

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

Nicola Leone

Department of Mathematics

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

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

Disjunctive Logic Programming (DLP)

Disjunctive Datalog

Disjunctive Databases

Answer Set Programming (ASP)

Roots – declarative programming

- Algorithm = Logic + Control (Kowalski, 1979)
- 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)

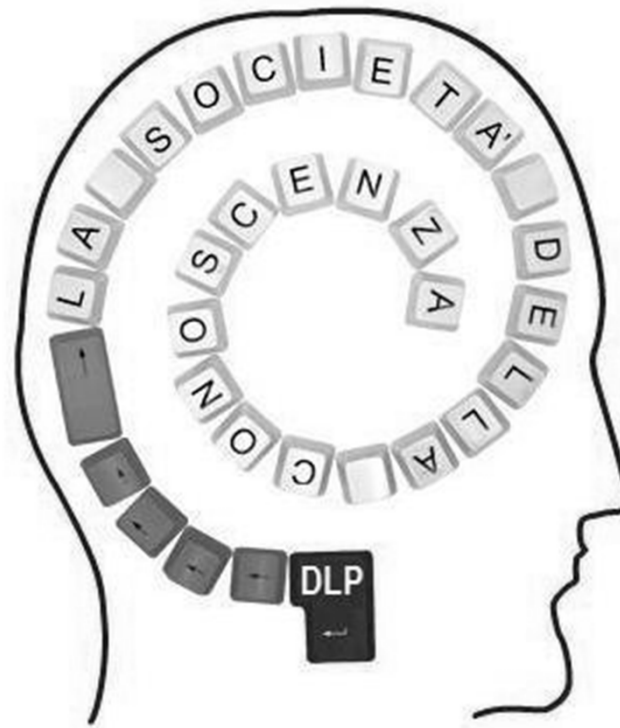
- Simple, yet powerful KR formalism
- Widely used in AI
 - Incomplete Knowledge
- Able to represent complex problems not (polynomially) translatable to SAT
- A declarative problem specification is executable

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 desired 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

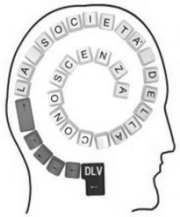
Applications

- 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

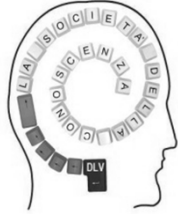


TEAM BUILDING at Seaport of Gioia Tauro

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



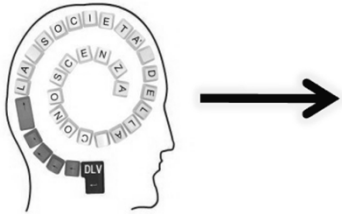
TEAM BUILDING at Seaport of Gioia Tauro

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



Automa - TeamBuilding

File Search View Help Run

Dipendenti Logistica Calendario Presenze

Metapiani

- gennaio
- febbraio
- marzo
- aprile
- maggio
- giugno
- luglio
- agosto
- settembre
- ottobre
- novembre
- dicembre

Logistica

Calendario Metapiani

Proprietà Team

Nome Cognome Ma

Logistic

Tipologia Turno
Consente di selezionare la tipologia del turno
 Turno Singolo Raddoppio su singola nave Raddoppio su due navi

Turno
Descrizioni delle proprietà del turno
Nome Turno _____
Data _____
Orario 16.02.49 16.02.49

Turno
Descrizioni delle proprietà del turno
 No Pausa
Inizio pausa 16.02.49 16.02.49
Durata turno _____
Volumi _____
Lavorazione _____

Turno Raddoppio
Inserimento dati per i turni di raddoppio
Nome Turno _____
Cambio Funzione _____ 16.02.49

Mansioni Richieste
Elenco delle mansioni richieste
D _____ HH _____ L _____ LC _____ M _____

Progress

Creazione Calendario (Finished at 16.03)
Creazione Calendario: 15/11/2009

Inclusioni

| Nome | Cognome | Mansione |
|------|---------|----------|
| | | |
| | | |
| | | |

