Answer Set Solving in Practice

Article *in* Synthesis Lectures on Artificial Intelligence and Machine Learning · December 2012

DOI: 10.2200/S00457ED1V01Y201211AIM019

Martin Gebser
Universität Potsdam

Torsten Schaub
Universität Potsdam

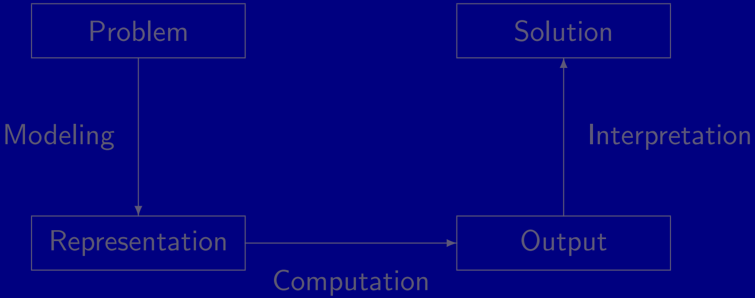
Answer Set Solving in Practice

Martin Gebser and Torsten Schaub
University of Potsdam
{gebser,torsten}@cs.uni-potsdam.de

http://www.cs.uni-potsdam.de/~torsten/ijcai11tutorial/asp.pdf

Goal: Declarative problem solving

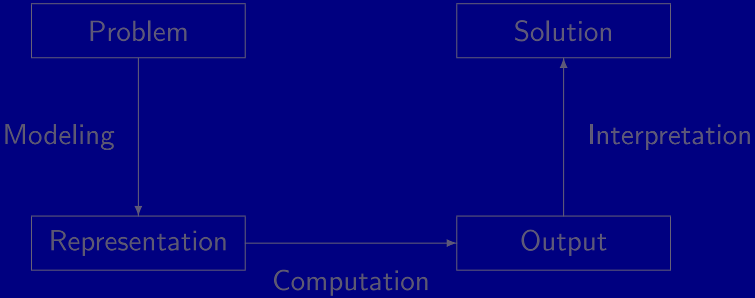
- *"What is the problem?"*
  instead of
- *"How to solve the problem?"*



Goal: Declarative problem solving

- *"What is the problem?"*

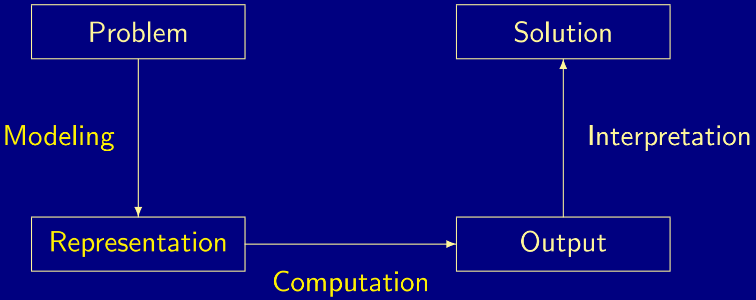
  instead of
- *"How to solve the problem?"*



Goal: Declarative problem solving

- *"What is the problem?"*

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



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
- ASP has its roots in
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - (deductive) databases
  - constraint solving (in particular, SATisfiability testing)
  - logic programming (with negation)
- 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
- ASP embraces many emerging application areas!

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
- ASP has its roots in
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - (deductive) databases
  - constraint solving (in particular, SATisfiability testing)
  - logic programming (with negation)
- 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
- ASP embraces many emerging application areas!

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
- ASP has its roots in
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - (deductive) databases
  - constraint solving (in particular, SATisfiability testing)
  - logic programming (with negation)
- 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
- ASP embraces many emerging application areas!

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
- ASP has its roots in
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - (deductive) databases
  - constraint solving (in particular, SATisfiability testing)
  - logic programming (with negation)
- 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
- ASP embraces many emerging application areas!

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
- ASP has its roots in
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - (deductive) databases
  - constraint solving (in particular, SATisfiability testing)
  - logic programming (with negation)
- 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
- ASP embraces many emerging application areas!

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
- ASP has its roots in
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - (deductive) databases
  - constraint solving (in particular, SATisfiability testing)
  - logic programming (with negation)
- 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
- ASP embraces many emerging application areas!

Logic Programming

- Algorithm = Logic + Control [53]
- 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

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

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

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

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

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

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

Common approach (eg. Prolog)

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

Model-based approach (eg. 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 [51]

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

Model-based Problem Solving

Common approach (eg. Prolog)

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

Model-based approach (eg. 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 [51]

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

Model-based Problem Solving

Common approach (eg. Prolog)

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

Model-based approach (eg. 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 [51]

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

Model-based Problem Solving

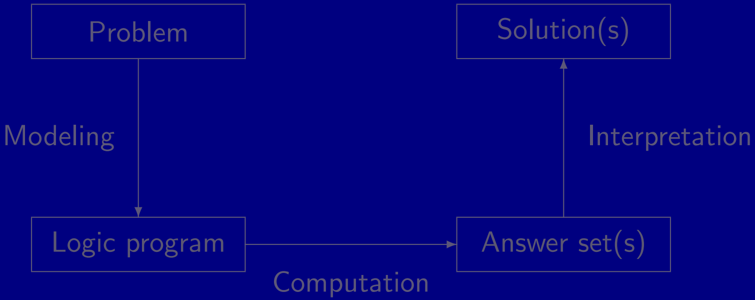
Specification	Associated Structures
constraint satisfaction problem	assignment
propositional horn theories	smallest model
propositional theories	models
propositional theories	minimal models
propositional theories	stable models
propositional programs	minimal models
propositional programs	supported models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

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
...	

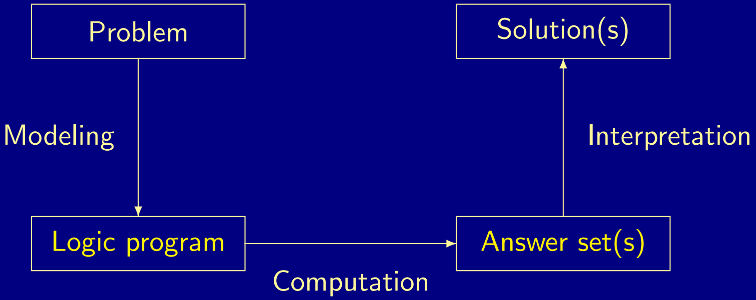
ASP as High-level Language

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



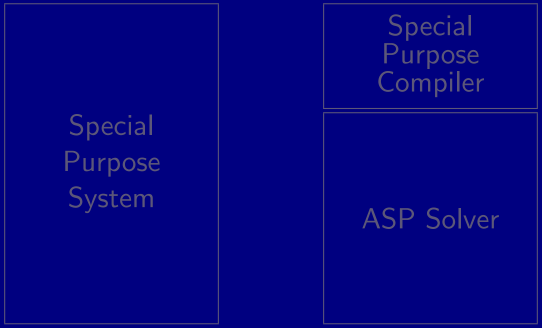
ASP as High-level Language

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



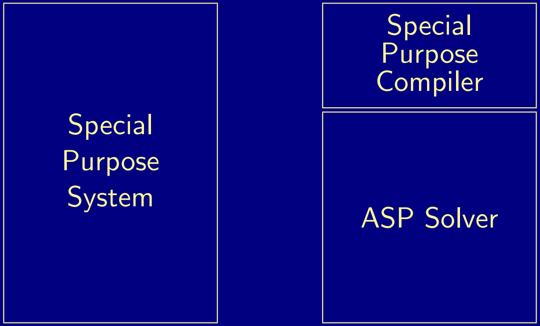
ASP as Low-level Language

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



ASP as Low-level Language

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



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 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 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.)

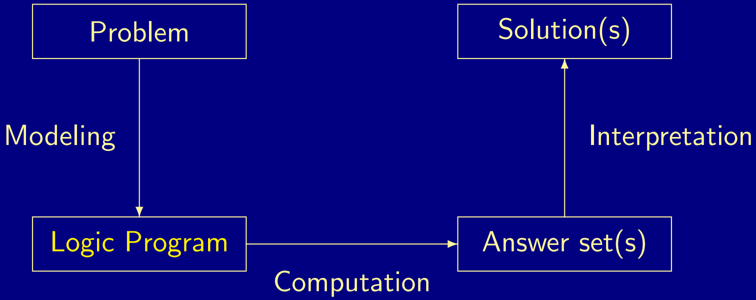
What does ASP offer?

- 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.)

$$\text{ASP} = \text{KR} + \text{DB} + \text{Search}$$

Introduction
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

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

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

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

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

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

Normal logic programs

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

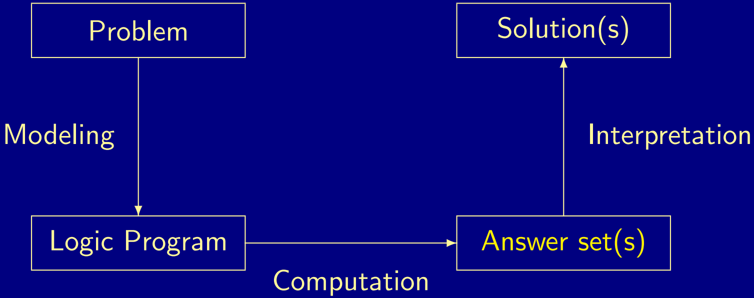
$$A_0 \leftarrow A_1, \dots, A_m, \mathit{not}\ A_{m+1}, \dots, \mathit{not}\ A_n,$$

  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

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

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

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.

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.

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.

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.

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.

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.

(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	$\rightarrow$	$\wedge$	$\vee$	$\sim/(\neg)$	$\neg$

Answer set: Basic idea

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

Φ $\boxed{q \ \wedge \ (q \wedge \neg r \rightarrow p)}$

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

Formula Φ has one stable model, called answer set:

$$\Pi_\Phi \quad \begin{array}{|lcl|} \hline q & \leftarrow & \\ p & \leftarrow & q, \ \mathit{not}\ r \\ \hline \end{array}$$

$$\{p, q\}$$

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: Basic idea

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

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

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

Formula Φ has one stable model, called answer set:

Π_Φ

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

$$\{p, q\}$$

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: Basic idea

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

Φ $\boxed{q \ \wedge \ (q \wedge \neg r \rightarrow p)}$

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

Formula Φ has one stable model, called answer set:

$$\boxed{\begin{array}{ccc} p & \mapsto & 1 \\ q & \mapsto & 1 \\ r & \mapsto & 0 \end{array}}$$

$\{p, q\}$

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

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: Basic idea

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

Φ $\boxed{q \ \land \ (q \land \neg r \rightarrow p)}$

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

Formula Φ has one stable model, called answer set:

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

$$\{p, q\}$$

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: Basic idea

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

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

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

Formula Φ has one stable model, called answer set:

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

$$\{p, q\}$$

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: Basic idea

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

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

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

Formula Φ has one stable model, called answer set:

$\{p,q\}$

$$\Pi_\Phi \quad \boxed{\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q,\ \mathit{not}\ r \end{array}}$$

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: Basic idea

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

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

Φ $\boxed{q \ \wedge \ (q \wedge \neg r \rightarrow p)}$

Formula Φ has one stable model, called answer set:

$$\{p, q\}$$

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

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

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

- A set X of atoms is an answer set of a 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"*

Answer set: Formal Definition

Normal programs

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

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

- A set X of atoms is an answer set of a 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"*

Answer set: Formal Definition

Normal programs

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

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

- A set X of atoms is an answer set of a 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"*

Answer set: Formal Definition

Normal programs

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

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

- A set X of atoms is an answer set of a 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"*

Answer set: Formal Definition

Normal programs

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

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

- A set X of atoms is an answer set of a 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.

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 first example

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

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 first example

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

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 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 first example

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

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

A third example

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

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

A third example

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

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

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 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: Some properties

- A 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 $X \not\subset Y$.

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^j_Π for $j \geq 0$, where $T^0_\Pi X = X$ and $T^i_\Pi X = T_\Pi T^{i-1}_\Pi X$ for $i \geq 1$.

Theorem

For any positive program Π, we have

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

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^j_Π for $j \geq 0$, where $T^0_\Pi X = X$ and $T^i_\Pi X = T_\Pi T^{i-1}_\Pi X$ for $i \geq 1$.

Theorem

For any positive program Π, we have

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

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

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

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

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 $X \subseteq \{p,q,r,t,s\}$.

Let's iterate T_Π

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

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

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 $X \subseteq \{p,q,r,t,s\}$.

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 $X \subseteq \{p,q,r,t,s\}$.

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}:

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

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

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

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}:

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

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

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

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:

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

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}:

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

  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

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

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

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

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

☞ Intelligent Grounding aims at reducing the ground instantiation.

An example

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

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

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

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

☞ Intelligent Grounding aims at reducing the ground instantiation.

An example

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

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

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

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

☞ 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 $Cn(ground(\Pi)^X) = X$.

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ 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`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ 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`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ 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`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ 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`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ 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`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ 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`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ 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`

Language Constructs

- Variables (over the Herbrand Universe)
  - `p(X) :- q(X)` over constants $\{a, b, c\}$ 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`

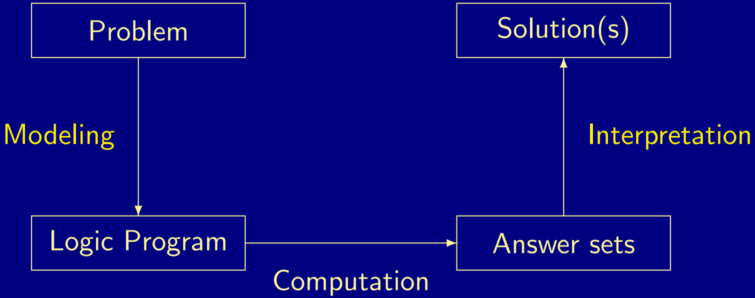
Reasoning Modes

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

[†] without solution recording

[‡] without solution enumeration

Modeling and Interpreting



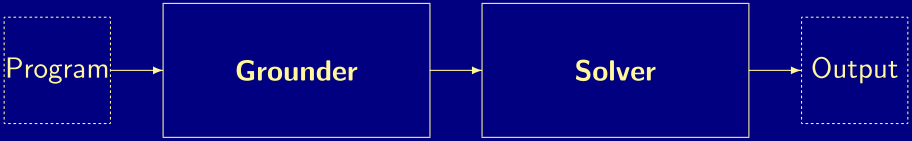
Modeling

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

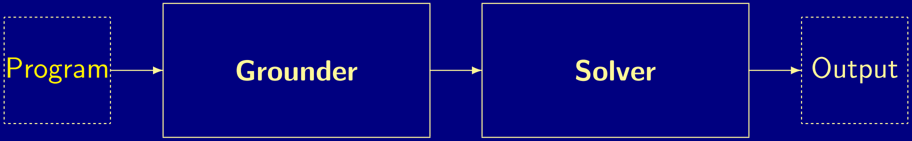
1. the problem instance I as a set $C(I)$ of facts and
2. the problem class P as a set $C(P)$ of rules

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

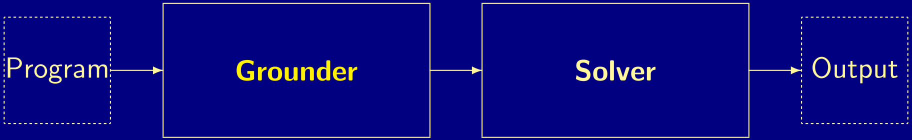
ASP Solving Process



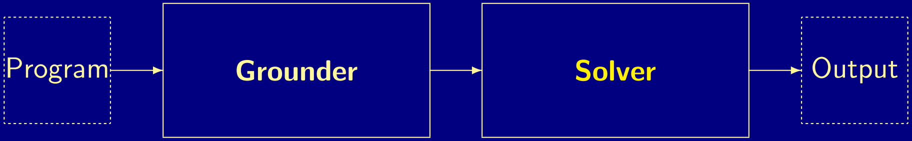
ASP Solving Process



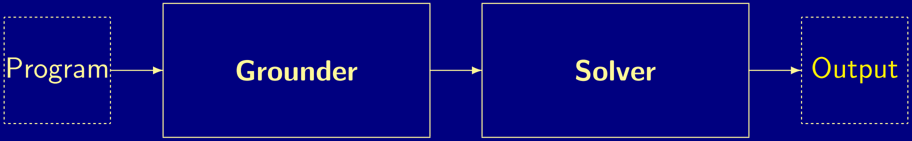
ASP Solving Process



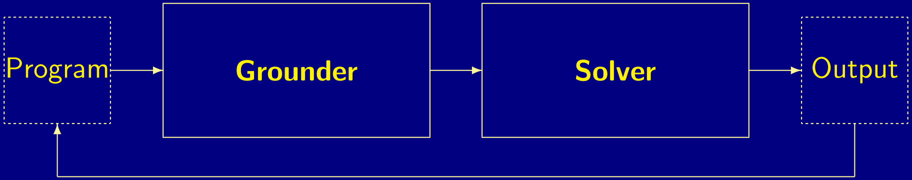
ASP Solving Process



ASP Solving Process



ASP Solving Process



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

```
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

```
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

```
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: 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)
```

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)
```

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

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

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	$X_2 = \{\}$

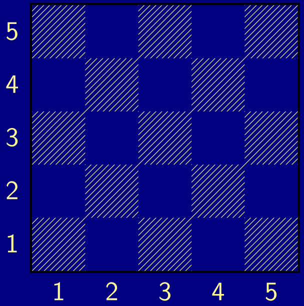
Satisfiability

- 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.

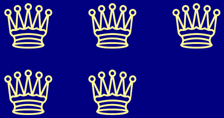
- Example: Consider formula $(a \vee \neg b) \wedge (\neg a \vee b)$.
- Logic Program:

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

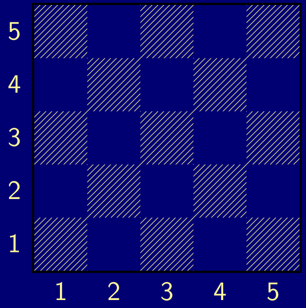
Placing some Queens

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+
```

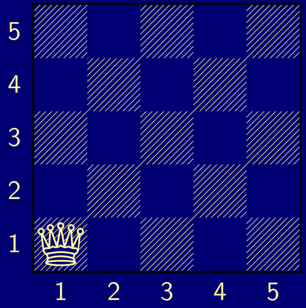
Placing some Queens: Answer 1

Answer 1



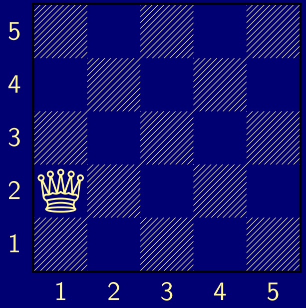
Placing some Queens: Answer 2

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

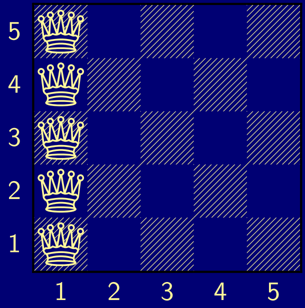
Placing n Queens

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)
...
```

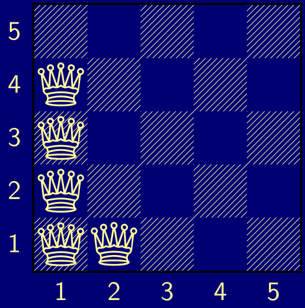
Placing n Queens: Answer 1

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

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

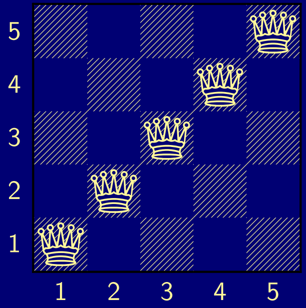
Horizontal and vertical Attack

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

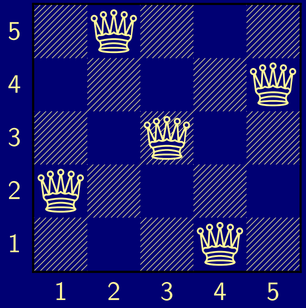
Diagonal Attack

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,p3).
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) }.
```

Reviewer Assignment
by Ilkka Niemelä

```
reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
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) }.
```

Reviewer Assignment
by Ilkka Niemelä

```
reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
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) }.
```

Reviewer Assignment

by Ilkka Niemelä

```
reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
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) }.
```

Reviewer Assignment
by Ilkka Niemelä

```
reviewer(r1). paper(p1). classA(r1,p1). classB(r1,p2). coi(r1,p3).
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

```
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

```
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).
```

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).
```

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).
```

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).
```

Language Extensions
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?

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?

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?

Integrity Constraints

Purpose Integrity constraints eliminate unwanted solution candidates

Syntax An integrity constraints 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 $\Pi'' = \Pi \cup \{\leftarrow \mathit{not}\ p\}$

Integrity Constraints

Purpose Integrity constraints eliminate unwanted solution candidates

Syntax An integrity constraints 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 $\Pi'' = \Pi \cup \{\leftarrow \mathit{not}\ p\}$

Integrity Constraints

Purpose Integrity constraints eliminate unwanted solution candidates

Syntax An integrity constraints 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 $\Pi'' = \Pi \cup \{\leftarrow \mathit{not}\ p\}$

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/nomore/clasp

Choice rules

Idea Choices over subsets.

Syntax

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

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/nomore/clasp

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/nomore/clasp

Embedding in normal logic programs

- A choice rule of form

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

can be translated into $2m+1$ rules

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

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

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
$$l\ \{A_1 : B_1, \ldots, A_m : B_m\}\ u$$
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/nomore/clasp

Cardinality rules

Idea Control cardinality of subsets.

Syntax

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

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/nomore/clasp

☞ gringo distinguishes sets and multi-sets!

Embedding in normal logic programs (ctd)

- Replace each cardinality rule

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

where atom $cc(A_i, j)$ represents the fact that at least j of the atoms in $\{A_i, \ldots, A_m\}$, 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?

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

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

can be represented by the cardinality rule

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

Cardinality rules with upper bounds

- A rule of the form

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

stands for

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

Cardinality constraints as heads

- A rule of the form

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

stands for

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

Full-fledged cardinality rules

- A rule of the form

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

stands for $0 \leq i \leq n$

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

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

Full-fledged cardinality rules

- A rule of the form

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

stands for $0 \leq i \leq n$

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

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

Weight constraints

Syntax $l\ [A_1 = w_1, \dots, A_m = w_m,$
$\quad not\ A_{m+1} = w_{m+1}, \dots, not\ A_n = w_n]\ u$

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/nomore/clasp

☞ gringo distinguishes sets and multi-sets!

Optimization statements

Idea Compute optimal answer sets by minimizing or maximizing a weighted sum of given atoms, 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/cmodels/nomore/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

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

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

Informal meaning

List all ground instances of A_0 such that corresponding instances of $A_1, \dots, A_m, not\ A_{m+1}, \dots, not\ A_n$ 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)}.
```

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

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

Informal meaning

List all ground instances of A_0 such that corresponding instances of $A_1, \dots, A_m, not\ A_{m+1}, \dots, not\ A_n$ 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 lparse and gringo documentations for further details.

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 `lparse` and `gringo` documentations for further details.

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 `lparse` and `gringo` documentations for further details.

Normal form in `lparse` and `gringo`

- Consider a logic program consisting of
  - normal rules
  - choice rules
  - cardinality rules
- Such a format is obtained by `lparse` or `gringo`
  and directly implemented by `smodels` and `clasp`.

Classical Negation
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.

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.

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.

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.

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.

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.

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

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

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

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

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

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: $\{q\}$
- $\Pi = \{p\,;\mathit{not}\ p \leftarrow \top,\ \neg p\,;\mathit{not}\ q \leftarrow \top,\ q\,;\mathit{not}\ q \leftarrow \top\}$

  ☞ Π is a *nested* logic program.

  $\Pi' = \Pi \cup \{A \leftarrow (B, \neg B),\ \neg A \leftarrow (B, \neg B) \mid A, B \in \{p, q\}\}$

  Answer sets: $\emptyset$, $\{p\}$, $\{\neg p, q\}$, and $\{p, \neg p, q, \neg q\}$

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: $\{q\}$
- $\Pi = \{p \,;\, \mathit{not}\ p \leftarrow \top,\ \neg p \,;\, \mathit{not}\ q \leftarrow \top,\ q \,;\, \mathit{not}\ q \leftarrow \top\}$
  ☞ Π is a *nested* logic program.
  $\Pi' = \Pi \cup \{A \leftarrow (B, \neg B),\ \neg A \leftarrow (B, \neg B) \mid A, B \in \{p, q\}\}$
  Answer sets: $\emptyset$, $\{p\}$, $\{\neg p, q\}$, and $\{p, \neg p, q, \neg q\}$

Example

- $\Pi = \{p \leftarrow,\ \neg p \leftarrow,\ q \leftarrow not\ r\}$
  $\Pi' = \Pi \cup \{A \leftarrow (B, \neg B),\ \neg A \leftarrow (B, \neg B) \mid A, B \in \{p, q, r\}\}$
  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: $\{q\}$
- $\Pi = \{p\ ; not\ p \leftarrow \top,\ \neg p\ ; not\ q \leftarrow \top,\ q\ ; not\ q \leftarrow \top\}$
  - ☞ Π is a *nested* logic program.

  $\Pi' = \Pi \cup \{A \leftarrow (B, \neg B),\ \neg A \leftarrow (B, \neg B) \mid A, B \in \{p, q\}\}$
  Answer sets: $\emptyset$, $\{p\}$, $\{\neg p, q\}$, and $\{p, \neg p, q, \neg q\}$

Example

- $\Pi = \{p \leftarrow,\ \neg p \leftarrow,\ q \leftarrow not\ r\}$
  $\Pi' = \Pi \cup \{A \leftarrow (B, \neg B),\ \neg A \leftarrow (B, \neg B) \mid A, B \in \{p, q, r\}\}$
  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: $\{q\}$
- $\Pi = \{p\,; not\ p \leftarrow \top,\ \neg p\,; not\ q \leftarrow \top,\ q\,; not\ q \leftarrow \top\}$
  ☞ Π is a *nested* logic program.
  $\Pi' = \Pi \cup \{A \leftarrow (B, \neg B),\ \neg A \leftarrow (B, \neg B) \mid A, B \in \{p, q\}\}$
  Answer sets: $\emptyset$, $\{p\}$, $\{\neg p, q\}$, and $\{p, \neg p, q, \neg q\}$

Example

- $\Pi = \{p \leftarrow,\ \neg p \leftarrow,\ q \leftarrow not\ r\}$
  $\Pi' = \Pi \cup \{A \leftarrow (B, \neg B),\ \neg A \leftarrow (B, \neg B) \mid A, B \in \{p, q, r\}\}$
  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: $\{q\}$
- $\Pi = \{p\,; not\ p \leftarrow \top,\ \neg p\,; not\ q \leftarrow \top,\ q\,; not\ q \leftarrow \top\}$
  ☞ Π is a *nested* logic program.
  $\Pi' = \Pi \cup \{A \leftarrow (B, \neg B),\ \neg A \leftarrow (B, \neg B) \mid A, B \in \{p, q\}\}$
  Answer sets: $\emptyset$, $\{p\}$, $\{\neg p, q\}$, and $\{p, \neg p, q, \neg q\}$

Example

- $\Pi = \{p \leftarrow,\ \neg p \leftarrow,\ q \leftarrow not\ r\}$
  $\Pi' = \Pi \cup \{A \leftarrow (B, \neg B),\ \neg A \leftarrow (B, \neg B) \mid A, B \in \{p, q, r\}\}$
  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: $\{q\}$
- $\Pi = \{p\,; not\ p \leftarrow \top,\ \neg p\,; not\ q \leftarrow \top,\ q\,; not\ q \leftarrow \top\}$
  ☞ Π is a *nested* logic program.
  $\Pi' = \Pi \cup \{A \leftarrow (B, \neg B),\ \neg A \leftarrow (B, \neg B) \mid A, B \in \{p, q\}\}$
  Answer sets: $\emptyset$, $\{p\}$, $\{\neg p, q\}$, and $\{p, \neg p, q, \neg q\}$

Example

- $\Pi = \{p \leftarrow,\ \neg p \leftarrow,\ q \leftarrow not\ r\}$
  $\Pi' = \Pi \cup \{A \leftarrow (B, \neg B),\ \neg A \leftarrow (B, \neg B) \mid A, B \in \{p, q, r\}\}$
  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: $\{q\}$
- $\Pi = \{p\ ; not\ p \leftarrow \top,\ \neg p\ ; not\ q \leftarrow \top,\ q\ ; not\ q \leftarrow \top\}$
  - ☞ Π is a *nested* logic program.

