Knowledge Representation for the Semantic Web

Lecture 8: Answer Set Programming III

Daria Stepanova
partially based on slides by Thomas Eiter

D5: Databases and Information Systems
Max Planck Institute for Informatics

WS 2017/18
Unit Outline

The DLV System and its Features

Weak Constraints

Aggregates

DLV Usage: Examples

Overview: DLV-Extensions
The DLV System
The DLV System: Introduction

http://www.dlvsystem.com/

- DLV is a premier disjunctive answer set solver
- Based on strong theoretical foundations
- Incorporates a lot of database technology
- Features non-monotonic negation and disjunction
- Rich program syntax (⇒ high expressiveness)
- Front-ends for specific problems (diagnosis, planning, etc.).
- Many extensions
  - DLVHEX, DLV$^{DB}$, DLT, DLV-Complex, DL-programs, OntoDLV, ...
- Industrial applications
  - Exeura Srl  www.exeura.it/
Features of DLV

- Language: logic programs admitting
  - disjunctions in rule heads,
  - default negation,
  - strong (classical) negation.

\(^1\)with the release of DLV 2010-10-14, function terms have been introduced.
Features of DLV

- **Language:** logic programs admitting
  - disjunctions in rule heads,
  - default negation,
  - strong (classical) negation.

- **Additionally:**
  - integer, arithmetic, and comparison built-ins,
  - integrity constraints,
  - weak constraints,
  - aggregate functions,
  - function symbols;\(^1\)
  - support for **brave & cautious** reasoning.
  - + further

---

\(^1\)with the release of DLV 2010-10-14, function terms have been introduced.
Frontends

• Besides the answer set semantics core, DLV offers front-ends for particular KR tasks:
  • diagnosis
  • inheritance
  • knowledge-based planning ($\mathcal{K}$ language)

• Also:
  • front-end to SQL3
  • weak constraints with weights and layers
  • aggregate functions
Using DLV

- DLV is command-line oriented
- 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, i.e., answer set
- Detailed documentation at http://www.dlvsystem.com
DLV Syntax

- **Rules:**

\[ a_1 \lor \cdots \lor a_n :\neg b_1, \ldots, \neg b_k, \neg b_{k+1}, \ldots, \neg b_m. \]

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.
DLV Syntax

- **Rules:**

  \[ a_1 \lor \cdots \lor a_n :\neg b_1, \ldots, b_k, \neg b_{k+1}, \ldots, \neg b_m. \]

  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:**

  \[ :\neg b_1, \ldots, b_k, \neg b_{k+1}, \ldots, \neg b_m. \]

  Can be regarded as rules with an empty (false) head.
**DLV Syntax**

- **Rules:**
  
  \[ a_1 \lor \cdots \lor a_n :- b_1, \ldots, b_k, \lnot b_{k+1}, \ldots, \lnot b_m. \]

  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:**
  
  \[ :- b_1, \ldots, b_k, \lnot b_{k+1}, \ldots, \lnot b_m. \]

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

- **Queries:**
  
  \[ b_1, \ldots, b_k, \lnot b_{k+1}, \ldots, \lnot b_m? \]

  Support for query answering besides model computation (satisfied in at least one / in all answer sets, called brave / cautious reasoning)
Rule Safety

Each variable occurring in a rule (resp., constraint) in

- the head,
- a default literal (\texttt{not} \texttt{b}), or
- a built-in comparison predicate,

must occur in at least one non-comparison \texttt{not}-free literal in the body.
Rule Safety

Each variable occurring in a rule (resp., constraint) in
- the head,
- a default literal (not \texttt{b}), or
- a built-in comparison predicate,
must occur in at least one non-comparison not-free literal in the body.

Example:

\begin{verbatim}
   a(X) :- not b(X), c(X).
   a(X) :- X > Y, node(X), node(Y).

   a(X) v -a(X).
   a(X) :- not b(X).
   :- X <= Y, node(X).
\end{verbatim}
Rule Safety

Each variable occurring in a rule (resp., constraint) in
- the head,
- a default literal (**not b**), or
- a built-in comparison predicate,

must occur in at least one non-comparison **not**-free literal in the body.

Example:

Safe!

\[
\begin{align*}
    a(X) & :- \text{not } b(X), \ c(X) . \\
    a(X) & :- X > Y, \ \text{node}(X), \ \text{node}(Y).
\end{align*}
\]

\[
\begin{align*}
    a(X) & \lor \neg a(X) . \\
    a(X) & :- \text{not } b(X) . \\
    :- X \leq Y, \ \text{node}(X) .
\end{align*}
\]
Rule Safety

Each variable occurring in a rule (resp., constraint) in
- the head,
- a default literal (\texttt{not }\texttt{b}), or
- a built-in comparison predicate,

must occur in at least one non-comparison \texttt{not}-free literal in the body.

Example:

Safe!

\begin{verbatim}
  a(X) :- not b(X), c(X).
  a(X) :- X > Y, node(X), node(Y).
\end{verbatim}

Unsafe!

\begin{verbatim}
  a(X) v -a(X).
  a(X) :- not b(X).
  :- X <= Y, node(X).
\end{verbatim}
**Built-in Predicates**

- **Comparison predicates** (for integers and strings):
  
  `<, >, <=, >=, =, !=`
**Built-in Predicates**

- **Comparison predicates** (for integers and strings):
  
  `<, >, <=, >=, =, !=`

- **Arithmetic predicates**:
  
  `#int, #succ, +, *`

  - `#int(X)`: $X$ is a known integer ($1 \leq X \leq N$).
  - `#succ(X, Y)`: $Y$ is successor of $X$, i.e., $Y = X + 1$.
  - `+(X, Y, Z)`: $Z = X + Y$. (both variants are possible)
  - `*(X, Y, Z)`: $Z = X \times Y$.  

Just auxiliary predicates. An upper bound for integers has to be specified when DLV is invoked.
Built-in Predicates

- **Comparison predicates** (for integers and strings):
  
  \(<, \>, \leq, \geq, =, \neq\)

- **Arithmetic predicates**:
  
  \(#\text{int}, \#\text{succ}, +, \ast\)

\[
\begin{align*}
\#\text{int}(X): & \quad X \text{ is a known integer } (1 \leq X \leq N). \\
\#\text{succ}(X, Y): & \quad Y \text{ is successor of } X, \text{ i.e., } Y = X + 1.
\end{align*}
\]

\[
\begin{align*}
+(X, Y, Z): & \quad Z = X + Y. \text{ (both variants are possible)} \\
\ast(X, Y, Z): & \quad Z = X \ast Y.
\end{align*}
\]

- Just auxiliary predicates. An upper bound for integers has to be specified when DLV is invoked.
Example: Fibonacci Numbers

- Except for first two numbers, each value is defined as the sum of the previous two.
Example: Fibonacci Numbers

- Except for first two numbers, each value is defined as the sum of the previous two.