Esclusioni

| Nome | Cognome |
|------|---------|
| | |
| | |
| | |

Statistica

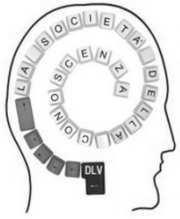
driver high_heavy lasher lasher_coord magazzinieri mobile_data_entry parker pdi quality_checker service_person taxi_driver trucks yard_mde

Data

| Data |
|------|
| |
| |
| |

start Google ... Skype... D:\wor... Java - ... Download padl09-... Automa... IT 16.04

DLP Revolution



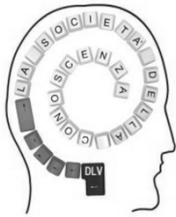
→ Automatic Itinerary Search

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 | Ora Partenza | Ora Arrivo | Durata | Cambi | Tipo | Dettagli |
|---|--------------|------------|--------|-------|------|----------|
| Partenza bivio fagnano castello (cs) | 06:05 | 07:40 | 01:35 | 1 | | |
| Destinazione cosenza - autostazione (cs) | 06:05 | 08:00 | 01:55 | 1 | | |
| Orario 06:00 - 12:00 | 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 $p(t_1, \dots, t_n)$, where p is a predicate name, t_1, \dots, 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.: $p(X, Y)$ - $\text{next}(1, 2)$ - q - $\text{i_am_an_atom}(1, 2, a, B, X)$
- An atom can be negated by means of “not”.
 - Ex: $\text{not } a$, $\text{not } p(X)$, ...
- 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

$$\text{Rule: } \underbrace{a}_{\text{head}} \text{ :- } \underbrace{b_1, \dots, b_k}_{\text{positive body}}, \underbrace{\text{not } b_{k+1}, \dots, \text{not } b_m}_{\text{negative body}} \quad (1)$$

body

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 $b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_m$ of all body literals
- $B^+(r)$: (positive body), the set b_1, \dots, b_k of positive body literals
- $B^-(r)$: (negative body), the set $\text{not } b_{k+1}, \dots, \text{not } b_m$ 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

$r_1:$ $a \text{ :- } b, c.$

$r_2:$ $c \text{ :- } d.$

$r_3:$ $d.$

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 .*

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

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

Example:

$a :- b.$ $c :- d.$ $e :- a.$

$I = \{b\} \rightarrow Tp(I) = \{a\}.$

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

Thus: Start with $I = \{\text{facts in the EDB}\}$ 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).

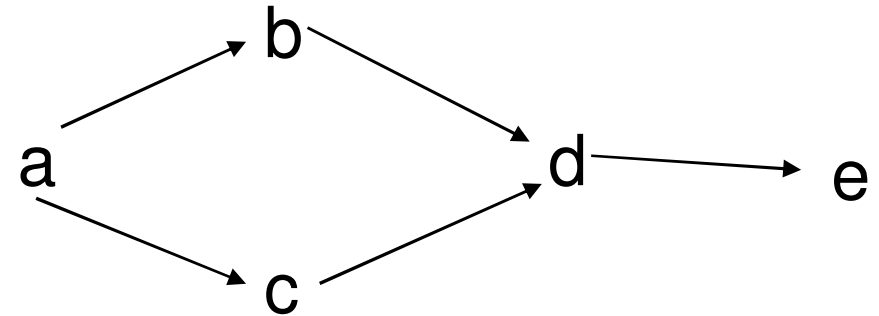
Thus we derive: { ancestor(a,d) } - *Note:* this is derived
exploiting “*ancestor(b,d)*” but also “*parent(a,b)*”, which was
derived before last stage.

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 $\text{arc}(X,Y)$ as arcs in a graph.