  $\Pi' = \Pi \cup \{A \leftarrow (B, \neg B),\ \neg A \leftarrow (B, \neg B) \mid A, B \in \{p, q\}\}$
  Answer sets: $\emptyset$, $\{p\}$, $\{\neg p, q\}$, and $\{p, \neg p, q, \neg q\}$

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: $\{q\}$
- $\Pi = \{p\ ; \mathit{not}\ p \leftarrow \top,\ \neg p\ ; \mathit{not}\ q \leftarrow \top,\ q\ ; \mathit{not}\ q \leftarrow \top\}$
  - ☞ Π is a *nested* logic program.

  $\Pi' = \Pi \cup \{A \leftarrow (B, \neg B),\ \neg A \leftarrow (B, \neg B) \mid A, B \in \{p, q\}\}$
  Answer sets: $\emptyset$, $\{p\}$, $\{\neg p, q\}$, and $\{p, \neg p, q, \neg q\}$

Disjunctive Logic Programs
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.

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 $X \in \min_\subseteq(\Pi^X)$.

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 $X \in \min_\subseteq(\Pi^X)$.

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 $X \in \min_\subseteq(\Pi^X)$.

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 $\min_{\subseteq}(\Pi) = \{\ \{a,b\},\ \{a,c\}\ \}$.

A "positive" example

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

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

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 $\min_{\subseteq}(\Pi) = \{\ \{a, b\},\ \{a, c\}\ \}$.

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 $\{b\}$.

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 $\{b\}$.

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 $\{b\}$.

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 $\{b\}$.

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 $\{b\}$.

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

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

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

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

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

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

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

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

- 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

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

- Consider $X = \{a(1,2),\ b(1)\}$.
- 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

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

- 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 $X = \{a(1,2),\ b(1)\}$.
- 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 $X = \{a(1,2),\ b(1)\}$.
- 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 $X \notin \min_{\subseteq}(ground(\Pi)^X)$.

An example with variables

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

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

An example with variables

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

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

An example with variables

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

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

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

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 $body(r) = G$.

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 $body(r) = G$.

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 $body(r) = G$.

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 $\min_{\subseteq}(\Pi)$.

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 $\min_{\subseteq}(\Pi)$.

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 $\bot$,
  - $(F, G)^X = (F^X, G^X)$,
  - $(F; G)^X = (F^X; G^X)$,
  - $(not\ F)^X = \begin{cases} \bot & \text{if } X \models F \\ \top & \text{otherwise} \end{cases}$
- 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 $X \in \min_{\subseteq}(\Pi^X)$.

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 $\bot$,
  - $(F, G)^X = (F^X, G^X)$,
  - $(F; G)^X = (F^X; G^X)$,
  - $(not\ F)^X = \begin{cases} \bot & \text{if } X \models F \\ \top & \text{otherwise} \end{cases}$
- 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 $X \in \min_{\subseteq}(\Pi^X)$.

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 $\bot$,
  - $(F, G)^X = (F^X, G^X)$,
  - $(F; G)^X = (F^X; G^X)$,
  - $(not\ F)^X = \begin{cases} \bot & \text{if } X \models F \\ \top & \text{otherwise} \end{cases}$
- 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 $X \in \min_\subseteq(\Pi^X)$.

Two examples

- $\Pi_1 = \{(p \,;\, \mathit{not}\ p) \leftarrow \top\}$

  For $X = \emptyset$, we get

  $\Pi_1^{\emptyset} = \{(p \,;\, \top) \leftarrow \top\}$
  $\min_{\subseteq}(\Pi_1^{\emptyset}) = \{\emptyset\}$.

  For $X = \{p\}$, we get

  $\Pi_1^{\{p\}} = \{(p \,;\, \bot) \leftarrow \top\}$
  $\min_{\subseteq}(\Pi_1^{\{p\}}) = \{\{p\}\}$.

- $\Pi_2 = \{p \leftarrow \mathit{not}\ \mathit{not}\ p\}$

  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,

  $F \leftarrow G,\ \mathit{not}\ \mathit{not}\ H$ is equivalent to $F \,;\, \mathit{not}\ H \leftarrow G$
  $F \,;\ \mathit{not}\ \mathit{not}\ G \leftarrow H$ is equivalent to $F \leftarrow H, \mathit{not}\ G$
  $\mathit{not}\ \mathit{not}\ \mathit{not}\ F$ is equivalent to $\mathit{not}\ F$

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

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)

Two examples

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

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

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)

Two examples

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

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

Two examples

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

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

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)

Two examples

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

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

Two examples

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

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

Two examples

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

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

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)

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)

Two examples

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

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

Two examples

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

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

Two examples

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

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

Two examples

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

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

Some more examples

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

Propositional Theories
Overview

Propositional theories

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

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

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

A propositional theory is a finite set of formulas.

Propositional theories

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

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

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

- A propositional theory is a finite set of formulas.

Propositional theories

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

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

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

- A propositional theory is a finite set of formulas.

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

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

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 $\circ \in \{\land, \lor, \rightarrow\}$
  - 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

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

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

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

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

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

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

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

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
  - $\min_{\subseteq}(\mathcal{F}^X) = \{X\}$.

This does generally not imply $X \in \min_{\subseteq}(\mathcal{F})$!

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
  - $\min_{\subseteq}(\mathcal{F}^X) = \{X\}$.

This does generally not imply $X \in \min_{\subseteq}(\mathcal{F})$!

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
  - $\min_{\subseteq}(\mathcal{F}^X) = \{X\}$.

  - Note: This does generally not imply $X \in \min_{\subseteq}(\mathcal{F})$!

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
  - $\min_{\subseteq}(\mathcal{F}^X) = \{X\}$.

  - Note: This does generally not imply $X \in \min_{\subseteq}(\mathcal{F})$!

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 $\min_{\subseteq}(\mathcal{F}_2^{\{q,r\}}) = \{\{q, r\}\}$.

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 $\min_{\subseteq}(\mathcal{F}_2^{\{q,r\}}) = \{\{q, r\}\}$.

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 $\min_{\subseteq}(\mathcal{F}_2^{\{q,r\}}) = \{\{q, r\}\}$.

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 $\min_{\subseteq}(\mathcal{F}_2^{\{q,r\}}) = \{\{q, r\}\}$.

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 $\min_{\subseteq}(\mathcal{F}_2^{\{q,r\}}) = \{\{q, r\}\}$.

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 $\min_\subseteq(\mathcal{F}_2^{\{q,r\}}) = \{\{q, r\}\}$.

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 $\min_{\subseteq}(\mathcal{F}_2^{\{q,r\}}) = \{\{q, r\}\}$.

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 $\min_{\subseteq}(\mathcal{F}_2^{\{q,r\}}) = \{\{q, r\}\}$.

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 $\min_{\subseteq}(\mathcal{F}_2^{\{q,r\}}) = \{\{q, r\}\}$.

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 $\min_{\subseteq}(\mathcal{F}_2^{\{q,r\}}) = \{\{q, r\}\}$.

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 $\min_{\subseteq}(\mathcal{F}_2^{\{q,r\}}) = \{\{q, r\}\}$. ✔

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 $\min_\subseteq(\mathcal{F}_2^{\{q,r\}}) = \{\{q, r\}\}$. ✔

Relationship with logic programs

- The translation, $\tau[(F \leftarrow G)]$, of a (nested) rule $(F \leftarrow G)$ is defined recursively as follows:
  - $\tau[(F \leftarrow G)] = (\tau[G] \rightarrow \tau[F])$,
  - $\tau[\bot] = \bot$,
  - $\tau[\top] = \top$,
  - $\tau[F] = F$ 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 $\tau[\Pi]$.

Relationship with logic programs

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

- 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 $\tau[\Pi]$.

Relationship with logic programs

- The translation, $\tau[(F \leftarrow G)]$, of a (nested) rule $(F \leftarrow G)$ is defined recursively as follows:
  - $\tau[(F \leftarrow G)] = (\tau[G] \rightarrow \tau[F])$,
  - $\tau[\bot] = \bot$,
  - $\tau[\top] = \top$,
  - $\tau[F] = F$ 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 $\tau[\Pi]$.

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 $\{p\}$

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 $\{p\}$

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 $\{p\}$

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 $\{p\}$

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 $\{p\}$

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 $\{p\}$

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

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

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

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

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.

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.

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.

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.

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.

Completion
Overview

Completion

Let Π be a normal logic program,
and recall that $atom(\Pi)$ denotes the set of atoms occurring in Π.
The completion of Π is defined as follows:

$$Comp(body(r)) = \bigwedge_{A \in body^+(r)} A \wedge \bigwedge_{A \in body^-(r)} \neg A$$

$$Comp(\Pi) = \{A \leftrightarrow \bigvee_{r \in \Pi, head(r)=A} Comp(body(r)) \mid A \in atom(\Pi)\}$$

- 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.

Completion

Let Π be a normal logic program,
and recall that $atom(\Pi)$ denotes the set of atoms occurring in Π.
The completion of Π is defined as follows:

$$Comp(body(r)) = \bigwedge_{A \in body^+(r)} A \wedge \bigwedge_{A \in body^-(r)} \neg A$$

$$Comp(\Pi) = \{A \leftrightarrow \bigvee_{r \in \Pi, head(r)=A} Comp(body(r)) \mid A \in atom(\Pi)\}$$

- 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.

Completion

Let Π be a normal logic program,
and recall that $atom(\Pi)$ denotes the set of atoms occurring in Π.
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.

Completion

Let Π be a normal logic program,
and recall that $atom(\Pi)$ denotes the set of atoms occurring in Π.
The completion of Π is defined as follows:

$$Comp(body(r)) = \bigwedge_{A \in body^+(r)} A \wedge \bigwedge_{A \in body^-(r)} \neg A$$

$$Comp(\Pi) = \{A \leftrightarrow \bigvee_{r \in \Pi, head(r)=A} Comp(body(r)) \mid A \in atom(\Pi)\}$$

- 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.

Completion

Let Π be a normal logic program,
and recall that $atom(\Pi)$ denotes the set of atoms occurring in Π.
The completion of Π is defined as follows:

$$Comp(body(r)) = \bigwedge_{A \in body^+(r)} A \wedge \bigwedge_{A \in body^-(r)} \neg A$$

$$Comp(\Pi) = \{A \leftrightarrow \bigvee_{r \in \Pi, head(r)=A} Comp(body(r)) \mid A \in atom(\Pi)\}$$

- 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.

Completion

Let Π be a normal logic program,
and recall that $atom(\Pi)$ denotes the set of atoms occurring in Π.
The completion of Π is defined as follows:

$$Comp(body(r)) = \bigwedge_{A \in body^+(r)} A \wedge \bigwedge_{A \in body^-(r)} \neg A$$

$$Comp(\Pi) = \{A \leftrightarrow \bigvee_{r \in \Pi, head(r)=A} Comp(body(r)) \mid A \in atom(\Pi)\}$$

- 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 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 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

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

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

A second example

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

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

A second example

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

- 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 $\emptyset$!

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 $\emptyset$!

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 $\emptyset$!

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: 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: 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

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

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 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 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 operator: Example

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

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 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:

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

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

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

Fitting semantics

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

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

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

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

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: Example

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

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

Fitting semantics: Example

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

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

Fitting semantics: Example

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

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

Fitting semantics: 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: Example

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

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

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

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.

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.

Fitting fixpoints: Example

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

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.

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.

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.

Fitting fixpoints: Example

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

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 !

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 !

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 !

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 !

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

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

$$\begin{array}{lcl} \Phi^0_\Pi \langle \emptyset, \emptyset \rangle & = & \langle \emptyset, \emptyset \rangle \\ \Phi^1_\Pi \langle \emptyset, \emptyset \rangle & = & \langle \emptyset, \emptyset \rangle \end{array}$$

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

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* !

(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

- 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

  $$\langle r_1, \dots, r_n \rangle$$

  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 $\{a \leftarrow b,\ b \leftarrow a\}$?

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

  $$\langle r_1, \dots, r_n \rangle$$

  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 $\{a \leftarrow b,\ b \leftarrow a\}$?

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

  $$\langle r_1, \dots, r_n \rangle$$

  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 $\{a \leftarrow b,\ b \leftarrow a\}$?

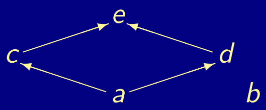
Positive atom dependency graph

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

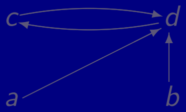
1. $V = atom(\Pi)$ and
2. $E = \{(p, q) \mid r \in \Pi, p \in body^+(r), head(r) = q\}$.

Examples

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

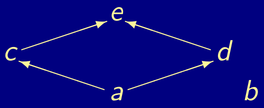


$$\Pi_3 = \left\{ \begin{array}{ll} a \leftarrow \mathit{not}\ b & b \leftarrow \mathit{not}\ a \\ c \leftarrow \mathit{not}\ a & c \leftarrow d \\ d \leftarrow a, b & d \leftarrow c \end{array} \right\}$$

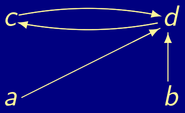


Examples

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



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



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.

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.

Tight programs

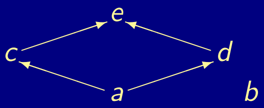
- A normal logic program Π is tight iff $G(\Pi)$ is acyclic.
- For example, Π_2 is tight, whereas Π_3 is not.
- If a normal logic program Π is tight, then

  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

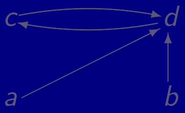
$$\Pi_2 = \left\{ \begin{array}{ll} a \leftarrow not\ b & b \leftarrow not\ a \\ c \leftarrow a, not\ d & d \leftarrow a, not\ c \\ e \leftarrow c, not\ a & e \leftarrow d, not\ b \end{array} \right\}$$



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

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

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

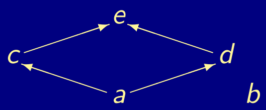


Answer sets: $\{\{a\}, \{b, c, d\}\}$

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

(Non-)tight programs: Examples

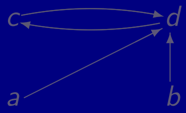
$$\Pi_2 = \left\{ \begin{array}{ll} a \leftarrow \mathit{not}\ b & b \leftarrow \mathit{not}\ a \\ c \leftarrow a, \mathit{not}\ d & d \leftarrow a, \mathit{not}\ c \\ e \leftarrow c, \mathit{not}\ a & e \leftarrow d, \mathit{not}\ b \end{array} \right\}$$



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

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

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

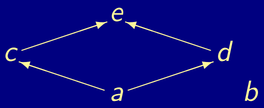


Answer sets: $\{\{a\}, \{b, c, d\}\}$

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

(Non-)tight programs: Examples

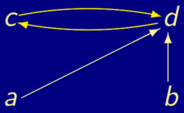
$$\Pi_2 = \left\{ \begin{array}{ll} a \leftarrow \mathit{not}\ b & b \leftarrow \mathit{not}\ a \\ c \leftarrow a, \mathit{not}\ d & d \leftarrow a, \mathit{not}\ c \\ e \leftarrow c, \mathit{not}\ a & e \leftarrow d, \mathit{not}\ b \end{array} \right\}$$



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

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

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

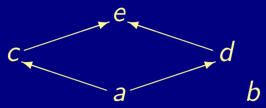


Answer sets: $\{\{a\}, \{b, c, d\}\}$

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

(Non-)tight programs: Examples

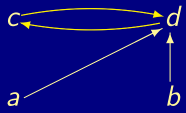
$$\Pi_2 = \left\{ \begin{array}{ll} a \leftarrow \mathit{not}\ b & b \leftarrow \mathit{not}\ a \\ c \leftarrow a, \mathit{not}\ d & d \leftarrow a, \mathit{not}\ c \\ e \leftarrow c, \mathit{not}\ a & e \leftarrow d, \mathit{not}\ b \end{array} \right\}$$



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

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

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



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

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

Unfounded Sets
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.

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.

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.

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.

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.

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.

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.

Example

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

- $\emptyset$ 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.

Example

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

- $\emptyset$ 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.

Example

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

- $\emptyset$ 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.

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.

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.

Example

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

- $\emptyset$ 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 $\langle T, F\rangle$.

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

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

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

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

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

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

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

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

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

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

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

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 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 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 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 operator: Example

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

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 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

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

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

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

Well-founded semantics: Example

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

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

Well-founded semantics: Example

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

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

Well-founded semantics: Example

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

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

Well-founded semantics: Example

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

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

Well-founded semantics: Example

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

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

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

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.

Well-founded fixpoints: Example

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

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.

Well-founded fixpoints: Example

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

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.

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.

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, originating from program completion (although this is unnecessary from a formal point of view).

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, originating from program completion (although this is unnecessary from a formal point of view).

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, originating from program completion (although this is unnecessary from a formal point of view).

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, originating from program completion (although this is unnecessary from a formal point of view).

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, originating from program completion (although this is unnecessary from a formal point of view).

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.

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.

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.

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 $Loop(\Pi) = \emptyset$.

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

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

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

- Define

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

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

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

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

- Define

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

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

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

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

- Define

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

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

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

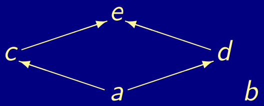
Lin-Zhao Theorem

Theorem

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

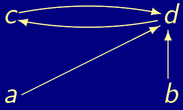
Loops and loop formulas: Examples

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



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

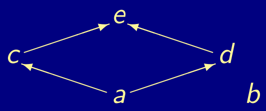
$$\Pi_3 = \left\{ \begin{array}{ll} a \leftarrow not\ b & b \leftarrow not\ a \\ c \leftarrow not\ a & c \leftarrow d \\ d \leftarrow a, b & d \leftarrow c \end{array} \right\}$$



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

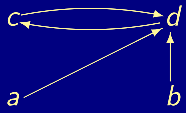
Loops and loop formulas: Examples

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



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

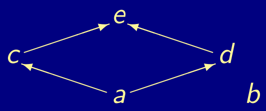
$$\Pi_3 = \left\{ \begin{array}{ll} a \leftarrow not\ b & b \leftarrow not\ a \\ c \leftarrow not\ a & c \leftarrow d \\ d \leftarrow a, b & d \leftarrow c \end{array} \right\}$$



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

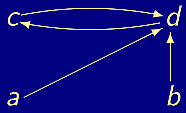
Loops and loop formulas: Examples

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



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

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



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

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

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

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

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

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

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] A conjecture from the theory of complexity that is widely believed to be true.

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] A conjecture from the theory of complexity that is widely believed to be true.

Loops and loop formulas: Properties (ctd)

If $\mathcal{P} \not\subseteq \mathcal{NC}^1/\mathit{poly}$,[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 $\mathit{atom}(\Pi)$.
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 $\mathit{Comp}(\Pi) \cup \mathit{LF}(\Pi)$ preserves the vocabulary of Π.
- The number of loops in $\mathit{Loop}(\Pi)$ may be exponential in $|\mathit{atom}(\Pi)|$.

[1] A conjecture from the theory of complexity that is widely believed to be true.

Tableau Calculi
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

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

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

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 [14].

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 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

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

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)	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)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	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.

Tableau calculus: Example (ctd)

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	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)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	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.

Tableau calculus: Example (ctd)

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	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)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	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.

Tableau calculus: Example (ctd)

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	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)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	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.

Tableau calculus: Example (ctd)

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	[φ]
(2)	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)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	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.

Tableau calculus: Example (ctd)

(1) $a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$ $[\varphi]$
(2) 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]
(5) $\neg b$ [4]
(6) $\neg a \vee b$ [4]

(7) $\neg a$ [6]

(8) b [6]

(9) $\neg\neg\neg a$ [3]
(10) $\neg a$ [9]

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.

Tableau calculus: Example (ctd)

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	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)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	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.

Tableau calculus: Example (ctd)

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	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)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	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.

Tableau calculus: Example (ctd)

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

Left branch:

(4)	$\neg b \wedge (\neg a \vee b)$	[3]
(5)	$\neg b$	[4]
(6)	$\neg a \vee b$	[4]

(7)	$\neg a$	[6]
(8)	b	[6]

Right branch:

(9)	$\neg\neg\neg a$	[3]
(10)	$\neg a$	[9]

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.

Tableau calculus: Example (ctd)

(1)	$a \wedge ((\neg b \wedge (\neg a \vee b)) \vee \neg\neg\neg a)$	$[\varphi]$
(2)	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)	$\neg b$	[4]	**(10)**	$\neg a$	[9]
(6)	$\neg a \vee b$	[4]			

(7)	$\neg a$	[6]	**(8)**	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.

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.

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.

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 [42]

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

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

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

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

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

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

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

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.

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.

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.

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.

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.

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 $v \in \mathit{atom}(\Pi) \cup \mathit{body}(\Pi)$.
- 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.

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 $v \in \mathit{atom}(\Pi) \cup \mathit{body}(\Pi)$.
- 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

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

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 $\{a, d\}$.

(Previewed) Example

$$a \leftarrow$$
$$c \leftarrow not\ b, not\ d$$
$$d \leftarrow a, not\ c$$

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

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

(Previewed) Example

$$a \leftarrow$$
$$c \leftarrow not\ b, not\ d$$
$$d \leftarrow a, not\ c$$

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

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

(Previewed) Example

$$
\begin{array}{rccccc}
 & & & a \leftarrow & & \\
 & & & c \leftarrow not\ b, not\ d & & \\
 & & & d \leftarrow a, not\ c & & \\
\text{(FTB)} & & & \mathbf{T}\emptyset & & \\
\text{(FTA)} & & & \mathbf{T}a & & \\
\text{(FFA)} & & & \mathbf{F}b & & \\
\text{(Cut}[atom(\Pi)]\text{)} & & \mathbf{T}c & & & \mathbf{F}c \\
 & \text{(BTA)} & \mathbf{T}\{not\ b, not\ d\} & & \text{(BFA)} & \mathbf{F}\{not\ b, not\ d\} \\
 & \text{(BTB)} & \mathbf{F}d & & \text{(BFB)} & \mathbf{T}d \\
 & \text{(FFB)} & \mathbf{F}\{a, not\ c\} & & \text{(FTB)} & \mathbf{T}\{a, not\ c\}
\end{array}
$$

Recall 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 $\{a, d\}$.

(Previewed) Example

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

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

Tableau rules: Auxiliary definitions

- The application of rules makes use of two conjugation functions, $\mathbf{t}$ and $\mathbf{f}$.
- For a literal l, define:

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

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

Examples

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

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:

$$\frac{\mathit{prerequisites}}{\mathit{consequence}}\ (\mathit{proviso}) \qquad \mathit{proviso}\text{: some condition(s)}$$

☞ 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

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

Example

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

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

$$\frac{\mathbf{F}\{l_1,\dots,l_i,\dots,l_n\} \quad \mathbf{t}l_1,\dots,\mathbf{t}l_{i-1},\mathbf{t}l_{i+1},\dots,\mathbf{t}l_n}{\mathbf{f}l_i}$$

Examples

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

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

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

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

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

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

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

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

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

Reviewing tableau rules for bodies

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

- Rules FTB and BFB amount to implication:

$$l_1 \wedge \cdots \wedge l_n \rightarrow B$$

- Rules FFB and BTB amount to implication:

$$B \rightarrow l_1 \wedge \cdots \wedge l_n$$

☞ Together they yield:

$$B \equiv l_1 \wedge \cdots \wedge l_n$$

Reviewing tableau rules for bodies

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

- Rules FTB and BFB amount to implication:

$$l_1 \wedge \cdots \wedge l_n \rightarrow B$$

- Rules FFB and BTB amount to implication:

$$B \rightarrow l_1 \wedge \cdots \wedge l_n$$

☞ Together they yield:

$$B \equiv l_1 \wedge \cdots \wedge l_n$$

Forward True Atom (FTA)

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

Consequence The atom is *true*.

Tableau Rule FTA

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

Examples

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

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

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

Examples

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

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

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

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

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

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

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

☞ 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

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

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

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 $body(p) = \{B_1, \ldots, B_m\}$.

- 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:

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

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 302)
- inferences via *smodels*' **atleast**

Given the same partial assignment (of atoms),

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

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 302)
- inferences via *smodels*' **atleast**

Given the same partial assignment (of atoms),

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

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 302)
- inferences via *smodels*' **atleast**

Given the same partial assignment (of atoms),

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

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 302)
- inferences via *smodels*' **atleast**

Given the same partial assignment (of atoms),

- any literal derived by *smodels*' **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.

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

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

Examples

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

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

Well-Founded 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

$$\frac{\mathbf{F}B_1,\ldots,\mathbf{F}B_m}{\mathbf{F}p}\ (p \in GUS(\{r \in \Pi \mid body(r) \notin \{B_1,\ldots,B_m\}\}))$$

Examples

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

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

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

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

Examples

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

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

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

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

Examples

$$\frac{a \leftarrow not\ b \quad \mathbf{T}a}{\mathbf{T}\{not\ b\}}\ (*) \qquad \frac{a \leftarrow a \quad a \leftarrow not\ b \quad \mathbf{T}a}{\mathbf{T}\{not\ b\}}\ (*)$$

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

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.

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 380)
2. $p \in GUS(\Pi')$.
   - ➥ Well-founded operator, *smodels*' **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.

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 380)
2. $p \in GUS(\Pi')$.

   ➥ Well-founded operator, *smodels*' **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.

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 380)
2. $p \in GUS(\Pi')$.
   - ➥ Well-founded operator, *smodels*' **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

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

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

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

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

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

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

Example

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

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

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

Example

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

Reviewing tableau rules for loops

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

Reviewing tableau rules for loops

Tableau rules FL and BL ensure non-circular support for *true* atoms.
For a loop L such that $EB(L) = \{B_1, \dots, B_m\}$,
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 422)
  - ☞ But impractical to precompute exponentially many loop formulas !
- an application of the Lin-Zhao Theorem (cf. Page 426)

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?

Relationship with loop formulas

Tableau rules FL and BL essentially provide:

- (straightforward) inferences from loop formulas (cf. Page 422)
  - ☞ But impractical to precompute exponentially many loop formulas !
- an application of the Lin-Zhao Theorem (cf. Page 426)

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?

Relationship with loop formulas

Tableau rules FL and BL essentially provide:

- (straightforward) inferences from loop formulas (cf. Page 422)
  - ☞ But impractical to precompute exponentially many loop formulas !
- an application of the Lin-Zhao Theorem (cf. Page 426)

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?

Relationship with loop formulas

Tableau rules FL and BL essentially provide:

- (straightforward) inferences from loop formulas (cf. Page 422)
  - ☞ But impractical to precompute exponentially many loop formulas !
- an application of the Lin-Zhao Theorem (cf. Page 426)

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?

