Knowledge Representation for the Semantic Web

Lecture 8: Answer Set Programming III

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

The DLV System

DLVSYSTEM S.R.L.
SPIN-OFF OF UNIVERSITY OF CALABRIA

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 ($\Rightarrow$ 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 [build BEN/Dec 17 2012   gcc 4.6.1]
usage: dlv {FRONTEND} {OPTIONS} [filename [filename [...]]]
Specify -help for more detailed usage information.
```

- 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**:

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

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

DLV Syntax

- **Rules**:

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

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

- **Integrity constraints**:

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

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

DLV **Syntax**

- **Rules**:

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

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

- **Integrity constraints**:

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

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

- **Queries**:

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

  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 (`not` b), or
- a built-in comparison predicate,

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

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:

```
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).
```

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!

```
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).
```

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!

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

Unsafe!

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

Built-in Predicates

- **Comparison predicates** (for integers and strings):

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

Built-in Predicates

- **Comparison predicates** (for integers and strings):

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

- **Arithmetic predicates**:

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

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

Built-in Predicates

- **Comparison predicates** (for integers and strings):

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

- **Arithmetic predicates**:

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

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

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

Example: Fibonacci Numbers

- $\underbrace{1}_{F_1}, \underbrace{1}_{F_2}, \underbrace{2}_{F_3}, \underbrace{3}_{...}, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots$
  - Except for first two numbers, each value is defined as the sum of the previous two.

Example: Fibonacci Numbers

- $\underbrace{1}_{F_1}, \underbrace{1}_{F_2}, \underbrace{2}_{F_3}, \underbrace{3}_{\dots}, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \dots$
  - 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),
            N=N2+2, N=N1+1, X=Y1+Y2.
```

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

Linear Ordering, Successor

- **Example: Employees**

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

Problem: Compute linear ordering and successor relation for employees

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 $\mathtt{empl}(\_,\_)$

  Problem: Compute linear ordering and successor relation for employees

Solve problem using projection and double negation!

% Order employees by id

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

% Define successor

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

$\mathtt{succ}(\mathtt{X},\mathtt{Y})$:- $\mathtt{prec}(\mathtt{X},\mathtt{Y})$, not $-\mathtt{succ}(\mathtt{X},\mathtt{Y})$.

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:

```
-first(X) :- succ(Y,X).
 first(X) :- empl(X,_), not -first(X).
 -last(X) :- succ(X,Y).
  last(X) :- empl(X,_), not -last(X).
```

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:

```
partialSum(X,S) :- first(X), empl(X,S).

partialSum(Y,S) :- succ(X,Y), partialSum(X,S1),
                   empl(Y,S2), S = S1 + S2.

         sum(S) :- last(X), 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):

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

  where
  - all b_i are atoms (resp. “classical” literals)
  - $Weight$, $Level$ 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):

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

  where
  - all b_i are atoms (resp. “classical” literals)
  - $Weight$, $Level$ 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 ($WC = \{wc_1, \ldots, wc_n\}$):

$$wc: \quad \texttt{:\sim} \ \ \texttt{b}_1, \ldots, \texttt{b}_\texttt{k}, \ \ \texttt{not}\ \texttt{b}_{\texttt{k}+1}, \ldots, \ \ \texttt{not}\ \texttt{b}_\texttt{m}\ . \quad [\texttt{Weight} : \texttt{Level}]$$

- 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 $I \cap \{b_{k+1}, \ldots, b_m\} = \emptyset$

Weak Constraints: Semantics for (P, WC)

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

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

- 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 ℓ is

  $$c(I, \ell) = \sum_{i=1}^{n} \sum_{(\theta,w) \in \mathcal{V}_i(I,\ell)} w,$$

  where

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

Weak Constraints: Semantics for (P, WC)

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

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

- 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 ℓ is

  $$c(I, \ell) = \sum_{i=1}^{n} \sum_{(\theta,w) \in \mathcal{V}_i(I,\ell)} w,$$

  where

  $\mathcal{V}_i(I, \ell) = \{(\theta, w) \mid wc_i\theta = :\sim B.\ [w, \ell] \in grnd(wc_i), I \not\models wc_i\theta\}$
- 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 ($WC = \{wc_1, \ldots, wc_n\}$):

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

- 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 ℓ is

  $$c(I, \ell) = \textstyle\sum_{i=1}^{n} \sum_{(\theta,w) \in \mathcal{V}_i(I,\ell)} w,$$

  where

  $\mathcal{V}_i(I, \ell) = \{(\theta, w) \mid wc_i\theta = :\sim B.\ [w, \ell] \in grnd(wc_i), I \not\models wc_i\theta\}$
- 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 ℓ and $c(M, \ell') = c(M', \ell')$ for all $\ell' > \ell$

Weak Constraints: Semantics for (P, WC)

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

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

- 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 ℓ is

  $$c(I, \ell) = \sum_{i=1}^{n} \sum_{(\theta,w) \in \mathcal{V}_i(I,\ell)} w,$$

  where

  $\mathcal{V}_i(I, \ell) = \{(\theta, w) \mid wc_i\theta = :\sim B.\ [w, \ell] \in grnd(wc_i), I \not\models wc_i\theta\}$
- 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 ℓ 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.
:~ b.
:~ c.
```