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

$s(X,Y) \text{ :- arc}(X,Y), \text{ not arc}(Y,X).$

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:

- $s(X) :- a.$
- $s(Y) :- b(Y), \text{not } r(X).$
- $s(X) :- \text{not } r(X).$
- $s(Y) :- b(Y), X < Y.$

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: $p(X) \text{ :- } q(X), \text{ not } p(X).$

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

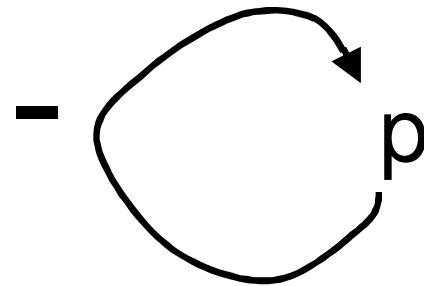
To formalize strata use the labeled *dependency graph*:

- Nodes = IDB predicates.
- Arc $b \rightarrow a$ 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

$p(X) \text{ :- } q(X), \text{ not } p(X).$



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”:

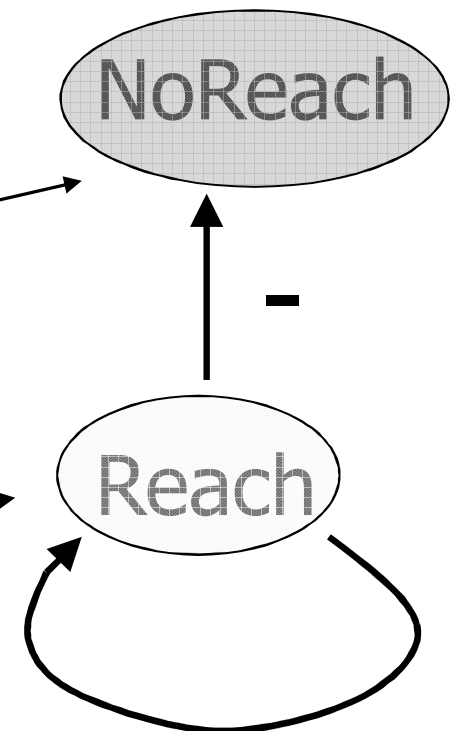
reach(X) :- source(X).

reach(X) :- reach(Y), arc(Y,X).

noReach(X) :- target(X), not reach(X).

Stratum 1:
some “-” arc
incoming from
Stratum 0

Stratum 0:
No “-” arcs on
any path in



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)

$a \text{ :- not } b.$

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 $\text{sub}P(C)$ 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

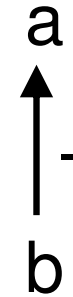
INPUT: EDB F, IDB P

- Compute the labeled dependency graph DG of P;
- Build a topological ordering C_1, \dots, C_n of the components of DG;
- $M = F$;
- For $i=1$ To n Do
 - $M = \text{SemiNaive}(M \cup \text{sub}P(C_i))$
 - *% compute the least fixpoint of T_p on $(M \cup \text{sub}P(C_i))$*
- OUTPUT M;

Stratified Model: example

a :- not b.

b :- d.



Two components: {a} and {b}.

$\text{subP}(\{b\}) = \{b \text{ :- } d.\}$

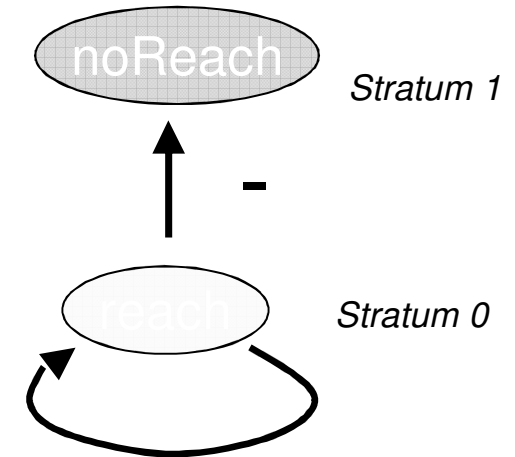
$\text{subP}(\{a\}) = \{a \text{ :- not } b.\}$

- {b} is at the lowest stratum -> start evaluating $\text{subP}(\{b\})$.
- The answer set of $\text{subP}(\{b\})$ is $\text{AS}(\text{subP}(\{b\})) = \{\}$.
 - “{}” is the input for $\text{subP}(\{a\})$.
- The answer set of $\text{subP}(\{a\}) \cup \{\}$ is $\text{AS}(\text{subP}(\{a\})) = \{a\}$, 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:

$C1 = \{\text{reach}\}$ $C2 = \{\text{noReach}\}$

And the related subprograms are:

