Idee di McCarthy sull'IA

- 1958, «Programs with Commonsense»
  - In questo lavoro si riconosce che lo sviluppo di un artefatto intelligente richiede di formalizzare il ragionamento di senso comune
- Il ragionamento di senso comune è non monotono
- Anni 80-90 del secolo scorso: definizione di formalismi logici adatti a rappresentare questo tipo di ragionamento
- Agli inizi degli anni 80 nasce la DLP (Jack Minker)

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

Nicola Leone

Department of Mathematics and Computer Science

University of Calabria

leone@unical.it

Topics

- Context and Motivation
- Datalog
- Theoretical Foundations of DLP
- Knowledge Representation and Applications
- Computational Issues
- DLP Systems
- ASP Development tools
- Exercises
- AI implementation: project (mandatory)

MAIN FOCUS:

- Knowledge Representation and Applications
- Examples, lot of Examples
- Lab (help of Francesco Calimeri and Simona Perri)

GOAL:

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

Roots – declarative programming

- First-order logic as a programming language
- Expectations, hopes
  - easy programming, fast prototyping
  - handle on program verification
  - advancement of software engineering

Disjunctive Logic Programming (DLP)

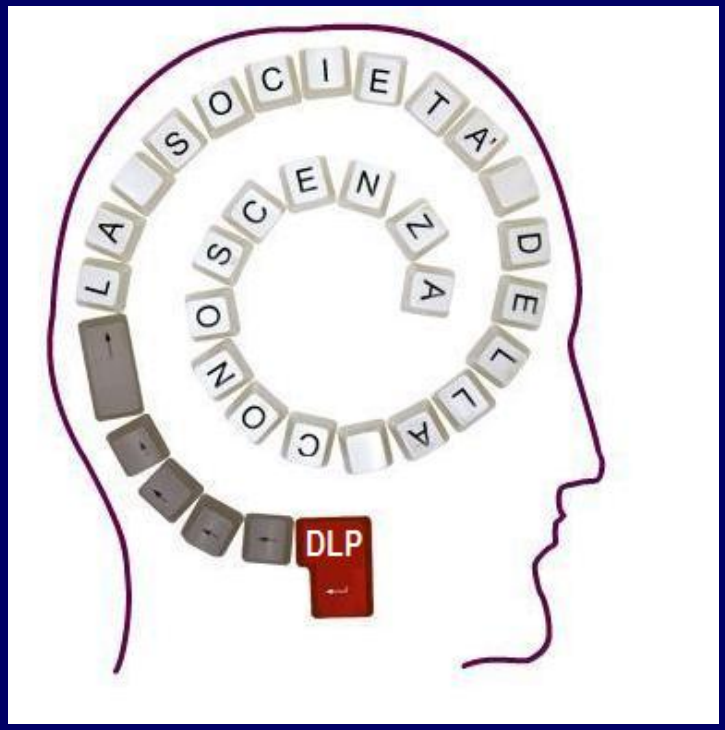
- Simple, yet powerful KR formalism
- Widely used in AI
  - Incomplete Knowledge
- Able to represent complex problems not (polynomially) translatable to SAT
- A declarative problem specification is executable

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 (Σ^P_2)

DLP Revolution

INTELLIGENT PROBLEM SOLVING



COMPLEX DATA/ *KNOWLEDGE* MANIPULATION

DLP Revolution

Why is DLP approach "revolutionary" ? :

DLP Declarative Programming

vs Traditional Procedural Programming

- ***Traditional PROGRAMMING (OLD):***
  - ***Implement an Algorithm to solve the problem***
  - ***List commands or steps that need to be carried out In order to achieve the results***
  - ***Tell the computer "HOW TO" solve the problem***

- **DLP DECLARATIVE PROGRAMMING**
  - ***Specify the features of the desidered solution***
  - ***NO ALGORITHMS***
  - ***Simply Provide a "Problem Specification"***

Drawbacks

- Computing Answer Sets is rather hard (Σ^P_2)
- Very few solid and efficient implementations

  ...but this has started to change:
  - DLV, Clasp, …
  - Cmodels, IDP, …

What is DLP Good for? (Applications)

- Artificial Intelligence, Knowledge Representation & Reasoning
- Information Integration, Data cleaning, Bioinformatics, ...
- Employed for developing industrial applications

Applications

- Planning
- Theory update/revision
- Preferences
- Diagnosis
- Learning
- Description logics and semantic web
- Probabilistic reasoning
- Data integration and question answering
- Multi-agent systems
- Multi-context systems
- Natural language processing/understanding

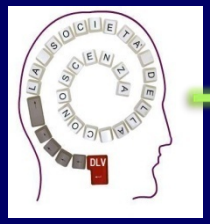
Applications

- Argumentation
- Product configuration
- Linux package configuration
- Wire routing
- Combinatorial auctions
- Game theory
- Decision support systems
- Logic puzzles
- Bioinformatics
- Phylogenetics
- Haplotype inference

Applications

- System biology
- Automatic music composition
- Assisted living
- Robotics
- Software engineering
- Boundend model checking
- Verification of cryptographic protocols
- E-tourism
- Team building
- Data Cleaning
- Business Games

DLP Revolution



TEAM BUILDING
at Seaport of Gioia Tauro

The Problem: producing an optimal allocation of the available personnel at the Seaport of Gioia Tauro

- *A Computationally Complex Problem* (NP-HARD)
- The complexity is due the presence of several constraints
  - the size and the slots occupied by cargo boats,
  - the allocation of each employee (e.g. each employee might be employed in several roles of different responsibility, roles have to be played by the available units by possibly applying a round-robin policy, etc.)
  - The choice of the suitable skills
  - Contractual/ labour union constraints

DLP Revolution

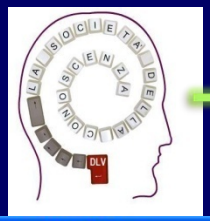
TEAM BUILDING
at Seaport of Gioia Tauro

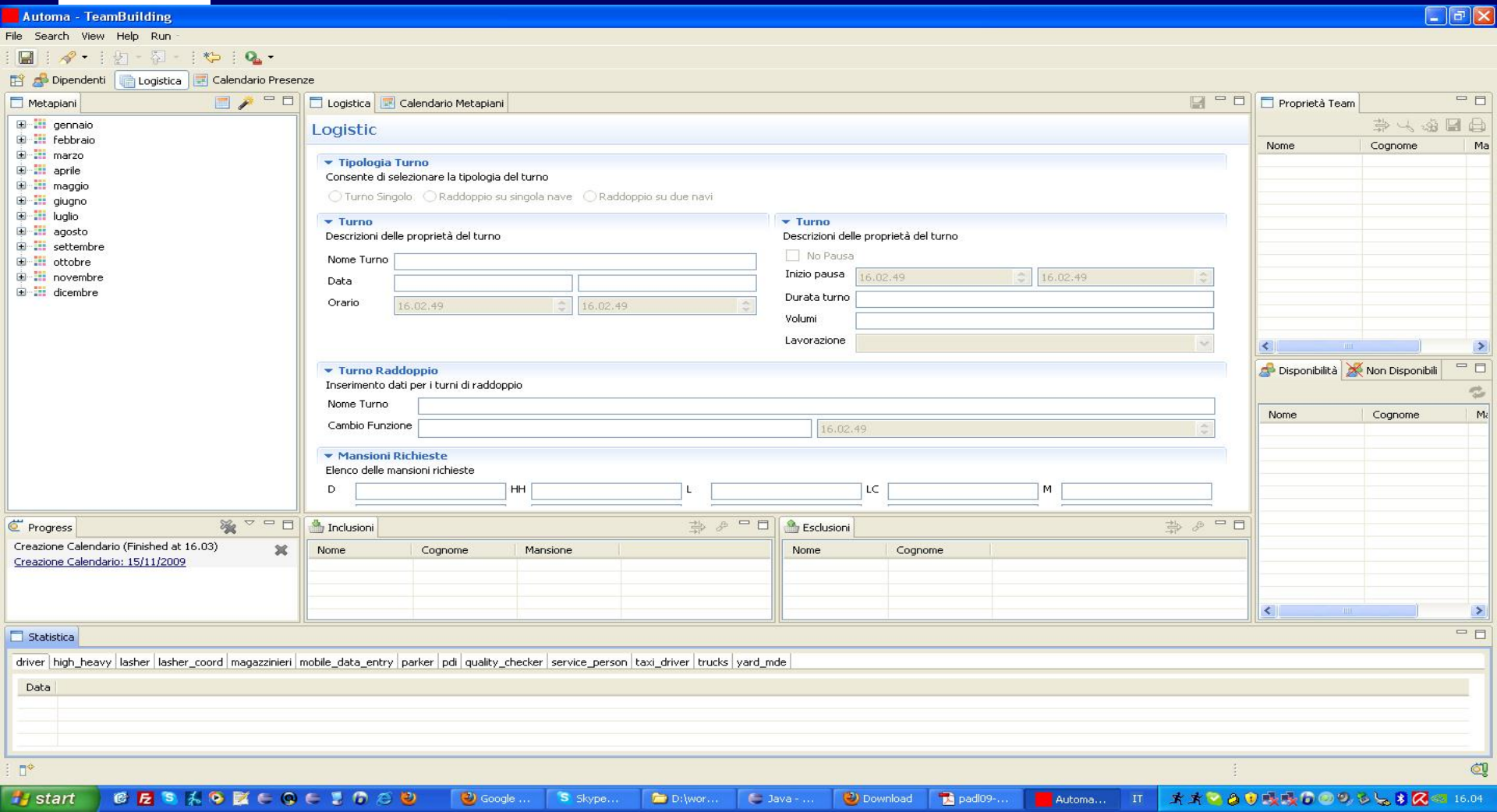
DLP Solution:

- **Informations and constraints of the domain are modeled in DLP.**
- **The *pure declarative nature* of DLP language allows to define reasoning modules for finding the desired allocation**
- **In a few seconds, the system can build new teams or complete the allocation automatically when the roles of some key employees are fixed manually.**
- **The port authority of Gioia Tauro is employing the system with great satisfaction**
- **The system has been implemented in two months with only one resource**
- **It is very flexible: can be modified in a few minutes, by addind/editing logic rules**

DLP Revolution

TEAM BUILDING
at Seaport of Gioia Tauro





DLP Revolution

→ Automatic Itinerary Search

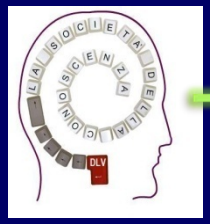
The Problem: automatic construction of a complete itinerary from a given place to another in the region Calabria.

DLP Solution:

- Implemented by exploiting an ONTOLOGY that models all the available transportation means, their timetables, and a map with all the streets, bus stops, railways and train stations
- A set of specifically devised DLP programs are used to build the required itineraries.
- **The system allows the selection of some options:**
  - **Departure and Arrival**
  - **Preferred mean**
  - **Preferred transportation company**
  - **Minimization of travel distances**
  - **Travel times**

- The application provides a web portal integrating the whole transportation system of the Italian region Calabria**,** including both public and private companies.

DLP Revolution



Automatic Itinerary Search

Trasporti

Dipartimento Organizzazione e Personale

Calcola il viaggio

Partenza: bivio fagnano castello (cs)

Destinazione: cosenza - autostazione (cs)

Orario: 06:00 - 12:00

Ora Partenza	Ora Arrivo	Durata	Cambi	Tipo	Dettagli
06:05	07:40	01:35	1		
06:05	08:00	01:55	1		
06:05	08:25	02:20	1		
06:05	08:40	02:35	1		
06:05	09:25	03:20	1		
06:05	09:25	03:20	1		
06:05	10:00	03:55	1		
06:05	10:00	03:55	1		
06:05	10:00	03:55	1		
07:30	11:59	04:29	1		
07:30	13:29	05:59	1		
06:05	14:10	08:05	1		
06:05	07:49	01:44	2		

Datalog

Datalog Syntax: Terms

- Terms are either constants or variables
- Constants can be either symbolic constants (strings starting with some lowercase letter), string constants (quoted strings) or integers.
  - Ex.: pippo, “this is a string constant”, 123, …
- Variables are denoted by strings starting with some uppercase letter.
  - Ex.: X, Pippo, THIS_IS_A_VARIABLE, White, …

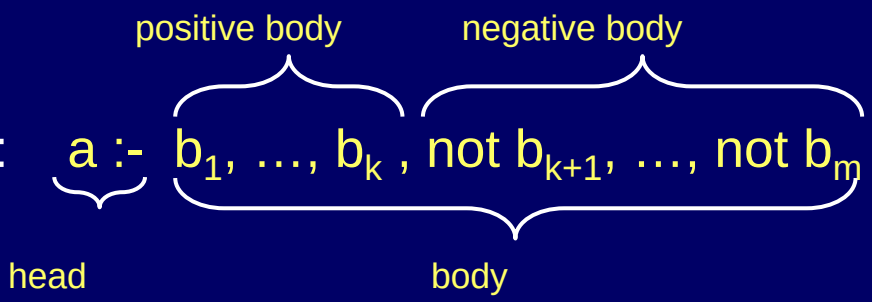
Datalog Syntax: Atoms and Literals

- A predicate atom has form $p(t_1,\ldots, t_n)$, where p is a predicate name, $t_1,\ldots, t_n$ are terms, and $n \geq 0$ is the arity of the predicate atom. A predicate atom p() of arity 0 is likewise represented by its predicate name p without parentheses.
  - Ex.: p(X,Y) - next(1,2) - q - i_am_an_atom(1,2,a,B,X)
- An atom can be negated by means of “not”.
  - Ex: not a, not p(X), …
- A literal is an atom or a negated atom. In the first case it is said to be positive, while in the second it is said to be negative.