Weak Constraints: Examples

Example: Default values for weights and levels

```
a v b.   c :- b.
:~ a.
:~ b.
:~ c.

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

Answer set $\{b, c\}$ is discarded because it violates two weak constraints!

Weak Constraints: Examples/2

Example: Weights vs levels

Weights:

```
a v b.
:∼ a. [1:]
:∼ a. [1:]
:∼ b. [2:]
```

Weak Constraints: Examples/2

Example: Weights vs levels

Weights:

```
a v b.
:~ a. [1:]
:~ a. [1:]
:~ b. [2:]
```

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

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

Note: $WC = \{wc_1, wc_2, wc_3\}$,

$wc_1 = :\sim a.[1:]$,

$wc_2 = :\sim a.[1:]$,

$wc_3 = :\sim b.[2:]$

Weak Constraints: Examples/2

Example: Weights vs levels

Weights:

```
a v b.
:~ a. [1:]
:~ a. [1:]
:~ b. [2:]
```

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

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

Note: $WC = \{wc_1, wc_2, wc_3\}$,

$wc_1 =:\sim a.[1 :]$,

$wc_2 =:\sim a.[1 :]$,

$wc_3 =:\sim b.[2 :]$

Levels:

```
a v b1 v b2.
:~ a. [:1]
:~ b1. [:2]
:~ b2. [:2]
```

Weak Constraints: Examples/2

Example: Weights vs levels

Weights:

```
a v b.
:~ a. [1:]
:~ a. [1:]
:~ b. [2:]
```

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

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

Note: $WC = \{wc_1, wc_2, wc_3\}$,

$wc_1 =:\sim a.[1 :]$,

$wc_2 =:\sim a.[1 :]$,

$wc_3 =:\sim b.[2 :]$

Levels:

```
a v b1 v b2.
:~ a. [:1]
:~ b1. [:2]
:~ b2. [:2]
```

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

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)

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

:~ assign(X,P), assign(Y,P), X!=Y, same_skill(X,Y). [:2]
:~ assign(X,P), assign(Y,P), X!=Y, married(X,Y). [:2]
:~ 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: `:~ B1.[3.5:1]` `:~ 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 ($n-1$, $n-2$, ...) together!
- Weak constraints are weighted to make finer distinctions among elements of the same priority: `:~ B1.[3.5:1]` `:~ 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

