2.0 VO 184.205, WS 2005/06

Verarbeitung deklarativen Wissens (Declarative Knowledge Processing)

Answer Set Programming /2

Thomas Eiter

Institute of Information Systems

Knowledge Based Systems Group

Vienna University of Technology

`http://www.kr.tuwien.ac.at/staff/eiter`

Roadmap

1. Introduction & Background
2. Concepts
3. Disjunctive Logic Programs
4. Answer Set Solvers
5. Guess and Check Paradigm
6. The DLV System – Core
7. ASP Extensions
8. KR Applications
9. DLV Frontends

5. The "Guess and Check" Methodology

- **Recall:** Uniform ASP problem encoding divides into problem specification, PS, and input facts F_I, encoding an an instance I of problem $\mathbf{P}$.
- **Issue:** How to define a program P_{PS}?

A "Guess & Check" program P for $\mathbf{P}$ consists of the following two main parts:

- **Guessing Part**

  The guessing part $G \subseteq P$ of the program defines the search space, such that stable models of $G \cup F_I$ represent "solution candidates" for I.

- **Checking Part (optional)**

  The checking part $C \subseteq P$ of the program filters the solution candidates in such a way that the stable models of $G \cup C \cup F_I$ represent the admissible solutions for the problem instance I.

The "Guess and Check" Methodology /2

- In the extremal case, $G = P$ and $C = \emptyset$
- In general, the generation of the search space may be guarded by some rules
- Such rules might be also regarded as part of C
- No strict formal definition here, more intuitive
- Often:
  - G includes disjunctive rules, which define the search space.
  - C consists of constraints and rules which prune illegal branches.

Example: Selecting an Element

Input: Departments, represented by `dept(_)`.

Problem: Select a single department

Note: Multiple solutions!

```
sel(D) ∨ omit(D) :- dept(D).                          } Guess

          :- sel(D1), sel(D2), D1 != D2.   }
some_sel  :- sel(D)                        } Check
          :- not some_sel                  }
```

Selecting an Element /2

More elegant: Use disjunction in checking

```
sel(D) :- dept(D), not omit(D).                    } Guess

omit(D1) ∨ omit(D2) :- dept(D1), dept(D2), D1! = D2.} Check
```

Alternative way (homework):

- normal logic program, use unstratified default negation

3-Colorability

Input: A graph represented by $\texttt{node}(\_)$ and $\texttt{edge}(\_,\_)$.

Problem: Assign a color to all nodes such that

adjacent nodes always have different colors.

NP-complete problem!

$$\texttt{col}(\texttt{X},\texttt{r}) \vee \texttt{col}(\texttt{X},\texttt{g}) \vee \texttt{col}(\texttt{X},\texttt{b}) \texttt{ :- node}(\texttt{X}).\ \}\ \textbf{Guess}$$

$$\texttt{:- edge}(\texttt{X},\texttt{Y}), \texttt{col}(\texttt{X},\texttt{C}), \texttt{col}(\texttt{Y},\texttt{C}).\ \}\ \textbf{Check}$$

Hamiltonian Path

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

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

```
inPath(X,Y) ∨ outPath(X,Y) :- arc(X,Y).        } Guess

:- inPath(X,Y), inPath(X,Y1), Y <> Y1.          }
:- inPath(X,Y), inPath(X1,Y), X <> X1.          } Check
:- node(X), not reached(X).                     }
reached(X) :- start(X).                         }
reached(X) :- reached(Y), inPath(Y,X).          }
```

Ramsey Numbers

$R(k, m)$ is the least integer n such that, no matter how we color the arcs of a clique (complete undirected graph) with n nodes using red and blue, there is a red clique with k nodes or a blue clique with m nodes.

Note: $R(k, m)$ exists for all pairs of positive integers k and m.

Consider the following program $P_{3,4}$:

```
blue(X,Y) ∨ red(X,Y) :- arc(X,Y).                    } Guess
:- red(X,Y), red(X,Z), red(Y,Z).                     }
:- blue(X,Y), blue(X,Z), blue(Y,Z),                  } Check
   blue(X,W), blue(Y,W), blue(Z,W).                  }
```

F_n: Facts on predicates *node* and *arc* encoding a complete (loop-free) graph on n nodes.

Notice: $P_{3,4} \cup F_n$ has some stable model if and only if $R(3, 4) > n$.

