Answer Set Programming

Torsten Schaub
University of Potsdam
torsten@cs.uni-potsdam.de

Answer Set Programming

Winter Semester 2011/12

- Martin Gebser
- Torsten Schaub
- Marius Schneider

Information

- Lecture: 2h (weekly)
- Exercises: 2h (weekly)
- Credits: 6 **if**
  1. Written exam (at least “ausreichend”)
  2. Two successful projects (= Implementation+Consultation)
- Mark: mark of written exam
- C(ourse)MS: `http://moodle.cs.uni-potsdam.de/`
- General Info: `http://www.cs.uni-potsdam.de/wv/lehre`
- Contact:
  - Lecture&Exercises: `asp@cs.uni-potsdam.de`
  - Projects: `asp1@cs.uni-potsdam.de`

Roadmap

- Introduction
- Modeling
- Language Extensions
- Operators, Algorithms, and Systems
- Applications

Resources

- Course material
  - `http://www.cs.uni-potsdam.de/wv/lehre`
  - `http://moodle.cs.uni-potsdam.de`
  - `http://www.cs.uni-potsdam.de/~torsten/asp`
- Systems
  - **clasp** `http://potassco.sourceforge.net`
  - dlv `http://www.dbai.tuwien.ac.at/proj/dlv`
  - smodels `http://www.tcs.hut.fi/Software/smodels`
  - **gringo** `http://potassco.sourceforge.net`
  - lparse `http://www.tcs.hut.fi/Software/smodels`
  - **clingo** `http://potassco.sourceforge.net`
  - iclingo `http://potassco.sourceforge.net`
  - oclingo `http://potassco.sourceforge.net`
  - asparagus `http://asparagus.cs.uni-potsdam.de`

Literature

Books [5], [65]

Surveys [59], [3], [47]

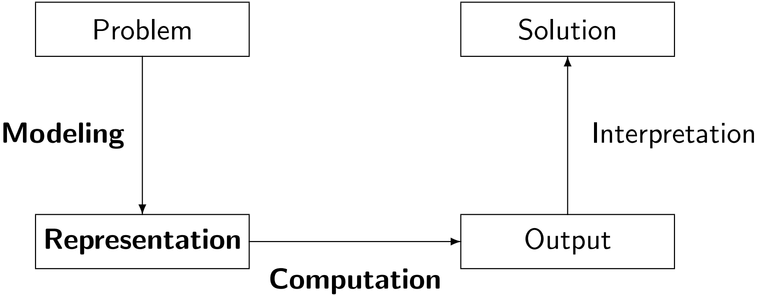
Articles [49], [50], [7], [71], [66], [58], [48], etc.

Motivation: Overview

Overview

Goal: Declarative problem solving

- *"What is the problem?"*
  instead of
- *"How to solve the problem?"*



Overview

Answer Set Programming (ASP) *in a Nutshell*

- ASP is an approach to **declarative problem solving**, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities

  tailored to Knowledge Representation and Reasoning
- ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way (being more compact than SAT)
- The versatility of ASP is reflected by the ASP solver `clasp`, winning first places at ASP'07/09/11, PB'09/11, and SAT'09/11
  - `http://potassco.sourceforge.net`
- ASP embraces many emerging application areas, eg.
  - second place at RoboCup@Home 2011 by USTC, Peking
  - configuration by SIEMENS, Vienna

Overview

Logic Programming

- Algorithm = Logic + Control [55]
- Logic as a programming language
  - ➥ Prolog (Colmerauer, Kowalski)
- Features of Prolog
  - Declarative (relational) programming language
  - Based on SLD(NF) Resolution
  - Top-down query evaluation
  - Terms as data structures
  - Parameter passing by unification
  - Solutions are extracted from instantiations of variables occurring in the query

Prolog: Programming in logic

Prolog is great, it's **almost** declarative!
To see this, consider

```
above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z),above(Z,Y).
```

and compare it to

```
above(X,Y) :- above(Z,Y),on(X,Z).
above(X,Y) :- on(X,Y).
```

An interpretation in classical logic amounts to

$$\forall xy(on(x,y) \vee \exists z(on(x,z) \wedge above(z,y)) \rightarrow above(x,y))$$

Model-based Problem Solving

Traditional approach (e.g. Prolog)

1. Provide a specification of the problem.
2. A solution is given by a **derivation** of an appropriate query.

Model-based approach (e.g. ASP and SAT)

1. Provide a specification of the problem.
2. A solution is given by a **model** of the specification.

Automated planning, Kautz and Selman [53]

Represent planning problems as propositional theories so that models not proofs describe solutions (e.g. `Satplan`)

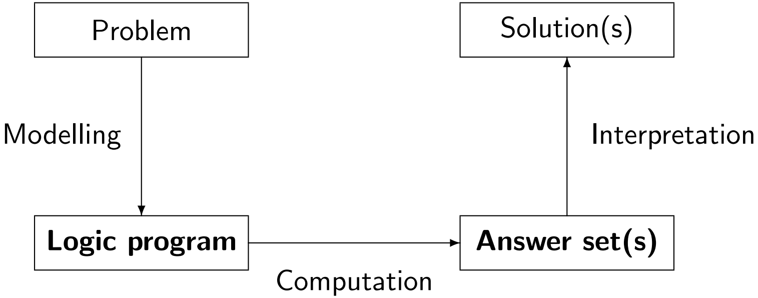
Overview

Model-based Problem Solving

Specification	Associated Structures
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	**stable models**
propositional programs	minimal models
propositional programs	supported models
propositional programs	**stable models**
first-order theories	models
default theories	extensions
...	

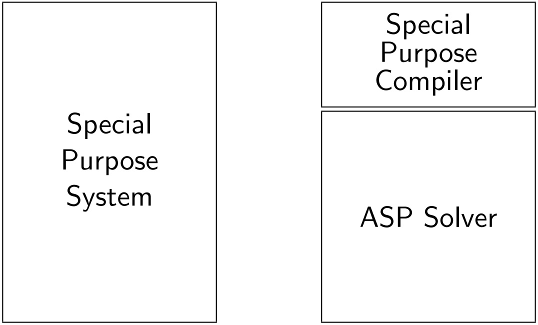
ASP as High-level Language

- Basic Idea:
  - Encode problem (class+instance) as a set of rules
  - Read off solutions from answer sets of the rules



ASP as Low-level Language

- Basic Idea:
  - Compile a problem automatically into a logic program
  - Solve the original problem by solving its compilation



Overview

What is ASP good for?

- Combinatorial search problems
  (some with substantial amount of data):
  - For instance, auctions, bio-informatics, computer-aided verification, configuration, constraint satisfaction, diagnosis, information integration, planning and scheduling, security analysis, semantic web, wire-routing, zoology and linguistics, and many more
- My favorite: Using ASP as a basis for a decision support system for NASA's space shuttle (Gelfond et al., Texas Tech)
- Our own applications:
  - Automatic synthesis of multiprocessor systems
  - Inconsistency detection, diagnosis, repair, and prediction
    in large biological networks
  - Home monitoring for risk prevention in ambient assisted living
  - General game playing

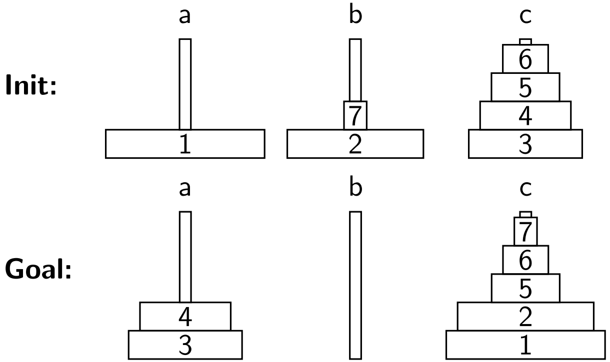
What does ASP offer?

- Integration of KR, DB, and search techniques
- Compact, easily maintainable problem representations
- Rapid application development tool
- Easy handling of dynamic, knowledge intensive applications
  (including: data, frame axioms, exceptions, defaults, closures, etc.)

ASP = KR + DB + Search

Overview

An instance of Towers of Hanoi




```
peg(a;b;c).                  disk(1..7).
init_on(1,a).                goal_on(3;4,a).
init_on(2;7,b).              goal_on(1;2;5;6;7,c).
init_on(3;4;5;6,c).          moves(70).
```

An encoding of Towers of Hanoi

```
on(D,P,0)     :- init_on(D,P).

1 { move(D,P,T) : disk(D) : peg(P) } 1 :- moves(M), T = 1..M.
move(D,T)     :- move(D,_,T).

on(D,P,T)     :- move(D,P,T).
on(D,P,T+1) :- on(D,P,T), not move(D,T+1), not moves(T).

blocked(D-1,P,T+1) :- on(D,P,T), not moves(T).
blocked(D-1,P,T)   :- blocked(D,P,T), disk(D).

:- move(D,P,T), blocked(D-1,P,T).
:- move(D,T), on(D,P,T-1), blocked(D,P,T).
:- not 1 { on(D,P,T) } 1, disk(D), moves(M), T = 1..M.
:- goal_on(D,P), not on(D,P,M), moves(M).
```

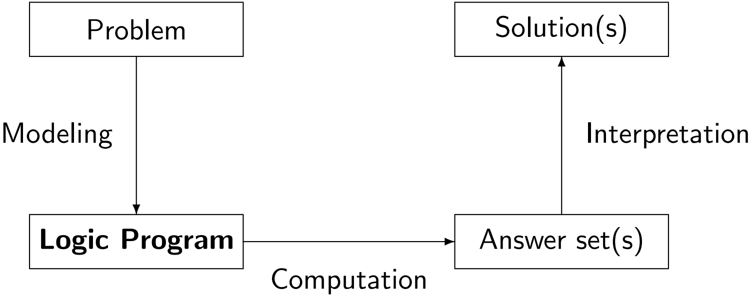
Let it run!

```
torsten@raz > gringo toh_instance.lp toh_encoding.lp | clasp --stats
clasp version 1.3.5
Reading from stdin
Solving...
Answer: 1
peg(a) peg(c) peg(b) init_on(1,a) init_on(2,b) ...
move(6,a,1)  move(7,a,2)  move(5,b,3)  move(7,c,4)
move(6,b,5)  move(7,b,6)  move(4,a,7)  move(7,a,8) ...
move(2,c,63) move(7,c,64) move(6,b,65) move(7,b,66)
move(5,c,67) move(7,a,68) move(6,c,69) move(7,c,70)
move(7,70) move(6,69) move(7,68) move(5,67) move(7,66) ...
SATISFIABLE

Models       : 1+
Time         : 3.280s (Solving: 3.23s 1st Model: 3.23s Unsat: 0.00s)
Choices      : 130907
Conflicts    : 35738
Restarts     : 12
```

Introduction: Overview

Overview

Problem solving in ASP: Syntax



Normal logic programs

- A (normal) **rule**, r, is an ordered pair of the form

$$A_0 \leftarrow A_1, \dots, A_m, not\ A_{m+1}, \dots, not\ A_n,$$

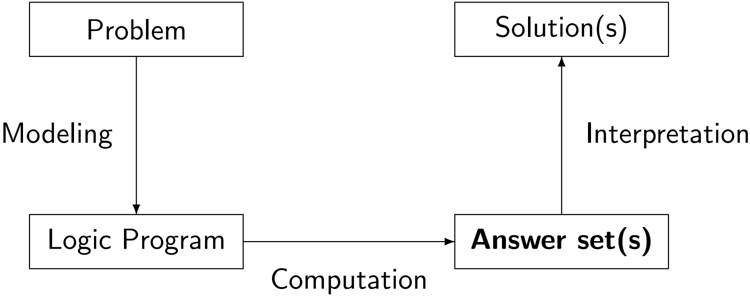
  where $n \geq m \geq 0$, and each A_i $(0 \leq i \leq n)$ is an atom.
- A (normal) **logic program** is a finite **set** of rules.
- Notation

$$\begin{array}{rcl} head(r) & = & A_0 \\ body(r) & = & \{A_1, \dots, A_m, not\ A_{m+1}, \dots, not\ A_n\} \\ body^+(r) & = & \{A_1, \dots, A_m\} \\ body^-(r) & = & \{A_{m+1}, \dots, A_n\} \end{array}$$

- A program is called **positive** if $body^-(r) = \emptyset$ for all its rules.

Overview

Problem solving in ASP: Semantics



Answer set: Formal Definition

Positive programs

- A set of atoms X is **closed under** a positive program Π iff for any $r \in \Pi$, $head(r) \in X$ whenever $body^+(r) \subseteq X$.
  - ➥ X corresponds to a model of Π (seen as a formula).
- The **smallest** set of atoms which is closed under a positive program Π is denoted by $Cn(\Pi)$.
  - ➥ $Cn(\Pi)$ corresponds to the $\subseteq$-smallest model of Π (ditto).
- The set $Cn(\Pi)$ of atoms is the **answer set** of a *positive* program Π.

Some “logical” remarks

- Positive rules are also referred to as **definite clauses**.
  - Definite clauses are disjunctions with **exactly one** positive atom:

    $$A_0 \vee \neg A_1 \vee \cdots \vee \neg A_m$$

  - A set of definite clauses has a (unique) smallest model.
- **Horn clauses** are clauses with **at most** one positive atom.
  - Every definite clause is a Horn clause but not vice versa.
  - A set of Horn clauses has a smallest model or none.
- This smallest model is the intended semantics of a set of Horn clauses.
  - ☞ Given a positive program Π, $Cn(\Pi)$ corresponds to the smallest model of the set of definite clauses corresponding to Π.

Answer set: Basic idea

Consider the logical formula Φ and its three (classical) models:

$\mathbf{\Phi}$ $\boxed{q \;\wedge\; (q \wedge \neg r \rightarrow p)}$

$\{\mathbf{p},\mathbf{q}\}, \{q,r\}$, and $\{p,q,r\}$

Formula Φ has one stable model, called **answer set**:

$\{\mathbf{p},\mathbf{q}\}$

p	$\mapsto$	1
q	$\mapsto$	1
r	$\mapsto$	0

$\mathbf{\Pi_\Phi}$

$$\begin{array}{|lll|}\hline q & \leftarrow & \\ p & \leftarrow & q,\ \mathit{not}\ r \\ \hline\end{array}$$

Informally, a set X of atoms is an **answer set** of a logic program Π

- if X is a (classical) model of Π and
- if all atoms in X are **justified** by some rule in Π

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Answer set: Formal Definition

Normal programs

- The **reduct**, Π^X, of a program Π relative to a set X of atoms is defined by

$$\Pi^X = \{\mathit{head}(r) \leftarrow \mathit{body}^+(r) \mid r \in \Pi \text{ and } \mathit{body}^-(r) \cap X = \emptyset\}.$$

- A set X of atoms is an **answer set** of a program Π if $Cn(\Pi^X) = X$.
Recall: $Cn(\Pi^X)$ is the $\subseteq$–smallest (classical) model of Π^X.

Intuition: X is **stable** under *"applying rules from Π"*

Note: Every atom in X is justified by an *"applying rule from Π"*

A closer look at Π^X

In other words, given a set X of atoms from Π,

Π^X is obtained from Π by deleting

1. each rule having a *not* A in its body with $A \in X$ and then
2. all negative atoms of the form *not* A in the bodies of the remaining rules.

Overview

A first example

$$\Pi = \{p \leftarrow p, \ q \leftarrow not \ p\}$$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✘
$\{p\}$	$p \leftarrow p$	$\emptyset$	✘
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✘

A second example

$$\Pi = \{p \leftarrow not\ q,\ \ q \leftarrow not\ p\}$$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✘
$\{p\}$	$p \leftarrow$	$\{p\}$	✔
$\{q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	✘

A third example

$\Pi = \{p \leftarrow not\ p\}$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow$	$\{p\}$	✘
$\{p\}$		$\emptyset$	✘

Answer set: Some properties

- A logic program may have zero, one, or multiple answer sets!
- If X is an answer set of a logic program Π,
  then X is a model of Π (seen as a formula).
- If X and Y are answer sets of a *normal* program Π,
  then $X \not\subset Y$.

Answer set: Alternative Definition

Let Π be a normal program and X a set of atoms.

- The set of **generating rules** of X relative to Π is defined by

$$\Pi_X = \{r \in \Pi \mid \mathit{body}^+(r) \subseteq X \text{ and } \mathit{body}^-(r) \cap X = \emptyset\}.$$

- X is an answer set of Π iff X is a $\subseteq$-minimal model of Π_X.
- Or, X is an answer set of Π iff $X \in \min_\subseteq(\Pi_X)$, where $\min_\subseteq(\Pi)$ is the set of $\subseteq$-minimal models of a program Π.

The second example revisited

$$\Pi = \{p \leftarrow \mathit{not}\ q,\ \ q \leftarrow \mathit{not}\ p\}$$

X	Π_X	*"logically"*	$\min_{\subseteq}(\Pi_X)$	
$\emptyset$	$p \leftarrow \mathit{not}\ q$ $q \leftarrow \mathit{not}\ p$	$p \vee q$	$\{p\}, \{q\}$	✘
$\{p\}$	$p \leftarrow \mathit{not}\ q$	$p \vee q$	$\{p\}, \{q\}$	✔
$\{q\}$	$q \leftarrow \mathit{not}\ p$	$p \vee q$	$\{p\}, \{q\}$	✔
$\{p, q\}$		$\top$	$\emptyset$	✘

A closer look at Cn
Inductive characterization

Let Π be a positive program and X a set of atoms.

- The **immediate consequence operator** T_Π is defined as follows:

$$T_\Pi X = \{head(r) \mid r \in \Pi \text{ and } body(r) \subseteq X\}$$

- Iterated applications of T_Π are written as T_Π^j for $j \geq 0$,
  where $T_\Pi^0 X = X$ and $T_\Pi^i X = T_\Pi T_\Pi^{i-1} X$ for $i \geq 1$.

Theorem

For any positive program Π, we have

- $Cn(\Pi) = \bigcup_{i \geq 0} T_\Pi^i \emptyset$,
- $X \subseteq Y$ *implies* $T_\Pi X \subseteq T_\Pi Y$,
- $Cn(\Pi)$ *is the smallest fixpoint of* T_Π.

Let's iterate T_Π

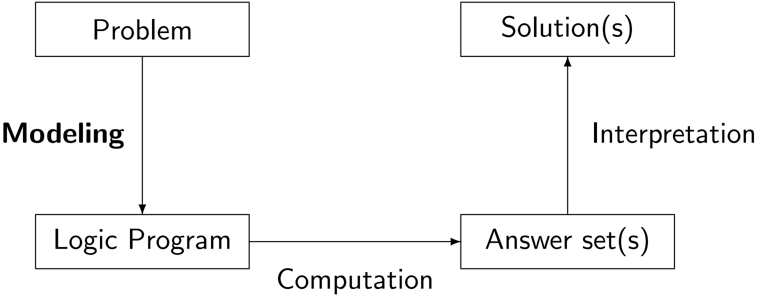
$$\Pi = \{p \leftarrow,\ q \leftarrow,\ r \leftarrow p,\ s \leftarrow q,t,\ t \leftarrow r,\ u \leftarrow v\}$$

$$\begin{array}{lllllll}
T_\Pi^0\emptyset & = & \emptyset & & & & \\
T_\Pi^1\emptyset & = & \{p,q\} & = & T_\Pi T_\Pi^0\emptyset & = & T_\Pi\emptyset \\
T_\Pi^2\emptyset & = & \{p,q,r\} & = & T_\Pi T_\Pi^1\emptyset & = & T_\Pi\{p,q\} \\
T_\Pi^3\emptyset & = & \{p,q,r,t\} & = & T_\Pi T_\Pi^2\emptyset & = & T_\Pi\{p,q,r\} \\
T_\Pi^4\emptyset & = & \{p,q,r,t,s\} & = & T_\Pi T_\Pi^3\emptyset & = & T_\Pi\{p,q,r,t\} \\
T_\Pi^5\emptyset & = & \{p,q,r,t,s\} & = & T_\Pi T_\Pi^4\emptyset & = & T_\Pi\{p,q,r,t,s\} \\
T_\Pi^6\emptyset & = & \{p,q,r,t,s\} & = & T_\Pi T_\Pi^5\emptyset & = & T_\Pi\{p,q,r,t,s\}
\end{array}$$

To see that $Cn(\Pi) = \{p,q,r,t,s\}$ is the smallest fixpoint of T_Π, note that $T_\Pi\{p,q,r,t,s\} = \{p,q,r,t,s\}$ and $T_\Pi X \neq X$ for every $X \subseteq \{p,q,r,t,s\}$.

Overview

Problem solving in ASP: Modeling



(Rough) notational convention

We sometimes use the following notation interchangeably in order to stress the respective view:

	if	and	or	negation as failure	classical negation
source code	`:-`	`,`	`|`	`not`	`-`
logic program	$\leftarrow$	$,$	$;$	*not*/$\sim$	$\neg$
formula	$\rightarrow$	$\wedge$	$\vee$	$\sim/(\neg)$	$\neg$

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ stands for
    `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals
  - `p :-` **q(X) : r(X)** given **r(a)**, **r(b)**, **r(c)** stands for
    `p :-` **q(a), q(b), q(c)**
- Disjunction
  - **p(X) — q(X)** `:- r(X)`
- Integrity Constraints
  - `:- q(X), p(X)`
- Choice
  - **2** $\{$ **p(X,Y) : q(X)** $\}$ **7** `:- r(Y)`
- Aggregates
  - `s(Y) :- r(Y),` **2 #count** $\{$ **p(X,Y) : q(X)** $\}$ **7**
  - also: `#sum, #avg, #min, #max, #even, #odd`

Overview

Programs with Variables

Let Π be a logic program.

- **Herbranduniverse** U^Π: Set of constants in Π
- **Herbrandbase** B^Π: Set of (variable-free) atoms constructible from U^Π

  ☞ We usually denote this as $\mathcal{A}$, and call it **alphabet**.
- Ground Instances of $r \in \Pi$: Set of variable-free rules obtained by replacing all variables in r by elements from U^Π:

  $$ground(r) = \{r\theta \mid \theta : var(r) \to U^\Pi\}$$

  where $var(r)$ stands for the set of all variables occurring in r; θ is a (ground) substitution.
- **Ground Instantiation** of Π:

  $$ground(\Pi) = \bigcup_{r\in\Pi} ground(r)$$

An example

$$\Pi = \{\ r(a,b) \leftarrow,\ \ r(b,c) \leftarrow,\ \ t(X,Y) \leftarrow r(X,Y)\ \}$$

$$U^{\Pi} = \{a,b,c\}$$

$$B^{\Pi} = \left\{ \begin{array}{l} r(a,a), r(a,b), r(a,c), r(b,a), r(b,b), r(b,c), r(c,a), r(c,b), r(c,c), \\ t(a,a), t(a,b), t(a,c), t(b,a), t(b,b), t(b,c), t(c,a), t(c,b), t(c,c) \end{array} \right\}$$

$$ground(\Pi) = \left\{ \begin{array}{lllll} r(a,b) \leftarrow , & & & & \\ r(b,c) \leftarrow , & & & & \\ t(a,a) \leftarrow r(a,a), & t(b,a) \leftarrow r(b,a), & t(c,a) \leftarrow r(c,a), \\ t(a,b) \leftarrow r(a,b), & t(b,b) \leftarrow r(b,b), & t(c,b) \leftarrow r(c,b), \\ t(a,c) \leftarrow r(a,c), & t(b,c) \leftarrow r(b,c), & t(c,c) \leftarrow r(c,c) \end{array} \right\}$$

☞ **Intelligent Grounding** aims at reducing the ground instantiation.

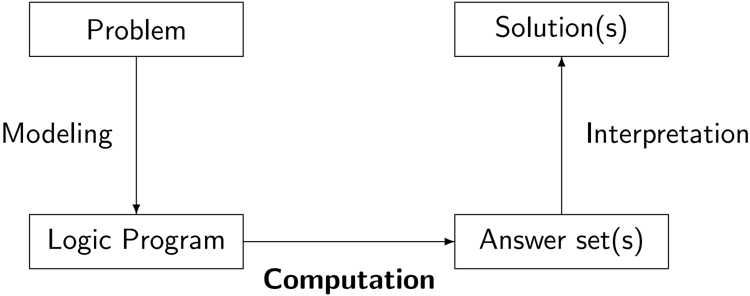
Answer sets of programs with Variables

Let Π be a normal logic program with variables.

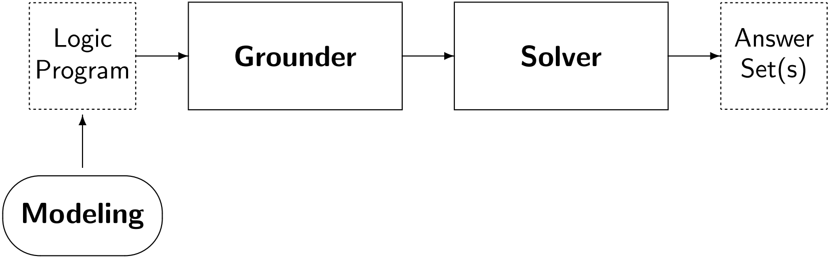
We define a set X of (**ground**) atoms as an **answer set** of Π
if $Cn(ground(\Pi)^X) = X$.

Overview

Problem solving in ASP: Computation



ASP Solving Process



Traditional Solving Procedure

Global parameters: Logic program Π and its set $\mathcal{A}$ of atoms.

$solve_\Pi(X, Y)$

$(X, Y) \leftarrow propagate_\Pi(X, Y)$
if $(X \cap Y) \neq \emptyset$ **then fail**
if $(X \cup Y) = \mathcal{A}$ **then return**(X)
select $A \in \mathcal{A} \setminus (X \cup Y)$
$solve_\Pi(X \cup \{A\}, Y)$
$solve_\Pi(X, Y \cup \{A\})$

Comments:

- (X, Y) is supposed to be a 3-valued model such that $X \subseteq Z$ and $Y \cap Z = \emptyset$ for any answer set Z of Π.
- Key operations: $propagate_\Pi(X, Y)$ and '**select** $A \in \mathcal{A} \setminus (X \cup Y)$'
- Worst case complexity: $\mathcal{O}(2^{|\mathcal{A}|})$

Overview

Reasoning Modes

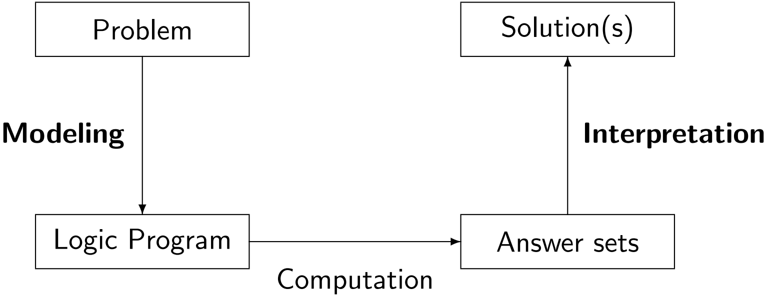
- Satisfiability
- Enumeration†
- Projection†
- Intersection‡
- Union‡
- Optimization
- Sampling

† without solution recording

‡ without solution enumeration

Basic Modeling: Overview

Modeling and Interpreting



Modeling

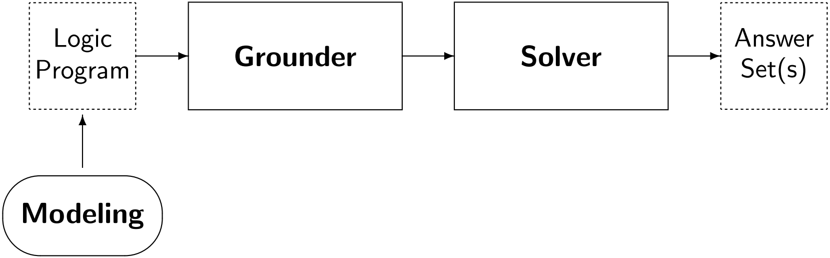
For solving a problem class **P** for a problem instance **I**,
encode

1. the problem instance **I** as a set **C(I)** of facts and
2. the problem class **P** as a set **C(P)** of rules

such that the solutions to **P** for **I** can be (polynomially) extracted from the answer sets of $\mathbf{C(I)} \cup \mathbf{C(P)}$.

Overview

ASP Solving Process



Overview

Graph Coloring

```
node(1..6).

edge(1,2).  edge(1,3).  edge(1,4).
edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).
edge(4,1).  edge(4,2).
edge(5,3).  edge(5,4).  edge(5,6).
edge(6,2).  edge(6,3).  edge(6,5).

col(r).   col(b).   col(g).

1 {color(X,C) : col(C)} 1 :- node(X).

:- edge(X,Y), color(X,C), color(Y,C).
```

Graph Coloring: Grounding

$ gringo -t color.lp

```
node(1).  node(2).  node(3).  node(4).  node(5).  node(6).

edge(1,2).  edge(1,3).  edge(1,4).  edge(2,4).  edge(2,5).  edge(2,6).
edge(3,1).  edge(3,4).  edge(3,5).  edge(4,1).  edge(4,2).  edge(5,3).
edge(5,4).  edge(5,6).  edge(6,2).  edge(6,3).  edge(6,5).

col(r).  col(b).  col(g).

1 {color(1,r), color(1,b), color(1,g)} 1.
1 {color(2,r), color(2,b), color(2,g)} 1.
1 {color(3,r), color(3,b), color(3,g)} 1.
1 {color(4,r), color(4,b), color(4,g)} 1.
1 {color(5,r), color(5,b), color(5,g)} 1.
1 {color(6,r), color(6,b), color(6,g)} 1.

 :- color(1,r), color(2,r).  :- color(2,g), color(5,g). ...  :- color(6,r), color(2,r).
 :- color(1,b), color(2,b).  :- color(2,r), color(6,r).       :- color(6,b), color(2,b).
 :- color(1,g), color(2,g).  :- color(2,b), color(6,b).       :- color(6,g), color(2,g).
 :- color(1,r), color(3,r).  :- color(2,g), color(6,g).       :- color(6,r), color(3,r).
 :- color(1,b), color(3,b).  :- color(3,r), color(1,r).       :- color(6,b), color(3,b).
 :- color(1,g), color(3,g).  :- color(3,b), color(1,b).       :- color(6,g), color(3,g).
 :- color(1,r), color(4,r).  :- color(3,g), color(1,g).       :- color(6,r), color(5,r).
 :- color(1,b), color(4,b).  :- color(3,r), color(4,r).       :- color(6,b), color(5,b).
 :- color(1,g), color(4,g).  :- color(3,b), color(4,b).       :- color(6,g), color(5,g).
 :- color(2,r), color(4,r).  :- color(3,g), color(4,g).
 :- color(2,b), color(4,b).  :- color(3,r), color(5,r).
 :- color(2,g), color(4,g).  :- color(3,b), color(5,b).
```

Graph Coloring: Solving

```
$ gringo color.lp | clasp 0
```

```
clasp version 1.2.1
Reading from stdin
Reading      : Done(0.000s)
Preprocessing: Done(0.000s)
Solving...
Answer: 1
color(1,b) color(2,r) color(3,r) color(4,g) color(5,b) color(6,g) node(1) ... edge(1,2) ... col(r) ...
Answer: 2
color(1,g) color(2,r) color(3,r) color(4,b) color(5,g) color(6,b) node(1) ... edge(1,2) ... col(r) ...
Answer: 3
color(1,b) color(2,g) color(3,g) color(4,r) color(5,b) color(6,r) node(1) ... edge(1,2) ... col(r) ...
Answer: 4
color(1,g) color(2,b) color(3,b) color(4,r) color(5,g) color(6,r) node(1) ... edge(1,2) ... col(r) ...
Answer: 5
color(1,r) color(2,b) color(3,b) color(4,g) color(5,r) color(6,g) node(1) ... edge(1,2) ... col(r) ...
Answer: 6
color(1,r) color(2,g) color(3,g) color(4,b) color(5,r) color(6,b) node(1) ... edge(1,2) ... col(r) ...

Models      : 6
Time        : 0.000  (Solving: 0.000)
```

Overview

Basic Methodology

Generate and Test (or: Guess and Check) approach

Generator Generate potential answer set candidates
(typically through non-deterministic constructs)

Tester Eliminate invalid candidates
(typically through integrity constraints)

Nutshell

$$\text{Logic program} \quad = \quad \text{Data} + \text{Generator} + \text{Tester}$$
$$(+ \text{Optimizer})$$

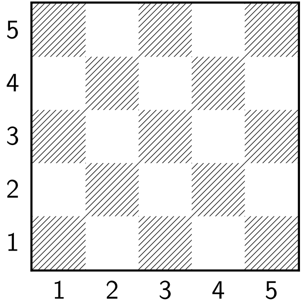
Satisfiability

- Problem Instance: A propositional formula ϕ.
- Problem Class: Is there an assignment of propositional variables to true and false such that a given formula ϕ is true.

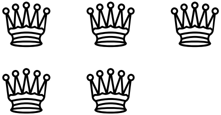
- Example: Consider formula $(a \vee \neg b) \wedge (\neg a \vee b)$.
- Logic Program:

Generator	**Tester**	**Answer sets**
$\{a,b\} \leftarrow$	$\leftarrow$ not a, b	$X_1 = \{a,b\}$
	$\leftarrow$ a, not b	$X_2 = \{\}$

The n-Queens Problem



- Place n queens on an $n \times n$ chess board
- Queens must not attack one another



Defining the Field

queens.lp

```
row(1..n).
col(1..n).
```

- Create file queens.lp
- Define the field
  - n rows
  - n columns

Defining the Field

Running ...

```
$ clingo queens.lp -c n=5
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5)
SATISFIABLE

Models       : 1
Time         : 0.000
  Prepare    : 0.000
  Prepro.    : 0.000
  Solving    : 0.000
```

Placing some Queens

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
```

- Guess a solution candidate
- Place some queens on the board

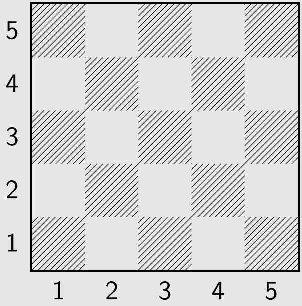
Placing some Queens

Running ...

