Disjunctive Logic Programming: Knowledge Representation Techniques, Systems, and Applications

Francesco Calimeri

Department of Mathematics

University of Calabria

calimeri@mat.unical.it

http://www.mat.unical.it/calimeri

Thanks to Nicola Leone

http://www.mat.unical.it/~leone/

Topics

MAIN FOCUS:

- Knowledge Representation (KR) and Applications

GOAL:

- Getting a Powerful Tool for Solving Problems in a Fast and Declarative way

Disjunctive Logic Programming (DLP)

Disjunctive Datalog

Disjunctive Databases

Answer Set Programming (ASP)

Disjunctive Logic Programming (DLP)

- Simple, yet powerful KR formalism
- Widely used in AI
  - Incomplete Knowledge
- Able to represent complex problems not (polynomially) translatable to SAT
- A declarative problem specification is executable

What is DLP Good for? (Applications)

- Artificial Intelligence
  - Knowledge Representation
  - Diagnosis
  - Planning
- Emerging Applications Areas
  - Knowledge Management
  - Information Integration

DLP Advantages

- Sound theoretical foundation (Model Theory)
- Nice formal properties (clear semantics)
- **Real** Declarativeness
  - Rules Ordering, and Goal Orderings is Immaterial!!!
  - Termination is always guaranteed
- High expressive power

Drawbacks

- Computing Answer Sets is rather hard
- Very few solid and efficient implementations

  ...but this has started to change: DLV/GnT

Foundations of DLP: Syntax and Semantics

a bit boring, but needed....

getFunTomorrow :- resistToday.

(Extended) Disjunctive Logic Programming

Logic Programming extended with

- disjunction
- integrity constraints
- weak constraints
- integers, arithmetic, and comparison builtins
- default negation
- strong negation
- aggregate functions

Disjunctive Logic Programming

SYNTAX

Rule: $a_1 \vee \ldots \vee a_n$:- $b_1, \ldots, b_k$, not b_{k+1} , …, not b_m

Constraints: :- b_1 , …, b_k , not b_{k+1} , …, not b_m

Program: A finite Set ***P*** of rules and constraints.

- a_i b_i are atoms or strongly negated atoms (-p)
- variables are allowed in atoms' arguments

mother(P,S) v father(P,S) :- parent(P,S).

Facts

- A rule with an empty body is called a fact.
- A fact is therefore a rule with a True body.
- The implication symbol is omitted for facts

```
parent(eugenio, peppe) :- true.
parent(mario, ciccio) :- true.
```

equivalently written by

```
parent(eugenio, peppe).
parent(mario, ciccio).
```

- Facts must be true in any answer set!

Informal Semantics

Rule: $a_1 \vee \ldots \vee a_n$:- $b_1, \ldots, b_k$, not b_{k+1} , …, not b_m

If all the $b_1 \ldots b_k$ are true and all the $b_{k+1} \ldots b_m$ are false, then at least one among $a_1 \ldots a_n$ is true.

isInterestedinDLP(john) v isCurious(john) :- attendsDLP(john).
attendsDLP(john).

Two (minimal) models, encoding two plausible scenarios:

M1: {attendsDLP(john), isInterestedinDLP(john) }
M2: {attendsDLP(john), isCurious(john) }

Disjunction

is *minimal*

$a \vee b \vee c \Rightarrow \{ a \}, \{ b \}, \{ c \}$

actually *subset minimal*

$a \vee b.$
$a \vee c. \Rightarrow \{a\}, \{b,c\}$

but *not exclusive*

$a \vee b.$
$a \vee c. \Rightarrow \{a,b\}, \{a,c\}, \{b,c\}$
$b \vee c.$

Informal Semantics – ctd.

Constraints: :- b_1 , …, b_k , not b_{k+1} , …, not b_m

Discard interpretations which verify the condition

:- hatesDLP(john), isInterestedinDLP(john).

hatesDLP(john).

isInterestedinDLP(john) v isCurious(john) :- attendsDLP(john).

attendsDLP(john).

first scenario ({attendsDLP(john), isInterested(john) }) is discarded.

only one plausible scenario:

M: { attendsDLP(john), hatesDLP(john), isCurious(john) }

Integrity Constraints

When encoding a problem, its solutions are given by the models of the resulting program. Rules usually construct these models. *Integrity constraints* can be used to discard models.

:- $L_1, \ldots, L_n$.

means: discard models in which $L_1, \ldots, L_n$ are simultaneously true.

$a \vee b.$
$a \vee c.$ $\Rightarrow \{a,b\}, \{a,c\}, \{b,c\}$
$b \vee c.$

:- a. $\Rightarrow \{b, c\}$

(Formal) Semantics: Program Instantiation

Herbrand Universe, U_P= Set of constants occurring in program P
Herbrand Base, B_P= Set of ground atoms constructible from U_P and Pred.
Ground instance of a Rule R: Replace each variable in R by a constant in U_P
Instantiation ground(P) of a program P: Set of the ground instances of its rules.

Example: isInterestedinDLP(X) v isCurious(X) :- attendsDLP(X).
attendsDLP(john).
attendsDLP(mary).

U_P={ john, mary }

isInterestedinDLP(john) v isCurious(john) :- attendsDLP(john).
isInterestedinDLP(mary) v isCurious(mary) :- attendsDLP(mary).
attendsDLP(john).
attendsDLP(mary).