The DLV System – Core

- DLV is a state-of-the-art disjunctive ASP system.
- Developed at University of Calabria and TU Wien.
- It possesses richer syntax than normal logic programs, resulting in **higher expressibility**!
- Offers frontends for specific KR-tasks (diagnosis, planning, etc.).
- DLV homepage:

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

- DLV description: [13]

DLV Features

- Language: **extended disjunctive logic programs**, i.e., logic programs admitting
  - disjunctions in rule heads
  - default negation ($\mathrm{not}\, a$)
  - strong negation ($\text{–}a$)
- Additionally:
  - integer, arithmetic, and comparison built-ins
  - constraints
  - weak constraints
  - aggregates
- Support for *brave & cautious reasoning*
- Optimal stable model computation
- frontends (KR tasks/formalisms, preferences,. . .)

DLV Syntax

- **Rules**

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

  where $\mathtt{n} \geq 1$, $\mathtt{m} \geq 0$ and all $\mathtt{a_i}$, $\mathtt{b_j}$ are atoms or strongly negated atoms

  No function symbols!

- **Constraints**

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

  can be regarded as shorthand (add $false$ in head, $\mathrm{not}\ false$ in body)

- **Queries**

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

Strong Negation

- Negated atoms $-a$
- Also known as "*Classic Negation*" or "*True Negation*."
- Strong negation more "*cautious*":

```
ok(X) :- -broken(X), rocket(X).
rocket(ariane5).
```

  . . . has no model.

- Strong negation may cause *inconsistency*:

```
true.
a :- true.
-a :- true.
```

  . . . has no model. Thus `true` cannot be bravely derived.

Default vs. Strong Negation

- Default Negation is **two-valued**: true, false
- Strong Negation is **three-valued**: true, false, *undefined* (= *don't know*)

Consider an agent A with the following task: "At a railroad crossing, cross the rails if no train approaches."

- `walk(A) :- crossing(A), not train_approaches(A).`
- `walk(A) :- crossing(A), -train_approaches(A).`

Answer Sets of Extended Disjunctive LPs

- Definition of **(consistent) answer sets** is similar to stable models.
  - Let P be an extended disjunctive logic program
  - for each predicate p, view $-p$ as new predicate
  - Add to P constraints

$$\texttt{:- } p(\vec{x}), -p(\vec{x}).$$

  - Let P^- be the resulting program

- **Definition:**

$M \subseteq HB(P^-)$ is a (consistent) answer set of P if and only if M is a stable model of P^-

DLV Built-in Predicates

- **Comparison Predicates:**

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

- **Arithmetic Predicates:**

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

$\#\mathtt{int}(\mathtt{X})$: $\mathtt{X}$ is known integer $(0 \leq \mathtt{X} \leq \mathtt{N})$.

$\#\mathtt{succ}(\mathtt{X},\mathtt{Y})$: $\mathtt{Y}$ is successor of $\mathtt{X}$, i.e., $\mathtt{Y} = \mathtt{X} + 1$.

$+(\mathtt{X},\mathtt{Y},\mathtt{Z})$: $\mathtt{Z} = \mathtt{X} + \mathtt{Y}$.

$*(\mathtt{X},\mathtt{Y},\mathtt{Z})$: $\mathtt{Z} = \mathtt{X} * \mathtt{Y}$.

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

Example: Fibonacci Numbers

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, . . .
  - 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).
fib(N,X) :- fib(N1,Y1), fib(N2,Y2),
            +(N2,2,N), +(N1,1,N), +(Y1,Y2,X).
```

Safety Requirement in DLV

- DLV requries that each variable occurring in a rule r (resp. constraint) must occur in r in at least one non-comparison, `not`-free literal in the body of r.
- Such a “safety” assumption is customary in datalog settings

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).
```

DLV Reasoning Tasks

- DLV implements two basic **reasoning modes** relevant for answer set programming:
  - **Brave Reasoning**: Is a query true in *some* answer set?
  - **Cautious Reasoning**: Is a query true in *all* answer sets?

Using DLV

- DLV is command-line oriented . . .
- . . . but there is also a simple GUI.
- Input is read from files whose names are passed on the command-line.
- If the command-line option "`--`" has been specified, input is also read from standard input (stdin).
- Output is printed to standard output (stdout), one line per model / answer set.
- Detailed documentation is at
  `http://www.dbai.tuwien.ac.at/proj/dlv/`

Selected DLV Command-line Options

`--`	Also read input from stdin
-n=<n>	Compute at most <n> answer sets.
-n=0 / -n=all	Compute all answer sets.
-N=<n>	Set #maxint to <n>.
	Equivalent to `#maxint=<n>.` in the input.
-brave	Brave reasoning
-cautious	Cautious reasoning
-instantiate	Only ground and print the instantiation.