Relationship with loop formulas

Tableau rules FL and BL essentially provide:

- (straightforward) inferences from loop formulas (cf. Page 422)
  - ☞ But impractical to precompute exponentially many loop formulas !
- an application of the Lin-Zhao Theorem (cf. Page 426)

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?

Relationship with loop formulas

Tableau rules FL and BL essentially provide:

- (straightforward) inferences from loop formulas (cf. Page 422)
  - ☞ But impractical to precompute exponentially many loop formulas !
- an application of the Lin-Zhao Theorem (cf. Page 426)

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 $Cut[\mathcal{C}]$

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

Examples $Cut[\mathcal{C}]$

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

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 $Cut[\mathcal{C}]$

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

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:

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

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:

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

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 [2, 59, 48, 74, 55, 52, 1]:

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

- 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.

Tableau calculi characterizing ASP-solvers

ASP-solvers combine propagation with case analysis.
We obtain the following tableau calculi characterizing [2, 59, 48, 74, 55, 52, 1]:

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

- 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 !

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 !

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 !

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:

$$\Pi_a^n = \left\{\begin{array}{l} x \leftarrow not\ x \\ x \leftarrow a_1, b_1 \\ \vdots \\ x \leftarrow a_n, b_n \end{array}\right\} \quad \Pi_b^n = \left\{\begin{array}{ll} \multicolumn{2}{l}{x \leftarrow c_1, \dots, c_n, not\ x} \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \vdots & \vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array}\right\} \quad \Pi_c^n = \left\{\begin{array}{l} a_1 \leftarrow not\ b_1 \\ b_1 \leftarrow not\ a_1 \\ \vdots \\ a_n \leftarrow not\ b_n \\ b_n \leftarrow not\ a_n \end{array}\right\}$$

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 $\mathcal{T}_{noMoRe}$.

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

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

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 $\mathcal{T}_{noMoRe}$.

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

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

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 $\mathcal{T}_{noMoRe}$.

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

$$\Pi_a^n = \left\{ \begin{array}{l} x \leftarrow not\ x \\ x \leftarrow a_1, b_1 \\ \quad\vdots \\ x \leftarrow a_n, b_n \end{array} \right\} \quad \Pi_b^n = \left\{ \begin{array}{ll} \multicolumn{2}{l}{x \leftarrow c_1, \dots, c_n, not\ x} \\ c_1 \leftarrow a_1 & c_1 \leftarrow b_1 \\ \quad\vdots & \quad\vdots \\ c_n \leftarrow a_n & c_n \leftarrow b_n \end{array} \right\} \quad \Pi_c^n = \left\{ \begin{array}{l} a_1 \leftarrow not\ b_1 \\ b_1 \leftarrow not\ a_1 \\ \quad\vdots \\ a_n \leftarrow not\ b_n \\ b_n \leftarrow not\ a_n \end{array} \right\}$$

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.

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.

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.

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

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

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

(29)	$\mathbf{F}f$	$[Cut]$
(30)	$\mathbf{T}\{not\ a, not\ f\}$	$[FTB: r_9, 16, 29]$
(31)	$\mathbf{T}g$	$[FTA: r_9, 30]$
(32)	$\mathbf{T}\{g\}$	$[FTB: r_4, r_6, 31]$
(33)	$\mathbf{F}\{not\ g\}$	$[FFB: r_8, 31]$

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

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

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

(30)	$\mathbf{F}\{not\ g\}$	$[Cut]$
(31)	$\mathbf{T}g$	$[BFB: 30]$
(32)	$\mathbf{T}\{g\}$	$[FTB: r_4, r_6, 31]$
(33)	$\mathbf{F}f$	$[FFA: r_8, 30]$
(34)	$\mathbf{T}\{not\ a, not\ f\}$	$[FTB: r_9, 17, 33]$

$\mathcal{T}_{nomore^{++}}$: 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)	$\mathbf{T}a$	$[Cut]$
(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)	$\mathbf{T}\{g\}$	$[FTB\text{: } r_4, r_6, 31]$
(33)	$\mathbf{F}f$	$[FFA\text{: } r_8, 30]$
(34)	$\mathbf{T}\{not\ a, not\ f\}$	$[FTB\text{: } r_9, 16, 33]$

Conflict-Driven Answer Set Solving
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

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

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

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

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

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

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

Assignments

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

$$(\sigma_1, \ldots, \sigma_n)$$

  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 $A = (\sigma_1, \ldots, \sigma_{k-1}, \sigma_k, \ldots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \ldots, \sigma_{k-1})$.
- We sometimes identify an assignment with the set of its literals. Given this, we access *true* and *false* propositions in A via

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

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

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

Assignments

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

  $$(\sigma_1, \ldots, \sigma_n)$$

  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 $A = (\sigma_1, \ldots, \sigma_{k-1}, \sigma_k, \ldots, \sigma_n)$, we let $A[\sigma_k] = (\sigma_1, \ldots, \sigma_{k-1})$.
- We sometimes identify an assignment with the set of its literals. Given this, we access *true* and *false* propositions in A via

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

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.

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 $A^{\mathbf{T}} \cup A^{\mathbf{F}} = \mathit{dom}(A)$ 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.

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.

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.

Nogoods from logic programs

via Clark's completion

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

$$\begin{aligned} & \{p_\beta \leftrightarrow p_1 \wedge \dots \wedge p_m \wedge \neg p_{m+1} \wedge \dots \wedge \neg p_n \mid \\ & \qquad \beta \in \mathit{body}(\Pi), \beta = \{p_1, \dots, p_m, \mathit{not}\ p_{m+1}, \dots, \mathit{not}\ p_n\}\} \\ \cup\ & \{p \leftrightarrow p_{\beta_1} \vee \dots \vee p_{\beta_k} \mid \\ & \qquad p \in \mathit{atom}(\Pi), \mathit{body}(p) = \{\beta_1, \dots, \beta_k\}\}\ , \end{aligned}$$

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

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

can be decomposed into two implications.

1. We get

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

which is equivalent to the conjunction of

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

This set of clauses expresses the following set of nogoods:

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

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

Let $\beta = \{p_1, \dots, p_m, \mathit{not}\ p_{m+1}, \dots, \mathit{not}\ p_n\}$ 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.

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

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

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

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

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

can be decomposed into two implications.

2. The converse of the previous implication, viz.

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

gives rise to the nogood

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

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 $body(p) = \{\beta_1, \ldots, \beta_k\}$, recall that

$$\begin{array}{rcl} \delta(p) & = & \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\} \\ \Delta(p) & = & \{\,\{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\,\}\ . \end{array}$$

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

$$\boxed{\begin{array}{lcl} x & \leftarrow & y \\ x & \leftarrow & not\ z \end{array}}$$

$$\begin{array}{rcl} \delta(x) & = & \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\} \\ \Delta(x) & = & \{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\}\,\} \end{array}$$

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

atom-oriented nogoods

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

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

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

$$\boxed{\begin{array}{lcl} x & \leftarrow & y \\ x & \leftarrow & not\ z \end{array}}$$

$$\begin{array}{rcl} \delta(x) & = & \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\} \\ \Delta(x) & = & \{\, \{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\} \,\} \end{array}$$

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

atom-oriented nogoods

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

$$\begin{array}{rcl} \delta(p) & = & \{\mathbf{T}p, \mathbf{F}\beta_1, \ldots, \mathbf{F}\beta_k\} \\ \Delta(p) & = & \{\,\{\mathbf{F}p, \mathbf{T}\beta_1\}, \ldots, \{\mathbf{F}p, \mathbf{T}\beta_k\}\,\}\ . \end{array}$$

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

$$\boxed{\begin{array}{lcl} x & \leftarrow & y \\ x & \leftarrow & not\ z \end{array}}$$

$$\begin{array}{rcl} \delta(x) & = & \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\} \\ \Delta(x) & = & \{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\}\,\} \end{array}$$

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

atom-oriented nogoods

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

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

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

$$\boxed{\begin{array}{lcl} x & \leftarrow & y \\ x & \leftarrow & \mathit{not}\ z \end{array}}$$

$$\begin{array}{rcl}\delta(x) & = & \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{\mathit{not}\ z\}\} \\ \Delta(x) & = & \{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{\mathit{not}\ z\}\}\,\}\end{array}$$

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

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

Nogoods from logic programs

atom-oriented nogoods

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

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

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

$$\boxed{\begin{array}{lcl} x & \leftarrow & y \\ x & \leftarrow & not\ z \end{array}}$$

$$\begin{array}{rcl} \delta(x) & = & \{\mathbf{T}x, \mathbf{F}\{y\}, \mathbf{F}\{not\ z\}\} \\ \Delta(x) & = & \{\,\{\mathbf{F}x, \mathbf{T}\{y\}\}, \{\mathbf{F}x, \mathbf{T}\{not\ z\}\}\,\} \end{array}$$

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

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

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

$$\boxed{\begin{array}{l} \dots \leftarrow x, not\ y \\ \qquad\vdots \\ \dots \leftarrow x, not\ y \end{array}}$$

$$\begin{aligned}\delta(\{x, not\ y\}) &= \{\mathbf{F}\{x, not\ y\}, \mathbf{T}x, \mathbf{F}y\}\\ \Delta(\{x, not\ y\}) &= \{\,\{\mathbf{T}\{x, not\ y\}, \mathbf{F}x\}, \{\mathbf{T}\{x, not\ y\}, \mathbf{T}y\}\,\}\end{aligned}$$

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

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

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

$$\boxed{\begin{array}{c}\dots \leftarrow x, \mathit{not}\ y \\ \vdots \\ \dots \leftarrow x, \mathit{not}\ y\end{array}}$$

$$\begin{array}{l}\delta(\{x, \mathit{not}\ y\}) = \{\mathbf{F}\{x, \mathit{not}\ y\}, \mathbf{T}x, \mathbf{F}y\} \\ \Delta(\{x, \mathit{not}\ y\}) = \{\,\{\mathbf{T}\{x, \mathit{not}\ y\}, \mathbf{F}x\}, \{\mathbf{T}\{x, \mathit{not}\ y\}, \mathbf{T}y\}\,\}\end{array}$$

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 $(\mathbf{F}\{x, \mathit{not}\ y\}, \mathbf{T}x)$.

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

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

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

$$\boxed{\begin{array}{c} \ldots \leftarrow x, \mathit{not}\ y \\ \vdots \\ \ldots \leftarrow x, \mathit{not}\ y \end{array}}$$

$$\begin{aligned}
\delta(\{x, \mathit{not}\ y\}) &= \{\mathbf{F}\{x, \mathit{not}\ y\}, \mathbf{T}x, \mathbf{F}y\} \\
\Delta(\{x, \mathit{not}\ y\}) &= \{\, \{\mathbf{T}\{x, \mathit{not}\ y\}, \mathbf{F}x\}, \{\mathbf{T}\{x, \mathit{not}\ y\}, \mathbf{T}y\} \,\}
\end{aligned}$$

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 $(\mathbf{F}\{x, \mathit{not}\ y\}, \mathbf{T}x)$.

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

$$\boxed{\begin{array}{c} \dots \leftarrow x, \mathit{not}\ y \\ \vdots \\ \dots \leftarrow x, \mathit{not}\ y \end{array}}$$

$$\begin{array}{r}\delta(\{x, \mathit{not}\ y\}) = \{\mathbf{F}\{x, \mathit{not}\ y\}, \mathbf{T}x, \mathbf{F}y\} \\ \Delta(\{x, \mathit{not}\ y\}) = \{\, \{\mathbf{T}\{x, \mathit{not}\ y\}, \mathbf{F}x\}, \{\mathbf{T}\{x, \mathit{not}\ y\}, \mathbf{T}y\} \,\}\end{array}$$

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 $(\mathbf{F}\{x, \mathit{not}\ y\}, \mathbf{T}x)$.

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

$$\boxed{\begin{array}{c}\dots \leftarrow x, \mathit{not}\ y \\ \vdots \\ \dots \leftarrow x, \mathit{not}\ y\end{array}}$$

$$\begin{array}{r}\delta(\{x, \mathit{not}\ y\}) = \{\mathbf{F}\{x, \mathit{not}\ y\}, \mathbf{T}x, \mathbf{F}y\} \\ \Delta(\{x, \mathit{not}\ y\}) = \{\,\{\mathbf{T}\{x, \mathit{not}\ y\}, \mathbf{F}x\}, \{\mathbf{T}\{x, \mathit{not}\ y\}, \mathbf{T}y\}\,\}\end{array}$$

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.

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.

Characterization of answer sets

for tight logic programs

Let Π be a logic program and

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

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: $\quad p \leftrightarrow p_{\beta_1} \vee \cdots \vee p_{\beta_k}$
- 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:

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

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:

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

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:

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

- 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:

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

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: $p \leftrightarrow p_{\beta_1} \vee \dots \vee p_{\beta_k}$
- 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:

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

- 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:

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

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:

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

- 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:

$$\frac{p \leftarrow l_1, \dots, l_i, \dots, l_n \quad \mathbf{f}l_i}{\mathbf{F}\{l_1, \dots, l_i, \dots, l_n\}} \qquad \frac{\mathbf{T}\{l_1, \dots, l_i, \dots, l_n\}}{\mathbf{t}l_i}$$

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

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

Body-oriented nogoods and tableau rules

- Tableau rules FTB, BFB, FFB, and BTB are body-oriented.
- For a body $\beta = \{p_1, \dots, p_m, \mathit{not}\ p_{m+1}, \dots, \mathit{not}\ p_n\} = \{l_1, \dots, l_n\}$, consider the equivalence: $p_\beta \leftrightarrow p_1 \wedge \dots \wedge p_m \wedge \neg p_{m+1} \wedge \dots \wedge \neg p_n$
- 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:

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

- 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:

$$\frac{p \leftarrow l_1, \dots, l_i, \dots, l_n \quad \mathbf{f}l_i}{\mathbf{F}\{l_1, \dots, l_i, \dots, l_n\}} \qquad \frac{\mathbf{T}\{l_1, \dots, l_i, \dots, l_n\}}{\mathbf{t}l_i}$$

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: $p_\beta \leftrightarrow p_1 \wedge \dots \wedge p_m \wedge \neg p_{m+1} \wedge \dots \wedge \neg p_n$
- 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 422)

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

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

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

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.

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

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

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

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.

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

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

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

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

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

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:

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

Example

Consider

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

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:

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

Example

Consider

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

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

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

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

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

Characterization of answer sets

For a logic program Π,
let Δ_Π and Λ_Π as defined on Page 582 and Page 594, 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 !

Characterization of answer sets

For a logic program Π,
let Δ_Π and Λ_Π as defined on Page 582 and Page 594, 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 !

Characterization of answer sets

For a logic program Π,
let Δ_Π and Λ_Π as defined on Page 582 and Page 594, 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 !

Characterization of answer sets

For a logic program Π,
let Δ_Π and Λ_Π as defined on Page 582 and Page 594, 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 !

Characterization of answer sets

For a logic program Π,
let Δ_Π and Λ_Π as defined on Page 582 and Page 594, 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 !

Characterization of answer sets

For a logic program Π,
let Δ_Π and Λ_Π as defined on Page 582 and Page 594, 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 !

Conflict-driven search

Boolean constraint solving algorithms pioneered for SAT led to:

Traditional approach

- (Unit) propagation
- Exhaustive (chronological) backtracking

☞ DPLL [17, 16]

State of the art

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

☞ CDCL [78, 62]

Idea

➥ Apply CDCL-style search in ASP solving !

Conflict-driven search

Boolean constraint solving algorithms pioneered for SAT led to:

Traditional approach

- (Unit) propagation
- Exhaustive (chronological) backtracking

☞ DPLL [17, 16]

State of the art

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

☞ CDCL [78, 62]

Idea

➥ Apply CDCL-style search in ASP solving !

Conflict-driven search

Boolean constraint solving algorithms pioneered for SAT led to:

Traditional approach

- (Unit) propagation
- Exhaustive (chronological) backtracking

☞ DPLL [17, 16]

State of the art

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

☞ CDCL [78, 62]

Idea

➥ Apply CDCL-style search in ASP solving !

Outline of CDNL-ASP algorithm

- Keep track of deterministic consequences by unit propagation on:
  - Clark's completion [Δ_Π]
  - Loop nogoods, determined and recorded on demand [Λ_Π]
    - ☞ Dedicated unfounded set detection !
  - Dynamic nogoods, derived from conflicts and unfounded sets [∇]
- When a nogood in $\Delta_\Pi \cup \nabla$ becomes violated:
  - Analyze the conflict by resolution until reaching the First Unique Implication Point (First-UIP) [63]
  - 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

Outline of CDNL-ASP algorithm

- Keep track of deterministic consequences by unit propagation on:
  - Clark's completion [Δ_Π]
  - Loop nogoods, determined and recorded on demand [Λ_Π]
    - ☞ Dedicated unfounded set detection !
  - Dynamic nogoods, derived from conflicts and unfounded sets [∇]
- When a nogood in $\Delta_\Pi \cup \nabla$ becomes violated:
  - Analyze the conflict by resolution until reaching the First Unique Implication Point (First-UIP) [63]
  - 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

Outline of CDNL-ASP algorithm

- Keep track of deterministic consequences by unit propagation on:
  - Clark's completion [Δ_Π]
  - Loop nogoods, determined and recorded on demand [Λ_Π]
    - ☞ Dedicated unfounded set detection !
  - Dynamic nogoods, derived from conflicts and unfounded sets [∇]
- When a nogood in $\Delta_\Pi \cup \nabla$ becomes violated:
  - Analyze the conflict by resolution until reaching the First Unique Implication Point (First-UIP) [63]
  - 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 !

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 !

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

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

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	$\mathbf{F}\{not\ y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$ ✘

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: 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	$\mathbf{T}u$		
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	$\mathbf{F}\{\mathit{not}\ y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\mathit{not}\ y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$ ✘

Example: CDNL-ASP

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	$\mathbf{F}\{not\ y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$ ✘

Example: 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	$\mathbf{T}u$		
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	$\mathbf{F}\{\mathit{not}\ y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\mathit{not}\ y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$ ✘

Example: 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	$\mathbf{T}u$		
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	$\mathbf{F}\{\mathit{not}\ y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\mathit{not}\ y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$ ✘

Example: CDNL-ASP

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	$\mathbf{F}\{not\ y\}$		
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$
		$\vdots$	$\vdots$
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$ ✘

Example (ctd): CDNL-ASP

Consider

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

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

Example (ctd): CDNL-ASP

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	$\mathbf{T}u$		
		$\mathbf{T}x$	$\{\mathbf{T}u, \mathbf{F}x\} \in \nabla$
		$\vdots$	$\vdots$
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{x\}\} \in \Delta(v)$
		$\mathbf{F}y$	$\{\mathbf{T}y, \mathbf{F}\{not\ x\}\} = \delta(y)$
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$

Example (ctd): CDNL-ASP

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	$\mathbf{T}u$		
		$\mathbf{T}x$	$\{\mathbf{T}u, \mathbf{F}x\} \in \nabla$
		$\vdots$	$\vdots$
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{x\}\} \in \Delta(v)$
		$\mathbf{F}y$	$\{\mathbf{T}y, \mathbf{F}\{not\ x\}\} = \delta(y)$
		$\mathbf{F}w$	$\{\mathbf{T}w, \mathbf{F}\{not\ x, not\ y\}\} = \delta(w)$

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 $p \in U$.

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 $p \in U$.

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 $p \in U$.

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 UNFOUNDEDSET

- Implementations of 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 !

Requirements for UNFOUNDEDSET

- Implementations of 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 !

Requirements for UNFOUNDEDSET

- Implementations of 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

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

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

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.

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*.

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 σ ∈ δ such that (δ \ A[σ]) = {σ} in
        k ← max({dl(ρ) | ρ ∈ δ \ {σ}} ∪ {0})
        if k = dl(σ) then
            let ε ∈ Δ_Π ∪ ∇ such that (ε \ A[σ]) = {σ̄} in
                δ ← (δ \ {σ}) ∪ (ε \ {σ̄})                    // 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	$\mathbf{T}u$			
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	$\mathbf{F}\{\mathit{not}\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\mathit{not}\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\mathit{not}\ x\}$	$\{\mathbf{F}\{\mathit{not}\ x\}, \mathbf{F}x\} = \delta(\{\mathit{not}\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\mathit{not}\ y\}, \mathbf{F}y\} = \delta(\{\mathit{not}\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

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	$\mathbf{F}\{not\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{not\ x\}$	$\{\mathbf{F}\{not\ x\}, \mathbf{F}x\} = \delta(\{not\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{not\ y\}, \mathbf{F}y\} = \delta(\{not\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

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	$\mathbf{T}u$			
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	$\mathbf{F}\{\mathit{not}\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\mathit{not}\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\mathit{not}\ x\}$	$\{\mathbf{F}\{\mathit{not}\ x\}, \mathbf{F}x\} = \delta(\{\mathit{not}\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\mathit{not}\ y\}, \mathbf{F}y\} = \delta(\{\mathit{not}\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

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	$\mathbf{F}\{not\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{not\ x\}$	$\{\mathbf{F}\{not\ x\}, \mathbf{F}x\} = \delta(\{not\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{not\ y\}, \mathbf{F}y\} = \delta(\{not\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

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	$\mathbf{T}u$			
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	$\mathbf{F}\{\mathit{not}\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\mathit{not}\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\mathit{not}\ x\}$	$\{\mathbf{F}\{\mathit{not}\ x\}, \mathbf{F}x\} = \delta(\{\mathit{not}\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\mathit{not}\ y\}, \mathbf{F}y\} = \delta(\{\mathit{not}\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

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	$\mathbf{F}\{not\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{not\ x\}$	$\{\mathbf{F}\{not\ x\}, \mathbf{F}x\} = \delta(\{not\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{not\ y\}, \mathbf{F}y\} = \delta(\{not\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

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	$\mathbf{T}u$			
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	$\mathbf{F}\{\mathit{not}\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\mathit{not}\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\mathit{not}\ x\}$	$\{\mathbf{F}\{\mathit{not}\ x\}, \mathbf{F}x\} = \delta(\{\mathit{not}\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\mathit{not}\ y\}, \mathbf{F}y\} = \delta(\{\mathit{not}\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

Example: ConflictAnalysis

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	$\mathbf{F}\{not\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{not\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{not\ x\}$	$\{\mathbf{F}\{not\ x\}, \mathbf{F}x\} = \delta(\{not\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{not\ y\}, \mathbf{F}y\} = \delta(\{not\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

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

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	$\mathbf{T}u$			
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	$\mathbf{F}\{\mathit{not}\ y\}$			
		$\mathbf{F}x$	$\{\mathbf{T}x, \mathbf{F}\{\mathit{not}\ y\}\} = \delta(x)$	
		$\mathbf{F}\{x\}$	$\{\mathbf{T}\{x\}, \mathbf{F}x\} \in \Delta(\{x\})$	$\{\mathbf{T}u, \mathbf{F}x\}$
		$\mathbf{F}\{x, y\}$	$\{\mathbf{T}\{x, y\}, \mathbf{F}x\} \in \Delta(\{x, y\})$	$\{\mathbf{T}u, \mathbf{F}x, \mathbf{F}\{x\}\}$
		$\mathbf{T}\{\mathit{not}\ x\}$	$\{\mathbf{F}\{\mathit{not}\ x\}, \mathbf{F}x\} = \delta(\{\mathit{not}\ x\})$	
		$\mathbf{T}y$	$\{\mathbf{F}\{\mathit{not}\ y\}, \mathbf{F}y\} = \delta(\{\mathit{not}\ y\})$	
		$\mathbf{T}\{v\}$	$\{\mathbf{T}u, \mathbf{F}\{x, y\}, \mathbf{F}\{v\}\} = \delta(u)$	
		$\mathbf{T}\{u, y\}$	$\{\mathbf{F}\{u, y\}, \mathbf{T}u, \mathbf{T}y\} = \delta(\{u, y\})$	
		$\mathbf{T}v$	$\{\mathbf{F}v, \mathbf{T}\{u, y\}\} \in \Delta(v)$	
			$\{\mathbf{T}u, \mathbf{F}\{x\}, \mathbf{F}\{x, y\}\} = \lambda(u, \{u, v\})$	✘

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 !

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 !

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 !

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 !

The `clasp` system

- 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 !

The `clasp` system

- 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`

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`

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

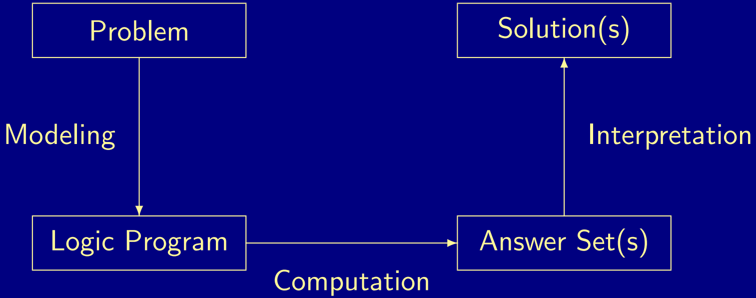
Front-ends also admit `clasp` to solve:

- Propositional CNF formulas
- Pseudo-Boolean formulas

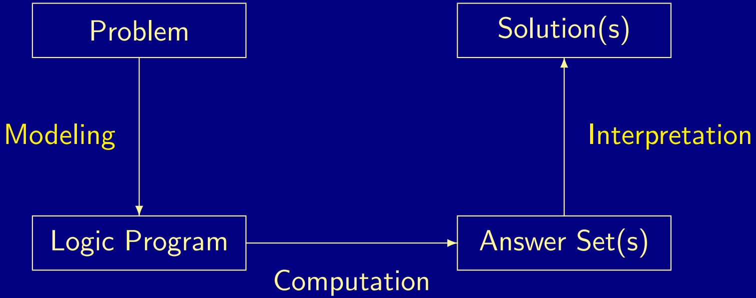
Find `clasp` at: `http://potassco.sourceforge.net`

Effective Modeling
Overview

Modeling and Interpreting



Modeling and Interpreting



Problem $\longmapsto$ Logic Program

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 $C(I) \cup C(P)$.

☞ A uniform encoding C(P) is a first-order logic program, encoding the solutions to P for any set C(I) of facts.

Problem $\longmapsto$ Logic Program

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 $C(I) \cup C(P)$.

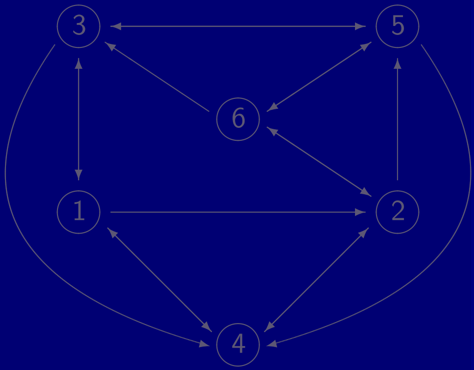
☞ A uniform encoding C(P) is a first-order logic program, encoding the solutions to P for any set C(I) of facts.