What is Datalog (I)

Datalog is the *non-disjunctive* fragment of DLP.

A (*general*) Datalog program is a set of rules of the form



where "a" and each "b_i" are atoms.

Given a rule r of the form (1) above, we denote by:

- H(r): (head of r), the atom "a"
- B(r): (body of r), the set $b_1, \ldots, b_k$, not $b_{k+1}, \ldots,$ not b_m of all body literals
- $B^+(r)$: (positive body), the set $b_1, \ldots, b_k$ of positive body literals
- $B^-(r)$: (negative body), the set not $b_{k+1}, \ldots,$ not b_m of negative body literals

Positive Datalog

A *positive* (pure) Datalog rule has the following form:

head :- atom1, atom2, …., atom,…

where all the atoms are positive (non-negated).

Ex.: britishProduct(X) :- product(X,Y,P), company(P,“UK”,SP).

Facts

- A ground rule with an empty body is called a fact.
- A fact is therefore a rule with a True body (an empty conjunction is true by definition).
- 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 always be true in the program answer!

What is Datalog (II)

We usually distinguish *EDB* predicates and *IDB* predicates

- EDB: predicates appearing only in bodies or in facts.

EDB's can be thought of as stored in a database.

- IDB: predicates defined (also) by rules. IDB's are intensionally defined, appear in both bodies and heads.

Intuitive meaning of a Datalog program:

- Start with the facts in the EDB and iteratively derive facts for IDBs.

Datalog as a Query Language

Datalog has been originally conceived as a query language, in order to overcome some expressive limits of SQL and other languages.

Exercise: write an SQL query retrieving all the cities reachable by flight from Lamezia Terme, through a direct or undirect connection.

Input: A set of direct connections between some cities represented by facts for connected(_,_).

Datalog as a Query Language

Exercise (2): write an SQL query retrieving all the cities indirectly reachable by flight from Lamezia Terme, with a stop/coincidence in a single city.

Exercise (3): write an SQL query retrieving all the cities indirectly reachable by flight from Lamezia Terme, with exactly 2 stops/coincidences in other cities.

Datalog and RECURSION

(original) Exercise: write a query retrieving all the cities reachable by flight from Lamezia Terme, through a direct or undirect connection.

A possible Datalog solution.

Input: A set of direct connections between some cities represented by facts for connected(_,_).

```
reaches(lamezia,B) :- connected(lamezia,B).
reaches(lamezia,C) :- reaches(lamezia,B), connected(B,C).
```

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

Later on, we will give the model-theoretic semantics for DLP, and obtain model-theoretic semantics of Datalog as a special case.

We next provide the operational semantics of Datalog, i.e., we specify the semantics by giving a procedural method for its computation.

Semantics: Interpretations and Models

Given a Datalog program P, an interpretation I for P is a set of ground atoms.

An atom “a” is true w.r.t. I if $a \in I$; it is false otherwise.
A negative literal “not a” is true w.r.t. I if $a \notin I$; it is false otherwise.

Thus, an interpretation I assigns a meaning to every atom: the atoms in I are true, while all the others are false.

An interpretation I is a MODEL for a ground program P if, for every rule r in P, the H(r) is True w.r.t. I, whenever B(r) is true w.r.t. I

Example: Interpretations

Given the program

```
a :- b, c.
c :- d.
d.
```

and the interpretation

I = {c,d}

the atoms c and d are true w.r.t. I, while the atoms a and b are false w.r.t. I.

Example: Models

Given the program

r_1: a :- b, c.
r_2: c :- d.
r_3: d.

and the interpretations

I_1 = {b,c,d} I_2={a,b,c,d} I_3={c,d}

we have that I_2 and I_3 are models, while I_1 is not, since the body of r_1 is true w.r.t. to I_1 and the head is false w.r.t. I_1.

Operational Semantics: ground programs

Given a ground positive Datalog program P and an interpretation I, the immediate consequences of I are the set of all atoms “a” such that there exists a rule “r” in P s.t. (1) “*a” is the head of “r”,* and (2) *the body of “r” is true w.r.t. I.*

$$T_P(I) = \{ a \mid \exists\, r \in P \text{ s.t. } a = H(r) \text{ and } B(r) \subseteq I \}$$

where H(r) is the head atom, and B(r) is the set of body literals.

Example:
a :- b. c :- d. e :- a. I = {b} → $T_P(I)$ = {a}.

THEOREM: On a positive Datalog program P, Tp always has a least fixpoint coinciding with the least model of P.

Thus: Start with I={facts in the EDB} and iteratively derive facts for IDBs, applying Tp operator.
Repeat until the least fixpoint is reached.

Operational Semantics: general case (non-ground)

What to do when dealing with a non-ground program?

Start with the EDB predicates, i.e.: “whatever the program dictates”, and with all IDB predicates empty.

Repeatedly examine the bodies of the rules, and see what new IDB facts can be discovered taking into account the EDB *plus* all IDB facts derived until the previous step.

Operational Semantics: Seminaive Evaluation

Since the EDB never changes, on each round we get new IDB tuples only if we use at least one IDB tuple that was obtained on the previous round.

Saves work; lets us avoid rediscovering *most* known facts (a fact could still be derived in a second way…).

Resuming: a new fact can be inferred by a rule in a given round only if it uses in the body some fact discovered on the previous (last) round. But while evaluating a rule, *remember* to take into account also the rest (EDB + all derived IDB).

Operational Semantics: Derivation

Relation can be expressed intentionally through logical rules.

grandParent(X,Y) :- parent (X,Z), parent(Z,Y).
parent(a,b). parent(b,c).

Semantics: evaluate the rules until the *fixpoint* is reached:

Iteration #0: { parent(a,b), parent(b,c) }
Iteration #1: the body of the rule can be instantiated with "*parent(a,b)*", "*parent(b,c)*"
thus deriving { grandParent(a,c) }
Iteration #2: nothing new can be derived (it is easy to see that we derived only "grandParent(a,c)", and no rule having "grandParent" in the body is present). Nothing changes → we stop.

M= { grandParent(a,c), parent(a,b), parent(b,c) }

Operational Semantics: Ancestor

(i) ancestor(X,Y) :- parent (X,Z), parent(Z,Y).
(ii) ancestor(X,Y) :- parent (X,Z), ancestor(Z,Y).
parent(a,b). parent(b,c). parent(c,d).

Iteration #0: { parent(a,b), parent(b,c), parent(c,d) }

Iteration #1: { ancestor(a,c), ancestor(b,d) } (from rule (i))
- useless to evaluate rule (ii): no facts for "ancestor" are true.

Iteration #2:
- useless to evaluate rule (i): body contains only "parent" facts, and no new were derived at last stage;
- some "ancestor" facts were just derived, and "ancestor" appears in the body of rule (ii).

Thus we derive: { ancestor(a,d) } - *Note:* this is derived exploiting "*ancestor(b,d)*" but also "*parent(a,b)*", which was derived before last stage.

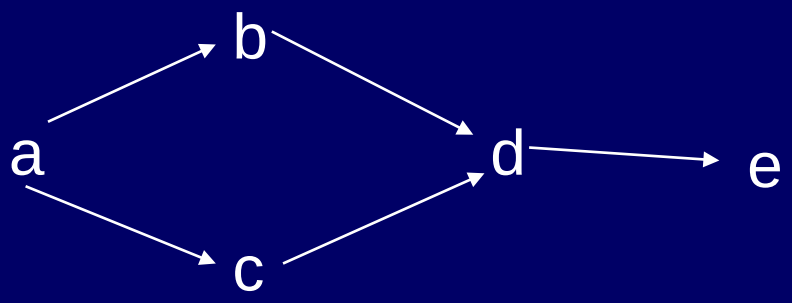
Iteration #3: nothing changes → we stop.

M= { parent(a,b), parent(b,c), parent(c,d), ancestor (a,c), ancestor(b,d), ancestor(a,d) }

Operational Semantics: Transitive Closure

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

edge(a,b). edge (a,c). edge(b,d).
edge(c,d). edge(d,e).



Iteration #0:	Edge: { (a,b), (a,c), (b,d), (c,d), (d,e) }
	Path: { }
Iteration #1:	Path: { (a,b), (a,c), (b,d), (c,d), (d,e) }
Iteration #2:	Path: { (a,d), (b,e), (c,e) }
Iteration #3:	Path: { (a,e) }
Iteration #4:	Nothing changes → We stop.

Note: number of iterations depends on the data. Cannot be anticipated by only looking at the rules!

Negated Atoms

We may put "*not*" in front of an atom, to negate its meaning.

Of course, programs having at least one rule in which negation appears aren't said to be *positive* anymore.

Example: Think of arc(X,Y) as arcs in a graph.

s(X,Y) singles out the pairs of nodes <a,b> which are not symmetric, i.e., there is an arc from *a* to *b*, but no arc from *b* to *a*.

```
s(X,Y) :- arc(X,Y), not arc(Y,X).
```

Safety

A rule r is *safe* if

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

also appears in a standard positive literal. In other words, all variables must appear at least once in the positive body.

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.

Problems with Negation and Recursion

Example:

IDB: p(X) :- q(X), not p(X).

EDB: q(1). q(2).

Iteration #0: q = {(1), (2)}, p = { }

Iteration #1: q = {(1), (2)}, p = {(1), (2)}

Iteration #2: q = {(1), (2)}, p = { }

Iteration #3: q = {(1), (2)}, p = {(1), (2)}

etc., etc. …

Recursion + Negation

"Naïve" evaluation doesn't work when there are negative literals.

In fact, negation wrapped in a recursion makes no sense in general.

Even when recursion and negation are separate, we can have ambiguity about the correct IDB relations.

Stratified Negation

Stratification is a constraint usually placed on Datalog with recursion and negation.

It rules out negation wrapped inside recursion.

Gives the sensible IDB relations when negation and recursion are separate.

Stratified Negation: Definition

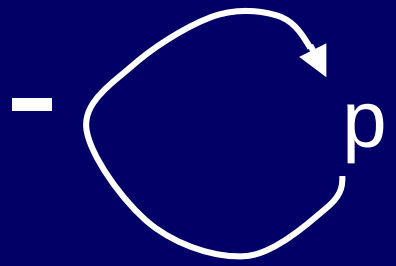
To formalize strata use the labeled *dependency graph*:

- Nodes = IDB predicates.
- Arc *b -> a* if predicate *a* depends on *b* (i.e., *b* appears in the body of a rule where *a* appears in the head), but label this arc "–" if the occurrence of *b* is negated.

A Datalog program is *stratified* if NO CYCLE of the labeled dependency graph contains an arc labeled "-".

Example: unstratified program

p(X) :- q(X), not p(X).



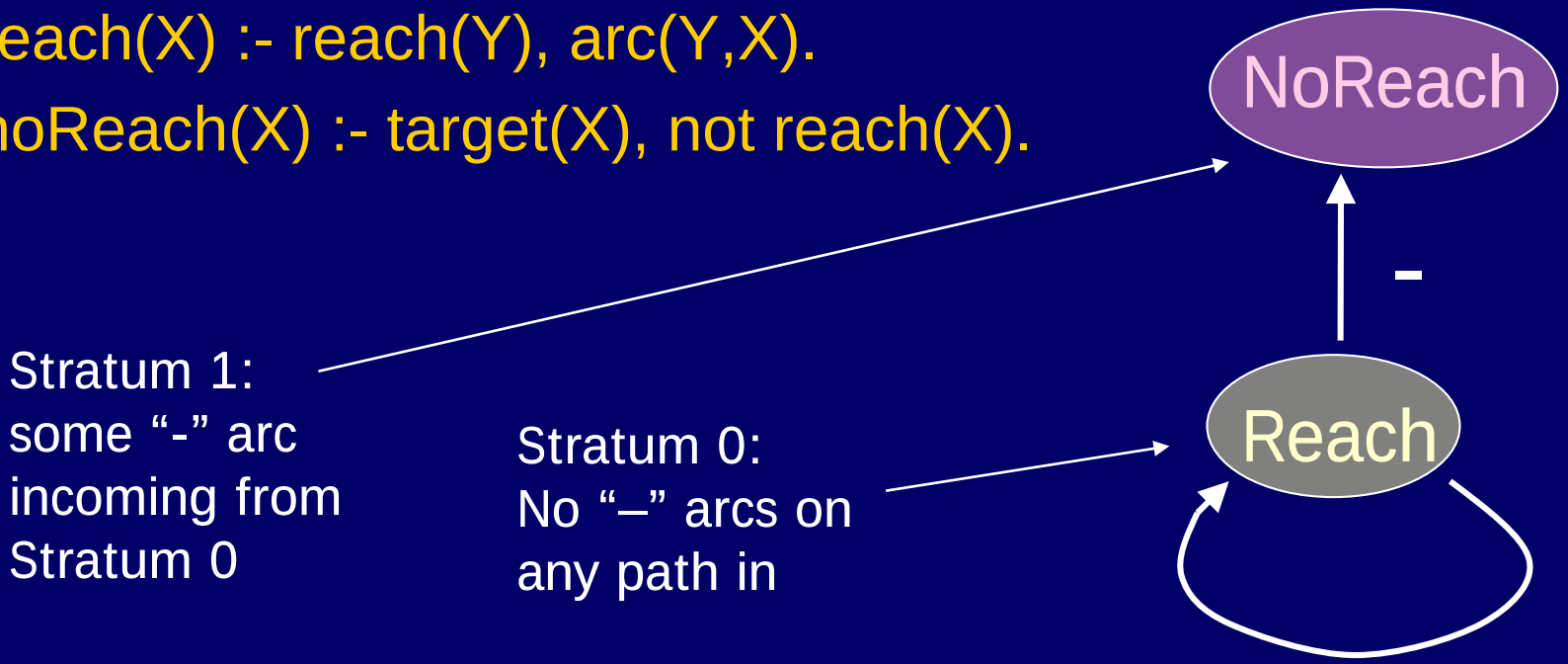
Unstratified: there is a cycle with a “-” arc.