Selected DLV Command-line Options /2

-silent	No debugging output, for use with non-human filters.
-stats	Print statistics and timings regarding the computation.
-v	Be a bit more verbose than usual.
-nofacts	Do not print facts from input.
-filter=<X>[<Y>,...]	Include only instances of predicate(s) <X>[<Y>,...] in output.
-pfilter=<X>[<Y>,...]	Include only positive instances of predicate(s) <X>[<Y>,...] in output.
-wait	Before terminating, wait until Return is pressed.

7. ASP Extensions

- Besides disjunction and strong negation, many extensions of normal logic programs have been proposed
- Some of these extensions are motivated by applications
- Some of these extensions are syntactic sugar, other strictly add expressiveness
- Comprehensive survey of extensions:

  See `http://www.tcs.hut.fi/Research/Logic/wasp/wp3/`

- Here, we consider some DLV specific extensions.

Weak Constraints in DLV

- Allow the formalization of optimization problems in an easy and natural way.
- Constraints vs. weak constraints:
  - Constraints “kill” unwanted models;
  - Weak constraints express desiderata which should be satisfied, if possible.
- The answer sets of a program P with a set W of weak constraints are those answer sets of P which *minimize the number of violated constraints*.
- Such answer sets are called *optimal or best models of* (P, W).

Weak Constraints

- Syntax:

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

- In the presence of weights, best models minimize the sum of the weights of violated constraints.
- Semantics minimizes the violation of constraints with highest priority level first; then with the lower priority levels in descending order.
- `Level` part is syntactic sugar, can be compiled into weights (exercise).

Weak Constraints: Examples

```
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

```
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]>

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

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

Guess-Check-Optimize Methodology

- Extend the “Guess & Check” Methodology
- Use weak constraints to filter out best (optimal) solutions

“Guess-Check-Optimize”: Divide P into three main parts:

- **Guessing Part**

  $G \subseteq P$: *Answer_Sets*$(G \cup F_I)$ represent “solution candidates” for I.

- **Checking Part (optional)**

  $C \subseteq P$: *Answer_Sets*$(G \cup C \cup F_I)$ represent the admissible solutions for I.

- **Optimization Part (optional)**

  The optimization part $O \subseteq P$ consists of weak constraints, and implicitly defines an objective function f : *Answer_Sets*$(G \cup C \cup F_I) \rightarrow \mathbb{N}$

  Those answer sets minimizing f are selected.

Example: Employee Assignment

- 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 1) is more important than 2) and 3), which are equally
- Layers express the relative importance of the requirements.

$$\texttt{assign(X, p1)} \vee \texttt{assign(X, p2)} \texttt{:-} \texttt{employee(X)}. \quad \} \textbf{ Guess}$$

$$\begin{aligned} &\texttt{:}\sim \texttt{assign(X, P), assign(Y, P), same\_skill(X, Y).}\ [\texttt{:}\ 1] \\ &\texttt{:}\sim \texttt{assign(X, P), assign(Y, P), married(X, Y).}\ [\texttt{:}\ 2] \\ &\texttt{:}\sim \texttt{assign(X, P), assign(Y, P), X!} = \texttt{Y}, \mathrm{not}\ \texttt{know(X, Y).}\ [\texttt{:}\ 2] \end{aligned} \quad \Big\} \textbf{ Optimize}$$

Weak Constraints with Weights

- A single weak constraints of in some layer n is more important than *all* weak constraints in higher layers ($n+1$, $n+2$,...) *together*!
- Weak constraints are weighted to make finer distinctions among elements of the same priority: `:~ G1.[3.5:1] :~ G2.[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

Traveling Salesperson

Given: Weighted directed graph $G = (V, E, C)$ and a node $a \in V$ of this graph.

Task: Find a minimum-cost cycle (closed path) in G starting at a and going through each node in V exactly once.