A program with variables is just a shorthand for its ground instantiation!

Interpretations and Models

Interpretation I of a program P: consistent set of (classical) atoms of P.

Atom q is true in I if q belongs to I; otherwise it is false. Literal not q is true in I if q is false in I; otherwise it is false.

Interpretation I is a MODEL for P if, for every R in P, the head of R is True in I, whenever the body of R is true in I

Semantics for Positive Programs

We assume now that Programs are ground (just replace P by ground(P)) and Positive (not - free)

I is an answer set for a positive program P if it is a minimal model (w.r.t. set inclusion) for P

-> Bodies of constraint must be false.

Example (Answer set for a positive program)

isInterestedinDLP(john) v isCurious(john) :- attendsDLP(john).

isInterestedinDLP(mary) v isCurious(mary) :- attendsDLP(mary).

attendsDLP(john).

attendsDLP(mary).

I_1 = { attendsDLP(john) } (not a model)

I_2 = { isCurious(john), attendsDLP(john), isInterestedinDLP(mary), isCurious(mary), attendsDLP(mary) } (model, non minimal)

I_3 = { isCurious(john), attendsDLP(john), isInterestedinDLP(mary), attendsDLP(mary) } (answer set)

I_4={ isInterestedinDLP(john), attendsDLP(john), isInterestedinDLP(mary), attendsDLP(mary) } (answer set)

I_5 = { isCurious(john), attendsDLP(john), isCurious(mary), attendsDLP(mary) } (answer set)

I_6={ isInterestedinDLP(john), attendsDLP(john), isCurious(mary), attendsDLP(mary) } (answer set)

Example (Answer set for a positive program) – ctd.

Let us ADD:

:- hatesDLP(john), isInterestedinDLP(john).

hatesDLP(john).

(same interpretations as before + hatesDLP(john))

I1 = { attendsDLP(john), hatesDLP(john) } (not a model)

I2 = { isCurious(john), attendsDLP(john), isInterestedinDLP(mary), isCurious(mary), attendsDLP(mary), hatesDLP(john) } (model, non minimal)

I3 = { isCurious(john), attendsDLP(john), isInterestedinDLP(mary), attendsDLP(mary) , hatesDLP(john) } (answer set)

I4={ isInterestedinDLP(john), attendsDLP(john), isInterestedinDLP(mary), attendsDLP(mary), hatesDLP(john) } (not a model)!!!

I5 = { isCurious(john), attendsDLP(john), isCurious(mary), attendsDLP(mary), hatesDLP(john) } (answer set)

I6={ isInterestedinDLP(john), attendsDLP(john), isCurious(mary), attendsDLP(mary), hatesDLP(john) } (not a model)!!!

Semantics for Programs with Negation

Consider general programs (with NOT)

The reduct or of a program P w.r.t. an interpretation I is the positive program P^I, obtained from P by

- deleting all rules with a negative literal false in I;
- deleting the negative literals from the bodies of the remaining rules.

An answer set of a program P is an interpretation I such that I is an answer set of P^I.

Answer Sets are also called Stable Models.

Example (Answer set for a general program)

P: a :- d, not b.
b :- not d.
d.

I = { a, d }

P^I : a :- d.
d.

I is an answer set of P^I and therefore it is an answer set of P.

Answer sets and minimality

An answer set is always a minimal model (also with negation).

In presence of negation minimal models are not necessarily answer sets

P: a :- not b.

Minimal Models: I1 = { a }

I2 = { b }

Reducts:

P^{I1} : a.

P^{I2} : {}

I1 is an answer set of P^{I1} while I2 is not an answer set of P^{I2} (it is not minimal, since empty set is a model of P^{I2}).

P^{I1} is the only answer set of P.

Knowledge Representation and Reasoning

Simple programs

road_free.
car_in_sight.

cross :- road_free.
stay :- car_in_sight.

--- --- ---

sunny.
cool.
want_to_play.

play_tennis :- sunny, cool.
play_tennis :- want_to_play.

Simple programs – ctd.

close_valve :- valve_open, temperature_1.
close_valve :- valve_open, temperature_2.
open_valve :- valve_closed, temperature_3.
open_valve :- valve_closed, temperature_4.

--- ---

have_money :- earn.
earn :- invest.
invest :- have_money.

(SUPPORT needed → Doesn't make you rich unless you earn from somewhere else, invest some other money or just have money already).

Transitive Closure

Suppose we are representing a graph by a relation *edge(X,Y).*

I want to express the query: *Find all nodes reachable from the others.*

path(X,Y) :- edge(X,Y).
path(X,Y) :- path(X,Z), path(Z,Y).

Recursion (ancestor)

If we want to define the relation of arbitrary ancestors rather than grandparents, we make use of recursion:

```
ancestor(A,B) :-  parent(A,B).
ancestor(A,C) :-  ancestor(A,B), ancestor(B,C).
```

An equivalent representation is

```
ancestor(A,B) :- parent(A,B).
ancestor(A,C) :- ancestor(A,B), parent(B,C).
```

Note the Full Declarativeness

The order of rules and of goals is immaterial:

```
ancestor(A,B) :-  parent(A,B).
ancestor(A,C) :-  ancestor(A,B), ancestor(B,C).
```

is fully equivalent to

```
ancestor(A,C) :-  ancestor(A,B), ancestor(B,C).
ancestor(A,B) :-  parent(A,B).
```

and also to

```
ancestor(A,C) :-  ancestor(B,C), ancestor(A,B).
ancestor(A,B) :-  parent(A,B).
```

NO LOOP!