Example: stratified program

EDB = source(X), target(X), arc(X,Y).

Define “targets not reached from any source”:

```
reach(X) :- source(X).
reach(X) :- reach(Y), arc(Y,X).
noReach(X) :- target(X), not reach(X).
```



Minimal Models

As already said, when there is no negation, a Datalog program has a unique minimal (thus minimum) model (one that does not contain any other model).

But with negation, there can be several minimal models.

Example: Multiple Models (1)

a :- not b.

Models: {a} {b}

Both are minimals. But stratification allows us to single out model {a}, which is indeed the (unique) answer set.

Subprograms

DEFINITION: Given a strongly-connected component *C* of the dependency graph of a given program *P*, the subprogram subP(C) is the set of rules with an head predicate belonging to *C*.

Evaluation of Stratified Programs 1

When the Datalog program is stratified, we can evaluate IDB predicates of the lowest-stratum-first.

Once evaluated, treat them as EDB for higher strata.

METHOD: Evaluate bottom-up the subprograms of the components of the dependency graph.

NOTE: The evaluation of a single subprogram is carried out by the (semi)NAÏVE method.

Evaluation of Stratified Programs 2

INPUT: EDB F, IDB P

- Compute the labeled dependency graph DG of P;
- Build a topological ordering C1,...,Cn of the components of DG;
- M= F;
- For i=1 To n Do
  - M = SemiNaive(*M* U *subP(Ci)*)
  - *% compute the least fixpoint of Tp on (M U subP(Ci))*
- OUTPUT M;

Stratified Model: example

a :- not b.

b :- d.

Two components: {a} and {b}.

subP({b}) = {b :- d.} subP({a}) = {a :- not b.}

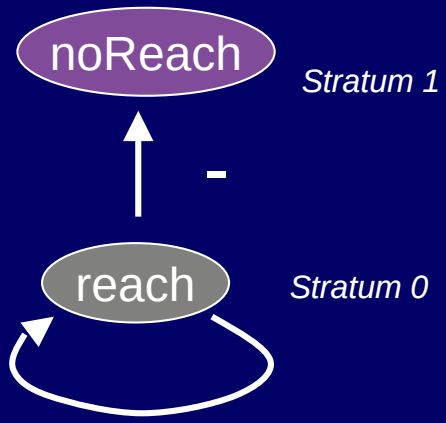
- {b} is at the lowest stratum -> start evaluating subP({b}).
- The answer set of subP({b}) is AS(subP({b})) = {}.

  → “{}” is the input for subP({a}).
- The answer set of subP({a}) U {} is AS(subP({a})) = {a}, which is the (unique) answer set of the original program.

Example: Stratified Evaluation (2-1)

IDB: reach(X) :- source(X).
reach(X) :- reach(Y), arc(Y,X).
noReach(X) :- target(X), not reach(X).

EDB: node(1). node(2). node(3). node(4).
arc(1,2), arc(3,4). arc(4,3)
source(1), target(2), target(3).



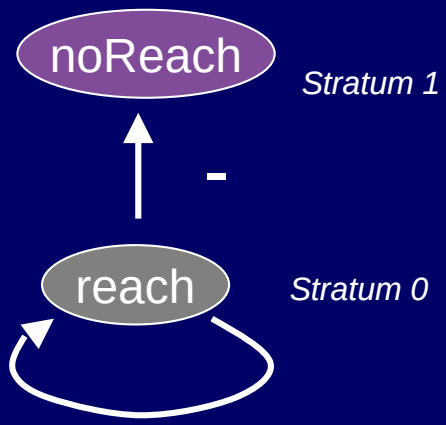
We have two components:
C1 = {reach} C2 = {noReach}

And the related subprograms are:
subP({reach}) = { reach(X) :- source(X).
reach(X) :- reach(Y), arc(Y,X). }
subP({noReach}) = { noReach(X) :- target(X), not reach(X). }

C1 is at a lower stratum w.r.t. C2, thus the subprogram of C1 has to be computed first.

Example: Stratified Evaluation (2-2)

IDB: reach(X) :- source(X).
reach(X) :- reach(Y), arc(Y,X).
noReach(X) :- target(X), not reach(X).
EDB: node(1). node(2). node(3). node(4).
arc(1,2), arc(3,4). arc(4,3)
source(1), target(2), target(3).



Answer Set of subP(C1) U EDB
Iteration #0: facts = { source(1), target(2), target(3),... }
Iteration #1: { reach(1) }
Iteration #2: { reach(2) }
Iteration #3: { } → we stop.

→ M(subP(C1)) = { reach(1), reach(2) + facts }

Answer Set of subP(C2) U M(subP(C1))
Iteration #0: M(subP(C1) = { reach(1), reach(2) + facts }
Iteration #1: { noReach(3) }
Iteration #2: { } → we stop.

→ M(subP(C2)) = { noReach(3), reach(1), reach(2) + facts }

Evaluating through strata →
Answer Set: { reach(1), reach(2), noReach(3), + facts }.

Disjunctive logic programming

Disjunctive Datalog

Answer Set Programming

Foundations of DLP: Syntax and Semantics

a bit boring, but needed....

getFunTomorrow :- resistToday.

(Extended) Disjunctive Logic Programming

Datalog extended with

- full negation (even unstratified)
- disjunction
- integrity constraints
- weak constraints
- aggregate functions
- function symbols, sets, and lists

Disjunctive Logic Programming

SYNTAX

Rule: $a_1 \mid \ldots \mid a_n$:- $b_1, \ldots, b_k$, 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
- variables are allowed in atoms' arguments

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

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) | genotype(P,T2) :-
                    parent(P,C), heterozygot(C,T1,T2).
```

In general, similar to programs with unstratified negation, programs which contain disjunction can have more than one minimal model.

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), #int(X).
```

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

Informal Semantics

Rule: $a_1 \mid \ldots \mid a_n$:- $b_1, \ldots, b_k$, 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) | 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 | b | c $\Rightarrow$ { a }, { b }, { c }

actually *subset minimal*

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

but *not exclusive*

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

Informal Semantics

Constraints: $:- b_1, \ldots, b_k, \text{not } b_{k+1}, \ldots, \text{not } b_m$

Discard interpretations which verify the condition

```
:- hatesDLP(john), isInterestedinDLP(john).
hatesDLP(john).
isInterestedinDLP(john) | 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 | b.
a | c. $\Rightarrow$ {a,b}, {a,c}, {b,c}
b | 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) | isCurious(X) :- attendsDLP(X).
attendsDLP(john).
attendsDLP(mary).

U_P={ john, mary }

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

set of ground 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 a ground program 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) | isCurious(john) :- attendsDLP(john).

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

attendsDLP(john).

attendsDLP(mary).

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

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

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

I4={ isInterestedinDLP(john), attendsDLP(john), isInterestedinDLP(mary), attendsDLP(mary) } (answer set)

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

I6={ isInterestedinDLP(john), attendsDLP(john), isCurious(mary), attendsDLP(mary) } (answer set)

Example (Answer set for a positive program)

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.

Some Useful Theorems

Datalog Semantics: a special case

The semantics of Datalog is the same as for DLP (Datalog programs are DLP programs).

Since Datalog programs have a simpler form, we can have for Datalog the following characterization:

- *the answer set of a positive datalog program is the least model of P*

  (i.e. the unique minimal model of P).

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.

Theorem 1

Let P be a logic program.

If the empty set ‘{}’ is an answer set for *P*, it is unique.

Example:

The logic program

```
a :- b.
```

has {} as unique answer set.

Theorem 2

Let *P* be a logic program, and *facts(P)* be the set containing all and only the facts in *P*.

If *S* is an answer set for *P*, then *facts(P)*$\subseteq$*S*.

Example:

The logic program

```
a | b :- c.
c.
d.
```

has {a,c,d} and {b,c,d} as answer sets, and both contain the facts {c,d}.

Theorem 3

Let *P* be a logic program, and *facts(P)* the set containing all and only the facts in *P*.

If *facts(P)* is an answer set, it is unique.

Example:

The logic program P