- G stored by facts over predicates $\texttt{node}(X)$ and $\texttt{arc}(X, Y)$.
- Starting node a is specified by the predicate `start` (unary).

$$\texttt{inPath(X,Y,C)} \vee \texttt{outPath(X,Y,C)} \texttt{:-start(X)}, \texttt{arc(X,Y,C)}.$$
$$\texttt{inPath(X,Y,C)} \vee \texttt{outPath(X,Y,C)} \texttt{:-reached(X)}, \texttt{arc(X,Y,C)}.$$
$$\texttt{reached(X)} \texttt{:-inPath(Y,X,C)}. \quad \textbf{(aux.)}$$

} **Guess**

$$\texttt{:-inPath(X,Y,\_)}, \texttt{inPath(X,Y1,\_)}, \texttt{Y} <> \texttt{Y1}.$$
$$\texttt{:-inPath(X,Y,\_)}, \texttt{inPath(X1,Y,\_)}, \texttt{X} <> \texttt{X1}.$$
$$\texttt{:-node(X)}, \text{not } \texttt{reached(X)}.$$

} **Check**

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

} **Optimize**

Minimum Spanning Tree

Given: A weighted directed graph, by means of `node(X)` and `edge(X, Y, C)`

Task: Compute a tree which starts at a `root` node, spans that graph, and has minimal cost

```
% Guess the edges that are part of the tree.
inTree(X, Y, C) ∨ outTree(X, Y) :- edge(X, Y, C), reached(X).
% Check that we are really dealing with a tree!
:- root(X), inTree(_, X, C).
:- inTree(X, Y, C1), inTree(Z, Y, C2), X != Z.
% Every node has to be reached.
reached(X) :- root(X).
reached(Y) :- reached(X), inTree(X, Y, C).
:- node(X), not reached(X).
% Nothing in life is free.
:∼ inTree(X, Y, C). [C : 1]
```

Aggregates

- Compute aggregate functions over a set of values, similar as in SQL (count, min, max, sum)
- A few use examples:

```
:- actiontime(T), #count{ move(B,L,T) } >= 4.

small :- #max{ X : f(A,X,C), b(C,G) } < 3.

ok_price :- 30 <= #sum{ Price :
                        bought(Good),
                        price(Good,Price) } < 50.
```

- Smodels offers similar constructs (cardinality atoms, weight constraints).

Aggregate Atoms – Syntax

- **Symbolic Set:** Expression

$$\{Vars : Conj\}$$

of a list $Vars$ of variables and a list $Conj$ of literals (safety required).

- **Aggregate Function:** Expression

$$f\,\{Vars : Conj\}$$

where

  - $f \in \{\#count, \#min, \#max, \#sum, \#times\}$, and
  - $\{Vars : Conj\}$ is a symbolic set

Aggregate Atoms – Syntax /2

- **Aggregate Atom:** Expression

$$
\begin{aligned}
Agg\_Atom ::= \quad & val\ \theta\ f\ \{Vars : Conj\} \\
| \quad & f\ \{Vars : Conj\}\ \theta \\
| \quad & val_l\ \theta_l f\ \{Vars : Conj\}\ \theta_r\ val_u
\end{aligned}
$$

where

  - val, val_l, val_u are constants or variables,
  - $\theta \in \{<, >, \leq, \geq, =\}$,
  - $\theta_l, \theta_r \in \{<, \leq\}$, and
  - $f\ \{Vars : Conj\}$ is an aggregate function.

Aggregate Atoms – Semantics

- Informally:

  Suppose I is an interpretation.

  - Evaluate symbolic set $\{Vars : Conj\}$ with respect to I: Collect all instances of $Vars$ for which $Conj$ is true in I (Result: $SemSet$).
  - Apply f on $SemSet$ (Result: $v = f(SemSet)$).
  - Evaluate comparison $val\ \theta\ v$ resp. $val_l\ \theta_l\ v \wedge v\ \theta_r\ val_u$ with (instantiated) value val resp. values val_l, val_u.