Datalog Semantics

Though the semantics is the same as for DLP, we can have for Datalog the following Model Theoretic: *the answer to a datalog program is the least model of P (i.e. the unique minimal model).*

Why does this work?

THEOREM: A positive Datalog program has always a (unique) minimal model.

PROOF: The intersection of two models is guaranteed to be still a model; thus, only one minimal model exists.

Safety

A rule r is *safe* if

- each variable in the head, and
- each variable in a negated subgoal, and
- each variable in a comparison operator (<,<=, etc.)

also appears in a standard positive subgoal. Only safe rules are allowed.

Ex.: The following rules are unsafe:

- s(X) :- a.
- s(Y) :- b(Y), not r(X).
- s(X) :- not r(X).
- s(Y) :- b(Y), X<Y.

In each case, an infinity of *x*'s can satisfy the rule, even if "*r*" is a finite relation.

Arithmetic Built-ins

Fibonacci

```
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).
```

Unbound builtins

```
less(X,Y) :- #int(X), #int(Y), X < Y.
num(X) :- *(X,1,X).
```

Note that an upper bound for integers has to be specified.

Default Negation

Often, it is desirable to express negation in the following sense: “ If we do not have evidence that X holds, conclude Y.” This is expressed by *default negation* (the operator not).

For example, an agent could act according to the following rule:

“At a railroad crossing, cross the rails if no train approaches”

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

Strong Negation

However, in this example default negation is not really the right notion of negation.

It is possible that a train approaches, but that we don.t have any evidence for it (e.g. we do not hear the train). Rather, it would be desirable to definitely know that no train approaches.

- This concept is called *strong negation:*

  cross_railroad(A) :- crossing(A), -train_approaches(A).

- The use of strong negation can lead to *inconsistencies:*
- *a. -a.*

Example Disjunction

In a blood group knowledge base one may express that the genotype of a parent ***P*** of a person ***C*** is either ***T1*** or ***T2***, if ***C*** is heterozygot with types ***T1*** and ***T2***:

genotype(P,T1) $\vee$ genotype(P,T2) :-

parent(P,C), heterozygot(C,T1,T2).

In general, programs which contain disjunction can have more than one model.

A (Declarative) Methodology for Programming in DLP

DLP – How To Program?

Idea: encode a search problem ***P*** by a **DLP** program ***LP***.

The answer sets of ***LP*** correspond one-to-one to the solutions of ***P*** .

- Rudiments of methodology
- Generate-and-test programming:
- - Generate (possible structures)
- - Weed out (unwanted ones)
- by adding constraints (“Killing” clauses)
- Separate data from program

"Guess and Check" Programming
Answer Set Programming (ASP)

A disjunctive rule "guesses" a solution candidate.

Integrity constraints check its admissibility.

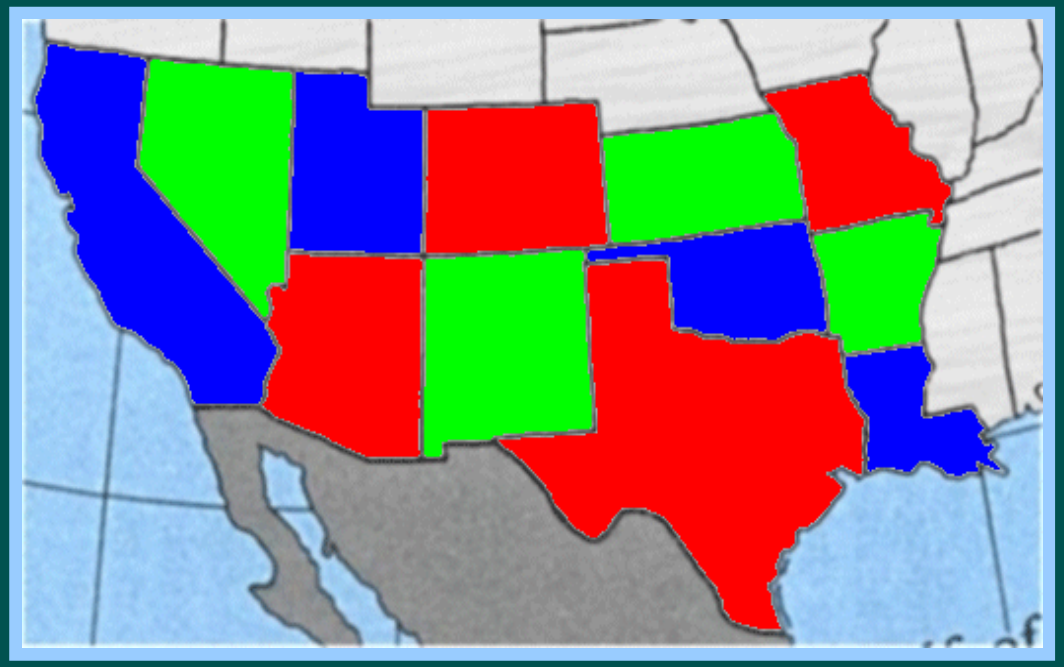
From another perspective:

- The disjunctive rule defines the search space.
- Integrity constraints prune illegal branches.

