}(a, b), \text{edge}(b, c), \text{edge}(a, c) \} \]
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 \texttt{node(\_)} and \texttt{edge(\_, \_)} and a starting node \texttt{start(\_)}.

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

\begin{align*}
\text{inPath}(X, Y) \lor \text{outPath}(X, Y) & \leftarrow \text{edge}(X, Y). \quad \text{Guess} \\
& \leftarrow \text{inPath}(X, Y), \text{inPath}(X, Y_1), Y \neq Y_1. \\
& \leftarrow \text{inPath}(X, Y), \text{inPath}(X_1, Y), X \neq X_1. \\
& \leftarrow \text{node}(X), \text{not reached}(X). \quad \text{Check}
\end{align*}

\begin{align*}
\text{reached}(X) & \leftarrow \text{start}(X). \\
\text{reached}(X) & \leftarrow \text{reached}(Y), \text{inPath}(Y, X). \quad \text{Auxiliary Predicate}
\end{align*}
Example: Hamiltonian Path/Cycle

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

Problem: find a path/cycle\(^1\) beginning at the starting node which visits all nodes of the graph exactly once.

\[
\text{inPath}(X, Y) \lor \text{outPath}(X, Y) \leftarrow \text{edge}(X, Y). \quad \text{Guess}
\]

\[
\leftarrow \text{inPath}(X, Y), \text{inPath}(X, Y_1), Y \neq Y_1.
\]

\[
\leftarrow \text{inPath}(X, Y), \text{inPath}(X_1, Y), X \neq X_1.
\]

\[
\leftarrow \text{node}(X), \text{not reached}(X).
\]

\[
\leftarrow \text{not start_reached}.
\]

\[
\text{reached}(X) \leftarrow \text{start}(X).
\]

\[
\text{reached}(X) \leftarrow \text{reached}(Y), \text{inPath}(Y, X).
\]

\[
\text{start_reached} \leftarrow \text{start}(Y), \text{inPath}(X, Y).
\]

\(^1\)The encoding for the Hamiltonian cycle contains red lines along with green ones.
Example: Hamiltonian Path/Cycle (cont’d)

\[ P_D = \{ \text{node}(a), \text{node}(b), \]
\[ \quad \text{node}(c), \text{node}(d), \]
\[ \quad \text{edge}(a, b), \text{edge}(a, c), \]
\[ \quad \text{edge}(c, b), \text{edge}(b, c), \]
\[ \quad \text{edge}(b, d), \text{edge}(d, c), \]
\[ \quad \text{edge}(d, a), \text{edge}(a, d), \]
\[ \quad \text{start}(a) \} \]
Example: Hamiltonian Path/Cycle (cont’d)

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

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

\( P_D: \)

\[
\text{member}(\text{sam}, \text{cs}). \quad \text{member}(\text{bob}, \text{cs}). \quad \text{member}(\text{tom}, \text{cs}).
\]
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

$P_D$:

\begin{align*}
\text{member}(\text{sam}, cs). & \quad \text{member}(\text{bob}, cs). & \quad \text{member}(\text{tom}, cs). \\
\text{course}(\text{java}, cs). & \quad \text{course}(\text{ai}, cs). & \quad \text{course}(c, cs). \\
\text{course}(\text{logic}, cs). & \quad & \\
\end{align*}
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
  $\text{teach}(X, Y) \leftarrow \text{member}(X, \text{cs}), \text{course}(Y, \text{cs}),$
  $\text{likes}(X, Y), \text{not} \; \neg \text{teach}(X, Y)$.

% determine when a course should not be assigned to a teacher
$\neg \text{teach}(X, Y) \leftarrow \text{member}(X, \text{cs}), \text{course}(Y, \text{cs}),$
  $\text{teach}(X_1, Y), X_1 \neq X$. 