- Appealing formal definition of semantics is a bit tricky

- Widely acknowledged proposal: Faber et al. [9]

Restaurant Seating Problem

- A restaurant has tables ($\texttt{table}(\texttt{T})$) with certain number of chairs ($\texttt{nchairs}(\texttt{T},\texttt{C})$).
- Persons ($\texttt{person}(\texttt{T})$) should be seated such that persons who like each other ($\texttt{likes}(\texttt{P1},\texttt{P2})$) are at the same table.
- Persons who dislike each other ($\texttt{dislikes}(\texttt{P1},\texttt{P2})$) are at different tables.

```
% Guess whether person P sits at table T or not.
at(P,T) v not_at(P,T) :- person(P), table(T).
% Check capacity of tables.
:- table(T), chairs(T,C), not #count{P : at(P,T)} <= C.
% Check seating of each person.
:- person(P), not #count{T : at(P,T)} = 1.
% Check "likes".
:- like(P1,P2), at(P1,T), notat(P2,T).
% Check "dislikes".
:- dislike(P1,P2), at(P1,T), at(P2,T).
```

Minimum Spanning Tree revisited

Variant: Use aggregates

```
% Guess a spanning tree.
inTree(X, Y, C) ∨ outTree(X, Y) :- edge(X, Y, C).
% The root has no incoming edge.
:- root(R), not #count{X : inTree(X, R, C)} = 0.
% Each other node must have exactly one incoming edge.
:- node(Y), not root(Y), not #count{X : inTree(X, Y, C)} = 1.
% Optimize.
:∼ inTree(X, Y, C). [C : 1]
```

Description Logic Atoms

- Recent ASP extension (in progress): Interface description logics [8]
- Use taxonomies, built in a description logic
  - Concepts (e.g., Bird, Penguin, Fliers), and
  - individuals (e.g., *Tweety*)
  - roles (e.g., being father of *Tweety*)
- Example knowledge base KB:

$Penguin \sqsubseteq Bird,$	(penguins are birds)
$Penguin \sqsubseteq \neg Fliers,$	(penguins can't fly)
$Bird(Tweety).$	(*Tweety* is a bird)

Description Logic Atoms /2

- Want to do terminological reasoning:

  E.g.,

  - Are penguins birds;
  - is *Tweety* a flier?

- Semantic Web:

  - OWL (Web Ontology Language) as a W3C standard ontology language
  - it is based on certain description logics

- Description logics have classical semantics (first-order logic)

- Question: How to couple ASP and description logics ?

Description Logic Atoms /3

- Access in a LP a description logic KB via *Description Logic atoms*:

  $$DL[Updates; Query](X)$$

  where

  - $Updates$ specifies changes (additions) to the facts in KB
  - $Query(X)$ is a query to KB

  Examples:

  - `DL[Bird](X)`
  - `DL[Penguin+=p_bird;Flier](''Tweety'')`

- Taxonomic knowledge can be imported in this way
- Rules can be applied on top of KB this way (e.g., Defaults may be expressed)

Example: “Birds usually fly”

Set up default rules:

```
% X flies, if it’s a bird and there is no
% evidence against it

  flies(X) :-DL[Bird](X), not -flies(X).

% X does not fly, if we can disprove it, taking
% assumptions into account

  -flies(X) :-DL[Flier+=flies;-Flier](X).
```

User Query: $flies(\text{“}Tweety\text{”})$?

More Information:

`http://www.kr.tuwien.ac.at/staff/roman/semweblp/`

8. KR Applications

- A number of different KR applications of ASP exist
- E.g., diagnosis, planning, reasoning about actions, inheritance reasoning, etc (see http://www.kr.tuwien.ac.at/research/WASP/report.html)
- ASP / Stable semantics is a fruitful formalisms for encoding similar domain specific languages
- Can serve as a “KR implementation tool”
- On top of DLV, several KR applications have been realized

Planning

Input: Fluents (state variables) F

Actions (that usually modify fluents) A

Initial state I, goal state G

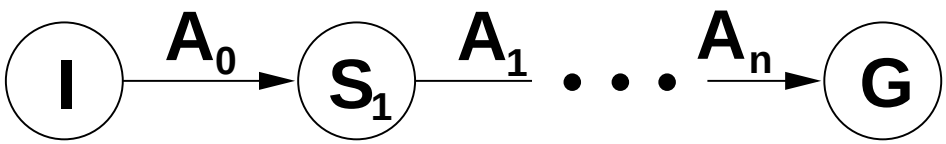
State constraints, action descriptions, inertia

Setting: States are usually provided implicitly (limited by state constraints)