Encoding:

```
fib0(1,1).
fib0(2,1).
fib(N,X) :- fib0(N,X).
```

```
\% F_{N+2} = F_N + F_{N+1}
fib(N,X) :- fib(N1,Y1), fib(N2,Y2),
\quad N=N2+2, N=N1+1, X=Y1+Y2.
```

An upper bound for integers has to be specified when \texttt{dlv} is invoked.
Linear Ordering, Successor

Example: Employees

Input: Employees and their salaries, represented by $\text{empl}(_,_)$

Problem: Compute linear ordering and successor relation for employees

Solve problem using projection and double negation!

$\text{Order employees by id}$

$\text{prec}(X,Y) :\text{- empl}(X,_)$, $\text{empl}(Y,_)$, $X < Y$.

$\text{Define successor}$

$\text{- succ}(X,Y) :\text{- prec}(X,Z)$, $\text{prec}(Z,Y)$.

$\text{succ}(X,Y) :\text{prec}(X,Y)$, $\text{not} \text{- succ}(X,Y)$. 
Linear Ordering, Successor

**Example: Employees**

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

**Problem:** Compute linear ordering and successor relation for employees

Solve problem using projection and double negation!
Linear Ordering, Successor

Example: Employees

Input: Employees and their salaries, represented by \texttt{empl(\_, \_)}

Problem: Compute linear ordering and successor relation for employees

Solve problem using projection and double negation!

% Order employees by id

\[
\text{prec}(X, Y) :\neg \text{empl}(X, \_), \text{empl}(Y, \_), X < Y.
\]

% Define successor

\[
\neg \text{succ}(X, Y) :\neg \text{prec}(X, Z), \text{prec}(Z, Y).
\]

\[
\text{succ}(X, Y) :\neg \text{prec}(X, Y), \text{not} \neg \text{succ}(X, Y).
\]
Smallest, Largest in a Linear Ordering

Example: Employees

Problem: Determine employee with smallest (resp., largest) id
Smallest, Largest in a Linear Ordering

Example: Employees

Problem: Determine employee with smallest (resp., largest) id

- Computing smallest and largest elements in a linear ordering works accordingly:

\[
\begin{align*}
- \text{first}(X) & : - \text{succ}(Y, X). \\
\text{first}(X) & : - \text{empl}(X, -), \text{not} \ - \text{first}(X). \\
- \text{last}(X) & : - \text{succ}(X, Y). \\
\text{last}(X) & : - \text{empl}(X, -), \text{not} \ - \text{last}(X).
\end{align*}
\]

Exercise: determine maximal (resp. minimal) salary of employees
Counting and Sum

How about counting or computing sums?

Example: Employees (cont’d)

Problem: Compute the sum of salaries of the employees
Counting and Sum

How about counting or computing sums?

Example: Employees (cont’d)

Problem: Compute the sum of salaries of the employees

- Recursion is needed:

  \[
  \text{partialSum}(X,S) :- \text{first}(X), \text{empl}(X,S).
  \]

  \[
  \text{partialSum}(Y,S) :- \text{succ}(X,Y), \text{partialSum}(X,S1),
  \quad \text{empl}(Y,S2), \ S = S1 + S2.
  \]

  \[
  \text{sum}(S) :- \text{last}(X), \text{partialSum}(X,S).
  \]
**Weak Constraints**

- Allow to formalize *optimization problems* in an easy and natural way.
- Integrity constraints vs. weak constraints:
  - integrity constraints “kill” unwanted models;
  - weak constraints express desiderata to satisfy if possible.
Weak Constraints

• Allow to formalize **optimization problems** in an easy and natural way.

• Integrity constraints vs. weak constraints:
  • integrity constraints “kill” unwanted models;
  • weak constraints express desiderata to satisfy if possible.

• Syntax (DLV):

  \[ \sim b_1, \ldots, b_k, \text{ not } b_{k+1}, \ldots, \text{ not } b_m \cdot \text{ [Weight: Level]} \]

  where

  • all \( b_i \) are atoms (resp. “classical” literals)
  • \textit{Weight}, \textit{Level} are numbers (or variables occurring in some \( b_i, i \leq k \), that instantiate to numbers)
Weak Constraints

- Allow to formalize **optimization problems** in an easy and natural way.
- Integrity constraints vs. weak constraints:
  - integrity constraints “kill” unwanted models;
  - weak constraints express desiderata to satisfy if possible.
- Syntax (DLV):
  \[ \sim b_1, \ldots, b_k, \; \text{not } b_{k+1}, \ldots, \; \text{not } b_m \; \square \; \text{Weight} : \text{Level} \]
  where
  - all \( b_i \) are atoms (resp. “classical” literals)
  - \text{Weight, Level} are numbers (or variables occurring in some \( b_i, i \leq k \), that instantiate to numbers)

- **Informally:** for \((P, WC)\), where \( P \) is a program and \( WC \) is a set of weak constraints, each \( M \in AS(P) \) with least violation of \( WC \) is an answer set (**best model**), where \( AS(P) = \) set of answer sets of \( P \).
Weak Constraints: Semantics for \((P, WC)\)

Semantics via aggregated violation cost \((WC = \{wc_1, \ldots, wc_n\})\):

\[ wc : \sim b_1, \ldots, b_k, \not b_{k+1}, \ldots, \not b_m. \quad [\text{Weight: Level}] \]

- as usual, consider the grounding \(\text{grnd}(wc)\) of \(wc\)
- Interpretation \(I\) violates a ground \(wc\) \((I \not\models wc)\), if \(\{b_1, \ldots, b_k\} \subseteq I\) and \(I \cap \{b_{k+1}, \ldots, b_m\} = \emptyset\)
Weak Constraints: Semantics for $\langle P, WC \rangle$

Semantics via aggregated violation cost ($WC = \{wc_1, \ldots, wc_n\}$):

$$wc: \, \sim b_1, \ldots, b_k, \, \text{not } b_{k+1}, \ldots, \, \text{not } b_m. \quad \text{[Weight: Level]}$$

- as usual, consider the grounding $\text{grnd}(wc)$ of $wc$
- Interpretation $I$ violates a ground $wc$ ($I \not\models wc$), if $\{b_1, \ldots, b_k\} \subseteq I$ and $I \cap \{b_{k+1}, \ldots, b_m\} = \emptyset$
- The cost of $I$ at level $\ell$ is
  $$c(I, \ell) = \sum_{i=1}^{n} \sum_{(\theta, w) \in \mathcal{V}_i(I, \ell)} w,$$
  where
  $$\mathcal{V}_i(I, \ell) = \{ (\theta, w) \mid wc_i \theta = \sim B. \, [w, \ell] \in \text{grnd}(wc_i), I \not\models wc_i \theta \}$$
Weak Constraints: Semantics for \((P, WC)\)