```
a | b :- c.
d.
e.
```

has {d,e} = *facts(P)* as unique answer set.

Definition

Let *P* be a logic program, and *I* an interpretation for *P*. An atom $a \in I$ is *supported* in *I* if there exists a rule $r \in P$ such that *body(r) is true* w.r.t. *I* and $head(r) \cap I = a$ *(i.e., a is the only true atom appearing in the head of r).*

Example:

P: d. c:-d. k :- not d. a | b :- c. e | c.

I = { d, c, k, a, e }

d, c, and a are supported w.r.t. I

k and e are not supported w.r.t. I

Theorem 4

Let *P* be a logic program, and *I* a model for *P*. *I* is an answer set for *P* only if for each atom $a \in I$, *a* is supported w.r.t. *I*.

Example:

For the logic program P

a | b :- c.

d.

e.

I={a,d,e} is a model, but *a* is not supported -> I is not an answer set.

Question

Is supportedness sufficient to guarantee that a model is an answer set?

That is, is every supported model an answer set?

The answer is NO!

Example:

P: a :- a.

I={a} is a supported model

I is not an answer set. {} is a model, I is not minimal.

Supportedness is not sufficient only in presence of CYCLES

Positive Dependency Graph

Positive *dependency graph*:

- Nodes = IDB predicates.
- Arc *b* -> *a* if predicate *a* depends on predicate *b* *positively* (i.e., *b* appears in the positive body of a rule featuring *a* in the head).

Note that negative literals do not produce any arc.

Example (DG) (1)

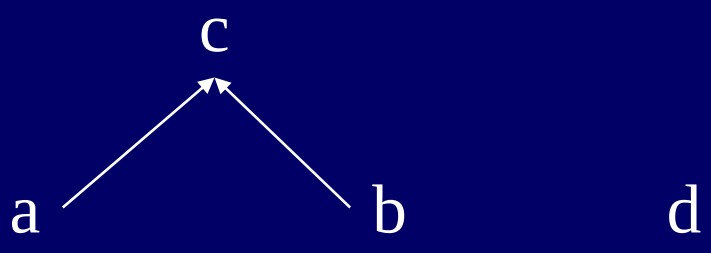
For the logic program P_1 :

a | b.

c :- a.

c :- b, not d.

the dependency graph is the following:



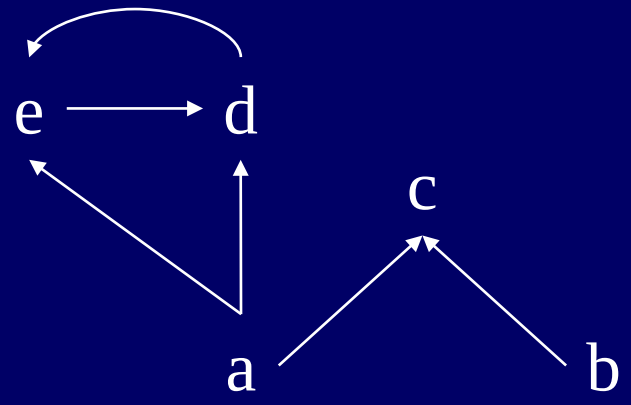
Example (DG) (2)

For the logic program P_2 :

a | b. c :- a. c :- b.

d | e :- a. d :- e. e :- d, not b.

the dependency graph is the following:



Definition (acyclic program)

Let *P* be a logic program. The program is acyclic iff its (positive) Dependency Graph is acyclic; it is cyclic otherwise.

Example

Consider the programs P_1 and P_2 from the previous examples.

P_1 is acyclic, while P_2 is not.

Theorem 5

Let *P* be an *acyclic* (i.e., non-recursive) logic program, and *I* a model for *P*. *I* is an answer set for *P if and only if* for each atom $a \in I$, *a* is supported w.r.t. *I*.

Example:

For the logic program P

a :- b.

b :- a.

I={a,b} is a model, and both atoms are supported, but since P is not acyclic this is not sufficient to guarantee that I is an answer set. Indeed, it is easy to see that *I* is not an answer set.

For the reader: Which are the answer sets for *P*?

Part II

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:

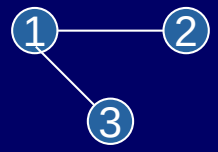
- The disjunctive rule defines the search space.
- Integrity constraints prune illegal branches.

Vertex Cover

Given a graph, select a subset S of the vertices so that all edges are covered (i.e., every edge has at least one of the two vertices in S)

Example: node(1). node(2). node(3). edge(1,2). edge(1,3).

Guess: inS(X) $\vee$ outS(X) :- node(X).



Eight answer sets, encoding the eight plausible scenarios:

inS = {1,2,3}, outS = $\varnothing$	AS1={inS(1), inS(2), inS(3)}
inS= {1,2}, outS = {3}	AS2={inS(1), inS(2), outS(3)}
…	...
inS= {1}, outS = {2,3}	AS7={inS(1), outS(2), outS(3)}
inS = $\varnothing$, outS = {1,2,3}	AS8={outS(1), outS(2), outS(3)}

Check: :- edge(X,Y), not inS(X), not inS(Y).

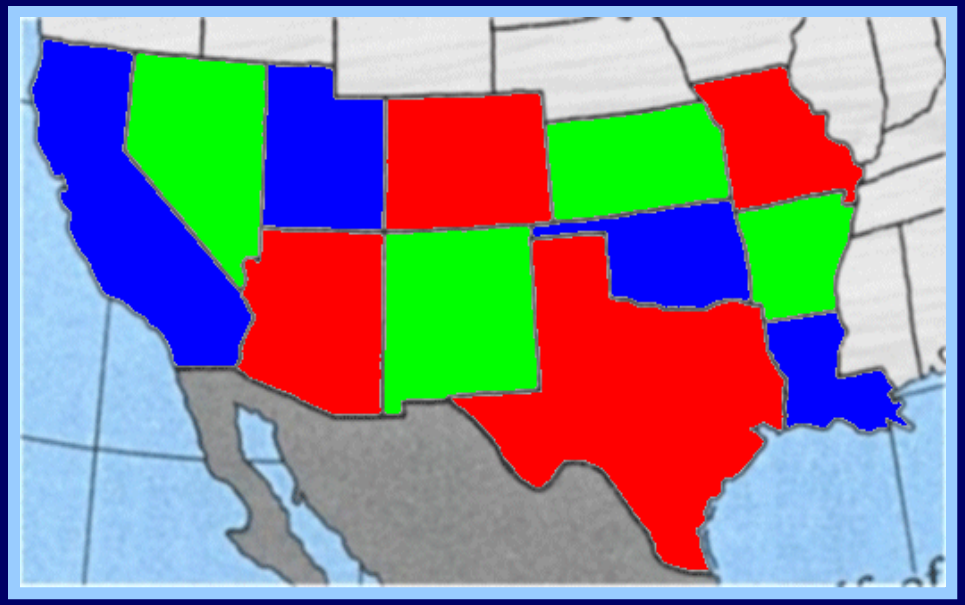
Discards:

{outS(1), outS(2),outS(3)}
{inS(2), outS(1), outS(3)},
{inS(3), outS(1), outS(2)}

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) | col(X,green) | col(X,blue) :-state(X).      } Guess
:- border(X,Y), col(X,C), col(Y,C).                       } Check
```

3-colorability

state(a). state(b). state(c). state(d).

border(a,b). border(b,a). border(a,c). border(c,a). border(c,d). border(d,c).

col(X,red) | col(X,green) | col(X,blue) :- state(X).

Instantiation:

col(a,red) | col(a,green) | col(a,blue) :- state(a).
col(b,red) | col(b,green) | col(b,blue) :- state(b).
col(c,red) | col(c,green) | col(c,blue) :- state(c).
col(d,red) | col(d,green) | col(d,blue) :- state(d).

Answer Sets:

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

3-colorability

state(a). state(b). state(c). state(d).

border(a,b). border(b,a). border(a,c). border(c,a). border(c,d). border(d,c).

col(X,red) | col(X,green) | col(X,blue) :- state(X).

Instantiation:
col(a,red) | col(a,green) | col(a,blue) :- state(a).
col(b,red) | col(b,green) | col(b,blue) :- state(b).
col(c,red) | col(c,green) | col(c,blue) :- state(c).
col(d,red) | col(d,green) | col(d,blue) :- state(d).

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

Instantiation:
:- border(a,b), col(a,red), col(b,red).
:- border(a,b), col(a,green), col(b,green).
:- border(a,b), col(a,blue), col(b,blue).
……

:- border(c,d), col(c,red), col(d,red).
:- border(c,d), col(c,green), col(d,green).
:- border(c,d), col(c,blue), col(d,blue).
……

3-colorability

Instantiation:

col(a,red) | col(a,green) | col(a,blue) :- state(a).
col(b,red) | col(b,green) | col(b,blue) :- state(b).
col(c,red) | col(c,green) | col(c,blue) :- state(c).
col(d,red) | col(d,green) | col(d,blue) :- state(d).

:- border(a,b), col(a,red), col(b,red).
:- border(a,b), col(a,green), col(b,green).
:- border(a,b), col(a,blue), col(c,blue).

:- border(c,d), col(c,red), col(d,red).
:- border(c,d), col(c,green), col(d,green).
:- border(c,d), col(c,blue), col(d,blue).

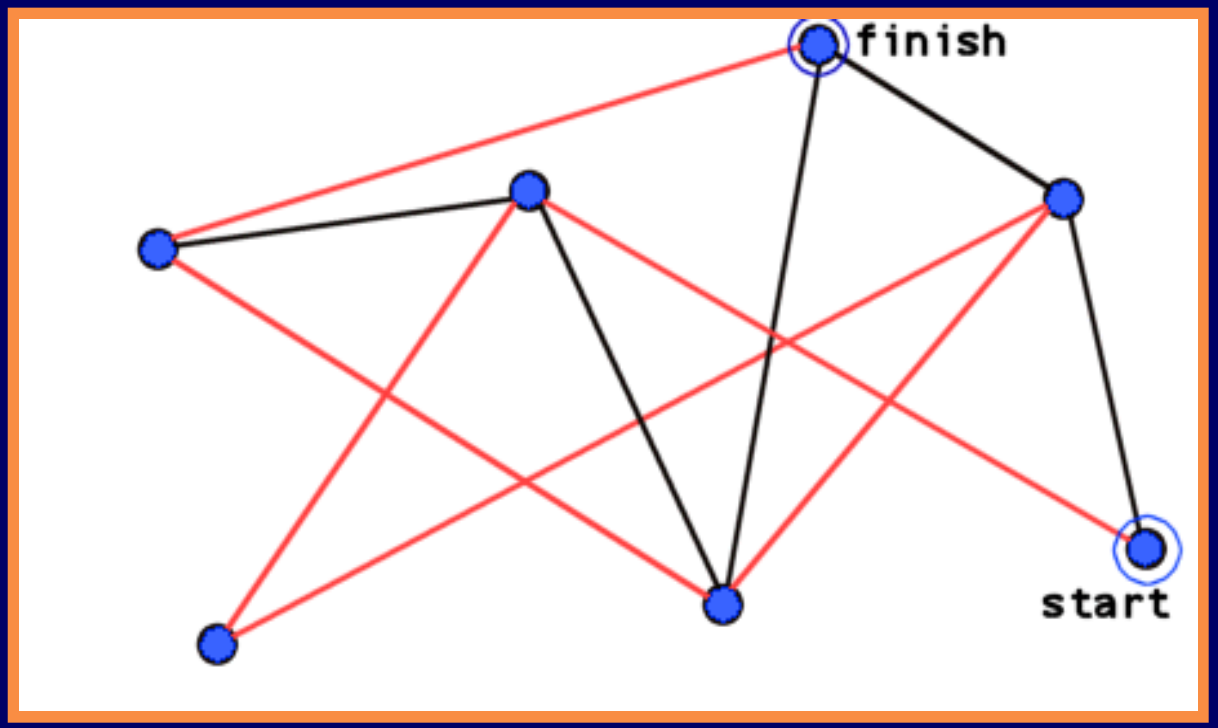
Answer Sets:

{ col(a,red), col(b,red), col(c,red), col(d,red) } NO
{ col(a,red), col(b,red), col(c,red), col(d,blue) } NO
……..
{ col(a,red), col(b,blue), col(c,blue), col(d,red) } YES
……..
{ col(a,green), col(b,green), col(c,green), col(d,green) } NO

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) | outPath(X,Y) :- arc(X,Y).
```

Guess

```
:- inPath(X,Y), inPath(X,Y1), Y <> Y1.
:- inPath(X,Y), inPath(X1,Y), X <> X1.
:- node(X), not reached(X).
:- inPath(X,Y), start(Y). % a path, not a cycle
```

Check

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

Auxiliary Predicate

Strategic Companies(1)

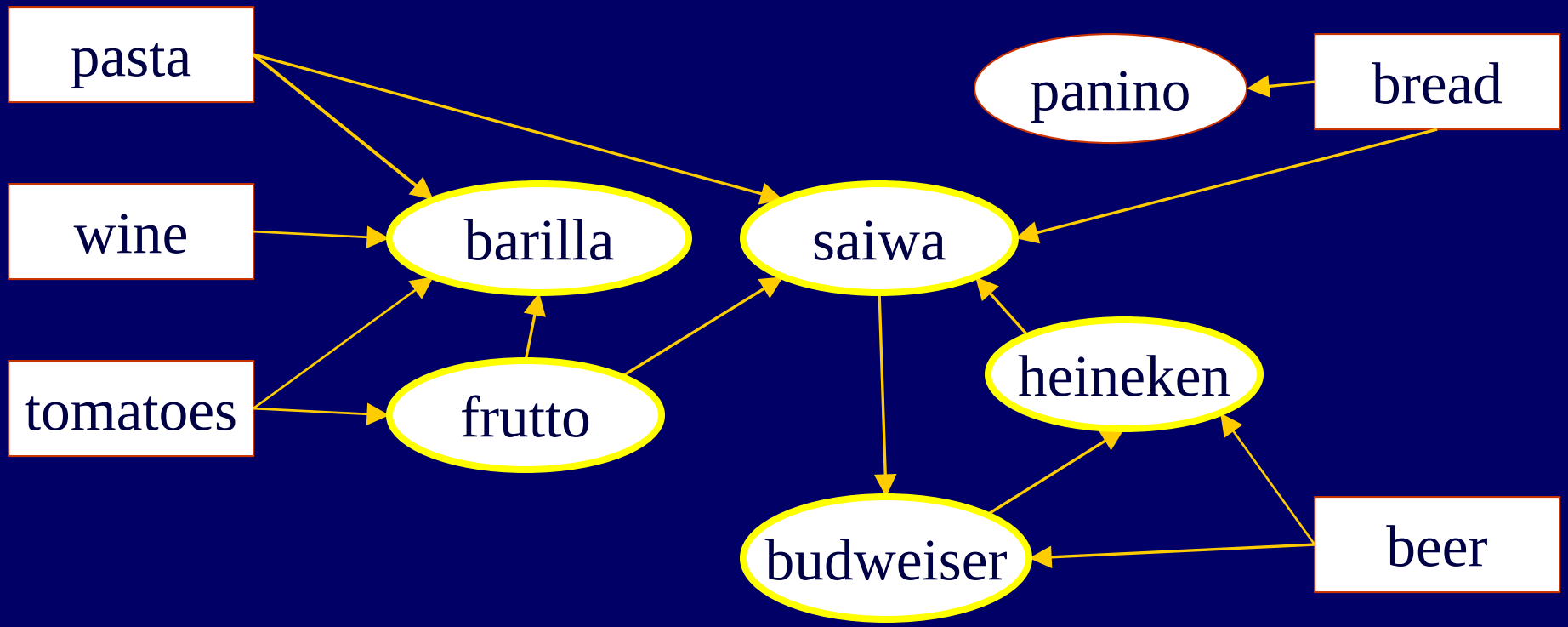
Input: There are various products, each one is produced by several companies.

Problem: We now have to sell some companies.
What are the minimal sets of *strategic companies*, such that all products can still be produced?
A company also belong to the set, if all its controlling companies belong to it.