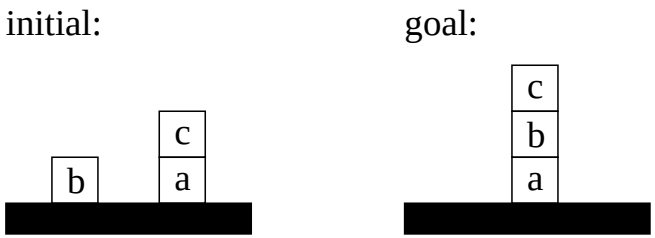
Discrete steps of time

Output: A sequence of action sets $\langle A_0, A_1, \ldots, A_n \rangle$

transforming the initial state into the goal state



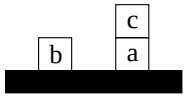
Blocksworld Planning – Example



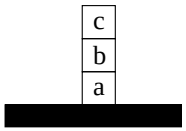

```
block(a).  block(b).  block(c).
on(b, table).  on(c, a).  on(a, table).
```

Blocksworld Planning – Modeling the problem

initial:



goal:



Fluents:

`on(B,L)`, block B is at location L.

Actions:

`move(B,L)`, move block B to location L.

Action description:

`on(X,Y)` holds after `move(X,Y)`

`on(X,Z)` does not hold after `move(X,Y)`

`on(Z,X)` disallows all `move(X,Y)`

`on(Z,X)` disallows all `move(Y,X)`

no `move(X,X)` makes sense

no concurrent actions

Inertia:

`on(_,_)` is inertial

Blocksworld Planning – Problem Encoding 1/5

Basic objects:

- Blocks and locations are defined in the particular instances

```
% Blocks can reside on the table and on other blocks.
location(table).
location(B) :- block(B).
```

Sequences of actions and intermediate states:

- Add a time argument to the fluents and actions.
- Specify the length of the sequence by builtin bounded integers.
- k states and $k-1$ actions (sets)

```
actiontime(T) :- #int(T), T < #maxint.
```

Blocksworld Planning – Problem Encoding 2/5

```
% The Guess: At any actiontime T, we either move a block B to some location L,
% or we don't.
move(B,L,T) v -move(B,L,T) :- block(B), location(L),
                              actiontime(T), B <> L.

% Inertia: Unless we know that a block has been moved, we assume
% that it is still at the same location at the next time step.
on(B,L,T1) :- on(B,L,T), not -on(B,L,T1), #succ(T,T1).
```

Blocksworld Planning – Problem Encoding 3/5

```
% Effects of moving a block: The block is now at the new location...
on(B,L,T1) :- move(B,L,T), #succ(T,T1).
% ...and it is no longer at the old location.
-on(B,L,T1) :- move(B,L1,T), on(B,L,T), #succ(T,T1).
               L1 <> L.

% Moreover, we want to move a block at any time, that is, we want to
% avoid points of time without a move.
moved(T) :- move(_,_,T).
:- actiontime(T), not moved(T).
```

Blocksworld Planning – Problem Encoding 4/5

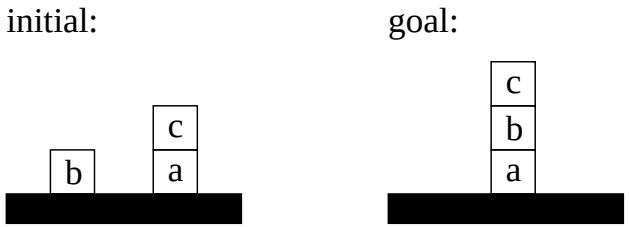
```
% A block can only be moved if it's clear.
:- move(B,L,T), on(_,B,T).

% Two blocks cannot be on the same block at the same time,
% thus a block cannot be moved to a block that is already occupied.
:- move(B,B1,T), on(_,B1,T), block(B1).

% No concurrency: We cannot move two blocks at the same time, nor can we
% move some block(s) to two different locations.
:- move(B,_,T), move(B1,_,T), B<>B1.
:- move(_,L,T), move(_,L1,T), L<>L1.
```

Blocksworld Planning – Problem Encoding 5/5

Initial state and goal:




```
% Initial state.
on(a,table,0).  on(b,table,0).  on(c,a,0).

% Goal condition – must specify the stage (number of steps) k !
goal :- on(a,table,k), on(b,a,k), on(c,b,k).
:- not goal.
```

Diagnosis

- **Input:**

  Hypotheses, a background theory, observations $\langle H, T, O \rangle$

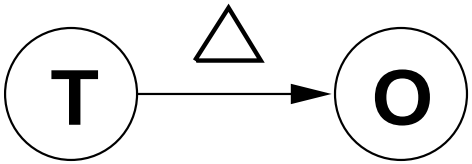
- **Options:**

  Semantics of explanation:

  - abductive; consistency-based
  - preference criterion: single failure, subset minimal, no restriction

- **Output:**

  A set of hypotheses $\Delta \subseteq H$ explaining the observations by the theory



Abductive Diagnosis: Formal Definition

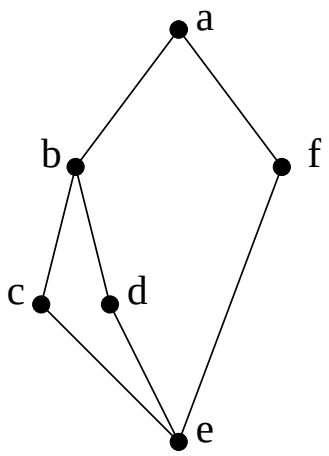
- An abductive diagnostic problem is a tuple $\langle H, T, O \rangle$, where
  - H is a set of ground literals, called *hypotheses*;
  - T is an ELP, called *background theory*;
  - O is a set of ground literals, called *observations*.
- A *diagnosis* is a set $\Delta \subseteq H$ such that there is an answer set of $T \cup \Delta$ in which all observations are true.
- Variations:
  - Single-error diagnosis: $|\Delta| = 1$
  - Subset-minimal diagnosis: there is no other diagnosis Δ' such that $\Delta' \subset \Delta$

Diagnosis: Implementation in DLV

Exploit the “Guess & Check” paradigm!

1. Specify a goal (i.e., the observations that we need to explain)
2. Guess a diagnosis
3. Check its validity

Example: Network Diagnosis



Theory:

```
connected(a,b). ......
node(X) :- connected(X,_).
node(Y) :- connected(_,Y).
reaches(X,X) :- node(X), not offline(X).
reaches(X,Z) :- reaches(X,Y),
                connected(Y,Z),
                not offline(Z).
```

Hypotheses:

```
offline(a).  offline(b).  offline(c).
offline(d).  offline(e).  offline(f).
```

Observations:

```
not offline(a).  not reaches(a,e).
```

Network Diagnosis /2

- The observations form a query (= goal):

```
not offline(a), not reaches(a,e)?
```

- No preference criterion: guess a subset of the hypotheses

```
offline(a) v -offline(a).
offline(b) v -offline(b).
offline(c) v -offline(c).
offline(d) v -offline(d).
offline(e) v -offline(e).
offline(f) v -offline(f).
```

- Single-failure diagnosis: guess exactly one hypothesis

```
offline(a) v offline(b) v ...  v offline(f).
```

9. DLV Frontends

- DLV offers frontends for particular KR tasks:
  - diagnosis
  - knowledge-based planning ($\mathcal{K}$ language)
  - inheritance reasoning
  - etc
- Also:
  - frontend to SQL3

Invoking Built-In DLV Frontends

-FD	Abductive diagnosis
-FDmin	Abductive diagnosis, subset minimal
-FDsingle	Abductive diagnosis, single error
-FR	Reiter's diagnosis
-FRmin	Reiter's diagnosis, subset minimal
-FRsingle	Reiter's diagnosis, single error
-FP	Planning with "K" Action Language
...	...
-FS	SQL3

Invoking Built-In DLV Frontends /2

Diagnosis: Hypotheses and observations are stored in files with extensions `.hyp` and `.obs`, respectively; the files for the theory use any extension different from these.

See `http://www.dbai.tuwien.ac.at/proj/dlv/DBAI-TR-98-20.ps.gz`

Planning: $\mathcal{K}$ input is stored in files with extension `.plan`.

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

SQL: SQL statements are stored in files with extension `.sql`.

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

Internal DLV Frontends: Diagnosis

- Built-in frontend of DLV (switches later)
- Abductive diagnosis, consistency-based (Reiter's) diagnosis [16]
- Different notions of explanations:

  single failure, subset minimal, no restriction
- Not fully implemented yet
- Further details: [4]; DLV homepage:

  `http://www.dbai.tuwien.ac.at/proj/dlv/DBAI-TR-98-20.ps.gz`

Internal DLV Frontends: Planning

- Action language $\mathcal{K}$ for describing actions and their effects
- Generate plans from initial state and goal description
- Described in [7, 6]

Example: Blocksworld problem

- **Background knowledge:**

```
BK = {block(a). block(b). block(c).
      location(table).
      location(L) :- block(L).}
```