```
$ clingo queens.lp -c n=5 3
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5)
Answer: 2
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) queen(1,1)
Answer: 3
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) queen(2,1)
SATISFIABLE

Models        : 3+
```

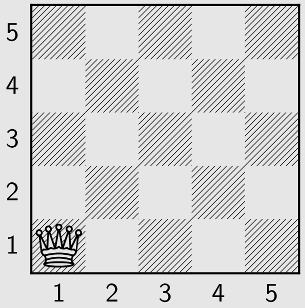
Placing some Queens: Answer 1

Answer 1



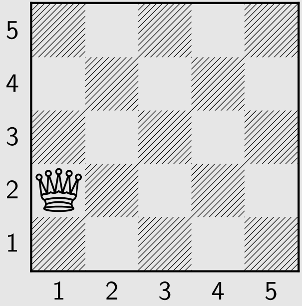
Placing some Queens: Answer 2

Answer 2



Placing some Queens: Answer 3

Answer 3



Placing n Queens

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
```
:- not { queen(I,J) } == n.

- Place exactly n queens on the board

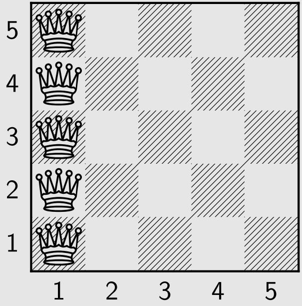
Placing n Queens

Running ...

```
$ clingo queens.lp -c n=5 2
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(5,1) queen(4,1) queen(3,1) \
queen(2,1) queen(1,1)
Answer: 2
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(1,2) queen(4,1) queen(3,1) \
queen(2,1) queen(1,1)
...
```

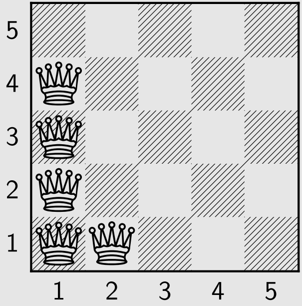
Placing n Queens: Answer 1

Answer 1



Placing *n* Queens: Answer 2

Answer 2



Horizontal and vertical Attack

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
:- not { queen(I,J) } == n.
:- queen(I,J), queen(I,JJ), J != JJ.
:- queen(I,J), queen(II,J), I != II.
```

- Forbid horizontal attacks
- Forbid vertical attacks

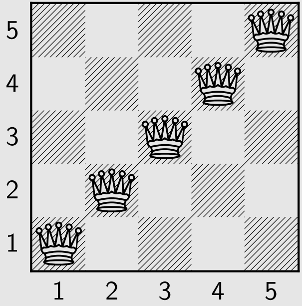
Horizontal and vertical Attack

Running ...

```
$ clingo queens.lp -c n=5
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(5,5) queen(4,4) queen(3,3) \
queen(2,2) queen(1,1)
...
```

Horizontal and vertical Attack: Answer 1

Answer 1



Diagonal Attack

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
:- not { queen(I,J) } == n.
:- queen(I,J), queen(I,JJ), J != JJ.
:- queen(I,J), queen(II,J), I != II.
:- queen(I,J), queen(II,JJ), (I,J) != (II,JJ),
   I-J == II-JJ.
:- queen(I,J), queen(II,JJ), (I,J) != (II,JJ),
   I+J == II+JJ.
```

- Forbid diagonal attacks

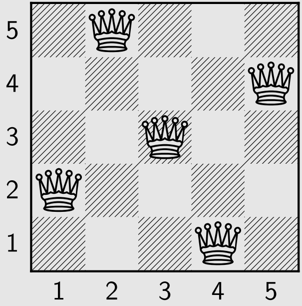
Diagonal Attack

Running ...

```
$ clingo queens.lp -c n=5
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(4,5) queen(1,4) queen(3,3) \
queen(5,2) queen(2,1)
SATISFIABLE

Models      : 1+
Time        : 0.000
  Prepare   : 0.000
  Prepro.   : 0.000
  Solving   : 0.000
```

Diagonal Attack: Answer 1

Answer 1



Optimizing

queens-opt.lp

```
{ queen(I,1..n) } == 1 :- I = 1..n.
{ queen(1..n,J) } == 1 :- J = 1..n.
:- { queen(D-J,J) } >= 2, D =   2..2*n.
:- { queen(D+J,J) } >= 2, D = 1-n..n-1.
```

- Encoding can be optimized
- Much faster to solve

- See Section *Tweaking N-Queens*

Reviewer Assignment

by Ilkka Niemelä

```
reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p5).
reviewer(r2). paper(p2). classA(r1,p3). classB(r1,p4). coi(r1,p6).
...

3 { assigned(P,R) : reviewer(R) } 3 :-  paper(P).

 :- assigned(P,R), coi(R,P).
 :- assigned(P,R), not classA(R,P), not classB(R,P).
 :- 9 { assigned(P,R) : paper(P) }  , reviewer(R).
 :-   { assigned(P,R) : paper(P) } 6, reviewer(R).

assignedB(P,R) :-  classB(R,P), assigned(P,R).
 :- 3 { assignedB(P,R) : paper(P) }, reviewer(R).

#minimize { assignedB(P,R) : paper(P) : reviewer(R) }.
```

Simplistic STRIPS Planning

```
fluent(p).      fluent(q).     fluent(r).
action(a).      pre(a,p).      add(a,q).      del(a,p).
action(b).      pre(b,q).      add(b,r).      del(b,q).
init(p).        query(r).

time(1..k).     lasttime(T) :- time(T), not time(T+1).

holds(P,0) :- init(P).

1 { occ(A,T) : action(A) } 1 :- time(T).
 :- occ(A,T), pre(A,F), not holds(F,T-1).

ocdel(F,T) :- occ(A,T), del(A,F).
holds(F,T) :- occ(A,T), add(A,F).
holds(F,T) :- holds(F,T-1), not ocdel(F,T), time(T).

 :- query(F), not holds(F,T), lasttime(T).
```

Simplistic STRIPS Planning with iASP

```
#base.

fluent(p).      fluent(q).      fluent(r).
action(a).      pre(a,p).       add(a,q).       del(a,p).
action(b).      pre(b,q).       add(b,r).       del(b,q).
init(p).        query(r).

holds(P,0) :- init(P).

#cumulative t.

1 { occ(A,t) : action(A) } 1.
 :- occ(A,t), pre(A,F), not holds(F,t-1).

ocdel(F,t) :- occ(A,t), del(A,F).
holds(F,t) :- occ(A,t), add(A,F).
holds(F,t) :- holds(F,t-1), not ocdel(F,t).

#volatile t.

 :- query(F), not holds(F,t).
```

Disjunctive logic programs: Overview

Overview

Disjunctive logic programs

- A **disjunctive rule**, r, is an ordered pair of the form

$$A_1 \,; \ldots ; A_m \leftarrow A_{m+1}, \ldots, A_n, not\ A_{n+1}, \ldots, not\ A_o,$$

  where $o \geq n \geq m \geq 0$, and each A_i $(0 \leq i \leq o)$ is an atom.
- A **disjunctive logic program** is a finite set of disjunctive rules.
- (Generalized) Notation

$$\begin{array}{rcl} head(r) & = & \{A_1, \ldots, A_m\} \\ body(r) & = & \{A_{m+1}, \ldots, A_n, not\ A_{n+1}, \ldots, not\ A_o\} \\ body^+(r) & = & \{A_{m+1}, \ldots, A_n\} \\ body^-(r) & = & \{A_{n+1}, \ldots, A_o\} \end{array}$$

- A program is called **positive** if $body^-(r) = \emptyset$ for all its rules.

Overview

Answer sets

- Positive programs:
  - A set X of atoms is **closed under** a positive program Π iff for any $r \in \Pi$, $head(r) \cap X \neq \emptyset$ whenever $body^+(r) \subseteq X$.
    - ➥ X corresponds to a model of Π (seen as a formula).
  - The set of all $\subseteq$-minimal sets of atoms being closed under a positive program Π is denoted by $\min_\subseteq(\Pi)$.
    - ➥ $\min_\subseteq(\Pi)$ corresponds to the $\subseteq$-minimal models of Π (ditto).
- Disjunctive programs:
  - The **reduct**, Π^X, of a disjunctive program Π relative to a set X of atoms is defined by

$$\Pi^X = \{head(r) \leftarrow body^+(r) \mid r \in \Pi \text{ and } body^-(r) \cap X = \emptyset\}.$$

  - A set X of atoms is an **answer set** of a disjunctive program Π if $X \in \min_\subseteq(\Pi^X)$.
- FYI: The alternative definition on Page 43 is applicable as well.

Overview

A "positive" example

$$\Pi = \left\{ \begin{array}{lcl} a & \leftarrow & \\ b\,;c & \leftarrow & a \end{array} \right\}$$

- The sets $\{a, b\}$, $\{a, c\}$, and $\{a, b, c\}$ are closed under Π.
- We have $\min_{\subseteq}(\Pi) = \{\ \{a, b\},\ \{a, c\}\ \}$.

3-colorability revisited

```
node(1..6).

edge(1,2;3;4).  edge(2,4;5;6).  edge(3,1;4;5).
edge(4,1;2).    edge(5,3;4;6).  edge(6,2;3;5).

colored(X,r) | colored(X,b) | colored(X,g) :- node(X).
:- edge(X,Y), color(X,C), color(Y,C).

col(r). col(b). col(g).
colored(X,C) : col(X) :- node(X).
:- edge(X,Y), color(X,C), color(Y,C).
```

More Examples

- $\Pi_1 = \{a\,;b\,;c \leftarrow\}$ has answer sets $\{a\}$, $\{b\}$, and $\{c\}$.
- $\Pi_2 = \{a\,;b\,;c \leftarrow\ ,\ \leftarrow a\}$ has answer sets $\{b\}$ and $\{c\}$.
- $\Pi_3 = \{a\,;b\,;c \leftarrow\ ,\ \leftarrow a\ ,\ b \leftarrow c\ ,\ c \leftarrow b\}$ has answer set $\{b, c\}$.
- $\Pi_4 = \{a\,;b \leftarrow c\ ,\ b \leftarrow \mathit{not}\ a, \mathit{not}\ c\ ,\ a\,;c \leftarrow \mathit{not}\ b\}$
  has answer sets $\{a\}$ and $\{b\}$.

Answer set: Some properties

- A disjunctive logic program may have zero, one, or multiple answer sets.
- If X is an answer set of a disjunctive logic program Π, then X is a model of Π (seen as a formula).
- If X and Y are answer sets of a disjunctive logic program Π, then $X \not\subset Y$.
- If $A \in X$ for some answer X set of a disjunctive logic program Π, then there is a rule $r \in \Pi_X$ such that $\{A\} = head(r) \cap X$.

An example with variables

$$\Pi = \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(X)\,;c(Y) & \leftarrow & a(X,Y), \mathit{not}\ c(Y) \end{array} \right\}$$

$$\mathit{ground}(\Pi) = \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(1)\,;c(1) & \leftarrow & a(1,1), \mathit{not}\ c(1) \\ b(1)\,;c(2) & \leftarrow & a(1,2), \mathit{not}\ c(2) \\ b(2)\,;c(1) & \leftarrow & a(2,1), \mathit{not}\ c(1) \\ b(2)\,;c(2) & \leftarrow & a(2,2), \mathit{not}\ c(2) \end{array} \right\}$$

For every answer set X of Π, we have

- $a(1,2) \in X$ and
- $\{a(1,1), a(2,1), a(2,2)\} \cap X = \emptyset$.

An example with variables

$$
ground(\Pi)^X \quad = \quad \left\{ \begin{array}{lcl}
a(1,2) & \leftarrow & \\
b(1)\,;c(1) & \leftarrow & a(1,1), \mathit{not}\ c(1) \\
b(1)\,;c(2) & \leftarrow & a(1,2), \mathit{not}\ c(2) \\
b(2)\,;c(1) & \leftarrow & a(2,1), \mathit{not}\ c(1) \\
b(2)\,;c(2) & \leftarrow & a(2,2), \mathit{not}\ c(2)
\end{array} \right\}
$$

- Consider $X = \{a(1,2),\ b(1)\}$.
- We get $\min_{\subseteq}(ground(\Pi)^X) = \{\ \{a(1,2),\ b(1)\},\ \{a(1,2),\ c(2)\}\ \}$.
- X is an answer set of Π because $X \in \min_{\subseteq}(ground(\Pi)^X)$.

An example with variables

$$ground(\Pi)^X = \left\{ \begin{array}{lcl} a(1,2) & \leftarrow & \\ b(1)\,;c(1) & \leftarrow & a(1,1), not\ c(1) \\ b(1)\,;c(2) & \leftarrow & a(1,2), not\ c(2) \\ b(2)\,;c(1) & \leftarrow & a(2,1), not\ c(1) \\ b(2)\,;c(2) & \leftarrow & a(2,2), not\ c(2) \end{array} \right\}$$

- Consider $X = \{a(1,2),\ c(2)\}$.
- We get $\min_{\subseteq}(ground(\Pi)^X) = \{\ \{a(1,2)\}\ \}$.
- X is no answer set of Π because $X \notin \min_{\subseteq}(ground(\Pi)^X)$.

Nested logic programs: Overview

Overview

Nested logic programs

- Formulas are formed from
  - propositional atoms and
  - $\top$ and $\bot$

  using
  - negation-as-failure (*not*),
  - conjunction (,), and
  - disjunction (;).
- A **nested rule**, r, is an ordered pair of the form $F \leftarrow G$ where F and G are formulas.
- A **nested program** is a finite set of rules.
- Notation: $head(r) = F$ and $body(r) = G$.

Overview

Satisfaction relation

- The **satisfaction relation** $X \models F$ between a set of atoms and a formula F is defined recursively as follows:
  - $X \models F$ if $F \in X$ for an atom F,
  - $X \models \top$,
  - $X \not\models \bot$,
  - $X \models (F, G)$ if $X \models F$ and $X \models G$,
  - $X \models (F; G)$ if $X \models F$ or $X \models G$,
  - $X \models \mathit{not}\ F$ if $X \not\models F$.
- A set X of atoms satisfies a nested program Π, written $X \models \Pi$, iff for any $r \in \Pi$, $X \models \mathit{head}(r)$ whenever $X \models \mathit{body}(r)$.
- The set of all $\subseteq$-minimal sets of atoms satisfying program Π is denoted by $\min_{\subseteq}(\Pi)$.

Reduct

- The **reduct**, F^X, of a formula F relative to a set X of atoms is defined recursively as follows:
  - $F^X = F$ if F is an atom or $\top$ or $\bot$,
  - $(F, G)^X = (F^X, G^X)$,
  - $(F; G)^X = (F^X; G^X)$,
  - $(\mathit{not}\ F)^X = \begin{cases} \bot & \text{if } X \models F \\ \top & \text{otherwise} \end{cases}$
- The **reduct**, Π^X, of a nested program Π relative to a set X of atoms is defined by

$$\Pi^X = \{head(r)^X \leftarrow body(r)^X \mid r \in \Pi\}.$$

- A set X of atoms is an **answer set** of a nested program Π if $X \in \min_{\subseteq}(\Pi^X)$.

Overview

Two examples

- $\Pi_1 = \{(p\,; not\;\;p) \leftarrow \top\}$
  - For $X = \emptyset$, we get
    - $\Pi_1^\emptyset = \{(p\,;\top) \leftarrow \top\}$
    - $\min_\subseteq(\Pi_1^\emptyset) = \{\emptyset\}$. ✔
  - For $X = \{p\}$, we get
    - $\Pi_1^{\{p\}} = \{(p\,;\bot) \leftarrow \top\}$
    - $\min_\subseteq(\Pi_1^{\{p\}}) = \{\{p\}\}$. ✔
- $\Pi_2 = \{p \leftarrow not\;\;not\;\;p\}$
  - For $X = \emptyset$, we get $\Pi_2^\emptyset = \{p \leftarrow \bot\}$ and $\min_\subseteq(\Pi_2^\emptyset) = \{\emptyset\}$. ✔
  - For $X = \{p\}$, we get $\Pi_2^{\{p\}} = \{p \leftarrow \top\}$ and $\min_\subseteq(\Pi_2^{\{p\}}) = \{\{p\}\}$. ✔
- In general,
  - $F \leftarrow G,\; not\;\;not\;\;H$ is equivalent to $F\,; not\;\;H \leftarrow G$
  - $F\,;\; not\;\;not\;\;G \leftarrow H$ is equivalent to $F \leftarrow H, not\;\;G$
  - $not\;\;not\;\;not\;\;F$ is equivalent to $not\;\;F$

➥ Intuitionistic Logics HT (Heyting, 1930) and G3 (Gödel, 1932)

Some more examples

$$\begin{aligned}
\Pi_3 &= \{p \leftarrow (q, r) \,;(\mathit{not}\ q, \mathit{not}\ s)\} \\
\Pi_4 &= \{(p \,; \mathit{not}\ p), (q \,; \mathit{not}\ q), (r \,; \mathit{not}\ r) \leftarrow \top\} \\
\Pi_5 &= \{(p \,; \mathit{not}\ p), (q \,; \mathit{not}\ q), (r \,; \mathit{not}\ r) \leftarrow \top,\ \bot \leftarrow p, q\}
\end{aligned}$$

Propositional Theories: Overview

Overview

Propositional theories

- Formulas are formed from
  - propositional atoms and
  - $\bot$

  using
  - conjunction ($\wedge$),
  - disjunction ($\vee$), and
  - implication ($\rightarrow$).
- Notation

$$\begin{array}{rcll} \top & = & (\bot \rightarrow \bot) & \\ \sim F & = & (F \rightarrow \bot) & \qquad (\text{or: } \textit{not } F) \end{array}$$

- A **propositional theory** is a finite set of formulas.

Overview

Reduct

- The satisfaction relation $X \models F$ between a set X of atoms and a (set of) formula(s) F is defined as in propositional logic.
- The **reduct**, F^X, of a formula F relative to a set X of atoms is defined recursively as follows:
  - $F^X = \bot$ if $X \not\models F$
  - $F^X = F$ if $F \in X$
  - $F^X = (G^X \circ H^X)$ if $X \models F$ and $F = (G \circ H)$ for $\circ \in \{\wedge, \vee, \rightarrow\}$

  ➥ If $F = {\sim} G = (G \rightarrow \bot)$,
  then $F^X = (\bot \rightarrow \bot) = \top$, if $X \not\models G$, and $F^X = \bot$, otherwise.
- The **reduct**, $\mathcal{F}^X$, of a propositional theory $\mathcal{F}$ relative to a set X of atoms is defined as

$$\mathcal{F}^X = \{F^X \mid F \in \mathcal{F}\}.$$

Answer sets

- The set of all $\subseteq$-minimal sets of atoms satisfying a propositional theory $\mathcal{F}$ is denoted by $\min_{\subseteq}(\mathcal{F})$.
- A set X of atoms is an **answer set** of a propositional theory $\mathcal{F}$ if $X \in \min_{\subseteq}(\mathcal{F}^X)$.
- If X is an answer set of $\mathcal{F}$, then
  - $X \models \mathcal{F}$ and
  - $\min_{\subseteq}(\mathcal{F}^X) = \{X\}$.

  ☞ In general, this does not imply $X \in \min_{\subseteq}(\mathcal{F})$!

Overview

Two examples

- $\mathcal{F}_1 = \{p \vee (p \rightarrow (q \wedge r))\}$
  - For $X = \{p, q, r\}$, we get
    $\mathcal{F}_1^{\{p,q,r\}} = \{p \vee (p \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\mathcal{F}_1^{\{p,q,r\}}) = \{\emptyset\}$. ✘
  - For $X = \emptyset$, we get
    $\mathcal{F}_1^{\emptyset} = \{\bot \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\mathcal{F}_1^{\emptyset}) = \{\emptyset\}$. ✔

- $\mathcal{F}_2 = \{p \vee (\sim p \rightarrow (q \wedge r))\}$
  - For $X = \emptyset$, we get
    $\mathcal{F}_2^{\emptyset} = \{\bot\}$ and $\min_{\subseteq}(\mathcal{F}_2^{\emptyset}) = \emptyset$. ✘
  - For $X = \{p\}$, we get
    $\mathcal{F}_2^{\{p\}} = \{p \vee (\bot \rightarrow \bot)\}$ and $\min_{\subseteq}(\mathcal{F}_2^{\{p\}}) = \{\emptyset\}$. ✘
  - For $X = \{q, r\}$, we get
    $\mathcal{F}_2^{\{q,r\}} = \{\bot \vee (\top \rightarrow (q \wedge r))\}$ and $\min_{\subseteq}(\mathcal{F}_2^{\{q,r\}}) = \{\{q, r\}\}$. ✔

Overview

Relationship with logic programs

- The translation, $\tau[(F \leftarrow G)]$, of a (nested) rule $(F \leftarrow G)$ is defined recursively as follows:
  - $\tau[(F \leftarrow G)] = (\tau[G] \rightarrow \tau[F])$,
  - $\tau[\bot] = \bot$,
  - $\tau[\top] = \top$,
  - $\tau[F] = F \qquad$ if F is an atom,
  - $\tau[not\ F] = \sim\tau[F]$,
  - $\tau[(F, G)] = (\tau[F] \wedge \tau[G])$,
  - $\tau[(F; G)] = (\tau[F] \vee \tau[G])$.

- The translation of a logic program Π is $\tau[\Pi] = \{\tau[r] \mid r \in \Pi\}$.
  - ➥ Given a logic program Π and a set X of atoms, X is an answer set of Π iff X is an answer set of $\tau[\Pi]$.

Logic programs as propositional theories

- The normal logic program $\Pi = \{p \leftarrow \mathit{not}\ q,\ q \leftarrow \mathit{not}\ p\}$ corresponds to $\tau[\Pi] = \{\sim q \rightarrow p,\ \sim p \rightarrow q\}$.
  - ➥ Answer sets: $\{p\}$ and $\{q\}$

- The disjunctive logic program $\Pi = \{p\,;q \leftarrow\}$ corresponds to $\tau[\Pi] = \{\top \rightarrow p \vee q\}$.
  - ➥ Answer sets: $\{p\}$ and $\{q\}$

- The nested logic program $\Pi = \{p \leftarrow \mathit{not}\ \mathit{not}\ p\}$ corresponds to $\tau[\Pi] = \{\sim\sim p \rightarrow p\}$.
  - ➥ Answer sets: $\emptyset$ and $\{p\}$

Classical Negation: Overview

Overview

Syntax

Status quo

- In logic programs *not* (or $\sim$) denotes **default negation**.

Generalization

- We allow **classical negation** for atoms (only!).
  - ➥ Logic programs in "negation normal form."
- Given an alphabet $\mathcal{A}$ of atoms, let $\overline{\mathcal{A}} = \{\neg A \mid A \in \mathcal{A}\}$.
  - ☞ We assume $\mathcal{A} \cap \overline{\mathcal{A}} = \emptyset$.
- The atoms A and $\neg A$ are **complementary**.
  - ➥ $\neg A$ is the classical negation of A, and vice versa.

Overview

Semantics

- A set X of atoms is **consistent**, if $X \cap \{\neg A \mid A \in (\mathcal{A} \cap X)\} = \emptyset$, and **inconsistent**, otherwise.
- A set X of atoms is an **answer set** of a logic program Π over $\mathcal{A} \cup \overline{\mathcal{A}}$ if X is an answer set of $\Pi \cup \{B \leftarrow A, \neg A \mid A \in \mathcal{A}, B \in (\mathcal{A} \cup \overline{\mathcal{A}})\}$
  - ➥ The only inconsistent answer set (candidate) is $X = \mathcal{A} \cup \overline{\mathcal{A}}$.
- For a normal or disjunctive logic program Π over $\mathcal{A} \cup \overline{\mathcal{A}}$, exactly one of the following two cases applies:
  1. All answer sets of Π are consistent or
  2. $X = \mathcal{A} \cup \overline{\mathcal{A}}$ is the only answer set of Π.

Overview

To cross or not to cross...?

- $\Pi_1 = \{cross \leftarrow not\ train\}$
  - Answer set: $\{cross\}$
- $\Pi_2 = \{cross \leftarrow \neg train\}$
  - Answer set: $\emptyset$
- $\Pi_3 = \{cross \leftarrow \neg train,\ \neg train \leftarrow\}$
  - Answer set: $\{cross, \neg train\}$
- $\Pi_4 = \{cross \leftarrow \neg train,\ \neg train \leftarrow,\ \neg cross \leftarrow\}$
  - Answer set: $\{cross, \neg cross, train, \neg train\}$
- $\Pi_5 = \{cross \leftarrow \neg train,\ \neg train \leftarrow not\ train,\ \neg cross \leftarrow\}$
  - No answer set

Example

- $\Pi = \{p \leftarrow,\ \neg p \leftarrow,\ q \leftarrow \mathit{not}\ r\}$
  $\Pi' = \Pi \cup \{A \leftarrow (B, \neg B),\ \neg A \leftarrow (B, \neg B) \mid A, B \in \{p, q, r\}\}$
  Answer set: $\{p, \neg p, q, \neg q, r, \neg r\}$
- $\Pi = \{p\,;q \leftarrow,\ r \leftarrow p,\ \neg r \leftarrow p\ \}$
  $\Pi' = \Pi \cup \{A \leftarrow (B, \neg B),\ \neg A \leftarrow (B, \neg B) \mid A, B \in \{p, q, r\}\}$
  Answer set: $\{q\}$
- $\Pi = \{p\,; \mathit{not}\ p \leftarrow \top,\ \neg p\,; \mathit{not}\ q \leftarrow \top,\ q\,; \mathit{not}\ q \leftarrow \top\}$
  $\Pi' = \Pi \cup \{A \leftarrow (B, \neg B),\ \neg A \leftarrow (B, \neg B) \mid A, B \in \{p, q\}\}$
  Answer sets: $\emptyset$, $\{p\}$, $\{\neg p, q\}$, and $\{p, \neg p, q, \neg q\}$

Complexity

Let A be an atom and X be a set of atoms.

- For a **positive normal** logic program Π:
  - Deciding whether X is the answer set of Π is **P**-complete.
  - Deciding whether A is in the answer set of Π is **P**-complete.
- For a **normal** logic program Π:
  - Deciding whether X is an answer set of Π is **P**-complete.
  - Deciding whether A is in an answer set of Π is **NP**-complete.

Complexity (ctd)

- For a **positive disjunctive** logic program Π:
  - Deciding whether X is an answer set of Π is **co-NP**-complete.
  - Deciding whether A is in an answer set of Π is $\mathbf{NP}^{\mathbf{NP}}$-complete.
- For a **disjunctive** logic program Π:
  - Deciding whether X is an answer set of Π is **co-NP**-complete.
  - Deciding whether A is in an answer set of Π is $\mathbf{NP}^{\mathbf{NP}}$-complete.
- For a **nested** logic program Π:
  - Deciding whether X is an answer set of Π is **co-NP**-complete.
  - Deciding whether A is in an answer set of Π is $\mathbf{NP}^{\mathbf{NP}}$-complete.
- For a **propositional theory** $\mathcal{F}$:
  - Deciding whether X is an answer set of $\mathcal{F}$ is **co-NP**-complete.
  - Deciding whether A is in an answer set of $\mathcal{F}$ is $\mathbf{NP}^{\mathbf{NP}}$-complete.

Language Extensions: Overview

Overview

Language extensions

- The expressiveness of a language can be enhanced by introducing new constructs.
- To this end, we must address the following issues:
  - What is the **syntax** of the new language construct?
  - What is the **semantics** of the new language construct?
  - How to **implement** the new language construct?
- A way of providing semantics is to furnish a translation removing the new constructs, eg. classical negation.
- This translation might also be used for implementing the language extension. When is this feasible?

Overview

Integrity Constraints

- Purpose Integrity constraints eliminate unwanted solution candidates
- Syntax An integrity constraint is of the form

$$\leftarrow A_1, \dots, A_m, \mathit{not}\ A_{m+1}, \dots, \mathit{not}\ A_n,$$

  where $n \geq m \geq 1$, and each A_i $(1 \leq i \leq n)$ is a atom.
- Example `:- edge(X,Y), color(X,C), color(Y,C).`
- Implementation For a new symbol x, map

$$\begin{array}{rrcl} & & \leftarrow & A_1, \dots, A_m, \mathit{not}\ A_{m+1}, \dots, \mathit{not}\ A_n \\ \mapsto & x & \leftarrow & A_1, \dots, A_m, \mathit{not}\ A_{m+1}, \dots, \mathit{not}\ A_n, \mathit{not}\ x \end{array}$$

- Another example $\Pi = \{p \leftarrow \mathit{not}\ q,\ q \leftarrow \mathit{not}\ p\}$
  versus $\Pi' = \Pi \cup \{\leftarrow p\}$ and $\Pi'' = \Pi \cup \{\leftarrow \mathit{not}\ p\}$

Overview

Choice rules

- Idea Choices over subsets.
- Syntax

$$\{A_1,\dots,A_m\} \leftarrow A_{m+1},\dots,A_n, \mathit{not}\ A_{n+1},\dots,\mathit{not}\ A_o,$$

- Informal meaning If the body is satisfied in an answer set, then any subset of $\{A_1,\dots,A_m\}$ can be included in the answer set.
- Example `1 {color(X,C) : col(C)} 1 :- node(X).`
- Another Example The program $\Pi = \{\ \{a\} \leftarrow b,\ b \leftarrow\}$ has two answer sets: $\{b\}$ and $\{a,b\}$.
- Implementation lparse/gringo + smodels/cmodels/clasp

Embedding in normal logic programs

- A choice rule of form

$$\{A_1, \dots, A_m\} \leftarrow A_{m+1}, \dots, A_n, not\ A_{n+1}, \dots, not\ A_o$$

can be translated into $2m + 1$ rules

$$\begin{array}{rclcrcl} A & \leftarrow & A_{m+1}, \dots, A_n, not\ A_{n+1}, \dots, not\ A_o & & & & \\ A_1 & \leftarrow & A, not\ \overline{A_1} & \dots & A_m & \leftarrow & A, not\ \overline{A_m} \\ \overline{A_1} & \leftarrow & not\ A_1 & \dots & \overline{A_m} & \leftarrow & not\ A_m \end{array}$$

by introducing new atoms $A, \overline{A_1}, \dots, \overline{A_m}$.

Overview

Cardinality constraints

- Syntax A (positive) cardinality constraint is of the form
  $l\ \{A_1, \ldots, A_m\}\ u$
- **Informal** meaning A cardinality constraint is satisfied in an answer set X, if the number of atoms from $\{A_1, \ldots, A_m\}$ satisfied in X is between l and u (inclusive).
  More formally, if $l \leq |\{A_1, \ldots, A_m\} \cap X| \leq u$.
- Conditions $\quad l\ \{A_1 : B_1, \ldots, A_m : B_m\}\ u$
  where $B_1, \ldots, B_m$ are used for restricting instantiations of variables occurring in $A_1, \ldots, A_m$.
- Example 2 {hd(a),...,hd(m)} 4
- Implementation lparse/gringo + smodels/cmodels/clasp

Overview

Cardinality rules

- Idea Control cardinality of subsets.
- Syntax

$$A_0 \leftarrow l \ \{A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n\}$$

- Informal meaning If at least l elements of the "body" are true in an answer set, then add A_0 to the answer set.
  - ➥ l is a **lower bound** on the "body"
- Example The program $\Pi = \{\ a \leftarrow 1\{b, c\},\ \ b \leftarrow\}$ has one answer set: $\{a, b\}$.
- Implementation lparse/gringo + smodels/cmodels/clasp
  ☞ gringo distinguishes sets and multi-sets!

Embedding in normal logic programs (ctd)

- Replace each cardinality rule

$$A_0 \leftarrow l \; \{A_1, \dots, A_m\} \qquad \text{by} \qquad A_0 \leftarrow cc(A_1, l)$$

where atom $cc(A_i, j)$ represents the fact that at least j of the atoms in $\{A_i, \dots, A_m\}$, that is, of the atoms that have an equal or greater index than i, are in a particular answer set.

- The definition of $cc(A_i, j)$ is given by the rules

$$\begin{array}{rcl} cc(A_i, j{+}1) & \leftarrow & cc(A_{i+1}, j), A_i \\ cc(A_i, j) & \leftarrow & cc(A_{i+1}, j) \\ cc(A_{m+1}, 0) & \leftarrow & \end{array}$$

- What about space complexity?

. . . and vice versa

- A normal rule

$$A_0 \leftarrow A_1, \dots, A_m, not\ A_{m+1}, \dots, not\ A_n,$$

can be represented by the cardinality rule

$$A_0 \leftarrow n{+}m\ \{A_1, \dots, A_m, not\ A_{m+1}, \dots, not\ A_n\}.$$

Cardinality rules with upper bounds

- A rule of the form

$$A_0 \leftarrow l \ \{A_1, \dots, A_m, \mathit{not}\ A_{m+1}, \dots, \mathit{not}\ A_n\} \ u$$

stands for

$$\begin{array}{rcl} A_0 & \leftarrow & B, \mathit{not}\ C \\ B & \leftarrow & l \ \{A_1, \dots, A_m, \mathit{not}\ A_{m+1}, \dots, \mathit{not}\ A_n\} \\ C & \leftarrow & u{+}1 \ \{A_1, \dots, A_m, \mathit{not}\ A_{m+1}, \dots, \mathit{not}\ A_n\} \end{array}$$

Cardinality constraints as heads

- A rule of the form

$$l \ \{A_1, \ldots, A_m\} \ u \leftarrow A_{m+1}, \ldots, A_n, \mathit{not} \ A_{n+1}, \ldots, \mathit{not} \ A_o,$$

stands for

$$\begin{array}{rcl} B & \leftarrow & A_{m+1}, \ldots, A_n, \mathit{not} \ A_{n+1}, \ldots, \mathit{not} \ A_o \\ \{A_1, \ldots, A_m\} & \leftarrow & B \\ C & \leftarrow & l \ \{A_1, \ldots, A_m\} \ u \\ & \leftarrow & B, \mathit{not} \ C \end{array}$$

Full-fledged cardinality rules

- A rule of the form

$$l_0 \ S_0 \ u_0 \leftarrow l_1 \ S_1 \ u_1, \ldots, l_n \ S_n \ u_n$$

stands for $0 \leq i \leq n$

$$\begin{array}{rcl} B_i & \leftarrow & l_i \ S_i \\ C_i & \leftarrow & u_i{+}1 \ S_i \\ A & \leftarrow & B_1, \ldots, B_n, not \ C_1, \ldots, not \ C_n \\ & \leftarrow & A, not \ B_0 \\ & \leftarrow & A, C_0 \\ S_0 \cap \mathcal{A} & \leftarrow & A \end{array}$$

where $\mathcal{A}$ is the underlying alphabet.