```
strategic(Y) | strategic(Z) :- produced_by(X, Y, Z).
```
Guess

```
strategic(W) :- controlled_by(W, X, Y, Z),
                strategic(X), strategic(Y), strategic(Z).
```
Constraints

Strategic Companies - Example



Complexity Remark

The complexity is in **NP,** if the checking part does not “interfere” with the guess.

“Interference” is needed to represent Σ_2^P problems.

Testing and Debugging with GC

Develop DLP programs incrementally:

- Design the Data Model
  - The way the data are represented (i.e., design predicates and facts representing the input)
- 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

Use small but meaningful problem test-instances!

Satisfiability

- Boolean, or propositional, satisfiability (abbreviated SAT) is the problem of determining if there exists an interpretation that satisfies a given Boolean formula.
- Conjunctive Normal form (CNF): a formula is a conjunction of clauses, where a clause is a disjunction of boolean variables.

$$\Phi = \bigwedge_{i=1}^{n} (d_{i1} \vee \ldots \vee d_{ic_i})$$

- 3-SAT: only 3-CNF formulas (i.e. exactly three variables for each clause)

$$\Phi = \bigwedge_{i=1}^{n} (d_{i1} \vee d_{i2} \vee d_{i3})$$

- Problem: Find satisfying truth assignments of Φ (if any).

SAT: example

$(d_1 \vee -d_2 \vee -d_3) \wedge (-d_1 \vee d_2 \vee d_3)$

- Satisfying assignments:
  $\{ d_1, d_2, d_3\}$
  $\{ d_1, -d_2, d_3\}$
  $\{ d_1, d_2, -d_3\}$
  $\{-d_1, -d_2, d_3\}$
  $\{-d_1, -d_2, -d_3\}$
  $\{-d_1, d_2, -d_3\}$
- Non Satisfying assignments:
  $\{ d_1, -d_2, -d_3\}$
  $\{-d_1, d_2, d_3\}$

Exercise

Design a uniform (non-ground) encoding for SAT.

Input: a fact for each propositional clause

Example: $(d_1 \vee -d_2 \vee -d_3) \wedge (-d_1 \vee d_2 \vee d_3)$

→ clause(d_1,nd_2 , nd_3). clause(nd_1,d_2 ,nd_3).

Design a program such that the answer sets of the program are in a one-to-one correspondence with the satisfying assignements of the input formula.

SAT: ASP encoding

Add a guessing rule for each propositional variable

$$\forall \; d_i \;\; \rightarrow \;\; d_i \mid nd_i.$$

Add a constraint for each clause, complementing the variables

$$\forall \; d_{i1} \vee d_{i2} \vee d_{i3} \;\; \rightarrow \;\; :- \; L_{i1}, \; L_{i2}, \; L_{i3}$$

where $L_{ij} = a$ if $d_{ij} = -a$, and $L_{ij} = \text{not } a$ if $d_{ij} = a$

Example: SAT → ASP

Formula

$$(d_1 \vee \neg d_2 \vee \neg d_3) \wedge (\neg d_1 \vee d_2 \vee d_3)$$

ASP encoding:

- $d_1 \mid nd_1.$:- not d_1, d_2, d_3.
- $d_2 \mid nd_2.$:- d_1, not d_2, not d_3.
- $d_3 \mid nd_3.$

Answer Sets

{ d1, d2, nd3} {nd1, nd2, nd3}

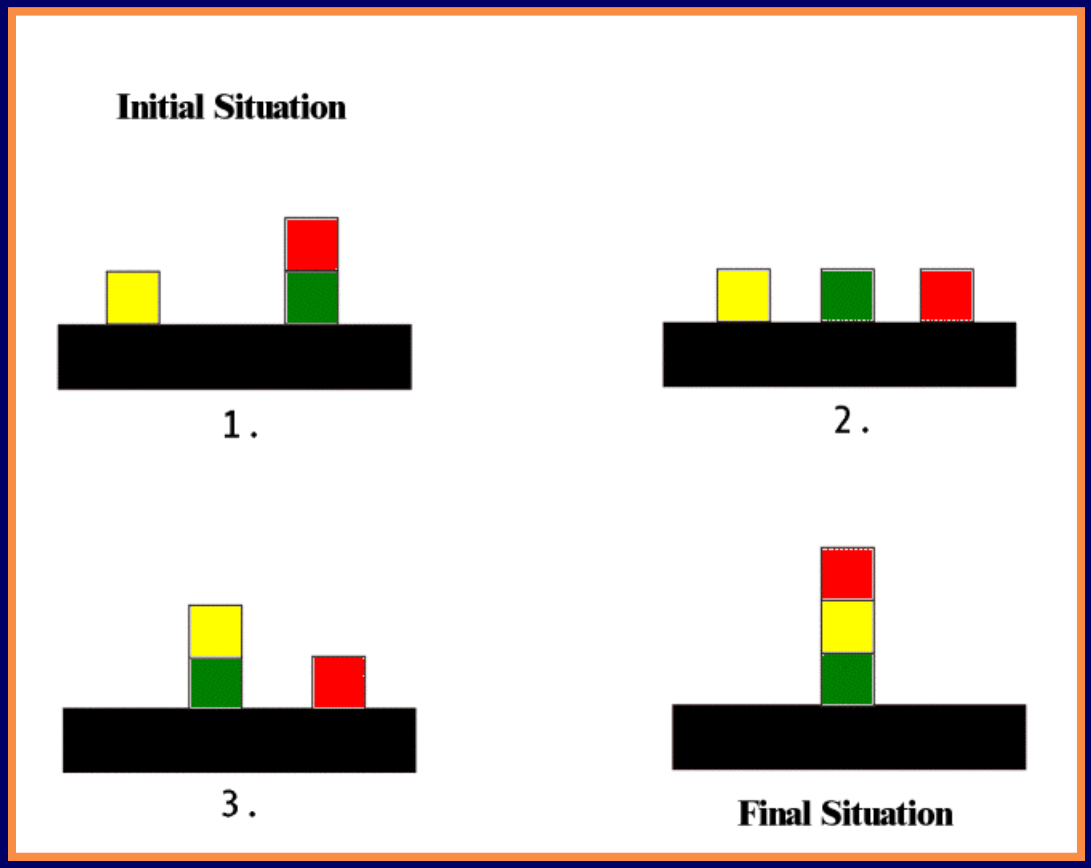
{nd1, d2, nd3} {nd1, nd2, d3}

{ d1, nd2, d3} { d1, d2, d3}

Planning - Blocksworld

- Objects: Some blocks and a table.
- Fluent: on(B, L, T): Exactly one block may be on another block, and arbitrary many block may be on the table. Every block must be on something.
- Action: move(B, L, T): A block is moved from one location to another one. The moved block must be clear and the goal location must be clear, if it is a block.
- Time: There is a finite number of timeslots. An action is carried out between two timeslots. Only one action can be carried out at a time.

Blocksworld(1)



Blocksworld(2)

Describe the move action and its effects:

move(B,L,T) | no_move(B,L,T) :- block(B), location(L), time(T).
on(B,L,T1) :- move(B, L, T), next(T,T1).

Enforce the preconditions for the move:

:- move(B,L,T), on(B1,B,T).
:- block(B1), move(B,B1,T), on(B2,B1,T).
:- move(B,L,T), lasttime(T).

No concurrency:

:- move(B,L,T), move(B1,L1,T), B<>B1.
:- move(B,L,T), move(B1,L1,T), L<>L1.

Blocksworld(3)

Inertia:

```
on(B,L,T1) :-   on(B,L,T), next(T,T1), not no_on(B,L,T1).
no_on(B,L,T1) :-  no_on(B,L,T), next(T,T1), not on(B,L,T1).
```

A block cannot be and not be on a location at the same time:

```
:- on(B,L,T), no_on(B,L,T).
```

A block cannot be at two locations or on itself:

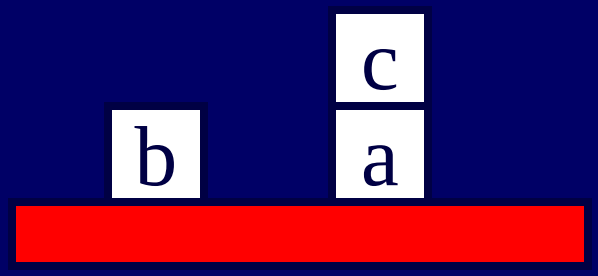
```
:- on(B,L1,T), on(B,L,T), L<>L1.
:- on(B,B,T).
```

Specification of time and objects:

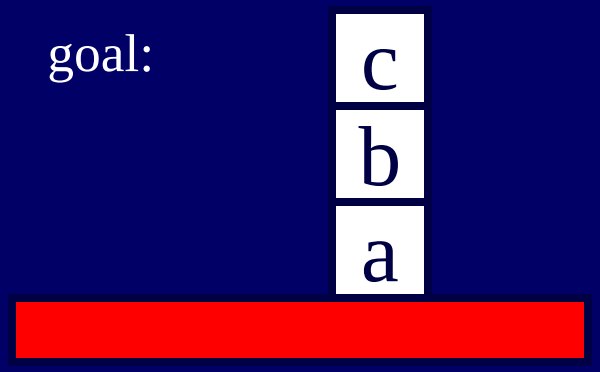
```
time(T) :-  #int(T).     lasttime(#maxint).
next(T,T1) :-  #succ(T,T1).
location(table).       location(L) :-  block(L).
```

Blocksworld Instance - Sussman anomaly

initial:



goal:



Define the involved blocks and the initial and goal situations:
block(a). block(b). block(c).

on(a,table,0). on(b,table,0). on(c,a,0).

on(a,table,#maxint), on(b,a,#maxint), on(c,b,#maxint) ?

The number of available timeslots is given when invoking DLV:
$ dlv blocksworld sussman -N=3 -pfilter=move

{move(c,table,0), move(b,a,1), move(c,b,2)}

Some programming tricks (efficiency)

Example: Clique

Given an indirected Graph compute a clique: a subset of the nodes such that no pair of nodes in the clique are not connected by an arc.

Input: node(_) and edge(_; _) (symmetric).

Natural Encoding:

```
inClique(X) | outClique(X) :- node(X).
:- inClique(X) | inClique(Y), not edge(X, Y), X <> Y.
```

Optimized Encoding:

```
inClique(X) | outClique(X) :- node(X).
:- inClique(X), inClique(Y), not edge(X, Y), X < Y.
```

Avoid redundant constraints

Example: 3-col – encoding 1

Encoding 1:

R1 col(X, red) | col(X, blue) | col(X, green) :- node(X).

R2 :- edge(X, Y), col(X,C), col(Y,C).

Keep the guess as small as possible

Example instance:

node(1). node(2). node(3).

edge(1,2). edge(1,3).

Grounding produces *(in addition to facts) :*

col(1,red) v col(1,blue) v col(1,green).

col(2,red) v col(2,blue) v col(2,green).

col(3,red) v col(3,blue) v col(3,green).

:- col(1,red), col(2,red).

:- col(1,blue), col(2,blue).

:- col(1,green), col(3,green).

:- col(1,green), col(2,green).

:- col(1,red), col(3,red).

:- col(1,blue), col(3,blue).

R1 → 3 ground rules

R2 → 6 ground rules

Example: 3-col - encoding 2

```
% given a node X and a color C, color X with C or not
R1    col(X, C) | ncol(X, C) :- node(X), color(C).
% no adjacent nodes with the same color
R2    :- edge(X, Y), col(X,C), col(Y,C).
% all nodes must be colored
R3    colored(X):- col(X,_).
R4    :- node(X), not colored(X).
% only one color per node
R5    :- col(X, C1), col(X,C2), C1<>C2.
```

Example instance:

```
node(1). node(2). node(3).
edge(1,2). edge(1,3).
color(red). color(blue). color(green).
```

Grounding produces:

- R1 → 9 ground rules
- R2 → 6 ground rules
- R3 → 9 ground rules
- R4 → 3 ground rules
- R5 → 18 ground rules

- Additional ground atoms (ncol)
- Additional ground rules

→ Larger grounding, Larger Search space

Example: 3-col - encoding 3

An alternative encoding can be obtained from encoding 2 by replacing the guessing rule R1 with the three following rules:

R1A col(X, red) | ncol(X, red) :- node(X).
R1B col(X, yellow) | ncol(X, yellow) :- node(X).
R1C col(X, green) | ncol(X, green) :- node(X).

and leaving the remaining rules unchanged.

Grounding produces exactly the same results of encoding 2
R1A + R1B + R1C → 9 ground rules

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).
```

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

Symbolic set: { Vars : Conj }

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

The set of ids of all employees.

{ 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, Sex, Skill, Salary) }

The number of all employees

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

The number of male employees

Aggregate atom

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

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

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 atoms: example (pure sets)

Aggregate sets are PURE (mathematical) sets → NO duplicates

Ex.: Count the number of different skills among employees

differentSkills(S) :- S = #count{Skill: emp(_,_, Skill,_)}

- emp(1, male, s1, 1000)
- emp(2, female, s3, 1000)
- emp(3, female, s2, 2000)
- emp(4, male, s3, 1500)

#count{<s1>, <s2>, <s3>}

Aggregate atoms: Dealing with Multisets

Sometimes we do want to consider duplicates.

Ex.: sum the salaries of all employees

sum(S) :- S = #sum{Y : emp(Id,_,_,Y)}

- emp(1, male, s1, 1000)
- emp(2, female, s3, 1000)
- emp(3, female, s2, 2000)
- emp(4, male, s3, 1500)

#sum{<1000>, ~~<1000>,~~ <2000>, <1500>} → 4500 instead of the (intended) 5500!!!