$\text{subP}(\{\text{reach}\}) = \{ \text{reach}(X) :- \text{source}(X). \\ \text{reach}(X) :- \text{reach}(Y), \text{arc}(Y,X). \}$

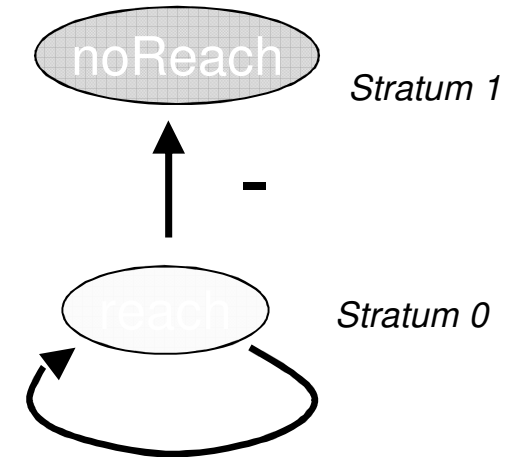
$\text{subP}(\{\text{noReach}\}) = \{ \text{noReach}(X) :- \text{target}(X), \text{not reach}(X). \}$

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: $a_1 \mid \dots \mid a_n \text{ :- } b_1, \dots, b_k, \text{ not } b_{k+1}, \dots, \text{ not } b_m$

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

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

- a_i b_i are atoms
- variables are allowed in atoms' arguments

$\text{mother}(P,S) \mid \text{father}(P,S) \text{ :- } \text{parent}(P,S).$

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, programs which contain disjunction can have more than one 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.

Default Negation

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

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

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

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

Strong Negation

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

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

This concept is called *strong negation*:

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

The use of strong negation can lead to *inconsistencies*:

a. -a.

Informal Semantics

Rule: $a_1 \mid \dots \mid a_n \text{ :- } b_1, \dots, b_k, \text{ not } b_{k+1}, \dots, \text{ not } b_m$

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

$\text{isInterestedinDLP(john)} \mid \text{isCurious(john)} \text{ :- } \text{attendsDLP(john)}.$
 $\text{attendsDLP(john)}.$

Two (minimal) models, encoding two plausible scenarios:

M1: { attendsDLP(john) , $\text{isInterestedinDLP(john)}$ }

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

Disjunction

is *minimal*

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

actually *subset minimal*

$$\begin{array}{l} a \mid b. \\ a \mid c. \end{array} \Rightarrow \{a\}, \{b,c\}$$

but *not exclusive*

$$\begin{array}{l} a \mid b. \\ a \mid c. \\ b \mid c. \end{array} \Rightarrow \{a,b\}, \{a,c\}, \{b,c\}$$

Informal Semantics

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

Discard interpretations which verify the condition

$\text{:- hatesDLP(john), isInterestedinDLP(john).}$

hatesDLP(john).

$\text{isInterestedinDLP(john) | isCurious(john) :- attendsDLP(john).}$

attendsDLP(john).

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

only one plausible scenario:

M: $\{\text{attendsDLP(john), hatesDLP(john), isCurious(john)}\}$

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.

$\text{: - } L_1, \dots, L_n.$

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

$a \mid b.$

$a \mid c. \quad \Rightarrow \{a,b\}, \{a,c\}, \{b,c\}$

$b \mid c.$

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

(Formal) Semantics: Program Instantiation

Herbrand Universe, UP = Set of constants occurring in program P

Herbrand Base, BP = Set of ground atoms constructible from UP and $Pred$.

Ground instance of a Rule R : Replace each variable in R by a constant in UP

Instantiation $ground(P)$ of a program P : Set of the ground instances of its rules.

Example: $isInterestedinDLP(X) \mid isCurious(X) :- attendsDLP(X).$

$attendsDLP(john).$

$attendsDLP(mary).$

$UP = \{ john, mary \}$

$isInterestedinDLP(john) \mid isCurious(john) :- attendsDLP(john).$

$isInterestedinDLP(mary) \mid isCurious(mary) :- attendsDLP(mary).$

$attendsDLP(john).$

$attendsDLP(mary).$

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 $\text{not } q$ 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 $\text{ground}(P)$) 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:

$\text{- hatesDLP(john), isInterestedinDLP(john).}$

hatesDLP(john).