Semantics via aggregated violation cost \((WC = \{wc_1, \ldots, wc_n\})\):

\[ wc : \sim b_1, \ldots, b_k, \ not b_{k+1}, \ldots, \ not b_m . \]  

[Weight: Level]

- as usual, consider the grounding \(\text{grnd}(wc)\) of \(wc\)
- Interpretation \(I\) violates a ground \(wc\) \((I \not\models wc)\), if \(\{b_1, \ldots, b_k\} \subseteq I\) and \(I \cap \{b_{k+1}, \ldots, b_m\} = \emptyset\)
- The cost of \(I\) at level \(\ell\) is
  \[ c(I, \ell) = \sum_{i=1}^{n} \sum_{(\theta, w) \in \mathcal{V}_i(I, \ell)} w, \]
  where
  \[ \mathcal{V}_i(I, \ell) = \{(\theta, w) \mid wc_i \theta = :\sim B. \ [w, \ell] \in \text{grnd}(wc_i), I \not\models wc_i \theta\} \]
- \(I\) is safe, if each \(c(I, \ell)\) is well-defined (all \(w\)'s are numbers)
Weak Constraints: Semantics for \((P, WC)\)

Semantics via aggregated violation cost \((WC = \{wc_1, \ldots, wc_n\})\):

\[
wc : \sim b_1, \ldots, b_k, \ not b_{k+1}, \ldots, \ not b_m . \quad \text{[Weight : Level]}
\]

- as usual, consider the grounding \(grnd(wc)\) of \(wc\)
- Interpretation \(I\) violates a ground \(wc\) \((I \not\models wc)\), if \(\{b_1, \ldots, b_k\} \subseteq I\) and \(I \cap \{b_{k+1}, \ldots, b_m\} = \emptyset\)
- The cost of \(I\) at level \(\ell\) is

\[
c(I, \ell) = \sum_{i=1}^{n} \sum_{(\theta, w) \in V_i(I, \ell)} w,
\]

where

\[
V_i(I, \ell) = \{(\theta, w) \mid wc_i \theta = :\sim B. \ [w, \ell] \in grnd(wc_i), I \not\models wc_i \theta\}
\]

- \(I\) is safe, if each \(c(I, \ell)\) is well-defined (all \(w\)'s are numbers)
- a safe \(M \in AS(P)\) dominates a safe \(M' \in AS(P)\), if \(c(M, \ell) < c(M', \ell)\) for some \(\ell\) and \(c(M, \ell') = c(M', \ell')\) for all \(\ell' > \ell\)
Weak Constraints: Semantics for $(P, WC)$

Semantics via aggregated violation cost ($WC = \{wc_1, \ldots, wc_n\}$):

$$wc : \sim b_1, \ldots, b_k, \ not b_{k+1}, \ldots, \ not b_m. \ \text{[Weight: Level]}$$

- as usual, consider the grounding $\text{grnd}(wc)$ of $wc$
- Interpretation $I$ violates a ground $wc$ ($I \not\models wc$), if $\{b_1, \ldots, b_k\} \subseteq I$ and $I \cap \{b_{k+1}, \ldots, b_m\} = \emptyset$
- The cost of $I$ at level $\ell$ is
  $$c(I, \ell) = \sum_{i=1}^{n} \sum_{(\theta, w) \in V_i(I, \ell)} w,$$
  where
  $$V_i(I, \ell) = \{(\theta, w) \mid wc_i \theta = \sim B. [w, \ell] \in \text{grnd}(wc_i), I \not\models wc_i \theta\}$$
- $I$ is safe, if each $c(I, \ell)$ is well-defined (all $w$’s are numbers)
- a safe $M \in AS(P)$ dominates a safe $M' \in AS(P)$, if $c(M, \ell) < c(M', \ell)$ for some $\ell$ and $c(M, \ell') = c(M', \ell')$ for all $\ell' > \ell$
- a safe $M \in AS(P)$ is best (optimal), if no $M' \in AS(P)$ dominates $M$
Weak Constraints: Examples

**Example:** Default values for weights and levels

```
a v b.   c :- b.
~ a.
~ a.
~ b.
~ c.
```
Weak Constraints: Examples

**Example:** Default values for weights and levels

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

Best model: \(a\)
Cost ([Weight:Level]): \([1:1]\)

Answer set \(\{b, c\}\) is discarded because it violates two weak constraints!
Example: Weights vs levels

Weights:

\( a \lor b \cdot \)
\( \neg a. [1:] \)
\( \neg a. [1:] \)
\( \neg b. [2:] \)
Example: Weights vs levels

Weights:

\[ a \lor b. \]
\[ \neg a. [1:] \]
\[ \neg a. [1:] \]
\[ \neg b. [2:] \]

Best model: \( b \)
Cost ([Weight:Level]): \( <[2:1]> \)

Best model: \( a \)
Cost ([Weight:Level]): \( <[2:1]> \)

Note: \( WC = \{ wc_1, wc_2, wc_3 \}, \)
\[ wc_1 = \neg a.[1:], \]
\[ wc_2 = \neg a.[1:], \]
\[ wc_3 = \neg b.[2:] \]
**Example: Weights vs levels**

**Weights:**

\[
\begin{align*}
\text{a} & \lor \text{b}.\\
\leadsto & \sim \text{a. [1:]}\\
\leadsto & \sim \text{a. [1:]}\\
\leadsto & \sim \text{b. [2:]} \\
\end{align*}
\]

Best model: \( \text{b} \)

Cost ([Weight:Level]): \(<[2:1]>\)

**Levels:**

\[
\begin{align*}
\text{a} & \lor \text{b1} \lor \text{b2}.\\
\leadsto & \sim \text{a. [:1]}\\
\leadsto & \sim \text{b1. [:2]}\\
\leadsto & \sim \text{b2. [:2]} \\
\end{align*}
\]

Best model: \( \text{a} \)

Cost ([Weight:Level]): \(<[2:1]>\)

**Note:** \( WC = \{wc_1, wc_2, wc_3\}, \)

\[
\begin{align*}
wc_1 & = \sim \text{a. [1:]} \\
wc_2 & = \sim \text{a. [1:]} \\
wc_3 & = \sim \text{b. [2:]} \\
\end{align*}
\]
Weak Constraints: Examples/2

Example: Weights vs levels

Weights:

\[ a \lor b. \]
\[ \neg a. \ [1:] \]
\[ \neg a. \ [1:] \]
\[ \neg b. \ [2:] \]

Best model: b
Cost ([Weight:Level]): \(<[2:1]>\)

Best model: a
Cost ([Weight:Level]): \(<[2:1]>\)

Levels:

\[ a \lor b1 \lor b2. \]
\[ \neg a. \ [::1] \]
\[ \neg b1. \ [::2] \]
\[ \neg b2. \ [::2] \]

Best model: a
Cost ([Weight:Level]): \(<[1:1],[0:2]>\)

Note: \( WC = \{wc_1, wc_2, wc_3\}, \)
\[ wc_1 =:\neg a. [1:], \]
\[ wc_2 =:\neg a. [1:], \]
\[ wc_3 =:\neg b. [2:] \]
Weak Constraints with Levels

Levels express the relative importance of the requirements.