Aggregate atoms: Dealing with Multisets

Duplicates can be simulated by using a key as aggregation variable

sum(S) :- S = #sum{**Y,Id**: emp(Id,_,_,Y)}

- emp(1, male, s1, 1000)
- emp(2, female, s3, 1000)
- emp(3, female, s2, 2000)
- emp(4, male, s3, 1500)

#sum{<1000,1>, <1000,2>, <2000,3>, <1500,4>} → 5500, as expected

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

An organization needs to create a proper team for an important task, according to the following requirements:

- A team consists of a certain number of employees
- At least a given number of different skills must be present in the team
- The sum of the salaries of the employees working in the team must not exceed the given budget
- The salary of each individual employee is within a specified limit
- The number of women working in the team must be greater than a given number

Example: Team Building

```
% An employee is either included in the team or not
inTeam(I) | outTeam(I) :- emp(I, Sx, Sk, Sa).

% The team consists of a certain number of employees
:- nEmp(N), not #count{I: inTeam(I)} = N.

% At least a given number of different skills must be present in
% the team
:- nSkill(M), #count{Sk : emp(I, Sx, Sk, Sa), inTeam(I)} < M.

% The sum of the salaries of the employees working in the team
% must not exceed the given budget
:- budget(B). not #sum{Sa, I: emp(I, Sx, Sk, Sa), inTeam(I)} ≤ B.

% The salary of each individual employee is within a specified limit
:- maxSal(M), not #max{Sa: emp(I, Sx, Sk, Sa), inTeam(I)} ≤ M.
```

Team Building: Homework

% The salary of each individual employee is within a specified limit
:- maxSal(M), not #max{Sa: emp(I, Sx, Sk, Sa), inTeam(I)} ≤ M.

Efficiency issue: Rule above, from previous example, might actually be removed, by means of more sophisticated guess.

How?

Team Building: Homework Solution

Imposing that the max salary is below a given threshold can be enforced by guessing only among salaries that are under that threshold ➔ “Push” the requirement on salary in the body of the guessing rule.

```
% An employee is either included in the team or not
inTeam(I) | outTeam(I) :- emp(I, Sx, Sk, Sa), maxSal(M), Sa ≤ M.

% The team consists of a certain number of employees
:- nEmp(N), not #count{I: inTeam(I)} = N.

% At least a given number of different skills must be present in
% the team
:- nSkill(M), #count{Sk : emp(I, Sx, Sk, Sa), inTeam(I)} < M.

% The sum of the salaries of the employees working in the team
% must not exceed the given budget
:- budget(B). not #sum{Sa, I: emp(I, Sx, Sk, Sa), inTeam(I)} ≤ B.
```

Seating

A gala dinner has to be organized and table composition must satisfy a number of requirements:

- Each table T has a given number of chairs.
- Each guest must be assigned one and only one table.
- People liking each other should sit at the same table.
- People disliking each other should not sit at the same table.

Example: Seating

% INPUT facts
table(T,NC) ← *the set of available tables with corresponding seats*
guest(P) ← *the set of guests to be accommodated*
like(P1,P2) ← *couples of guests who are friends*
dislike(P1,P2) ← *couples of guests disliking each other*

% OUTPUT predicates
at(P,T) ← *guest P is accommodated to table T*

Example: Seating

```
% Given some tables of nc chairs each, generate a sitting
% arrangement for a number of given guests.
at(P,T) | not_at(P,T) :- guest(P), table(T,_).

% Each table must not host more than NC guests.
:- table(T,NC), not  #count{P : at(P,T)} <= NC.

%Each guest must be assigned one and only one table.
:- guest(P), not #count {T : at(P,T) }=1.

% People liking each other should sit at the same table.
:- like(P1,P2), at(P1,T), not at(P2,T).

% People disliking each other should not sit at the same table.
:- dislike(P1,P2), at(P1,T), at(P2,T).
```

Products control (unstratification)

```
bought(C,N)  |  notBought(C,N) :- company(C), forSale(C,N, Price).

% A product A is produced by us if it is produced by a company under our control.
produced(A) :- producedBy(A,C), controlled(C).

% A company C is under our direct control, if we bought more than 50% of its shares.
controlled(C) :- bought(C,N), N > 50.

% A company C is under our (indirect) control, if companies under our control
% (together) own more than 50% of C.
controlled(C) :- company(C), #sum{ N, C1 : shares(C,C1,N), controlled(C1) } > 50.

% 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.
controlled(C) :- bought(C,N), N  ≤ 50,
                 #sum{ N, C1 : shares(C,C1,N), controlled(C1) } > K, 50 = K + N.

% Each desired product has to be produced.
:- desired(P), not produced(P).

% The budget must not be exceeded.
:- budget(B), #sum{ Price, C : forSale(C,N,Price), bought(C,N) } > B.
```

Weak Constraints: a Linguistic Extension to Encode Wishes

Weak Constraints (DLV syntax)

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

Syntax `:~ B.`

Minimize the number of (instances of) violated weak constraints.

Weak Constraints (DLV syntax) (cont.)

Weak constraints can be weighted according to their importance (the higher the weight, the more important the constraint).

Syntax `:~ B. [ W : ]`

Minimize the sum of the weights of violated (instances of the) weak constraint.

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.

Weak Constraints (DLV syntax) (cont.)

Weak constraints can also be prioritized.

Syntax :~ B. [W : L]

Minimize the sum of the weights of the (instances of) violated weak constraints at the highest priority level first; then the lower priority levels are considered one after the other in descending order.

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

Weak Constraints: formal semantics

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.

Weak Constraints: ASP-Core-2 syntax (DLV2, I-DLV+wasp)

Syntax :~ B [W@P, $T_1 \ldots T_n$].

Satisfy B if possible; if not, pay W at priority P for each distinct tuple of terms $T_1 \ldots T_n$.

Weak Constraints: projecting

:~ p(X,Y). [1:1]

is equivalent to

:~ p(X,Y). [1@1, X,Y]

while

:~ p(X,Y). [1@1, X]

is different, and corresponds to

:~ q(X). [1:1]

q(X) :- p(X,Y).

Weak Constraints: projecting (cont.)

(a) :~ p(X,Y). [1@1, X] (equivalent to :~ q(X). [1:1]
q(X) :- p(X,Y).)

(b) :~ p(X,Y). [1@1, X,Y] (equivalent to :~ p(X,Y) [1:1])

With facts:
p(1,2). p(1,3).

(a) costs 1@1, while (b) costs 2@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@1, X,Y]
```

Team Building (with GCO)

```
% Guess
assign(X,p1) | assign(X,p2) :- employee(X).

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

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@1, X,Y,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@1, X,Y,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!

Part III

Computational Issues

Computational Issues

Problem: The complexity of DLP is very high (Σ^P_2 and even Δ^P_3), how to deal with that?

Tackle high complexity by isolating simpler sub-tasks

Tool: An in-depth Complexity Analysis

Main Decision Problems

[Brave Reasoning]

Given a DLP program P, and a ground atom A,
is A true in SOME answer sets of P?

[Cautious Reasoning]

Given a DLP program P, and a ground atom A,
is A true in ALL answer sets of P?

A relevant subproblem

[Answer Set Checking]

Given a DLP program P and an interpretation M, is M an answer set of Rules(P)?

Syntactic restrictions on DLP programs

- Head-Cycle Free Property

  [Ben-Eliyahu, Dechter]

- Stratification

  [Apt, Blair, Walker]

Level Mapping: a function || || from ground (classical) literals of the Herbrand Base B_P of P to positive integers.

Stratified Programs

Forbid recursion through negation.

P is (locally) stratified if there is a level mapping $||\ ||_s$ of P such that for every rule r of P

- For any l in Body+(r), and for any l' in Head(r), $||\ l\ ||_s <= ||\ l'\ ||_s$;
- For any not l in Body-(r), and for any l' in Head(r), $||\ l\ ||_s < ||\ l'\ ||_s$

Example: A stratified program

P1: p(a) | p(c) :- not q(a).
p(b) :- not q(b).

P1 is stratified:

$||p(a)||_s = 2$, $||p(b)||_s = 2$, $||p(c)||_s = 2$

$||q(a)||_s = 1$, $||q(b)||_s = 1$

Example: An unstratified program

P2: p(a) | p(c) :- not q(b).
q(b) :- not p(a)

P2 is not stratified,

No stratified level mapping exists,

as there is recursion through negation!

Stratification Theorem

- If a program P is stratified and V-free, then P has at most one answer set.
- If, in addition, P does not contain strong negation and integrity constraint, then P has precisely one answer set.
- Under the above conditions, the answer set of P is polynomial-time computable.

Head-Cycle Free Programs

P is head-cycle free if there is a level mapping $|| \; ||_h$ of P such that for every rule r of P:

- For any atom l in Body+(r), and for any l' in Head(r), $|| \; l \; ||_h <= || \; l' \; ||_h$;
- For any pair of atoms, l, l' in Head(r), $|| \; l \; ||_h <> || \; l' \; ||_h$

Example: an head-cycle free program

P3: a | b.

a :- b.

P3 is head-cycle free:

$|| a ||_h = 2;$ $|| b ||_h = 1$

Example: a non head-cycle free program

P4: a | b.

a :- b.

b :- a.

P4 is not head-cycle free

No head-cycle free level mapping exists, as there is recursion through disjunction!

Head-Cycle Free Theorem

Every head-cycle free program P is equivalent to an V-free program shift(P) where disjunction is “shifted” to the body.

P: a | b :- c.

shift(P): a :- c, not b.
b :- c, not a.

Complexity of Answer-Set Checking

	{}	not_s	not
{}	P	P	P
V_h	P	P	P
V	coNP	coNP	coNP

Complexity of Brave Reasoning

	{}	not_s	w	w,not_s	not	w, not
{}	P	P	P	P	NP	Δ^P_2
V_h	NP	NP	Δ^P_2	Δ^P_2	NP	Δ^P_2
V	Σ^P_2	Σ^P_2	Δ^P_3	Δ^P_3	Σ^P_2	Δ^P_3

Completeness under Logspace reductions

Intuitive Explaination

Three main sources of complexity:

1. the exponential number of answer set “candidates”
2. the difficulty of checking whether a candidate M is an answer set of Rules(P) (the minimality of M can be disproved by exponentially many subsets of M)
3. the difficulty of determining the optimality of the answer set w.r.t. the violation of the weak constraints

The absence of source 1 eliminates both source 2 and source 3

Complexity of Cautious Reasoning

	{}	not_s	w	w,not_s	not	w, not
{}	P	P	P	P	coNP	Δ^P_2
V_h	coNP	coNP	Δ^P_2	Δ^P_2	coNP	Δ^P_2
V	coNP	Π^P_2	Δ^P_3	Δ^P_3	Π^P_2	Δ^P_3

Note that < V, {} > is “only” coNP-complete!

Complexity of aggregates

(Brave Reasoning) Given a DLP$^{\mathcal{A}}$ program $\boldsymbol{P}$ and a ground atom L, is L true in SOME answer set of $\boldsymbol{P}$?

(Cautious Reasoning) Given a DLP$^{\mathcal{A}}$ program $\boldsymbol{P}$ and ground atom L, is L true in ALL answer sets of $\boldsymbol{P}$?

Theorem Brave Reasoning on ground DLP$^{\mathcal{A}}$ programs is $\sum_2^p$-complete

Theorem Cautios Reasoning on ground DLP$^{\mathcal{A}}$ programs is coNP-complete

Part IV

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

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

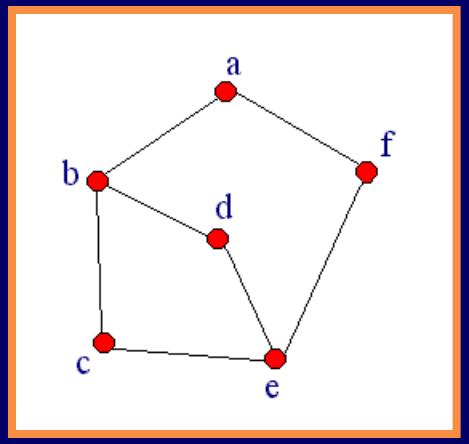
Frontends

- Diagnosis
- Planning
- Inheritance
- Meta-Interpreter
- SQL3
- External Frontends

Diagnosis Frontend

The frontend can handle Abductive Diagnosis and Consistency - Based Diagnosis.

Example - Diagnosing a computer network:



We work at computer **a** and cannot reach computer **e**.

Diagnosis Frontend II

Theory

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).

Inheritance Frontend

Object: Set of **DLP** rules.
Program: Hierarchy of Objects.

bird { flies }
penguin : bird { - flies.}
tweety : penguin { }

Objects: bird, penguin, and tweety. tweety < penguin < bird.
Contradictions are solved according with the inheritance hierarchy:

-flies overrides flies.

The only model is { -flies}.

SQL3 Frontend - Bill of materials

The forthcoming ANSI SQL3 will include support for computing transitive closures.