Example: Course Assignment (cont’d)

Problem specification $P_{PS}$:

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

\[
teach(X, Y) \leftarrow member(X, cs), course(Y, cs), \\
likes(X, Y), not \neg teach(X, Y).
\]

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

\[
\neg teach(X, Y) \leftarrow member(X, cs), course(Y, cs), \\
teach(X_1, Y), X_1 \neq X.
\]

% check each cs member has some course

\[
\text{has\_course}(X) \leftarrow member(X, cs), teach(X, Y). \\
\neg member(X, cs), not \text{has\_course}(X).
\]
Example: Course Assignment (cont’d)

Problem specification $P_{PS}$:

% assign a course to a teacher who likes it, by default
$teach(X, Y) \leftarrow member(X, cs), course(Y, cs),$
$likes(X, Y), not \neg teach(X, Y).$

% determine when a course should not be assigned to a teacher
$\neg teach(X, Y) \leftarrow member(X, cs), course(Y, cs),$
$teach(X_1, Y), X_1 \neq X.$

% 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
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

\[
\text{sel}(\vec{X}, \vec{Y}) \leftarrow p(\vec{X}, \vec{Y}), \text{not } \neg \text{sel}(\vec{X}, \vec{Y}).
\]
\[
\neg \text{sel}(\vec{X}, \vec{Y}) \leftarrow p(\vec{X}, \vec{Y}), \text{sel}(\vec{X}, \vec{Z}), Y_i \neq Z_i. \quad i = 1, \ldots, m
\]

where $\vec{Z} = Z_1, \ldots, Z_m$. 
Selecting One Element from a Set (cont’d)

Example: Course assignment

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

\[
p(X, Y) \leftarrow \text{member}(Y, cs), \text{course}(X, cs), \text{likes}(Y, X)
\]

Select one element from a set: Disjunctive rule encoding

\[
\text{sel}(\vec{X}, \vec{Y}) \leftarrow p(\vec{X}, \vec{Y}), \text{not} \ - \text{sel}(\vec{X}, \vec{Y}).
\]

\[
-\text{sel}(\vec{X}, \vec{Y}) \lor -\text{sel}(\vec{X}, \vec{Z}) \leftarrow p(\vec{X}, \vec{Y}), p(\vec{X}, \vec{Z}), X_i \neq Z_i. \quad i = 1, \ldots, m
\]
Selecting One Element from a Set (cont’d)

Example: Course assignment

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

\[
p(X, Y) \leftarrow \text{member}(Y, cs), \text{course}(X, cs), \text{likes}(Y, X)
\]

Select one element from a set: Disjunctive rule encoding

\[
\text{sel}(\vec{X}, \vec{Y}) \leftarrow p(\vec{X}, \vec{Y}), \neg \text{sel}(\vec{X}, \vec{Y}).
\]

\[
\neg \text{sel}(\vec{X}, \vec{Y}) \lor \neg \text{sel}(\vec{X}, \vec{Z}) \leftarrow p(\vec{X}, \vec{Y}), p(\vec{X}, \vec{Z}), X_i \neq Z_i. \quad i = 1, \ldots, m
\]

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

\[
1\{\text{sel}(\vec{X}, \vec{Y}) : p(\vec{X}, \vec{Y})\}1 \leftarrow p(\vec{X}, \vec{Z})
\]
Use of Double Negation

Defining a predicate \( p \) in terms of its negation \( \neg p \)

Greatest Common Divisor — Euclid-style

% base case
\[
\text{gcd}(X, X, X) \leftarrow \text{int}(X), X > 1.
\]

% subtract smaller from larger number
\[
\text{gcd}(D, X, Y) \leftarrow X < Y, \text{gcd}(D, X, Y_1), Y = Y_1 + X.
\]
\[
\text{gcd}(D, X, Y) \leftarrow X > Y, \text{gcd}(D, X_1, Y), X = X_1 + Y.
\]

This is not easy to come up with and needs more care in Prolog.
Greatest Common Divisor — ASP-style

% Declare when $D$ divides a number $N$.
$\text{divisor}(D, N) \leftarrow \text{int}(D), \text{int}(N), \text{int}(M), N = D \times M$.

% Declare common divisors
$\text{cd}(T, N_1, N_2) \leftarrow \text{divisor}(T, N_1), \text{divisor}(T, N_2)$.