**Example:** Divide employees in two project groups \( p_1 \) and \( p_2 \)

1. Skills of group members should be different
2. Persons in the same group should not be married to each other
3. Members of a group should possibly know each other

Requirement (3) is less important than (1) and (2)
### Weak Constraints with Levels

Levels express the relative importance of the requirements.

**Example:** Divide employees in two project groups $p_1$ and $p_2$

1. Skills of group members should be different
2. Persons in the same group should not be married to each other
3. Members of a group should possibly know each other

Requirement (3) is less important than (1) and (2)

```prolog
assign(X,p1) v assign(X,p2) :- employee(X).

\[\sim\] assign(X,P), assign(Y,P), X!=Y, same_skill(X,Y). [:2]
\[\sim\] assign(X,P), assign(Y,P), X!=Y, married(X,Y). [:2]
\[\sim\] assign(X,P), assign(Y,P), X!=Y, not know(X,Y). [:1]
```
Weak Constraints with Weights

- A single weak constraint in some layer $n$ is more important than all weak constraints in lower layers ($n - 1$, $n - 2$, ...) together!

- Weak constraints are weighted to make finer distinctions among elements of the same priority: $\sim B1. [3.5:1] \sim B2. [4.6:1]$

- The weights of violated weak constraints are summed up for each layer.
Weak Constraints with Weights

• A single weak constraint in some layer $n$ is more important than all weak constraints in lower layers $(n-1, n-2, \ldots)$ together!

• Weak constraints are weighted to make finer distinctions among elements of the same priority: $\sim B1. [3.5:1] \sim B2. [4.6:1]$

• The weights of violated weak constraints are summed up for each layer.

Example: High School Time Tabling Problem

Structural Requirements $>$ Pedagogical Requirements $>$ Personal Wishes.
Example: Traveling Salesperson (TSP)

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

Problem: find a cheapest roundtrip beginning at the starting node
**Example: Traveling Salesperson (TSP)**

**Input:** a directed graph represented by `node(_),` straight connections `edge(_,_,_,_)` and a starting node `start(_).`

**Problem:** find a cheapest roundtrip beginning at the starting node

\[
\text{inPath}(X,Y) \lor \text{outPath}(X,Y) :\neg \text{edge}(X,Y) .
\]

\[
\begin{align*}
\text{Guess} \\
\text{:-inPath}(X,Y), \text{inPath}(X,Y1), Y \neq Y1. \\
\text{:-inPath}(X,Y), \text{inPath}(X1,Y), X \neq X1. \\
\text{:-node}(X), \text{notreached}(X). \\
\text{:-not start_reached}.^2 \\
\text{reached}(X):-\text{start}(X). \\
\text{reached}(X):-\text{reached}(Y), \text{inPath}(Y,X). \\
\text{start_reached} :- \text{start}(Y), \text{inPath}(X,Y).
\end{align*}
\]

\[
^2\text{This line is added, since the trip must be round.}
\]
Example: Traveling Salesperson (TSP)

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

Problem: find a cheapest roundtrip beginning at the starting node

\[
\text{inPath}(X,Y,C) \lor \text{outPath}(X,Y,C) :- \text{edge}(X,Y,C). \]

\[
\begin{align*}
\text{inPath}(X,Y,C), \text{inPath}(X,Y1,C1), Y & \neq Y1. \\
\text{inPath}(X,Y,C), \text{inPath}(X1,Y,C1), X & \neq X1. \\
\text{node}(X), \text{notreached}(X). \\
\text{not start_reached}.^2
\end{align*}
\]

\[
\text{reached}(X) :- \text{start}(X). \\
\text{reached}(X) :- \text{reached}(Y), \text{inPath}(Y,X,C). \\
\text{start_reached} :- \text{start}(Y), \text{inPath}(X,Y,C).
\]

\[
\sim \text{inPath}(X,Y,C).[C:1]
\]

\[^2\text{This line is added, since the trip must be round.}\]
Example: Minimum Spanning Tree

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

Problem: Find a minimum spanning tree with root at the starting node

\[
\text{inTree}(X,Y) \lor \text{outTree}(X,Y) :\neg \text{edge}(X,Y). \quad \text{Guess}
\]

\[
\begin{align*}
: & \neg \text{inTree}(X,Y), \text{start}(Y). \\
: & \neg \text{inTree}(X,Y), \text{inTree}(X1,Y), X \neq X1. \\
: & \text{node}(X), \neg \text{reached}(X).
\end{align*}
\]

\[
\text{reached}(X): \neg \text{start}(X). \\
\text{reached}(X): \neg \text{reached}(Y), \text{inTree}(Y,X).
\]

\[
\text{Auxiliary Def.}
\]
Example: Minimum Spanning Tree

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

Problem: Find a minimum spanning tree with root at the starting node

\[
\text{inTree}(X,Y) \lor \text{outTree}(X,Y) :\neg \text{edge}(X,Y). \quad \text{Guess}
\]
\[
:\neg \text{inTree}(X,Y), \text{start}(Y).
:\neg \text{inTree}(X,Y), \text{inTree}(X1,Y), X \neq X1.
:\neg \text{node}(X), \text{not reached}(X).
\]

\[
\text{reached}(X) :\neg \text{start}(X).
\text{reached}(X) :\neg \text{reached}(Y), \text{inTree}(Y,X).
\]

Auxiliary Def.
Example: Minimum Spanning Tree

Input: A directed graph represented by `node(_,_)`, weighted edges `edge(_,_,_)` and a starting node `start(_)`.