- **Action description:**

```
fluents : on(B,L) requires block(B), location(L).
          occupied(B) requires location(B).
actions : move(B,L) requires block(B), location(L).
always :  executable move(B,L) if not occupied(B),
                                   not occupied(L), B <> L.
          inertial on(B,L).
          caused on(B,L) after move(B,L).
          caused - on(B,L1) after move(B,L), on(B,L1), L <> L1.
          caused occupied(B) if on(B1,B), block(B).
          noConcurrency.
```

- **Initial state & goal:**

```
initially : on(a,table). on(b,table). on(c,a).
goal :  on(c,b), on(b,a), on(a,table) ? (3)
```

Plan (length 3): $P = \langle \{\texttt{move(c,table)}\}; \{\texttt{move(b,a)}\}; \{\texttt{move(c,b)}\} \rangle$

External Frontends: Inheritance

- **Object**: Set of arbitrary DLP rules (and facts).
- **Program**: Hierarchy of Objects.
- Contradictions are solved according with the inheritance hierarchy in favor of rules appearing in more specific objects.
- See [2] for details

- The Frontend does not increase the expressive power of the language (everything can be translated into plain DLV programs), but is much stronger from a knowledge representation point of view.

Example: “Tweety”

A solution to the “Tweety” example using inheritance:

```
bird                { flies.  }
penguin :  bird     { −flies.  }
tweety :  penguin   { }
```

- Objects: `bird`, `penguin`, and `tweety`
- `tweety` $<$ `penguin` $<$ `bird`
- $-$`flies` overrides `flies` (no inconsistency!)
- The only model is $\{-$`flies`$\}$.

Inheritance: Modeling the “Yale Shooting” Problem

```
          inertia { alive(T1) :- alive(T), #succ(T,T1).
                    -alive(T1) :- -alive(T), #succ(T,T1).
                    loaded(T1) :- loaded(T), #succ(T,T1).
                    -loaded(T1) :- -loaded(T), #succ(T,T1).  }
domain : inertia { loaded(T1) :- load(T), #succ(T,T1).
                    -loaded(T1) :- shoot(T), loaded(T), #succ(T,T1).
                    -alive(T1) :- shoot(T), loaded(T), #succ(T,T1).
                    load(T) v wait(T) v shoot(T)
                        :- #int(T), T < #maxint.
                  }
  yale : domain  { -loaded(0).
                    alive(0).      }
```

Query: `-loaded(#maxint), -alive(#maxint)?`

Most of the articles below are available on the WWW.

Bibliography

[1] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, Cambridge, 2003.

[2] F. Buccafurri, W. Faber, and N. Leone. Disjunctive Logic Programs with Inheritance. *Journal of the Theory and Practice of Logic Programming*, 2(3), May 2002.

[3] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI) 2003*, pages 847–852, Acapulco, Mexico, Aug. 2003. Morgan Kaufmann Publishers.

[4] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The Diagnosis Frontend of the dlv System. *AI Communications – The European Journal on Artificial Intelligence*, 12(1–2):99–111, 1999.

[5] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving Using the DLV System. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

[6] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning, II: The $\mathtt{dlv}^{\mathcal{K}}$ system. *Artificial Intelligence*, 144(1-2):157–211, 2003.

[7] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity. *ACM Transactions on Computational Logic*, 5(2):206–263, Apr. 2004.

[8] T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining Answer Set Programming with Description Logics for the Semantic Web. In D. Dubois, C. Welty, and M.-A. Williams, editors, *Proceedings Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR 2004), June 2-5, Whistler, British Columbia, Canada*, pages 141–151. Morgan Kaufmann, 2004.

[9] W. Faber, N. Leone, and G. Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In J. J. Alferes and J. A. Leite, editors, *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Proceedings*, volume 3229 of *LNCS*, pages 200–212. Springer Verlag, 2004.

[10] W. Faber and G. Pfeifer. DLV homepage, since 1996. `http://www.dlvsystem.com/`.

[11] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

[12] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[13] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 2005. To appear. Available via `http://www.arxiv.org/ps/cs.AI/0211004`.

[14] I. Niemelä. The implementation of answer set solvers, 2004. Tutorial at ICLP 2004. Available at `http://www.tcs.hut.fi/~init/papers/niemela-iclp04-tutorial.ps.gz/`.

[15] T. C. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.

[16] R. Reiter. A Theory of Diagnosis From First Principles. *Artificial Intelligence*, 32:57–95, 1987.