3-colorability

Input: a Map represented by state(_) and border(_,_).

Problem: assign one color out of 3 colors to each state such that two neighbouring states have always different colors.



Solution:

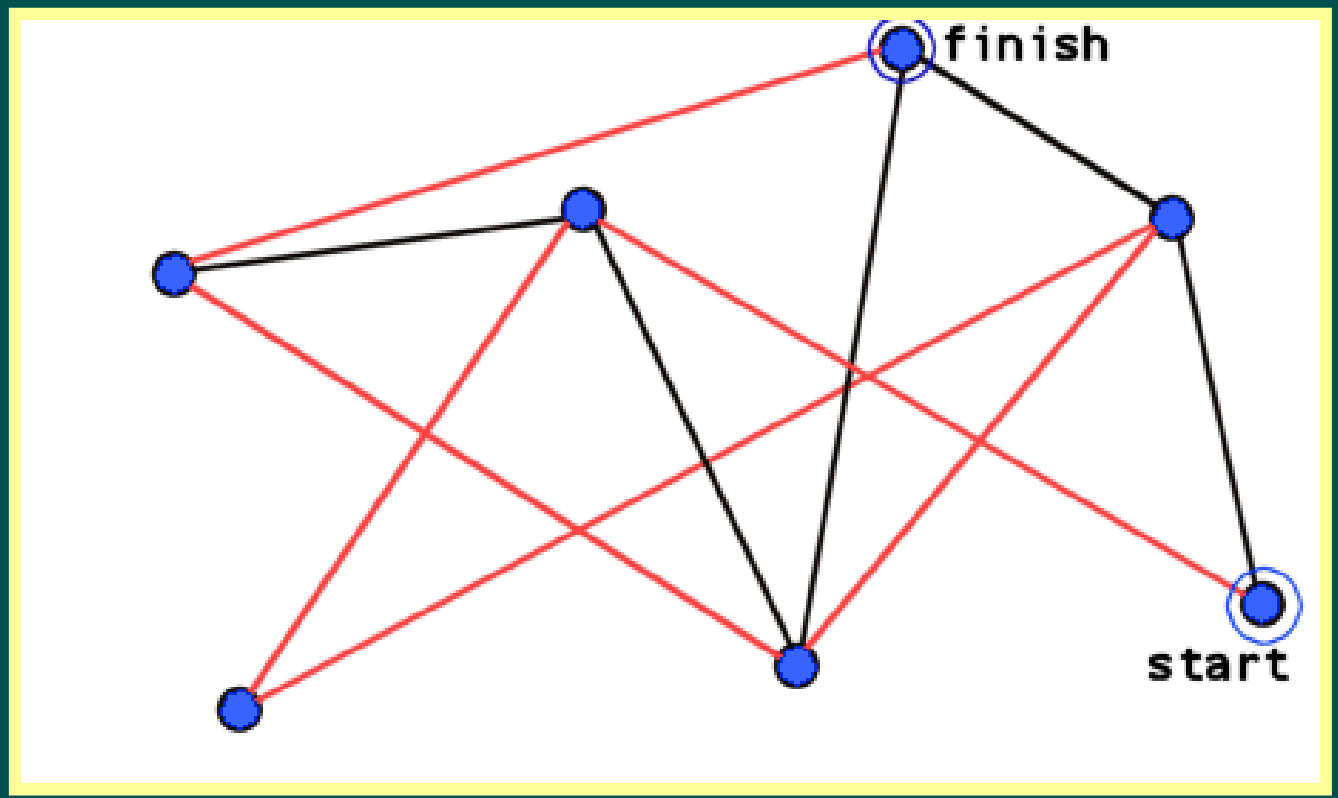
col(X,red) $\vee$ col(X,green) $\vee$ col(X,blue) :-state(X). } Guess

:- border(X,Y), col(X,C), col(Y,C). } Check

Hamiltonian Path (HP) (1)

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.



Hamiltonian Path (HP) (2)

inPath(X,Y) $\vee$ 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). **Auxiliary Predicate**
reached(X) :- reached(Y), inPath(Y,X).

Linguistic Extensions

- Aggregate functions
- Weak constraints

Aggregate Functions

Aggregate functions

emp(EmpId, Salary)

Compute the sum of salaries of the employees

- Easily expressed in SQL
- Representation in DLP rather unnatural
  - recursion needed to express Sum

Sum (DLP vs DLPA)

```
% Order employees by id
    precedes(X,Y) :- emp(X,_), emp(Y,_), X<Y.

% Define successor, first and last
    succ(X,Y) :- precedes(X,Y), not elementInMiddle(X,Y).
    elementInMiddle(X,Y) :- precedes(X,Z), precedes(Z,Y).
    first(X) :- emp(X,_), not hasPredecessor(X).
    last(X) :- emp(X,_), not hasSuccessor(X).
    hasPredecessor(X) :- succ(Y,X).
    hasSuccessor(Y) :- succ(Y,X).

% sum salaries recursively
    partialSum(X,Sx) :-  first(X), emp(X,Sx).
    partialSum(Y,S) :-  succ(X,Y), partialSum(X,PSx), emp(Y,Sy), S=PSx+Sy.

% select the total
    sum(S) :- last(L), partialSum(L,S).
```

$DLP^{\mathcal{A}}$ = DLP + aggregates

Symbolic set:

{ Vars : Conj }

{ EmpId : emp(EmpId, male, Skill, Salary) }

The set of ids of male employees.