Overview

Weight constraints

- Syntax $l \; [A_1 = w_1, \dots, A_m = w_m,$
  $\qquad not \; A_{m+1} = w_{m+1}, \dots, not \; A_n = w_n] \; u$
- **Informal** meaning A weight constraint is satisfied in an answer set X, if

$$l \leq \left(\sum_{1 \leq i \leq m, A_i \in X} w_i + \sum_{m < i \leq n, A_i \notin X} w_i \right) \leq u \; .$$

  ➥ Generalization of cardinality constraints.
- Example 80 [hd(a)=50,...,hd(m)=100] 400
- Implementation lparse/gringo + smodels/cmodels/clasp
  ☞ gringo distinguishes sets and multi-sets!

Optimization statements

- Idea Compute optimal answer sets by minimizing or maximizing a weighted sum of given elements, respectively.
- Syntax
  - $\#minimize\ [A_1 = w_1, \dots, A_m = w_m,$
    $not\ A_{m+1} = w_{m+1}, \dots, not\ A_n = w_n]$
  - $\#maximize\ [A_1 = w_1, \dots, A_m = w_m,$
    $not\ A_{m+1} = w_{m+1}, \dots, not\ A_n = w_n]$
- Several optimization statements are interpreted lexicographically.
- Example
  - #minimize [hd(a)=30,. . . ,hd(m)=50]
  - #minimize [road(X,Y) : length(X,Y,L) = L]
- Implementation lparse/gringo + smodels/clasp

Weak integrity constraints

- Syntax $:\sim\ A_1, \dots, A_m, not\ A_{m+1}, \dots, not\ A_n\ [w : l]$
- Informal meaning
  1. minimize the sum of weights of violated constraints in the highest level;
  2. minimize the sum of weights of violated constraints in the next lower level;
  3. etc
- Implementation dlv

Overview

Conditional literals in `lparse` and `gringo`

- We often want to encode the contents of a (multi-)set rather than enumerating each of the elements.
- To support this, `lparse` and `gringo` allow for **conditional literals**.

- Syntax
  $A_0 : A_1 : \ldots : A_m : not\ A_{m+1} : \ldots : not\ A_n$
- Informal meaning
  List all ground instances of A_0 such that corresponding instances of $A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n$ are true.
- Example `gringo` instantiates the program:

```
p(1). p(2). p(3).  q(2).        {r(X) : p(X) : not q(X)}.
```

  to:

```
p(1). p(2). p(3).  q(2).        {r(1), r(3)}.
```

Domain predicates in `lparse` and `gringo`

- The predicates of literals on the right-hand side of a colon (:) must be defined from facts without any negative recursion.
- Such **domain predicates** are fully evaluated by `lparse` and `gringo`.
- Example

```
p(1). p(2).
q(X) :- p(X), not p(X+1).
q(X) :- p(X), q(X+1).
r(X) :- p(X), not r(X+1).
```

  - `p/1` and `q/1` are domain predicates because none of them negatively depends on itself.
  - `r/1` is not a domain predicate because it is defined in terms of `not r(X+1)`.
- See `gringo` documentation for further details.

Normal form in `lparse` and `gringo`

- Consider a logic program consisting of
  - normal rules
  - choice rules
  - cardinality rules
  - weight rules
  - optimization statements
- Such a format is obtained by `lparse` or `gringo`

  and directly implemented by `smodels` and `clasp`.

Aggregates: Overview

Overview

Motivation

- Aggregates provide a general way to obtain a single value from a collection of input values given as a set, a bag, or a list.
- Popular aggregate (functions):
  - Average
  - Count
  - Maximum
  - Minimum
  - Sum
- Cardinality and Weight constraints rely on Count and Sum aggregates.

Overview

Syntax

- An **aggregate** has the form:

  $$F \ \langle A_1 = w_1, \dots, A_m = w_m, \mathit{not}\ A_{m+1} = w_{m+1}, \dots, \mathit{not}\ A_n = w_n \rangle \prec k$$

  where
  - F stands for a function mapping multi-sets of $\mathbb{Z}$ to $\mathbb{Z} \cup \{+\infty, -\infty\}$,
  - $\prec$ stands for a relation between $\mathbb{Z} \cup \{+\infty, -\infty\}$ and $\mathbb{Z}$,
  - k an integer,
  - A_i is an atom, and
  - w_i are integers

  for $1 \leq i \leq n$.
- For instance, $\mathit{sum}\ \langle hd(a) = 30, \dots, hd(m) = 50 \rangle \leq 300$

Overview

Semantics

- A (positive) aggregate $F\ \langle A_1 = w_1, \dots, A_n = w_n \rangle \prec k$ can be represented by the formula:

$$\bigwedge_{I \subseteq \{1,\dots,n\}, F\langle w_i \mid i \in I\rangle \not\prec k} \left(\bigwedge_{i \in I} A_i \rightarrow \bigvee_{i \in \overline{I}} A_i \right)$$

  where $\overline{I} = \{1, \dots, n\} \setminus I$ and $\not\prec$ is the complement of $\prec$.

- Then, $F\ \langle A_1 = w_1, \dots, A_n = w_n \rangle \prec k$ is true in X iff the above formula is true in X.

An example

- Consider $sum\langle p = 1, q = 1\rangle \neq 1$
  ➥ that is, $A_1 = p$, $A_2 = q$ and $w_1 = 1$, $w_2 = 1$
- Calculemus!

I	$\langle w_i \mid i \in I\rangle$	$\sum\langle w_i \mid i \in I\rangle$	$\sum\langle w_i \mid i \in I\rangle = 1$
$\emptyset$	$\langle\rangle$	0	*false*
$\{1\}$	$\langle 1\rangle$	1	*true*
$\{2\}$	$\langle 1\rangle$	1	*true*
$\{1,2\}$	$\langle 1,1\rangle$	2	*false*

- We get $(p \rightarrow q) \wedge (q \rightarrow p)$
- Analogously, we obtain $(p \vee q) \wedge \neg(p \wedge q)$ for $sum\langle p = 1, q = 1\rangle = 1$.

Monotonicity

- Monotone aggregates
  - For instance,
    - $body^+(r)$
    - $sum\langle p = 1, q = 1\rangle > 1$ amounts to $p \wedge q$
  - We get a simpler characterization: $\bigwedge_{I\subseteq\{1,\dots,n\},F\langle w_i \mid i\in I\rangle \not\prec k} \bigvee_{i\in\bar{I}} A_i$
- Anti-monotone aggregates
  - For instance,
    - $body^-(r)$
    - $sum\langle p = 1, q = 1\rangle < 1$ amounts to $\neg p \wedge \neg q$
  - We get a simpler characterization: $\bigwedge_{I\subseteq\{1,\dots,n\},F\langle w_i \mid i\in I\rangle \not\prec k} \neg \bigwedge_{i\in I} A_i$
- Non-monotone aggregates
  - For instance, $sum\langle p = 1, q = 1\rangle \neq 1$ is non-monotone.

The smodels approach: Overview

Overview

(Towards) the `smodels` approach

- Wanted:
  - An efficient procedure to compute answer sets
- The `smodels` approach:
  - Backtracking search building a binary search tree
  - A node in the search tree corresponds to a 3-valued interpretation
  - The search space is pruned by
    - deriving deterministic consequences and detecting conflicts (**expand**)
    - making one choice at a time by appeal to a heuristic (**select**)
  - ☞ Heuristic choices are made on atoms

Overview

Approximating answer sets

First Idea Approximate an answer set X by two sets of atoms L and U such that $L \subseteq X \subseteq U$.

- ➥ L and U constitute lower and upper bounds on X.
- ➥ L and $(\mathcal{A} \setminus U)$ describe a 3-valued model of the program.

Observation

$$X \subseteq Y \text{ implies } \Pi^Y \subseteq \Pi^X \text{ implies } Cn(\Pi^Y) \subseteq Cn(\Pi^X)$$

Properties Let X be an answer set of normal logic program Π.

- If $L \subseteq X$, then $X \subseteq Cn(\Pi^L)$.
- If $X \subseteq U$, then $Cn(\Pi^U) \subseteq X$.
- If $L \subseteq X \subseteq U$, then $L \cup Cn(\Pi^U) \subseteq X \subseteq U \cap Cn(\Pi^L)$.

Approximating answer sets (ctd)

Second Idea

Iterate

- Replace L by $L \cup Cn(\Pi^U)$
- Replace U by $U \cap Cn(\Pi^L)$

until L and U do not change anymore.

Observations

- At each iteration step
  - L becomes larger (or equal)
  - U becomes smaller (or equal)
- $L \subseteq X \subseteq U$ is invariant for every answer set X of Π
- If $L \not\subseteq U$, then Π has no answer set!
- If $L = U$, then L is an answer set of Π.

The simplistic expand algorithm

expand(L, U)

repeat

$L' \leftarrow L$
$U' \leftarrow U$
$L \leftarrow L' \cup Cn(\Pi^{U'})$
$U \leftarrow U' \cap Cn(\Pi^{L'})$
if $L \not\subseteq U$ **then return**

until $L = L'$ and $U = U'$

☞ Π is a global parameter!

Let's expand!

$$\Pi = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, not\ c \\ d \leftarrow b, not\ e \\ e \leftarrow \ not\ d \end{array} \right\}$$

	L'	$Cn(\Pi^{U'})$	L	U'	$Cn(\Pi^{L'})$	U
1	$\emptyset$	$\{a\}$	$\{a\}$	$\{a,b,c,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
2	$\{a\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$
3	$\{a,b\}$	$\{a,b\}$	$\{a,b\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$	$\{a,b,d,e\}$

➥ We have $\{a,b\} \subseteq X$ and $(\mathcal{A} \setminus \{a,b,d,e\}) \cap X = (\{c\} \cap X) = \emptyset$ for every answer set X of Π.

The simplistic expand algorithm (ctd)

expand

- tightens the approximation on answer sets
- is answer set preserving

Let's expand with d !

$$\Pi = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, not\ c \\ d \leftarrow b, not\ e \\ e \leftarrow\ not\ d \end{array} \right\}$$

	L'	$Cn(\Pi^{U'})$	L	U'	$Cn(\Pi^{L'})$	U
1	$\{d\}$	$\{a\}$	$\{a, d\}$	$\{a, b, c, d, e\}$	$\{a, b, d\}$	$\{a, b, d\}$
2	$\{a, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$
3	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$	$\{a, b, d\}$

➥ $\{a, b, d\}$ is an answer set X of Π.

Let's expand with "*not d*" !

$$\Pi = \left\{ \begin{array}{l} a \leftarrow \\ b \leftarrow a, not\ c \\ d \leftarrow b, not\ e \\ e \leftarrow\ not\ d \end{array} \right\}$$

	L'	$Cn(\Pi^{U'})$	L	U'	$Cn(\Pi^{L'})$	U
1	$\emptyset$	$\{a, e\}$	$\{a, e\}$	$\{a, b, c, e\}$	$\{a, b, d, e\}$	$\{a, b, e\}$
2	$\{a, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$
3	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$	$\{a, b, e\}$

➥ $\{a, b, e\}$ is an answer set X of Π.

Overview

Interlude: Partial interpretations

or: 3-valued interpretations

A **partial interpretation** of a logic program Π maps atoms on truth values: $\{\mathit{true}, \mathit{false}, \mathit{unknown}\}$.

Representation $\langle T, F\rangle$, where

- T is the set of all *true* atoms and
- F is the set of all *false* atoms.
- Truth of atoms in $\mathit{atom}(\Pi) \setminus (T \cup F)$ is *unknown*.

☞ By $\mathit{atom}(\Pi)$, we denote the set of atoms occuring in Π.

Properties
- $\langle T, F\rangle$ is **conflicting** iff $T \cap F \neq \emptyset$.
- $\langle T, F\rangle$ is **total** iff $T \cup F = \mathit{atom}(\Pi)$ and $T \cap F = \emptyset$.

Definition For $\langle T_1, F_1\rangle$ and $\langle T_2, F_2\rangle$, define:
- $\langle T_1, F_1\rangle \sqsubseteq \langle T_2, F_2\rangle$ iff $T_1 \subseteq T_2$ and $F_1 \subseteq F_2$
- $\langle T_1, F_1\rangle \sqcup \langle T_2, F_2\rangle = \langle T_1 \cup T_2, F_1 \cup F_2\rangle$

Overview

The **smodels** (decision) algorithm

Global: Normal logic program Π

smodels($\langle T, F\rangle$**)**

$\langle T, F\rangle \leftarrow$ **expand(**$\langle T, F\rangle$**)**
if $\langle T, F\rangle$ is **conflicting then return**
else if $\langle T, F\rangle$ is **total then exit with** T
else

$A \leftarrow$ **select(**$atom(\Pi) \setminus (T \cup F)$**)**
smodels($\langle T \cup \{A\}, F\rangle$**)**
smodels($\langle T, F \cup \{A\}\rangle$**)**

Call: **smodels(**$\langle \emptyset, \emptyset\rangle$**)**

Deterministic consequences via **expand**

Global: Normal logic program Π

expand($\langle T, F\rangle$)

repeat
  $\langle T, F\rangle \leftarrow$ **atleast**($\langle T, F\rangle$)
  if $\langle T, F\rangle$ is **conflicting then return** $\langle T, F\rangle$
  else
    $F' \leftarrow F$
    $F \leftarrow F \cup$ **atmost**($\langle T, F\rangle$)
until $F = F'$
return $\langle T, F\rangle$

- ☞ **atleast**($\langle T, F\rangle$) derives deterministic consequences from **Clark's completion**
- ☞ **atmost**($\langle T, F\rangle$) derives deterministic consequences from **unfounded sets**

A glimpse at **atleast**($\langle T, F \rangle$)

repeat

if $\langle T, F \rangle$ is **conflicting then return** $\langle T, F \rangle$

$\langle T', F' \rangle \leftarrow \langle T, F \rangle$

case of

$r \in \Pi$ **such that** $head(r) \notin T$ **and**
$body^+(r) \subseteq T, body^-(r) \subseteq F$**:**

$$T \leftarrow T \cup \{head(r)\}$$

$A \in (atom(\Pi) \setminus F)$ **such that for all** $r \in \Pi$:
$head(r) \neq A$ **or** $(body^+(r) \cap F) \cup (body^-(r) \cap T) \neq \emptyset$**:**

$$F \leftarrow F \cup \{A\}$$

$head(r) \in F, r \in \Pi$ **such that** $body^+(r) \cap body^-(r) = \emptyset$ **and**
$(body^+(r) \setminus T) \cup (body^-(r) \setminus F) = \{A\}$**:**

if $A \in body^+(r)$ **then** $F \leftarrow F \cup \{A\}$ **else** $T \leftarrow T \cup \{A\}$

$(A = head(r)) \in T, r \in \Pi$ **such that** $body^+(r) \not\subseteq T$ **or** $body^-(r) \not\subseteq F$ **and**
for all $r' \in \Pi \setminus \{r\}$: $head(r') \neq A$ **or** $(body^+(r') \cap F) \cup (body^-(r') \cap T) \neq \emptyset$**:**

$$T \leftarrow T \cup body^+(r)$$
$$F \leftarrow F \cup body^-(r)$$

until $\langle T, F \rangle = \langle T', F' \rangle$

return $\langle T, F \rangle$

A glimpse at **atmost**$(\langle T, F\rangle)$

return $\mathbf{U}_\Pi\langle T, F\rangle$

Completion: Overview

Overview

Completion

Let Π be a normal logic program.
The **completion** of Π is defined as follows:

$$\begin{aligned} Comp(body(r)) &= \bigwedge_{A \in body^+(r)} A \wedge \bigwedge_{A \in body^-(r)} \neg A \\ Comp(\Pi) &= \{A \leftrightarrow \bigvee_{r \in \Pi, head(r)=A} Comp(body(r)) \mid A \in atom(\Pi)\} \end{aligned}$$

- Every answer set of Π is a model of $Comp(\Pi)$, but not vice versa.
- Models of $Comp(\Pi)$ are called the **supported models** of Π.

- In other words, every answer set of Π is a supported model of Π.
- By definition, every supported model of Π is also a model of Π.

A first example

$$\Pi = \left\{ \begin{array}{lcl} a & \leftarrow & \\ b & \leftarrow & a \\ c & \leftarrow & b \\ c & \leftarrow & d \\ d & \leftarrow & c, e \end{array} \right\} \qquad Comp(\Pi) = \left\{ \begin{array}{lcl} a & \leftrightarrow & \top \\ b & \leftrightarrow & a \\ c & \leftrightarrow & (b \vee d) \\ d & \leftrightarrow & (c \wedge e) \\ e & \leftrightarrow & \bot \end{array} \right\}$$

- The supported model of Π is $\{a, b, c\}$.
- The answer set of Π is $\{a, b, c\}$.

A second example

$$\Pi = \left\{ \begin{array}{lcl} q & \leftarrow & not\ p \\ p & \leftarrow & not\ q, not\ x \end{array} \right\} \quad Comp(\Pi) = \left\{ \begin{array}{lcl} q & \leftrightarrow & \neg p \\ p & \leftrightarrow & (\neg q \wedge \neg x) \\ x & \leftrightarrow & \bot \end{array} \right\}$$

- The supported models of Π are $\{p\}$ and $\{q\}$.
- The answer sets of Π are $\{p\}$ and $\{q\}$.

A third example

$$\Pi = \{ \; p \leftarrow p \; \} \qquad Comp(\Pi) = \{ \; p \leftrightarrow p \; \}$$

- The supported models of Π are $\emptyset$ and $\{p\}$.
- The answer set of Π is $\emptyset$!

Overview

Fitting operator: Basic idea

Idea Extend T_Π to normal logic programs.

Logical background Completion

- The head atom of a rule must be *true*
  if the rule's body is *true*.
- An atom must be *false*
  if the body of each rule having it as head is *false*.

Fitting operator: Definition

Let Π be a normal logic program.

Define

$$\mathbf{\Phi}_\Pi\langle T,F\rangle = \langle \mathbf{T}_\Pi\langle T,F\rangle, \mathbf{F}_\Pi\langle T,F\rangle\rangle$$

where

$$\begin{aligned}
\mathbf{T}_\Pi\langle T,F\rangle &= \{\mathit{head}(r) \mid r \in \Pi, \mathit{body}^+(r) \subseteq T, \mathit{body}^-(r) \subseteq F\} \\
\mathbf{F}_\Pi\langle T,F\rangle &= \{A \in \mathit{atom}(\Pi) \mid \mathit{body}^+(r) \cap F \neq \emptyset \text{ or } \mathit{body}^-(r) \cap T \neq \emptyset \\
&\qquad\qquad \text{for each } r \in \Pi \text{ such that } \mathit{head}(r) = A\}
\end{aligned}$$

Fitting operator: Example

$$\Pi_1 = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \mathit{not}\ d & e \leftarrow b \\ b \leftarrow \mathit{not}\ a & d \leftarrow \mathit{not}\ c, \mathit{not}\ e & e \leftarrow e \end{array} \right\}$$

Let's iterate $\mathbf{\Phi}_{\Pi_1}$ on $\langle\{a\},\{d\}\rangle$:

$$\begin{array}{rcl} \mathbf{\Phi}_{\Pi_1}\langle\{a\},\{d\}\rangle & = & \langle\{a,c\},\{b\}\rangle \\ \mathbf{\Phi}_{\Pi_1}\langle\{a,c\},\{b\}\rangle & = & \langle\{a\},\{b,d\}\rangle \\ \mathbf{\Phi}_{\Pi_1}\langle\{a\},\{b,d\}\rangle & = & \langle\{a,c\},\{b\}\rangle \\ & \vdots & \end{array}$$

Fitting semantics

Define the iterative variant of $\mathbf{\Phi}_\Pi$ analogously to T_Π:

$$\mathbf{\Phi}_\Pi^0\langle T,F\rangle = \langle T,F\rangle \qquad \mathbf{\Phi}_\Pi^{i+1}\langle T,F\rangle = \mathbf{\Phi}_\Pi\mathbf{\Phi}_\Pi^i\langle T,F\rangle$$

Define the **Fitting semantics** of a normal logic program Π as the partial interpretation:

$$\bigsqcup_{i\geq 0}\mathbf{\Phi}_\Pi^i\langle\emptyset,\emptyset\rangle$$

Fitting semantics: Example

$$\Pi_1 = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, not\ d & e \leftarrow b \\ b \leftarrow not\ a & d \leftarrow not\ c, not\ e & e \leftarrow e \end{array} \right\}$$

$$\begin{array}{rcccl} \mathbf{\Phi}^0_{\Pi_1}\langle\emptyset,\emptyset\rangle & = & & & \langle\emptyset,\emptyset\rangle \\ \mathbf{\Phi}^1_{\Pi_1}\langle\emptyset,\emptyset\rangle & = & \mathbf{\Phi}_{\Pi_1}\langle\emptyset,\emptyset\rangle & = & \langle\{a\},\emptyset\rangle \\ \mathbf{\Phi}^2_{\Pi_1}\langle\emptyset,\emptyset\rangle & = & \mathbf{\Phi}_{\Pi_1}\langle\{a\},\emptyset\rangle & = & \langle\{a\},\{b\}\rangle \\ \mathbf{\Phi}^3_{\Pi_1}\langle\emptyset,\emptyset\rangle & = & \mathbf{\Phi}_{\Pi_1}\langle\{a\},\{b\}\rangle & = & \langle\{a\},\{b\}\rangle \\ \bigsqcup_{i\geq 0}\mathbf{\Phi}^i_{\Pi_1}\langle\emptyset,\emptyset\rangle & = & \langle\{a\},\{b\}\rangle & & \end{array}$$

Fitting semantics: Properties

Let Π be a normal logic program.

- $\mathbf{\Phi}_\Pi\langle\emptyset,\emptyset\rangle$ is monotonic.
  That is, $\mathbf{\Phi}^i_\Pi\langle\emptyset,\emptyset\rangle \sqsubseteq \mathbf{\Phi}^{i+1}_\Pi\langle\emptyset,\emptyset\rangle$.
- The Fitting semantics of Π is
  - not conflicting,
  - and generally not total.

Fitting fixpoints

Let Π be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation.

Define $\langle T, F\rangle$ as a **Fitting fixpoint** of Π if $\mathbf{\Phi}_\Pi\langle T, F\rangle = \langle T, F\rangle$.

- The Fitting semantics is the $\sqsubseteq$-least Fitting fixpoint of Π.
- Any other Fitting fixpoint extends the Fitting semantics.
- Total Fitting fixpoints correspond to supported models.

Fitting fixpoints: Example

$$\Pi_1 = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \mathit{not}\ d & e \leftarrow b \\ b \leftarrow \mathit{not}\ a & d \leftarrow \mathit{not}\ c, \mathit{not}\ e & e \leftarrow e \end{array} \right\}$$

Π_1 has three total Fitting fixpoints:

1. $\langle \{a, c\}, \{b, d, e\} \rangle$
2. $\langle \{a, d\}, \{b, c, e\} \rangle$
3. $\langle \{a, c, e\}, \{b, d\} \rangle$

Π_1 has three supported models, two of them are answer sets.

Properties of Fitting operator

Let Π be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation.

- Let $\mathbf{\Phi}_\Pi \langle T, F \rangle = \langle T', F' \rangle$.
  If X is an answer set of Π such that $T \subseteq X$ and $X \cap F = \emptyset$,
  then $T' \subseteq X$ and $X \cap F' = \emptyset$.
- That is, $\mathbf{\Phi}_\Pi$ is **answer set preserving**.
  - ➥ $\mathbf{\Phi}_\Pi$ can be used for approximating answer sets and so for propagation in ASP-solvers.

However, $\mathbf{\Phi}_\Pi$ is still insufficient, because total fixpoints correspond to supported models, not necessarily answer sets.

☞ The problem is the same as with program completion.

The missing piece is non-circularity of derivations !

Example

$$\Pi = \left\{ \begin{array}{ccc} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\} \qquad \begin{array}{rcl} \Phi^0_\Pi\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \\ \Phi^1_\Pi\langle\emptyset,\emptyset\rangle & = & \langle\emptyset,\emptyset\rangle \end{array}$$

That is, Fitting semantics cannot assign *false* to a and b, although they can never become *true* !

Overview

Rebuilding **atleast**($\langle T, F\rangle$)

repeat from Fitting operator

if $\langle T, F\rangle$ is **conflicting then return** $\langle T, F\rangle$
$\langle T', F'\rangle \leftarrow \langle T, F\rangle$
case of
$r \in \Pi$ **such that** $head(r) \notin T$ **and**
$body^+(r) \subseteq T, body^-(r) \subseteq F$**:**
$T \leftarrow T \cup \{head(r)\}$
$A \in (atom(\Pi) \setminus F)$ **such that for all** $r \in \Pi$:
$head(r) \neq A$ **or** $(body^+(r) \cap F) \cup (body^-(r) \cap T) \neq \emptyset$**:**
$F \leftarrow F \cup \{A\}$

until $\langle T, F\rangle = \langle T', F'\rangle$
return $\langle T, F\rangle$

Relationship with Fitting semantics

Let Π be a normal logic program.

- $\mathbf{atleast(}\langle\emptyset,\emptyset\rangle\mathbf{)} = \bigsqcup_{i\geq 0}\mathbf{\Phi}^i_\Pi\langle\emptyset,\emptyset\rangle$

What about supported models?
Consider:

$$\Pi = \left\{ \begin{array}{lll} a \leftarrow b & b \leftarrow \mathit{not}\ c & c \leftarrow \mathit{not}\ b \\ d \leftarrow e & e \leftarrow \mathit{not}\ f & f \leftarrow \mathit{not}\ e \end{array} \right\}$$

- $\mathbf{atleast(}\langle\{a\},\{d\}\rangle\mathbf{)} = \langle\{a\},\{d\}\rangle$
- The only supported model X of Π such that $a \in X$ and $d \notin X$ is $\{a, b, f\}$!

We can enhance $\mathbf{atleast(}\langle T, F\rangle\mathbf{)}$ by backward propagation !

Rebuilding **atleast**($\langle T, F\rangle$) from supported models

repeat

if $\langle T, F\rangle$ is **conflicting then return** $\langle T, F\rangle$

$\langle T', F'\rangle \leftarrow \langle T, F\rangle$

case of

$r \in \Pi$ **such that** $head(r) \notin T$ **and**
$body^+(r) \subseteq T, body^-(r) \subseteq F$**:**

$T \leftarrow T \cup \{head(r)\}$

$A \in (atom(\Pi) \setminus F)$ **such that for all** $r \in \Pi$:
$head(r) \neq A$ **or** $(body^+(r) \cap F) \cup (body^-(r) \cap T) \neq \emptyset$**:**

$F \leftarrow F \cup \{A\}$

$head(r) \in F, r \in \Pi$ **such that** $body^+(r) \cap body^-(r) = \emptyset$ **and**
$(body^+(r) \setminus T) \cup (body^-(r) \setminus F) = \{A\}$**:**

if $A \in body^+(r)$ **then** $F \leftarrow F \cup \{A\}$ **else** $T \leftarrow T \cup \{A\}$

$(A = head(r)) \in T, r \in \Pi$ **such that** $body^+(r) \not\subseteq T$ **or** $body^-(r) \not\subseteq F$ **and**
for all $r' \in \Pi \setminus \{r\}$: $head(r') \neq A$ **or** $(body^+(r') \cap F) \cup (body^-(r') \cap T) \neq \emptyset$**:**

$T \leftarrow T \cup body^+(r)$
$F \leftarrow F \cup body^-(r)$

until $\langle T, F\rangle = \langle T', F'\rangle$

return $\langle T, F\rangle$

Relationship with supported models

Let Π be a normal logic program and $\langle T, F\rangle$ a total interpretation.

- **atleast(**$\langle T, F\rangle$**)** $= \langle T, F\rangle$ iff T is a supported model of Π

Assuming **atmost(**$\langle T, F\rangle$**)** $= \emptyset$ for all $\langle T, F\rangle$,
we can apply **smodels** to compute supported models !

Reconsider:

$$\Pi_1 = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \mathit{not}\ d & e \leftarrow b \\ b \leftarrow \mathit{not}\ a & d \leftarrow \mathit{not}\ c, \mathit{not}\ e & e \leftarrow e \end{array} \right\}$$

Call	Interpretation	Result
smodels	$\langle\emptyset,\emptyset\rangle$	
expand	$\langle\emptyset,\emptyset\rangle$	$\langle\{a\},\{b\}\rangle$
select	$\langle\{a\},\{b\}\rangle$	$\langle\{a,e\},\{b\}\rangle$
expand	$\langle\{a,e\},\{b\}\rangle$	$\langle\{a,c,e\},\{b,d\}\rangle$
smodels	$\langle\emptyset,\emptyset\rangle$	$\{a,c,e\}$

Overview

(Non-)cyclic derivations

- Cyclic derivations are causing the mismatch between supported models and answer sets.
- Atoms in an answer set can be "derived" from a program in a finite number of steps.
- Atoms in a cycle (not being "supported from outside the cycle") cannot be "derived" from a program in a finite number of steps.
  - ☞ But they do not contradict the completion of a program.

Non-cyclic derivations

Let X be an answer set of normal logic program Π.

- For every atom $A \in X$, there is a finite sequence of positive rules

  $$\langle r_1, \ldots, r_n \rangle$$

  such that

  1. $head(r_1) = A$,
  2. $body^+(r_i) \subseteq \{head(r_j) \mid i < j \leq n\}$ for $1 \leq i \leq n$,
  3. $r_i \in \Pi^X$ for $1 \leq i \leq n$.

- That is, each atom of X has a non-cyclic derivation from Π^X.
- Is a derivable from program $\{a \leftarrow b,\ b \leftarrow a\}$?

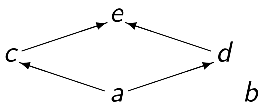
Positive atom dependency graph

Let Π be a normal logic program.
The **positive atom dependency graph** of Π is a directed graph $G(\Pi) = (V, E)$ such that

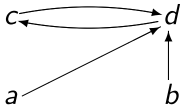
1. $V = atom(\Pi)$ and
2. $E = \{(p, q) \mid r \in \Pi, p \in body^+(r), head(r) = q\}$.

Examples

$$\Pi_2 = \left\{ \begin{array}{ll} a \leftarrow not\ b & b \leftarrow not\ a \\ c \leftarrow a, not\ d & d \leftarrow a, not\ c \\ e \leftarrow c, not\ a & e \leftarrow d, not\ b \end{array} \right\}$$



$$\Pi_3 = \left\{ \begin{array}{ll} a \leftarrow not\ b & b \leftarrow not\ a \\ c \leftarrow not\ a & c \leftarrow d \\ d \leftarrow a, b & d \leftarrow c \end{array} \right\}$$



Tight programs

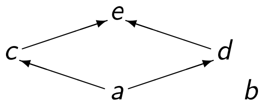
- A normal logic program Π is **tight** iff $G(\Pi)$ is acyclic.
- For example, Π_2 is tight, whereas Π_3 is not.
- If a normal logic program Π is tight, then

  X is an answer set of Π iff X is a model of $Comp(\Pi)$.

  That is, for tight programs, answer sets and supported models coincide.
- Also, for tight programs, $\mathbf{\Phi}_\Pi$ is sufficient for propagation.

(Non-)tight programs: Examples

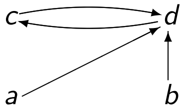
$$\Pi_2 = \left\{ \begin{array}{ll} a \leftarrow not\ b & b \leftarrow not\ a \\ c \leftarrow a, not\ d & d \leftarrow a, not\ c \\ e \leftarrow c, not\ a & e \leftarrow d, not\ b \end{array} \right\}$$



Answer sets: $\{\{a, c\}, \{a, d, e\}, \{b\}\}$

Supported models: $\{\{a, c\}, \{a, d, e\}, \{b\}\}$

$$\Pi_3 = \left\{ \begin{array}{ll} a \leftarrow not\ b & b \leftarrow not\ a \\ c \leftarrow not\ a & c \leftarrow d \\ d \leftarrow a, b & d \leftarrow c \end{array} \right\}$$



Answer sets: $\{\{a\}, \{b, c, d\}\}$

Supported models: $\{\{a\}, \{b, c, d\}, \{\mathbf{a}, \mathbf{c}, \mathbf{d}\}\}$

Unfounded Sets: Overview

Overview

Unfounded sets

Let Π be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation.

A set $U \subseteq atom(\Pi)$ is an **unfounded set** of Π with respect to $\langle T, F \rangle$ if, for each rule $r \in \Pi$, we have

1. $head(r) \notin U$,
2. $body^+(r) \cap F \neq \emptyset$ or $body^-(r) \cap T \neq \emptyset$, or
3. $body^+(r) \cap U \neq \emptyset$.

- Intuitively, $\langle T, F \rangle$ is what we already know about Π.
- Rules satisfying Condition 1 or 2 are not usable for further derivations.
- Condition 3 is the unfounded set condition treating cyclic derivations: **All rules still being usable to derive an atom in U require an(other) atom in U to be true.**

Example

$$\Pi = \left\{ \begin{array}{ccc} a & \leftarrow & b \\ b & \leftarrow & a \end{array} \right\}$$

- $\emptyset$ is an unfounded set (by definition).
- $\{a\}$ is not an unfounded set of Π wrt $\langle\emptyset, \emptyset\rangle$.
- $\{a\}$ is an unfounded set of Π wrt $\langle\emptyset, \{b\}\rangle$.
- $\{a\}$ is not an unfounded set of Π wrt $\langle\{b\}, \emptyset\rangle$.

  ➥ Analogously for $\{b\}$.
- $\{a, b\}$ is an unfounded set of Π wrt $\langle\emptyset, \emptyset\rangle$.
- $\{a, b\}$ is an unfounded set of Π wrt any partial interpretation.

Greatest unfounded sets

Observation The union of two unfounded sets is an unfounded set.

Let Π be a normal logic program,
and let $\langle T, F \rangle$ be a partial interpretation.
The **greatest unfounded set** of Π with respect to $\langle T, F \rangle$, denoted by $\mathbf{U}_\Pi \langle T, F \rangle$, is the union of all unfounded sets of Π with respect to $\langle T, F \rangle$.