N-Colorability

Problem Instance as Facts

Given: a (directed) graph G

$$G = \begin{pmatrix} V = \{1,2,3,4,5,6\}, & \\ E = \{(1,2),(1,3),(1,4), & \\ \quad (2,4),(2,5),(2,6), & \\ \quad (3,1),(3,4),(3,5), & \\ \quad (4,1),(4,2), & \\ \quad (5,3),(5,4),(5,6), & \\ \quad (6,2),(6,3),(6,5)\} & \end{pmatrix}$$

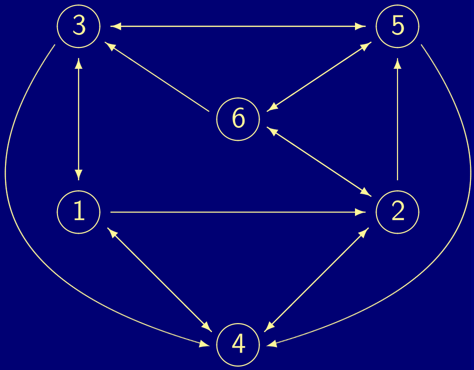


N-Colorability

Problem Instance as Facts

Given: a (directed) graph G

$$G = \begin{pmatrix} V = \{1,2,3,4,5,6\}, \\ E = \{(1,2),(1,3),(1,4), \\ \quad (2,4),(2,5),(2,6), \\ \quad (3,1),(3,4),(3,5), \\ \quad (4,1),(4,2), \\ \quad (5,3),(5,4),(5,6), \\ \quad (6,2),(6,3),(6,5)\} \end{pmatrix}$$

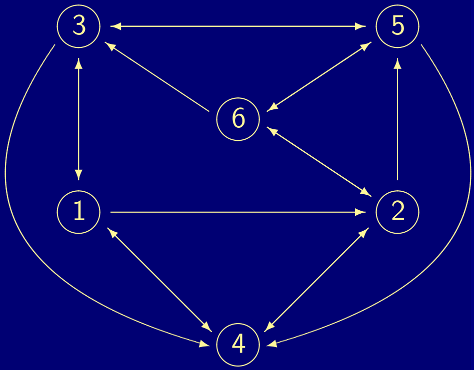


N-Colorability

Problem Instance as Facts

Given: a (directed) graph G

```
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).
```

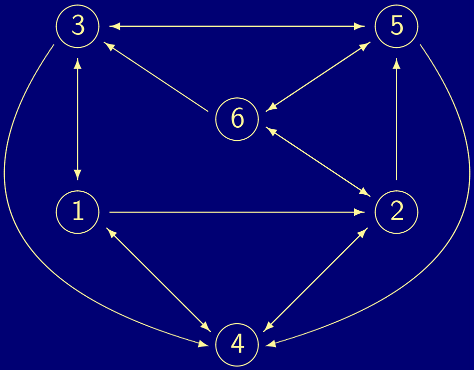


N-Colorability

Problem Instance as Facts

Given: a (directed) graph G

```
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).
```



N-Colorability

(Extended) Problem Encoding

Natural Language

1. Each node has a unique color.
2. Any two connected nodes must not have the same color.
3. Let there be three colors.
4. A solution is a coloring.

Logical Language

1.
```
color(X,C) :- iscol(C),
node(X), not other(X,C).

other(X,C) :- iscol(C),
color(X,D), D != C.
```
2.
```
:- color(X,C), color(Y,C),
   edge(X,Y).
```
3.
```
#const n=3.
iscol(1..n).
```
4.
```
#hide.
#show color/2.
```

N-Colorability

(Extended) Problem Encoding

Natural Language

1. Each node has a unique color.
2. Any two connected nodes must not have the same color.
3. Let there be three colors.
4. A solution is a coloring.

Logical Language

1.
```
color(X,C) :- iscol(C),
node(X), not other(X,C).

other(X,C) :- iscol(C),
color(X,D), D != C.
```
2.
```
:- color(X,C), color(Y,C),
   edge(X,Y).
```
3.
```
#const n=3.
iscol(1..n).
```
4.
```
#hide.
#show color/2.
```

N-Colorability

(Extended) Problem Encoding

Natural Language	Logical Language
1 Each node has a unique color.	1 `color(X,C) :- iscol(C), node(X), not other(X,C).` `other(X,C) :- iscol(C), color(X,D), D != C.`
2 Any two connected nodes must not have the same color.	2 `:- color(X,C), color(Y,C), edge(X,Y).`
3 Let there be three colors.	3 `#const n=3.` `iscol(1..n).`
4 A solution is a coloring.	4 `#hide.` `#show color/2.`

N-Colorability

(Extended) Problem Encoding

Natural Language	Logical Language
1 Each node has a unique color.	1 `color(X,C) :- iscol(C), node(X), not other(X,C).` `other(X,C) :- iscol(C), color(X,D), D != C.`
2 Any two connected nodes must not have the same color.	2 `:- color(X,C), color(Y,C), edge(X,Y).`
3 Let there be three colors.	3 `#const n=3.` `iscol(1..n).`
4 A solution is a coloring.	4 `#hide.` `#show color/2.`

N-Colorability

(Extended) Problem Encoding

Natural Language	Logical Language
1 Each node has a unique color.	1 `color(X,C) :- iscol(C), node(X), not other(X,C).` `other(X,C) :- iscol(C), color(X,D), D != C.`
2 Any two connected nodes must not have the same color.	2 `:- color(X,C), color(Y,C), edge(X,Y).`
3 Let there be three colors.	3 `#const n=3.` `iscol(1..n).`
4 A solution is a coloring.	4 `#hide.` `#show color/2.`

N-Colorability

(Extended) Problem Encoding

Natural Language	Logical Language
1 Each node has a unique color.	1 `color(X,C) :- iscol(C), node(X), not other(X,C).` `other(X,C) :- iscol(C), color(X,D), D != C.`
2 Any two connected nodes must not have the same color.	2 `:- color(X,C), color(Y,C), edge(X,Y).`
3 Let there be three colors.	3 `#const n=3.` `iscol(1..n).`
4 A solution is a coloring.	4 `#hide.` `#show color/2.`

N-Colorability

(Extended) Problem Encoding

Natural Language

1. Each node has a unique color.
2. Any two connected nodes must not have the same color.
3. Let there be three colors.
4. A solution is a coloring.

Logical Language

1.
```
color(X,C) :- iscol(C),
node(X), not other(X,C).

other(X,C) :- iscol(C),
color(X,D), D != C.
```
2.
```
:- color(X,C), color(Y,C),
   edge(X,Y).
```
3.
```
#const n=3.
iscol(1..n).
```
4.
```
#hide.
#show color/2.
```

N-Colorability

(Extended) Problem Encoding

Natural Language

1. Each node has a unique color.
2. Any two connected nodes must not have the same color.
3. Let there be three colors.
4. A solution is a coloring.

Logical Language

1.
```
1 #count{ color(X,C) :
                  iscol(C) } 1
              :- node(X).
```
2.
```
:- color(X,C), color(Y,C),
   edge(X,Y).
```
3.
```
#const n=3.
iscol(1..n).
```
4.
```
#hide.
#show color/2.
```

N-Colorability

Recapitulation I

Instance as Facts (in `graph.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).
```

N-Colorability

Recapitulation II

Uniform Encoding (in `color.lp`)

```
% DOMAIN
#const n=3. iscol(1..n).

% GENERATE
1 #count{ color(X,C) : iscol(C) } 1 :- node(X).
% color(X,C) :- iscol(C), node(X), not other(X,C).
% other(X,C) :- iscol(C), color(X,D), D != C.

% TEST
:- color(X,C), color(Y,C), edge(X,Y).

% DISPLAY
#hide. #show color/2.
```

N-Colorability

Let's Run it!

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

```
clasp version 2.0.2
Reading from stdin
Solving...
Answer: 1
color(6,2) color(5,3) color(4,2) color(3,1) color(2,1) color(1,3)
Answer: 2
color(6,1) color(5,3) color(4,1) color(3,2) color(2,2) color(1,3)
Answer: 3
color(6,3) color(5,2) color(4,3) color(3,1) color(2,1) color(1,2)
Answer: 4
color(6,1) color(5,2) color(4,1) color(3,3) color(2,3) color(1,2)
Answer: 5
color(6,3) color(5,1) color(4,3) color(3,2) color(2,2) color(1,1)
Answer: 6
color(6,2) color(5,1) color(4,2) color(3,3) color(2,3) color(1,1)
```

N-Colorability

Let's Run it!

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

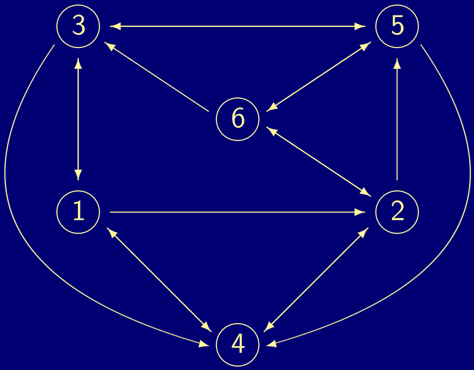
```
clasp version 2.0.2
Reading from stdin
Solving...
Answer: 1
color(6,2) color(5,3) color(4,2) color(3,1) color(2,1) color(1,3)
Answer: 2
color(6,1) color(5,3) color(4,1) color(3,2) color(2,2) color(1,3)
Answer: 3
color(6,3) color(5,2) color(4,3) color(3,1) color(2,1) color(1,2)
Answer: 4
color(6,1) color(5,2) color(4,1) color(3,3) color(2,3) color(1,2)
Answer: 5
color(6,3) color(5,1) color(4,3) color(3,2) color(2,2) color(1,1)
Answer: 6
color(6,2) color(5,1) color(4,2) color(3,3) color(2,3) color(1,1)
```

N-Colorability

Let's Interpret it!

Found: 3-coloring(s)

```
Answer: 1

color(1,3) color(5,3)

color(2,1) color(3,1)

color(4,2) color(6,2)
```

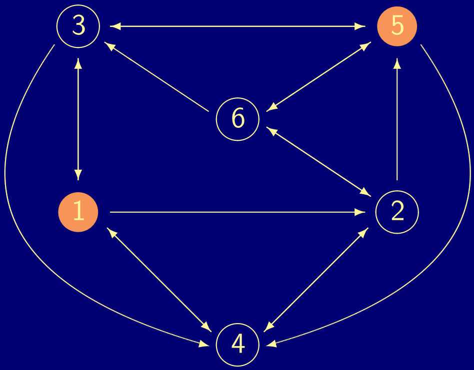


N-Colorability

Let's Interpret it!

Found: 3-coloring(s)

```
Answer: 1

color(1,3) color(5,3)

color(2,1) color(3,1)

color(4,2) color(6,2)
```

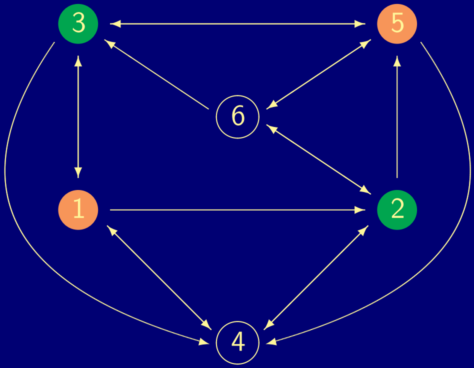


N-Colorability

Let's Interpret it!

Found: 3-coloring(s)

```
Answer: 1

color(1,3) color(5,3)

color(2,1) color(3,1)

color(4,2) color(6,2)
```

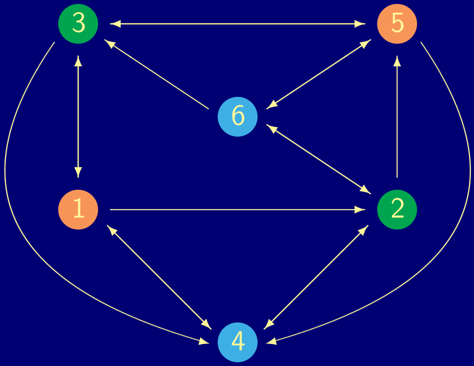


N-Colorability

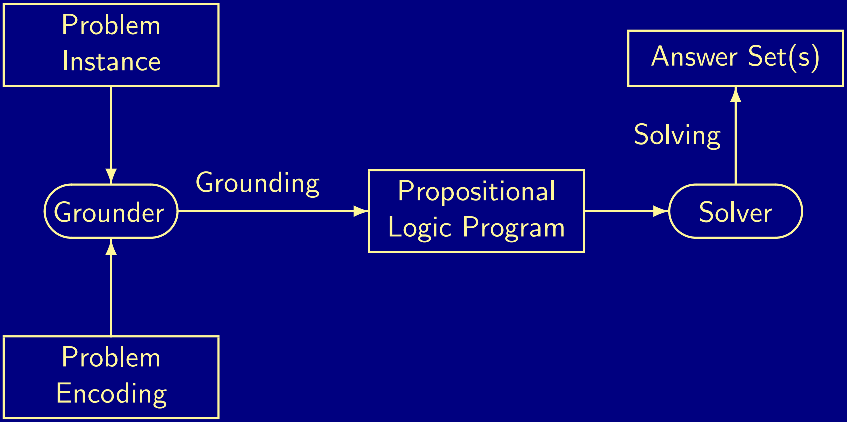
Let's Interpret it!

Found: 3-coloring(s)

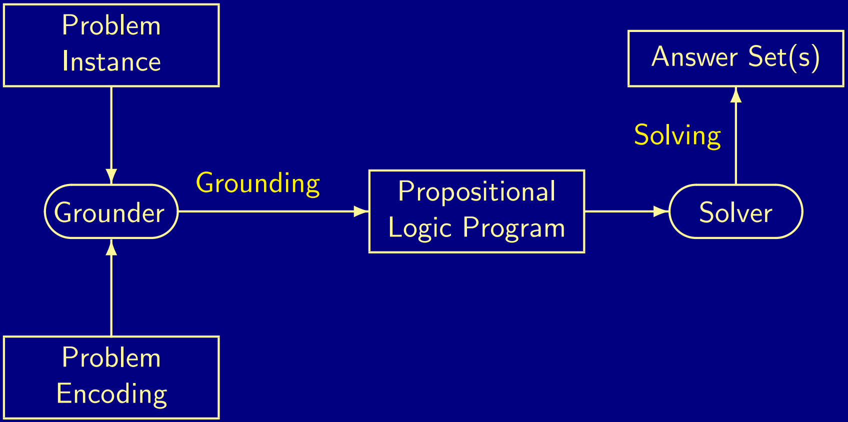
```
Answer: 1

color(1,3) color(5,3)

color(2,1) color(3,1)

color(4,2) color(6,2)
```



Interlude: Answer Set(s) Computation



Interlude: Answer Set(s) Computation



N-Colorability

Grounding

```
gringo -t graph.lp 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). ...

iscol(1).  iscol(2).  iscol(3).

1 #count{ color(1,1), color(1,2), color(1,3) } 1.
1 #count{ color(2,1), color(2,2), color(2,3) } 1.
1 #count{ color(3,1), color(3,2), color(3,3) } 1.
1 #count{ color(4,1), color(4,2), color(4,3) } 1.
1 #count{ color(5,1), color(5,2), color(5,3) } 1.
1 #count{ color(6,1), color(6,2), color(6,3) } 1.

:- color(1,1), color(2,1).
:- color(1,2), color(2,2).
:- color(1,3), color(2,3). ...
```

N-Colorability

Grounding

```
gringo -t graph.lp 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). ...

iscol(1).  iscol(2).  iscol(3).

1 #count{ color(1,1), color(1,2), color(1,3) } 1.
1 #count{ color(2,1), color(2,2), color(2,3) } 1.
1 #count{ color(3,1), color(3,2), color(3,3) } 1.
1 #count{ color(4,1), color(4,2), color(4,3) } 1.
1 #count{ color(5,1), color(5,2), color(5,3) } 1.
1 #count{ color(6,1), color(6,2), color(6,3) } 1.

:- color(1,1), color(2,1).
:- color(1,2), color(2,2).
:- color(1,3), color(2,3). ...
```

N-Colorability

Solving

```
gringo graph.lp color.lp | clasp --stats 0
```

```
...
Models       : 6
Time         : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
Choices      : 5
Conflicts    : 0
Restarts     : 0

Atoms        : 63
Rules        : 113    (1: 95 2: 12 3: 6)
Bodies       : 64
Equivalences: 106    (Atom=Atom: 31 Body=Body: 6 Other: 69)
Tight        : Yes

Variables    : 63     (Eliminated: 0 Frozen:   30)
Constraints  : 45     (Binary: 73.3% Ternary:  0.0% Other: 26.7%)
Lemmas       : 0      (Binary:  0.0% Ternary:  0.0% Other:  0.0%)
```

N-Colorability

Solving

```
gringo graph.lp color.lp | clasp --stats 0
```

```
...
Models       : 6
Time         : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
Choices      : 5
Conflicts    : 0
Restarts     : 0

Atoms        : 63
Rules        : 113    (1: 95 2: 12 3: 6)
Bodies       : 64
Equivalences: 106    (Atom=Atom: 31 Body=Body: 6 Other: 69)
Tight        : Yes

Variables    : 63     (Eliminated: 0 Frozen:   30)
Constraints : 45     (Binary: 73.3% Ternary:  0.0% Other: 26.7%)
Lemmas       : 0      (Binary:  0.0% Ternary:  0.0% Other:  0.0%)
```

N-Colorability

Solving

```
gringo graph.lp color.lp | clasp --stats 0
```

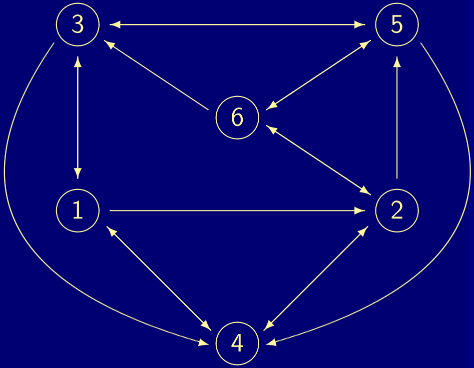
```
...
Models       : 6
Time         : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
Choices      : 5
Conflicts    : 0
Restarts     : 0

Atoms        : 63
Rules        : 113    (1: 95 2: 12 3: 6)
Bodies       : 64
Equivalences: 106    (Atom=Atom: 31 Body=Body: 6 Other: 69)
Tight        : Yes

Variables    : 63     (Eliminated: 0 Frozen:   30)
Constraints  : 45     (Binary: 73.3% Ternary:  0.0% Other: 26.7%)
Lemmas       : 0      (Binary:  0.0% Ternary:  0.0% Other:  0.0%)
```

Hamiltonian Cycle

Problem Instance as Facts

Recall: a directed graph G

```
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).
```



Hamiltonian Cycle

Engineering an Encoding

Problem Specification

A (directed) graph $G = (V, E)$ is Hamiltonian if it contains a cycle C that visits every node of V exactly once.

- ☞ C traverses exactly one incoming and one outgoing edge per node.
- ☞ C traverses every node of V (starting from an arbitrary node in V).

Problem Encoding

```
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).
```

Hamiltonian Cycle

Engineering an Encoding

Problem Specification

A (directed) graph $G = (V, E)$ is Hamiltonian if it contains a cycle C that visits every node of V exactly once.

- ☞ C traverses exactly one incoming and one outgoing edge per node.
- ☞ C traverses every node of V (starting from an arbitrary node in V).

Problem Encoding

```
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).
```

Hamiltonian Cycle

Engineering an Encoding

Problem Specification

A (directed) graph $G = (V, E)$ is Hamiltonian if it contains a cycle C that visits every node of V exactly once.

- ☞ C traverses exactly one incoming and one outgoing edge per node.
- ☞ C traverses every node of V (starting from an arbitrary node in V).

Problem Encoding

```
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).
```

Hamiltonian Cycle

Engineering an Encoding

Problem Specification

A (directed) graph $G = (V, E)$ is Hamiltonian if it contains a cycle C that visits every node of V exactly once.

- ☞ C traverses exactly one incoming and one outgoing edge per node.
- ☞ C traverses every node of V (starting from an arbitrary node in V).

Problem Encoding

```
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).
```

Hamiltonian Cycle

Engineering an Encoding

Problem Specification

A (directed) graph $G = (V, E)$ is Hamiltonian if it contains a cycle C that visits every node of V exactly once.

- ☞ C traverses exactly one incoming and one outgoing edge per node.
- ☞ C traverses every node of V (starting from an arbitrary node in V).

Problem Encoding

```
reach(X) :- first(X).
reach(Y) :- reach(X), cycle(X,Y).
```

Hamiltonian Cycle

Engineering an Encoding

Problem Specification

A (directed) graph $G = (V, E)$ is Hamiltonian if it contains a cycle C that visits every node of V exactly once.

- ☞ C traverses exactly one incoming and one outgoing edge per node.
- ☞ C traverses every node of V (starting from an arbitrary node in V).

Problem Encoding

```
reach(X) :- first(X).
reach(Y) :- reach(X), cycle(X,Y).
```

- ☞ The definition of `reach` is recursive!

Hamiltonian Cycle

Engineering an Encoding

Problem Specification

A (directed) graph $G = (V, E)$ is Hamiltonian if it contains a cycle C that visits every node of V exactly once.

- ☞ C traverses exactly one incoming and one outgoing edge per node.
- ☞ C traverses every node of V (starting from an arbitrary node in V).

Problem Encoding

```
reach(X) :- first(X).
reach(Y) :- reach(X), cycle(X,Y).

first(X) :- X = #min[ node(Y) = Y ].
```

Hamiltonian Cycle

Engineering an Encoding

Problem Specification

A (directed) graph $G = (V, E)$ is Hamiltonian if it contains a cycle C that visits every node of V exactly once.

- ☞ C traverses exactly one incoming and one outgoing edge per node.
- ☞ C traverses every node of V (starting from an arbitrary node in V).

Problem Encoding

```
reach(X) :- first(X).
reach(Y) :- reach(X), cycle(X,Y).

:- node(Y), not reach(Y).
```

Hamiltonian Cycle

The Complete Picture

Uniform Encoding (in `cycle.lp`)

```
% DOMAIN
first(X) :- X = #min[ node(Y) = Y ].

% GENERATE
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(X).
1 #count{ cycle(X,Y) : edge(X,Y) } 1 :- node(Y).

% DEFINE
reach(X) :- first(X).
reach(Y) :- reach(X), cycle(X,Y).

% TEST
:- node(Y), not reach(Y).

% DISPLAY
#hide. #show cycle/2.
```

Hamiltonian Cycle

Let's Run it!

```
gringo graph.lp cycle.lp | clasp --stats
```

```
Answer: 1
cycle(6,5) cycle(5,3) cycle(4,2) cycle(3,1) cycle(2,6) cycle(1,4)
SATISFIABLE

Models       : 1+
Time         : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
Choices      : 3
Conflicts    : 0
Restarts     : 0

Atoms        : 84
Rules        : 117    (1: 84 2: 21 3: 12)
Bodies       : 81
Equivalences: 174    (Atom=Atom: 36 Body=Body: 12 Other: 126)
Tight        : No     (SCCs: 1 Nodes: 20)
```

Hamiltonian Cycle

Let's Run it!

```
gringo graph.lp cycle.lp | clasp --stats
```

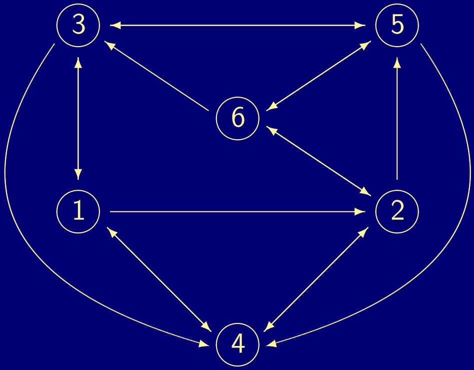
```
Answer: 1
cycle(6,5) cycle(5,3) cycle(4,2) cycle(3,1) cycle(2,6) cycle(1,4)
SATISFIABLE

Models       : 1+
Time         : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
Choices      : 3
Conflicts    : 0
Restarts     : 0

Atoms        : 84
Rules        : 117    (1: 84 2: 21 3: 12)
Bodies       : 81
Equivalences: 174    (Atom=Atom: 36 Body=Body: 12 Other: 126)
Tight        : No     (SCCs: 1 Nodes: 20)
```

Hamiltonian Cycle

Let's Interpret it!

Found: Hamiltonian cycle

```
Answer: 1

cycle(1,4)
cycle(4,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,1)
```

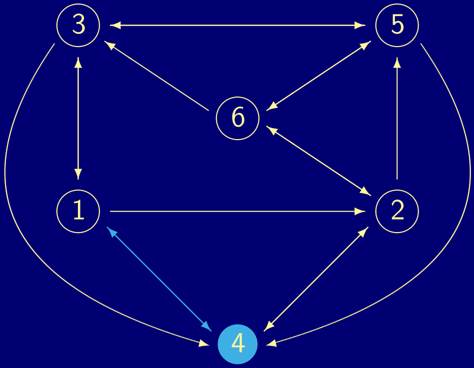


Hamiltonian Cycle

Let's Interpret it!

Found: Hamiltonian cycle

```
Answer: 1

cycle(1,4)
cycle(4,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,1)
```

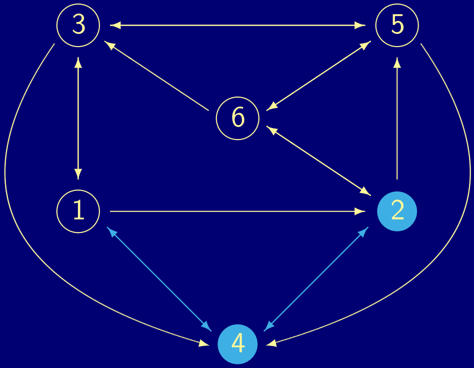


Hamiltonian Cycle

Let's Interpret it!

Found: Hamiltonian cycle

```
Answer: 1

cycle(1,4)
cycle(4,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,1)
```

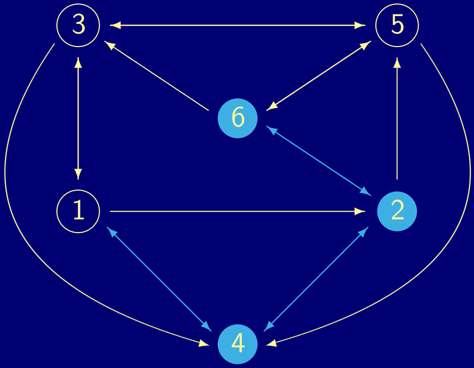


Hamiltonian Cycle

Let's Interpret it!

Found: Hamiltonian cycle

```
Answer: 1

cycle(1,4)
cycle(4,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,1)
```

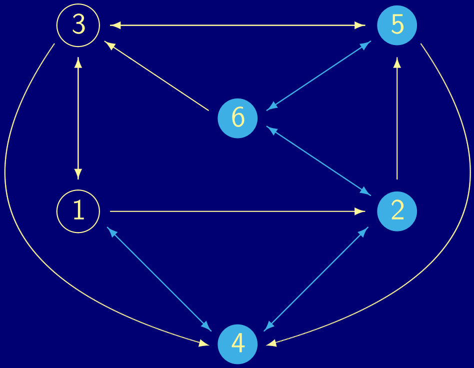


Hamiltonian Cycle

Let's Interpret it!

Found: Hamiltonian cycle

```
Answer: 1

cycle(1,4)
cycle(4,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,1)
```

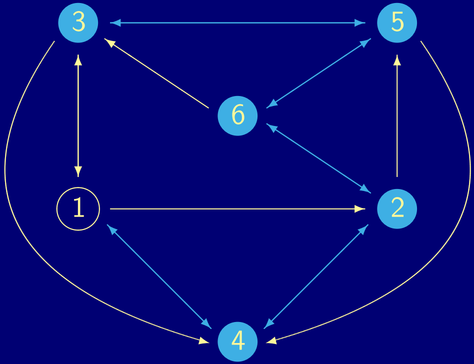


Hamiltonian Cycle

Let's Interpret it!

Found: Hamiltonian cycle