(same interpretations as before + hatesDLP(john))

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

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

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

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

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

$I6 = \{ \text{isInterestedinDLP(john), attendsDLP(john), isCurious(mary), attendsDLP(mary), hatesDLP(john)} \}$ (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: $a \text{ :- not } b.$

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.

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.

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.

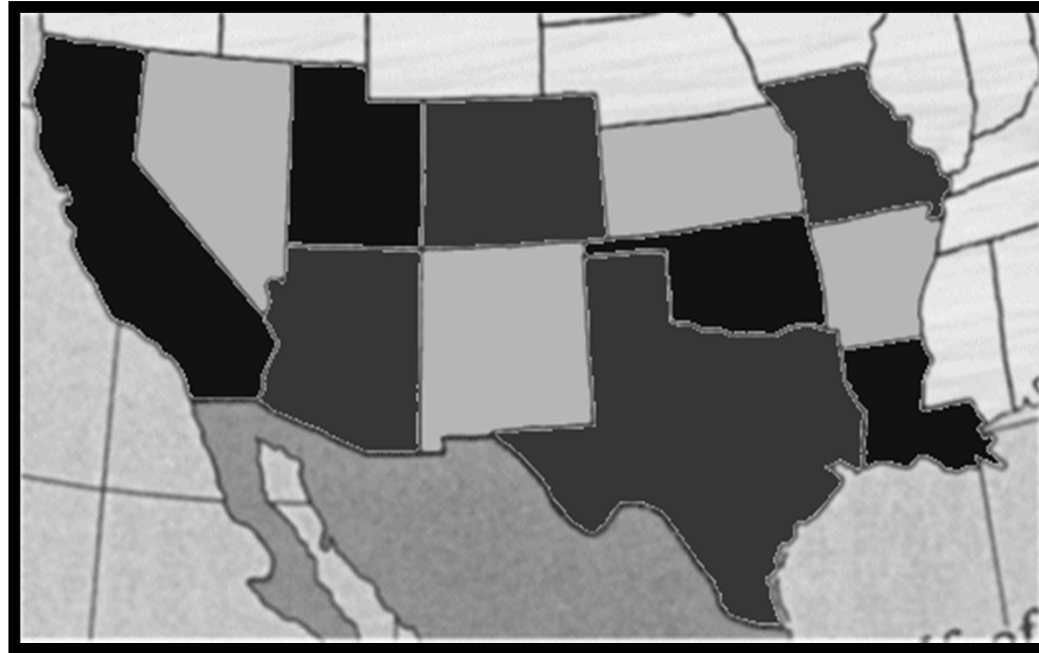
From another perspective:

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

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.