Problem: Find a minimum spanning tree with root at the starting node

```
inTree(X,Y,C) v outTree(X,Y,C) :- edge(X,Y,C).  } Guess

:-inTree(X,Y,C), start(Y).
:-inTree(X,Y,C), inTree(X1,Y,C), X != X1.
:-node(X), not reached(X).

reached(X):-start(X).
reached(X):-reached(Y), inTree(Y,X,C).  } Auxiliary Def.

:\~ inPath(X,Y,C).[C:1]  } Optimize
```

Check

Optimize
Example: Minimum Spanning Tree (ctd.)

\[ P_D = \{ \text{node}(a), \text{node}(b), \text{node}(c), \text{node}(d), \text{edge}(a, b, 1), \text{edge}(a, c, 1), \text{edge}(c, b, 2), \text{edge}(b, c, 1), \text{edge}(b, d, 1), \text{edge}(c, d, 1), \text{start}(a) \} \]
Example: Minimum Spanning Tree (ctd.)

\[ P_D = \{ \text{node}(a), \text{node}(b), \text{node}(c), \text{node}(d), \text{edge}(a, b, 1), \text{edge}(a, c, 1) \text{edge}(c, b, 2), \text{edge}(b, c, 1) \text{edge}(b, d, 1), \text{edge}(c, d, 1) \text{start}(a) \} \]
Aggregates

- Allow arithmetic operations over a set of elements, as e.g. in SQL:
  
  ```sql
  select count(*) from empl;
  ```
Aggregates

- Allow **arithmetic operations over a set** of elements, as e.g. in SQL:
  
  ```sql
  select count(*) from empl;
  ```

- ASP provides aggregation functions **#count, #sum, #min, #max**
  
  ```prolog
  #count{Emp,Dept,Job: empl(Emp,Dept,Job)}
  ```
Aggregates

- Allow **arithmetic operations over a set** of elements, as e.g. in SQL:
  
  ```
  select count(*) from empl;
  ```

- ASP provides aggregation functions **#count, #sum, #min, #max**
  
  ```
  #count{Emp,Dept,Job: empl(Emp,Dept,Job)}
  ```

- these aggregate functions occur in aggregate atoms in rule bodies
  
  ```
  small_dept(D) :- #count{ E,D: empl(E,D,J) } < 10, dept(D)
  ```
Aggregates

- Allow arithmetic operations over a set of elements, as e.g. in SQL:
  
  select count(*) from empl;

- ASP provides aggregation functions \( \#\text{count}, \#\text{sum}, \#\text{min}, \#\text{max} \)
  
  \[
  \#\text{count}\{\text{Emp, Dept, Job}: \text{empl}(\text{Emp, Dept, Job})\}
  \]

- these aggregate functions occur in aggregate atoms in rule bodies
  
  \[
  \text{small_dept}(D) :- \#\text{count}\{\text{E, D}: \text{empl}(\text{E, D, J})\} < 10, \text{dept}(D)
  \]

- aggregates as first-class citizen: need no auxiliary computations
  
  - linear ordering, successor relation, smallest and largest element, and
  - recursion needed to count the employees
Aggregates

- Allow arithmetic operations over a set of elements, as e.g. in SQL:
  ```sql
  select count(*) from empl;
  ```

- ASP provides aggregation functions `#count`, `#sum`, `#min`, `#max`
  ```prolog
  #count{Emp,Dept,Job: empl(Emp,Dept,Job)}
  ```

- these aggregate functions occur in aggregate atoms in rule bodies
  ```prolog
  small_dept(D) :- #count{ E,D: empl(E,D,J) } < 10, dept(D)
  ```

- aggregates as first-class citizen: need no auxiliary computations
  - linear ordering, successor relation, smallest and largest element, and
  - recursion needed to count the employees

- challenging: semantics of aggregates (problem: recursion)

- we consider non-recursive aggregates, DLV (general: ASP-Core2)
## Symbolic Set

### Symbolic Set Expression

\[
\{ Vars : Conj \}
\]

where

- *Vars* is a set of variables, and
- *Conj* is a conjunction of standard literals, i.e., literals and default negated literals.
Symbolic Set

Symbolic Set Expression

\{ Vars : Conj \}

where

- \( Vars \) is a set of variables, and
- \( Conj \) is a conjunction of standard literals, i.e., literals and default negated literals.

Example: \{ S, X : empl(X, S) \}

Informal Meaning: The set of ids and salaries of all employees, i.e.,

- for a set of standard literals (an interpretation)
  \( I = \{ empl(1, 2200), empl(2, 1800) \} \),
- the symbolic set above represents a set of tuples
  \( S = \{ \langle 2200, 1 \rangle, \langle 1800, 2 \rangle \} \).
Aggregate Functions

Aggregate Function Expression

\[ f\{S\} \]

where

- \( S \) is a symbolic set, and
- \( f \) is a function among \(#\text{count}, #\text{sum}, #\text{times}, #\text{min}, #\text{max}\)
Aggregate Functions

Aggregate Function Expression

\[ f\{S\} \]

where

- \( S \) is a symbolic set, and
- \( f \) is a function among \{\#count, \#sum, \#times, \#min, \#max\}

Example: \#sum\{S, X : empl(X, S)\}

Informal Meaning: The sum of salaries of all employees.
**Aggregate Functions**

**Aggregate Function Expression**

\[ f\{S\} \]

where

- \( S \) is a symbolic set, and
- \( f \) is a function among \{\#count, \#sum, \#times, \#min, \#max\}

**Example:** \#sum\{S, X : empl(X, S)\}

**Informal Meaning:** The sum of salaries of all employees.

- \#count returns the cardinality of the symbolic set;
- the other functions apply to the multiset of the elements in the symbolic set projected to the first component.
Identical Projections

Note:

$$\#\text{sum}\{S : \text{empl}(X, S)\} \neq \#\text{sum}\{S, X : \text{empl}(X, S)\}$$

as identical projections $S$ of different elements count multiple times
Aggregate Functions, cont’d

Identical Projections

Note:

$$\#\text{sum}\{S : \text{empl}(X, S)\} \neq \#\text{sum}\{S, X : \text{empl}(X, S)\}$$

as identical projections $S$ of different elements count multiple times

for $S = \emptyset$:

- $\#\text{sum}$ returns 0
- $\#\text{times}$ returns 1
- $\#\text{min}$ and $\#\text{max}$ undefined
**Aggregate Atoms**

**Aggregate Atom Syntax**

\[ Lg <_{1} f\{S\} <_{2} Rg \]

where

- \( Lg \) and \( Ug \) are terms, called **left guard** and **right guard**, respectively,
- and \(<_{1}, <_{2}\) in \( \{=, <, \leq, >, \geq\} \);
- one of the guards can be omitted (assuming “0 \leq” and “\leq +\infty”)

---

**Example:**

\[ \#\text{sum}\{S, X: \text{empl}(X, S)\} \leq 3800 \]

**Informal Meaning:**

True if sum of salaries \( \leq 3800 \), false otherwise.

**If the argument of an aggregate function does not belong to its domain, then false and warning.**
Aggregate Atoms

Aggregate Atom Syntax

\[ Lg <_1 f\{S}\] <_2 Rg \]

where

- \( Lg \) and \( Ug \) are terms, called left guard and right guard, respectively,
- and \(<_1, <_2 \) in \( \{=, <, \leq, >, \geq\} \);
- one of the guards can be omitted (assuming “0 \( \leq \)” and “\( \leq +\infty \)”)

Example: \#\text{sum}\{S, X : \text{empl}(X, S)\} \leq 3800

Informal Meaning: True if sum of salaries \( \leq 3800 \), false otherwise.