% Single out non-maximal common divisors $T$
$-\text{gcd}(T, N_1, N_2) \leftarrow \text{cd}(T, N_1, N_2), \text{cd}(T_1, N_1, N_2), T < T_1$.

% Apply double negation: take non non-maximal divisor
$\text{gcd}(T, N_1, N_2) \leftarrow \text{cd}(T, N_1, N_2), \text{not} - \text{gcd}(T, N_1, N_2)$.
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 \( y \)
  \( \iff \) 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 $y$
  $\iff$ 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):
  \[ \leftarrow \text{node}(X), \text{not reached}(X). \]

- Check that all nodes of the graph are reached via the selected edges (\( \text{inPath}(X, Y) \)) by iteration (\( S = \text{nodes of the graph} \))

- Use
  - \textit{all_reached_uppto(\_)}
  - \textit{all_reached}

- Supply in the input an enumeration of the nodes via
  \textit{first(\_), succ(\_, \_), last(\_)}
  - Alternative: build the enumeration \textit{dynamically} in the program, using e.g. string comparison.
Example: Hamiltonian Path 2 (cont’d)

\[
\begin{align*}
\textit{Guess} & : \\
\text{inPath}(X, Y) \lor \text{outPath}(X, Y) & \leftarrow \text{edge}(X, Y). \\
\leftarrow & \text{inPath}(X, Y), \text{inPath}(X, Y_1), Y \neq Y_1. \\
\leftarrow & \text{inPath}(X, Y), \text{inPath}(X_1, Y), X \neq X_1. \\
\text{Check} & : \\
\text{reached}(X) & \leftarrow \text{start}(X). \\
\text{reached}(X) & \leftarrow \text{reached}(Y), \text{inPath}(Y, X). \\
\text{Auxiliary Predicates} & : \\
\text{all_reached_upto}(X) & \leftarrow \text{first}(X), \text{reached}(X). \\
\text{all_reached_upto}(X) & \leftarrow \\
& \text{all_reached_upto}(Y), \text{succ}(Y, X), \text{reached}(X). \\
\text{all_reached} & \leftarrow \text{last}(X), \text{all_reached_upto}(X). \\
\end{align*}
\]

\[\text{reached} = \text{nodes}\]
Example: Hamiltonian Path 2 (cont’d)

\[ P_D = \{ \text{edge}(a, b), \text{edge}(a, c), \]
\[ \text{edge}(c, b), \text{edge}(b, c), \]
\[ \text{edge}(b, d), \text{edge}(d, c), \]
\[ \text{edge}(d, a), \text{edge}(a, d), \]
\[ \text{first}(a), \text{succ}(a, b), \]
\[ \text{succ}(b, c), \text{succ}(c, d), \]
\[ \text{last}(d), \text{start}(a) \} \]
Example: Hamiltonian Path 2 (cont’d)

$$P_D = \{\text{edge}(a, b), \text{edge}(a, c), \text{edge}(c, b), \text{edge}(b, c), \text{edge}(b, d), \text{edge}(d, c), \text{edge}(d, a), \text{edge}(a, d), \text{first}(a), \text{succ}(a, b), \text{succ}(b, c), \text{succ}(c, d), \text{last}(d), \text{start}(a)\}$$
Example: Hamiltonian Path 2 (cont’d)

Some path guesses not reaching all nodes from $a$:

- $all\_reached\_upto(c)$
  - $a \rightarrow b \rightarrow c \rightarrow d$
  - $a \rightarrow d \rightarrow c \rightarrow b$

- $all\_reached\_upto(a)$
  - $d \rightarrow a \rightarrow b \rightarrow c$
  - $d \rightarrow c \rightarrow a \rightarrow b$

- $all\_reached\_upto(b)$
  - $a \rightarrow b \rightarrow d \rightarrow c$
  - $a \rightarrow b \rightarrow c \rightarrow d$
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
(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**  potassco.sourceforge.net/#clasp/
**CMODELS**  www.cs.utexas.edu/users/tag/cmodels/
**DLV**  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 + CLASP\(D\), 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 + CLASP\textsuperscript{D}, CLINGO, CLINGCON etc. (http://potassco.sourceforge.net/)
2 + DLV\textsuperscript{HEX}, DLV\textsuperscript{DB}, DLT, DLV-COMPLEX, ONTO-DLV etc.