SCHEMA consists_of(major,minor);

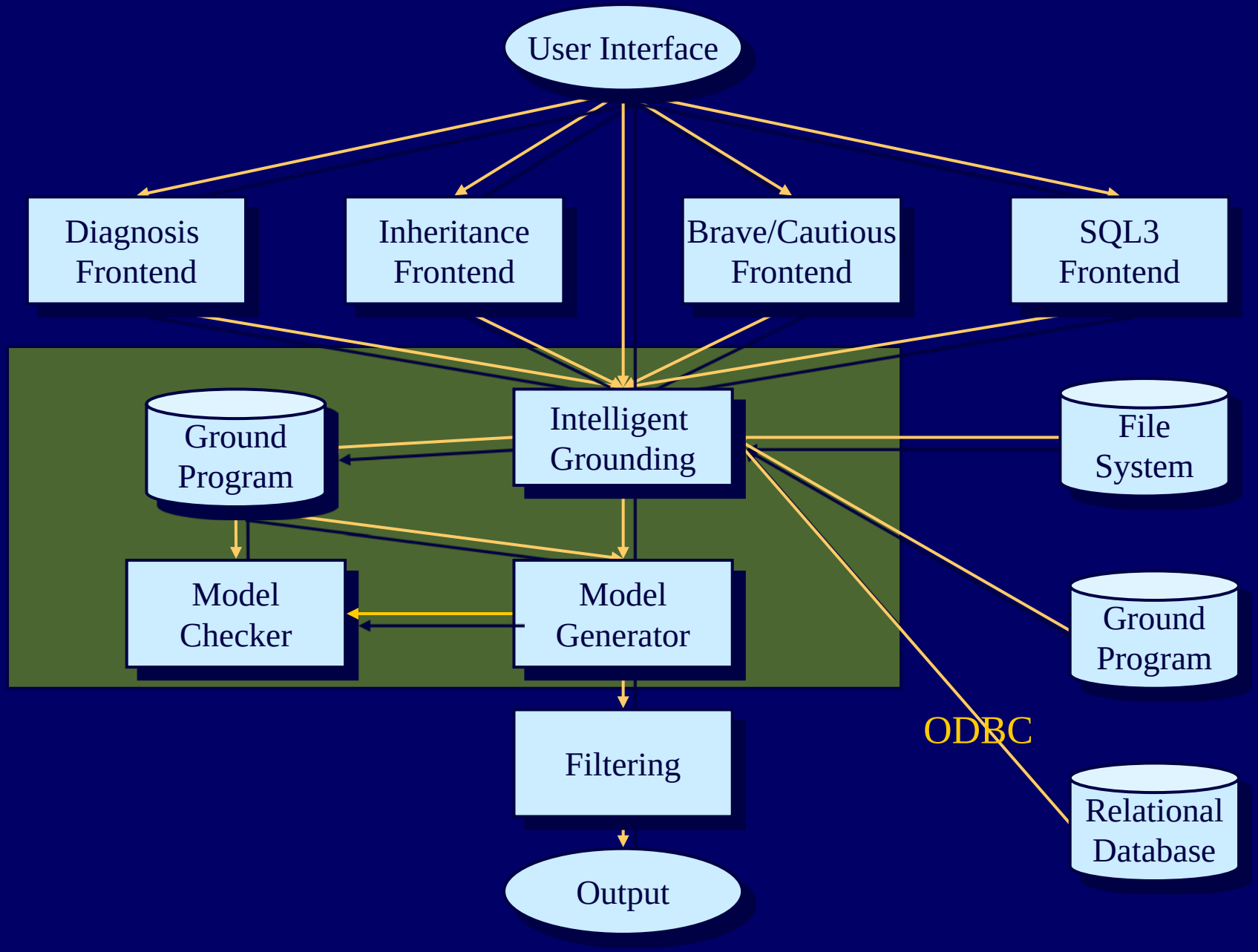
```
WITH RECURSIVE listofmaterials(major,minor) AS
    (
    SELECT c.major, c.minor FROM consists_of AS c
    UNION
    SELECT c1.major, c2.minor
    FROM consists_of AS c1, listofmaterials AS c2
    WHERE c1.minor = c2.major
    )
SELECT major, minor FROM listofmaterials;
```

Planning Frontend

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 occupied(B) if on(B1,B), block(B).
caused on(B,L) after move(B,L).
caused -on(B,L1) after move(B,L), on(B,L1), L<>L1.
noConcurrency.

System Architecture



Some Exemplary Benchmarks (june 2001)

Problem instance	Time
Strategic Companies (770 C; 770 P)	1.29 s
2QBF (1000 ∀; 20 ∃; 10000 C)	25.94s
3COL (3000 E; 2000 N)	22.15s
Hamcycle (60 N)	2.09s
Blocksworld (10 blocks, 11 steps)	8.56s
Hanoi (3 stacks, 4 disks, 15 steps)	6.39s
Ramsey(3,6) ≠ 17	13.87s
Cristal	15.26s
Timetabling	26.11s

Other DLP Systems

Most systems accept only the V-free fragment of DLP

- SmodelS/GnT
- ASSAT (V-free programs, rewriting to SAT)
- NoMoRe (V-free programs, graph-colorings)
- Dislop (Model elimination, hyper-tableau calculi)
- DeReS (Default Logic)
- CCALC (Acyclic V-free programs, rewriting to SAT)
- XSB (Well-Founded Semantics)
- ASPPS (logic of propositional schemes)
- QUIP (based on QBF evaluators)
- SLG (Meta-Interpreter over prolog systems)
- DCS
- DisLog

Smodels/GnT

- Smodels is the most widely used system for V-free programs.
- Disjunction is not supported in Smodels.
- GnT extends Smodels by disjunction.
  - Rewriting + Nested calls to Smodels for answer set checking.
- Smodels supports a powerful construct for representing cardinality and weight constraints

Summary of Benchmarks

maximum size where all out of 50 random instances where solved in 2 hours of cpu time and 256MB of memory - August 2002

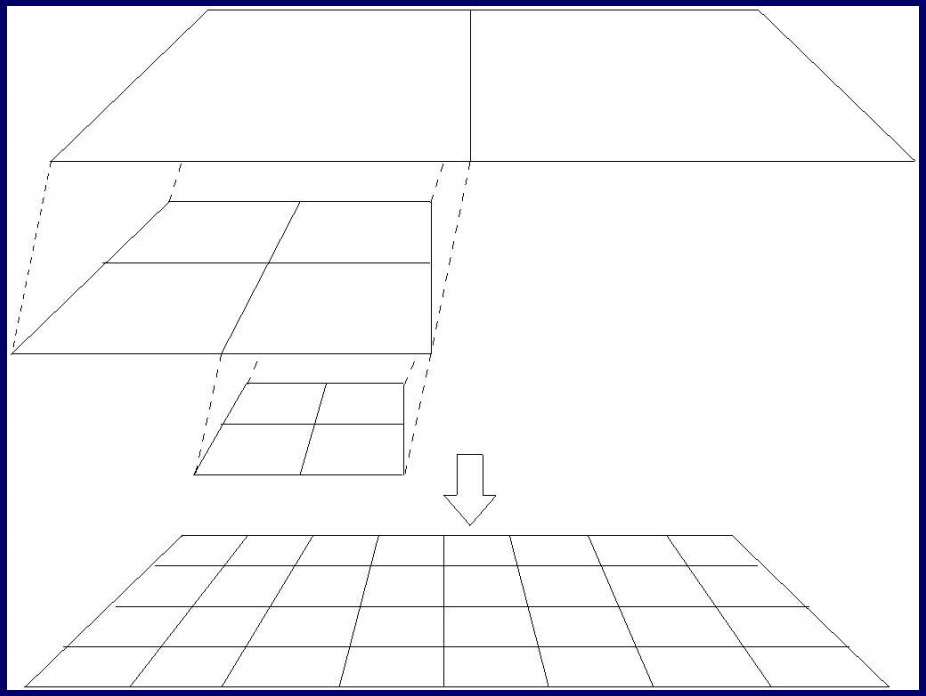
Problem	DLV	Smodels	Smodels2	ASSAT
REACH	8500	450	n/a	450
SAMEGEN	9025	676	n/a	676
STRATCOMP	170	115	n/a	n/a
QBF	48	40	n/a	n/a
HAMPATH	105	50	40	50
TSP	26	24	28	n/a
SOKOBAN	95%	47%	47%	n/a
RAMSEY	100%	29%	29%	39%