- If the argument of an aggregate function does not belong to its domain, then false and warning.
Aggregate Atom: Common Mistakes

Let \( \text{pay}(\text{transaction}, \text{person}, \text{value}) \) represent a payment, consider: \{\text{pay}(t1, p1, 5), \text{pay}(t2, p1, 8), \text{pay}(t3, p1, 5), \text{pay}(t4, p2, 10), \text{pay}(t5, p2, 20)\}. Task: Compute the sum of payments for each person.
Aggregate Atom: Common Mistakes

Let \texttt{pay(transaction, person, value)} represent a payment, consider: 
\{\texttt{pay(t1, p1, 5), pay(t2, p1, 8), pay(t3, p1, 5), pay(t4, p2, 10), pay(t5, p2, 20)}\}. 
Task: Compute the sum of payments for each person.

- \textbf{Correct}: \texttt{sum(P, S) :- person(P), S = \#sum\{V, T : pay(T, P, V)\};}
  
    symbolic set is \{\langle 5, t1 \rangle, \langle 8, t2 \rangle, \langle 5, t3 \rangle\} for \(p1 \Rightarrow \text{sum}(p1, 18)\);
    symbolic set is \{\langle 10, t2 \rangle, \langle 20, t2 \rangle\} for \(p2 \Rightarrow \text{sum}(p2, 30)\).
Aggregate Atom: Common Mistakes

Let pay(transaction, person, value) represent a payment, consider:
\{pay(t1, p1, 5), pay(t2, p1, 8), pay(t3, p1, 5), pay(t4, p2, 10), pay(t5, p2, 20)\}.

Task: Compute the sum of payments for each person.

- **Correct:** sum(P, S) :- person(P), S = \#sum\{V, T : pay(T, P, V)\};
  symbolic set is \{\langle 5, t1 \rangle, \langle 8, t2 \rangle, \langle 5, t3 \rangle\} for p1 \Rightarrow sum(p1, 18);
  symbolic set is \{\langle 10, t2 \rangle, \langle 20, t2 \rangle\} for p2 \Rightarrow sum(p2, 30).

- **Mistake 1:** sum(P, S) :- person(P), S = \#sum\{T, V : pay(T, P, V)\};
  symbolic set is \{\langle t1, 5 \rangle, \langle t1, 8 \rangle, \langle t1, 5 \rangle\} for p1 \Rightarrow wrong first element! (here t1 is not even numeric)
Aggregate Atom: Common Mistakes

Let \texttt{pay(transaction, person, value)} represent a payment, consider: 
\{\texttt{pay(t1, p1, 5)}, \texttt{pay(t2, p1, 8)}, \texttt{pay(t3, p1, 5)}, \texttt{pay(t4, p2, 10)}, \texttt{pay(t5, p2, 20)}\}. 

Task: Compute the sum of payments for each person.

- **Correct**: \texttt{sum(P, S) :- person(P), S = \#sum\{V, T : \texttt{pay(T, P, V)}\};}  
symbolic set is \{\langle 5, t1 \rangle, \langle 8, t2 \rangle, \langle 5, t3 \rangle\} for \texttt{p1} \Rightarrow \texttt{sum(p1, 18)};  
symbolic set is \{\langle 10, t2 \rangle, \langle 20, t2 \rangle\} for \texttt{p2} \Rightarrow \texttt{sum(p2, 30)}.

- **Mistake 1**: \texttt{sum(P, S) :- person(P), S = \#sum\{T, V : \texttt{pay(T, P, V)}\};}  
symbolic set is \{\langle t1, 5 \rangle, \langle t1, 8 \rangle, \langle t1, 5 \rangle\} for \texttt{p1} \Rightarrow \texttt{wrong first element!}  
  (here \texttt{t1} is not even numeric)

- **Mistake 2**: \texttt{sum(P, S) :- person(P), S = \#sum\{V : \texttt{pay(T, P, V)}\};}  
symbolic set is \{\langle 5 \rangle, \langle 8 \rangle\} for \texttt{p1}, \texttt{value 5 is added only once}.  


Aggregate Atom: Common Mistakes

Let `pay(transaction, person, value)` represent a payment, consider: `{pay(t1, p1, 5), pay(t2, p1, 8), pay(t3, p1, 5), pay(t4, p2, 10), pay(t5, p2, 20)}`.

Task: Compute the sum of payments for each person.

- **Correct:** `sum(P, S) :- person(P), S = #sum{T, V: pay(T, P, V)}`;
  symbolic set is `{⟨5, t1⟩, ⟨8, t2⟩, ⟨5, t3⟩}` for p1 ⇒ `sum(p1, 18)`;
  symbolic set is `{⟨10, t2⟩, ⟨20, t2⟩}` for p2 ⇒ `sum(p2, 30)`.

- **Mistake 1:** `sum(P, S) :- person(P), S = #sum{T, V: pay(T, P, V)}`;
  symbolic set is `{⟨t1, 5⟩, ⟨t1, 8⟩, ⟨t1, 5⟩}` for p1 ⇒ wrong first element!
  (here t1 is not even numeric)

- **Mistake 2:** `sum(P, S) :- person(P), S = #sum{V: pay(T, P, V)}`;
  symbolic set is `{⟨5⟩, ⟨8⟩}` for p1, value 5 is added only once.

- **Mistake 3:** `sum(S) :- S = #sum{P: pay(T, P, V)}`;
  symbolic set is `{⟨5, p1⟩, ⟨8, p1⟩, ⟨10, p2⟩, ⟨20, p2⟩}`, persons merged.
Safety

Variables that appear solely in aggregate functions are called local variables.

- Additional safety requirements:
  - Each local variable in \{ Vars : Conj \} also appears in a positive literal in Conj.
  - Each global variable also appears
    - in a non-comparison, non-aggregate, not-free literal in the body; or
    - as a guard of an assignment aggregate atom \( X = f\{S\}, f\{S\} = X \), or \( X = f\{S\} = X \), respectively
  - Each guard of an aggregate atom is either a constant or a global variable.
Semantics of Programs with Aggregates

Generalized Gelfond-Lifschitz Reduct

Given a set $M$ of literals and a ground program $P$, the reduct (or Gelfond-Lifschitz reduct) $P^M$ is now as follows:

- remove rules from $P$
  - with $\neg a$ in the body, such that $a$ is true wrt. $M$, or
  - with $a$ in the body, such that $a$ is an aggregate atom that is false wrt. $M$; and

- remove literals $\neg a$ and aggregate atoms from all other rules.
Semantics of Programs with Aggregates

Generalized Gelfond-Lifschitz Reduct

Given a set $M$ of literals and a ground program $P$, the reduct (or Gelfond-Lifschitz reduct) $P^M$ is now as follows:

- remove rules from $P$
  - with $\textit{not } a$ in the body, such that $a$ is true wrt. $M$, or
  - with $a$ in the body, such that $a$ is an aggregate atom that is false wrt. $M$; and
- remove literals $\textit{not } a$ and aggregate atoms from all other rules.