- Many ASP solvers are available (mostly function-free programs)
- \texttt{clasp} was first ASP solver competitive to top SAT solvers
- another state-of-the-art solver is \texttt{dlv}
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 $\text{grnd}(P)$ that has the same models
2-Level Architecture

1. **Intelligent grounding**
   Given a program $P$, generate a (subset) of $\text{grnd}(P)$ 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 $\text{grnd}(P)$ 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 co-NP-complete
Intelligent Grounding

- Grounding is a hard problem

\[ \text{bit}(0). \text{bit}(1). \]
\[ p(X_1, \ldots, X_n) \leftarrow \text{bit}(X_1), \ldots, \text{bit}(X_n). \]
Intelligent Grounding

- Grounding is a hard problem

  \[ \text{bit}(0). \ \text{bit}(1). \tag{2} \]

  \[ p(X_1, \ldots, X_n) \leftarrow \text{bit}(X_1), \ldots, \text{bit}(X_n). \tag{3} \]

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

- Grounding is a hard problem

\[
b(0). \ b(1). \\
p(X_1, \ldots, X_n) \leftarrow b(X_1), \ldots, b(X_n). \tag{3}
\]

- 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
  
  \[\text{bit}(0). \quad \text{bit}(1).\]
  
  \[p(X_1, \ldots, X_n) \leftarrow \text{bit}(X_1), \ldots, \text{bit}(X_n).\]

- 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

\[ \text{bit}(0). \text{bit}(1). \]
\[ p(X_1, ..., X_n) \leftarrow \text{bit}(X_1), ..., \text{bit}(X_n). \]

- 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);
  - lparses (smodels), gringo (clasp)

- Special techniques used:
Intelligent Grounding

- Grounding is a hard problem

\[ \text{bit}(0). \quad \text{bit}(1). \]
\[ p(X_1, \ldots, X_n) \leftarrow \text{bit}(X_1), \ldots, \text{bit}(X_n). \]

- 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
  \[ \text{bit}(0). \text{bit}(1). \]
  \[ p(X_1, ..., X_n) \leftarrow \text{bit}(X_1), ..., \text{bit}(X_n). \]

- 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 (smoodels), 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 (smoodels)
Intelligent Grounding

- Grounding is a hard problem

\[
\text{bit}(0). \quad \text{bit}(1).
\]

\[
p(X_1, \ldots, X_n) \leftarrow \text{bit}(X_1), \ldots, \text{bit}(X_n).
\]

- 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, \ldots
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), …

```plaintext
a:~ not b.
b:~ not a.
c:~ not c, a.
```

```
\[ \begin{array}{c}
b \\
\hline not b \\
\hline a \\
\hline not a \\
\hline c \\
\hline not c \\
\hline b \\
\hline not b \\
\end{array} \]
```

- 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)
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), ...

\[
\begin{align*}
  a & : \neg b. \\
  b & : \neg a. \\
  c & : \neg c, a. \\
\end{align*}
\]

- 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), ... 

\[
\begin{align*}
  a &: \neg b. \\
  b &: \neg a. \\
  c &: \neg c, a.
\end{align*}
\]

- 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
Paula-Andra Busoniu, Johannes Oetsch, Jörg Pührer, Peter Skocovsky, and Hans Tompits.

Sealion: An eclipse-based IDE for answer-set programming with advanced debugging support.


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

Declarative problem-solving using the DLV system.


Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner.

Answer set programming: A primer.

Onofrio Febbraro, Kristian Reale, and Francesco Ricca.

ASPIDE: integrated development environment for answer set programming.

Martin Gebser and Torsten Schaub.

Modeling and language extensions.

Tomi Janhunen and Ilkka Niemelä.

The answer set programming paradigm.
Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello.

The DLV System for Knowledge Representation and Reasoning.


Vladimir Lifschitz.

Answer Set Programming and Plan Generation.