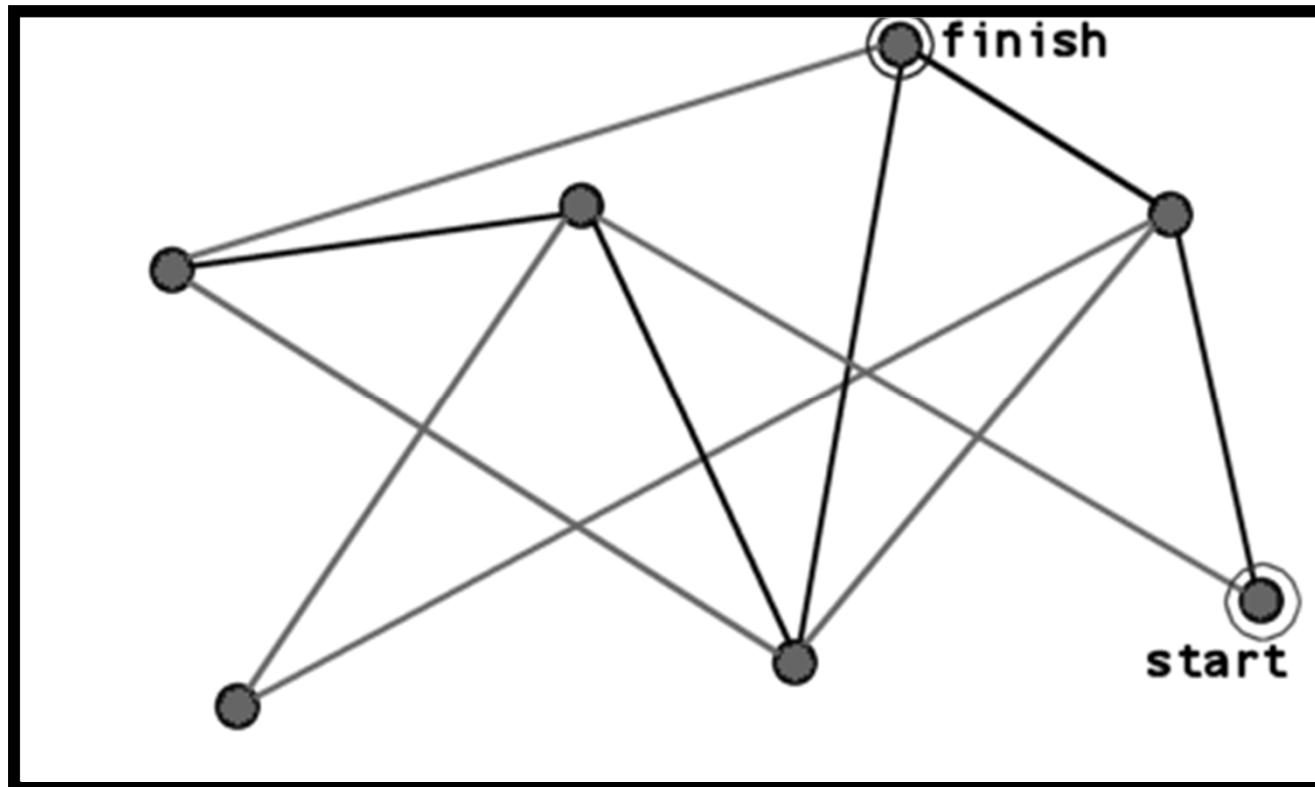
Solution:

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

Hamiltonian Path (HP) (1)

Input: A directed graph represented by $\text{node}(_)$ and $\text{arc}(_,_)$, and a starting node $\text{start}(_)$.