- limitations (dlv build 21-12-2012):
  - $\#\text{min, } \#\text{max}$ just on integer constants like $\#\text{sum}$ and $\#\text{times}$
  - no recursion through aggregates (aggregate stratification)
Semantics of Programs with Aggregates

Generalized Gelfond-Lifschitz Reduct

Given a set $M$ of literals and a ground program $P$, the reduct (or Gelfond-Lifschitz reduct) $P^M$ is now as follows:

- remove rules from $P$
  - with $\text{not } a$ in the body, such that $a$ is true wrt. $M$, or
  - with $a$ in the body, such that $a$ is an aggregate atom that is false wrt. $M$; and
- remove literals $\text{not } a$ and aggregate atoms from all other rules.

- limitations (dlv build 21-12-2012):
  - $\#\text{min}, \#\text{max}$ just on integer constants like $\#\text{sum}$ and $\#\text{times}$
  - no recursion through aggregates (aggregate stratification)

- recursion through aggregates: use instead GL-reduct $P^M$ the FLP-reduct $fP^M = \{ r \in P \mid r = H \leftarrow B, M \models B \}$;
that is, keep the rules $r$ whose bodies are satisfied.
DLV Usage: Examples
Example: Minimum Spanning Tree Using Aggregates

Minimum spanning tree (with aggregates and weak constraints)

% Guess the edges that are part of the tree.
inTree(X,Y,C) v outTree(X,Y,C) :- edge(X,Y,C).
Example: Minimum Spanning Tree Using Aggregates

Minimum spanning tree (with aggregates and weak constraints)

% Guess the edges that are part of the tree.
inTree(X,Y,C) v outTree(X,Y,C) :- edge(X,Y,C).

% Check that we are really dealing with a tree!
:- start(R), not #count{X : inTree(X,R,C)} = 0.
:- edge(_,Y,_) , not start(Y),
    not #count{X : inTree(X,Y,C)} = 1.
% Note: ensures also that each node
% in the graph is reached.
Example: Minimum Spanning Tree Using Aggregates

Minimum spanning tree (with aggregates and weak constraints)

% Guess the edges that are part of the tree.
inTree(X,Y,C) v outTree(X,Y,C) :- edge(X,Y,C).

% Check that we are really dealing with a tree!
:- start(R), not #count{X : inTree(X,R,C)} = 0.
:- edge(_,Y,_), not start(Y),
    not #count{X : inTree(X,Y,C)} = 1.
% Note: ensures also that each node
% in the graph is reached.

% Nothing in life is free..
% pay for every edge that is in the solution
~ inTree(X,Y,C). [C:1]
Example: Seating Problem

**Problem:** Given some tables of a given number of chairs each, generate a sitting arrangement for a number of given guests, such that:

- people liking each other should sit at the same table, and
- people disliking each other should not sit at the same table.
Example: Seating Problem

**Problem:** Given some tables of a given number of chairs each, generate a sitting arrangement for a number of given guests, such that:

- people liking each other should sit at the same table, and
- people disliking each other should not sit at the same table.

\[
\text{at}(P, T) \lor \text{not } \text{at}(P, T) :\text{- person}(P), \text{ table}(T).
\]
Example: Seating Problem

Problem: Given some tables of a given number of chairs each, generate a sitting arrangement for a number of given guests, such that:

- people liking each other should sit at the same table, and
- people disliking each other should not sit at the same table.

\[
\text{at(P,T) v not at(P,T) :- person(P), table(T).}
\]
\[
:- \text{table(T), nchairs(C), not#count}\{P : \text{at(P,T)}\} \leq C.
\]
Example: Seating Problem

**Problem:** Given some tables of a given number of chairs each, generate a sitting arrangement for a number of given guests, such that:

- people liking each other should sit at the same table, and
- people disliking each other should not sit at the same table.

\[
\text{at}(P,T) \lor \neg \text{at}(P,T) :- \text{person}(P), \text{table}(T).
\]

\[
:- \text{table}(T), \text{nchairs}(C), \neg \#\text{count}\{P : \text{at}(P,T)\} \leq C.
\]

\[
:- \text{person}(P), \neg \#\text{count}\{T : \text{at}(P,T)\} = 1.
\]

\[
:- \text{like}(P1,P2), \text{at}(P1,T), \neg \text{at}(P2,T).
\]

\[
:- \text{dislike}(P1,P2), \text{at}(P1,T), \text{at}(P2,T).
\]
Example: Seating Problem, cont’d

\[ P_D = \{\text{person}(p1), \text{person}(p2), \text{person}(p3), \text{person}(p4), \text{table}(t1), \text{table}(t2), \text{nchairs}(4), \text{like}(p1, p2), \text{dislike}(p1, p3)\} \]
Example: Seating Problem, cont’d

\[ P_D = \{person(p1), person(p2), \]
\[ person(p3), person(p4), \]
\[ table(t1), table(t2), \]
\[ nchairs(4), \]
\[ like(p1, p2), \]
\[ dislike(p1, p3)\} \]
Example: Optimal Golomb Ruler (OGR)

**Problem:** Place a given number of marks on a ruler, such that no two pairs of marks measure the same distance, and the length of the ruler is minimal.

- **Applications:** antenna design, mobile communication technology

![Diagram of an OGR example]
Example: Optimal Golomb Ruler (OGR)

**Problem:** Place a given number of marks on a ruler, such that no two pairs of marks measure the same distance, and the length of the ruler is minimal.

- **Applications:** antenna design, mobile communication technology

% Example input for an OGR of size 4
position(0..10).
mark(1..4).
Example: Optimal Golomb Ruler (OGR), cont’d

% The position 0 is always used,
% a position is used if a mark is placed on it.
used(0).

% Guess the other positions.
free(P) v used(P) :- position(P).
% The position 0 is always used,
% a position is used if a mark is placed on it.
used(0).

% Guess the other positions.
free(P) v used(P) :- position(P).

% Exactly N used positions, where N is the number of marks.
um(N) :- #count{M : mark(M)} = N.
:- num(N), not #count{P : used(P)} = N.
Example: Optimal Golomb Ruler (OGR), cont’d

% The position 0 is always used,
% a position is used if a mark is placed on it.
used(0).

% Guess the other positions.
free(P) v used(P) :- position(P).

% Exactly N used positions, where N is the number of marks.
num(N) :- #count{M : mark(M)} = N.
:- num(N), not #count{P : used(P)} = N.

% For each used position P1, compute distance
% with each successive used position P2.
d(P1,D) :- used(P1), used(P2), P1 < P2, D = P2 - P1.
Example: Optimal Golomb Ruler (OGR), cont’d

% The position 0 is always used,
% a position is used if a mark is placed on it.
used(0).

% Guess the other positions.
free(P) v used(P) :- position(P).