Aggregate function

f{S}

S : symbolic set

f : function name among

{ #count, #sum, #times, #min, #max }

#count { EmpId : emp(EmpId, male, Skill, Salary) }

The number of male employees

Aggregate atom

$$Lg <_1 f\{S\} <_2 Ug$$

$$5 < \#count\ \{\ EmpId : emp(\ EmpId, Male, Skill, Salary\ )\ \} \leq 10$$

The atom is true if the number of male employees is greater than 5 and does not exceed 10.

Formal semantics: extension of the notion of answer set.

Aggregate Semantics

The *reduct* or ***Gelfond-Lifschitz transform*** of a ground program P w.r.t. a set $X \subseteq BP$ is the positive ground program P^X obtained from P by

1. deleting all rules $r \in P$ for which a negative literal in B(r) is false w.r.t. X or an aggregate literal is false w.r.t. X;

2. deleting the aggregate literals and the negative literals from the remaining rules.

An **answer set** of a program P is a set $X \subseteq BP$ such that X is an answer set of P^X.

Team Building

p_1 The team consists of a certain number of employees

p_2 At least a given number of different skills must be present in the team

p_3 The sum of the salaries of the employees working in the team must not exceed the given budget

p_4 The salary of each individual employee is within a specified limit

p_5 The number of women working in the team must be greater than a given number

```
in(I) v out(I) :- emp(I,Sx,Sk,Sa).

:- nEmp(N), not #count{ I : in(I) } = N.

:- nSkill(M), not #count{ Sk : emp(I,Sx,Sk,Sa), in(I) }  ≥ M.

:- budget(B), not #sum{ Sa, I : emp(I,Sx,Sk,Sa), in(I) } ≤ B.

:- maxSal(M), not #max{ Sa : emp(I,Sx,Sk,Sa), in(I) } ≤ M.

:- women(W), not #count{ I : emp(I,f,Sk,Sa), in(I) }  ≥ W.
```

Seating

Given some tables of nc chairs each, generate a sitting arrangement for a number of given guests.

People liking each other should sit at the same table.

People disliking each other should not sit at the same table.

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

:- table(T), not  #count{P : at(P,T)} <= nc.

:- guest(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).
```

Products control (unstratification)

p_1 A product A is produced by us if it is produced by a company under our control.
p_2 A company C is under our direct control, if we bought more than 50% of its shares.
p_3 A company C is under our (indirect) control, if companies under our control (together) own more than 50% of C.
p_4 The majority of the shares of C can be reached by summing up the C shares we bought directly with the shares owned by the companies under our control.
p_5 Each desired product has to be produced.
p_6 The budget must not be exceeded.

```
bought(C,N)  v  notBought(C,N) :- company(C), forSale(C,N, Price).
produced(A) :- producedBy(A,C), controlled(C).
controlled(C) :- bought(C,N), N > 50.
controlled(C) :- company(C),
                 #sum{ N, C1 : shares(C,C1,N), controlled(C1) } > 50.
controlled(C) :- bought(C,N), N ≤ 50,
                 #sum{ N, C1 : shares(C,C1,N), controlled(C1) } > K,
                 50 = K + N.
:- desired(P), not produced(P).
:- budget(B), #sum{ Price, C : forSale(C,N,Price), bought(C,N) } > B.
```

Related Works

- Many related works on DDBs
- #sum and #count similar to Smodels cardinality and weight constraints
- No negation in Smodels
- #min, #max, #times no counterpart in Smodels
- Pure sets vs multisets
- Smodels allows unstratification
  - Only recently introduced in DLP$^{\mathcal{A}}$
- Duplicated sets recognition in DLP$^{\mathcal{A}}$, other optimizations in Smodels
- Related theoretical work by Pelov

Weak Constraints: a Linguistic Extension to Encode Wishes

Weak Constraints

Express desiderata - constraints which should possibly be satisfied, like Soft Constraints in CSP

Syntax :~ B.

Satisfy B if possible.

Weak constraints can be weighted and prioritized:

:~ B. [w:p]

higher weights/priorities -> higher importance

A useful tool to encode optimization problems.

Semantics of Weak Constraints

Rules(P): set of the rules (including facts and strong constraints) of P

WC(P): weak constraints of P

Semantcs of programs without Priorities (in weak constraints):

Answer sets of Rules(P) minimizing the sum of the weights of the violated constraints in WC(P)

Semantics of programs with Priorities:

- minimize the sum of the weights of the violated constraints in the highest priority level;
- then minimize the sum of the weights of the violated constraints in the next lower level, etc.

Exams Scheduling

1. Assign course exams to time slots avoiding overlapping of exams of courses with common students

r_1: assign(X,s1) ∨ assign(X,s2) ∨ assign(X,s3) :- course(X).

s_1: :- assign(X,S), assign(Y,S), commonStudents(X,Y,N), N>0.

2. If overlapping is unavoidable, then reduce it "As Much As Possible" – Find an approximate solution

r_2: assign(X,s1) ∨ assign(X,s2) ∨ assign(X,s3) :- course(X).

w_2: :~ assign(X,S), assign(Y,S), commonStudents(X,Y,N), N>0. [N:]

Scenarios (models) that minimizes the total number of "lost" exams are preferred.

Team Building
(Prioritized Constraints)

Divide employees in two project groups p_1 and p_2.

A. Skills of group members should be different.

B. Persons in the same group should not be married each other.

C. Members of a group should possibly know each other.

Requirement A) is more important than B) and C)