```
Answer: 1

cycle(1,4)
cycle(4,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,1)
```

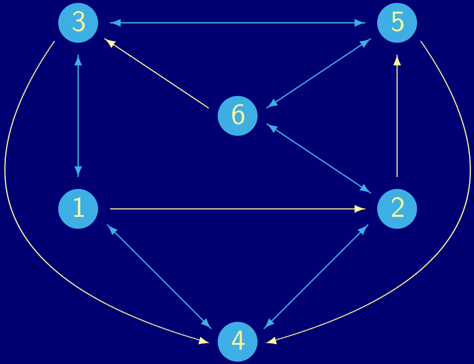


Hamiltonian Cycle

Let's Interpret it!

Found: Hamiltonian cycle

```
Answer: 1

cycle(1,4)
cycle(4,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,1)
```

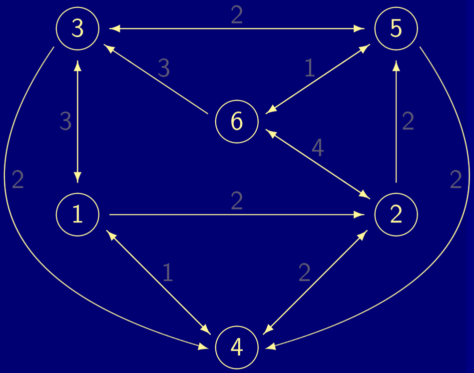


Mr Hamilton as Traveling Salesperson

Problem Instance as Facts

Given: a directed graph *G* plus edge costs

```
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).
```

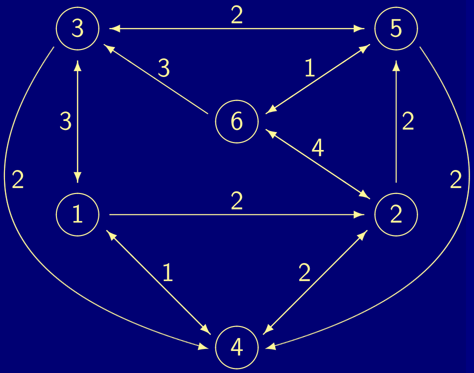


Mr Hamilton as Traveling Salesperson

Problem Instance as Facts

Given: a directed graph G plus edge costs

```
cost(1,2,2).
cost(1,3,3). cost(3,1,3).
cost(1,4,1). cost(4,1,1).
cost(2,4,2). cost(4,2,2).
cost(2,5,2).
cost(2,6,4). cost(6,2,4).
cost(3,4,2).
cost(3,5,2). cost(5,3,2).
cost(5,4,2).
cost(5,6,1). cost(6,5,1).
cost(6,3,3).
```



Mr Hamilton as Traveling Salesperson

Solution Optimization

Optimization Objective

A Hamiltonian cycle is optimal if its accumulated edge costs are minimal.

☞ Use `#minimize` (and/or `#maximize`) to associate each answer set with objective value(s).

Optimization Encoding

```
% OPTIMIZE
#minimize[ cycle(X,Y) : cost(X,Y,C) = C@1 ].
```

Target: minimal sum of costs `C` (at priority level `1`) associated with instances of `cycle` in an answer set

Mr Hamilton as Traveling Salesperson

Solution Optimization

Optimization Objective

A Hamiltonian cycle is optimal if its accumulated edge costs are minimal.

☞ Use `#minimize` (and/or `#maximize`) to associate each answer set with objective value(s).

Optimization Encoding

```
% OPTIMIZE
#minimize[ cycle(X,Y) : cost(X,Y,C) = C@1 ].
```

Target: minimal sum of costs `C` (at priority level `1`) associated with instances of `cycle` in an answer set

Mr Hamilton as Traveling Salesperson

Solution Optimization

Optimization Objective

A Hamiltonian cycle is optimal if its accumulated edge costs are minimal.

☞ Use `#minimize` (and/or `#maximize`) to associate each answer set with objective value(s).

Optimization Encoding

```
% OPTIMIZE
#minimize[ cycle(X,Y) : cost(X,Y,C) = C@1 ].
```

Target: minimal sum of costs `C` (at priority level 1) associated with instances of `cycle` in an answer set

Mr Hamilton as Traveling Salesperson

Solution Optimization

Optimization Objective

A Hamiltonian cycle is optimal if its accumulated edge costs are minimal.

☞ Use `#minimize` (and/or `#maximize`) to associate each answer set with objective value(s).

Optimization Encoding

```
% OPTIMIZE
#minimize[ cycle(X,Y) : cost(X,Y,C) = C@1 ].
```

Target: minimal sum of costs `C` (at priority level `1`) associated with instances of `cycle` in an answer set

Mr Hamilton as Traveling Salesperson

Solution Optimization

Optimization Objective

A Hamiltonian cycle is optimal if its accumulated edge costs are minimal.

☞ Use `#minimize` (and/or `#maximize`) to associate each answer set with objective value(s).

Optimization Encoding

```
% OPTIMIZE
#minimize[ cycle(X,Y) : cost(X,Y,C) = C@1 ].
```

Target: minimal sum of costs `C` (at priority level `1`) associated with instances of `cycle` in an answer set

Mr Hamilton as Traveling Salesperson

Solution Optimization

Optimization Objective

A Hamiltonian cycle is optimal if its accumulated edge costs are minimal.

☞ Use `#minimize` (and/or `#maximize`) to associate each answer set with objective value(s).

Optimization Encoding

```
% OPTIMIZE
#minimize[ cycle(X,Y) : cost(X,Y,C) = C@1 ].
```

Target: minimal sum of costs `C` (at priority level `1`) associated with instances of `cycle` in an answer set

Mr Hamilton as Traveling Salesperson

Let's Run it!

```
gringo graph.lp costs.lp cycle.lp price.lp | clasp --stats 0
```

```
Answer: 1
cycle(6,5) cycle(5,3) cycle(4,2) cycle(3,1) cycle(2,6) cycle(1,4)
Optimization: 13
Answer: 2
cycle(6,5) cycle(5,3) cycle(4,1) cycle(3,4) cycle(2,6) cycle(1,2)
Optimization: 12
Answer: 3
cycle(6,3) cycle(5,6) cycle(4,1) cycle(3,4) cycle(2,5) cycle(1,2)
Optimization: 11
OPTIMUM FOUND

Models      : 1
  Enumerated: 3
  Optimum   : yes
Optimization: 11
Time        : 0.004s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
```

Mr Hamilton as Traveling Salesperson

Let's Run it!

```
gringo graph.lp costs.lp cycle.lp price.lp | clasp --stats 0
```

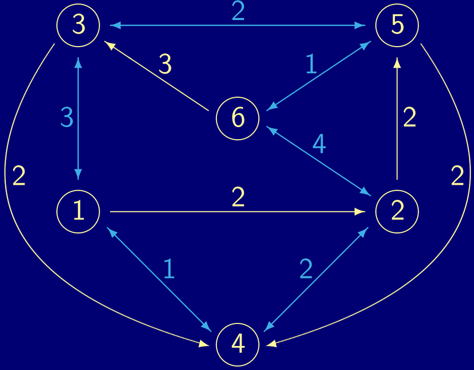
```
Answer: 1
cycle(6,5) cycle(5,3) cycle(4,2) cycle(3,1) cycle(2,6) cycle(1,4)
Optimization: 13
Answer: 2
cycle(6,5) cycle(5,3) cycle(4,1) cycle(3,4) cycle(2,6) cycle(1,2)
Optimization: 12
Answer: 3
cycle(6,3) cycle(5,6) cycle(4,1) cycle(3,4) cycle(2,5) cycle(1,2)
Optimization: 11
OPTIMUM FOUND

Models      : 1
  Enumerated: 3
  Optimum   : yes
Optimization: 11
Time        : 0.004s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
```

Mr Hamilton as Traveling Salesperson

Let's Interpret it!

Found: optimal Hamiltonian cycle

```
Answer: 1

cycle(1,4)
cycle(4,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,1)
```

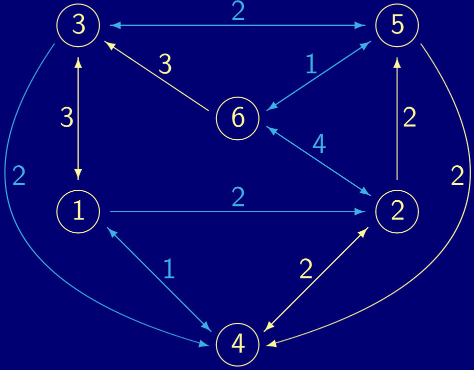


Mr Hamilton as Traveling Salesperson

Let's Interpret it!

Found: optimal Hamiltonian cycle

```
Answer: 2

cycle(1,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,4)
cycle(4,1)
```

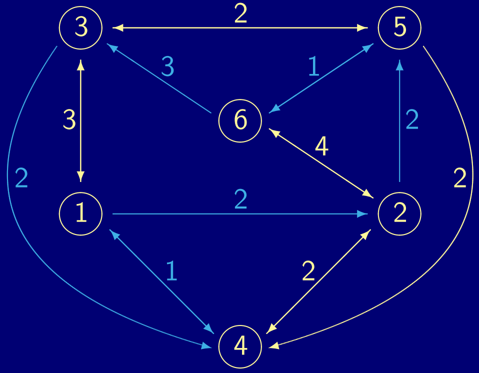


Mr Hamilton as Traveling Salesperson

Let's Interpret it!

Found: optimal Hamiltonian cycle

```
Answer: 3

cycle(1,2)
cycle(2,5)
cycle(5,6)
cycle(6,3)
cycle(3,4)
cycle(4,1)
```



Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

Take-Home Messages

For solving a problem (class) in ASP, provide

1. facts describing an instance and
2. a (uniform) encoding of solutions.

Encodings are often structured by the following logical parts:

1. Domain information (by deduction from facts)
2. Generator providing solution candidates (choice rules)
3. Define rules analyzing properties of candidates (normal rules)
4. Tester eliminating invalid candidates (integrity constraints)
5. Display statements projecting answer sets (onto characteristic atoms)
6. Optimizer evaluating answer sets (`#minimize`/`#maximize`)

In a Nutshell

Logic Program $\subseteq$ (Data + Deduction) + (Generation + Analysis) + Selection + Projection [+ Optimization]

The Camel through the Eye of a Needle

ASP offers

1. rich yet easy modeling languages
2. efficient instantiation procedures
3. powerful search engines

Question: Anything left to worry about?

Answer: Yes! (unfortunately)

☞ Even in declarative programming, the problem encoding matters.

Consider sorting $[4, 7, 2, 5, 1, 8, 6, 3]$

- divide-and-conquer (Quicksort) $\sim 8(\log_2 8) = 16$ "operations"
- permutation guessing $\sim 8!/2 = 20,160$ "operations"

The Camel through the Eye of a Needle

ASP offers

1. rich yet easy modeling languages
2. efficient instantiation procedures
3. powerful search engines

Question: Anything left to worry about?
Answer: Yes! (unfortunately)

☞ Even in declarative programming, the problem encoding matters.

Consider sorting $[4, 7, 2, 5, 1, 8, 6, 3]$

- divide-and-conquer (Quicksort) $\sim 8(\log_2 8) = 16$ "operations"
- permutation guessing $\sim 8!/2 = 20,160$ "operations"

The Camel through the Eye of a Needle

ASP offers

1. rich yet easy modeling languages
2. efficient instantiation procedures
3. powerful search engines

Question: Anything left to worry about?
Answer: Yes! (unfortunately)

☞ Even in declarative programming, the problem encoding matters.

Consider sorting $[4, 7, 2, 5, 1, 8, 6, 3]$

- divide-and-conquer (Quicksort) $\sim 8(\log_2 8) = 16$ "operations"
- permutation guessing $\sim 8!/2 = 20,160$ "operations"

The Camel through the Eye of a Needle

ASP offers

1. rich yet easy modeling languages
2. efficient instantiation procedures
3. powerful search engines

Question: Anything left to worry about?
Answer: Yes! (unfortunately)

☞ Even in declarative programming, the problem encoding matters.

Consider sorting $[4, 7, 2, 5, 1, 8, 6, 3]$

- divide-and-conquer (Quicksort) $\sim 8(\log_2 8) = 16$ "operations"
- permutation guessing $\sim 8!/2 = 20,160$ "operations"

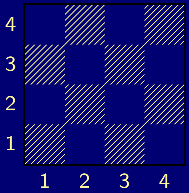
N-Queens Problem

Problem Specification

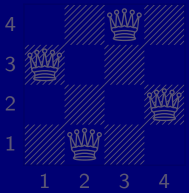
Given an $N \times N$ chessboard,
place N queens such that they do not attack each other
(neither horizontally, vertically, nor diagonally).

$N = 4$

Chessboard



Placement



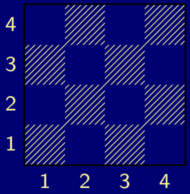
N-Queens Problem

Problem Specification

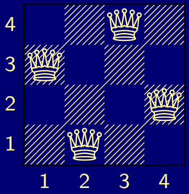
Given an $N \times N$ chessboard,
place N queens such that they do not attack each other
(neither horizontally, vertically, nor diagonally).

$N = 4$

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.

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.

% DISPLAY
#hide. #show queen/2.
```

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.

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.

% DISPLAY
#hide. #show queen/2.
```

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.

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(X2,Y1), X1 < X2.

% DISPLAY
#hide. #show queen/2.
```

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.

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(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.

% DISPLAY
#hide. #show queen/2.
```

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.

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
[...]

% DISPLAY
#hide. #show queen/2.
```

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.

queens_0.lp

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
[...]

% DISPLAY
#hide. #show queen/2.
```

Anything missing?

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

```
% DOMAIN
#const n=4. square(1..n,1..n).

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).

% TEST
[...]
:- 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 8 Queens!

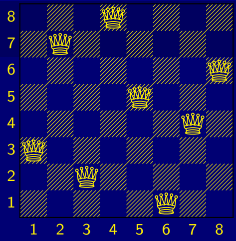
```
gringo -c n=8 queens_0.lp | clasp --stats
```

```
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 8 Queens!

```
gringo -c n=8 queens_0.lp | clasp --stats
```

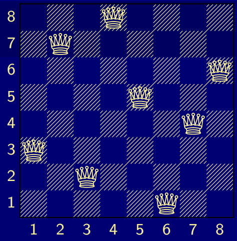



```
Answer: 1
queen(1,6) queen(2,3) queen(3,1) queen(4,7)
queen(5,5) queen(6,8) queen(7,2) queen(8,4)
SATISFIABLE

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 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 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

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
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.
:- 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

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
:- 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|.
:- not n #count{ queen(X,Y) }.

% DISPLAY
#hide. #show queen/2.
```

A First Refinement

At least N queens? Exactly one queen per row and column!

queens_1.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 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 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

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

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
```

☞ Grounding makes the problem!

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | wc)

1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

☞ Grounding makes the problem!

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | wc)

1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

Just kidding :-)

☞ Grounding makes the problem!

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | wc)
```

```
1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

☞ Grounding makes the problem!

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | wc)
```

```
1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

☞ Grounding makes the problem!

A First Refinement

Where Time Has Gone

```
time(gringo -c n=122 queens_1.lp | wc)
```

```
1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys  0m0.090s
```

☞ Grounding makes the problem!

A First Refinement

Grounding Time ∼ 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.
```

A First Refinement

Grounding Time ∼ 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.
```

A First Refinement

Grounding Time ∼ 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.
```

A First Refinement

Grounding Time ∼ 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.
```

A First Refinement

Grounding Time ∼ 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.
```

A First Refinement

Grounding Time ∼ 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.
```

A First Refinement

Grounding Time ∼ 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.
```

A First Refinement

Grounding Time ∼ 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.
```

A First Refinement

Grounding Time ∼ 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.
```

A First Refinement

Grounding Time ∼ 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.
```

A First Refinement

Grounding Time ∼ Space

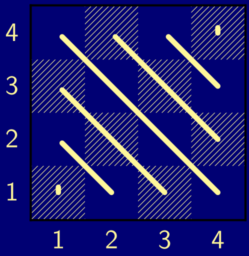
queens_1.lp

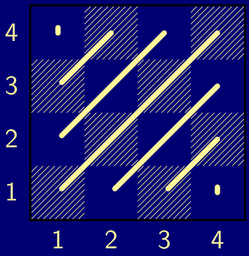
```
% DOMAIN
#const n=4. square(1..n,1..n).                                        O(n×n)

% 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 Nomenclature for Diagonals

$N = 4$

4	**4**	**5**	**6**	**7**
3	**3**	**4**	**5**	**6**
2	**2**	**3**	**4**	**5**
1	**1**	**2**	**3**	**4**
	1	2	3	4

$\#\text{diagonal}_1 =$
$(\#\text{row} + \#\text{column}) - 1$

4	**7**	**6**	**5**	**4**
3	**6**	**5**	**4**	**3**
2	**5**	**4**	**3**	**2**
1	**4**	**3**	**2**	**1**
	1	2	3	4

$\#\text{diagonal}_2 =$
$(\#\text{row} - \#\text{column}) + N$

☞ $\#\text{diagonal}_{1/2}$ can be determined in this way for arbitrary N.

A Nomenclature for Diagonals

$N = 4$

4	**4**	**5**	**6**	**7**
3	**3**	**4**	**5**	**6**
2	**2**	**3**	**4**	**5**
1	**1**	**2**	**3**	**4**
	1	2	3	4

$\#\text{diagonal}_1 =$
$(\#\text{row} + \#\text{column}) - 1$

4	**7**	**6**	**5**	**4**
3	**6**	**5**	**4**	**3**
2	**5**	**4**	**3**	**2**
1	**4**	**3**	**2**	**1**
	1	2	3	4

$\#\text{diagonal}_2 =$
$(\#\text{row} - \#\text{column}) + N$

☞ $\#\text{diagonal}_{1/2}$ can be determined in this way for arbitrary N.

A Nomenclature for Diagonals

$N = 4$

4	**4**	**5**	**6**	**7**
3	**3**	**4**	**5**	**6**
2	**2**	**3**	**4**	**5**
1	**1**	**2**	**3**	**4**
	1	2	3	4

$\#\text{diagonal}_1 =$
$(\#\text{row} + \#\text{column}) - 1$

4	**7**	**6**	**5**	**4**
3	**6**	**5**	**4**	**3**
2	**5**	**4**	**3**	**2**
1	**4**	**3**	**2**	**1**
	1	2	3	4

$\#\text{diagonal}_2 =$
$(\#\text{row} - \#\text{column}) + N$

☞ $\#\text{diagonal}_{1/2}$ can be determined in this way for arbitrary N.

A Second Refinement

Let's go for Diagonals!

queens_1.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 go for Diagonals!

queens_1.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.
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : D == (X+Y)-1 }. % Diagonal 1


% DISPLAY
#hide. #show queen/2.
```

A Second Refinement

Let's go for Diagonals!

queens_1.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.
:- 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 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.
:- 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 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 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 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 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 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_2.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 Precompute Diagonals!

queens_2.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 Precompute Diagonals!

queens_2.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) : diag1(X,Y,D) }. % Diagonal 1
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : diag2(X,Y,D) }. % Diagonal 2

% DISPLAY
#hide. #show queen/2.
```

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) : diag1(X,Y,D) }. % Diagonal 1
:- D = 1..2*n-1, 2 #count{ queen(X,Y) : diag2(X,Y,D) }. % 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 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 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 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
```

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
```

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(1,422) queen(2,458) queen(3,224) queen(4,408) queen(5,405) ...
SATISFIABLE

Models      : 1+
Time        : 37.454s (Solving: 26.38s 1st Model: 26.26s Unsat: 0.00s)
CPU Time    : 29.580s
Choices     : 961315
Conflicts   : 3222
Restarts    : 7

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(1,422) queen(2,458) queen(3,224) queen(4,408) queen(5,405) ...
SATISFIABLE

Models      : 1+
Time        : 37.454s (Solving: 26.38s 1st Model: 26.26s Unsat: 0.00s)
CPU Time    : 29.580s
Choices     : 961315
Conflicts   : 3222
Restarts    : 7

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(1,90) queen(2,452) queen(3,494) queen(4,145) queen(5,84) ...
SATISFIABLE

Models      : 1+
Time        : 22.654s (Solving: 10.53s 1st Model: 10.47s Unsat: 0.00s)
CPU Time    : 15.750s
Choices     : 1058729
Conflicts   : 2128
Restarts    : 6

Variables   : 403123
Constraints : 49636
```

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).
```

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,cheap), pro(X,fresh), pro(X,tasty).
```

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).
pro(asparagus,clean).

buy(X) :- veg(X), pro(X,cheap), pro(X,fresh), pro(X,tasty), pro(X,clean).
```

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).
pro(asparagus,clean).
```

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).   pre(cheap).
pro(asparagus,fresh).   pro(cucumber,fresh).   pre(fresh).
pro(asparagus,tasty).   pro(cucumber,tasty).   pre(tasty).
pro(asparagus,clean).

buy(X) :- veg(X), pro(X,P) : pre(P).
```

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).  pre(cheap).
pro(asparagus,fresh).  pro(cucumber,fresh).  pre(fresh).
pro(asparagus,tasty).  pro(cucumber,tasty).  pre(tasty).
pro(asparagus,clean).                        pre(clean).

buy(X) :- veg(X), pro(X,P) : pre(P).
```

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).   pre(cheap).
pro(asparagus,fresh).   pro(cucumber,fresh).   pre(fresh).
pro(asparagus,tasty).   pro(cucumber,tasty).   pre(tasty).
pro(asparagus,clean).
```

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). pre(cheap).
pro(asparagus,fresh). pro(cucumber,fresh). pre(fresh).
pro(asparagus,tasty). pro(cucumber,tasty). pre(tasty).
pro(asparagus,clean).

buy(X) :- veg(X), not bye(X).     bye(X) :- veg(X), pre(P), not pro(X,P).
```

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).   pre(cheap).
pro(asparagus,fresh).   pro(cucumber,fresh).   pre(fresh).
pro(asparagus,tasty).   pro(cucumber,tasty).   pre(tasty).
pro(asparagus,clean).                          pre(clean).

buy(X) :- veg(X), not bye(X).     bye(X) :- veg(X), pre(P), not pro(X,P).
```

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).    pre(cheap).
pro(asparagus,fresh).    pro(cucumber,fresh).    pre(fresh).
pro(asparagus,tasty).    pro(cucumber,tasty).    pre(tasty).
pro(asparagus,clean).                            pre(clean).

buy(X) :- veg(X), not bye(X).     bye(X) :- veg(X), pre(P), not pro(X,P).
```

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`	`gringo latin_1.lp | wc`
105480 2558984 14005258	42056 273672 1690522

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

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

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
squareX(X) :- square(X,Y).      squareY(Y) :- square(X,Y).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- squareX(X1)  , N = 1..n, not num(X1,Y2,N) : square(X1,Y2).
:- squareY(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

Projecting Irrelevant Details Out

A Latin square encoding

```
% DOMAIN
#const n=32. square(1..n,1..n).
squareX(X) :- square(X,Y).      squareY(Y) :- square(X,Y).

% GENERATE
1 #count{ num(X,Y,N) : N = 1..n } 1 :- square(X,Y).

% TEST
:- squareX(X1)  , N = 1..n, not num(X1,Y2,N) : square(X1,Y2).
:- squareY(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

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

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

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

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
:- num(X1,Y1,N), num(X1,Y2,N), Y1 < Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 < X2.
```

☞ 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

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
:- num(X1,Y1,N), num(X1,Y2,N), Y1 < Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 < X2.
```

☞ uniqueness of `N` in a row/column checked by ENUMERATING PAIRS!

gringo latin_3.lp | wc	gringo latin_4.lp | wc
1055752 6294536 21099558	228360 1205256 4780744

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
:- num(X1,Y1,N), num(X1,Y2,N), Y1 < Y2.
:- num(X1,Y1,N), num(X2,Y1,N), X1 < X2.
```

☞ 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

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).

% DEFINE + 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

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).

% DEFINE + 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

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).

% DEFINE + 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) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc          gringo latin_6.lp | wc
```

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) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc          gringo latin_6.lp | wc
```

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) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc                gringo latin_6.lp | wc
```

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) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc          gringo latin_6.lp | wc
```

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

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) }.           ✗
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.           ✗
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3. #show sigma/1.
```

```
gringo latin_5.lp | wc
```

```
gringo latin_6.lp | wc
```

Assigning Aggregate Values

Yet 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).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3.
```

```
gringo latin_5.lp | wc
```

```
gringo latin_6.lp | wc
```

48136 373768 2185042

Assigning Aggregate Values

Yet 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).

% DEFINE + TEST
occX(X,N,C) :- X = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
occY(Y,N,C) :- Y = 1..n, N = 1..n, C = #count{ num(X,Y,N) }.
:- occX(X,N,C), C != 1.  :- occY(Y,N,C), C != 1.

% DISPLAY
#hide. #show num/3.
```

```
gringo latin_5.lp | wc
```

304136 5778440 30252505

```
gringo latin_6.lp | wc
```

48136 373768 2185042

Assigning Aggregate Values

Yet 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
:- 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 latin_5.lp | wc
```

```
304136 5778440 30252505
```

```
gringo latin_6.lp | wc
```

Assigning Aggregate Values

Yet 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
:- 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 latin_5.lp | wc
```

304136 5778440 30252505

```
gringo latin_6.lp | wc
```

48136 373768 2185042

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.
```

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.
```

☞ many symmetric solutions (mirroring, rotation, value permutation)

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.
```

☞ easy and safe to fix a full row/column!

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.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#hide. #show num/3.
```

☞ easy and safe to fix a full row/column!

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.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#hide. #show num/3.
```

☞ Let's compare enumeration speed!

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_6.lp | clasp -q 0
```

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_6.lp | clasp -q 0
```

```
Models : 161280     Time : 2.078s
```

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.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#hide. #show num/3.
```

```
gringo -c n=5 latin_7.lp | clasp -q 0
```

```
Models : 161280    Time : 2.078s
```

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.
:- square(1,Y), not num(1,Y,Y). % Symmetry Breaking

% DISPLAY
#hide. #show num/3.
```

```
gringo -c n=5 latin_7.lp | clasp -q 0
```

```
Models : 1344      Time : 0.024s
```

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.

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.

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.

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.

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.

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)

- develop and test incrementally
  - prepare toy instances with "interesting features"
  - build the encoding bottom-up and verify additions (eg. new predicates)
- 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)
- ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

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)

- develop and test incrementally
  - prepare toy instances with "interesting features"
  - build the encoding bottom-up and verify additions (eg. new predicates)
- 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)
- ask tools (eg. `http://www.kr.tuwien.ac.at/research/projects/mmdasp/`)

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/`)

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/`)

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/`)

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

  - try auto-configuration (offered by `claspfolio`)
  - try manual fine-tuning (requires expert knowledge!)
  - if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

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

try auto-configuration (offered by `claspfolio`)
try manual fine-tuning (requires expert knowledge!)
if possible, reformulate the problem or add domain knowledge ("redundant" constraints) to help the solver

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

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

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

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

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

Hitori

A Japanese Grid Puzzle (Beyond Sudoku)

The Puzzle

Given: an $N \times N$ board of numbered squares

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)

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

	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

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

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
8	7	1	4	2	3	5	6

Example Solution

Black squares given by instances of `blackOut(X,Y)`:

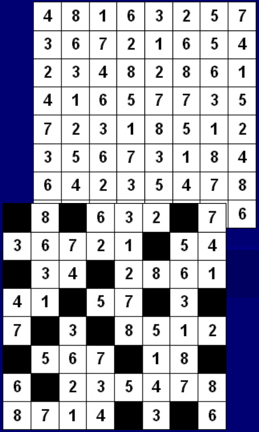
```
blackOut(1,1)   blackOut(2,5)   ...
blackOut(1,3)   ...   blackOut(8,4)
blackOut(1,6)   ...   blackOut(8,6)
```

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).
```



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)
```

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 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 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 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.
```

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.
```

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 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

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
```

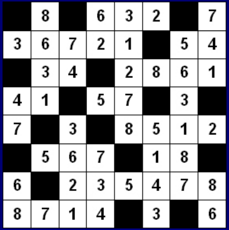
	8		6	3	2		7
3	6	7	2	1		5	4
	3	4		2	8	6	1
4	1		5	7		3	
7		3		8	5	1	2
	5	6	7		1	8	
6		2	3	5	4	7	8
8	7	1	4		3		6

A Working Encoding

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
```



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).
```

☞ no 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 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 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).
```

☞ `blackOut(X,Y)` and `-blackOut(X,Y)` exclusive in view of upper bound!

```
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 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).
```

☞ `blackOut(X,Y)` and `-blackOut(X,Y)` exclusive in view of upper bound!

```
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 Classical Negation?

hitori_1.lp

```
% Every square is blacked out or normal
1 { blackOut(X,Y), negBlackOut(X,Y) } 1 :- state(X,Y,_).