- Alternatively, we may define

$$\mathbf{U}_\Pi \langle T, F \rangle = \mathit{atom}(\Pi) \setminus \mathit{Cn}(\{r \in \Pi \mid \mathit{body}^+(r) \cap F = \emptyset\}^T).$$

- Observe that $\mathit{Cn}(\{r \in \Pi \mid \mathit{body}^+(r) \cap F = \emptyset\}^T)$ contains all non-circularly derivable atoms from Π wrt $\langle T, F \rangle$.

Overview

Well-founded operator

Let Π be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation.

Observation Condition 2 (in the definition of an unfounded set) corresponds to set $\mathbf{F}_\Pi\langle T, F\rangle$ of Fitting's $\mathbf{\Phi}_\Pi\langle T, F\rangle$.

Idea Extend (negative part of) Fitting's operator $\mathbf{\Phi}_\Pi$.
That is,

- keep definition of $\mathbf{T}_\Pi\langle T, F\rangle$ from $\mathbf{\Phi}_\Pi\langle T, F\rangle$ and
- replace $\mathbf{F}_\Pi\langle T, F\rangle$ from $\mathbf{\Phi}_\Pi\langle T, F\rangle$ by $\mathbf{U}_\Pi\langle T, F\rangle$.

In words, an atom must be *false*
if it belongs to the greatest unfounded set.

Definition $\mathbf{\Omega}_\Pi\langle T, F\rangle = \langle \mathbf{T}_\Pi\langle T, F\rangle, \mathbf{U}_\Pi\langle T, F\rangle\rangle$

Property $\mathbf{\Phi}_\Pi\langle T, F\rangle \sqsubseteq \mathbf{\Omega}_\Pi\langle T, F\rangle$

Well-founded operator: Example

$$\Pi_1 = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \mathit{not}\ d & e \leftarrow b \\ b \leftarrow \mathit{not}\ a & d \leftarrow \mathit{not}\ c, \mathit{not}\ e & e \leftarrow e \end{array} \right\}$$

Let's iterate $\mathbf{\Omega}_{\Pi_1}$ on $\langle\{c\},\emptyset\rangle$:

$$\begin{array}{rcl} \mathbf{\Omega}_{\Pi_1}\langle\{c\},\emptyset\rangle & = & \langle\{a\},\{d\}\rangle \\ \mathbf{\Omega}_{\Pi_1}\langle\{a\},\{d\}\rangle & = & \langle\{a,c\},\{b,e\}\rangle \\ \mathbf{\Omega}_{\Pi_1}\langle\{a,c\},\{b,e\}\rangle & = & \langle\{a\},\{b,d,e\}\rangle \\ \mathbf{\Omega}_{\Pi_1}\langle\{a\},\{b,d,e\}\rangle & = & \langle\{a,c\},\{b,e\}\rangle \\ & \vdots & \end{array}$$

Well-founded semantics

Define the iterative variant of $\mathbf{\Omega}_\Pi$ analogously to $\mathbf{\Phi}_\Pi$:

$$\mathbf{\Omega}^0_\Pi\langle T, F\rangle = \langle T, F\rangle \qquad \mathbf{\Omega}^{i+1}_\Pi\langle T, F\rangle = \mathbf{\Omega}_\Pi\mathbf{\Omega}^i_\Pi\langle T, F\rangle$$

Define the **well-founded semantics** of a normal logic program Π as the partial interpretation:

$$\bigsqcup_{i\geq 0}\mathbf{\Omega}^i_\Pi\langle\emptyset,\emptyset\rangle$$

Well-founded semantics: Example

$$\Pi_1 = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, not\ d & e \leftarrow b \\ b \leftarrow not\ a & d \leftarrow not\ c, not\ e & e \leftarrow e \end{array} \right\}$$

$$\begin{array}{rcccl} \mathbf{\Omega}^0_{\Pi_1}\langle\emptyset,\emptyset\rangle & = & & & \langle\emptyset,\emptyset\rangle \\ \mathbf{\Omega}^1_{\Pi_1}\langle\emptyset,\emptyset\rangle & = & \mathbf{\Omega}_{\Pi_1}\langle\emptyset,\emptyset\rangle & = & \langle\{a\},\emptyset\rangle \\ \mathbf{\Omega}^2_{\Pi_1}\langle\emptyset,\emptyset\rangle & = & \mathbf{\Omega}_{\Pi_1}\langle\{a\},\emptyset\rangle & = & \langle\{a\},\{b,e\}\rangle \\ \mathbf{\Omega}^3_{\Pi_1}\langle\emptyset,\emptyset\rangle & = & \mathbf{\Omega}_{\Pi_1}\langle\{a\},\{b,e\}\rangle & = & \langle\{a\},\{b,e\}\rangle \\ \bigsqcup_{i\geq 0}\mathbf{\Omega}^i_{\Pi_1}\langle\emptyset,\emptyset\rangle & = & \langle\{a\},\{b,e\}\rangle & & \end{array}$$

Well-founded semantics: Properties

Let Π be a normal logic program.

- $\mathbf{\Omega}_\Pi\langle\emptyset,\emptyset\rangle$ is monotonic.
  That is, $\mathbf{\Omega}^i_\Pi\langle\emptyset,\emptyset\rangle \sqsubseteq \mathbf{\Omega}^{i+1}_\Pi\langle\emptyset,\emptyset\rangle$.
- The well-founded semantics of Π is
  - not conflicting,
  - and generally not total.
- We have $\bigsqcup_{i\geq 0}\mathbf{\Phi}^i_\Pi\langle\emptyset,\emptyset\rangle \sqsubseteq \bigsqcup_{i\geq 0}\mathbf{\Omega}^i_\Pi\langle\emptyset,\emptyset\rangle$.

Well-founded fixpoints

Let Π be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation.
Define $\langle T, F\rangle$ as a **well-founded fixpoint** of Π if $\mathbf{\Omega}_{\Pi}\langle T, F\rangle = \langle T, F\rangle$.

- The well-founded semantics is the $\sqsubseteq$-least well-founded fixpoint of Π.
- Any other well-founded fixpoint extends the well-founded semantics.
- Total well-founded fixpoints correspond to answer sets.

Well-founded fixpoints: Example

$$\Pi_1 = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \mathit{not}\ d & e \leftarrow b \\ b \leftarrow \mathit{not}\ a & d \leftarrow \mathit{not}\ c, \mathit{not}\ e & e \leftarrow e \end{array} \right\}$$

Π_1 has two total well-founded fixpoints:

1. $\langle \{a, c\}, \{b, d, e\} \rangle$
2. $\langle \{a, d\}, \{b, c, e\} \rangle$

Both of them represent answer sets.

Properties of well-founded operator

Let Π be a normal logic program,
and let $\langle T, F\rangle$ be a partial interpretation.

- Let $\mathbf{\Omega}_\Pi\langle T, F\rangle = \langle T', F'\rangle$.
  If X is an answer set of Π such that $T \subseteq X$ and $X \cap F = \emptyset$,
  then $T' \subseteq X$ and $X \cap F' = \emptyset$.
- That is, $\mathbf{\Omega}_\Pi$ is **answer set preserving**.
  - ➥ $\mathbf{\Omega}_\Pi$ can be used for approximating answer sets and so for propagation in ASP-solvers.

Unlike $\mathbf{\Phi}_\Pi$, operator $\mathbf{\Omega}_\Pi$ is sufficient for propagation because total fixpoints correspond to answer sets.

☞ In addition to $\mathbf{\Omega}_\Pi$, most ASP-solvers apply backward propagation (cf. Page 210), originating from program completion (although this is unnecessary from a formal point of view).

Overview

Rebuilding $\mathbf{atmost}(\langle T, F\rangle)$

from (greatest) unfounded sets

return $\mathbf{U}_{\Pi}\langle T, F\rangle$

Recalling **expand**

Global: Normal logic program Π

expand($\langle T, F\rangle$)

repeat
  $\langle T, F\rangle \leftarrow$ **atleast**($\langle T, F\rangle$)
  if $\langle T, F\rangle$ is **conflicting then return** $\langle T, F\rangle$
  else
    $F' \leftarrow F$
    $F \leftarrow F \cup$ **atmost**($\langle T, F\rangle$)
until $F = F'$
return $\langle T, F\rangle$

- ☞ **atleast**($\langle T, F\rangle$) derives deterministic consequences from **Clark's completion**
- ☞ **atmost**($\langle T, F\rangle$) derives deterministic consequences from **unfounded sets**

Relationship with well-founded semantics

Let Π be a normal logic program.

- $\mathbf{expand}(\langle\emptyset,\emptyset\rangle) = \bigsqcup_{i\geq 0} \mathbf{\Omega}^i_\Pi\langle\emptyset,\emptyset\rangle$

☞ That is, **expand** is basically an implementation of well-founded semantics !

☞ Additional backward propagation in **atleast** prunes the search space further !

Relationship with answer sets

Let Π be a normal logic program and $\langle T, F\rangle$ a total interpretation.

- **expand**$(\langle T, F\rangle) = \langle T, F\rangle$ iff T is an answer set of Π

Given **atmost**$(\langle T, F\rangle) = \mathbf{U}_\Pi\langle T, F\rangle$,
we can apply **smodels** to compute answer sets !
Reconsider:

$$\Pi_1 = \left\{ \begin{array}{lll} a \leftarrow & c \leftarrow a, \mathit{not}\ d & e \leftarrow b \\ b \leftarrow \mathit{not}\ a & d \leftarrow \mathit{not}\ c, \mathit{not}\ e & e \leftarrow e \end{array} \right\}$$

Call	Interpretation	Result
smodels	$\langle \emptyset, \emptyset \rangle$	
expand	$\langle \emptyset, \emptyset \rangle$	$\langle \{a\}, \{b, e\} \rangle$
select	$\langle \{a\}, \{b, e\} \rangle$	$\langle \{a, c\}, \{b, e\} \rangle$
expand	$\langle \{a, c\}, \{b, e\} \rangle$	$\langle \{a, c\}, \{b, d, e\} \rangle$
smodels	$\langle \emptyset, \emptyset \rangle$	$\{a, c\}$

Additional remarks on `smodels`

The `smodels` implementation also features:

- Extended rules
  - Cardinality constraints
  - Weight constraints
- Optimiziation via *minimize* and *maximize*
- Efficient counter-based propagation
- Lazy implementation of **atmost** based on “source pointers”
- Failed-literal detection, also called lookahead, for stronger propagation

Overview

Characterizing non-cyclic derivations

An alternative approach

Question Is there a propositional formula $F(\Pi)$ such that the models of $F(\Pi)$ correspond to the answer sets of Π ?

- ☞ If we consider the completion of a program, $Comp(\Pi)$, then the problem boils down to eliminating the circular support of atoms that are true in the supported models of Π.

Idea Add formulas to $Comp(\Pi)$ that prohibit circular support of sets of atoms.

- ☞ Circular support between atoms p and q is possible if p has a path to q and q has a path to p in a program's positive atom dependency graph.

Loops

Let Π be a normal logic program, and
let $G(\Pi) = (atom(\Pi), E)$ be the positive atom dependency graph of Π.

- A set $\emptyset \subset L \subseteq atom(\Pi)$ is a **loop** of Π if it induces a non-trivial strongly connected subgraph of $G(\Pi)$.
- That is, each pair of atoms in L is connected by a path of non-zero length in $(L, E \cap (L \times L))$.
- We denote the set of all loops of Π by $Loop(\Pi)$.

Observation Program Π is tight iff $Loop(\Pi) = \emptyset$.

Loop formulas

Let Π be a normal logic program.

- For $L \subseteq atom(\Pi)$, define the **external supports** of L for Π as

$$ES_\Pi(L) = \{ \, r \in \Pi \mid head(r) \in L, body^+(r) \cap L = \emptyset \, \}.$$

- The (disjunctive) **loop formula** of L for Π is

$$\begin{array}{rcl} LF_\Pi(L) & = & \left(\bigvee_{A \in L} A\right) \rightarrow \left(\bigvee_{r \in ES_\Pi(L)} Comp(body(r))\right) \\ & \equiv & \left(\bigwedge_{r \in ES_\Pi(L)} \neg Comp(body(r))\right) \rightarrow \left(\bigwedge_{A \in L} \neg A\right). \end{array}$$

  ☞ The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported.

- Define

$$LF(\Pi) = \{ \, LF_\Pi(L) \mid L \in Loop(\Pi) \, \}.$$

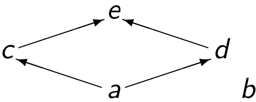
Lin-Zhao Theorem

Theorem

Let Π *be a normal logic program and* $X \subseteq atom(\Pi)$.
Then, X *is an answer set of* Π *iff* $X \models Comp(\Pi) \cup LF(\Pi)$.

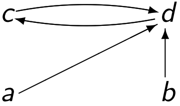
Loops and loop formulas: Examples

$$\Pi_2 = \left\{ \begin{array}{ll} a \leftarrow \textit{not } b & b \leftarrow \textit{not } a \\ c \leftarrow a, \textit{not } d & d \leftarrow a, \textit{not } c \\ e \leftarrow c, \textit{not } a & e \leftarrow d, \textit{not } b \end{array} \right\}$$



$Loop(\Pi_2) = \emptyset$
$LF(\Pi_2) = \emptyset$

$$\Pi_3 = \left\{ \begin{array}{ll} a \leftarrow \textit{not } b & b \leftarrow \textit{not } a \\ c \leftarrow \textit{not } a & c \leftarrow d \\ d \leftarrow a, b & d \leftarrow c \end{array} \right\}$$



$Loop(\Pi_3) = \{\{c, d\}\}$
$LF(\Pi_3) = \{(c \vee d) \rightarrow (\neg a \vee (a \wedge b))\}$

Loops and loop formulas: Properties

Let X be a supported model of normal logic program Π.

Then, X is an answer set of Π iff

- $X \models \{ LF_{\Pi}(U) \mid U \subseteq atom(\Pi) \}$;
- $X \models \{ LF_{\Pi}(U) \mid U \subseteq X \}$;
- $X \models \{ LF_{\Pi}(L) \mid L \in Loop(\Pi) \}$, that is, $X \models LF(\Pi)$;
- $X \models \{ LF_{\Pi}(L) \mid L \in Loop(\Pi), L \subseteq X \}$.
  - ➥ If X is not an answer set of Π,
    then there is a loop $L \subseteq X \setminus Cn(\Pi^X)$ such that $X \not\models LF_{\Pi}(L)$.

Loops and loop formulas: Properties (ctd)

If $\mathcal{P} \not\subseteq \mathcal{NC}^1/poly$,[1] then there is no translation $\mathcal{T}$ from logic programs to propositional formulas such that, for each normal logic program Π, both of the following conditions hold:

1. The propositional variables in $\mathcal{T}[\Pi]$ are a subset of $atom(\Pi)$.
2. The size of $\mathcal{T}[\Pi]$ is polynomial in the size of Π.
   - ☞ Every vocabulary-preserving translation from normal logic programs to propositional formulas must be exponential
     (in the worst case).

Observations

- Translation $Comp(\Pi) \cup LF(\Pi)$ preserves the vocabulary of Π.
- The number of loops in $Loop(\Pi)$ may be exponential in $|atom(\Pi)|$.

1

Tableau Calculi: Overview

Overview

Motivation

Goal Analyze computations in ASP-solvers

Wanted A declarative and fine-grained instrument for characterizing operations as well as strategies of ASP-solvers

Idea View answer set computations as derivations in an inference system

➥ **Tableau-based proof system for analyzing ASP-solving**

Tableau calculi

- Traditionally, tableau calculi are used for
  - automated theorem proving and
  - proof theoretical analysis

  in classical as well as non-classical logics.
- **General idea:** Given an input, prove some property by decomposition. Decomposition is done by applying deduction rules.
- For details, see [17].

Overview

Tableau calculi: General definitions

- A **tableau** is a (mostly binary) tree.
- A **branch** in a tableau is a path from the root to a leaf.
- A branch containing $\gamma_1, \dots, \gamma_m$ can be extended by applying **tableau rules** of form:

$$\frac{\gamma_1, \dots, \gamma_m}{\begin{array}{c}\alpha_1\\ \vdots \\ \alpha_n\end{array}} \qquad\qquad \frac{\gamma_1, \dots, \gamma_m}{\beta_1 \mid \dots \mid \beta_n}$$

- Rules of the former format append entries $\alpha_1, \dots, \alpha_n$ to the branch.
- Rules of the latter format create multiple sub-branches for $\beta_1, \dots, \beta_n$.

Tableau calculus: Example

A simple tableau calculus for proving unsatisfiability of propositional formulas, composed from $\neg$, $\wedge$, and $\vee$, consists of rules:

$$\frac{\neg\neg\alpha}{\alpha} \qquad \frac{\alpha_1 \wedge \alpha_2}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} \qquad \frac{\beta_1 \vee \beta_2}{\beta_1 \mid \beta_2}$$

- All rules are semantically valid, interpreting entries in a branch as connected via "**and**" and distinct (sub-)branches as connected via "**or**".
- A propositional formula φ (composed from $\neg$, $\wedge$, and $\vee$) is unsatisfiable iff there is a tableau with φ as the root node such that
  1. all other entries can be produced by tableau rules and
  2. every branch contains some formulas α and $\neg\alpha$.

Tableau calculus: Example (ctd)

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	$\mathbf{a}$	[1]
(3)	$(\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a$	[1]

(4)	$\neg b \wedge (\neg a \vee b)$	[3]	**(9)**	$\neg\neg\neg a$	[3]
(5)	$\neg\mathbf{b}$	[4]	**(10)**	$\neg\mathbf{a}$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg\mathbf{a}$	[6]	**(8)**	$\mathbf{b}$	[6]

All three branches of the tableau are contradictory (cf. 2, 5, 7, 8, 10).

➥ $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ is unsatisfiable.

Overview

Tableaux and ASP: The idea

- A tableau rule captures an elementary inference scheme in an ASP-solver.
- A **branch** in a tableau corresponds to a successful or unsuccessful **computation** of an answer set.
- An **entire tableau** represents a traversal of the **search space**.

Tableaux and ASP: Specific definitions

- A (signed) **tableau** for a logic program Π is a binary tree such that
  - the root node of the tree consists of the rules in Π;
  - the other nodes in the tree are **entries** of the form $\mathbf{T}v$ or $\mathbf{F}v$, called **signed literals**, where v is a variable,
  - generated by extending a tableau using deduction rules (given below).
- An entry $\mathbf{T}v$ ($\mathbf{F}v$) reflects that variable v is *true* (*false*) in a corresponding variable assignment.
  - ➥ A set of signed literals constitutes a partial assignment.
- For a normal logic program Π,
  - atoms of Π in $atom(\Pi)$ and
  - bodies of Π in $body(\Pi) = \{body(r) \mid r \in \Pi\}$

  can occur as variables in signed literals.

Tableau rules for ASP at a glance [43]

$$\text{(FTB)}\quad \frac{p \leftarrow l_1, \dots, l_n \quad \mathbf{t}l_1, \dots, \mathbf{t}l_n}{\mathbf{T}\{l_1, \dots, l_n\}} \qquad \text{(BFB)}\quad \frac{\mathbf{F}\{l_1, \dots, l_i, \dots, l_n\} \quad \mathbf{t}l_1, \dots, \mathbf{t}l_{i-1}, \mathbf{t}l_{i+1}, \dots, \mathbf{t}l_n}{\mathbf{f}l_i}$$

$$\text{(FTA)}\quad \frac{p \leftarrow l_1, \dots, l_n \quad \mathbf{T}\{l_1, \dots, l_n\}}{\mathbf{T}p} \qquad \text{(BFA)}\quad \frac{p \leftarrow l_1, \dots, l_n \quad \mathbf{F}p}{\mathbf{F}\{l_1, \dots, l_n\}}$$

$$\text{(FFB)}\quad \frac{p \leftarrow l_1, \dots, l_i, \dots, l_n \quad \mathbf{f}l_i}{\mathbf{F}\{l_1, \dots, l_i, \dots, l_n\}} \qquad \text{(BTB)}\quad \frac{\mathbf{T}\{l_1, \dots, l_i, \dots, l_n\}}{\mathbf{t}l_i}$$

$$\text{(FFA)}\quad \frac{\mathbf{F}B_1, \dots, \mathbf{F}B_m}{\mathbf{F}p}\ (\S) \qquad \text{(BTA)}\quad \frac{\mathbf{T}p \quad \mathbf{F}B_1, \dots, \mathbf{F}B_{i-1}, \mathbf{F}B_{i+1}, \dots, \mathbf{F}B_m}{\mathbf{T}B_i}$$

$$\text{(WFN)}\quad \frac{\mathbf{F}B_1, \dots, \mathbf{F}B_m}{\mathbf{F}p}\ (\dagger) \qquad \text{(WFJ)}\quad \frac{\mathbf{T}p \quad \mathbf{F}B_1, \dots, \mathbf{F}B_{i-1}, \mathbf{F}B_{i+1}, \dots, \mathbf{F}B_m}{\mathbf{T}B_i}$$

$$\text{(FL)}\quad \frac{\mathbf{F}B_1, \dots, \mathbf{F}B_m}{\mathbf{F}p}\ (\ddagger) \qquad \text{(BL)}\quad \frac{\mathbf{T}p \quad \mathbf{F}B_1, \dots, \mathbf{F}B_{i-1}, \mathbf{F}B_{i+1}, \dots, \mathbf{F}B_m}{\mathbf{T}B_i}$$

$$(\mathrm{Cut}[X]) \quad \frac{}{\mathbf{T}v \mid \mathbf{F}v} \quad (\sharp[X])$$

More concepts

- A **tableau calculus** is a set of tableau rules.
- A branch in a tableau is **conflicting**, if it contains both $\mathbf{T}v$ and $\mathbf{F}v$ for some variable v.
- A branch in a tableau is **total** for a program Π, if it contains either $\mathbf{T}v$ or $\mathbf{F}v$ for each $v \in atom(\Pi) \cup body(\Pi)$.
- A branch in a tableau of some calculus $\mathcal{T}$ is **closed**, if no rule in $\mathcal{T}$ other than *Cut* can produce any new entries.
- A branch in a tableau is **complete**, if it is either conflicting or both total and closed.
- A tableau is **complete**, if all its branches are complete.
- A tableau of some calculus $\mathcal{T}$ is a **refutation** of $\mathcal{T}$ for a program Π, if every branch in the tableau is conflicting.

Example

Consider the program

$$\Pi = \left\{ \begin{array}{l} a \leftarrow \\ c \leftarrow not\ b, not\ d \\ d \leftarrow a, not\ c \end{array} \right\}$$

having two answer sets $\{a, c\}$ and $\{a, d\}$.

(Previewed) Example

$$a \leftarrow$$
$$c \leftarrow \mathit{not}\ b, \mathit{not}\ d$$
$$d \leftarrow a, \mathit{not}\ c$$

(FTB)	$\mathbf{T}\emptyset$			
(FTA)	$\mathbf{T}a$			
(FFA)	$\mathbf{F}b$			
(Cut[$atom(\Pi)$])		$\mathbf{T}c$		$\mathbf{F}c$
	(BTA)	$\mathbf{T}\{\mathit{not}\ b, \mathit{not}\ d\}$	(BFA)	$\mathbf{F}\{\mathit{not}\ b, \mathit{not}\ d$
	(BTB)	$\mathbf{F}d$	(BFB)	$\mathbf{T}d$
	(FFB)	$\mathbf{F}\{a, \mathit{not}\ c\}$	(FTB)	$\mathbf{T}\{a, \mathit{not}\ c\}$

Recall answer sets $\{a, c\}$ and $\{a, d\}$.

Tableau rules: Auxiliary definitions

- The application of rules makes use of two conjugation functions, **t** and **f**.
- For a literal l, define:

$$\mathbf{t}l = \begin{cases} \mathbf{T}l & \text{if } l \text{ is an atom} \\ \mathbf{F}p & \text{if } l = \mathit{not}\ p \text{ for an atom } p \end{cases}$$

$$\mathbf{f}l = \begin{cases} \mathbf{F}l & \text{if } l \text{ is an atom} \\ \mathbf{T}p & \text{if } l = \mathit{not}\ p \text{ for an atom } p \end{cases}$$

Examples

$$\mathbf{t}p = \mathbf{T}p \qquad \mathbf{f}p = \mathbf{F}p \qquad \mathbf{t}\mathit{not}\ p = \mathbf{F}p \qquad \mathbf{f}\mathit{not}\ p = \mathbf{T}p$$

Tableau rules: Auxiliary definitions (ctd)

- Some tableau rules require conditions for their application. Such conditions are specified as **provisos**:

$$\frac{\textit{prerequisites}}{\textit{consequence}}\ (\textit{proviso})$$

proviso: some condition(s)

☞ All tableau rules given in the sequel are answer set preserving.

Forward True Body (FTB)

Prerequisites All of a body's literals are *true*.

Consequence The body is *true*.

Tableau Rule FTB

$$\frac{\begin{array}{c} p \leftarrow l_1, \dots, l_n \\ \mathbf{t}l_1, \dots, \mathbf{t}l_n \end{array}}{\mathbf{T}\{l_1, \dots, l_n\}}$$

Example

$$\frac{\begin{array}{c} a \leftarrow b, not\ c \\ \mathbf{T}b \\ \mathbf{F}c \end{array}}{\mathbf{T}\{b, not\ c\}}$$

Backward False Body (BFB)

Prerequisites A body is *false*, and all its literals except for one are *true*.

Consequence The residual body literal is *false*.

Tableau Rule BFB

$$\frac{\begin{array}{c}\mathbf{F}\{l_1,\dots,l_i,\dots,l_n\}\\ \mathbf{t}l_1,\dots,\mathbf{t}l_{i-1},\mathbf{t}l_{i+1},\dots,\mathbf{t}l_n\end{array}}{\mathbf{f}\,l_i}$$

Examples

$$\frac{\begin{array}{c}\mathbf{F}\{b, not\ c\}\\ \mathbf{T}b\end{array}}{\mathbf{T}c} \qquad \frac{\begin{array}{c}\mathbf{F}\{b, not\ c\}\\ \mathbf{F}c\end{array}}{\mathbf{F}b}$$

Forward False Body (FFB)

Prerequisites Some literal of a body is *false*.

Consequence The body is *false*.

Tableau Rule FFB

$$\frac{\begin{array}{c} p \leftarrow l_1, \dots, l_i, \dots, l_n \\ \mathbf{f}\, l_i \end{array}}{\mathbf{F}\{l_1, \dots, l_i, \dots, l_n\}}$$

Examples

$$\frac{\begin{array}{c} a \leftarrow b, \mathit{not}\ c \\ \mathbf{F}b \end{array}}{\mathbf{F}\{b, \mathit{not}\ c\}} \qquad \frac{\begin{array}{c} a \leftarrow b, \mathit{not}\ c \\ \mathbf{T}c \end{array}}{\mathbf{F}\{b, \mathit{not}\ c\}}$$

Backward True Body (BTB)

Prerequisites A body is *true*.

Consequence The body's literals are *true*.

Tableau Rule BTB

$$\frac{\mathbf{T}\{l_1, \dots, l_i, \dots, l_n\}}{\mathbf{t}l_i}$$

Examples

$$\frac{\mathbf{T}\{b, \mathit{not}\ c\}}{\mathbf{T}b} \qquad \frac{\mathbf{T}\{b, \mathit{not}\ c\}}{\mathbf{F}c}$$

Reviewing tableau rules for bodies

Consider rule body $B = \{l_1, \dots, l_n\}$.

- Rules FTB and BFB amount to implication:

$$l_1 \wedge \dots \wedge l_n \rightarrow B$$

- Rules FFB and BTB amount to implication:

$$B \rightarrow l_1 \wedge \dots \wedge l_n$$

☞ Together they yield:

$$B \equiv l_1 \wedge \dots \wedge l_n$$

Forward True Atom (FTA)

Prerequisites Some of an atom's bodies is *true*.

Consequence The atom is *true*.

Tableau Rule FTA

$$\frac{\begin{array}{c} p \leftarrow l_1, \dots, l_n \\ \mathbf{T}\{l_1, \dots, l_n\} \end{array}}{\mathbf{T}p}$$

Examples

$$\frac{\begin{array}{c} a \leftarrow b, not\ c \\ \mathbf{T}\{b, not\ c\} \end{array}}{\mathbf{T}a} \qquad \frac{\begin{array}{c} a \leftarrow d, not\ e \\ \mathbf{T}\{d, not\ e\} \end{array}}{\mathbf{T}a}$$

Backward False Atom (BFA)

Prerequisites An atom is *false*.

Consequence The bodies of all rules with the atom as head are *false*.

Tableau Rule BFA

$$\frac{\begin{array}{c} p \leftarrow l_1, \ldots, l_n \\ \mathbf{F}p \end{array}}{\mathbf{F}\{l_1, \ldots, l_n\}}$$

Examples

$$\frac{\begin{array}{c} a \leftarrow b, not\ c \\ \mathbf{F}a \end{array}}{\mathbf{F}\{b, not\ c\}} \qquad \frac{\begin{array}{c} a \leftarrow d, not\ e \\ \mathbf{F}a \end{array}}{\mathbf{F}\{d, not\ e\}}$$

Forward False Atom (FFA)

Prerequisites For some atom, the bodies of all rules with the atom as head are *false*.

Consequence The atom is *false*.

Tableau Rule FFA

$$\frac{\mathbf{F}B_1,\ldots,\mathbf{F}B_m}{\mathbf{F}p}\ (body(p)=\{B_1,\ldots,B_m\})$$

☞ For an atom p occurring in a logic program Π, we let $body(p)=\{body(r)\mid r\in\Pi, head(r)=p\}$.

Example

$$\frac{\begin{array}{c}\mathbf{F}\{b, not\ c\}\\ \mathbf{F}\{d, not\ e\}\end{array}}{\mathbf{F}a}\ (body(a)=\{\{b, not\ c\},\{d, not\ e\}\})$$

Backward True Atom (BTA)

Prerequisites An atom is *true*, and the bodies of all rules with the atom as head except for one are *false*.

Consequence The residual body is *true*.

Tableau Rule BTA

$$\frac{\begin{array}{c}\mathbf{T}p\\ \mathbf{F}B_1,\dots,\mathbf{F}B_{i-1},\mathbf{F}B_{i+1},\dots,\mathbf{F}B_m\end{array}}{\mathbf{T}B_i}\ (body(p)=\{B_1,\dots,B_m\})$$

Examples

$$\frac{\begin{array}{c}\mathbf{T}a\\ \mathbf{F}\{b, not\ c\}\end{array}}{\mathbf{T}\{d, not\ e\}}\ (*) \qquad \frac{\begin{array}{c}\mathbf{T}a\\ \mathbf{F}\{d, not\ e\}\end{array}}{\mathbf{T}\{b, not\ c\}}\ (*)$$

$$(*):\quad body(a)=\{\{b, not\ c\},\{d, not\ e\}\}$$

Reviewing tableau rules for atoms

Consider an atom p such that $body(p) = \{B_1, \dots, B_m\}$.

- Rules FTA and BFA amount to implication:

$$B_1 \vee \cdots \vee B_m \rightarrow p$$

- Rules FFA and BTA amount to implication:

$$p \rightarrow B_1 \vee \cdots \vee B_m$$

☞ Together they yield:

$$p \equiv B_1 \vee \cdots \vee B_m$$

Relationship with Clark's completion

Let Π be a normal logic program.
The eight tableau rules introduced so far essentially provide:

- (straightforward) inferences from $Comp(\Pi)$ (cf. Page 192)
- inferences via **atleast** (cf. Page 210)

Given the same partial assignment (of atoms),

- any literal derived by **atleast** is also derived by tableau rules,
- while the converse does not hold in general.

Preliminaries for unfounded sets

Let Π be a normal logic program.

- For $\Pi' \subseteq \Pi$, define the **greatest unfounded set**, denoted by $GUS(\Pi')$, of Π with respect to Π' as:

$$GUS(\Pi') = atom(\Pi) \setminus Cn((\Pi')^{\emptyset})$$

- For a loop $L \in Loop(\Pi)$, define

$$EB(L) = \{body(r) \mid r \in \Pi, head(r) \in L, body^{+}(r) \cap L = \emptyset\}$$

as the **external bodies** of L.

Well-Founded Negation (WFN)

Prerequisites An atom is in the greatest unfounded set with respect to rules whose bodies are *false*.

Consequence The atom is *false*.

Tableau Rule WFN

$$\frac{\mathbf{F}B_1, \ldots, \mathbf{F}B_m}{\mathbf{F}p} \;(p \in GUS(\{r \in \Pi \mid body(r) \not\in \{B_1, \ldots, B_m\}\}))$$

Examples

$$\frac{\begin{array}{c} a \leftarrow not\ b \\ \mathbf{F}\{not\ b\} \end{array}}{\mathbf{F}a}\;(*) \qquad\qquad \frac{\begin{array}{c} a \leftarrow a \\ a \leftarrow not\ b \\ \mathbf{F}\{not\ b\} \end{array}}{\mathbf{F}a}\;(*)$$

$$(*):\quad a \in GUS(\Pi \setminus \{a \leftarrow not\ b\})$$

Well-Founded Justification (WFJ)

Prerequisites A *true* atom is in the greatest unfounded set with respect to rules whose bodies are *false* if a particular body is made *false*.

Consequence The respective body is *true*.

Tableau Rule WFJ

$$\frac{\begin{array}{c}\mathbf{T}p\\ \mathbf{F}B_1,\dots,\mathbf{F}B_{i-1},\mathbf{F}B_{i+1},\dots,\mathbf{F}B_m\end{array}}{\mathbf{T}B_i}\ (p \in GUS(\{r \in \Pi \mid body(r) \notin \{B_1,\dots,B_m\}\}))$$

Examples

$$\frac{\begin{array}{c}a \leftarrow not\ b\\ \mathbf{T}a\end{array}}{\mathbf{T}\{not\ b\}}\ (*) \qquad \frac{\begin{array}{c}a \leftarrow a\\ a \leftarrow not\ b\\ \mathbf{T}a\end{array}}{\mathbf{T}\{not\ b\}}\ (*)$$

$$(*):\quad a \in GUS(\Pi \setminus \{a \leftarrow not\ b\})$$

Reviewing well-founded tableau rules

Tableau rules WFN and WFJ ensure non-circular support for *true* atoms.
Note that

1. WFN subsumes falsifying atoms via FFA,
2. WFJ can be viewed as "backward propagation" for unfounded sets,
3. WFJ subsumes backward propagation of *true* atoms via BTA.