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.` **Check**

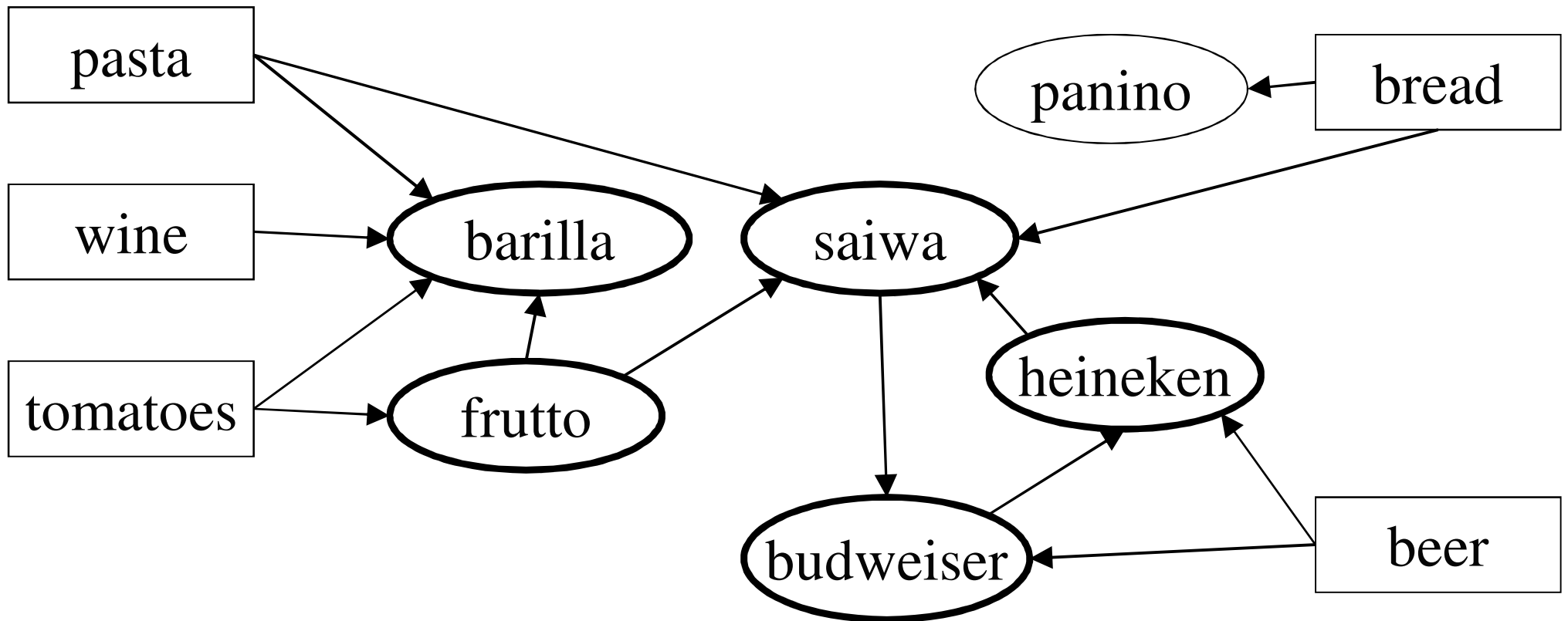
`:- node(X), not reached(X).`

`:- inPath(X,Y), start(Y). % a path, not a cycle`

`reached(X) :- start(X).` **Auxiliary Predicate**


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

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  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 $G \cup C$ 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.

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

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

$$d_{i1} \vee \dots \vee d_{ic_i}$$

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

- 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 }

SAT: ASP encoding

Add a guessing rule for each propositional variable

$$\forall d_i \rightarrow d_i \mid \text{nd}_i.$$

Add a constraint for each clause, complementing the variables

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

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

Example: SAT \rightarrow ASP

Formula

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

ASP encoding:

- $d_1 \mid \neg d_1.$ $:- \text{not } d_1, d_2, d_3.$
- $d_2 \mid \neg d_2.$ $:- d_1, \text{not } d_2, \text{not } d_3.$
- $d_3 \mid \neg d_3.$

Answer Sets

| | |
|-----------------------------|------------------------------------|
| { d1, d2, \neg d3} | { \neg d1, \neg d2, \neg d3} |
| { \neg d1, d2, \neg d3} | { \neg d1, \neg d2, d3} |
| { d1, \neg d2, d3} | { d1, d2, d3} |

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 $\text{Rules}(P)$?

Syntactic restrictions on DLP programs

- Head-Cycle Free Property

[Ben-Eliyahu, Dechter]

- Stratification

[Apt, Blair, Walker]

Level Mapping: a function $|| \cdot ||$ 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 $\| \cdot \|_s$ of P such that for every rule r of P
- For any l in $\text{Body}_+(r)$, and for any l' in $\text{Head}(r)$, $\| l \|_s \leq \| l' \|_s$;
 - For any $\text{not } l$ in $\text{Body}_-(r)$, and for any l' in $\text{Head}(r)$, $\| l \|_s < \| l' \|_s$

Example: A stratified program

P1: $p(a) \mid p(c) \text{ :- not } q(a).$
 $p(b) \text{ :- not } q(b).$

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: $p(a) \mid p(c) \text{ :- not } q(b).$
 $q(b) \text{ :- not } p(a)$

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.

Complexity of Answer-Set Checking

| | | | |
|--------|--------|----------------|------|
| | $\{\}$ | not_s | not |
| $\{\}$ | P | P | P |
| V | coNP | coNP | coNP |

Complexity of Brave Reasoning

| | | | |
|--------|--------------|----------------|--------------|
| | $\{\}$ | not_s | not |
| $\{\}$ | P | P | NP |
| V | Σ^P_2 | Σ^P_2 | Σ^P_2 |

Completeness under Logspace reductions

Intuitive Explanation

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 $\text{Rules}(P)$ (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 | not |
| $\{\}$ | P | P | coNP |
| V | coNP | Π^P_2 | Π^P_2 |

Note that $\langle V, \{\} \rangle$ is “only” coNP-complete!