:- blackOut(X,Y), -blackOut(X,Y).
```

☞ no 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 Classical Negation?

hitori_1.lp

```
% Every square is blacked out or normal
1 { blackOut(X,Y), negBlackOut(X,Y) } 1 :- state(X,Y,_).

:- blackOut(X,Y), -blackOut(X,Y).
```

☞ no 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 Classical Negation?

hitori_1.lp

```
% Every square is blacked out or normal
1 { blackOut(X,Y), negBlackOut(X,Y) } 1 :- state(X,Y,_).

:- blackOut(X,Y), -blackOut(X,Y).
```

☞ no 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.
 ...
```

☞ `blackOut(X,Y)` and `negBlackOut(X,Y)` are two sides of the same coin

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.
 ...
```

☞ blackOut(X,Y) and negBlackOut(X,Y) are two sides of the same coin

Why Not Default Negation?

hitori_2.lp

```
% Every square is blacked out or normal
 { blackOut(X,Y) } :- state(X,Y,_).

% 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.
 ...
```

☞ replace `negBlackOut(X,Y)` by "`not blackOut(X,Y)`"

A First Improvement

```
gringo hitori_1.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
```

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
```

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        : 10.177s (Solving: 8.42s 1st Model: 8.41s Unsat: 0.00s)
CPU Time    : 9.990s
Choices     : 344
Conflicts   : 264
Restarts    : 2

Variables   : 260433
Constraints : 1018825
```

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

Remember Symmetric Inequalities

hitori_3.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

Remember Symmetric Inequalities

hitori_3.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
 :- 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.
```

Let's Use Counting

hitori_4.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_3.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
```

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
```

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        : 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
```

Why Double-Check Reachability?

hitori_4.lp

```
% Define mutual reachability
reachable(X1,Y1,X2,Y2) :- adjacent(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).




% 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,_).
```

☞ reachable(X1,Y1,X2,Y2) and reachable(X2,Y2,X1,Y1) hold jointly

Why Double-Check Reachability?

hitori_4.lp

```
% Define mutual reachability
reachable(X1,Y1,X2,Y2) :- adjacent(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).




% 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,_).
```

☞ `reachable(X1,Y1,X2,Y2)` and `reachable(X2,Y2,X1,Y1)` hold jointly

Why Double-Check Reachability?

hitori_4.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)`

Why Double-Check Reachability?

hitori_4.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)`

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_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        : 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
```

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
```

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        : 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
```

Two Orders of Magnitude!

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).
```

☞ grounding size: $O(8^6)$

Two Orders of Magnitude!

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).
```

☞ grounding size: $O(8^6)$

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: $O(8^6)$

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: $O(8^4)$

A First 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        : 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
```

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
```

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        : 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 Think a Bit More

hitori_6.lp

```
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).

% Can't have unreachable non-black square
 :- not blackOut(X1,Y1), not blackOut(X2,Y2), not reachable(X1,Y1,X2,Y2),
    (X1,Y1) < (X2,Y2),   state(X1,Y1,_), state(X2,Y2,_).
```

Q: How many squares adjacent to `(1,1)` can possibly be black?

A:

	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

Let's Think a Bit More

hitori_6.lp

```
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).

% Can't have unreachable non-black square
 :- not blackOut(X1,Y1), not blackOut(X2,Y2), not reachable(X1,Y1,X2,Y2),
    (X1,Y1) < (X2,Y2),   state(X1,Y1,_), state(X2,Y2,_).
```

Q: How many squares adjacent to (1,1) can possibly be black?

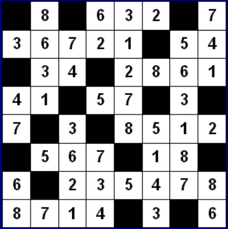
A: At most one!

	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

Let's Think a Bit More

hitori_6.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
 :- not blackOut(X1,Y1), not blackOut(X2,Y2), not reachable(X1,Y1,X2,Y2),
    (X1,Y1) < (X2,Y2),   state(X1,Y1,_), state(X2,Y2,_).
```



Q: How many squares adjacent to (1,1) can possibly be black?

A: At most one!

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_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        : 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
```

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
```

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        : 0.009s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time    : 0.000s
Choices     : 77
Conflicts   : 25
Restarts    : 0

Variables   : 539
Constraints : 1137
```

Let's Reach All Squares (Anyway)

hitori_7.lp

```
% Define reachability
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).
```

☞ require all white squares to be reached

Let's Reach All Squares (Anyway)

hitori_7.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

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 square
 :- state(X,Y,_), not reachable(X,Y).
```

☞ require all white squares to be reached

The Final Result

```
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        : 0.009s (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 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 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        : 0.006s (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 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
```

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!

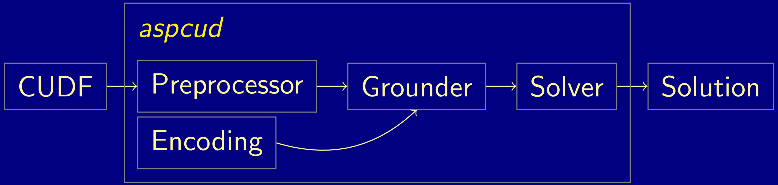
Some Real-World Applications
Overview

Motivation

- difficulties in maintaining packages of modern Linux distributions
    - complex dependencies
    - large package repositories
    - ever changing in view of software development
- challenges for package configuration tools
    - large problem size
    - soft (and hard) constraints
    - multiple optimization criteria
- advantages of ASP
    - uniform modeling by encoding plus instance(s)
    - solving techniques for multi-criteria optimization

Overview

aspcud tool for solving package configuration problems



Preprocessor converts CUDF input to ASP instance

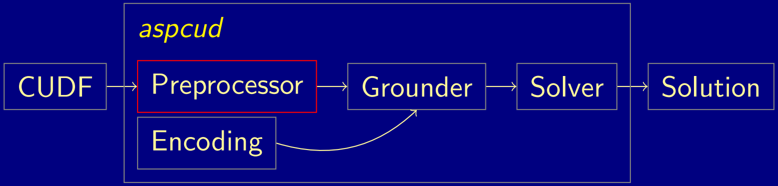
Encoding first-order problem specification

Grounder instantiates first-order variables

Solver searches for (optimal) answer sets

Overview

aspcud tool for solving package configuration problems



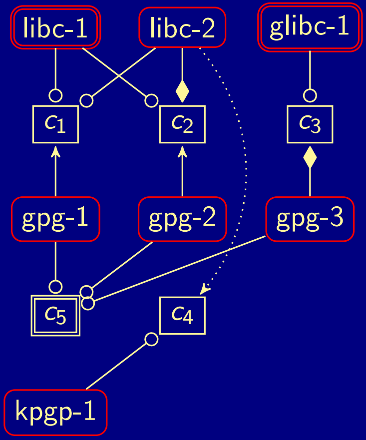
Preprocessor converts CUDF input to ASP instance

Encoding first-order problem specification

Grounder instantiates first-order variables

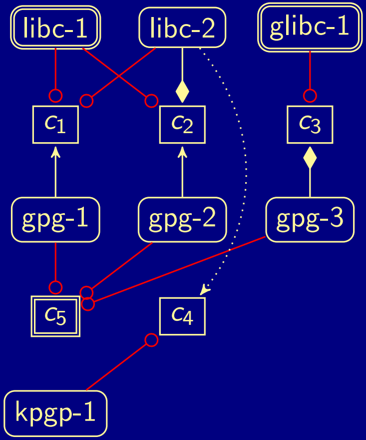
Solver searches for (optimal) answer sets

Instance Format



Installable Packages:

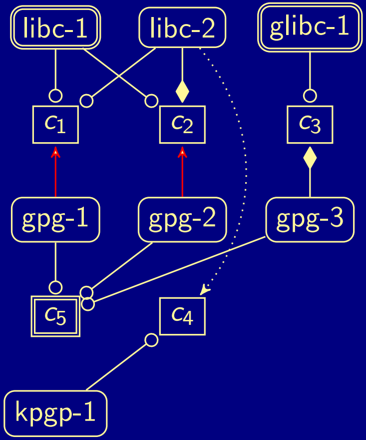
```
package(libc,1).
package(libc,2).

package(glibc,1).

package(gpg,1).
package(gpg,2).
package(gpg,3).

package(kpgp,1).
```



Instance Format

Package Clauses:

```
satisfies(libc,1,c1).
satisfies(libc,1,c2).
satisfies(libc,2,c1).

satisfies(glibc,1,c3).

satisfies(gpg,1,c5).
satisfies(gpg,2,c5).
satisfies(gpg,3,c5).

satisfies(kpgp,1,c4).
```

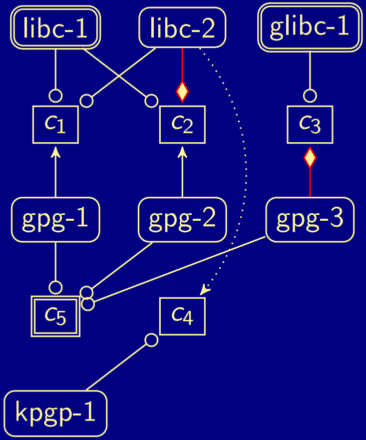


Instance Format

Package Dependencies:

```
depends(gpg,1,c1).
depends(gpg,2,c2).
```

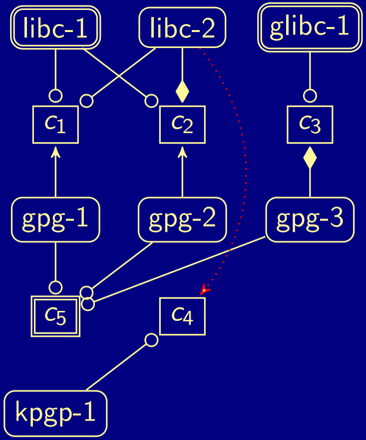
Instance Format



Package Conflicts:

```
conflicts(libc,2,c2).

conflicts(gpg,3,c3).
```

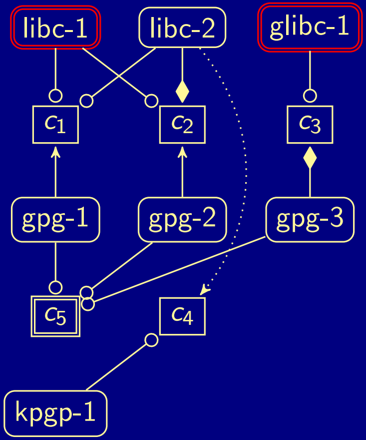
Instance Format



Package Recommendations:

```
recommends(libc,2,c4).
```

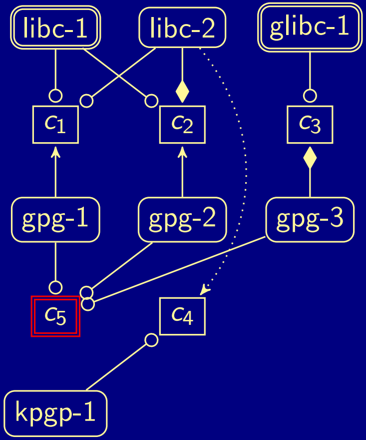
Instance Format



Installed Packages:

```
installed(libc,1).

installed(glibc,1).
```

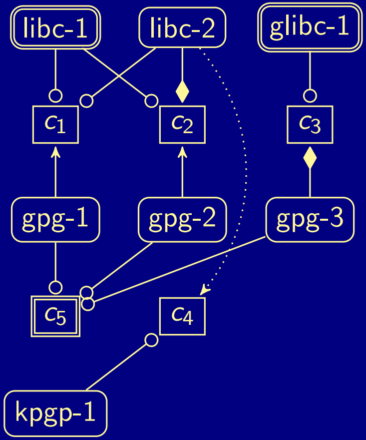
Instance Format



Requests:

```
requested(c5).
```

Instance Format

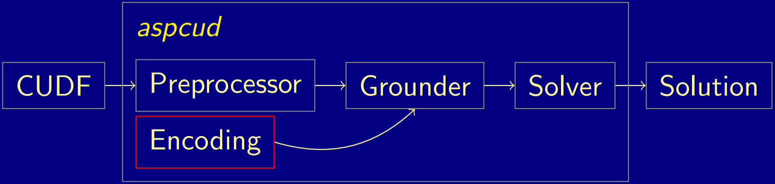


Optimization Criteria:

```
utility(delete,-1).
utility(change,-2).
```

Overview

aspcud tool for solving package configuration problems



Preprocessor converts CUDF input to ASP instance

Encoding first-order problem specification

Grounder instantiates first-order variables

Solver searches for (optimal) answer sets

Hard Constraints

```
% choose packages to install
{ install(N,V) } :- package(N,V).

% derive required clauses
exclude(C) :- install(N,V), conflicts(N,V,C).
include(C) :- install(N,V), depends(N,V,C).
% derive satisfied clauses
satisfy(C) :- install(N,V), satisfies(N,V,C).

% assert required clauses to be (un)satisfied
 :- exclude(C),     satisfy(C).
 :- include(C), not satisfy(C).
 :- request(C), not satisfy(C).
```

Hard Constraints

```
% choose packages to install
{ install(N,V) } :- package(N,V).

% derive required clauses
exclude(C) :- install(N,V), conflicts(N,V,C).
include(C) :- install(N,V), depends(N,V,C).
% derive satisfied clauses
satisfy(C) :- install(N,V), satisfies(N,V,C).

% assert required clauses to be (un)satisfied
 :- exclude(C),     satisfy(C).
 :- include(C), not satisfy(C).
 :- request(C), not satisfy(C).
```

Hard Constraints

```
% choose packages to install
{ install(N,V) } :- package(N,V).

% derive required clauses
exclude(C) :- install(N,V), conflicts(N,V,C).
include(C) :- install(N,V), depends(N,V,C).
% derive satisfied clauses
satisfy(C) :- install(N,V), satisfies(N,V,C).

% assert required clauses to be (un)satisfied
 :- exclude(C),     satisfy(C).
 :- include(C), not satisfy(C).
 :- request(C), not satisfy(C).
```

"Redundant" Hard Constraints

```
% lift package interdependencies (applying to all version)
pconflicts(N,C) :-  conflicts(N,V,C).
 conflicts(N,C) :- pconflicts(N,  C), conflicts(N,V,C) : package(N,V).

pdepends(N,C)   :-  depends(N,V,C).
 depends(N,C)   :- pdepends(N,  C),     depends(N,V,C) : package(N,V).

psatisfies(N,C) :-  satisfies(N,V,C).
 satisfies(N,C) :- psatisfies(N,  C), satisfies(N,V,C) : package(N,V).

% lifted derivations of required and satisfied clauses
install(N) :- install(N,V).

exclude(C) :- install(N), conflicts(N,C).
include(C) :- install(N), depends(N,C).

satisfy(C) :- install(N), satisfies(N,C).
```

"Redundant" Hard Constraints

```
% lift package interdependencies (applying to all version)
pconflicts(N,C) :-  conflicts(N,V,C).
 conflicts(N,C) :- pconflicts(N,  C), conflicts(N,V,C) : package(N,V).

pdepends(N,C)   :-  depends(N,V,C).
 depends(N,C)   :- pdepends(N,  C),     depends(N,V,C) : package(N,V).

psatisfies(N,C) :-  satisfies(N,V,C).
 satisfies(N,C) :- psatisfies(N,  C), satisfies(N,V,C) : package(N,V).

% lifted derivations of required and satisfied clauses
install(N) :- install(N,V).

exclude(C) :- install(N), conflicts(N,C).
include(C) :- install(N), depends(N,C).

satisfy(C) :- install(N), satisfies(N,C).
```

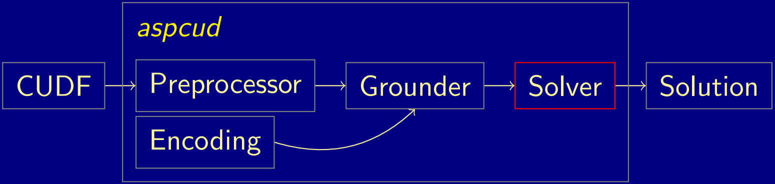
Soft Constraints

```
% auxiliary definition
installed(N) :- installed(N,V).

% derive optimization criteria violations
violate(newpkg,N) :-
    utility(newpkg,L), install(N), not installed(N).
violate(delete,N) :-
    utility(delete,L), installed(N), not install(N).
% similar for other criteria
...

% impose soft constraints
#minimize[ violate(U,T) = 1 @ -L : utility(U,L) : L < 0 ].
#maximize[ violate(U,T) = 1 @  L : utility(U,L) : L > 0 ].
```

Soft Constraints

```
% auxiliary definition
installed(N) :- installed(N,V).

% derive optimization criteria violations
violate(newpkg,N) :-
    utility(newpkg,L), install(N), not installed(N).
violate(delete,N) :-
    utility(delete,L), installed(N), not install(N).
% similar for other criteria
...