Relationship with well-founded operator

Let Π be a normal logic program, $\langle T, F\rangle$ a partial interpretation, and $\Pi' = \{r \in \Pi \mid body^+(r) \cap F = \emptyset, body^-(r) \cap T = \emptyset\}$.
Then the following conditions are equivalent:

1. $p \in \mathbf{U}_\Pi\langle T, F\rangle$; (cf. Page 223)
2. $p \in \mathbf{atmost(}\langle T, F\rangle\mathbf{)}$; (cf. Page 234)
3. $p \in GUS(\Pi')$.
   - ➥ Well-founded operator, **atmost**, and WFN coincide.

- ☞ In contrast to the former, WFN does not necessarily require a rule body to contain a *false* literal for the rule being inapplicable.

Forward Loop (FL)

Prerequisites The external bodies of a loop are *false*.

Consequence The atoms in the loop are *false*.

Tableau Rule FL

$$\frac{\mathbf{F}B_1, \dots, \mathbf{F}B_m}{\mathbf{F}p} \ (p \in L, L \in Loop(\Pi), EB(L) = \{B_1, \dots, B_m\})$$

Example

$$\begin{array}{c} a \leftarrow a \\ a \leftarrow not\ b \\ \hline \end{array}$$

$$\frac{\mathbf{F}\{not\ b\}}{\mathbf{F}a} \ (EB(\{a\}) = \{\{not\ b\}\})$$

Backward Loop (BL)

Prerequisites An atom of a loop is *true*, and all external bodies except for one are *false*.

Consequence The residual external body is *true*.

Tableau Rule BL

$$\frac{\begin{array}{c}\mathbf{T}p \\ \mathbf{F}B_1, \ldots, \mathbf{F}B_{i-1}, \mathbf{F}B_{i+1}, \ldots, \mathbf{F}B_m\end{array}}{\mathbf{T}B_i} \; (p \in L, L \in \mathit{Loop}(\Pi), \mathit{EB}(L) = \{B_1, \ldots, B_m\})$$

Example

$$\frac{\begin{array}{c}a \leftarrow a \\ a \leftarrow \mathit{not}\ b \\ \mathbf{T}a\end{array}}{\mathbf{T}\{\mathit{not}\ b\}} \; (\mathit{EB}(\{a\}) = \{\{\mathit{not}\ b\}\})$$

Reviewing tableau rules for loops

Tableau rules FL and BL ensure non-circular support for *true* atoms.
For a loop L such that $EB(L) = \{B_1, \ldots, B_m\}$,
they amount to implication:

$$\bigvee_{p \in L} p \rightarrow B_1 \vee \cdots \vee B_m$$

Comparison to well-founded tableau rules yields:

- FL (plus FFA and FFB) is equivalent to WFN (plus FFB),
- BL cannot simulate inferences via WFJ.

Relationship with loop formulas

Tableau rules FL and BL essentially provide:

- (straightforward) inferences from loop formulas (cf. Page 242)
  - ☞ But impractical to precompute exponentially many loop formulas !
- an application of the Lin-Zhao Theorem (cf. Page 243)

In practice, ASP-solvers such as `smodels`:

- exploit strongly connected components of positive atom dependency graphs
  - ➥ Can be viewed as an interpolation of FL.
- do not directly implement BL (and neither WFJ)
  - ➥ Probably difficult to do efficiently.
- could simulate BL via FL/WFN by means of failed-literal detection (lookahead)
  - ➥ What about the computational cost?

Case analysis by *Cut*

Up to now, all tableau rules are deterministic.
That is, rules extend a single branch but cannot create sub-branches.

☞ In general, closing a branch leads to a partial assignment.

Case analysis is done by $Cut[\mathcal{C}]$ where $\mathcal{C} \subseteq atom(\Pi) \cup body(\Pi)$.

Tableau Rule $Cut[\mathcal{C}]$

$$\frac{}{\mathbf{T}v \quad \mid \quad \mathbf{F}v} \; (v \in \mathcal{C})$$

Examples $Cut[\mathcal{C}]$

$$\frac{\begin{array}{c} a \leftarrow not\ b \\ b \leftarrow not\ a \end{array}}{\mathbf{T}a \quad \mid \quad \mathbf{F}a} \; (\mathcal{C} = atom(\Pi)) \qquad \frac{\begin{array}{c} a \leftarrow not\ b \\ b \leftarrow not\ a \end{array}}{\mathbf{T}\{not\ b\} \quad \mid \quad \mathbf{F}\{not\ b\}} \; (\mathcal{C} = body(\Pi))$$

Well-known tableau calculi

Fitting's operator $\mathbf{\Phi}$ applies forward propagation without sophisticated unfounded set checks. We have:

$$\mathcal{T}_{\mathbf{\Phi}} = \{FTB, FTA, FFB, FFA\}$$

Well-founded operator $\mathbf{\Omega}$ replaces negation of single atoms with negation of unfounded sets. We have:

$$\mathcal{T}_{\mathbf{\Omega}} = \{FTB, FTA, FFB, WFN\}$$

"Local" propagation via a program's completion can be determined by elementary inferences on atoms and rule bodies. We have:

$$\mathcal{T}_{completion} = \{FTB, FTA, FFB, FFA, BTB, BTA, BFB, BFA\}$$

Tableau calculi characterizing ASP-solvers

ASP-solvers combine propagation with case analysis.
We obtain the following tableau calculi characterizing [4, 63, 51, 77, 57, 54, 2]:

$$
\begin{aligned}
\mathcal{T}_{\mathit{cmodels\text{-}1}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{Cut}[\mathit{atom}(\Pi) \cup \mathit{body}(\Pi)]\} \\
\mathcal{T}_{\mathit{assat}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{FL}\} \cup \{\mathit{Cut}[\mathit{atom}(\Pi) \cup \mathit{body}(\Pi)]\} \\
\mathcal{T}_{\mathit{smodels}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[\mathit{atom}(\Pi)]\} \\
\mathcal{T}_{\mathit{noMoRe}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[\mathit{body}(\Pi)]\} \\
\mathcal{T}_{\mathit{nomore}^{++}} &= \mathcal{T}_{\mathit{completion}} \cup \{\mathit{WFN}\} \cup \{\mathit{Cut}[\mathit{atom}(\Pi) \cup \mathit{body}(\Pi)]\}
\end{aligned}
$$

- SAT-based ASP-solvers, `assat` and `cmodels`, incrementally add loop formulas to a program's completion.
- Genuine ASP-solvers, `smodels`, `dlv`, `noMoRe`, and `nomore++`, essentially differ only in their *Cut* rules.

Proof complexity

The notion of **proof complexity** is used for describing the relative efficiency of different proof systems.
It compares proof systems based on **minimal refutations**.

➥ Proof complexity does not depend on heuristics.

A proof system $\mathcal{T}$ **polynomially simulates** a proof system $\mathcal{T}'$ if every refutation of $\mathcal{T}'$ can be polynomially mapped to a refutation of $\mathcal{T}$.
Otherwise, $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$.
For showing that proof system $\mathcal{T}$ does not polynomially simulate $\mathcal{T}'$, we have to provide an infinite **witnessing family** of programs such that minimal refutations of $\mathcal{T}$ asymptotically are exponentially larger than minimal refutations of $\mathcal{T}'$.
The size of tableaux is simply the number of their entries.

☞ We do not need to know the precise number of entries: Counting required *Cut* applications is sufficient !

$\mathcal{T}_{smodels}$ versus $\mathcal{T}_{noMoRe}$

Recall that $\mathcal{T}_{smodels}$ restricts *Cut* to $atom(\Pi)$ and $\mathcal{T}_{noMoRe}$ to $body(\Pi)$.
Are both approaches similar or is one of them superior to the other?
Let $\{\Pi_a^n\}$, $\{\Pi_b^n\}$, and $\{\Pi_c^n\}$ be infinite families of programs as follows:

$$\Pi_a^n = \left\{ \begin{array}{l} x \leftarrow not\ x \\ x \leftarrow a_1, b_1 \\ \quad\vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad \Pi_b^n = \left\{ \begin{array}{ll} x \leftarrow c_1, \dots, c_n, not\ x & \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \quad\vdots & \quad\vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad \Pi_c^n = \left\{ \begin{array}{l} a_1 \leftarrow not\ b \\ b_1 \leftarrow not\ a \\ \quad\vdots \\ a_n \leftarrow not\ b \\ b_n \leftarrow not\ a \end{array} \right.$$

In minimal refutations for $\Pi_a^n \cup \Pi_c^n$, the number of applications of $Cut[body(\Pi_a^n \cup \Pi_c^n)]$ with $\mathcal{T}_{noMoRe}$ is linear in n, whereas $\mathcal{T}_{smodels}$ requires exponentially many applications of $Cut[atom(\Pi_a^n \cup \Pi_c^n)]$.
Vice versa, minimal refutations for $\Pi_b^n \cup \Pi_c^n$ require linearly many applications of $Cut[atom(\Pi_b^n \cup \Pi_c^n)]$ with $\mathcal{T}_{smodels}$ and exponentially many applications of $Cut[body(\Pi_b^n \cup \Pi_c^n)]$ with $\mathcal{T}_{noMoRe}$.

Relative efficiency

As witnessed by $\{\Pi_a^n \cup \Pi_c^n\}$ and $\{\Pi_b^n \cup \Pi_c^n\}$, respectively, $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ do not polynomially simulate one another.
Any refutation of $\mathcal{T}_{smodels}$ or $\mathcal{T}_{noMoRe}$ is a refutation of $\mathcal{T}_{nomore^{++}}$ (but not vice versa).

It follows that

- both $\mathcal{T}_{smodels}$ and $\mathcal{T}_{noMoRe}$ are polynomially simulated by $\mathcal{T}_{nomore^{++}}$ and
- $\mathcal{T}_{nomore^{++}}$ is polynomially simulated by neither $\mathcal{T}_{smodels}$ nor $\mathcal{T}_{noMoRe}$.

➥ The proof system obtained with $Cut[atom(\Pi) \cup body(\Pi)]$ is **exponentially stronger** than the ones with either $Cut[atom(\Pi)]$ or $Cut[body(\Pi)]$!

☞ Case analyses (at least) on atoms and bodies are mandatory in powerful ASP-solvers.

$\mathcal{T}_{smodels}$: Example tableau

(r_1)	$a \leftarrow not\ b$	(r_2)	$b \leftarrow d, not\ a$	(r_3)	$c \leftarrow b, d$
(r_4)	$c \leftarrow g$	(r_5)	$d \leftarrow c$	(r_6)	$d \leftarrow g$
(r_7)	$e \leftarrow f, not\ c$	(r_8)	$f \leftarrow not\ g$	(r_9)	$g \leftarrow not\ a, not\ f$

(1)	$\mathbf{T}a$	$[Cut]$
(2)	$\mathbf{T}\{not\ b\}$	$[BTA\text{: } r_1, 1]$
(3)	$\mathbf{F}b$	$[BTB\text{: } 2]$
(4)	$\mathbf{F}\{d, not\ a\}$	$[BFA\text{: } r_2, 3]$
(5)	$\mathbf{F}\{not\ a, not\ f\}$	$[FFB\text{: } r_9, 1]$
(6)	$\mathbf{F}g$	$[FFA\text{: } r_9, 5]$
(7)	$\mathbf{T}\{not\ g\}$	$[FTB\text{: } r_8, 6]$
(8)	$\mathbf{T}f$	$[FTA\text{: } r_8, 7]$
(9)	$\mathbf{F}\{b, d\}$	$[FFB\text{: } r_3, 3]$
(10)	$\mathbf{F}\{g\}$	$[FFB\text{: } r_4, r_6, 6]$
(11)	$\mathbf{F}c$	$[FFA\text{: } r_3, r_4, 9, 10]$
(12)	$\mathbf{F}\{c\}$	$[FFB\text{: } r_5, 11]$
(13)	$\mathbf{F}d$	$[FFA\text{: } r_5, r_6, 10, 12]$
(14)	$\mathbf{T}\{f, not\ c\}$	$[FTB\text{: } r_7, 8, 11]$
(15)	$\mathbf{T}e$	$[FTA\text{: } r_7, 14]$

(16)	$\mathbf{F}a$	$[Cut]$
(17)	$\mathbf{F}\{not\ b\}$	$[BFA\text{: } r_1, 16]$
(18)	$\mathbf{T}b$	$[BFB\text{: } 17]$
(19)	$\mathbf{T}\{d, not\ a\}$	$[BTA\text{: } r_2, 18]$
(20)	$\mathbf{T}d$	$[BTB\text{: } 19]$
(21)	$\mathbf{T}\{b, d\}$	$[FTB\text{: } r_3, 18, 20]$
(22)	$\mathbf{T}c$	$[FTA\text{: } r_3, 21]$
(23)	$\mathbf{F}\{f, not\ c\}$	$[FFB\text{: } r_7, 22]$
(24)	$\mathbf{F}e$	$[FFA\text{: } r_7, 23]$
(25)	$\mathbf{T}\{c\}$	$[FTB\text{: } r_5, 22]$

(26)	$\mathbf{T}f$	$[Cut]$
(27)	$\mathbf{F}\{not\ a, not\ f\}$	$[FFB\text{: } r_9, 26]$
(28)	$\mathbf{F}c$	$[WFN\text{: } 27]$

(29)	$\mathbf{F}f$	$[Cut]$
(30)	$\mathbf{T}\{not\ a, not\ f\}$	$[FTB\text{: } r_9,$
(31)	$\mathbf{T}g$	$[FTA\text{: } r_9, 3$
(32)	$\mathbf{T}\{g\}$	$[FTB\text{: } r_4,$
(33)	$\mathbf{F}\{not\ g\}$	$[FFB\text{: } r_8, 3$

$\mathcal{T}_{noMoRe}$: Example tableau

(r_1)	$a \leftarrow not\ b$	(r_2)	$b \leftarrow d, not\ a$	(r_3)	$c \leftarrow b, d$
(r_4)	$c \leftarrow g$	(r_5)	$d \leftarrow c$	(r_6)	$d \leftarrow g$
(r_7)	$e \leftarrow f, not\ c$	(r_8)	$f \leftarrow not\ g$	(r_9)	$g \leftarrow not\ a, not\ f$

(1)	$\mathbf{T}\{not\ b\}$	$[Cut]$
(2)	$\mathbf{T}a$	$[FTA\text{: } r_1, 1]$
(3)	$\mathbf{F}b$	$[BTB\text{: } 1]$
(4)	$\mathbf{F}\{d, not\ a\}$	$[BFA\text{: } r_2, 3]$
(5)	$\mathbf{F}\{not\ a, not\ f\}$	$[FFB\text{: } r_9, 2]$
(6)	$\mathbf{F}g$	$[FFA\text{: } r_9, 5]$
(7)	$\mathbf{T}\{not\ g\}$	$[FTB\text{: } r_8, 6]$
(8)	$\mathbf{T}f$	$[FTA\text{: } r_8, 7]$
(9)	$\mathbf{F}\{b, d\}$	$[FFB\text{: } r_3, 3]$
(10)	$\mathbf{F}\{g\}$	$[FFB\text{: } r_4, r_6, 6]$
(11)	$\mathbf{F}c$	$[FFA\text{: } r_3, r_4, 9, 10]$
(12)	$\mathbf{F}\{c\}$	$[FFB\text{: } r_5, 11]$
(13)	$\mathbf{F}d$	$[FFA\text{: } r_5, r_6, 10, 12]$
(14)	$\mathbf{T}\{f, not\ c\}$	$[FTB\text{: } r_7, 8, 11]$
(15)	$\mathbf{T}e$	$[FTA\text{: } r_7, 14]$

(16)	$\mathbf{F}\{not\ b\}$	$[Cut]$
(17)	$\mathbf{F}a$	$[FFA\text{: } r_1, 16]$
(18)	$\mathbf{T}b$	$[BFB\text{: } 16]$
(19)	$\mathbf{T}\{d, not\ a\}$	$[BTA\text{: } r_2, 18]$
(20)	$\mathbf{T}d$	$[BTB\text{: } 19]$
(21)	$\mathbf{T}\{b, d\}$	$[FTB\text{: } r_3, 18, 20]$
(22)	$\mathbf{T}c$	$[FTA\text{: } r_3, 21]$
(23)	$\mathbf{F}\{f, not\ c\}$	$[FFB\text{: } r_7, 22]$
(24)	$\mathbf{F}e$	$[FFA\text{: } r_7, 23]$
(25)	$\mathbf{T}\{c\}$	$[FTB\text{: } r_5, 22]$

(26)	$\mathbf{T}\{not\ g\}$	$[Cut]$
(27)	$\mathbf{F}g$	$[BTB\text{: } 26]$
(28)	$\mathbf{F}\{g\}$	$[FFB\text{: } r_4, r_6, 27]$
(29)	$\mathbf{F}c$	$[WFN\text{: } 28]$

(30)	$\mathbf{F}\{not\ g\}$	$[Cut]$
(31)	$\mathbf{T}g$	$[BFB\text{: } 30]$
(32)	$\mathbf{T}\{g\}$	$[FTB\text{: } r_4, r$
(33)	$\mathbf{F}f$	$[FFA\text{: } r_8, 3$
(34)	$\mathbf{T}\{not\ a, not\ f\}$	$[FTB\text{: } r_9, 1$

$\mathcal{T}_{nomore^{++}}$: Example tableau

$$
\begin{array}{llllll}
(r_1) & a \leftarrow not\ b & (r_2) & b \leftarrow d, not\ a & (r_3) & c \leftarrow b, d \\
(r_4) & c \leftarrow g & (r_5) & d \leftarrow c & (r_6) & d \leftarrow g \\
(r_7) & e \leftarrow f, not\ c & (r_8) & f \leftarrow not\ g & (r_9) & g \leftarrow not\ a, not\ f
\end{array}
$$

(1)	$\mathbf{T}a$	$[Cut]$
(2)	$\mathbf{T}\{not\ b\}$	$[BTA\text{: } r_1, 1]$
(3)	$\mathbf{F}b$	$[BTB\text{: } 2]$
(4)	$\mathbf{F}\{d, not\ a\}$	$[BFA\text{: } r_2, 3]$
(5)	$\mathbf{F}\{not\ a, not\ f\}$	$[FFB\text{: } r_9, 1]$
(6)	$\mathbf{F}g$	$[FFA\text{: } r_9, 5]$
(7)	$\mathbf{T}\{not\ g\}$	$[FTB\text{: } r_8, 6]$
(8)	$\mathbf{T}f$	$[FTA\text{: } r_8, 7]$
(9)	$\mathbf{F}\{b, d\}$	$[FFB\text{: } r_3, 3]$
(10)	$\mathbf{F}\{g\}$	$[FFB\text{: } r_4, r_6, 6]$
(11)	$\mathbf{F}c$	$[FFA\text{: } r_3, r_4, 9, 10]$
(12)	$\mathbf{F}\{c\}$	$[FFB\text{: } r_5, 11]$
(13)	$\mathbf{F}d$	$[FFA\text{: } r_5, r_6, 10, 12]$
(14)	$\mathbf{T}\{f, not\ c\}$	$[FTB\text{: } r_7, 8, 11]$
(15)	$\mathbf{T}e$	$[FTA\text{: } r_7, 14]$

(16)	$\mathbf{F}a$	$[Cut]$
(17)	$\mathbf{F}\{not\ b\}$	$[BFA\text{: } r_1, 16]$
(18)	$\mathbf{T}b$	$[BFB\text{: } 17]$
(19)	$\mathbf{T}\{d, not\ a\}$	$[BTA\text{: } r_2, 18]$
(20)	$\mathbf{T}d$	$[BTB\text{: } 19]$
(21)	$\mathbf{T}\{b, d\}$	$[FTB\text{: } r_3, 18, 20]$
(22)	$\mathbf{T}c$	$[FTA\text{: } r_3, 21]$
(23)	$\mathbf{F}\{f, not\ c\}$	$[FFB\text{: } r_7, 22]$
(24)	$\mathbf{F}e$	$[FFA\text{: } r_7, 23]$
(25)	$\mathbf{T}\{c\}$	$[FTB\text{: } r_5, 22]$

(26)	$\mathbf{T}\{not\ g\}$	$[Cut]$
(27)	$\mathbf{F}g$	$[BTB\text{: } 26]$
(28)	$\mathbf{F}\{g\}$	$[FFB\text{: } r_4, r_6, 27]$
(29)	$\mathbf{F}c$	$[WFN\text{: } 28]$

(30)	$\mathbf{F}\{not\ g\}$	$[Cut]$
(31)	$\mathbf{T}g$	$[BFB\text{: } 30]$
(32)	$\mathbf{T}\{g\}$	$[FTB\text{: } r_4,$
(33)	$\mathbf{F}f$	$[FFA\text{: } r_8, 3$
(34)	$\mathbf{T}\{not\ a, not\ f\}$	$[FTB\text{: } r_9,$

Conflict-Driven Answer Set Solving: Overview

Overview

Motivation

Goal New approach to computing answer sets of logic programs, based on concepts from

- Constraint Processing (CSP) and
- Satisfiability Checking (SAT)

Idea View inferences in Answer Set Programming (ASP) as unit propagation on nogoods.

Benefits

- A uniform constraint-based framework for different kinds of inferences in ASP
- Advanced techniques from the areas of CSP and SAT
- Highly competitive implementation

Overview

Assignments

- An **assignment** A over $dom(A) = atom(\Pi) \cup body(\Pi)$ is a sequence

  $$(\sigma_1, \dots, \sigma_n)$$

  of **signed literals** σ_i of form $\mathbf{T}p$ or $\mathbf{F}p$ for $p \in dom(A)$ and $1 \leq i \leq n$.

  ☞ $\mathbf{T}p$ expresses that p is *true* and $\mathbf{F}p$ that it is *false*.

- The complement, $\overline{\sigma}$, of a literal σ is defined as $\overline{\mathbf{T}p} = \mathbf{F}p$ and $\overline{\mathbf{F}p} = \mathbf{T}p$.
- $A \circ B$ denotes the concatenation of assignments A and B.
- Given $A = (\sigma_1, \dots, \sigma_{k-1}, \sigma_k, \dots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \dots, \sigma_{k-1})$.
- We sometimes identify an assignment with the set of its literals. Given this, we access *true* and *false* propositions in A via

  $$A^{\mathbf{T}} = \{p \in dom(A) \mid \mathbf{T}p \in A\} \text{ and } A^{\mathbf{F}} = \{p \in dom(A) \mid \mathbf{F}p \in A\} .$$

Nogoods, Solutions, and Unit Propagation

- A **nogood** is a set $\{\sigma_1, \dots, \sigma_n\}$ of signed literals, expressing a **constraint** violated by any assignment containing $\sigma_1, \dots, \sigma_n$.
- An assignment A such that $A^{\mathbf{T}} \cup A^{\mathbf{F}} = dom(A)$ and $A^{\mathbf{T}} \cap A^{\mathbf{F}} = \emptyset$ is a **solution** for a set Δ of nogoods, if $\delta \not\subseteq A$ for all $\delta \in \Delta$.
- For a nogood δ, a literal $\sigma \in \delta$, and an assignment A, we say that $\overline{\sigma}$ is **unit-resulting** for δ wrt A, if
  1. $\delta \setminus A = \{\sigma\}$ and
  2. $\overline{\sigma} \notin A$.
- For a set Δ of nogoods and an assignment A, **unit propagation** is the iterated process of extending A with unit-resulting literals until no further literal is unit-resulting for any nogood in Δ.

Overview

Nogoods from logic programs

via Clark's completion

The completion of a logic program Π can be defined as follows:

$$\{p_\beta \leftrightarrow p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n \mid \\ \beta \in \mathit{body}(\Pi), \beta = \{p_1, \ldots, p_m, \mathit{not}\ p_{m+1}, \ldots, \mathit{not}\ p_n\}\}$$

$$\cup \quad \{p \leftrightarrow p_{\beta_1} \vee \cdots \vee p_{\beta_k} \mid \\ p \in \mathit{atom}(\Pi), \mathit{body}(p) = \{\beta_1, \ldots, \beta_k\}\} \ ,$$

where $\mathit{body}(p) = \{\mathit{body}(r) \mid r \in \Pi, \mathit{head}(r) = p\}$.

Nogoods from logic programs (ctd)

via Clark's completion

Let $\beta = \{p_1, \ldots, p_m, \mathit{not}\ p_{m+1}, \ldots, \mathit{not}\ p_n\}$ be a body.
The equivalence

$$p_\beta \leftrightarrow p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n$$

can be decomposed into two implications.

1. We get

$$p_\beta \rightarrow p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n \ ,$$

which is equivalent to the conjunction of

$$\neg p_\beta \vee p_1,\ \ldots,\ \neg p_\beta \vee p_m,\ \neg p_\beta \vee \neg p_{m+1},\ \ldots,\ \neg p_\beta \vee \neg p_n \ .$$

This set of clauses expresses the following set of nogoods:

$$\Delta(\beta) = \{\, \{\mathbf{T}\beta, \mathbf{F}p_1\}, \ldots, \{\mathbf{T}\beta, \mathbf{F}p_m\}, \{\mathbf{T}\beta, \mathbf{T}p_{m+1}\}, \ldots, \{\mathbf{T}\beta, \mathbf{T}p_n\} \,\} \ .$$

Nogoods from logic programs (ctd)

via Clark's completion

2 The converse of the previous implication, viz.

$$p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n \rightarrow p_\beta \ ,$$

gives rise to the nogood

$$\delta(\beta) = \{\mathbf{F}\beta, \mathbf{T}p_1, \ldots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \ldots, \mathbf{F}p_n\} \ .$$

Intuitively, $\delta(\beta)$ is a constraint enforcing the truth of body β, or the falsity of a contained literal.

Nogoods from logic programs (ctd)

via Clark's completion

Proceeding analogously with the atom-based equivalences, viz.

$$p \leftrightarrow p_{\beta_1} \vee \cdots \vee p_{\beta_k}$$

we obtain for an atom $p \in atom(\Pi)$ along with its bodies $body(p) = \{\beta_1, \ldots, \beta_k\}$ the nogoods

$$\Delta(p) = \{\, \{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\,\} \text{ and}$$

$$\delta(p) = \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\} \ .$$

Nogoods from logic programs

atom-oriented nogoods

For an atom p where $body(p) = \{\beta_1, \dots, \beta_k\}$, recall that

$$\begin{aligned} \delta(p) &= \{\mathbf{T}p, \mathbf{F}\beta_1, \dots, \mathbf{F}\beta_k\} \\ \Delta(p) &= \{\, \{\mathbf{F}p, \mathbf{T}\beta_1\}, \dots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\,\} \;. \end{aligned}$$

For example, for atom x with $body(x) = \{\{y\}, \{not\ z\}\}$, we obtain

$$\boxed{\begin{array}{lcl} x & \leftarrow & y \\ x & \leftarrow & not\ z \end{array}} \qquad \begin{aligned} \delta(x) &= \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\} \\ \Delta(x) &= \{\, \{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\}\,\} \end{aligned}$$

For nogood $\delta(x) = \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\}$, the signed literal

- $\mathbf{F}x$ is unit-resulting wrt assignment $(\mathbf{F}\{y\}, \mathbf{F}\{not\ z\})$ and
- $\mathbf{T}\{not\ z\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}\{y\})$.

Nogoods from logic programs

body-oriented nogoods

For a body $\beta = \{p_1, \dots, p_m, \mathit{not}\ p_{m+1}, \dots, \mathit{not}\ p_n\}$, recall that

$$\begin{array}{rcl} \delta(\beta) & = & \{\mathbf{F}\beta, \mathbf{T}p_1, \dots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \dots, \mathbf{F}p_n\} \\ \Delta(\beta) & = & \{\,\{\mathbf{T}\beta, \mathbf{F}p_1\}, \dots, \{\mathbf{T}\beta, \mathbf{F}p_m\}, \{\mathbf{T}\beta, \mathbf{T}p_{m+1}\}, \dots, \{\mathbf{T}\beta, \mathbf{T}p_n\}\,\} \end{array}$$

For example, for body $\{x, \mathit{not}\ y\}$, we obtain

$$\boxed{\begin{array}{c} \dots \leftarrow x, \mathit{not}\ y \\ \vdots \\ \dots \leftarrow x, \mathit{not}\ y \end{array}} \quad \begin{array}{l} \delta(\{x, \mathit{not}\ y\}) = \{\mathbf{F}\{x, \mathit{not}\ y\}, \mathbf{T}x, \mathbf{F}y\} \\ \Delta(\{x, \mathit{not}\ y\}) = \{\,\{\mathbf{T}\{x, \mathit{not}\ y\}, \mathbf{F}x\}, \{\mathbf{T}\{x, \mathit{not}\ y\}, \mathbf{T} \end{array}$$

For nogood $\delta(\{x, \mathit{not}\ y\}) = \{\mathbf{F}\{x, \mathit{not}\ y\}, \mathbf{T}x, \mathbf{F}y\}$, the signed literal

- $\mathbf{T}\{x, \mathit{not}\ y\}$ is unit-resulting wrt assignment $(\mathbf{T}x, \mathbf{F}y)$ and
- $\mathbf{T}y$ is unit-resulting wrt assignment $(\mathbf{F}\{x, \mathit{not}\ y\}, \mathbf{T}x)$.

Characterization of answer sets

for tight logic programs

Let Π be a logic program and

$$\begin{array}{rcl} \Delta_\Pi & = & \{\delta(p) \mid p \in atom(\Pi)\} \cup \{\delta \in \Delta(p) \mid p \in atom(\Pi)\} \\ & \cup & \{\delta(\beta) \mid \beta \in body(\Pi)\} \cup \{\delta \in \Delta(\beta) \mid \beta \in body(\Pi)\} \ . \end{array}$$

Theorem

Let Π be a **tight** *logic program. Then,*
$X \subseteq atom(\Pi)$ is an answer set of Π **iff**
$X = A^{\mathbf{T}} \cap atom(\Pi)$ for a (unique) solution A for Δ_Π.

☞ The set Δ_Π of nogoods captures inferences from (program Π and) Clark's completion.

Atom-oriented nogoods and tableau rules

- Tableau rules FTA, BFA, FFA, and BTA are atom-oriented.
- For an atom p such that $body(p) = \{\beta_1, \dots, \beta_k\}$,
  consider the equivalence: $p \leftrightarrow p_{\beta_1} \vee \cdots \vee p_{\beta_k}$
- Inferences from nogoods $\Delta(p) = \{ \{\mathbf{F}p, \mathbf{T}\beta_1\}, \dots, \{\mathbf{F}p, \mathbf{T}\beta_k\} \}$
  correspond to those from tableau rules **FTA** and **BFA**:

$$\frac{p \leftarrow \beta \quad \mathbf{T}\beta}{\mathbf{T}p} \qquad \frac{p \leftarrow \beta \quad \mathbf{F}p}{\mathbf{F}\beta}$$

- Inferences from nogood $\delta(p) = \{\mathbf{T}p, \mathbf{F}\beta_1, \dots, \mathbf{F}\beta_k\}$
  correspond to those from tableau rules **FFA** and **BTA**:

$$\frac{\mathbf{F}\beta_1, \dots, \mathbf{F}\beta_k}{\mathbf{F}p} \qquad \frac{\mathbf{T}p \quad \mathbf{F}\beta_1, \dots, \mathbf{F}\beta_{i-1}, \mathbf{F}\beta_{i+1}, \dots, \mathbf{F}\beta_k}{\mathbf{T}\beta_i}$$

Body-oriented nogoods and tableau rules

- Tableau rules FTB, BFB, FFB, and BTB are body-oriented.
- For a body $\beta = \{p_1, \dots, p_m, not\ p_{m+1}, \dots, not\ p_n\} = \{l_1, \dots, l_n\}$, consider the equivalence: $p_\beta \leftrightarrow p_1 \wedge \cdots \wedge p_m \wedge \neg p_{m+1} \wedge \cdots \wedge \neg p_n$
- Inferences from nogood $\delta(\beta) = \{\mathbf{F}\beta, \mathbf{T}p_1, \dots, \mathbf{T}p_m, \mathbf{F}p_{m+1}, \dots, \mathbf{F}p_n\}$ correspond to those from tableau rules **FTB** and **BFB**:

$$\frac{\begin{array}{c} p \leftarrow l_1, \dots, l_n \\ \mathbf{t}l_1, \dots, \mathbf{t}l_n \end{array}}{\mathbf{T}\{l_1, \dots, l_n\}} \qquad \frac{\begin{array}{c} \mathbf{F}\{l_1, \dots, l_n\} \\ \mathbf{t}l_1, \dots, \mathbf{t}l_{i-1}, \mathbf{t}l_{i+1}, \dots, \mathbf{t}l_n \end{array}}{\mathbf{f}l_i}$$

- Inferences from nogoods $\Delta(\beta) = \{\,\{\mathbf{T}\beta, \mathbf{F}p_1\}, \dots, \{\mathbf{T}\beta, \mathbf{F}p_m\}, \{\mathbf{T}\beta, \mathbf{T}p_{m+1}\}, \dots, \{\mathbf{T}\beta, \mathbf{T}p_n\}\,\}$ correspond to those from tableau rules **FFB** and **BTB**:

$$\frac{\begin{array}{c} p \leftarrow l_1, \dots, l_i, \dots, l_n \\ \mathbf{f}l_i \end{array}}{\mathbf{F}\{l_1, \dots, l_i, \dots, l_n\}} \qquad \frac{\mathbf{T}\{l_1, \dots, l_i, \dots, l_n\}}{\mathbf{t}l_i}$$

Nogoods from logic programs

via loop formulas (cf. Page 242)

Let Π be a normal logic program and recall that:

- For $L \subseteq atom(\Pi)$, the external supports of L for Π are

$$ES_\Pi(L) \;=\; \{r \in \Pi \mid head(r) \in L, body^+(r) \cap L = \emptyset\}.$$

- The (disjunctive) loop formula of L for Π is

$$\begin{aligned} LF_\Pi(L) &= \left(\bigvee_{A \in L} A\right) \rightarrow \left(\bigvee_{r \in ES_\Pi(L)} Comp(body(r))\right) \\ &\equiv \left(\bigwedge_{r \in ES_\Pi(L)} \neg Comp(body(r))\right) \rightarrow \left(\bigwedge_{A \in L} \neg A\right). \end{aligned}$$

  ☞ The loop formula of L enforces all atoms in L to be *false* whenever L is not externally supported.