% Exactly N used positions, where N is the number of marks.
num(N) :- #count{M : mark(M)} = N.
:- num(N), not #count{P : used(P)} = N.

% For each used position P1, compute distance
% with each successive used position P2.
d(P1,D) :- used(P1), used(P2), P1 < P2, D = P2 - P1.

% Discard models in which more than one pair
% of used positions have the same distance.
:- d(P1,D), d(P2,D), P1 < P2.
% The position 0 is always used,
% a position is used if a mark is placed on it.
used(0).

% Guess the other positions.
free(P) v used(P) :- position(P).

% Exactly N used positions, where N is the number of marks.
num(N) :- #count{M : mark(M)} = N.
    :- num(N), not #count{P : used(P)} = N.

% For each used position P1, compute distance
% with each successive used position P2.
d(P1,D) :- used(P1), used(P2), P1 < P2, D = P2 - P1.

% Discard models in which more than one pair
% of used positions have the same distance.
:- d(P1,D), d(P2,D), P1 < P2.

% Find the maximum used position P.
non_maxused(P1) :- used(P1), used(P2), P1 < P2.
maxused(P) :- used(P), not non_maxused(P).
Example: Optimal Golomb Ruler (OGR), cont’d

% The position 0 is always used,
% a position is used if a mark is placed on it.
used(0).

% Guess the other positions.
free(P) v used(P) :- position(P).

% Exactly N used positions, where N is the number of marks.
um(N) :- #count{M : mark(M)} = N.
/- num(N), not #count{P : used(P)} = N.

% For each used position P1, compute distance
% with each successive used position P2.
d(P1,D) :- used(P1), used(P2), P1 < P2, D = P2 - P1.

% Discard models in which more than one pair
% of used positions have the same distance.
:- d(P1,D), d(P2,D), P1 < P2.

% Find the maximum used position P.
non_maxused(P1) :- used(P1), used(P2), P1 < P2.
maxused(P) :- used(P), not non_maxused(P).

% Minimize the cost of the solution.
/~ maxused(P). [P:1]
Example: Optimal Golomb Ruler (OGR) Variants

More elegant: use the \texttt{#max} aggregate atom to find the maximum used position:

\begin{verbatim}
\% Minimize the cost of the solution,
\% i.e., the value of the largest used position.
:- #int(P1), P1 = #max\{P:used(P)\}. [P1:]
\end{verbatim}
Example: Optimal Golomb Ruler (OGR) Variants

More elegant: use the #max aggregate atom to find the maximum used position:

% Minimize the cost of the solution,
% i.e., the value of the largest used position.
:^1 #int(P1), P1 = #max{P:used(P)}. [P1:]

Program output for both variants (run with option -filter=used):

Best model: used(0), used(2), used(5), used(6)
Cost ([Weight:Level]): <[6:1]>

Best model: used(0), used(1), used(4), used(6)
Cost ([Weight:Level]): <[6:1]>
Example: Optimal Golomb Ruler (OGR) Variants

More elegant: use the \( \#\text{max} \) aggregate atom to find the maximum used position:

\[
\text{\% Minimize the cost of the solution,}
\text{\% i.e., the value of the largest used position.}
\]

\[
:\sim \#\text{int}(P1), \ P1 = \#\text{max}\{P: \text{used}(P)\}. \ [P1:]
\]

Program output for both variants (run with option \(-\text{filter}=\text{used}\)):

Best model: \text{used}(0), \text{used}(2), \text{used}(5), \text{used}(6)
Cost ([Weight:Level]): <[6:1]>

Best model: \text{used}(0), \text{used}(1), \text{used}(4), \text{used}(6)
Cost ([Weight:Level]): <[6:1]>

Results are by chance perfect optimal Golomb Rulers (i.e., no gaps in the sequence of all occurring distances).

Exercise: Which additional constraint would be needed to ensure only perfect optimal Golomb Rulers to be calculated?
Overview: DLV Extensions

DLV-Complex: extension of DLV with function symbols, lists and sets fully integrated into DLV since release 2010-10-14

*dlvex* an extension of DLV providing access to "external predicates" which are supplied via libraries

*dlvhex* a system for ASP with external computation sources

\[\text{http://www.kr.tuwien.ac.at/research/systems/dlvhex/}\]
\[\text{http://www.kr.tuwien.ac.at/research/systems/dlvhex/demo.php}\]

- enables queries to Description Logic KBs in rules

DLT extends DLV with reusable template predicate definitions

DLV^DB an extension of DLV with a tight coupling to relational DBs

- native DLV offers an ODBC interface

NLP-DL a coupling of ASP programs with Description Logics

\[\text{https://www.mat.unical.it/ianni/swlp/index.html}\]
Summary

1. The DLV system
   - DLV syntax
   - Rule safety
   - Built-in predicates

2. Weak constraints
   - Weights
   - Levels

3. Aggregates
   - Symbolic sets
   - Aggregate functions

4. DLV usage: Examples

5. DLV extensions
Software Engineering Issues

- Software engineering tools for ASP are subject of ongoing research.
  IDEs: ASPIDE\textsuperscript{3}, SeaLion\textsuperscript{4}

- Particular problem: debugging

- What to do if my program does not have (intended) answer sets?

- Some naive suggestions:
  - Decompose: divide & conquer
  - Use small/specific instances for testing
  - Test constraints one by one
  - Check auxiliary predicates separately

- Support for debugging: e.g. Spock\textsuperscript{5}

\textsuperscript{3}www.mat.unical.it/~ricca/aspide/
\textsuperscript{4}www.kr.tuwien.ac.at/research/projects/mmdasp/#Software
\textsuperscript{5}www.kr.tuwien.ac.at/research/systems/debug/index.html
ASP Integrated Development Environments (IDEs)

IDE: ease programming for both novice and skilled developers

- **SEA LION** [Busoniu et al., 2013]
  - first environment offering debugging for non-ground programs
  - unique tools for model-based engineering (ER diagrams), testing via annotations, and bi-directional visualization of interpretations.

- **ASPIIDE** [Febbraro et al., 2011]
  - comprehensive framework integrating several tools for advanced program composition and execution.
  - test-driven software development in the style of JUnit, e.g.
    - dependency graph visualizer, designed to inspect predicate dependencies and browsing the program,
    - debugger (Dodaro et al. 2015),
    - DLV profiler,
    - ARVis comparator of answer sets,
    - answer set visualizer IDPDraw.
    - data source plugin for JDBC connectivity
ASP Development Environments, cont’d

- ASPIDE is extensible
- user can provide new plugins:
  - new input formats
  - new program rewritings
  - customizing the visualization/output format of solver results
- more information: See RR 2013 tutorial

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.


Onofrio Febbraro, Kristian Reale, and Francesco Ricca.

ASPIDE: integrated development environment for answer set programming.