% impose soft constraints
#minimize[ violate(U,T) = 1 @ -L : utility(U,L) : L < 0 ].
#maximize[ violate(U,T) = 1 @  L : utility(U,L) : L > 0 ].
```

Optimization Algorithm



- package configuration problems often under-constrained
- lexicographical optimization algorithm enumerates too much

Alternative Approach

- optimize criteria in the order of significance
- decrease upper bounds (costs) w.r.t. witnesses
- proceed to next criterion upon unsatisfiability

Design Goals

- incorporate into conflict-driven solving
- keep as much learned information as possible
- build upon standard features like assumptions

Experimental Results

- optimization of (Debian) Linux installations wrt. multiple criteria
- approaches of participants include
  1. Maximum Satisfiability: *cudf2msu*
  2. Pseudo-Boolean Optimization: *cudf2pbo*, *p2cudf*
  3. Answer Set Programming: *aspcud*
     - configurable optimization strategies and heuristics
- benchmarks and scoring from the 3rd MISC-live run (5 tracks)
  - MISC(-live) regularly organized by mancoosi consortium

	paranoid		trendy		user1		user2		user3	
Solver	S	T/O	S	T/O	S	T/O	S	T/O	S	T/O
$clasp_0^0$-r	431	2,287/6	1730	23,829/ 80	935	14,349/35	525	5,097/12	1031	14,184/37
$clasp_0^0$	416	2,294/6	2375	29,781/105	1727	21,897/73	1224	14,697/45	671	11,178/21
$clasp_0^1$-r	**410**	2,210/6	1560	22,660/ 73	898	13,466/30	502	4,654/ 9	980	13,682/35
$clasp_0^1$	**410**	2,326/6	2079	26,471/ 92	1723	21,525/72	922	10,767/31	658	10,675/23
$clasp_0^2$-r	427	2,135/6	712	16,867/ 51	527	5,891/11	426	2,981/ 5	587	7,628/20
$clasp_0^3$-r	429	**2,134**/6	740	17,079/ 52	507	5,863/12	425	3,044/ 6	576	7,769/21
$clasp_1^0$-r	425	2,428/6	579	16,713/ 50	550	5,819/14	434	3,000/ 6	710	8,958/25
$clasp_1^0$	417	2,418/6	549	16,544/ 50	475	5,318/12	411	2,538/ 5	502	6,279/16
$clasp_1^1$-r	429	2,405/6	622	17,304/ 50	518	5,908/13	438	2,976/ 6	676	8,938/23
$clasp_1^1$	427	2,372/6	613	16,946/ 49	490	5,478/12	416	2,562/ 5	496	6,144/16
$clasp_1^2$-r	427	2,352/6	571	16,646/ 50	518	5,358/13	418	2,582/ 5	471	6,356/16
$clasp_1^3$-r	429	2,346/6	547	16,386/ 50	499	5,306/12	413	2,498/ 5	497	6,255/16
$clasp_2^0$-r	425	2,392/6	806	16,598/ 50	523	5,583/13	421	2,677/ 6	479	5,548/12
$clasp_2^0$	417	2,364/7	748	17,132/ 50	487	5,823/14	422	2,583/ 5	482	5,592/15
$clasp_2^1$-r	416	2,378/6	752	17,269/ 52	492	5,663/12	414	2,409/ 5	451	5,349/11
$clasp_2^1$	425	2,365/6	864	17,128/ 51	517	6,151/15	412	2,681/ 5	463	5,972/14
$clasp_2^2$-r	445	2,402/6	706	16,551/ 50	528	5,788/13	419	2,700/ 5	436	5,519/13
$clasp_2^3$-r	434	2,345/6	748	16,982/ 51	518	5,850/14	415	2,559/ 5	457	5,360/13
cudf2msu	610	3,051/8	669	5,318/ 8	1270	8,709/18	548	3,238/ 7	504	4,750/ 9
cudf2pbo	465	**2,727**/7	1082	21,302/ 68	520	6,168/13	462	3,575/ 7	537	3,487/ 8
p2cudf	**463**	2,920/8	696	19,105/ 60	516	3,947/ 7	573	6,927/16	577	8,063/21

	paranoid		trendy		user1		user2		user3	
Solver	*S*	*T/O*	*S*	*T/O*	*S*	*T/O*	*S*	*T/O*	*S*	*T/O*
clasp$_0^0$-r	431	2,287/6	1730	23,829/ 80	935	14,349/35	525	5,097/12	1031	14,184/37
clasp$_0^0$	416	2,294/6	2375	29,781/105	1727	21,897/73	1224	14,697/45	671	11,178/21
clasp$_0^1$-r	**410**	2,210/6	1560	22,660/ 73	898	13,466/30	502	4,654/ 9	980	13,682/35
clasp$_0^1$	**410**	2,326/6	2079	26,471/ 92	1723	21,525/72	922	10,767/31	658	10,675/23
clasp$_0^2$-r	427	2,135/6	712	16,867/ 51	527	5,891/11	426	2,981/ 5	587	7,628/20
clasp$_0^3$-r	429	**2,134**/6	740	17,079/ 52	507	5,863/12	425	3,044/ 6	576	7,769/21
clasp$_1^0$-r	425	2,428/6	579	16,713/ 50	550	5,819/14	434	3,000/ 6	710	8,958/25
clasp$_1^0$	417	2,418/6	549	16,544/ 50	475	5,318/12	411	2,538/ 5	502	6,279/16
clasp$_1^1$-r	429	2,405/6	622	17,304/ 50	518	5,908/13	438	2,976/ 6	676	8,938/23
clasp$_1^1$	427	2,372/6	613	16,946/ 49	490	5,478/12	416	2,562/ 5	496	6,144/16
clasp$_1^2$-r	427	2,352/6	571	16,646/ 50	518	5,358/13	418	2,582/ 5	471	6,356/16
clasp$_1^3$-r	429	2,346/6	**547**	**16,386**/ 50	499	5,306/12	413	2,498/ 5	497	6,255/16
clasp$_2^0$-r	425	2,392/6	806	16,598/ 50	523	5,583/13	421	2,677/ 6	479	5,548/12
clasp$_2^0$	417	2,364/7	748	17,132/ 50	487	5,823/14	422	2,583/ 5	482	5,592/15
clasp$_2^1$-r	416	2,378/6	752	17,269/ 52	492	5,663/12	414	2,409/ 5	451	5,349/11
clasp$_2^1$	425	2,365/6	864	17,128/ 51	517	6,151/15	412	2,681/ 5	463	5,972/14
clasp$_2^2$-r	445	2,402/6	706	16,551/ 50	528	5,788/13	419	2,700/ 5	436	5,519/13
clasp$_2^3$-r	434	2,345/6	748	16,982/ 51	518	5,850/14	415	2,559/ 5	457	5,360/13
cudf2msu	610	3,051/8	**669**	**5,318**/ 8	1270	8,709/18	548	3,238/ 7	504	4,750/ 9
cudf2pbo	465	**2,727**/7	1082	21,302/ 68	520	6,168/13	462	3,575/ 7	537	3,487/ 8
p2cudf	**463**	2,920/8	696	19,105/ 60	516	3,947/ 7	573	6,927/16	577	8,063/21

	paranoid		trendy		user1		user2		user3	
Solver	S	T/O	S	T/O	S	T/O	S	T/O	S	T/O
$clasp_0^0$-r	431	2,287/6	1730	23,829/ 80	935	14,349/35	525	5,097/12	1031	14,184/37
$clasp_0^0$	416	2,294/6	2375	29,781/105	1727	21,897/73	1224	14,697/45	671	11,178/21
$clasp_0^1$-r	**410**	2,210/6	1560	22,660/ 73	898	13,466/30	502	4,654/ 9	980	13,682/35
$clasp_0^1$	**410**	2,326/6	2079	26,471/ 92	1723	21,525/72	922	10,767/31	658	10,675/23
$clasp_0^2$-r	427	2,135/6	712	16,867/ 51	527	5,891/11	426	2,981/ 5	587	7,628/20
$clasp_0^3$-r	429	**2,134**/6	740	17,079/ 52	507	5,863/12	425	3,044/ 6	576	7,769/21
$clasp_1^0$-r	425	2,428/6	579	16,713/ 50	550	5,819/14	434	3,000/ 6	710	8,958/25
$clasp_1^0$	417	2,418/6	549	16,544/ 50	**475**	5,318/12	411	2,538/ 5	502	6,279/16
$clasp_1^1$-r	429	2,405/6	622	17,304/ 50	518	5,908/13	438	2,976/ 6	676	8,938/23
$clasp_1^1$	427	2,372/6	613	16,946/ 49	490	5,478/12	416	2,562/ 5	496	6,144/16
$clasp_1^2$-r	427	2,352/6	571	16,646/ 50	518	5,358/13	418	2,582/ 5	471	6,356/16
$clasp_1^3$-r	429	2,346/6	**547**	**16,386**/ 50	499	**5,306**/12	413	2,498/ 5	497	6,255/16
$clasp_2^0$-r	425	2,392/6	806	16,598/ 50	523	5,583/13	421	2,677/ 6	479	5,548/12
$clasp_2^0$	417	2,364/7	748	17,132/ 50	487	5,823/14	422	2,583/ 5	482	5,592/15
$clasp_2^1$-r	416	2,378/6	752	17,269/ 52	492	5,663/12	414	2,409/ 5	451	5,349/11
$clasp_2^1$	425	2,365/6	864	17,128/ 51	517	6,151/15	412	2,681/ 5	463	5,972/14
$clasp_2^2$-r	445	2,402/6	706	16,551/ 50	528	5,788/13	419	2,700/ 5	436	5,519/13
$clasp_2^3$-r	434	2,345/6	748	16,982/ 51	518	5,850/14	415	2,559/ 5	457	5,360/13
cudf2msu	610	3,051/8	**669**	**5,318**/ 8	1270	8,709/18	548	3,238/ 7	504	4,750/ 9
cudf2pbo	465	**2,727**/7	1082	21,302/ 68	520	6,168/13	462	3,575/ 7	537	3,487/ 8
p2cudf	**463**	2,920/8	696	19,105/ 60	**516**	**3,947**/ 7	573	6,927/16	577	8,063/21

	paranoid		trendy		user1		user2		user3	
Solver	S	T/O	S	T/O	S	T/O	S	T/O	S	T/O
clasp$^0_0$-r	431	2,287/6	1730	23,829/ 80	935	14,349/35	525	5,097/12	1031	14,184/37
clasp$^0_0$	416	2,294/6	2375	29,781/105	1727	21,897/73	1224	14,697/45	671	11,178/21
clasp$^1_0$-r	**410**	2,210/6	1560	22,660/ 73	898	13,466/30	502	4,654/ 9	980	13,682/35
clasp$^1_0$	**410**	2,326/6	2079	26,471/ 92	1723	21,525/72	922	10,767/31	658	10,675/23
clasp$^2_0$-r	427	2,135/6	712	16,867/ 51	527	5,891/11	426	2,981/ 5	587	7,628/20
clasp$^3_0$-r	429	**2,134**/6	740	17,079/ 52	507	5,863/12	425	3,044/ 6	576	7,769/21
clasp$^0_1$-r	425	2,428/6	579	16,713/ 50	550	5,819/14	434	3,000/ 6	710	8,958/25
clasp$^0_1$	417	2,418/6	549	16,544/ 50	**475**	5,318/12	**411**	2,538/ 5	502	6,279/16
clasp$^1_1$-r	429	2,405/6	622	17,304/ 50	518	5,908/13	438	2,976/ 6	676	8,938/23
clasp$^1_1$	427	2,372/6	613	16,946/ 49	490	5,478/12	416	2,562/ 5	496	6,144/16
clasp$^2_1$-r	427	2,352/6	571	16,646/ 50	518	5,358/13	418	2,582/ 5	471	6,356/16
clasp$^3_1$-r	429	2,346/6	**547**	**16,386**/ 50	499	**5,306**/12	413	2,498/ 5	497	6,255/16
clasp$^0_2$-r	425	2,392/6	806	16,598/ 50	523	5,583/13	421	2,677/ 6	479	5,548/12
clasp$^0_2$	417	2,364/7	748	17,132/ 50	487	5,823/14	422	2,583/ 5	482	5,592/15
clasp$^1_2$-r	416	2,378/6	752	17,269/ 52	492	5,663/12	414	**2,409**/ 5	451	5,349/11
clasp$^1_2$	425	2,365/6	864	17,128/ 51	517	6,151/15	412	2,681/ 5	463	5,972/14
clasp$^2_2$-r	445	2,402/6	706	16,551/ 50	528	5,788/13	419	2,700/ 5	436	5,519/13
clasp$^3_2$-r	434	2,345/6	748	16,982/ 51	518	5,850/14	415	2,559/ 5	457	5,360/13
cudf2msu	610	3,051/8	**669**	**5,318**/ 8	1270	8,709/18	548	**3,238**/ 7	504	4,750/ 9
cudf2pbo	465	**2,727**/7	1082	21,302/ 68	520	6,168/13	**462**	3,575/ 7	537	3,487/ 8
p2cudf	**463**	2,920/8	696	19,105/ 60	**516**	**3,947**/ 7	573	6,927/16	577	8,063/21

	paranoid		trendy		user1		user2		user3	
Solver	*S*	*T/O*	*S*	*T/O*	*S*	*T/O*	*S*	*T/O*	*S*	*T/O*
$clasp_0^0$-r	431	2,287/6	1730	23,829/ 80	935	14,349/35	525	5,097/12	1031	14,184/37
$clasp_0^0$	416	2,294/6	2375	29,781/105	1727	21,897/73	1224	14,697/45	671	11,178/21
$clasp_0^1$-r	**410**	2,210/6	1560	22,660/ 73	898	13,466/30	502	4,654/ 9	980	13,682/35
$clasp_0^1$	**410**	2,326/6	2079	26,471/ 92	1723	21,525/72	922	10,767/31	658	10,675/23
$clasp_0^2$-r	427	2,135/6	712	16,867/ 51	527	5,891/11	426	2,981/ 5	587	7,628/20
$clasp_0^3$-r	429	**2,134**/6	740	17,079/ 52	507	5,863/12	425	3,044/ 6	576	7,769/21
$clasp_1^0$-r	425	2,428/6	579	16,713/ 50	550	5,819/14	434	3,000/ 6	710	8,958/25
$clasp_1^0$	417	2,418/6	549	16,544/ 50	**475**	5,318/12	**411**	2,538/ 5	502	6,279/16
$clasp_1^1$-r	429	2,405/6	622	17,304/ 50	518	5,908/13	438	2,976/ 6	676	8,938/23
$clasp_1^1$	427	2,372/6	613	16,946/ 49	490	5,478/12	416	2,562/ 5	496	6,144/16
$clasp_1^2$-r	427	2,352/6	571	16,646/ 50	518	5,358/13	418	2,582/ 5	471	6,356/16
$clasp_1^3$-r	429	2,346/6	**547**	**16,386**/ 50	499	**5,306**/12	413	2,498/ 5	497	6,255/16
$clasp_2^0$-r	425	2,392/6	806	16,598/ 50	523	5,583/13	421	2,677/ 6	479	5,548/12
$clasp_2^0$	417	2,364/7	748	17,132/ 50	487	5,823/14	422	2,583/ 5	482	5,592/15
$clasp_2^1$-r	416	2,378/6	752	17,269/ 52	492	5,663/12	414	**2,409**/ 5	451	**5,349**/11
$clasp_2^1$	425	2,365/6	864	17,128/ 51	517	6,151/15	412	2,681/ 5	463	5,972/14
$clasp_2^2$-r	445	2,402/6	706	16,551/ 50	528	5,788/13	419	2,700/ 5	**436**	5,519/13
$clasp_2^3$-r	434	2,345/6	748	16,982/ 51	518	5,850/14	415	2,559/ 5	457	5,360/13
cudf2msu	610	3,051/8	**669**	**5,318**/ 8	1270	8,709/18	548	**3,238**/ 7	**504**	4,750/ 9
cudf2pbo	465	**2,727**/7	1082	21,302/ 68	520	6,168/13	**462**	3,575/ 7	537	**3,487**/ 8
p2cudf	**463**	2,920/8	696	19,105/ 60	**516**	**3,947**/ 7	573	6,927/16	577	8,063/21

Regulatory Networks vs Experimental Profiles

Molecular Biology

- Repositories of biochemical reactions and genetic regulations
  - *Often established experimentally*
- High-throughput methods for collecting experimental profiles
  - *Often incompatible with biological knowledge*
- Incompatibilities due to unreliable data or missing reactions
  - *It is still a common practice to shift the task of making biological sense out of experimental profiles on human experts!*

Qualitative Approach

- Represent regulatory networks by influence graphs
- Represent experimental profiles by observed variations
- An experimental profile is consistent with a regulatory network **iff** each observed variation can be explained by some influence
  - *Inconsistencies point to unreliable data or missing reactions!*

Regulatory Networks vs Experimental Profiles

Molecular Biology

- Repositories of biochemical reactions and genetic regulations
  - *Often established experimentally*
- High-throughput methods for collecting experimental profiles
  - *Often incompatible with biological knowledge*
- Incompatibilities due to unreliable data or missing reactions
  - *It is still a common practice to shift the task of making biological sense out of experimental profiles on human experts!*

Qualitative Approach

- Represent regulatory networks by influence graphs
- Represent experimental profiles by observed variations
- An experimental profile is consistent with a regulatory network **iff** each observed variation can be explained by some influence
  - *Inconsistencies point to unreliable data or missing reactions!*

Regulatory Networks vs Experimental Profiles

Molecular Biology

- Repositories of biochemical reactions and genetic regulations
  - *Often established experimentally*
- High-throughput methods for collecting experimental profiles
  - *Often incompatible with biological knowledge*
- Incompatibilities due to unreliable data or missing reactions
  - *It is still a common practice to shift the task of making biological sense out of experimental profiles on human experts!*

Qualitative Approach

- Represent regulatory networks by influence graphs
- Represent experimental profiles by observed variations
- An experimental profile is consistent with a regulatory network **iff** each observed variation can be explained by some influence
  - *Inconsistencies point to unreliable data or missing reactions!*

Regulatory Networks vs Experimental Profiles

Molecular Biology

- Repositories of biochemical reactions and genetic regulations
  - *Often established experimentally*
- High-throughput methods for collecting experimental profiles
  - *Often incompatible with biological knowledge*
- Incompatibilities due to unreliable data or missing reactions
  - *It is still a common practice to shift the task of making biological sense out of experimental profiles on human experts!*

Qualitative Approach

- Represent regulatory networks by influence graphs
- Represent experimental profiles by observed variations
- An experimental profile is consistent with a regulatory network **iff** each observed variation can be explained by some influence
  - *Inconsistencies point to unreliable data or missing reactions!*

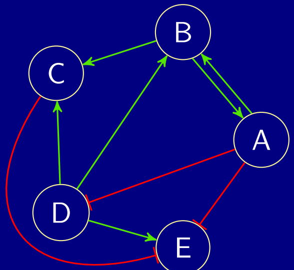
Influence Graphs

Vertices: genes, metabolites, proteins

Edges: regulations

- activation
- inhibition

Example:

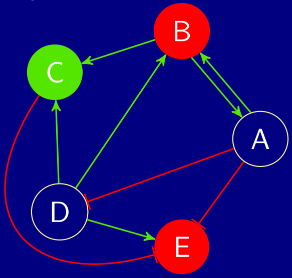


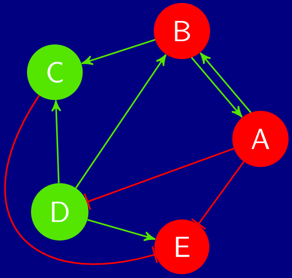
Observations

Labels: variations found in genetic profiles

- increase
- decrease

Examples:



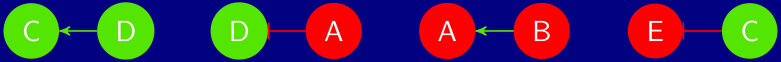


Note: Observations and regulation labelings can be partial

Sign Consistency Constraints (SCCs)

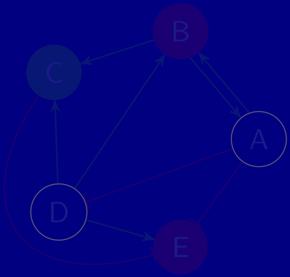
Local Consistency:

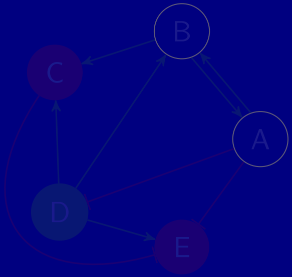
- A variation is consistent **iff** it is explained by some influence



Global Consistency:

- A (partially) labeled influence graph is consistent **iff** there is a total labeling such that every variation is explained

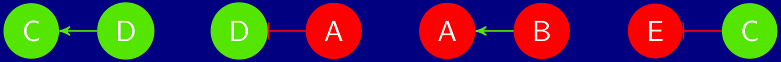




Sign Consistency Constraints (SCCs)

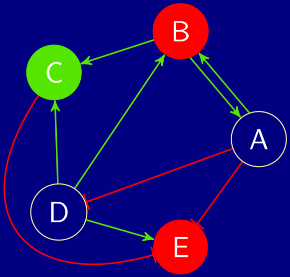
Local Consistency:

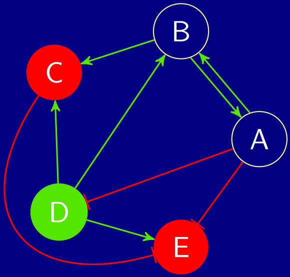
- A variation is consistent **iff** it is explained by some influence



Global Consistency:

- A (partially) labeled influence graph is consistent **iff** there is a total labeling such that every variation is explained

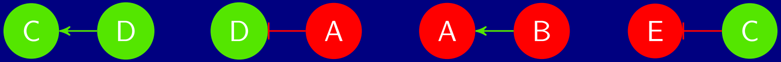




Sign Consistency Constraints (SCCs)

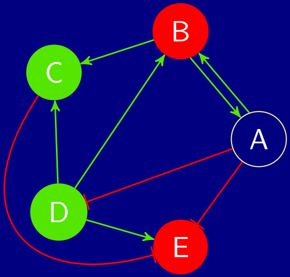
Local Consistency:

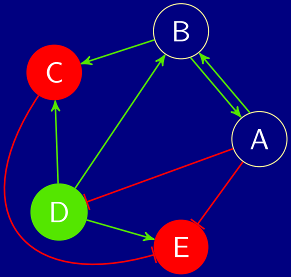
- A variation is consistent **iff** it is explained by some influence



Global Consistency:

- A (partially) labeled influence graph is consistent **iff** there is a total labeling such that every variation is explained

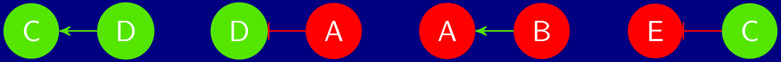




Sign Consistency Constraints (SCCs)

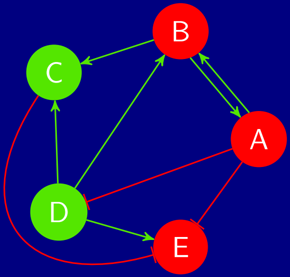
Local Consistency:

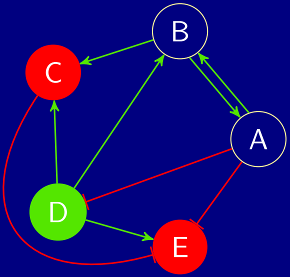
- A variation is consistent **iff** it is explained by some influence



Global Consistency:

- A (partially) labeled influence graph is consistent **iff** there is a total labeling such that every variation is explained

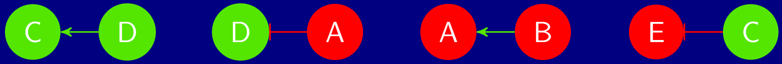




Sign Consistency Constraints (SCCs)

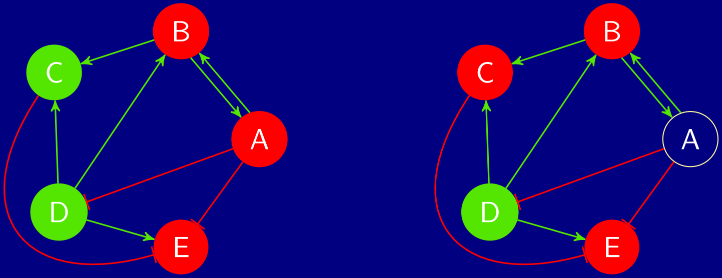
Local Consistency:

- A variation is consistent **iff** it is explained by some influence



Global Consistency:

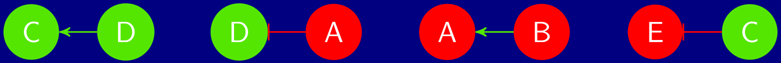
- A (partially) labeled influence graph is consistent **iff** there is a total labeling such that every variation is explained



Sign Consistency Constraints (SCCs)

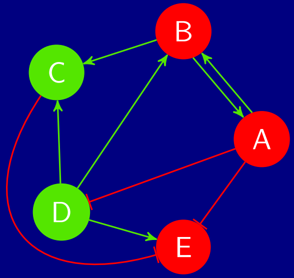
Local Consistency:

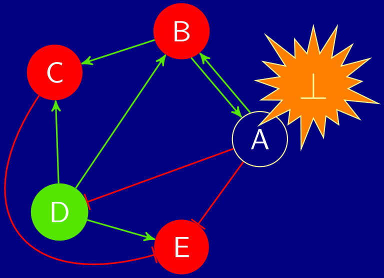
- A variation is consistent **iff** it is explained by some influence



Global Consistency:

- A (partially) labeled influence graph is consistent **iff** there is a total labeling such that every variation is explained

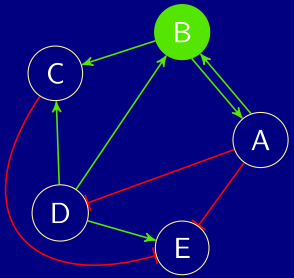




Predicting Variations
under Consistency

A partially labeled influence graph may admit several solutions.

Example:

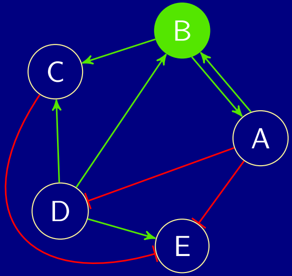


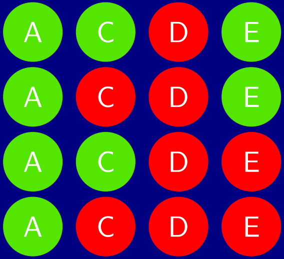
Predicted Variations:

Predicting Variations
under Consistency

A partially labeled influence graph may admit several solutions.

Example:



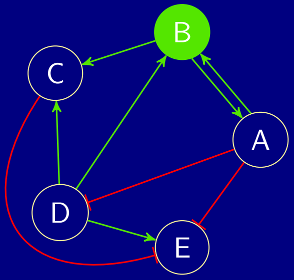


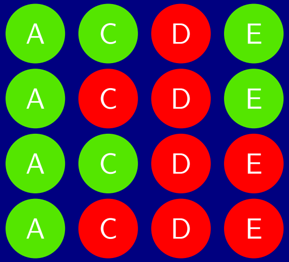
Predicted Variations:

Predicting Variations
under Consistency

A partially labeled influence graph may admit several solutions.

Example:



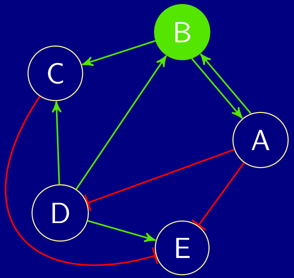


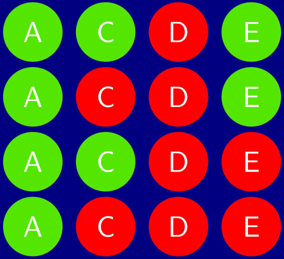
Predicted Variations:

Predicting Variations
under Consistency

A partially labeled influence graph may admit several solutions.

Example:



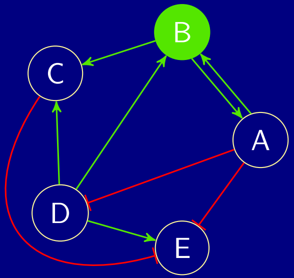


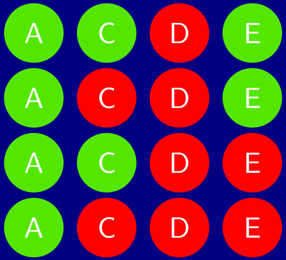
Predicted Variations:

Predicting Variations
under Consistency

A partially labeled influence graph may admit several solutions.

Example:





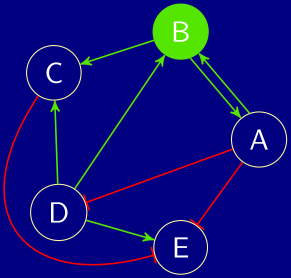
Predicted Variations:

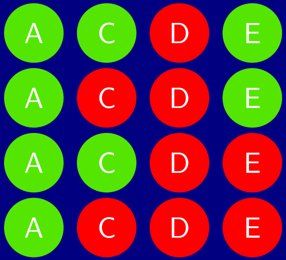
Predicting Variations

under Consistency

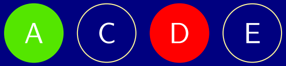
A partially labeled influence graph may admit several solutions.

Example:





Predicted Variations:



Influence Graphs and Variations

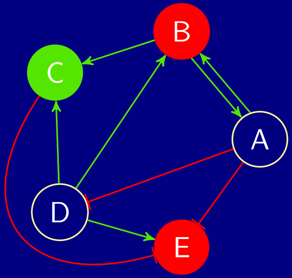
Vertices: *vertex*(i).

Edges: *edge*(j, i).

- — *observedE*(j, i, +1).
- — *observedE*(j, i, −1).

Variations:

- ● *observedV*(i, +1).
- ● *observedV*(i, −1).



Example:

vertex(A). ... *vertex*(E).

edge(A, B). *edge*(A, D). ... *edge*(D, C). *edge*(D, E).

observedE(A, B, +1). *observedE*(A, D, −1). ...
observedE(D, C, +1). *observedE*(D, E, +1).

observedV(B, −1). *observedV*(C, +1). *observedV*(E, −1).

Influence Graphs and Variations

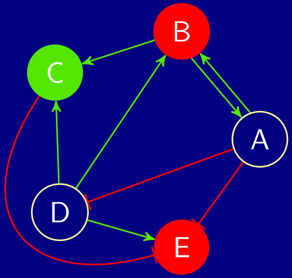
Vertices: *vertex*(i).

Edges: *edge*(j, i).

- *observedE*$(j, i, +1)$.
- *observedE*$(j, i, -1)$.

Variations:

- *observedV*$(i, +1)$.
- *observedV*$(i, -1)$.



Example:

vertex(A). ... *vertex*(E).

edge(A, B). *edge*(A, D). ... *edge*(D, C). *edge*(D, E).

observedE(A, B, +1). *observedE*(A, D, −1). ...
observedE(D, C, +1). *observedE*(D, E, +1).

observedV(B, −1). *observedV*(C, +1). *observedV*(E, −1).

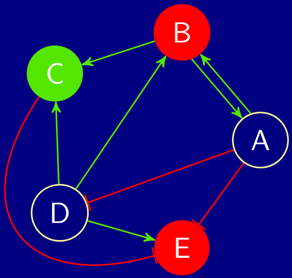
Generating Total Labelings

Edge Labels:

$$1\{labelE(J,I,+1), labelE(J,I,-1)\}1 \leftarrow edge(J,I).$$
$$labelE(J,I,S) \leftarrow observedE(J,I,S).$$

Vertex Labels:

$$1\{labelV(I,+1), labelV(I,-1)\}1 \leftarrow vertex(I).$$
$$labelV(I,S) \leftarrow observedV(I,S).$$



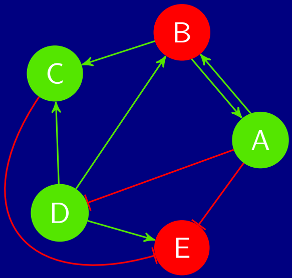
Generating Total Labelings

Edge Labels:

$$1\{labelE(J,I,+1), labelE(J,I,-1)\}1 \leftarrow edge(J,I).$$
$$labelE(J,I,S) \leftarrow observedE(J,I,S).$$

Vertex Labels:

$$1\{labelV(I,+1), labelV(I,-1)\}1 \leftarrow vertex(I).$$
$$labelV(I,S) \leftarrow observedV(I,S).$$



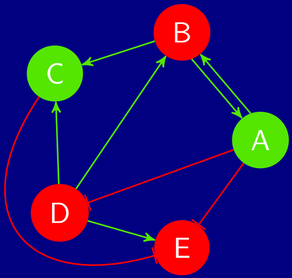
Generating Total Labelings

Edge Labels:

$$1\{labelE(J,I,+1), labelE(J,I,-1)\}1 \leftarrow edge(J,I).$$
$$labelE(J,I,S) \leftarrow observedE(J,I,S).$$

Vertex Labels:

$$1\{labelV(I,+1), labelV(I,-1)\}1 \leftarrow vertex(I).$$
$$labelV(I,S) \leftarrow observedV(I,S).$$



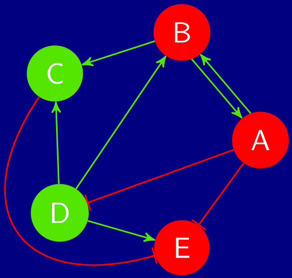
Generating Total Labelings