- The external bodies of L for Π are

$$\begin{aligned} EB(L) &= \{body(r) \mid r \in \Pi, head(r) \in L, body^+(r) \cap L = \emptyset\} \\ &= \{body(r) \mid r \in ES_\Pi(L)\}. \end{aligned}$$

Nogoods from logic programs

loop nogoods

For a logic program Π and some $\emptyset \subset U \subseteq atom(\Pi)$,
define the **loop nogood** of an atom $p \in U$ as

$$\lambda(p, U) \quad = \quad \{\mathbf{T}p, \mathbf{F}\beta_1, \dots, \mathbf{F}\beta_k\}$$

where $EB(U) = \{\beta_1, \dots, \beta_k\}$.

In all, we get the following set of loop nogoods for Π:

$$\Lambda_\Pi \quad = \quad \bigcup_{\emptyset \subset U \subseteq atom(\Pi)} \{\lambda(p, U) \mid p \in U\}$$

☞ The set Λ_Π of loop nogoods denies cyclic support among *true* atoms.

Example

Consider

$$\Pi = \left\{ \begin{array}{ll} x \leftarrow \mathit{not}\ y & u \leftarrow x \\ & \mathbf{u} \leftarrow \mathbf{v} \\ y \leftarrow \mathit{not}\ x & \mathbf{v} \leftarrow \mathbf{u}, y \end{array} \right\}$$

For u in the set $\{u, v\}$, we obtain the loop nogood:

$$\lambda(u, \{u, v\}) \quad = \quad \{\mathbf{T}u, \mathbf{F}\{x\}\}$$

Similarly for v in $\{u, v\}$, we get:

$$\lambda(v, \{u, v\}) \quad = \quad \{\mathbf{T}v, \mathbf{F}\{x\}\}$$

Characterization of answer sets

For a logic program Π,
let Δ_Π and Λ_Π as defined on Page 306 and Page 310, respectively.

Theorem

Let Π be a logic program. Then,
$X \subseteq atom(\Pi)$ *is an answer set of* Π **iff**
$X = A^{\mathbf{T}} \cap atom(\Pi)$ *for a (unique) solution A for* $\Delta_\Pi \cup \Lambda_\Pi$.

Some remarks

- Nogoods in Λ_Π augment Δ_Π with conditions checking for unfounded sets, in particular, those being loops.
- While $|\Delta_\Pi|$ is linear in the size of Π, Λ_Π may contain exponentially many (non-redundant) loop nogoods !

Overview

Conflict-driven search

Boolean constraint solving algorithms pioneered for SAT led to:

Traditional approach

- (Unit) propagation
- Exhaustive (chronological) backtracking

☞ DPLL [20, 19]

State of the art

- (Unit) propagation
- Conflict analysis (via resolution)
- Learning + Backjumping + Assertion

☞ CDCL [83, 67]

Idea

➥ Apply CDCL-style search in ASP solving !

Outline of CDNL-ASP algorithm

[38]

- Keep track of deterministic consequences by unit propagation on:
  - Clark's completion $[\Delta_\Pi]$
  - Loop nogoods, determined and recorded on demand $[\Lambda_\Pi]$
    - ☞ Dedicated unfounded set detection !
  - Dynamic nogoods, derived from conflicts and unfounded sets $[\nabla]$
- When a nogood in $\Delta_\Pi \cup \nabla$ becomes violated:
  - Analyze the conflict by resolution until reaching the **First Unique Implication Point** (First-UIP) [68]
  - Learn the derived conflict nogood δ
  - Backjump to the earliest (heuristic) choice such that the complement of the First-UIP is unit-resulting for δ
  - Assert the complement of the First-UIP and proceed (by unit propagation)
- Terminate when either:
  - Finding an answer set (a solution for $\Delta_\Pi \cup \Lambda_\Pi$)
  - Deriving a conflict independently of (heuristic) choices

Algorithm 1: CDNL-ASP

Input : A logic program Π.
Output : An answer set of Π or "no answer set".

$A \leftarrow \emptyset$ // *assignment over* $atom(\Pi) \cup body(\Pi)$
$\nabla \leftarrow \emptyset$ // *set of (dynamic) nogoods*
$dl \leftarrow 0$ // *decision level*
loop
  $(A, \nabla) \leftarrow \textsc{NogoodPropagation}(\Pi, \nabla, A)$
  if $\varepsilon \subseteq A$ for some $\varepsilon \in \Delta_\Pi \cup \nabla$ **then**
    if $dl = 0$ **then return** no answer set
    $(\delta, k) \leftarrow \textsc{ConflictAnalysis}(\varepsilon, \Pi, \nabla, A)$
    $\nabla \leftarrow \nabla \cup \{\delta\}$ // *learning*
    $A \leftarrow (A \setminus \{\sigma \in A \mid k < dl(\sigma)\})$ // *backjumping*
    $dl \leftarrow k$
  else if $A^{\mathbf{T}} \cup A^{\mathbf{F}} = atom(\Pi) \cup body(\Pi)$ **then**
    return $A^{\mathbf{T}} \cap atom(\Pi)$ // *answer set*
  else
    $\sigma_d \leftarrow \textsc{Select}(\Pi, \nabla, A)$ // *heuristic choice of* $\sigma_d \notin A$
    $dl \leftarrow dl + 1$
    $A \leftarrow A \circ (\sigma_d)$ // $dl(\sigma_d) = dl$

Observations

- Decision level *dl*, initially set to 0, is used to count the number of heuristically chosen literals in assignment A.
- For a heuristically chosen literal $\sigma_d = \mathbf{T}p$ or $\sigma_d = \mathbf{F}p$, respectively, we require $p \in (atom(\Pi) \cup body(\Pi)) \setminus (A^{\mathbf{T}} \cup A^{\mathbf{F}})$.
- For any literal $\sigma \in A$, $dl(\sigma)$ denotes the decision level of σ, viz. the value *dl* had when σ was assigned.
- A conflict is detected from violation of a nogood $\varepsilon \subseteq \Delta_\Pi \cup \nabla$.
- A conflict at decision level 0 (where A contains no heuristically chosen literals) indicates non-existence of answer sets.
- A nogood δ derived by conflict analysis is **asserting**, that is, some literal is unit-resulting for δ at a decision level $k < dl$.
  - ➥ After learning δ and backjumping to decision level k, at least one literal is newly derivable by unit propagation.
  - ☞ No explicit flipping of heuristically chosen literals !

Example: CDNL-ASP

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow \textit{not } y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \textit{not } x, \textit{not } y \\ y \leftarrow \textit{not } x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{\textit{not } x, \textit{not } y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\textit{not } x, \textit{not } y\}\} = \delta(w)$	
3	$\mathbf{F}\{\textit{not } y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\textit{not } y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	
		$\vdots$	$\vdots$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example (ctd): CDNL-ASP

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow \mathit{not}\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \mathit{not}\ x, \mathit{not}\ y \\ y \leftarrow \mathit{not}\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ
1	$\mathbf{T}u$		
		$\mathbf{T}x$	$\{\mathbf{T}u, \mathbf{F}x\} \in \nabla$
		$\vdots$	$\vdots$
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{x\}\} \in \Delta(v)$
		$\mathbf{F}y$	$\{\mathbf{T}y, \mathbf{F}\{\mathit{not}\ x\}\} = \delta(y)$
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\mathit{not}\ x, \mathit{not}\ y\}\} = \delta(w)$

Outline of NogoodPropagation

- Derive deterministic consequences via:
  - Unit propagation on Δ_Π and ∇;
  - Unfounded sets $U \subseteq atom(\Pi)$.
- Note that U is **unfounded** if $EB(U) \subseteq A^{\mathbf{F}}$.
  - ☞ For any $p \in U$, we have $(\lambda(p, U) \setminus \{\mathbf{T}p\}) \subseteq A$.
- An "interesting" unfounded set U satisfies:

$$\emptyset \subset U \subseteq (atom(\Pi) \setminus A^{\mathbf{F}}) \ .$$

- Wrt a fixpoint of unit propagation,
  such an unfounded set contains some loop of Π.
  - ➥ Tight programs do not yield "interesting" unfounded sets !
- Given an unfounded set U and some $p \in U$, adding $\lambda(p, U)$ to ∇ triggers a conflict or further derivations by unit propagation.
  - ☞ Add loop nogoods atom by atom to eventually falsify all $p \in U$.

Algorithm 2: NOGOODPROPAGATION

Input : A logic program Π, a set ∇ of nogoods, and an assignment A.
Output : An extended assignment and set of nogoods.

$U \leftarrow \emptyset$ *// set of unfounded atoms*
loop
  repeat
    if $\delta \subseteq A$ for some $\delta \in \Delta_\Pi \cup \nabla$ **then return** (A, ∇) *// conflict*
    $\Sigma \leftarrow \{\delta \in \Delta_\Pi \cup \nabla \mid (\delta \setminus A) = \{\sigma\}, \overline{\sigma} \notin A\}$ *// unit-resulting nogoods*
    if $\Sigma \neq \emptyset$ **then**
      let $\sigma \in (\delta \setminus A)$ for some $\delta \in \Sigma$ **in**
        $A \leftarrow A \circ (\overline{\sigma})$ *// $dl(\overline{\sigma}) = max(\{dl(\rho) \mid \rho \in (\delta \setminus \{\sigma\})\} \cup \{0\})$*
  until $\Sigma = \emptyset$
  if Π is tight **then return** (A, ∇) *// no unfounded set $\emptyset \subset U \subseteq (atom(\Pi) \setminus A^{\mathbf{F}})$*
  else
    $U \leftarrow (U \setminus A^{\mathbf{F}})$
    if $U = \emptyset$ **then** $U \leftarrow$ UNFOUNDEDSET(Π, A)
    if $U = \emptyset$ **then return** (A, ∇) *// no unfounded set $\emptyset \subset U \subseteq (atom(\Pi) \setminus A^{\mathbf{F}})$*
    let $p \in U$ **in**
      $\nabla \leftarrow \nabla \cup \{\lambda(p, U)\}$ *// record unit-resulting or violated loop nogood*

Requirements for $\textsc{UnfoundedSet}$

- Implementations of $\textsc{UnfoundedSet}$ must guarantee the following for a result U:
  1. $U \subseteq (atom(\Pi) \setminus A^{\mathbf{F}})$;
  2. $EB(U) \subseteq A^{\mathbf{F}}$;
  3. $U = \emptyset$ iff there is no nonempty unfounded subset of $(atom(\Pi) \setminus A^{\mathbf{F}})$.
- Beyond that, there are various alternatives, such as:
  - Calculating the greatest unfounded set.
  - Calculating unfounded sets within strongly connected components of the positive atom dependency graph of Π.

  ☞ Usually, the latter option is implemented in ASP solvers !

Example: NogoodPropagation

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow not\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow not\ x, not\ y \\ y \leftarrow not\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}u$			
2	$\mathbf{F}\{not\ x, not\ y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$	
3	$\mathbf{F}\{not\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	
		$\mathbf{T}\{not\ x\}$	$\{\mathbf{F}\{not\ x\}, \mathbf{F}x\} = \delta(\{not\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{not\ y\}, \mathbf{F}y\} = \delta(\{not\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Outline of ConflictAnalysis

- Conflict analysis is triggered whenever some nogood $\delta \in \Delta_\Pi \cup \nabla$ becomes violated, viz. $\delta \subseteq A$, at a decision level $dl > 0$.

☞ Note that all but the first literal assigned at dl have been unit-resulting for nogoods $\varepsilon \in \Delta_\Pi \cup \nabla$.

  ➥ If $\sigma \in \delta$ has been unit-resulting for ε, we obtain a new violated nogood by resolving δ and ε as follows:

$$(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\}) .$$

- Resolution is directed by resolving first over the literal $\sigma \in \delta$ derived last, viz. $(\delta \setminus A[\sigma]) = \{\sigma\}$.

  ☞ Iterated resolution progresses in inverse order of assignment.

- Iterated resolution stops as soon as it generates a nogood δ containing exactly one literal σ assigned at decision level dl.
  - This literal σ is called **First Unique Implication Point** (First-UIP).

  ☞ All literals in $(\delta \setminus \{\sigma\})$ are assigned at decision levels smaller than dl.

Algorithm 3: ConflictAnalysis

Input : A violated nogood δ, a logic program Π, a set ∇ of nogoods, and an assignment A.

Output : A derived nogood and a decision level.

loop
  let $\sigma \in \delta$ such that $(\delta \setminus A[\sigma]) = \{\sigma\}$ **in**
    $k \leftarrow max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\})$
    if $k = dl(\sigma)$ **then**
      let $\varepsilon \in \Delta_\Pi \cup \nabla$ such that $(\varepsilon \setminus A[\sigma]) = \{\overline{\sigma}\}$ **in**
        $\delta \leftarrow (\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$ *// resolution*
    else return (δ, k)

Example: ConflictAnalysis

Consider

$$\Pi = \left\{ \begin{array}{llll} x \leftarrow \mathit{not}\ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \mathit{not}\ x, \mathit{not}\ y \\ y \leftarrow \mathit{not}\ x & u \leftarrow v & v \leftarrow u, y & \end{array} \right\}$$

dl	σ_d	$\overline{\sigma}$	δ	
1	$\mathbf{T}\mathbf{u}$			
2	$\mathbf{F}\{\mathit{not}\ x, \mathit{not}\ y\}$			
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{\mathit{not}\ x, \mathit{not}\ y\}\} = \delta(w)$	
3	$\mathbf{F}\{\mathit{not}\ y\}$			
		$\mathbf{F}\mathbf{x}$	$\{\mathbf{T}x, \mathbf{F}\{\mathit{not}\ y\}\} = \delta(x)$	
		$\mathbf{F}\{\mathbf{x}\}$	$\{\mathbf{T}\{\mathbf{x}\}, \mathbf{F}\mathbf{x}\} \in \Delta(\{x\})$	$\{\mathbf{T}\mathbf{u}, \mathbf{F}\mathbf{x}\}$
		$\mathbf{F}\{\mathbf{x}, \mathbf{y}\}$	$\{\mathbf{T}\{\mathbf{x}, \mathbf{y}\}, \mathbf{F}\mathbf{x}\} \in \Delta(\{x, y\})$	$\{\mathbf{T}\mathbf{u}, \mathbf{F}\mathbf{x}, \mathbf{F}\{\mathbf{x}\}\}$
		$\mathbf{T}\{\mathit{not}\ x\}$	$\{\mathbf{F}\{\mathit{not}\ x\}, \mathbf{F}x\} = \delta(\{\mathit{not}\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\mathit{not}\ y\}, \mathbf{F}y\} = \delta(\{\mathit{not}\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}\mathbf{u}, \mathbf{F}\{\mathbf{x}\}, \mathbf{F}\{\mathbf{x}, \mathbf{y}\}\} = \lambda(u, \{u, v\})$	✘

Remarks

- There always is a First-UIP at which conflict analysis terminates.

☞ In the worst, resolution stops at the heuristically chosen literal assigned at decision level *dl*.

- The nogood δ containing First-UIP σ is violated by A, viz. $\delta \subseteq A$.
- We have $k = max(\{dl(\rho) \mid \rho \in \delta \setminus \{\sigma\}\} \cup \{0\}) < dl$.
  - ➥ After recording δ in ∇ and backjumping to decision level k, $\overline{\sigma}$ is unit-resulting for δ !
  - ☞ Such a nogood δ is called **asserting**.

☞ Asserting nogoods direct conflict-driven search into a different region of the search space than traversed before,
without explicitly flipping any heuristically chosen literal !

Overview

The clasp system

[40]

- Native ASP solver combining conflict-driven search with sophisticated reasoning techniques:
  - Advanced preprocessing including, e.g., equivalence reasoning
  - Lookback-based decision heuristics
  - Restart policies
  - Nogood deletion
  - Progress saving
  - Dedicated data structures for binary and ternary nogoods
  - Lazy data structures (watched literals) for long nogoods
  - Dedicated data structures for cardinality and weight constraints
  - Lazy unfounded set checking based on "source pointers"
  - Tight integration of unit propagation and unfounded set checking
  - **Reasoning modes**
  - ...

☞ Many of these techniques are configurable !

Reasoning modes of `clasp`

Beyond deciding answer set existence, `clasp` allows for:

- Optimization
- Enumeration [without solution recording]
- Projective Enumeration [without solution recording]
- Brave and Cautious Reasoning determining the
  - union or
  - intersection

  of all answer sets by computing only linearly many of them

☞ Reasoning applicable wrt answer sets as well as supported models

Front-ends also admit `clasp` to solve:

- Propositional CNF formulas
- Pseudo-Boolean formulas

Find `clasp` at: `http://potassco.sourceforge.net`

Grounding: Overview

Overview

Non-Ground

$q(a, b).$
$q(b, a).$
$q(a, c).$
$p(X, Y) \leftarrow q(X, Y), q(Y, Z).$

Ground

$q(a, b).\quad q(b, a).\quad q(a, c).$

$p(a, a) \leftarrow q(a, a), q(a, a).$	$p(b, a) \leftarrow q(b, a), q(a, a).$	$p(c, a) \leftarrow q(c, a), q(a, a).$
$p(a, a) \leftarrow q(a, a), q(a, b).$	$\mathbf{p}(\mathbf{b}, \mathbf{a}) \leftarrow \mathbf{q}(\mathbf{b}, \mathbf{a}), \mathbf{q}(\mathbf{a}, \mathbf{b}).$	$p(c, a) \leftarrow q(c, a), q(a, b).$
$p(a, a) \leftarrow q(a, a), q(a, c).$	$\mathbf{p}(\mathbf{b}, \mathbf{a}) \leftarrow \mathbf{q}(\mathbf{b}, \mathbf{a}), \mathbf{q}(\mathbf{a}, \mathbf{c}).$	$p(c, a) \leftarrow q(c, a), q(a, c).$
$\mathbf{p}(\mathbf{a}, \mathbf{b}) \leftarrow \mathbf{q}(\mathbf{a}, \mathbf{b}), \mathbf{q}(\mathbf{b}, \mathbf{a}).$	$p(b, b) \leftarrow q(b, b), q(b, a).$	$p(c, b) \leftarrow q(c, b), q(b, a).$
$p(a, b) \leftarrow q(a, b), q(b, b).$	$p(b, b) \leftarrow q(b, b), q(b, b).$	$p(c, b) \leftarrow q(c, b), q(b, b).$
$p(a, b) \leftarrow q(a, b), q(b, c).$	$p(b, b) \leftarrow q(b, b), q(b, c).$	$p(c, b) \leftarrow q(c, b), q(b, c).$
$p(a, c) \leftarrow q(a, c), q(c, a).$	$p(b, c) \leftarrow q(b, c), q(c, a).$	$p(c, c) \leftarrow q(c, c), q(c, a).$
$p(a, c) \leftarrow q(a, c), q(c, b).$	$p(b, c) \leftarrow q(b, c), q(c, b).$	$p(c, c) \leftarrow q(c, c), q(c, b).$
$p(a, c) \leftarrow q(a, c), q(c, c).$	$p(b, c) \leftarrow q(b, c), q(c, c).$	$p(c, c) \leftarrow q(c, c), q(c, c).$

☞ Only a small part of the program is relevant !

Non-Ground

$q(a).$
$q(f(a)).$
$p(X) \leftarrow q(X).$

Ground

$q(a).$
$q(f(a)).$
$\mathbf{p}(\mathbf{a}) \leftarrow \mathbf{q}(\mathbf{a}).$
$\mathbf{p}(\mathbf{f}(\mathbf{a})) \leftarrow \mathbf{q}(\mathbf{f}(\mathbf{a})).$
$p(f(f(a))) \leftarrow q(f(f(a))).$
$p(f(f(f(a)))) \leftarrow q(f(f(f(a)))).$
$\dots$

- ☞ With functions of non-zero arity, the grounding is **infinite** !
- ☞ Given a logic program Π, we are interested in a **subset** Π' of *ground*(Π) s.t. the answer sets of Π' and *ground*(Π) coincide.

Non-Ground

$q(f(a)).$
$p(X) \leftarrow not\ q(X).$

Ground

$q(f(a)).$
$p(a) \leftarrow not\ q(a).$
$p(f(a)) \leftarrow not\ q(f(a)).$
$p(f(f(a))) \leftarrow not\ q(f(f(a))).$
$\dots$

- ☞ All (but one) rules are relevant !
- ☞ The answer set is **infinite** !
- ☞ For practical reasons, such programs should be rejected.

Goals

- **First Part**: What classes of programs yield finite equivalent ground programs?
- **Second Part**: How to efficiently instantiate a program?

Terminology I

- **Variables**: $X, Y, Z, \ldots$
- **Functions**: $a/0, f/1, g/2, \ldots$ (associated with arities)
- **Predicates**: $p/0, q/1, r/2, \ldots$ (associated with arities)
- **Terms**: variables or $f(t_1, \ldots, t_n)$ s.t. each t_i is a term and f/n is a function
- **Atoms**: $p(t_1, \ldots, t_n)$ s.t. each t_i is a term and p/n is a predicate
  - An atom **binds** all variables that occur in it.
- **Literals**: an atom or an atom preceded by *not*
- **Ground terms (atoms, literals)**: terms (atoms, literals) without variables

Terminology II

- **Signature** σ: a pair of functions and predicates
- **Herbrand universe** U_σ: the set of all ground terms over functions in σ
- **Herbrand base** B_σ: the set of all ground atoms over predicates and functions in σ

Example

Given the signature $\sigma = (\{a/0, f/1\}, \{p/1\})$:

- $U_\sigma = \{a, f(a), f(f(a)), f(f(f(a))), \dots\}$
- $B_\sigma = \{p(a), p(f(a)), p(f(f(a))), p(f(f(f(a)))), \dots\}$

In the following, signature σ is often implicitly given by functions and predicates occurring in a logic program.

Terminology III

Let Π be a logic program with signature σ.

- Ground instances of $r \in \Pi$: Set of variable-free rules obtained by replacing all variables in r by elements from U_σ:

  $$ground(r) = \{r\theta \mid \theta : \mathrm{vars}(r) \rightarrow U_\sigma\}$$

  where

  - $\mathrm{vars}(r)$ stands for the set of all variables occurring in r and
  - θ is a (ground) substitution.

- **Ground instantiation** of Π:

  $$ground(\Pi) = \bigcup_{r \in \Pi} ground(r)$$

- A set $X \subseteq B_\sigma$ is an **answer set** of Π if $Cn(ground(\Pi)^X) = X$.

Overview

ω-Restricted Programs

Definition

Given a logic program Π:

1. A predicate p/n is a **domain predicate** if there is a level mapping from predicates to integers s.t., for each rule where p/n occurs in the head, all predicates in the body are domain predicates s.t. their levels are strictly smaller than that of p/n.
2. Π is ω-**restricted** if, for each rule, every variable occurring in the rule is bound by some atom $p(t_1, \dots, t_n)$ in the positive body s.t. p/n is a domain predicate.

☞ Every ω-restricted program has a finite equivalent ground program.

- Implementation lparse

Example

Example

$\mathbf{d^0}(\mathbf{a}).\;\; \mathbf{d^0}(\mathbf{b}).\;\; \mathbf{g^0}(\mathbf{b}).$

$\mathbf{r^1}(\mathbf{X}) \leftarrow \mathbf{d^0}(\mathbf{X}), \mathit{not}\ \mathbf{g^0}(\mathbf{X}).$

$p^1(X) \leftarrow q^2(X), \mathbf{d^0}(X).$

$q^2(X) \leftarrow p^1(X), \mathbf{r^1}(X).$

Level mapping

$d/1 \rightarrow 0$

$g/1 \rightarrow 0$

$r/1 \rightarrow 1$

$p/1 \rightarrow 1$

$q/1 \rightarrow 2$

- ☞ Domain predicates: $d/1, g/1, r/1$.
- ☞ The program is ω-restricted.

λ-Restricted Programs

Definition

A logic program is λ-**restricted** if there is a level mapping from predicates to integers s.t., for each rule, every variable occurring in the rule is bound by some atom in the positive body whose predicate has a strictly smaller level than the head predicate(s).

- ☞ Every λ-restricted program has a finite equivalent ground program.
- ☞ Every ω-restricted program is also λ-restricted.

- Implementation gringo (below version 3.0.0)

Example

Example

$d^0(a).\quad d^0(b).\quad g^0(b).$

$p^1(X) \leftarrow q^2(X), \mathbf{d^0}(X).$

$q^2(X) \leftarrow \mathbf{p^1}(X).$

$r^3 \leftarrow \mathbf{q^2}(X), \mathit{not}\ g^0(X), \mathit{not}\ r^3.$

Level mapping

$d/1 \rightarrow 0$

$g/1 \rightarrow 0$

$p/1 \rightarrow 1$

$q/1 \rightarrow 2$

$r/0 \rightarrow 3$

- ☞ The program is λ-restricted.
- ☞ The program is **not** ω-restricted.

Safe Programs

Definition

A logic program is **safe** if, for each rule, every variable occurring in the rule is bound by some atom in the positive body.

- ☞ Every safe program **(without functions of non-zero arity)** has a finite equivalent ground program.
- ☞ Every λ-restricted program is also safe.

- Implementation dlv & gringo (from version 3.0.0)

Example

Example I

$d(a).\quad d(b).\quad g(b).$

$p(X) \leftarrow \mathbf{q}(\mathbf{X}).$

$q(X) \leftarrow \mathbf{p}(\mathbf{X}).$

$r \leftarrow \mathbf{q}(\mathbf{X}), \mathit{not}\ g(X), \mathit{not}\ r.$

- ☞ The program is safe.
- ☞ The program is not λ-restricted.

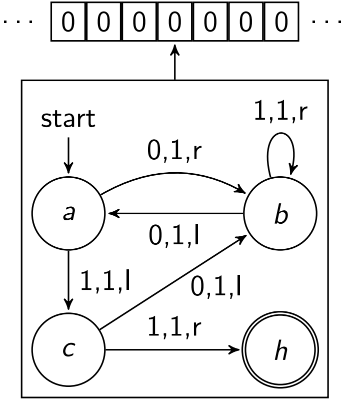
Example II

$p(a).$

$p(f(X)) \leftarrow p(X).$

- The grounding is infinite !

Encoding a 3-State Busy Beaver Machine




```
$ cat beaver.lp
start(a).
blank(0).
tape(n,0,n).

trans(a,0,1,b,r).
trans(a,1,1,c,l).
trans(b,0,1,a,l).
trans(b,1,1,b,r).
trans(c,0,1,b,l).
trans(c,1,1,h,r).
```

Encoding a Universal Turing Machine

```
$ cat turing.lp
conf(S,L,A,R) :- start(S), tape(L,A,R).

conf(SN,l(L,AN),AR,R) :- conf(S,L,A,r(AR,R)),
                         trans(S,A,AN,SN,r).
conf(SN,l(L,AN),AR,n) :- conf(S,L,A,n), blank(AR),
                         trans(S,A,AN,SN,r).
conf(SN,L,AL,r(AN,R)) :- conf(S,l(L,AL),A,R),
                         trans(S,A,AN,SN,l).
conf(SN,n,AL,r(AN,R)) :- conf(S,n,A,R), blank(AL),
                         trans(S,A,AN,SN,l).
```

Running the Turing Machine

```
$ gringo -t beaver.lp turing.lp
...
conf(a,n,0,n).
conf(b,l(n,1),0,n).
conf(a,n,1,r(1,n)).
...
conf(a,l(l(l(l(n,1),1),1),1),1,r(1,n)).
conf(c,l(l(l(n,1),1),1),1,r(1,r(1,n))).
conf(h,l(l(l(l(n,1),1),1),1),1,r(1,n)).
```

- Halts if Turing machine halts
- Finiteness check for safe programs is undecidable

Overview

Definition

Given a set P of atoms, a signature σ, and a **domain** $D \subseteq B_\sigma$:

$$\mathrm{inst}(P, D) = \{\theta : \mathrm{vars}(P) \to U_\sigma \mid A\theta \in D \text{ for all } A \in P\}$$

Algorithm: $\mathrm{instantiate}_\omega(\Pi)$

Input : An ω-restricted program Π with level mapping λ
Output : A ground program G

```
X ← set of predicates occurring in Π
D ← ∅
G ← ∅
while X ≠ ∅ do
    remove a predicate p/n with smallest level λ(p/n) from X
    foreach rule r ∈ Π with p/n in the head do
        P ← {A ∈ body⁺(r) | the predicate of A is a domain predicate}
        foreach θ ∈ inst(P, D) do
            D ← D ∪ {head(r)θ}
            G ← G ∪ {rθ}
```

Instantiating λ-Restricted Programs

Algorithm: instantiate$_\lambda(\Pi)$

Input : A λ-restricted program Π with level mapping λ
Output : A ground program G

$X \leftarrow$ set of predicates occurring in Π
$D \leftarrow \emptyset$
$G \leftarrow \emptyset$
while $X \neq \emptyset$ **do**
    remove a predicate p/n with smallest level $\lambda(p/n)$ from X
    foreach *rule* $r \in \Pi$ *with* p/n *in the head* **do**
        $P \leftarrow \{A \in \mathrm{body}^+(r) \mid \lambda(p/n)$ is greater than the level of the predicate of $A\}$
        foreach $\theta \in \mathrm{inst}(P, D)$ **do**
            $D \leftarrow D \cup \{\mathrm{head}(r)\theta\}$
            $G \leftarrow G \cup \{r\theta\}$

- **More** predicates to instantiate with !
- Possibly **smaller** grounding.

Example

$d^0(a).\;\; d^0(b).\;\; g^0(b).\;\;\; p^1(X) \leftarrow q^2(X), d^0(X).$

$q^2(X) \leftarrow p^1(X).\qquad\qquad r^3 \leftarrow q^2(X), not\ g^0(X), not\ r^3.$

p	P	$\mathrm{inst}(P, D)$	D	G
$d/1$	$\emptyset$	$\{\emptyset\}$	$\cup\{d(a)\}$	$\cup\{d(a).\}$
	$\emptyset$	$\{\emptyset\}$	$\cup\{d(b)\}$	$\cup\{d(b).\}$
$g/1$	$\emptyset$	$\{\emptyset\}$	$\cup\{g(b)\}$	$\cup\{g(b).\}$
$p/1$	$\{d(X)\}$	$\{\{X \to a\},$	$\cup\{p(a)\}$	$\cup\{p(a) \leftarrow q(a), d(a).\}$
		$\{X \to b\}\}$	$\cup\{p(b)\}$	$\cup\{p(b) \leftarrow q(b), d(b).\}$
$q/1$	$\{p(X)\}$	$\{\{X \to a\},$	$\cup\{q(a)\}$	$\cup\{q(a) \leftarrow p(a).\}$
		$\{X \to b\}\}$	$\cup\{q(b)\}$	$\cup\{q(b) \leftarrow p(b).\}$
$r/0$	$\{q(X)\}$	$\{\{X \to a\},$	$\cup\{r\}$	$\cup\{r \leftarrow q(a), not\ g(a), not\ r.\}$
		$\{X \to b\}\}$	$\cup\{r\}$	$\cup\{r \leftarrow q(b), not\ g(b), not\ r.\}$

Instantiating Safe Programs

Algorithm: $\text{instantiate}_{\text{safe}}(\Pi)$

Input : A safe program Π
Output : A ground program G

$D \leftarrow \emptyset$
$G \leftarrow \emptyset$
repeat
  $D' \leftarrow D$
  foreach $r \in \Pi$ **do**
    $P \leftarrow \text{body}^{+}(r)$
    foreach $\theta \in \text{inst}(P, D)$ **do**
      $D \leftarrow D \cup \{\text{head}(r)\theta\}$
      $G \leftarrow G \cup \{r\theta\}$
until $D = D'$

- ☞ Possibly generates **fewer** rules than $\text{instantiate}_{\omega}$ and $\text{instantiate}_{\lambda}$.
- ☞ Real implementations have to carefully **avoid regrounding** rules (semi-naive evaluation).

Example

$p(a, b).$

$p(b, c).$

$p(c, d).$

$p(X, Z) \leftarrow p(X, Y), p(Y, Z).$

$\mathrm{inst}(P, D)$	D	G
$\{\emptyset\}$	$\cup\, \{p(a, b)\}$	$\cup\, \{p(a, b).\}$
$\{\emptyset\}$	$\cup\, \{p(b, c)\}$	$\cup\, \{p(b, c).\}$
$\{\emptyset\}$	$\cup\, \{p(c, d)\}$	$\cup\, \{p(c, d).\}$
$\{\{X \to a, Y \to b, Z \to c\},$	$\cup\, \{p(a, c)\}$	$\cup\, \{p(a, c) \leftarrow p(a, b), p(b, c).\}$
$\{X \to b, Y \to c, Z \to d\}\}$	$\cup\, \{p(b, d)\}$	$\cup\, \{p(b, d) \leftarrow p(b, c), p(c, d).\}$
$\{\{X \to a, Y \to c, Z \to d\},$	$\cup\, \{p(a, d)\}$	$\cup\, \{p(a, d) \leftarrow p(a, c), p(c, d).\}$
$\{X \to a, Y \to b, Z \to d\}\}$	$\cup\, \{p(a, d)\}$	$\cup\, \{p(a, d) \leftarrow p(a, b), p(b, d).\}$
Fixpoint		

Optimizations

- Remove facts from rule bodies:
  1. $r(c)$. has already been found
  2. $p(a) \leftarrow q(b), r(c)$. is found

  ☞ Simplify ground rule to $p(a) \leftarrow q(b)$.
- Skip rules that contain false literals:
  1. $r(c)$. has already been found
  2. $p(a) \leftarrow q(b), \mathit{not}\ r(c)$. is found

  ☞ Skip the ground rule.

☞ Allows for finitely grounding larger class of programs:

- Consider $\Pi = \{\, p(a). \quad q(f(f(a))). \quad p(f(X)) \leftarrow p(X), \mathit{not}\ q(X).\}$

  ☞ $\text{instantiate}_{\text{safe}}(\Pi)$ will terminate !

Overview

Predicate-Rule Dependency Graph

Definition

Let Π be a logic program.