```
inPath(X,Y  ) v outPath(X,Y  ) :- edge(X,Y  ).
```
} **Guess**

```
:-inPath(X,Y  ), inPath(X,Y1  ), Y != Y1.
:-inPath(X,Y  ), inPath(X1,Y  ), X != X1.
:-node(X), notreached(X).
:-not start_reached.
```
[2]

} **Check**

```
reached(X):-start(X).
reached(X):-reached(Y), inPath(Y,X  ).
start_reached :- start(Y), inPath(X,Y  ).
```
} **Auxiliary**

[2] 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

```
inPath(X,Y,C) v outPath(X,Y,C) :- edge(X,Y,C).        } Guess

:-inPath(X,Y,C), inPath(X,Y1,C1), Y != Y1.
:-inPath(X,Y,C), inPath(X1,Y,C1), X != X1.
:-node(X), notreached(X).                               } Check
:-not start_reached.[2]

reached(X):-start(X).
reached(X):-reached(Y), inPath(Y,X,C).                  } Auxiliary
start_reached :- start(Y), inPath(X,Y,C).

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

[2] 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 minium spanning tree with root at the starting node

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

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

```
reached(X):-start(X).
reached(X):-reached(Y), 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 minium spanning tree with root at the starting node

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

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

```
reached(X):-start(X).
reached(X):-reached(Y), 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 minium spanning tree with root at the starting node

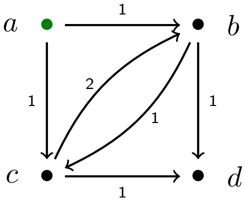
```
inTree(X,Y,C) v outTree(X,Y,C) :- edge(X,Y,C).
```
} **Guess**

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

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

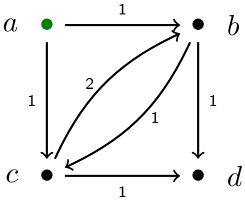
```
:~ inPath(X,Y,C).[C:1]
```
} **Optimize**

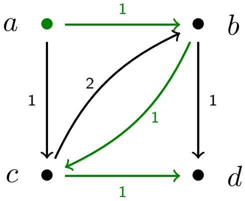
Example: Minimum Spanning Tree (ctd.)



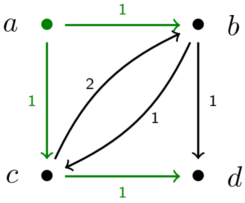
$$
\begin{aligned}
P_D = \{&node(a), node(b),\\
&node(c), node(d),\\
&edge(a,b,1), edge(a,c,1)\\
&edge(c,b,2), edge(b,c,1)\\
&edge(b,d,1), edge(c,d,1)\\
&start(a)\}
\end{aligned}
$$

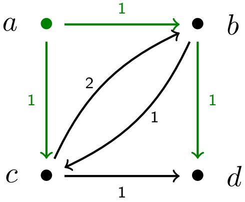
Example: Minimum Spanning Tree (ctd.)





$$
\begin{aligned}
P_D = \{& node(a), node(b),\\
& node(c), node(d),\\
& edge(a,b,1), edge(a,c,1)\\
& edge(c,b,2), edge(b,c,1)\\
& edge(b,d,1), edge(c,d,1)\\
& start(a)\}
\end{aligned}
$$





Aggregates

- Allow arithmetic operations over a set of elements, as e.g. in SQL:

```
select count(*) from empl;
```

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)}
```

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 `#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 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:

```
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 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: $\{\mathtt{S}, \mathtt{X} : \mathtt{empl}(\mathtt{X}, \mathtt{S})\}$

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 $\{\#\texttt{count}, \#\texttt{sum}, \#\texttt{times}, \#\texttt{min}, \#\texttt{max}\}$

Aggregate Functions

Aggregate Function Expression

$$f\{S\}$$

where

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

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

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 $\{\#\mathtt{count}, \#\mathtt{sum}, \#\mathtt{times}, \#\mathtt{min}, \#\mathtt{max}\}$

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

Informal Meaning: The sum of salaries of all employees.