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, Day, Hour]
```

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

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

New Applications
(Under Investigation)

- Information Integration
  - Complex (CONP-hard) reasoning tasks arise in this context
- Knowledge Management
  - Powerful Tool to Represent Complex Knowledge (Support for Decisions Taking, Planning)
  - Ontologies Specification and Reasoning
  - Advanced Querying/Reasoning on Top of the Result of Data/Text Mining
    (Combination of Induction with Deduction)
- Maneuvers Generation for the Space Shuttle
  - Weak Constraints improve the quality of plans

Ongoing EU Projects

- INFOMIX: Boosting Information Integration
  IST-2001-33570
  3 Years (start: April 2002)
  Main Contractor: University of Calabria

- ICONS: Intelligent CONtent Management System
  IST 2001-32429
  2 Years (start: January 2002)

- Enhancing Disjunctive Logic Programming
  for Knowledge Management Applications
  Marie Curie Grant
  2 Years (start November 2002)

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/

Function Symbols and Lists

Function Symbols in ASP

Function symbols can be used to introduce «structured» terms in ASP (like «records» or «structs» in programming languages).

- A binary tree can be represented as bt(R,LT,RT):
  - subtree(bt(R,LT,RT), T)
- An employee can be represented as emp(ID,Name,Salary):
  - assign_employee(emp(ID,Name,Salary), Dept)

Function terms can be nested:

- bt(R,bt(TL,LTL,TRL),RT)
- bt(R,bt(TL,LTL,bt(RL,nil,nil)),RT)

Function Symbols in ASP (cont.)

Functions can be used also in recursive predicates, allowing for the definition of infinite domains:

- int(0).
- int(s(N)) :- int(N).

{ int(0), int(s(0)), int(s(s(0))), int(s(s(s(0)))) […] } (Answer Set)

Functions can be used to simulate existential quantification of classical logics via skolemization:

- parent(X,Y) → person(X)
- parent(X,Y) → person(Y)
- person(X) → ∃ Y : parent(X,Y)

first-order logics

- person(X) :- parent(X,Y).
- person(Y) :- parent(X,Y).
- parent(X,f(X)) :- person(X).
- person(a).

ASP with functions

{ person(a), parent(a,f(a)), person(f(a)), parent(f(a),f(f(a))) […] } (Answer Set)

Function Symbols in ASP (cont.)

Thus, functional terms are explicitly allowed:

a TERM can be «simple» or «functional»

- p(s(X)) :- a(X, h(c)).

- We still focus on programs with safe rules!
  - equal(X,X).
  - p(X,f(Y)) :- q(X).

Not allowed

Functions?

- Functional terms are intended in the «traditional» logic programming sense: no explicit semantics is attached. A ground functional term represents a «value», just as a function-free ground term.

- Functional terms can represent values that are not originally present in the Herbrand Universe
  - Ex.: hasFather(ciccio) :- father(ciccio,f(ciccio)).
    - The name of ciccio's father does not need to be known

Functions and finiteness

- As already pointed out, a program with recursive function symbols might have an infinite ground program, which makes the computation infeasible in practice
  - Ex.: int(s(X)) :- int(X). int(0).

$TP^0 = TP(\varnothing) = \{int(0)\}$

$TP^1 = TP^0 \cup \{int(s(0))\}$

$TP^2 = TP^1 \cup \{int(s(s(0)))\}$

$TP^3 = TP^2 \cup \{int(s(s(s(0))))\}$

….

The immediate consequence operator TP does not converge finitely to a fixpoint, even on a simple positive program like this.

Ensure Computability

- In general:
  - Function symbols + recursion → Undecidability
    - Horn Logic Programming is R.E.-complete (Tarnlünd 1977)
- Even a Turing Machine can be simulated
  - see e.g., an encoding in DLV syntax at https://www.mat.unical.it/dlv-complex#Examples

- Some subclasses of programs with function symbols can be identified, that ensure termination.
  - Ex.: Finite-Domain programs, omega-restricted programs, argument-restricted programs, etc.
  - Program belonging to such classes must comply with some syntactical restrictions

Arguments

- A predicate p of arity n has n arguments.
- p[k] stands for the argument at position k
  - Ex.: predicate «p» of arity 3
    - p(X,f(ciccio),a)
    - in this case, p[1] = X, p[2] = f(ciccio), p[3] = a

Argument Graph $G^A(P)$

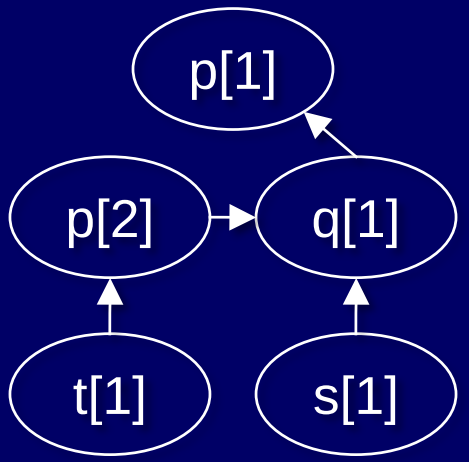
Given a program P, $G^A(P)$ is a directed graph containing a node for each argument p[i] of an IDB predicate p of P; there is an arc (q[j], p[i]) iff there is a rule r in P having an atom $p(t_1 \ldots t_n)$ in the head, an atom $q(v_1 \ldots v_m)$ in $B^+(r)$ and $p(t_1 \ldots t_n)$ and $q(v_1 \ldots v_m)$ are such that the same variable appears within terms t_i and v_j, respectively.

q(g(3)).

p(X,Y) :- q(g(X)), t(f(Y)).

s(X) | t(f(X)) :- a(X), not q(X).

q(X) :- s(X), p(Y,X).



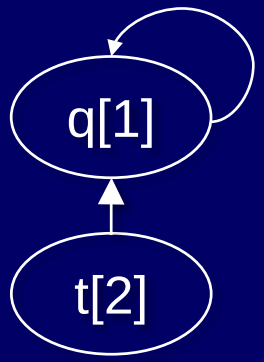
Recursive Arguments

Given a program P, an argument p[i] is said to be recursive with q[j] if there exists a cycle in the Argument Graph of P involving both p[i] and q[j].

- the Argument Graph keeps track of (body-head) dependencies between the arguments of predicates sharing some variable.
- it is a more detailed version of the commonly used (predicate) dependency graph.

t(X,Y):- a(X,Y), not q(X).

q(f(X)) :- q(X), t(Y, f(X)).



Argument q[1] is recursive.

FD-programs: definition

- FINITE-DOMAIN programs: a simple, decidable subclass of ASP programs
- All the arguments of a program P must be *finite-domain.*

- An argument q[k] in the Argument Graph is FINITE-DOMAIN if for all occurrences q(…,t,…) of it appearing in the head of a rule, one of the following conditions holds:
  1. t is variable free (e.g. q(…, f(a,g(b)),…) is OK.), or
  2. t is subterm of another fd-argument appearing in the body (e.g. q(…, X,…)) :- p(f(X), …) is OK if p[1] is fd), or
  3. t appears in some body arguments which is not recursive with q[k] (e.g. q(…, X,…)) :- p(f(X), …) is OK if p[1] is not recursive with q[k]

FD-programs: examples

- FD-program (q[1] is a FD argument)

  q(f(0)). (condition 1 holds)

  q(X) :- q(f(X)). (condition 2 holds)

  q(f(X)) :- q(X), t(f(X)). (condition 3 holds)

- Non FD-program (s([1]) is not a FD argument)

  q(f(0)). (q[1]: condition 1 holds)

  q(X) :- q(f(X)). (q[1]: condition 2 holds)

  s(f(X)) :- s(X). (s[1]: no condition hold)

  v(X) :- q(X), s(X). (v[1]: condition 2 holds)

Finitely-ground (FG) Programs

INTUITION: The class of Finitely Ground Programs is the set of programs for which the «intelligent» instantiation is finite and *computable*, i.e., the TP operator converges finitely to a fix-point.

Finitely-ground (FG) and Finite Domain (FD) Programs

THEOREM: Every FD program is Finitely Ground.

Example: The following program is not FD, even if it is Finitely Ground.

p(f(X)) :- q(X).

q(X) :- p(X), r(X).

Functions in DLV / DLV2

Functions are Implemented in DLV: it fully supports both Finitely Ground Programs and Finite Domain programs. Actually, DLV is able to recognize Argument Restricted (AR) programs, which is a super-class of FD. AR programs require a more involved and less intuitive syntactic check to be recognized.

If a program P is finitely ground, then DLV instantiator terminates over P, generating its correct instantiation. By default, DLV checks the input program P: if it belongs to the class of AR programs, then DLV guarantees termination. If not, DLV returns an error.

The user must take over the responsibility about the program being finitely ground in order to run non-AR programs. With option –nofinitecheck, DLV skips the finite domain check. The instantiation will then terminates if the program is FG.

DLV2 & I-DLV+WASP currently do not apply any syntactical check, so the responsibility is left to the user.

Lists in DLV-DLV2-IDLV

- Very common data structure
- Easily obtained via function symbols

However, due to the usefulness

- explicit syntax is supported
- Dedicated built-ins are available

Lists in DLV-DLV2-IDLV

A list is a binary function denoted with a special syntax:

[H | T]

where the first argument «H» is a term, called the head of the list, and the second argument «T» is a list.

In addition, a list can be represented by explicitly listing its elements.

[a, b, c] = [a | [b, c]] = [a | [b | [c]]]

= [a | [b | [c | [ ]]]]

Lists in DLV / DLV2 / I-DLV

LIST TERMS

A list term can be of the two forms:

– [t1, . . . , tn], where t1, . . . , tn are terms;

– [h | t], where h (the head of the list) is a term, and t (the tail of the list) is a list term.

Examples:

- The term [a,d,a] in the atom palindromic([a,d,a])
- [jan,feb,mar]
- [jan | [feb,mar,apr,may,jun]]
- [[jan,31] | [[feb,28], [mar,31], [apr,30], [may,31], [jun,30]]].

Built-in predicates for Lists

I-DLV (and DLV2) comes with a rich library of built-in predicates for list manipulation.

A built-in atom is of the form

$$\&p(t_0,..,\ t_n\ ;\ u_0,\ldots,\ u_m)$$

where n,m >= 0

- $t_0,..,\ t_n$ are input terms, and are separated from the output terms $u_0,\ldots,\ u_m$ by a semicolon (“;”);
- an input/output term can be either a constant or a variable.

Intuitively, output terms are computed on the basis of the input ones, according to a semantics which is defined “a priori” for each predicate, as reported next.

Built-in predicates for Lists

&head (L ; Elem)	Given a list L, binds $Elem$ to the first element in L.
&tail (L ; ListR)	Given a list L, binds $ListR$ to the sublist consisting of L without the first element.
&append (L1, L2 ; ListR)	Given two lists $L1$, $L2$, binds $ListR$ to the list consisting of $L2$ appended to $L1$.
&delNth (L, N ; ListR)	Given a list L and a natural number N, binds $ListR$ to the list consisting of L where the N-th element has been removed.
&flatten (L ; ListR)	Given a list L of the form $[H\|T]$, binds $ListR$ to the flattened list of the form $[T_1, \ldots, T_n]$.
&insLast (L, E ; ListR)	Given a list L and an element E, binds $ListR$ to the list obtained by appending E to L.
&insNth (L, E, N ; ListR)	Given a list L, an element E and a natural number N, binds $ListR$ to the list obtained by inserting E into L at position N.
&last (L ; E)	Given a list L, binds E to the last element of L.
&length (L ; X)	Given a list L, binds X to the number of elements in L.
&member (E, L ;)	Given an element E and a list L, is true iff $E \in L$.
&memberNth (L, N ; E)	Given a list L and a natural number N, binds E to the N-th element of L.
&subList (L1, L2 ;)	Given two lists $L1$ and $L2$, is true iff $L1$ is a sublist of $L2$.
&reverse (L ; ListR)	Given a list R, binds $ListR$ to the list obtained by reverting L.
&delete (E, L ; ListR)	Given an element E and a list R, binds $ListR$ to the list obtained by removing E from L.

Lists: example

Compute all simple (i.e. with no repeated vertices) paths in a graph:

```
simplePath([X|[Y]]) :- edge(X,Y).
simplePath([X|[Y|W]]) :- edge(X,Y), simplePath([Y|W]),
                         not &member(X,[Y|W];).
```

Compute all simple cycles in a graph:

```
simpleCycle([X]) :-    edge(X,X).
simpleCycle([X|L]) :-  simplePath([X|L]), &last(L;Y), edge(Y,X).
```

Compute simple paths of maximum length:

```
maxPath(P) :- simplePath(P), &length(P;L),
              L = #max { X : simplePath(Q), &length(Q;X) }.
```

LISTS: Applications

- Lists can easily represent trees:
- [root| List-of-subtrees]
- Leaf node: [a|[]] = [a]
- [a | [[b], [c | [d]]]]
  - the list rooted in a
  - b and c are the children of a
  - d is a child of c
  - b and d are leaves

List can therefore represent the HTML trees of web pages

DIADEM

ERC Advanced Grant @Oxford - G. Gottlob

Domain-centric, Intelligent, Automated Data Extraction

- fully automated extraction from domain-specific websites
  - no per site training, no user input other than the domain model
- main target: websites with structured records
- based on extensive domain knowledge
  - web form understanding
  - result page analysis (records, attributes)
  - navigation blocks classification (next page link, detail pages)
- Template language on Datalog$^{\neg,Agg}$ rules compiled to DLV, plus Gazetteers, GATE annotation&regex, ML classifiers

Web Form Understanding with OPAL

Ontology-based Pattern Analysis with Logic

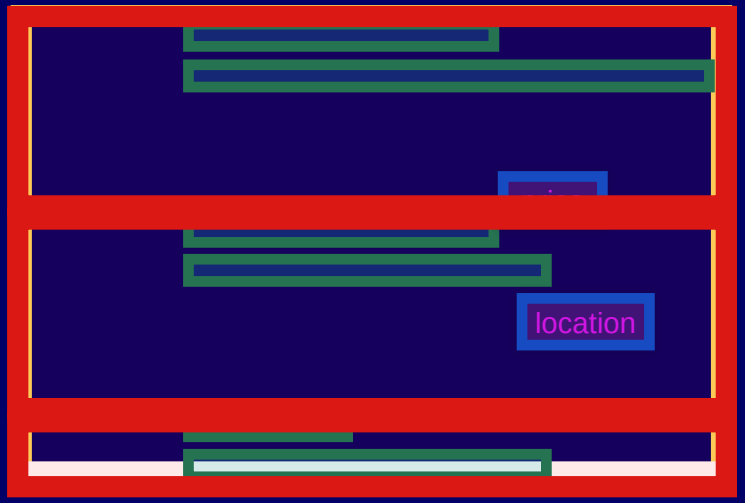
- Recognizes and labels groups of fields + classifies them w.r.t. the domain ontology
- Reasoning on structural & visual patterns + annotations

```
group(Es) :-  similarFieldSequence(Es),
              leastCommonAncestor(A,Es),
              not hasAdditionalField(A,Es).

leastCommonAncestor(A,Es) :- commonAncestor(A,Es),
              not ( child(C,A), commonAncestor(C,Es) ).

partOf(E,A) :-  group(Es),
              member(E,Es), leastCommonAncestor(A,Es).
```

Result Page Analyses with AMBER

Adaptable Model Based Extraction of Result Pages

Reasoning on annotations and page structure to identify records & attributes



consistent_cluster_members(C, N1, N2, N3) :- pivot(N1), pivot(N2), ...
similar_depth(N1, N2), similar_depth(N2, N3), similar_depth(N1,N3),
similar_tree_distance(N1, N2, N3).

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 | 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 | b.
:- not a.
```

what are the answer sets?

Homework (5)

Consider program

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

is there any answer set?

Homework (6)

Consider program

a :- b.

b :- a.

a | 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) | 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) | 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) | 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) | 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) | 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 | such that for each edge <u,v> in E at least one of u and | belongs to V'.

Answer to Homework (10) (Node Cover)

```
%Guess a set of nodes
inCover(X) | 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 | of minimum cardinality such that for each edge <u,v> in E at least one of u and | belongs to V'.

Answer to Homework (11) (Minimum Node Cover)

```
%Guess a set of nodes
inCover(X) | 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). [1@1, 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) | col(X,blue) | 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)}

>
```

Combinatorial Auctions

In combinatorial auctions, bidders can bid for portfolios of several goods at once. Each bidder can place as many bids as s/he wants. The goal is to maximize the auctioneers revenue.

The input can be represented as follows, where requires states that some item is part of a bid, and bid represents the price associated with a bid:

```
requires(bid1, item1).
requires(bid1, item2).
requires(bid2, item1).
requires(bid2, item3).
      :
bid(bid1, 21).
bid(bid2, 37).
```

Write a GCO program for DLV that solves this problem and also develop two test cases, where you see that your program works correctly.

Hints

Hint 1: The output should list the bids which are accepted by the auctioneer with the respective prices. It should be of the form {accept(bid1,p1), accept(bid3,p3),... }.

Hint 2: By means of weak constraints one can more easily perform minimization than maximization. However, if you minimize the cost of those items not accepted, then the cost of those that are accepted is maximized in consequence!

More on Combinatorial Auctions

The auctioneer, while preserving the maximization of the revenue as the first criterion, wants to be parsimonious w.r.t. the given items, that is, (s)he wants to keep how many items as possible.

Refine the previous program to prefer, if two solutions give the same revenue, the solution where a smaller number of items is given away by the auctioneer.

Hint 1: use an auxiliary predicate defining the givenItems in a solution.

Hint 2: Priorities in weak constraints are needed.