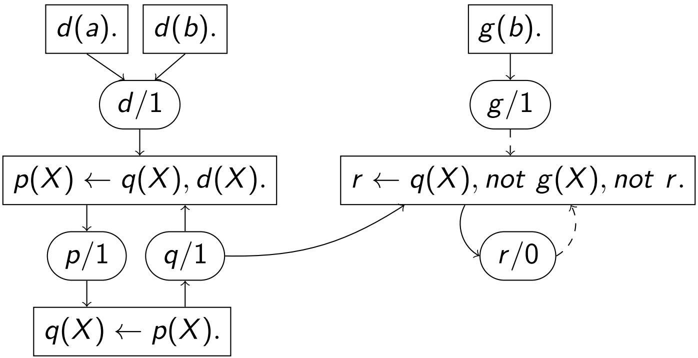
1. The **predicate-rule dependency graph** $G_\Pi = (V, E)$ of Π is a directed graph s.t.:
   - V is the set of predicates and rules of Π
   - $(p/n, r) \in E$ if predicate p/n occurs in the body of rule r
   - $(r, p/n) \in E$ if predicate p/n occurs in the head of rule r
2. $(p/n, r) \in E$ is **negative** if predicate p/n occurs in the negative body of rule r

☞ More fine-grained static program analysis.

Example

$d(a).\quad d(b).\quad g(b).\quad p(X) \leftarrow q(X), d(X).$

$q(X) \leftarrow p(X).\qquad r \leftarrow q(X), \mathit{not}\ g(X), \mathit{not}\ r.$



Strongly Connected Components I

A graph is **strongly connected** if all vertices pairwise reach each other via some path.

Definition

Let $G = (V, E)$ be a graph.

1. A set $C \subseteq V$ of vertices belonging to a maximal strongly connected subgraph of G is called a **strongly connected component (SCC)** of G.
2. An SCC A **depends** on an SCC B if $(B \times A) \cap E \neq \emptyset$.

- ☞ Dependencies among SCCs are acyclic.
- ☞ The SCCs of a predicate-rule dependency graph can be used to partition a logic program.

Strongly Connected Components II

Definition

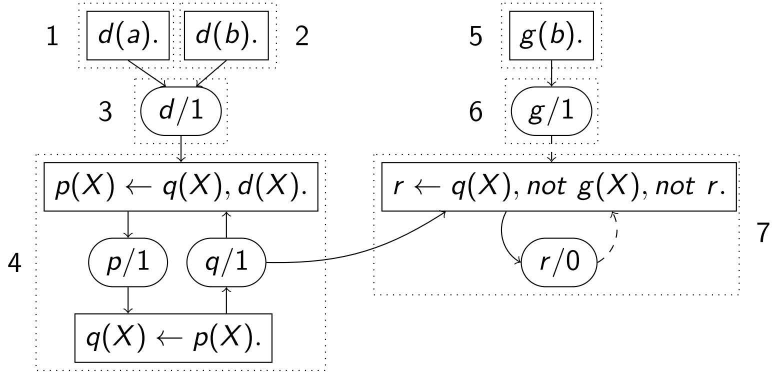
Given a logic program Π, an SCC of G_Π is

- **normal** if it contains a negative edge or depends on a normal SCC,
- **basic** if it is not normal and contains at least one edge,
- **fact** otherwise.

☞ A program is λ-restricted if its components are λ-restricted.

☞ Basic and fact components do not involve "choices".

☞ SCCs can be grounded in topological order.



$$\begin{array}{rl}
\text{fact} & \{d(a).\}\ , \{d(b).\}\ , \{g(b).\}\ , \{d/1\}\ , \{g/1\} \\
\text{basic} & \{p(X) \leftarrow q(X), d(X).\ , q(X) \leftarrow p(X).\ , p/1\ , q/1\} \\
\text{normal} & \{r \leftarrow q(X), not\ g(X), not\ r.\ , r/0\}
\end{array}$$

A topological order

Overview

The Backtracking Instantiator

Definition

Given a signature σ, a substitution θ, an atom A, and a domain D:

$$\mathsf{match}(\theta, A, D) = \{\theta \cup \theta' \mid \theta' : \mathsf{vars}(A\theta) \rightarrow U_\sigma, (A\theta)\theta' \in D\}$$

Algorithm: $\mathsf{instantiate}_{\mathsf{bt}}(\theta, P)$

Input : A substitution θ and a list P of atoms
Output : Set of (ground) substitutions
Global : Domain D

```
if P = [] then return {θ}
else
    S ← ∅
    foreach θ' ∈ match(θ, first(P), D) do
        S ← S ∪ instantiate_bt(θ', tail(P))
    return S
```

Example

Example

$P = [p(X, Y), q(Y, Z), r(Z)]$

$$
\begin{tikzcd}
\mathbf{p(a,b)} \arrow[r] \arrow[dr] & \mathbf{q(b,c)} \arrow[r] & \mathbf{r(c)} \\
\mathbf{p(b,a)} \arrow[dr] & \mathbf{q(b,a)} & \\
& \mathbf{q(a,c)} \arrow[uur] &
\end{tikzcd}
$$

- $S = \{\{X \to a, Y \to b, Z \to c\}, \{X \to b, Y \to a, Z \to c\}\}$

The Backjumping Instantiator

Algorithm: $\text{instantiate}_{\text{bj}}(\theta, P)$

Input : A substitution θ and a list P of atoms
Output : Set of (ground) substitutions and variables to bind
Global : Output variables O and domain D

if $P = []$ **then** **return** $(\{\theta\}, O)$
else
    $A \leftarrow \text{first}(P)$
    $M \leftarrow \text{match}(\theta, A, D)$
    if $M = \emptyset$ **then**
        return $(\emptyset, \text{vars}(A))$
    else
        $S \leftarrow \emptyset$
        $B \leftarrow \emptyset$
        foreach $\theta' \in M$ **do**
            $(S, B) \leftarrow (S, B) \sqcup \text{instantiate}_{\text{bj}}(\theta', \text{tail}(P))$
            if $\text{vars}(A\theta) \cap B = \emptyset$ **then** **return** (S, B)
        return $(S, B \cup \text{vars}(A))$

Advanced Modeling: Overview

Overview

Motivation

- Many problems can nicely be encoded using ASP
  - There are often many degrees of freedom to encode a problem
  - Even worse, different encodings may lead to drastically different solving times
  - ☞ We will try to find some hints on how to efficiently encode problems using ASP
- Some problems can, due to increased complexity, no longer be (polynomially) represented using normal logic programs
  - ☞ We will take a look on how disjunctive rules can be used to overcome this situation

Solving a Problem Using ASP

- My (Roland) steps to solve a problem using ASP
  1. Create a **small test instance**
  2. Come up with a **quick solution**
  3. Debug this solution using the test instance
     - Use **ASPViz** or write some small scripts
  4. Switch to larger instances
  5. **Analyze the flaws** of the quick solution
     - Size of the grounding
     - Time needed to solve the problem
  6. **Incrementally refine** the solution
     - The quick solution serves as cross-check
  7. Throw away everything and try something different

☞ Basically it is a **Trial and Error** process

Overview

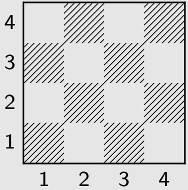
N-Queens Problem

Problem Specification

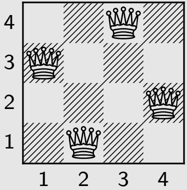
Given an $N \times N$ chessboard,
place N queens such that they do not attack each other.

$N = 4$

Chessboard



Placement



A First Encoding

1. Each square may host a queen.
2. No row, column, or diagonal hosts two queens.
3. A placement is given by instances of `queen` in an answer set.
4. We have to place (at least) N queens.

queens_0.lp

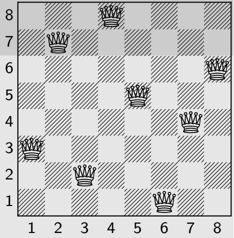
Anything missing?

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X1,Y1), queen(X1,Y2), Y1 < Y2.
:- not n #count{ queen(X,Y) }.

% DISPLAY
#hide. #show queen/2.
```

A First Encoding

Let's Place 8 Queens!

```
gringo -c n=8 queens_0.lp | clasp --stats
```




```
Answer: 1
queen(1,6) queen(2,3) queen(3,1) queen(4,7)
queen(5,5) queen(6,8) queen(7,2) queen(8,4)
SATISFIABLE

Models      : 1+
Time        : 0.006s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
Choices     : 18
Conflicts   : 13
Restarts    : 0

Variables   : 793
Constraints : 729
```

A First Encoding

Let's Place 22 Queens!

```
gringo -c n=22 queens_0.lp | clasp --stats
```

```
Answer: 1
queen(1,10) queen(2,6) queen(3,16) queen(4,14) queen(5,8) ...
SATISFIABLE

Models      : 1+
Time        : 150.531s (Solving: 150.37s 1st Model: 150.34s Unsat: 0.00s)
CPU Time    : 147.480s
Choices     : 594960
Conflicts   : 574565
Restarts    : 19

Variables   : 17271
Constraints : 16787
```

A First Refinement

At least N queens? Exactly one queen per row and column!

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- queen(X1,Y1), queen(X1,Y2), Y1 < Y2.
:- queen(X1,Y1), queen(X2,Y1), X1 < X2.
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.
:- not n #count{ queen(X,Y) }.

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

Let's Place 22 Queens!

```
gringo -c n=22 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,18) queen(2,10) queen(3,21) queen(4,3) queen(5,5) ...
SATISFIABLE

Models      : 1+
Time        : 0.113s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.020s
Choices     : 132
Conflicts   : 105
Restarts    : 1

Variables   : 7238
Constraints : 6710
```

A First Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_1.lp | clasp --stats
```

```
Answer: 1
queen(1,24) queen(2,52) queen(3,37) queen(4,60) queen(5,76) ...
SATISFIABLE

Models      : 1+
Time        : 79.475s (Solving: 1.06s 1st Model: 1.06s Unsat: 0.00s)
CPU Time    : 6.930s
Choices     : 1373
Conflicts   : 845
Restarts    : 4

Variables   : 1211338
Constraints : 1196210
```

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | clasp --stats
```

```
1241358 7402724 24950848

real  1m15.468s
user  1m15.980s
sys   0m0.090s
```

Just kidding :-)

☞ Grounding makes the problem!

A First Refinement

Grounding Time $\sim$ Space

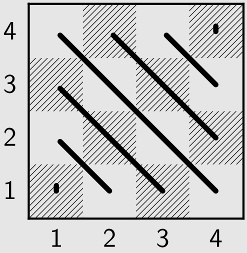
queens_1.lp

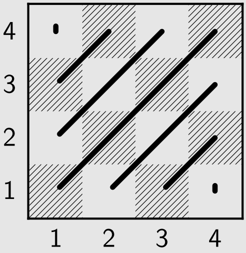
```
% DOMAIN
#const n=4. square(1..n,1..n).                                   O(n×n)

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).                        O(n×n)

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.                      O(n×n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.   O(n^2×n^2)

% DISPLAY
#hide. #show queen/2.
```

Diagonals make trouble!

A Nomenclature for Diagonals

$N = 4$





$\#\text{diagonal}_1 =$
$(\#\text{row} + \#\text{column}) - 1$

$\#\text{diagonal}_2 =$
$(\#\text{row} - \#\text{column}) + N$

☞ $\#\text{diagonal}_{1/2}$ can be determined in this way for arbitrary N.

A Second Refinement

Let's go for Diagonals!

queens_2.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.


% DISPLAY
#hide. #show queen/2.
```

A Second Refinement

Let's Place 122 Queens!

```
gringo -c n=122 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,98) queen(2,54) queen(3,89) queen(4,83) queen(5,59) ...
SATISFIABLE

Models       : 1+
Time         : 2.211s (Solving: 0.13s 1st Model: 0.13s Unsat: 0.00s)
CPU Time     : 0.210s
Choices      : 11036
Conflicts    : 499
Restarts     : 3

Variables    : 16098
Constraints  : 970
```

A Second Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_2.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models      : 1+
Time        : 35.450s (Solving: 6.69s 1st Model: 6.68s Unsat: 0.00s)
CPU Time    : 7.250s
Choices     : 141445
Conflicts   : 7488
Restarts    : 9

Variables   : 92994
Constraints : 2394
```

A Third Refinement

Let's Precompute Diagonals!

queens_3.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).
diag1(X,Y,(X+Y)-1) :- square(X,Y). diag2(X,Y,(X-Y)+n) :- square(X,Y).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X+Y)-1 }. % Diagonal 1
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X-Y)+n }. % Diagonal 2

% DISPLAY
#hide. #show queen/2.
```

A Third Refinement

Let's Place 300 Queens!

```
gringo -c n=300 queens_3.lp | clasp --stats
```

```
Answer: 1
queen(1,62) queen(2,232) queen(3,176) queen(4,241) queen(5,207) ...
SATISFIABLE

Models      : 1+
Time        : 8.889s (Solving: 6.61s 1st Model: 6.60s Unsat: 0.00s)
CPU Time    : 7.320s
Choices     : 141445
Conflicts   : 7488
Restarts    : 9

Variables   : 92994
Constraints : 2394
```

A Third Refinement

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
```

```
Answer: 1
queen(1,477) queen(2,365) queen(3,455) queen(4,470) queen(5,237) ...
SATISFIABLE

Models       : 1+
Time         : 76.798s (Solving: 65.81s 1st Model: 65.75s Unsat: 0.00s)
CPU Time     : 68.620s
Choices      : 869379
Conflicts    : 25746
Restarts     : 12

Variables    : 365994
Constraints  : 4794
```

A Case for Oracles

Let's Place 600 Queens!

```
gringo -c n=600 queens_3.lp | clasp --stats
--heuristic=vsids --trans-ext=dynamic
```

```
Answer: 1
queen(1,477) queen(2,365) queen(3,455) queen(4,470) queen(5,237) ...
SATISFIABLE

Models      : 1+
Time        : 76.798s (Solving: 65.81s 1st Model: 65.75s Unsat: 0.00s)
CPU Time    : 68.620s
Choices     : 869379
Conflicts   : 25746
Restarts    : 12

Variables   : 365994
Constraints : 4794
```

Overview

Implementing Universal Quantification

Goal: identify objects such that ALL properties from a "list" hold

1. check all properties explicitly . . . obsolete if properties change ✘
2. use variable-sized conjunction (via ':') . . . adapts to changing facts ✔
3. use negation of complement . . . adapts to changing facts ✔

Example: vegetables to buy

```
veg(asparagus).         veg(cucumber).
pro(asparagus,cheap).   pro(cucumber,cheap).
pro(asparagus,fresh).   pro(cucumber,fresh).
pro(asparagus,tasty).   pro(cucumber,tasty).


buy(X) :- veg(X), pro(X,P) : pre(P).
```

Running Example

Latin Square

Problem Specification

Fill an $N \times N$ grid with numbers 1 to N such that each number occurs in every row and column.

$N = 4$

Grid

				1
				2
				3
				4
1	2	3	4	

Placement

1	2	3	4	1
4	1	2	3	2
3	4	1	2	3
2	3	4	1	4
1	2	3	4	

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).



% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- square(X1,Y1), N = 1..n, not num(X1,Y2,N) : square(X1,Y2).
:- square(X1,Y1), N = 1..n, not num(X2,Y1,N) : square(X2,Y1).
```

☞ unreused "singleton variables"

```
gringo latin_0.lp | wc
```

105480 2558984 14005258

```
gringo latin_1.lp | wc
```

42056 273672 1690522

Unraveling Symmetric Inequalities

Another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- num(X1,Y1,N), num(X1,Y2,N), Y1 != Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 != X2.
```

☞ duplicate ground rules (swapping `Y1`/`Y2` and `X1`/`X2` gives the "same")

```
gringo latin_2.lp | wc
```

```
2071560 12389384 40906946
```

```
gringo latin_3.lp | wc
```

```
1055752 6294536 21099558
```

Linearizing Existence Tests

Still another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
gtX(X-1,Y,N) :- num(X,Y,N), 1 < X.     gtY(X,Y-1,N) :- num(X,Y,N), 1 < Y.
gtX(X-1,Y,N) :- gtX(X,Y,N), 1 < X.     gtY(X,Y-1,N) :- gtY(X,Y,N), 1 < Y.
 :- num(X,Y,N), gtX(X,Y,N).             :- num(X,Y,N), gtY(X,Y,N).
```

☞ uniqueness of `N` in a row/column checked by ENUMERATING PAIRS!

```
gringo latin_3.lp | wc
```

1055752 6294536 21099558

```
gringo latin_4.lp | wc
```

228360 1205256 4780744

Assigning Aggregate Values

Yet another Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
sigma(S) :- S = #sum[ square(X,n) = X ].                              ✔

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C #count{ num(X,Y,N) } C, C = 0..n. ✗
occY(Y,N,C) :- Y = 1..n, N = 1..n, C #count{ num(X,Y,N) } C, C = 0..n. ✗
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3. #show sigma/1.
```

☞ internal transformation by `gringo`

Breaking Symmetries

The ultimate Latin square encoding?

```
% DOMAIN
#const n=32. square(1..n,1..n).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- X = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.
:- Y = 1..n, N = 1..n, not 1 #count{ num(X,Y,N) } 1.



% DISPLAY
#hide. #show num/3.
```

```
gringo -c n=5 latin_7.lp | clasp -q 0
```

```
Models : 161280     Time : 2.078s
```

Encode With Care!

1. Create a working encoding
   - Q1: Do you need to modify the encoding if the facts change?
   - Q2: Are all variables significant (or statically functionally dependent)?
   - Q3: Can there be (many) identic ground rules?
   - Q4: Do you enumerate pairs of values (to test uniqueness)?
   - Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   - Q6: Do you admit (obvious) symmetric solutions?
   - Q7: Do you have additional domain knowledge simplifying the problem?
   - Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?
2. Revise until no "Yes" answer!
   - ☞ If the format of facts makes encoding painful (for instance, abusing grounding for "scientific calculations"), revise the fact format as well.

Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic ... follow error messages by the grounder
- semantic ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

1. develop and test incrementally
   - prepare toy instances with "interesting features"
   - build the encoding bottom-up and verify additions (eg. new predicates)
2. compare the encoded to the intended meaning
   - check whether the grounding fits (use `gringo -t`)
   - if answer sets are unintended, investigate conditions that fail to hold
   - if answer sets are missing, examine integrity constraints (add heads)
3. ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

Overcoming Performance Bottlenecks

Grounding

- monitor time spent by and output size of `gringo`
  1. system tools (eg. `time(gringo [...] | wc)`)
  2. profiling info (eg. `gringo --gstats --verbose=3 [...] > /dev/null`)

☞ once identified, reformulate "critical" logic program parts

Solving

- check solving statistics (use `clasp --stats`)

☞ if great search efforts (`Conflicts`/`Choices`/`Restarts`), then
  1. try auto-configuration (offered by `claspfolio`)
  2. try manual fine-tuning (requires expert knowledge!)
  3. if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

Overview

Hitori

A Japanese Grid Puzzle (Beyond Sudoku)

The Puzzle

Given: an $N \times N$ board of numbered squares

4	8	1	6	3	2	5	7
3	6	7	2	1	6	5	4
2	3	4	8	2	8	6	1
4	1	6	5	7	7	3	5
7	2	3	1	8	5	1	2
3	5	6	7	3	1	8	4
6	4	2	3	5	4	7	8
8	7	1	4	2	3	5	6

Wanted: a set of black squares such that

1. no black squares are horizontally or vertically adjacent
2. numbers of white squares are unique for each row and column
3. every pair of white squares is connected via a path (not passing black squares)

	8		6	3	2		7
3	6	7	2	1		5	4
	3	4		2	8	6	1
4	1		5	7		3	
7		3		8	5	1	2
	5	6	7		1	8	
6		2	3	5	4	7	8
8	7	1	4		3		6

Fact and Solution Format

Facts provide instances of `state(X,Y,N)` to express that the square in column `X` and row `Y` contains number `N`.

Example Instance

```
state(1,1,4). state(2,1,8). ... state(8,1,7).
state(1,2,3). state(2,2,6). ... state(8,2,4).
state(1,3,2). state(2,3,3). ... state(8,3,1).
state(1,4,4). state(2,4,1). ... state(8,4,5).
state(1,5,7). state(2,5,2). ... state(8,5,2).
state(1,6,3). state(2,6,5). ... state(8,6,4).
state(1,7,6). state(2,7,4). ... state(8,7,8).
state(1,8,8). state(2,8,7). ... state(8,8,6).
```

4	8	1	6	3	2	5	7
3	6	7	2	1	6	5	4
2	3	4	8	2	8	6	1
4	1	6	5	7	7	3	5
7	2	3	1	8	5	1	2
3	5	6	7	3	1	8	4
6	4	2	3	5	4	7	8
							6

Example Solution

Black squares given by instances of `blackOut(X,Y)`:

```
blackOut(1,1)   blackOut(2,5)   ...
blackOut(1,3)   ...   blackOut(8,4)
blackOut(1,6)   ...   blackOut(8,6)
```

	8		6	3	2		7
3	6	7	2	1		5	4
	3	4		2	8	6	1
4	1		5	7		3	
7		3		8	5	1	2
	5	6	7		1	8	
6		2	3	5	4	7	8
8	7	1	4		3		6

A Working Encoding I

Found on the WWW (and Adapted to gringo Syntax)

hitori_0.lp (under GNU GPL: COPYING)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% (A) Adjacent grid locations %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Domain predicate (evaluated upon grounding)
adjacent(X,Y,X+1,Y) :- state(X,Y,_), state(X+1,Y,_).
adjacent(X,Y,X,Y+1) :- state(X,Y,_), state(X,Y+1,_).
adjacent(X2,Y2,X1,Y1) :- adjacent(X1,Y1,X2,Y2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% (B) Generate solution candidate %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Every square is blacked out or normal
1 { blackOut(X,Y), -blackOut(X,Y) } 1 :- state(X,Y,_).
```

A Working Encoding II

Found on the WWW (and Adapted to gringo Syntax)

hitori_0.lp

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% (C.1) Test eliminating adjacent blanks %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Can't have adjacent black squares
 :- adjacent(X1,Y1,X2,Y2), blackOut(X1,Y1), blackOut(X2,Y2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% (C.2) Tests eliminating number recurrences %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Can't have the same number twice in the same row
 :- state(X1,Y,N), state(X2,Y,N), -blackOut(X1,Y), -blackOut(X2,Y), X1 != X2.

% Can't have the same number twice in the same column
 :- state(X,Y1,N), state(X,Y2,N), -blackOut(X,Y1), -blackOut(X,Y2), Y1 != Y2.
```

Already spot something?

A Working Encoding III

Found on the WWW (and Adapted to gringo Syntax)

hitori_0.lp

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% (C.3) Test eliminating disconnected numbers %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Define mutual reachability
reachable(X1,Y1,X2,Y2) :- adjacent(X1,Y1,X2,Y2), -blackOut(X1,Y1),
     -blackOut(X2,Y2).
reachable(X1,Y1,X3,Y3) :- reachable(X1,Y1,X2,Y2), reachable(X2,Y2,X3,Y3).

% Can't have mutually unreachable non-black squares
 :- -blackOut(X1,Y1), -blackOut(X2,Y2), not reachable(X1,Y1,X2,Y2),
     (X1,Y1) != (X2,Y2).
```

☞ Answer sets (of hitori_0.lp plus instance) match Hitori solutions. ✔

A Working Encoding

Let's Run it!

```
gringo hitori_0.lp instance.lp | clasp --stats
```

```
Answer: 1
blackOut(1,1) blackOut(1,3) blackOut(1,6) blackOut(2,5)
blackOut(2,7) ...           blackOut(8,4) blackOut(8,6)
SATISFIABLE

Models      : 1+
Time        : 13.485s (Solving: 11.77s 1st Model: 11.77s Unsat: 0.00s)
CPU Time    : 13.290s
Choices     : 458
Conflicts   : 323
Restarts    : 2

Variables   : 260625
Constraints : 1018953
```

4	8	1	6	3	2	5	7
3	6	7	2	1	6	5	4
2	3	4	8	2	8	6	1
4	1	6	5	7	7	3	5
7	2	3	1	8	5	1	2
3	5	6	7	3	1	8	4
6	4	2	3	5	4	7	8
8	7	1	4	2	3	5	6

Why Classical Negation?

hitori_0.lp

```
% Every square is blacked out or normal
1 { blackOut(X,Y), -blackOut(X,Y) } 1 :- state(X,Y,_).

:- blackOut(X,Y), -blackOut(X,Y).
```

☞ internal transformation by gringo

```
gringo hitori_0.lp instance.lp | wc
```

```
267534 1608172 5535208
```

```
gringo hitori_1.lp instance.lp | wc
```

```
267470 1607788 5534184
```

☞ no noticeable effect on grounding/solving performance

Why Not Default Negation?

hitori_1.lp

```
% Every square is blacked out or normal
1 { blackOut(X,Y), negBlackOut(X,Y) } 1 :- state(X,Y,_).

% Can't have the same number twice in the same row
 :- state(X1,Y,N), state(X2,Y,N), negBlackOut(X1,Y), negBlackOut(X2,Y), X1 != X2.
 ...
```

☞ replace `negBlackOut(X,Y)` by "`not blackOut(X,Y)`"

A First Improvement

```
gringo hitori_2.lp instance.lp | clasp --stats
```

```
Answer: 1
blackOut(1,1) blackOut(1,3) blackOut(1,6) blackOut(2,5)
blackOut(2,7) ...           blackOut(8,4) blackOut(8,6)
SATISFIABLE

Models      : 1+
Time        : 13.485s (Solving: 11.77s 1st Model: 11.77s Unsat: 0.00s)
CPU Time    : 13.290s
Choices     : 458
Conflicts   : 323
Restarts    : 2

Variables   : 260625
Constraints : 1018953
```

Remember Symmetric Inequalities

hitori_2.lp

```
% Can't have the same number twice in the same row
 :- state(X1,Y,N), state(X2,Y,N), not blackOut(X1,Y), not blackOut(X2,Y), X1 < X2.

% Can't have the same number twice in the same column
 :- state(X,Y1,N), state(X,Y2,N), not blackOut(X,Y1), not blackOut(X,Y2), Y1 < Y2.
```

☞ no noticeable effect on grounding/solving performance

Let's Use Counting

hitori_3.lp

```
% Can't have the same number twice in the same row or column
 :- state(X1,Y1,N), 2 { not blackOut(X1,Y2) : state(X1,Y2,N) }.
 :- state(X1,Y1,N), 2 { not blackOut(X2,Y1) : state(X2,Y1,N) }.
```

A Second Improvement?

```
gringo hitori_4.lp instance.lp | clasp --stats
```

```
Answer: 1
blackOut(1,1) blackOut(1,3) blackOut(1,6) blackOut(2,5)
blackOut(2,7) ...           blackOut(8,4) blackOut(8,6)
SATISFIABLE

Models      : 1+
Time        : 10.182s (Solving: 8.47s 1st Model: 8.47s Unsat: 0.00s)
CPU Time    : 10.010s
Choices     : 344
Conflicts   : 264
Restarts    : 2

Variables   : 260433
Constraints : 1018825
```

Why Double-Check Reachability?

hitori_5.lp

```
% Define mutual reachability
reachable(X1,Y1,X2,Y2) :- adjacent(X1,Y1,X2,Y2), (X1,Y1) < (X2,Y2),
                          not blackOut(X1,Y1), not blackOut(X2,Y2).
reachable(X1,Y1,X3,Y3) :- reachable(X1,Y1,X2,Y2), reachable(X2,Y2,X3,Y3).
reachable(X1,Y1,X3,Y3) :- reachable(X1,Y1,X2,Y2), reachable(X3,Y3,X2,Y2),
                          (X1,Y1) < (X3,Y3).
reachable(X2,Y2,X3,Y3) :- reachable(X1,Y1,X2,Y2), reachable(X1,Y1,X3,Y3),
                          (X2,Y2) < (X3,Y3).

% Can't have mutually unreachable non-black squares
 :- not blackOut(X1,Y1), not blackOut(X2,Y2), not reachable(X1,Y1,X2,Y2),
    (X1,Y1) <  (X2,Y2),  state(X1,Y1,_), state(X2,Y2,_).
```

☞ enforce `(X1,Y1) < (X2,Y2)` for instances of `reachable(X1,Y1,X2,Y2)`

A Real Breakthrough?

```
gringo hitori_5.lp instance.lp | clasp --stats
```

```
Answer: 1
blackOut(1,1) blackOut(1,3) blackOut(1,6) blackOut(2,5)
blackOut(2,7) ...           blackOut(8,4) blackOut(8,6)
SATISFIABLE

Models      : 1+
Time        : 9.781s (Solving: 7.99s 1st Model: 7.99s Unsat: 0.00s)
CPU Time    : 9.610s
Choices     : 278
Conflicts   : 227
Restarts    : 1

Variables   : 260432
Constraints : 1018828
```

Two Orders of Magnitude!

hitori_6.lp

```
% Define mutual reachability
reachable(X1,Y1,X2,Y2) :- adjacent(X1,Y1,X2,Y2), (X1,Y1) < (X2,Y2),
                          not blackOut(X1,Y1), not blackOut(X2,Y2).

reachable(X1,Y1,X3,Y3) :- reachable(X1,Y1,X2,Y2), adjacent(X2,Y2,X3,Y3),
                          (X1,Y1) < (X3,Y3), not blackOut(X3,Y3).
reachable(X2,Y2,X3,Y3) :- reachable(X1,Y1,X2,Y2), adjacent(X1,Y1,X3,Y3),
                          (X2,Y2) < (X3,Y3), not blackOut(X3,Y3).
```

☞ grounding size: $O(8^6)$

A First Breakthrough

```
gringo hitori_6.lp instance.lp | clasp --stats
```

```
Answer: 1
blackOut(1,1) blackOut(1,3) blackOut(1,6) blackOut(2,5)
blackOut(2,7) ...           blackOut(8,4) blackOut(8,6)
SATISFIABLE

Models      : 1+
Time        : 4.054s (Solving: 3.07s 1st Model: 3.07s Unsat: 0.00s)
CPU Time    : 3.810s
Choices     : 438
Conflicts   : 318
Restarts    : 2

Variables   : 129328
Constraints : 504573
```

Let's Think a Bit More

hitori_7.lp

```
reachable(1,1).
reachable(X2,Y2) :- reachable(X1,Y1), adjacent(X1,Y1,X2,Y2), not blackOut(X2,Y2).




% Can't have unreachable non-black square
 :- state(X,Y,_), not blackOut(X,Y), not reachable(X,Y).
```

Q: How many squares adjacent to `(1,1)` can possibly be black?

A: At most one!

	8		6	3	2		7
3	6	7	2	1		5	4
	3	4		2	8	6	1
4	1		5	7		3	
7		3		8	5	1	2
	5	6	7		1	8	
6		2	3	5	4	7	8
8	7	1	4		3		6

Not That Much Left to Save

```
gringo hitori_7.lp instance.lp | clasp --stats
```

```
Answer: 1
blackOut(1,1) blackOut(1,3) blackOut(1,6) blackOut(2,5)
blackOut(2,7) ...           blackOut(8,4) blackOut(8,6)
SATISFIABLE

Models      : 1+
Time        : 0.093s (Solving: 0.01s 1st Model: 0.01s Unsat: 0.00s)
CPU Time    : 0.040s
Choices     : 64
Conflicts   : 23
Restarts    : 0

Variables   : 11231
Constraints : 32234
```

Let's Reach All Squares (Anyway)

hitori_8.lp

```
% Define reachability
reachable(1,1). reachable(1,2).
reachable(X2,Y2) :- reachable(X1,Y1), adjacent(X1,Y1,X2,Y2), not blackOut(X1,Y1).

% Can't have unreachable non-black square
 :- state(X,Y,_), not blackOut(X,Y), not reachable(X,Y).
```

☞ require all white squares to be reached

The Final Result

```
gringo hitori_8.lp instance.lp | clasp --stats
```

```
Answer: 1
blackOut(1,1) blackOut(1,3) blackOut(1,6) blackOut(2,5)
blackOut(2,7) ...           blackOut(8,4) blackOut(8,6)
SATISFIABLE

Models      : 1+
Time        : 0.009s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
Choices     : 77
Conflicts   : 25
Restarts    : 0

Variables   : 539
Constraints : 1137
```

The Final Encoding (Pretty-Printed) I

hitori_9.lp

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% (A) Adjacent grid locations %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Domain predicate (evaluated upon grounding)
adjacent(X,Y,X+1,Y) :- state(X,Y,_;;X+1,Y,_).
adjacent(X,Y,X,Y+1) :- state(X,Y,_;;X,Y+1,_).
adjacent(X2,Y2,X1,Y1) :- adjacent(X1,Y1,X2,Y2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% (B) Generate solution candidate %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Every square is blacked out or normal
{ blackOut(X,Y) } :- state(X,Y,_).
```

The Final Encoding (Pretty-Printed) II

hitori_9.lp

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% (C.1) Test eliminating adjacent blanks %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Can’t have adjacent black squares
 :- adjacent(X1,Y1,X2,Y2), blackOut(X1,Y1;;X2,Y2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% (C.2) Tests eliminating number recurrences %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Can’t have the same number twice in the same row or column
 :- state(X1,Y1,N), 2 { not blackOut(X1,Y2) : state(X1,Y2,N) }.
 :- state(X1,Y1,N), 2 { not blackOut(X2,Y1) : state(X2,Y1,N) }.
```

The Final Encoding (Pretty-Printed) III

hitori_9.lp

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% (C.3) Test eliminating disconnected numbers %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Define reachability
reachable(1,1).
reachable(1,2).
reachable(X2,Y2) :- reachable(X1,Y1), adjacent(X1,Y1,X2,Y2),
                    not blackOut(X1,Y1).