- $\#\mathtt{count}$ 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.

Aggregate Functions, cont'd

Identical Projections

Note:

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

as identical projections S of different elements count multiple times

Aggregate Functions, cont'd

Identical Projections

Note:

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

as identical projections S of different elements count multiple times

for $S = \emptyset$:

- $\#\mathtt{sum}$ returns 0
- $\#\mathtt{times}$ returns 1
- $\#\mathtt{min}$ and $\#\mathtt{max}$ 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 "$\leq +\infty$"

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: $\#\mathtt{sum}\{\mathtt{S}, \mathtt{X} : \mathtt{empl}(\mathtt{X}, \mathtt{S})\} \leq 3800$

Informal Meaning: True if sum of salaries ≤ 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 $\mathtt{pay(transaction, person, value)}$ represent a payment, consider:
$\{\mathtt{pay(t1, p1, 5), pay(t2, p1, 8), pay(t3, p1, 5), pay(t4, p2, 10), pay(t5, p2, 20)}\}$.
Task: Compute the sum of payments for each person.

Aggregate Atom: Common Mistakes

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

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

Aggregate Atom: Common Mistakes

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

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

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

Aggregate Atom: Common Mistakes

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

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

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

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

Aggregate Atom: Common Mistakes

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

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

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

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

- Mistake 3: $\mathtt{sum}(S) \text{ :- } S = \#\mathtt{sum}\{\mathtt{V}, \mathtt{P} : \mathtt{pay(T, P, V)}\}$;
  symbolic set is $\{\langle \mathbf{5}, \mathtt{p1}\rangle, \langle \mathbf{8}, \mathtt{p1}\rangle, \langle \mathbf{10}, \mathtt{p2}\rangle, \langle \mathbf{20}, \mathtt{p2}\rangle\}$, 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 $not\, a$ 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 $not\, a$ 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 $not\, a$ 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 $not\, a$ and **aggregate atoms** from all other rules.

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

Semantics of Programs with Aggregates

Generalized Gelfond-Lifschitz Reduct

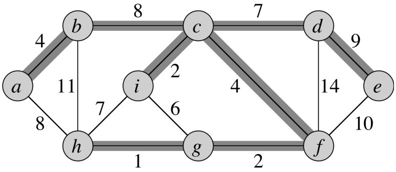
Given a set M of literals and a ground program P, the reduct (or Gelfond-Lifschitz reduct) P^M is now as follows:

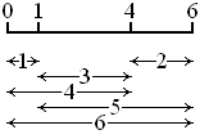
- remove rules from P
  - with $not\, a$ in the body, such that a is true wrt. M, or
  - with a in the body, such that a is an **aggregate atom** that is false wrt. M; and
- remove literals $not\, a$ and **aggregate atoms** from all other rules.

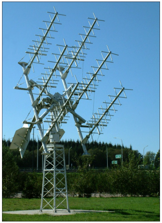
- limitations (dlv build 21-12-2012):
  - #min, #max just on integer constants like #sum and #times
  - no recursion through aggregates (aggregate stratification)

- 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.

```
at(P,T) v not_at(P,T) :- person(P), 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.