```
assign(X,p1) ∨ assign(X,p2) :- employee(X).
:~ assign(X,P), assign(Y,P), same_skill(X,Y).          [:2]
:~ assign(X,P), assign(Y,P), married(X,Y).             [:1]
:~ assign(X,P), assign(Y,P), X<>Y, not know(X,Y).      [:1]
```

The GCO (Guess/Check/Optimize) Programming Technique

Generalization of the Guess and Check method to express optimization problems

A program is made of 3 Modules:

[Guessing Part] defines the search space

[Checking Part] checks solution admissibility

[Optimizing Part] specifies a preference criterion (by means of weak constraints)

Exams Scheduling (with GCO)

%Guess:

assign(X,s1) ∨ assign(X,s2) ∨ assign(X,s3) :- course(X).

%Optimize:

:~ assign(X,S), assign(Y,S), commonStudents(X,Y,N), N>0. [N:]

Team Building (with GCO)

assign(X,p1) ∨ assign(X,p2) :- employee(X). Guess

:~ assign(X,P), assign(Y,P), same_skill(X,Y). [:2] |
:~ assign(X,P),assign(Y,P), married(X,Y). [:1] | Optimize
:~ assign(X,P), assign(Y,P), X<>Y, not know(X,Y). [:1] |

The Traveling Salesman Person (TSP)*

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). |

:~ inPath(X,Y), arc(X,Y,C). [C:] Optimize

reached(X) :- start(X). **Auxiliary Predicate**
reached(X) :- reached(Y), inPath(Y,X).

* "Path version" of TSP.

Minimum Spanning Tree

Given a weighted graph by means of edge(Node1,Node2,Cost), and node(N), compute a tree that starts at a root node, spans that graph, and has minimum cost

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

%Check that we are really dealing with a tree!
:- root(X), inTree(_,X).                            |
:- inTree(X,Y), inTree(X1,Y), X <> X1.              |
%and the tree is connected                          | Check
:- node(X), not reached(X).                         |

%Minimize the cost of the tree
:~ inTree(X,Y), edge(X,Y,C).    [C:]                Optimize

reached(X) :- root(X).                              Auxiliary Predicate
reached(X) :- reached(Y), inTree(Y,X).
```

Testing and Debugging with GCO

Develop DLP programs incrementally:

- Design the Guess module G first
  - test that the answer sets of G (+the input facts) correctly define the search space
- Then the Check module C
  - verify that the answer sets of G U C are the admissible problem solutions
- Finally the Optimize module O
  - test that G U C U O generates the optimal solutions of the problem at hand.

Use small but meaningful problem test-instances!

DLV:
The state-of-the-art implementation of DLP

http://www.dlvsystem.com

(manual and tutorial on the website)

DLV: a KR System based on DLP

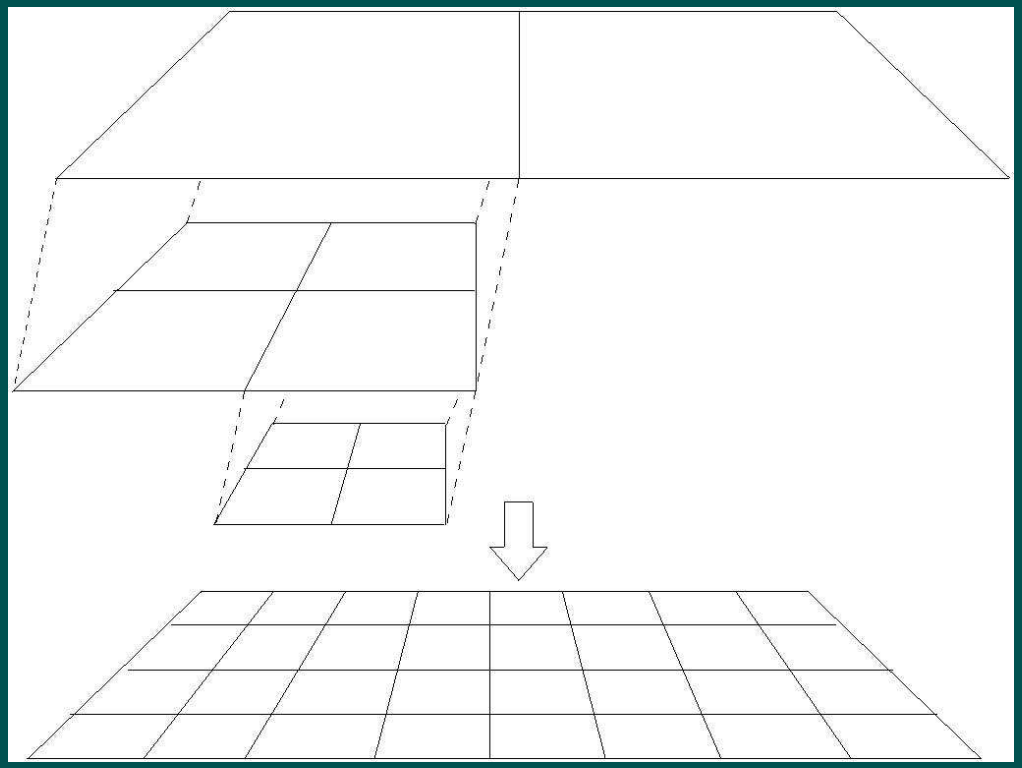
- Advanced knowledge modelling features
  - Extended DLP
  - Declarative “Guess/Check/Optimize” Programming Paradigm
  - Front-ends for specific AI Applications
- Solid Implementation
  - Implementation of DDB optimization techniques
  - Implementation of NMR optimization techniques
- Interfaced to Relational and Object-Oriented Databases