% Can't have unreachable square
 :- state(X,Y,_), not reachable(X,Y).
```

Recall Where We Started

```
gringo hitori_0.lp instance.lp | clasp --stats
```

```
Answer: 1
blackOut(1,1) blackOut(1,3) blackOut(1,6) blackOut(2,5)
blackOut(2,7) ...           blackOut(8,4) blackOut(8,6)
SATISFIABLE

Models      : 1+
Time        : 13.485s (Solving: 11.77s 1st Model: 11.77s Unsat: 0.00s)
CPU Time    : 13.290s
Choices     : 458
Conflicts   : 323
Restarts    : 2

Variables   : 260625
Constraints : 1018953
```

And Where We Came

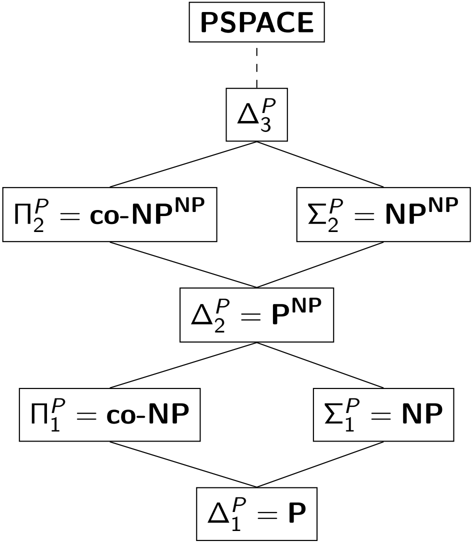
```
gringo hitori_9.lp instance.lp | clasp --stats
```

```
Answer: 1
blackOut(1,1) blackOut(1,3) blackOut(1,6) blackOut(2,5)
blackOut(2,7) ...           blackOut(8,4) blackOut(8,6)
SATISFIABLE

Models      : 1+
Time        : 0.006s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
Choices     : 16
Conflicts   : 5
Restarts    : 0

Variables   : 317
Constraints : 315
```

The encoding matters!

Overview

The Polynomial Time Hierarchy



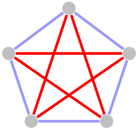
The $\mathbf{NP}^{\mathbf{NP}}$ Class

- What is an $\mathbf{NP}^{\mathbf{NP}}$ problem?
  - ☞ A problem decidable in non-deterministic polynomial time using a (second) **NP** oracle
- How does this relate to disjunctive logic programs?
  1. Guess an answer set candidate for a given disjunctive program
  2. Query the **NP** oracle to verify the guess

The Ramsey Problem

Theorem

For two numbers r and b, there exists a least number $R(r, b) = n$ s.t. every complete graph with n vertices and edges colored either red or blue contains a complete subgraph (clique) on r vertices whose edges are all red, or a complete subgraph on b vertices whose edges are all blue.



- Contains neither a red nor a blue clique of size 3

☞ Shows that $R(3, 3) > 5$

- We will model the problem accordingly
  1. Guess a total edge labeling (ASP as usual)
  2. Verify that the labeling does not admit a clique of size 3 (disjunctive **co-NP** tests)

  ☞ Satisfiability if $n < R(r, b)$ is not yet a Ramsey number

The Plan

1. Choose some total edge labeling for a complete graph of size `n`
2. Disjunctive tests to verify that the labeling does not admit a clique
   1. Guess subgraphs supposed to form (mono-colored) cliques
   2. For each color, derive a special atom if the subgraph is not a clique
   3. Derive everything if such a special atom holds
   4. Since any answer set is a minimal model of its reduct, some subgraph that is a clique will be chosen whenever possible
      - ☞ A special atom will only be derived if there is no clique
   5. We may not use default negation/anti-monotone aggregates in the disjunctive part
      - ☞ Default negation/anti-monotone aggregates are removed in the reduct
3. Fail whenever some special atom could not be derived
   - ☞ In case there was a clique

Modeling

- Helper predicates

```
col(red,3).  col(blue,3).
col(C) :- col(C,N).
```

- Choose a total edge labeling (usual ASP)

```
1 { col(U,V,C) : col(C) } 1 :- U = 1..n, V = (U+1)..n.
```

- Disjunctively guess a clique per color

```
in(U,C) | out(U,C) :- U = 1..n, col(C).
```

- Derive a special bot atom if the guess is invalid or not a clique

```
bot(C) :- col(C,N), (n-N)+1 { out(1..n,C) }.
bot(C) :- col(C,N), N+1 { in(1..n,C) }.
bot(C) :- in(U,C), in(V,C), U < V, not col(U,V,C).
```

- Derive everything if bot holds for a color

```
in(1..n,C)  :- bot(C).
out(1..n,C) :- bot(C).
```

- Fail if some clique has been found

```
:- col(C), not bot(C).
```

Summary

- Disjunctive programs can be used to solve problems beyond **NP**
  - We use **claspD** for some biological application problems
- Disjunctive program parts are suitable for modeling an additional **co-NP** test per answer set candidate
- It requires some practice to write such programs
  - No default negation/anti-monotone aggregates may be used in the disjunctive part
  - ☞ Instead provide "direct derivations" for conditions that do not hold
- Debugging disjunctive programs is even harder than debugging normal programs
  - ☞ Answer sets usually include all atoms from the disjunctive part

Equivalence of Logic Programs: Overview

Overview

Motivation

Questions

- How to optimize logic programs?
- How to remove redundancies in automatically generated logic programs?

Difficulty Given that ASP is nonmonotonic,
it is difficult to attribute meaning to

- program parts or
- incomplete programs

because the addition of further rules generally changes the overall semantics.

Notions of Equivalence

Two logic programs Π_1 and Π_2 are

- **equivalent** ($\Pi_1 \equiv \Pi_2$) if $AS(\Pi_1) = AS(\Pi_2)$.
- **strongly equivalent** ($\Pi_1 \equiv_s \Pi_2$) if $AS(\Pi_1 \cup \Pi') = AS(\Pi_2 \cup \Pi')$ for any logic program Π'.
- **uniformly equivalent** ($\Pi_1 \equiv_u \Pi_2$) if $AS(\Pi_1 \cup F) = AS(\Pi_2 \cup F)$ for any set F of facts.

Example $\Pi_1 = \{a \vee b \leftarrow\ \}$ and $\Pi_2 = \{a \leftarrow not\ b\ , b \leftarrow not\ a\ \}$

- $\Pi_1 \equiv \Pi_2$ since $AS(\Pi_1) = \{\{a\}, \{b\}\} = AS(\Pi_2)$
- $\Pi_1 \not\equiv_s \Pi_2$, e.g. $\Pi' = \{a \leftarrow b\ , b \leftarrow a\ \}$
- $\Pi_1 \equiv_u \Pi_2$

Implications

- strong equivalence implies uniform equivalence and
- uniform equivalence implies (ordinary) equivalence.

Overview

Ordinary Equivalence

- Consider $\Pi_1 = \{a \leftarrow b\}$ and $\Pi_2 = \{a \leftarrow c\}$.

$$\Pi_1 \equiv \Pi_2 \quad \text{but} \quad (\Pi_1 \cup \{b \leftarrow\}) \not\equiv (\Pi_2 \cup \{b \leftarrow\}).$$

- Consider $\Pi_1 = \{a \leftarrow \mathit{not}\ b\}$ and $\Pi_2 = \{a \leftarrow\}$.

$$\Pi_1 \equiv \Pi_2 \quad \text{but} \quad (\Pi_1 \cup \{b \leftarrow\}) \not\equiv (\Pi_2 \cup \{b \leftarrow\}).$$

- Ordinary equivalence in ASP does **not** allow for **substitution of equivalents**:

$$\Pi_1 \equiv \Pi_2 \quad \textbf{not} \text{ implies} \quad \Pi \equiv \Pi[\Pi_1/\Pi_2],$$

  for any logic programs Π_1, Π_2, and Π.

  ➥ The non-monotonicity of ASP makes equivalence of programs a much weaker concept than equivalence in (monotonic) classical logic.

Overview

Strong Equivalence

- Two logic programs Π_1 and Π_2 are strongly equivalent if

$$(\Pi_1 \cup \Pi') \equiv (\Pi_2 \cup \Pi') \text{ for any logic programs } \Pi \;.$$

- Strong Equivalence (SE) guarantees substitution of equivalents.
- How to test strong equivalence?
  - ➥ How to avoid testing $AS(\Pi_1 \cup \Pi') = AS(\Pi_2 \cup \Pi')$ for any logic program Π'?

SE-models

Model-theoretic characterization of Strong Equivalence.
Let Π be a logic program over alphabet $\mathcal{A}$.

- An **SE-interpretation** over $\mathcal{A}$ is a pair (X, Y) such that $X \subseteq Y \subseteq \mathcal{A}$
- An SE-interpretation (X, Y) is an **SE-model** of Π if
  1. $Y \models \Pi$
  2. $X \models \Pi^Y$
- $SE(\Pi)$ denotes the set of all SE-models of Π

Theorem $\Pi_1 \equiv_s \Pi_2$ iff $SE(\Pi_1) = SE(\Pi_2)$

Observation If (X, X) is the unique SE-model of Π whose second component is X, then X is an answer set of Π.

Example: SE-models

$\Pi_1 = \{a \vee b \leftarrow \}$ and $\Pi_2 = \{a \leftarrow \mathit{not}\ b\ , b \leftarrow \mathit{not}\ a\ \}$
We get the following SE-models over $\{a, b\}$:

$$
\begin{array}{rcl}
SE(\Pi_1) & = & \{(\{a\},\{a\}),(\{b\},\{b\}), \\
 & & \ (\{a\},\{a,b\}),(\{b\},\{a,b\}),(\{a,b\},\{a,b\})\} \\
SE(\Pi_2) & = & SE(\Pi_1) \cup \{(\emptyset,\{a,b\})\}
\end{array}
$$

We have

- $SE(\Pi_1) \neq SE(\Pi_2)$ implies $\Pi_1 \not\equiv_s \Pi_2$
- **Counterexample** Take $\Pi' = \{a \leftarrow b\ , b \leftarrow a\ \}$

Example: SE-models

$\Pi_1 = \{a \leftarrow \}$ and $\Pi_2 = \{a \leftarrow , a \leftarrow b , a \leftarrow not\ c \}$
We get the following SE-models over $\{a, b, c\}$:

$$
\begin{aligned}
SE(\Pi_1) \quad &= \quad \{(\{a\},\{a\}), (\{a\},\{a,b\}), (\{a\},\{a,c\}), \\
&\qquad (\{a\},\{a,b,c\}), (\{a,b\},\{a,b\}), (\{a,b\},\{a,b,c\}), \\
&\qquad (\{a,c\},\{a,c\}), (\{a,c\},\{a,b,c\}), (\{a,b,c\},\{a,b,c\})\,\} \\
SE(\Pi_2) \quad &= \quad SE(\Pi_1)
\end{aligned}
$$

Observation For rules r_1 and r_2, we have $\{r_1\} \equiv_s \{r_1, r_2\}$ whenever $SE(\{r_1\}) \subseteq SE(\{r_2\})$

Example

- $SE(\{r_1\}) \subseteq SE(\{r_2\})$ holds for any rules where $head(r_1) = head(r_2)$ and $body(r_1) \subseteq body(r_2)$
  - ➥ In any program, delete a rule r_2 if there is some rule r_1 such that $head(r_1) = head(r_2)$ and $body(r_1) \subseteq body(r_2)$.

Strong Equivalence

Normal versus Disjunctive logic programs

Reduct-Intersection Let Π be a normal logic program.
If $(U, Y) \in SE(\Pi)$ and $(V, Y) \in SE(\Pi)$, then $(U \cap V, Y) \in SE(\Pi)$.

(☞ Since for any X, Π^X is a Horn program.)

- Reduct-Intersection is not satisfied by disjunctive logic programs.
- If the SE-models of a disjunctive program do not satisfy reduct-intersection, then no strongly equivalent normal programs exists.

Example

- Recall program $\Pi_1 = \{a \vee b \leftarrow\}$ along with

$$\begin{aligned} SE(\Pi_1) = \{ & (\{a\},\{a\}), (\{b\},\{b\}), \\ & (\{a\},\{a,b\}), (\{b\},\{a,b\}), (\{a,b\},\{a,b\})\} \end{aligned}$$

- $SE(\Pi_1)$ is not closed under reduct-intersection, since $(\{a\},\{a,b\})$ and $(\{b\},\{a,b\})$ call for $(\emptyset,\{a,b\})$.
  - ➥ No normal logic program is strongly equivalent to $\{a \vee b \leftarrow\}$.

From SE-models to counterexamples

Let Π_1, Π_2 be (disjunctive) logic programs and $(X, Y) \in SE(\Pi_1) \setminus SE(\Pi_2)$.

1. If $(Y, Y) \in SE(\Pi_2)$,
   let $\Pi' = \{A \leftarrow \ \mid A \in X\} \cup \{A \leftarrow B \mid A, B \in Y \setminus X\}$.
   We get $X \subset Y$ and
   - $X \models (\Pi_1 \cup \Pi')^Y$,
   - $Y \models (\Pi_2 \cup \Pi')^Y$ but $Z \not\models (\Pi_2 \cup \Pi')^Y$ for any $Z \subset Y$.
     - ➥ That is, $Y \in AS(\Pi_2 \cup \Pi') \setminus AS(\Pi_1 \cup \Pi')$.
2. If $(Y, Y) \notin SE(\Pi_2)$,
   let $\Pi' = \{A \leftarrow \ \mid A \in Y\}$.
   We get
   - $Y \models (\Pi_1 \cup \Pi')^Y$ but $Z \not\models (\Pi_1 \cup \Pi')^Y$ for any $Z \subset Y$,
   - $Y \not\models (\Pi_2 \cup \Pi')^Y$.
     - ➥ That is, $Y \in AS(\Pi_1 \cup \Pi') \setminus AS(\Pi_2 \cup \Pi')$.

Overview

UE-models

Model-theoretic characterization of **Uniform Equivalence**.

Let Π be a logic program over alphabet $\mathcal{A}$.

- An SE-interpretation (X, Y) is a **UE-model** of Π if
  1. $(X, Y) \in SE(\Pi)$ and
  2. for each Z with $X \subset Z \subset Y$, we have $(Z, Y) \notin SE(\Pi)$.
- $UE(\Pi)$ denotes the set of all UE-models of Π.

Theorem $\Pi_1 \equiv_u \Pi_2$ iff $UE(\Pi_1) = UE(\Pi_2)$

Observation UE-models of a program Π are

- all SE-models (X, X) of Π,
- all further SE-models (X, Y) of Π, where $X \subset Y$ is maximal in being a model of Π^Y.

Example: UE-models

$\Pi_1 = \{a \vee b \leftarrow \}$ and $\Pi_2 = \{a \leftarrow \mathit{not}\ b\ , b \leftarrow \mathit{not}\ a\ \}$

$$\begin{aligned} UE(\Pi_1) &= SE(\Pi_1) \\ &= \{(\{a\},\{a\}), (\{b\},\{b\}), \\ &\quad\ (\{a\},\{a,b\}), (\{b\},\{a,b\}), (\{a,b\},\{a,b\})\} \end{aligned}$$

$$\begin{aligned} UE(\Pi_2) &= SE(\Pi_2) \setminus \{(\emptyset,\{a,b\})\} \\ &= SE(\Pi_1) \end{aligned}$$

We have

- $UE(\Pi_1) = UE(\Pi_2)$ implies $\Pi_1 \equiv_u \Pi_2$ and $\Pi_1 \equiv \Pi_2$ although $SE(\Pi_1) \neq SE(\Pi_2)$.
- Note that the SE-model $(\emptyset, \{a,b\})$ is no UE-model of Π_2, since $(\{a\},\{a,b\})$ is an UE-model of Π_2.

From UE-models to counterexamples

Let Π_1, Π_2 be (disjunctive) logic programs and $(X, Y) \in UE(\Pi_1) \setminus UE(\Pi_2)$.

1. If $(Y, Y) \in UE(\Pi_2)$ and $(X', Y) \in UE(\Pi_2)$ such that $X \subset X' \subset Y$, let $\Pi' = \{A \leftarrow \mid A \in X'\}$.
   We get
   - $Y \models (\Pi_1 \cup \Pi')^Y$ but $Z \not\models (\Pi_1 \cup \Pi')^Y$ for any $Z \subset Y$,
   - $X' \models (\Pi_2 \cup \Pi')^Y$.
     - ➥ That is, $Y \in AS(\Pi_1 \cup \Pi') \setminus AS(\Pi_2 \cup \Pi')$.
2. If $(Y, Y) \in UE(\Pi_2)$ and $(X', Y) \notin UE(\Pi_2)$ for any $X \subset X' \subset Y$, let $\Pi' = \{A \leftarrow \mid A \in X\}$.
   We get $X \subset Y$ and
   - $X \models (\Pi_1 \cup \Pi')^Y$,
   - $Y \models (\Pi_2 \cup \Pi')^Y$ but $Z \not\models (\Pi_2 \cup \Pi')^Y$ for any $Z \subset Y$.
     - ➥ That is, $Y \in AS(\Pi_2 \cup \Pi') \setminus AS(\Pi_1 \cup \Pi')$.
3. If $(Y, Y) \notin UE(\Pi_2)$, let $\Pi' = \{A \leftarrow \mid A \in Y\}$.
   As with SE-models, we get $Y \in AS(\Pi_1 \cup \Pi') \setminus AS(\Pi_2 \cup \Pi')$.

Overview

Program Transformations

Let be Π a (disjunctive) logic program.

- TAUT: if $head(r) \cap body^+(r) \neq \emptyset$ then $\Pi \equiv_s \Pi \setminus \{r\}$ and $\Pi \equiv_u \Pi \setminus \{r\}$,
  e.g. $\{a \leftarrow, a \leftarrow a\} \equiv_s \{a \leftarrow\}$
- RED$^-$: $r_1, r_2 \in \Pi$, $body(r_2) = \emptyset, head(r_2) \subseteq body^-(r_1)$, then $\Pi \equiv_s \Pi \setminus \{r_1\}$ and $\Pi \equiv_u \Pi \setminus \{r_1\}$,
  e.g. $\{a \leftarrow, b \leftarrow not\ a\} \equiv_s \{a \leftarrow\}$
- NONMIN: $r_1, r_2 \in \Pi, head(r_2) \subseteq head(r_1), body(r_2) \subseteq body(r_1)$, then $\Pi \equiv_s \Pi \setminus \{r_1\}$ and $\Pi \equiv_u \Pi \setminus \{r_1\}$,
  e.g. $\{a \leftarrow, a \leftarrow b\} \equiv_s \{a \leftarrow\}$
- CONTRA: $body^+(r) \cap body^-(r) \neq \emptyset$, then $\Pi \equiv_s \Pi \setminus \{r\}$ and $\Pi \equiv_u \Pi \setminus \{r\}$,
  e.g. $\{b \leftarrow a, not\ a\} \equiv_s \emptyset$

Program Transformations (ctd)

WGPPE $r_1 \in \Pi$, $a \in body^+(r_1)$,
$G_a = \{r_2 \in \Pi \mid head(r_2) = a\}, G_a \neq \emptyset$,
then $\Pi \equiv_s \Pi \cup G'_a$ and $\Pi \equiv_u \Pi \cup G'_a$ where $G'_a = \{head(r_1) \leftarrow (body^+(r_1) \setminus \{a\}) \cup not\ body^-(r_1) \cup body(r_2) \mid r_2 \in G_a\}$
e.g. $\{a \leftarrow b, c, not\ d\ , c \leftarrow e, not\ f\} \equiv_s$
$\{a \leftarrow b, c, not\ d\ , c \leftarrow e, not\ f\ , a \leftarrow b, e, not\ f, not\ d\}$

S-IMP $r_1, r_2 \in \Pi$ such that there exists an $A \subseteq body^-(r_1)$ such that $head(r_2) \subseteq head(r_1) \cup A, body^-(r_2) \subseteq body^-(r_1) \setminus A$ and $body^+(r_2) \subseteq body^+(r_1)$,
then $\Pi \equiv_s \Pi \setminus \{r_1\}$ and $\Pi \equiv_u \Pi \setminus \{r_1\}$
e.g. $\{a \leftarrow b, not\ c, not\ d\ , a \vee d \leftarrow b, not\ c\} \equiv_s$
$\{a \vee d \leftarrow b, not\ c\}$

S. Abiteboul, R. Hull, and V. Vianu.
Foundations of Databases.
Addison-Wesley, 1995.

C. Anger, M. Gebser, T. Linke, A. Neumann, and T. Schaub.
The `nomore++` approach to answer set solving.
In G. Sutcliffe and A. Voronkov, editors, *Proceedings of the Twelfth International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'05)*, volume 3835 of *Lecture Notes in Artificial Intelligence*, pages 95–109. Springer-Verlag, 2005.

C. Anger, K. Konczak, T. Linke, and T. Schaub.
A glimpse of answer set programming.
Künstliche Intelligenz, 19(1):12–17, 2005.

Y. Babovich and V. Lifschitz.
Computing answer sets using program completion.
Unpublished draft; available at
`http://www.cs.utexas.edu/users/tag/cmodels.html`, 2003.

C. Baral.
Knowledge Representation, Reasoning and Declarative Problem Solving.
Cambridge University Press, 2003.

C. Baral, G. Brewka, and J. Schlipf, editors.
Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07), volume 4483 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2007.

C. Baral and M. Gelfond.
Logic programming and knowledge representation.
Journal of Logic Programming, 12:1–80, 1994.

A. Biere.
Adaptive restart strategies for conflict driven SAT solvers.
In H. Kleine Büning and X. Zhao, editors, *Proceedings of the Eleventh International Conference on Theory and Applications of Satisfiability*

Testing (SAT'08), volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer-Verlag, 2008.

A. Biere.
PicoSAT essentials.
Journal on Satisfiability, Boolean Modeling and Computation, 4:75–97, 2008.

A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors.
Handbook of Satisfiability, volume 185 of *Frontiers in Artificial Intelligence and Applications*.
IOS Press, 2009.

M. Brain, O. Cliffe, and M. de Vos.
A pragmatic programmer's guide to answer set programming.
In M. de Vos and T. Schaub, editors, *Proceedings of the Second Workshop on Software Engineering for Answer Set Programming (SEA'09)*, Department of Computer Science, University of Bath, Technical Report Series, pages 49–63, 2009.

M. Brain, M. Gebser, J. Pührer, T. Schaub, H. Tompits, and S. Woltran.
Debugging ASP programs by means of ASP.
In Baral et al. [6], pages 31–43.

M. Brain, M. Gebser, J. Pührer, T. Schaub, H. Tompits, and S. Woltran.
That is illogical captain! — the debugging support tool spock for answer-set programs: System description.
In M. de Vos and T. Schaub, editors, *Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07)*, number CSBU-2007-05 in Department of Computer Science, University of Bath, Technical Report Series, pages 71–85, 2007.
ISSN 1740-9497.

S. Brass and J. Dix.
Semantics of (disjunctive) logic programs based on partial evaluation.
Journal of Logic Programming, 40(1):1–46, 1999.

K. Clark.

Negation as failure.
In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

O. Cliffe, M. de Vos, M. Brain, and J. Padget.
ASPVIZ: Declarative visualisation and animation using answer set programming.
In M. Garcia de la Banda and E. Pontelli, editors, *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*, volume 5366 of *Lecture Notes in Computer Science*, pages 724–728. Springer-Verlag, 2008.

M. D'Agostino, D. Gabbay, R. Hähnle, and J. Posegga, editors.
Handbook of Tableau Methods.
Kluwer Academic Publishers, 1999.

E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov.
Complexity and expressive power of logic programming.

In *Proceedings of the Twelfth Annual IEEE Conference on Computational Complexity (CCC'97)*, pages 82–101. IEEE Computer Society Press, 1997.

M. Davis, G. Logemann, and D. Loveland.
A machine program for theorem-proving.
Communications of the ACM, 5:394–397, 1962.

M. Davis and H. Putnam.
A computing procedure for quantification theory.
Journal of the ACM, 7:201–215, 1960.

C. Drescher, M. Gebser, T. Grote, B. Kaufmann, A. König, M. Ostrowski, and T. Schaub.
Conflict-driven disjunctive answer set solving.
In G. Brewka and J. Lang, editors, *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, pages 422–432. AAAI Press, 2008.

C. Drescher, M. Gebser, B. Kaufmann, and T. Schaub.
Heuristics in conflict resolution.

In M. Pagnucco and M. Thielscher, editors, *Proceedings of the Twelfth International Workshop on Nonmonotonic Reasoning (NMR'08)*, number UNSW-CSE-TR-0819 in School of Computer Science and Engineering, The University of New South Wales, Technical Report Series, pages 141–149, 2008.

N. Eén and N. Sörensson.
An extensible SAT-solver.
In E. Giunchiglia and A. Tacchella, editors, *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer-Verlag, 2004.

T. Eiter, M. Fink, H. Tompits, and S. Woltran.
Simplifying logic programs under uniform and strong equivalence.
In V. Lifschitz and I. Niemelä, editors, *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04)*, volume 2923 of *Lecture Notes in Artificial Intelligence*, pages 87–99. Springer-Verlag, 2004.

T. Eiter and G. Gottlob.
On the computational cost of disjunctive logic programming: Propositional case.
Annals of Mathematics and Artificial Intelligence, 15(3-4):289–323, 1995.

F. Fages.
Consistency of Clark's completion and the existence of stable models.
Journal of Methods of Logic in Computer Science, 1:51–60, 1994.

P. Ferraris.
Answer sets for propositional theories.
In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, volume 3662 of *Lecture Notes in Artificial Intelligence*, pages 119–131. Springer-Verlag, 2005.

M. Fitting.
A Kripke-Kleene semantics for logic programs.
Journal of Logic Programming, 2(4):295–312, 1985.

M. Gebser, C. Guziolowski, M. Ivanchev, T. Schaub, A. Siegel, S. Thiele, and P. Veber.
Repair and prediction (under inconsistency) in large biological networks with answer set programming.
In F. Lin and U. Sattler, editors, *Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR'10)*, pages 497–507. AAAI Press, 2010.

M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele.
A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`.
Available at `http://potassco.sourceforge.net`.

M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub.
On the implementation of weight constraint rules in conflict-driven ASP solvers.
In P. Hill and D. Warren, editors, *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, volume

5649 of *Lecture Notes in Computer Science*, pages 250–264. Springer-Verlag, 2009.

M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub.
Multi-criteria optimization in answer set programming.
In J. Gallagher and M. Gelfond, editors, *Technical Communications of the Twenty-seventh International Conference on Logic Programming (ICLP'11)*, volume 11, pages 1–10. Leibniz International Proceedings in Informatics (LIPIcs), 2011.

M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub.
Multi-criteria optimization in ASP and its application to Linux package configuration.
Unpublished draft, 2011.
Available at
`http://www.cs.uni-potsdam.de/wv/pdfformat/gekakasc11b.pdf`.

M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, M. Schneider, and S. Ziller.
A portfolio solver for answer set programming: Preliminary report.

In J. Delgrande and W. Faber, editors, *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, volume 6645 of *Lecture Notes in Artificial Intelligence*, pages 352–357. Springer-Verlag, 2011.

M. Gebser, R. Kaminski, and T. Schaub.
Complex optimization in answer set programming.
Theory and Practice of Logic Programming, 11(4-5):821–839, 2011.

M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
clasp: A conflict-driven answer set solver.
In Baral et al. [6], pages 260–265.

M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Conflict-driven answer set enumeration.
In Baral et al. [6], pages 136–148.

M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Conflict-driven answer set solving.
In Veloso [82], pages 386–392.

M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Advanced preprocessing for answer set solving.
In M. Ghallab, C. Spyropoulos, N. Fakotakis, and N. Avouris, editors, *Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI'08)*, pages 15–19. IOS Press, 2008.

M. Gebser, B. Kaufmann, and T. Schaub.
The conflict-driven answer set solver clasp: Progress report.
In E. Erdem, F. Lin, and T. Schaub, editors, *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *Lecture Notes in Artificial Intelligence*, pages 509–514. Springer-Verlag, 2009.

M. Gebser, B. Kaufmann, and T. Schaub.
Solution enumeration for projected Boolean search problems.
In W. van Hoeve and J. Hooker, editors, *Proceedings of the Sixth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*

(CPAIOR'09), volume 5547 of *Lecture Notes in Computer Science*, pages 71–86. Springer-Verlag, 2009.

M. Gebser, J. Pührer, T. Schaub, and H. Tompits.
A meta-programming technique for debugging answer-set programs.
In D. Fox and C. Gomes, editors, *Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI'08)*, pages 448–453. AAAI Press, 2008.

M. Gebser and T. Schaub.
Tableau calculi for answer set programming.
In S. Etalle and M. Truszczyński, editors, *Proceedings of the Twenty-second International Conference on Logic Programming (ICLP'06)*, volume 4079 of *Lecture Notes in Computer Science*, pages 11–25. Springer-Verlag, 2006.

M. Gebser and T. Schaub.
Generic tableaux for answer set programming.
In V. Dahl and I. Niemelä, editors, *Proceedings of the Twenty-third International Conference on Logic Programming (ICLP'07)*, volume

4670 of *Lecture Notes in Computer Science*, pages 119–133. Springer-Verlag, 2007.

M. Gebser, T. Schaub, and S. Thiele.
Gringo: A new grounder for answer set programming.
In Baral et al. [6], pages 266–271.

M. Gebser, T. Schaub, S. Thiele, and P. Veber.
Detecting inconsistencies in large biological networks with answer set programming.
Theory and Practice of Logic Programming, 11(2-3):323–360, 2011.

M. Gelfond.
Answer sets.
In V. Lifschitz, F. van Harmelen, and B. Porter, editors, *Handbook of Knowledge Representation*, chapter 7, pages 285–316. Elsevier Science, 2008.

M. Gelfond and N. Leone.

Logic programming and knowledge representation — the A-Prolog perspective.
Artificial Intelligence, 138(1-2):3–38, 2002.

M. Gelfond and V. Lifschitz.
The stable model semantics for logic programming.
In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*, pages 1070–1080. MIT Press, 1988.

M. Gelfond and V. Lifschitz.
Logic programs with classical negation.
In *Proceedings of the International Conference on Logic Programming*, pages 579–597, 1990.

E. Giunchiglia, Y. Lierler, and M. Maratea.
Answer set programming based on propositional satisfiability.
Journal of Automated Reasoning, 36(4):345–377, 2006.

J. Huang.

The effect of restarts on the efficiency of clause learning.
In Veloso [82], pages 2318–2323.

H. Kautz and B. Selman.
Planning as satisfiability.
In B. Neumann, editor, *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363. John Wiley & sons, 1992.

K. Konczak, T. Linke, and T. Schaub.
Graphs and colorings for answer set programming.
Theory and Practice of Logic Programming, 6(1-2):61–106, 2006.

R. Kowalski.
Logic for data description.
In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 77–103. Plenum Press, 1978.

J. Lee.
A model-theoretic counterpart of loop formulas.

In L. Kaelbling and A. Saffiotti, editors, *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 503–508. Professional Book Center, 2005.

N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello.
The DLV system for knowledge representation and reasoning.
ACM Transactions on Computational Logic, 7(3):499–562, 2006.

V. Lifschitz.
Answer set programming and plan generation.
Artificial Intelligence, 138(1-2):39–54, 2002.

V. Lifschitz.
Introduction to answer set programming.
Unpublished draft; available at
`http://www.cs.utexas.edu/users/vl/papers/esslli.ps`, 2004.

V. Lifschitz, D. Pearce, and A. Valverde.
Strongly equivalent logic programs.

ACM Transactions on Computational Logic, 2(4):526–541, 2001.

V. Lifschitz and A. Razborov.
Why are there so many loop formulas?
ACM Transactions on Computational Logic, 7(2):261–268, 2006.

V. Lifschitz, L. Tang, and H. Turner.
Nested expressions in logic programs.
Annals of Mathematics and Artificial Intelligence, 25(3-4):369–389, 1999.

F. Lin and Y. Zhao.
ASSAT: computing answer sets of a logic program by SAT solvers.
Artificial Intelligence, 157(1-2):115–137, 2004.

J. Lloyd.
Foundations of Logic Programming.
Symbolic Computation. Springer-Verlag, 2nd edition, 1987.

V. Marek and M. Truszczyński.
Nonmonotonic logic: context-dependent reasoning.

Artifical Intelligence. Springer-Verlag, 1993.

V. Marek and M. Truszczyński.
Stable models and an alternative logic programming paradigm.
In K. Apt, W. Marek, M. Truszczyński, and D. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999.

J. Marques-Silva, I. Lynce, and S. Malik.
Conflict-driven clause learning SAT solvers.
In Biere et al. [10], chapter 4, pages 131–153.

J. Marques-Silva and K. Sakallah.
GRASP: A search algorithm for propositional satisfiability.
IEEE Transactions on Computers, 48(5):506–521, 1999.

D. Mitchell.
A SAT solver primer.
Bulletin of the European Association for Theoretical Computer Science, 85:112–133, 2005.

M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik.
Chaff: Engineering an efficient SAT solver.
In *Proceedings of the Thirty-eighth Conference on Design Automation (DAC'01)*, pages 530–535. ACM Press, 2001.

I. Niemelä.
Logic programs with stable model semantics as a constraint programming paradigm.
Annals of Mathematics and Artificial Intelligence, 25(3-4):241–273, 1999.

J. Oetsch, J. Pührer, and H. Tompits.
Catching the ouroboros: On debugging non-ground answer-set programs.
In *Theory and Practice of Logic Programming. Twenty-sixth International Conference on Logic Programming (ICLP'10) Special Issue*, volume 10(4-6), pages 513–529. Cambridge University Press, 2010.

M. Osorio, J. Navarro, and J. Arrazola.

Equivalence in answer set programming.
In A. Pettorossi, editor, *Proceedings of the Eleventh International Workshop on Logic Based Program Synthesis and Transformation (LOPSTR'01)*, volume 2372 of *Lecture Notes in Computer Science*, pages 57–75. Springer-Verlag, 2001.

K. Pipatsrisawat and A. Darwiche.
A lightweight component caching scheme for satisfiability solvers.
In J. Marques-Silva and K. Sakallah, editors, *Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer-Verlag, 2007.

L. Ryan.
Efficient algorithms for clause-learning SAT solvers.
Master's thesis, Simon Fraser University, 2004.

J. Schlipf.
The expressive powers of the logic programming semantics.
Journal of Computer and System Sciences, 51:64–86, 1995.

P. Simons, I. Niemelä, and T. Soininen.
Extending and implementing the stable model semantics.
Artificial Intelligence, 138(1-2):181–234, 2002.

T. Syrjänen.
Lparse 1.0 user's manual.
http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz.

H. Turner.
Strong equivalence made easy: nested expressions and weight constraints.
Theory and Practice of Logic Programming, 3(4-5):609–622, 2003.

J. Ullman.
Principles of Database and Knowledge-Base Systems.
Computer Science Press, 1988.

A. van Gelder, K. Ross, and J. Schlipf.
The well-founded semantics for general logic programs.
Journal of the ACM, 38(3):620–650, 1991.

M. Veloso, editor.
Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07). AAAI Press/The MIT Press, 2007.

L. Zhang, C. Madigan, M. Moskewicz, and S. Malik.
Efficient conflict driven learning in a Boolean satisfiability solver.
In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285, 2001.