```
at(P,T) v not_at(P,T) :- person(P), table(T).
:- table(T), nchairs(C), not#count{P : at(P,T)} <= C.
```

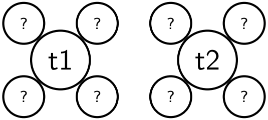
Example: Seating Problem

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.

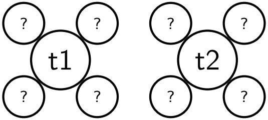
```
at(P,T) v not_at(P,T) :- person(P), table(T).
:- table(T), nchairs(C), not#count{P : at(P,T)} <= C.
:- person(P), not #count{T : at(P,T)} = 1.
:- like(P1,P2), at(P1,T), not at(P2,T).
:- dislike(P1,P2), at(P1,T), at(P2,T).
```

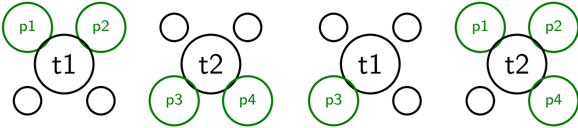
Example: Seating Problem, cont'd



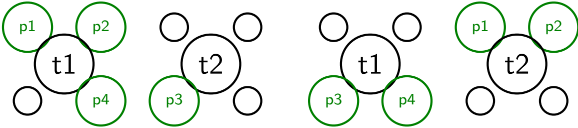
$$
\begin{aligned}
P_D = \{&person(p1), person(p2),\\
&person(p3), person(p4),\\
&table(t1), table(t2),\\
&nchairs(4),\\
&like(p1, p2),\\
&dislike(p1, p3)\}
\end{aligned}
$$

Example: Seating Problem, cont'd



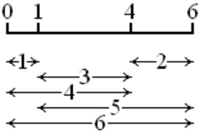


$$
\begin{aligned}
P_D = \{ & person(p1), person(p2), \\
& person(p3), person(p4), \\
& table(t1), table(t2), \\
& nchairs(4), \\
& like(p1, p2), \\
& dislike(p1, p3)\}
\end{aligned}
$$



Example: Optimal Golomb Ruler (OGR)

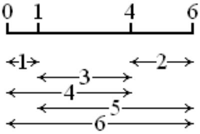
Problem: Place a given number of marks on a ruler, such that no two pairs of marks measure the same distance, and the length of the ruler is minimal.



- Applications: antenna design, mobile communication technology

Example: 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).
```

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.
```

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.
```

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.

% 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.
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).

% Minimize the cost of the solution.
:~ maxused(P). [P:1]
```

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.
:~ #int(P1), P1 = #max{P:used(P)}. [P1:]
```

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.
:~ #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 `#max` aggregate atom to find the maximum used position:

```
% Minimize the cost of the solution,
% i.e.,the value of the largest used position.
:~ #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]>
```

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 DLVproviding access to "external predicates" which are supplied via libraries

dlvhex a system for ASP with external computation sources
http://www.kr.tuwien.ac.at/research/systems/dlvhex/
http://www.kr.tuwien.ac.at/research/systems/dlvhex/demo.php

- enables queries to Description Logic KBs in rules

DLT extends DLVwith reusable template predicate definitions

DLVˆDB an extension of DLVwith a tight coupling to relational DBs

- native DLV offers an ODBC interface

NLP-DL a coupling of ASP programs with Description Logics
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[3], SeaLion[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[5]

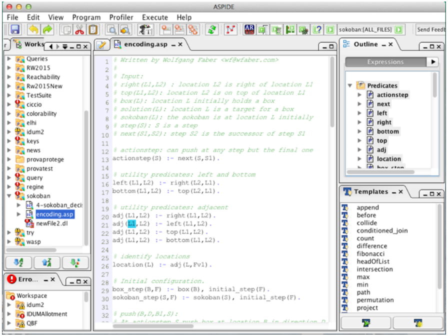
[3] `www.mat.unical.it/~ricca/aspide/`
[4] `www.kr.tuwien.ac.at/research/projects/mmdasp/#Software`
[5] `www.kr.tuwien.ac.at/research/systems/debug/index.html`

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.
- **ASPIDE** [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

`https://www.mat.unical.it/ricca/downloads/rr2013-tutorial.pdf`

References I

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.

TPLP, 13(4-5):657–673, 2013.

Onofrio Febbraro, Kristian Reale, and Francesco Ricca.

ASPIDE: integrated development environment for answer set programming.

In James P. Delgrande and Wolfgang Faber, editors, *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, volume 6645 of *Lecture Notes in Computer Science*, pages 317–330. Springer, 2011.