Applications

- The CMS project at CERN: An advanced deductive Database ApplicationCheck and Automatic Repair of Census Data
- Timetabling
- Education: Courses on Databases AI in European and American Universities
- Authorization Database Model
- “Implementation Engine” for KR purposes, experiments with new semantics and KR languages.

CERN DLV Example(1)

Projection of product data (assembly tree and part characteristics provided by PDMS) onto matrix of detector readout channels



EDB: product data & part composition members (description of relative location of immediate constituent parts of a composite part)

CERN DLV Example(2)

- Depth of assembly trees: 5 to 15
- ∃ integrity constraints (assembly finished?, manually inserted data feasible?)
- 85*20 (1700) readout channels out of totally 80000.

The goal of this projection is to assign product data to particle detector readout channels, which will collect observation data that will be used for the diagnosis of the correct functioning of the particle detector.

Timetabling

Structural constraints give rise to integrity constraints.

Weak Constraints express desiderata.

The teacher of Geometry doesn't like teaching in the afternoon.

```
:~  timetable(Day,Hour,geometry), Hour >12.  [1:1]
```

The teacher of French doesn't like teaching on Saturday.

```
:~  timetable(saturday,Hour,french).          [1:1]
```

Conclusion

- Easy representation of hard problems
- Possibility to solve problems unsolvable by SAT Checker or by other LP System (e.g., subset-minimal diagnosis, planning under incomplete knowledge)
- Front-end for other non-monotonic formalisms
- Interface to relational and object-oriented databases
- Used by researchers around the world
- Fully operational prototype available from

  http://www.dlvsystem.com/

Bibliography (1)

Foundations of DLP:

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

- J. Minker. On Indefinite Data Bases and the Closed World Assumption. In Proceedings 6th Conference on Automated Deduction (CADE '82), D.~Loveland, Ed. Number 138 in Lecture Notes in Computer Science. Springer, New York, 1982, pp. 292--308.

- N. Leone, P. Rullo, F. Scarcello, Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation, *Information and Computation*, Academic Press, New York, Vol. 135, N. 2, 15 June 1997, pp. 69-112.

Bibliography (2)

Knowledge Representation:

M. Gelfond, N. Leone, Logic Programming and Knowledge Representation --- the A-Prolog perspective. *Artificial Intelligence*, Elsevier, 138(1&2), June, 2002.

F. Buccafurri, N. Leone, P. Rullo, Enhancing Disjunctive Datalog by Constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5) Settembre/Ottobre 2000, pp 845-860.

Bibliography (3)

DLV System (Overviews):

T. Eiter, N. Leone, C. Mateis, G. Pfeifer, F. Scarcello, The Knowledge Representation System dlv: Progress Report, Comparisons, and Benchmarks, in *Proc. of KR'98*, pp. 406-417, Morgan Kaufman, 1998.

T. Eiter, W. Faber, N. Leone, G. Pfeifer, Declarative Problem-Solving Using the DLV System in *Logic in Artificial Intelligence*. J. Minker editor, Kluwer Academic Publisher, 2000, pp 79-103.

DLV Manual and Tutorial at http://www.dlvsystem.com

Bibliography (4)

DLV System (Algorithms and Optimizations):

W. Faber, N. Leone, G. Pfeifer, "Experimenting with Heuristics for Answer Set Programming". Proceedings of the 17th International Joint Conference on Artificial Intelligence -- IJCAI '01, Morgan Kaufmann Publishers, Seattle, USA, August 2001, pp. 635--640.

C. Koch, N. Leone, "Stable Model Checking Made Easy". Proceedings of the 16th International Joint Conference on Artificial Intelligence -- IJCAI '99, Morgan Kaufmann Publishers, pp. 70--75, Stockolm, August 1999.

N. Leone, S. Perri, F. Scarcello. "Improving ASP Instantiators by Join-Ordering Methods". Proceedings of the 6th International Conference on Logic Programming and Non-Monotonic Reasoning -- LPNMR'01, Lecture Notes in Artificial Intelligence (LNAI) 2173, Springer-Verlag, Vienna, Austria, 17--19 September 2001.

Exercises and DLV Lab

Homework (1)

Consider program

```
a :- b.
b.
```

and Interpretations:

I1 = { a }, I2 = { b }, I3 = { a, b }

which interpretations are models?

which interpretations are answer sets?

Homework (2)

Consider program

a :- b.

b :- not c.

and Interpretations:

I1 = { a }, I2 = { c }, I3 = { a, b }

which interpretations are models?

which interpretations are answer sets?

Homework (3)

Consider program

a :- b.

a v b.

and Interpretations:

I1 = { a }, I2 = { b }, I3 = { a, b }

which interpretations are models?

which interpretations are answer sets?

Homework (4)

Consider program

a :- b.

a v b.

:- not a.

what are the answer sets?

Homework (5)

Consider program

a :- b.

a v b.

:- a.

is there any answer set?

Homework (6)

Consider program

a :- b.

b :- a.

a v b.

and Interpretations:

I1 = { a }, I2 = { b }, I3 = { a, b }

which interpretations are models?

which interpretations are answer sets?

Homework (7)

Compute the ground instantiation of

```
p(X) :- q(X), not r(X).
q(a).
q(b).
r(a).
```

and determine the answer sets of the program.

Answer to Homework (7)

Instantiation:

p(a) :- q(a), not r(a).

p(b) :- q(b), not r(b).

q(a).

q(b).

r(a).

Answer sets: I = { p(b), q(a), q(b), r(a) }

Homework (8)

inPath(X,Y) $\vee$ outPath(X,Y) :- arc(X,Y).

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

reached(X) :- start(X).
reached(X) :- reached(Y), inPath(Y,X).

Transform the Hamiltonian Path program, to make sure that the start arc is not reached again (i.e., is not the endpoint of some arc in the path).

Answer to Homework (8)

inPath(X,Y) $\vee$ outPath(X,Y) :- arc(X,Y).

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

reached(X) :- start(X).
reached(X) :- reached(Y), inPath(Y,X).

:- start(X), inPath(Y,X).

Homework (9)

inPath(X,Y) $\vee$ outPath(X,Y) :- arc(X,Y).

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

reached(X) :- start(X).
reached(X) :- reached(Y), inPath(Y,X).

Transform the Hamiltonian Path program, in a program for Hamiltonian Cycle (make sure that the start arc is reached again, i.e., it is the end point of some arc in the path).
- Use the program to get the “tour version” of TSP.

Answer to Homework (9) (a safety problem)

WARNING:

:- start(X), not inPath(Y,X).

does not work.

It would require each node to be connected to the start!

Suppose that the graph has 3 nodes: a, b, c.
The above constraint then has 3 instances (disregarding those with start(b) or start(c)):

:- start(a), not inPath(b,a).
:- start(a), not inPath(c,a).
:- start(a), not inPath(a,a).

Answer to Homework (9) (safety)

LESSON:To avoid any problem, always use safe negation!

SAFETY: A rule R is safe if each variable appearing in R occurs also in a positive body literal of R.

```
p(X)   :- a(X,Y), not q(Y,Z).   (unsafe Z)
p(X,Y) :- not a(X).             (unsafe X,Y)
p(X)   :- a(X,Y), not q(_).     (unsafe _)
```

DLP systems anyway require safety.

Answer to Homework (9) first solution

endPoint(X) :- inPath(_,X).
:- start(X), not endPoint(X).

inPath(X,Y) $\vee$ outPath(X,Y) :- arc(X,Y).

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

reached(X) :- start(X).
reached(X) :- reached(Y), inPath(Y,X).

Answer to Homework (9)
a better solution

inPath(X,Y) $\vee$ outPath(X,Y) :- arc(X,Y).

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

reached(X) :- start(Y), inPath(Y,X).
reached(X) :- reached(Y), inPath(Y,X).

Homework (10)
(Node Cover)

Design a DLP program to represent the following problem.

Given a graph <V,E> by means of edge(Node1,Node2), and node(N), find a node cover, that is, a subset V' of V such that for each edge <u,v> in E at least one of u and v belongs to V'.

Answer to Homework (10) (Node Cover)

```
%Guess a set of nodes
inCover(X) V outCover(X) :- node(X).

%Check that all arcs are covered.
:-  edge(X,Y), not inCover(X), not inCover(Y) .
```

Homework (11)
(Minimum Node Cover)

Design a DLP program to represent the following problem.

Given a graph <V,E> by means of edge(Node1,Node2), and node(N), find a minimum node cover, that is, a subset V' of V of minimum cardinality such that for each edge <u,v> in E at least one of u and v belongs to V'.

Answer to Homework (11) (Minimum Node Cover)

```
%Guess a set of nodes
inCover(X) V outCover(X) :- node(X).

%Check that all arcs are covered.
:-  edge(X,Y), not inCover(X), not inCover(Y) .

% Prefer smaller covering
% Minimize the cardinality of the coverings
:~ inCover(X).
```

Running DLV

- Copy DLV to a dir which is in your PATH
- Or make “alias dlv DLV-DIR/dlv” in your .cshrc (or equivalent file).
- edit two files 3col and graph:

3col: col(X,green) v col(X,blue) v col(X,red) :- node(X).

:- edge(X,Y), col(X,C), col(Y,C), X<>Y.

graph:node(a). node(b). node(c). node(d).

edge(a,b).

edge(b,c).

edge(c,a).

edge(a,d).

edge(d,c).

Running DLV

```
> dlv 3col graph -filter=col -n=1

DLV [build DEV/Aug  1 2002   gcc 2.95.2 1999 1024 (release)]

{col(a,green), col(b,blue), col(c,red), col(d,blue)}

>
```