Edge Labels:

$$1\{labelE(J,I,+1), labelE(J,I,-1)\}1 \leftarrow edge(J,I).$$
$$labelE(J,I,S) \leftarrow observedE(J,I,S).$$

Vertex Labels:

$$1\{labelV(I,+1), labelV(I,-1)\}1 \leftarrow vertex(I).$$
$$labelV(I,S) \leftarrow observedV(I,S).$$



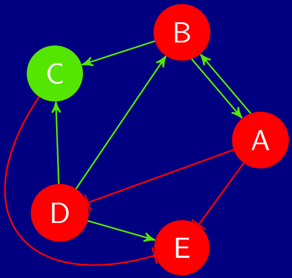
Generating Total Labelings

Edge Labels:

$$1\{labelE(J,I,+1), labelE(J,I,-1)\}1 \leftarrow edge(J,I).$$
$$labelE(J,I,S) \leftarrow observedE(J,I,S).$$

Vertex Labels:

$$1\{labelV(I,+1), labelV(I,-1)\}1 \leftarrow vertex(I).$$
$$labelV(I,S) \leftarrow observedV(I,S).$$



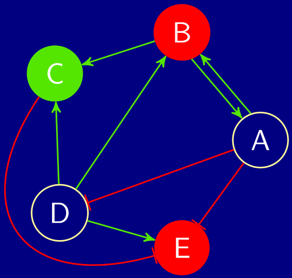
Testing Total Labelings

Influences:

$receive(I, S * T) \leftarrow labelE(J, I, S), labelV(J, T).$

Sign Consistency:

$\leftarrow labelV(I, S), not\ receive(I, S).$



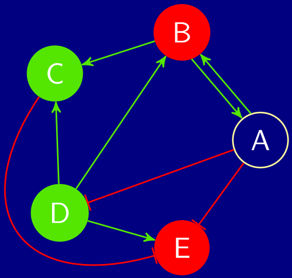
Testing Total Labelings

Influences:

$receive(I, S * T) \leftarrow labelE(J, I, S), labelV(J, T).$

Sign Consistency:

$\leftarrow labelV(I, S), not\ receive(I, S).$



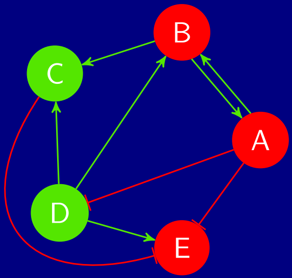
Testing Total Labelings

Influences:

$receive(I, S * T) \leftarrow labelE(J, I, S), labelV(J, T).$

Sign Consistency:

$\leftarrow labelV(I, S), not\ receive(I, S).$



Motivation

Observation: Regulatory networks and experimental profiles are often inconsistent with each other!

Question: How to predict unobserved variations in this case?

Idea:

1. Repair inconsistencies
2. Predict from repaired networks and/or profiles

Motivation

Observation: Regulatory networks and experimental profiles are often inconsistent with each other!

Question: How to predict unobserved variations in this case?

Idea:

1. Repair inconsistencies
2. Predict from repaired networks and/or profiles

Repairing Networks and/or Profiles

Network Repair:

- Adding edges completes an incomplete network (w.r.t. profiles)
- Flipping edge labels curates an improper network
- Making vertices input indicates incompleteness or oscillations

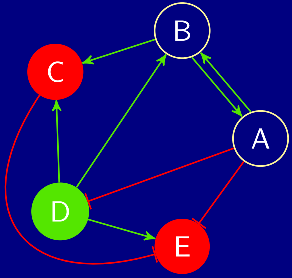
Profile Repair:

- Flipping vertex labels indicates aberrant experimental data

Repair Operations

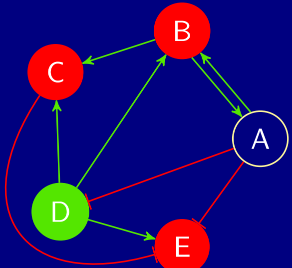
Adding Edges

$$rep(add\_e(U, V)) \leftarrow vertex(U), vertex(V), U \neq V, not\ edge(U, V).$$



Repair Operations
Adding Edges

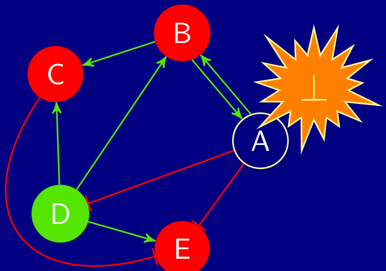
$$rep(add\_e(U,V)) \leftarrow vertex(U), vertex(V), U \neq V, not\ edge(U,V).$$



Repair Operations

Adding Edges

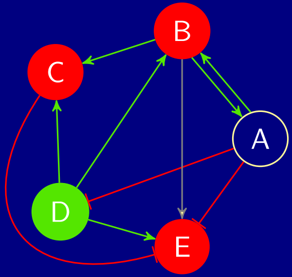
$$rep(add\_e(U,V)) \leftarrow vertex(U), vertex(V), U \neq V, not\ edge(U,V).$$



Repair Operations

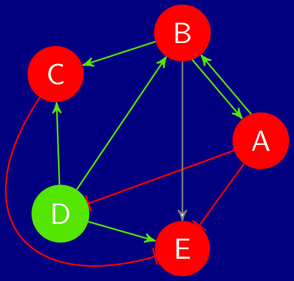
Adding Edges

$$rep(add\_e(U,V)) \leftarrow vertex(U), vertex(V), U \neq V, not\ edge(U,V).$$



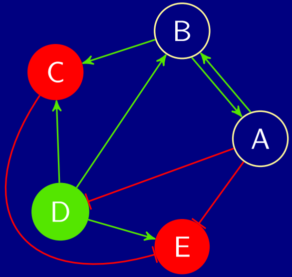
Repair Operations
Adding Edges

$$rep(add\_e(U,V)) \leftarrow vertex(U), vertex(V), U \neq V, not\ edge(U,V).$$



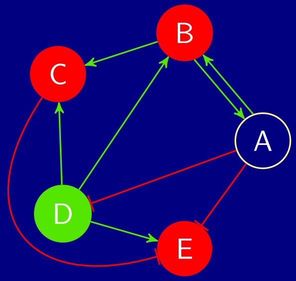
Repair Operations
Flipping Edge Labels

$$rep(flip\_e(U,V,S)) \leftarrow observedE(U,V,S).$$



Repair Operations
Flipping Edge Labels

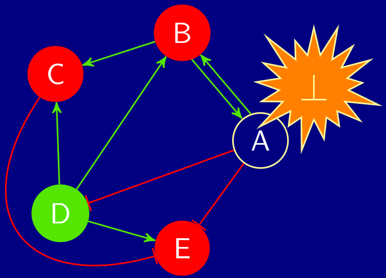
$rep(flip\_e(U, V, S)) \leftarrow observedE(U, V, S).$



Repair Operations

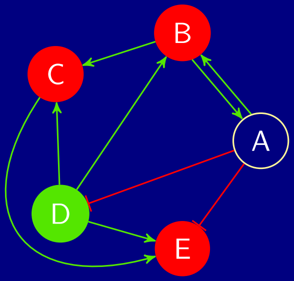
Flipping Edge Labels

$$rep(flip\_e(U,V,S)) \leftarrow observedE(U,V,S).$$



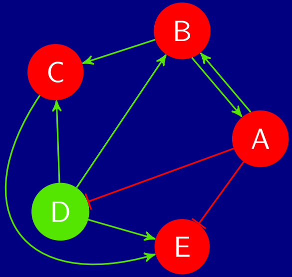
Repair Operations
Flipping Edge Labels

$rep(flip\_e(U, V, S)) \leftarrow observedE(U, V, S).$



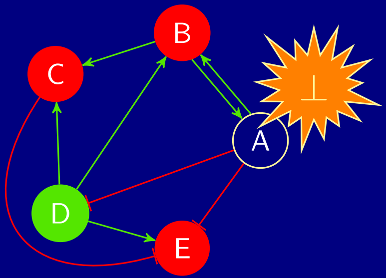
Repair Operations
Flipping Edge Labels

$$rep(flip\_e(U, V, S)) \leftarrow observedE(U, V, S).$$



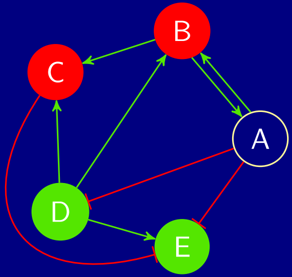
Repair Operations
Flipping Vertex Labels

$rep(flip\_v(V, S)) \leftarrow observedV(V, S).$



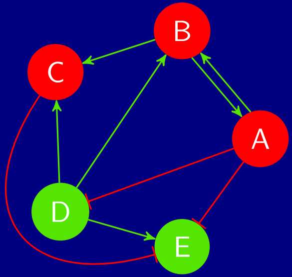
Repair Operations
Flipping Vertex Labels

$rep(flip\_v(V, S)) \leftarrow observedV(V, S).$



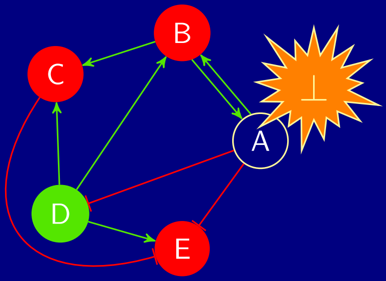
Repair Operations
Flipping Vertex Labels

$rep(flip\_v(V, S)) \leftarrow observedV(V, S).$



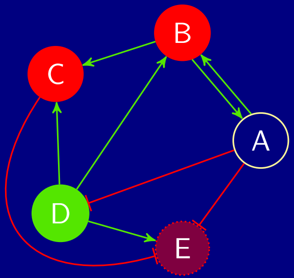
Repair Operations
Making Vertices Input

$$rep(inp\_v(V)) \leftarrow vertex(V), not\ input(V).$$



Repair Operations
Making Vertices Input

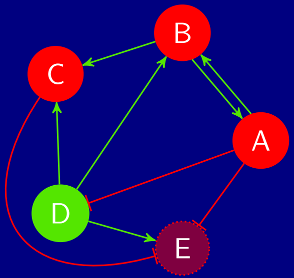
$$rep(inp\_v(V)) \leftarrow vertex(V), not\ input(V).$$



Repair Operations

Making Vertices Input

$$rep(inp\_v(V)) \leftarrow vertex(V), not\ input(V).$$



Generating Total Labelings under Repair

Applying Repair Operations:

$0\{app(R)\}1 \leftarrow rep(R).$

Generating Edge Labelings:

$1\{labelE(U,V,+1), labelE(U,V,-1)\}1 \leftarrow edge(U,V).$
$1\{labelE(U,V,+1), labelE(U,V,-1)\}1 \leftarrow app(add\_e(U,V)).$
$labelE(U,V,S) \leftarrow observedE(U,V,S), not\ app(flip\_e(U,V,S)).$
$labelE(U,V,-S) \leftarrow app(flip\_e(U,V,S)).$

Generating Vertex Labelings:

$1\{labelV(V,+1), labelV(V,-1)\}1 \leftarrow vertex(V).$
$labelV(V,S) \leftarrow observedV(V,S), not\ app(flip\_v(V,S)).$
$labelV(V,-S) \leftarrow app(flip\_v(V,S)).$

Generating Total Labelings under Repair

Applying Repair Operations:

$0\{app(R)\}1 \leftarrow rep(R).$

Generating Edge Labelings:

$1\{labelE(U,V,+1), labelE(U,V,-1)\}1 \leftarrow edge(U,V).$
$1\{labelE(U,V,+1), labelE(U,V,-1)\}1 \leftarrow app(add\_e(U,V)).$
$labelE(U,V,S) \leftarrow observedE(U,V,S), not\ app(flip\_e(U,V,S)).$
$labelE(U,V,-S) \leftarrow app(flip\_e(U,V,S)).$

Generating Vertex Labelings:

$1\{labelV(V,+1), labelV(V,-1)\}1 \leftarrow vertex(V).$
$labelV(V,S) \leftarrow observedV(V,S), not\ app(flip\_v(V,S)).$
$labelV(V,-S) \leftarrow app(flip\_v(V,S)).$

Testing Total Labelings under Repair

Enforcing Sign Consistency Constraints:

$$\begin{aligned} receive(I, S*T) &\leftarrow labelE(J, I, S), labelV(J, T). \\ &\leftarrow labelV(I, S), not\ receive(I, S), \\ &\quad\ not\ input(V), not\ app(inp\_v(V)). \end{aligned}$$

Minimal Repair

Goal:

Minimal change of networks/profiles
(re)establishing consistency

Implementation (cardinality minimality):

$$\#minimize\{app(R) : rep(R)\}.$$

- See KR'10 paper for disjunctive subset minimality encoding
- **NEW**(@ICLP'11): subset minimality via meta-programming

Testing Total Labelings under Repair

Enforcing Sign Consistency Constraints:

$$receive(I, S * T) \leftarrow labelE(J, I, S), labelV(J, T).$$
$$\leftarrow labelV(I, S), not\ receive(I, S),$$
$$not\ input(V), not\ app(inp\_v(V)).$$

Minimal Repair

Goal:

Minimal change of networks/profiles
(re)establishing consistency

Implementation (cardinality minimality):

$$\#minimize\{app(R) : rep(R)\}.$$

- See KR'10 paper for disjunctive subset minimality encoding
- **NEW**(@ICLP'11): subset minimality via meta-programming

Testing Total Labelings under Repair

Enforcing Sign Consistency Constraints:

$$receive(I, S * T) \leftarrow labelE(J, I, S), labelV(J, T).$$
$$\leftarrow labelV(I, S), not\ receive(I, S), not\ input(V), not\ app(inp\_v(V)).$$

Minimal Repair

Goal:

Minimal change of networks/profiles
(re)establishing consistency

Implementation (cardinality minimality):

$$\#minimize\{app(R) : rep(R)\}.$$

- See KR'10 paper for disjunctive subset minimality encoding
- **NEW**(@ICLP'11): subset minimality via meta-programming

Predicting under Repair

Two Phase Approach:

1. Compute minimal number of required repair operations
2. Intersect consistent labelings under minimal repair
   - Cautious reasoning (supported by answer set solver `clasp`)

Predicting Variations

under Inconsistency

- Transcriptional network of *Escherichia coli*, obtained from RegulonDB by Gama-Castro *et al.* [2008], consisting of
  - 5150 interactions between 1914 genes
- Two datasets
  - Exponential-Stationary growth shift by Bradley *et al.* [2007]
  - Heatshock by Allen *et al.* [2003]
- The data of both experiments is highly noisy and inconsistent with the (well-curated) RegulonDB model
- For enabling prediction rate and accuracy assessment, we randomly select samples of significantly expressed genes (3%,6%,9%,12%,15% of the whole data, 200 samples each) and use them for testing both our repair modes and prediction

Predicting Variations
under Inconsistency

- Transcriptional network of *Escherichia coli*, obtained from RegulonDB by Gama-Castro *et al.* [2008], consisting of
  - 5150 interactions between 1914 genes
- Two datasets
  - Exponential-Stationary growth shift by Bradley *et al.* [2007]
  - Heatshock by Allen *et al.* [2003]
- The data of both experiments is highly noisy and inconsistent with the (well-curated) RegulonDB model
- For enabling prediction rate and accuracy assessment, we randomly select samples of significantly expressed genes (3%,6%,9%,12%,15% of the whole data, 200 samples each) and use them for testing both our repair modes and prediction

Repair and Prediction Times

	Repair			Exponential-Stationary 3%	6%	9%	12%	15%	Heatshock 3%	6%	9%	12%	15%
Repair	e			6.58	8.44	11.60	14.88	26.20	25.54	42.76	50.46	69.23	84.77
		i		2.18	2.15	2.21	2.23	2.21	2.10	2.13	2.13	2.05	2.08
			v	1.41	1.40	1.40	1.41	1.37	1.41	1.47	1.42	1.37	1.39
	e	i		73.16	202.66	392.97	518.50	574.85	120.91	374.69	553.00	593.20	595.99
	e		v	28.53	85.17	189.27	327.98	470.48	67.92	236.05	465.92	579.88	596.17
		i	v	2.09	2.14	2.45	3.08	6.06	2.27	4.94	60.63	257.68	418.93
	e	i	v	133.84	391.60	538.93	593.33	600.00	232.29	542.48	593.88	600.00	600.00
Prediction	e			13.27	12.19	14.76	15.34	25.90	25.77	37.18	29.09	36.23	41.88
		i		6.18	5.26	4.77	4.60	4.42	6.57	5.93	5.17	4.86	4.54
			v	4.64	4.45	4.39	4.40	4.30	4.86	5.06	5.34	5.42	5.52
	e	i		35.25	97.66	293.80	456.55	550.33	85.47	293.28	524.19	591.81	594.74
	e		v	14.35	26.17	90.17	200.25	363.36	23.32	111.99	338.95	545.56	591.23
		i	v	6.43	5.75	6.27	6.69	8.61	6.91	6.63	30.33	176.14	371.95
	e	i	v	42.51	248.30	468.71	579.58	—	101.82	466.91	585.64	—	—

'e': flipping edge labels 'i': making vertices input 'v': flipping vertex labels

Repair and Prediction Times

	Repair	Exponential-Stationary 3%	6%	9%	12%	15%	Heatshock 3%	6%	9%	12%	15%
Repair	e	6.58	8.44	11.60	14.88	26.20	25.54	42.76	50.46	69.23	84.77
	i	2.18	2.15	2.21	2.23	2.21	2.10	2.13	2.13	2.05	2.08
	v	1.41	1.40	1.40	1.41	1.37	1.41	1.47	1.42	1.37	1.39
	e i	73.16	202.66	392.97	518.50	574.85	120.91	374.69	553.00	593.20	595.99
	e v	28.53	85.17	189.27	327.98	470.48	67.92	236.05	465.92	579.88	596.17
	i v	2.09	2.14	2.45	3.08	6.06	2.27	4.94	60.63	257.68	418.93
	e i v	133.84	391.60	538.93	593.33	600.00	232.29	542.48	593.88	600.00	600.00
Prediction	e	13.27	12.19	14.76	15.34	25.90	25.77	37.18	29.09	36.23	41.88
	i	6.18	5.26	4.77	4.60	4.42	6.57	5.93	5.17	4.86	4.54
	v	4.64	4.45	4.39	4.40	4.30	4.86	5.06	5.34	5.42	5.52
	e i	35.25	97.66	293.80	456.55	550.33	85.47	293.28	524.19	591.81	594.74
	e v	14.35	26.17	90.17	200.25	363.36	23.32	111.99	338.95	545.56	591.23
	i v	6.43	5.75	6.27	6.69	8.61	6.91	6.63	30.33	176.14	371.95
	e i v	42.51	248.30	468.71	579.58	—	101.82	466.91	585.64	—	—

'e': flipping edge labels 'i': making vertices input 'v': flipping vertex labels

Repair and Prediction Times

	Repair			Exponential-Stationary 3%	Exponential-Stationary 6%	Exponential-Stationary 9%	Exponential-Stationary 12%	Exponential-Stationary 15%	Heatshock 3%	Heatshock 6%	Heatshock 9%	Heatshock 12%	Heatshock 15%
Repair	e			6.58	8.44	11.60	14.88	26.20	25.54	42.76	50.46	69.23	84.77
Repair		i		2.18	2.15	2.21	2.23	2.21	2.10	2.13	2.13	2.05	2.08
Repair			v	1.41	1.40	1.40	1.41	1.37	1.41	1.47	1.42	1.37	1.39
Repair	e	i		73.16	202.66	392.97	518.50	574.85	120.91	374.69	553.00	593.20	595.99
Repair	e		v	28.53	85.17	189.27	327.98	470.48	67.92	236.05	465.92	579.88	596.17
Repair		i	v	2.09	2.14	2.45	3.08	6.06	2.27	4.94	60.63	257.68	418.93
Repair	e	i	v	133.84	391.60	538.93	593.33	600.00	232.29	542.48	593.88	600.00	600.00
Prediction	e			13.27	12.19	14.76	15.34	25.90	25.77	37.18	29.09	36.23	41.88
Prediction		i		6.18	5.26	4.77	4.60	4.42	6.57	5.93	5.17	4.86	4.54
Prediction			v	4.64	4.45	4.39	4.40	4.30	4.86	5.06	5.34	5.42	5.52
Prediction	e	i		35.25	97.66	293.80	456.55	550.33	85.47	293.28	524.19	591.81	594.74
Prediction	e		v	14.35	26.17	90.17	200.25	363.36	23.32	111.99	338.95	545.56	591.23
Prediction		i	v	6.43	5.75	6.27	6.69	8.61	6.91	6.63	30.33	176.14	371.95
Prediction	e	i	v	42.51	248.30	468.71	579.58	—	101.82	466.91	585.64	—	—

'e': flipping edge labels 'i': making vertices input 'v': flipping vertex labels

Repair and Prediction Times

It's feasible!

				Exponential-Stationary					Heatshock				
Repair				3%	6%	9%	12%	15%	3%	6%	9%	12%	15%
Repair	e			6.58	8.44	11.60	14.88	26.20	25.54	42.76	50.46	69.23	84.77
		i		2.18	2.15	2.21	2.23	2.21	2.10	2.13	2.13	2.05	2.08
			v	1.41	1.40	1.40	1.41	1.37	1.41	1.47	1.42	1.37	1.39
	e	i		73.16	202.66	392.97	518.50	574.85	120.91	374.69	553.00	593.20	595.99
	e		v	28.53	85.17	189.27	327.98	470.48	67.92	236.05	465.92	579.88	596.17
		i	v	2.09	2.14	2.45	3.08	6.06	2.27	4.94	60.63	257.68	418.93
	e	i	v	133.84	391.60	538.93	593.33	600.00	232.29	542.48	593.88	600.00	600.00
Prediction	e			13.27	12.19	14.76	15.34	25.90	25.77	37.18	29.09	36.23	41.88
		i		6.18	5.26	4.77	4.60	4.42	6.57	5.93	5.17	4.86	4.54
			v	4.64	4.45	4.39	4.40	4.30	4.86	5.06	5.34	5.42	5.52
	e	i		35.25	97.66	293.80	456.55	550.33	85.47	293.28	524.19	591.81	594.74
	e		v	14.35	26.17	90.17	200.25	363.36	23.32	111.99	338.95	545.56	591.23
		i	v	6.43	5.75	6.27	6.69	8.61	6.91	6.63	30.33	176.14	371.95
	e	i	v	42.51	248.30	468.71	579.58	—	101.82	466.91	585.64	—	—

'e': flipping edge labels 'i': making vertices input 'v': flipping vertex labels

Prediction Rate and Accuracy in Percent

	Repair			Exponential-Stationary					Heatshock				
				3%	6%	9%	12%	15%	3%	6%	9%	12%	15%
Rate	e			15.00	18.51	20.93	22.79	23.94	15.47	19.54	21.87	23.17	24.78
Rate		i		15.00	18.51	20.93	22.79	23.93	15.48	19.62	21.89	23.20	24.80
Rate			v	14.90	18.37	20.86	22.73	23.77	15.32	19.59	21.37	22.13	23.79
Rate	e	i		14.92	18.61	20.55	21.96	22.80	15.37	19.62	22.83	23.44	24.05
Rate	e		v	14.89	18.33	21.07	22.52	23.74	15.33	19.21	21.00	22.65	24.90
Rate		i	v	14.89	18.33	20.79	22.59	23.66	15.41	19.47	21.36	21.81	23.55
Rate	e	i	v	14.58	19.00	20.29	21.13	—	15.01	19.11	22.52	—	—
Accuracy	e			90.93	91.98	92.42	92.70	92.81	91.87	92.93	92.92	92.83	92.71
Accuracy		i		90.93	91.98	92.42	92.70	92.81	91.93	92.90	92.94	92.87	92.76
Accuracy			v	90.99	92.05	92.44	92.73	92.89	92.29	93.27	93.88	94.27	94.36
Accuracy	e	i		91.09	91.90	92.57	93.03	93.19	91.99	92.49	91.16	93.62	94.44
Accuracy	e		v	90.99	92.03	92.50	92.82	92.94	92.30	93.37	93.66	94.36	94.35
Accuracy		i	v	90.99	92.03	92.42	92.71	92.87	92.24	93.34	93.90	94.26	94.38
Accuracy	e	i	v	91.35	92.29	92.52	93.04	—	92.26	93.04	91.78	—	—

'e': flipping edge labels 'i': making vertices input 'v': flipping vertex labels

Prediction Rate and Accuracy in Percent

	Repair	Exponential-Stationary 3%	6%	9%	12%	15%	Heatshock 3%	6%	9%	12%	15%
Rate	e	15.00	18.51	20.93	22.79	23.94	15.47	19.54	21.87	23.17	24.78
	i	15.00	18.51	20.93	22.79	23.93	15.48	19.62	21.89	23.20	24.80
	v	14.90	18.37	20.86	22.73	23.77	15.32	19.59	21.37	22.13	23.79
	e i	14.92	18.61	20.55	21.96	22.80	15.37	19.62	22.83	23.44	24.05
	e v	14.89	18.33	21.07	22.52	23.74	15.33	19.21	21.00	22.65	24.90
	i v	14.89	18.33	20.79	22.59	23.66	15.41	19.47	21.36	21.81	23.55
	e i v	14.58	19.00	20.29	21.13	—	15.01	19.11	22.52	—	—
Accuracy	e	90.93	91.98	92.42	92.70	92.81	91.87	92.93	92.92	92.83	92.71
	i	90.93	91.98	92.42	92.70	92.81	91.93	92.90	92.94	92.87	92.76
	v	90.99	92.05	92.44	92.73	92.89	92.29	93.27	93.88	94.27	94.36
	e i	91.09	91.90	92.57	93.03	93.19	91.99	92.49	91.16	93.62	94.44
	e v	90.99	92.03	92.50	92.82	92.94	92.30	93.37	93.66	94.36	94.35
	i v	90.99	92.03	92.42	92.71	92.87	92.24	93.34	93.90	94.26	94.38
	e i v	91.35	92.29	92.52	93.04	—	92.26	93.04	91.78	—	—

'e': flipping edge labels 'i': making vertices input 'v': flipping vertex labels

Prediction Rate and Accuracy in Percent

	Repair			Exponential-Stationary					Heatshock				
				3%	6%	9%	12%	15%	3%	6%	9%	12%	15%
Rate	e			15.00	18.51	20.93	22.79	23.94	15.47	19.54	21.87	23.17	24.78
Rate		i		15.00	18.51	20.93	22.79	23.93	15.48	19.62	21.89	23.20	24.80
Rate			v	14.90	18.37	20.86	22.73	23.77	15.32	19.59	21.37	22.13	23.79
Rate	e	i		14.92	18.61	20.55	21.96	22.80	15.37	19.62	22.83	23.44	24.05
Rate	e		v	14.89	18.33	21.07	22.52	23.74	15.33	19.21	21.00	22.65	24.90
Rate		i	v	14.89	18.33	20.79	22.59	23.66	15.41	19.47	21.36	21.81	23.55
Rate	e	i	v	14.58	19.00	20.29	21.13	—	15.01	19.11	22.52	—	—
Accuracy	e			90.93	91.98	92.42	92.70	92.81	91.87	92.93	92.92	92.83	92.71
Accuracy		i		90.93	91.98	92.42	92.70	92.81	91.93	92.90	92.94	92.87	92.76
Accuracy			v	90.99	92.05	92.44	92.73	92.89	92.29	93.27	93.88	94.27	94.36
Accuracy	e	i		91.09	91.90	92.57	93.03	93.19	91.99	92.49	91.16	93.62	94.44
Accuracy	e		v	90.99	92.03	92.50	92.82	92.94	92.30	93.37	93.66	94.36	94.35
Accuracy		i	v	90.99	92.03	92.42	92.71	92.87	92.24	93.34	93.90	94.26	94.38
Accuracy	e	i	v	91.35	92.29	92.52	93.04	—	92.26	93.04	91.78	—	—

'e': flipping edge labels 'i': making vertices input 'v': flipping vertex labels

Prediction Rate and Accuracy in Percent

	Repair			Exponential-Stationary 3%	6%	9%	12%	15%	Heatshock 3%	6%	9%	12%	15%
Rate	e			15.00	18.51	20.93	22.79	23.94	15.47	19.54	21.87	23.17	24.78
Rate		i		15.00	18.51	20.93	22.79	23.93	15.48	19.62	21.89	23.20	24.80
Rate			v	14.90	18.37	20.86	22.73	23.77	15.32	19.59	21.37	22.13	23.79
Rate	e	i		14.92	18.61	20.55	21.96	22.80	15.37	19.62	22.83	23.44	24.05
Rate	e		v	14.89	18.33	21.07	22.52	23.74	15.33	19.21	21.00	22.65	24.90
Rate		i	v	14.89	18.33	20.79	22.59	23.66	15.41	19.47	21.36	21.81	23.55
Rate	e	i	v	14.58	19.00	20.29	21.13	—	15.01	19.11	22.52	—	—
Accuracy	e			90.93	91.98	92.42	92.70	92.81	91.87	92.93	92.92	92.83	92.71
Accuracy		i		90.93	91.98	92.42	92.70	92.81	91.93	92.90	92.94	92.87	92.76
Accuracy			v	90.99	92.05	92.44	92.73	92.89	92.29	93.27	93.88	94.2[illegible]	94.36
Accuracy	e	i		91.09	91.90	92.57	93.03	93.19	91.99	92.49	91.16	93.[illegible]2	94.44
Accuracy	e		v	90.99	92.03	92.50	92.82	92.94	92.30	93.37	93.66	94.36	94.35
Accuracy		i	v	90.99	92.03	92.42	92.71	92.87	92.24	93.34	93.90	94.2[illegible]	94.38
Accuracy	e	i	v	91.35	92.29	92.52	93.04	—	92.26	93.0[illegible]	91.78	—	—

Accuracy over 90%!

'e': flipping edge labels 'i': making vertices input 'v': flipping vertex labels

Subset-Minimal Repairs

Direct Encoding versus Meta-Programming

- 100 samples per repair mode
- 4,000 seconds time(out) per run

Repair	direct Σ time	direct Σ out	meta Σ time	meta Σ out
e	365,227	78	366,798	79
i	45,736	0	42,203	2
v	315,801	72	4,823	0

'e': flipping edge labels 'i': making vertices input 'v': flipping vertex labels

Subset-Minimal Repairs

Direct Encoding versus Meta-Programming

- 100 samples per repair mode
- 4,000 seconds time(out) per run

	direct		**meta**	
Repair	Σ time	Σ out	Σ time	Σ out
e	365,227	78	366,798	79
i	45,736	0	42,203	2
v	**315,801**	**72**	**4,823**	**0**

'e': flipping edge labels   'i': making vertices input   'v': flipping vertex labels

Incremental Grounding and Solving
Overview

Motivation

Many real-world applications, having exponential state spaces, like

- bio-informatics,
- planning,
- model checking,
- etc.

have associated PSPACE-decision problems.

☞ For instance, the plan existence problem of deterministic planning is PSPACE-complete.

But the problem of whether there is a plan having a length bounded by a given polynomial is in NP.

Motivation

Many real-world applications, having exponential state spaces, like

- bio-informatics,
- planning,
- model checking,
- etc.

have associated PSPACE-decision problems.

☞ For instance, the plan existence problem of deterministic planning is PSPACE-complete.

But the problem of whether there is a plan having a length bounded by a given polynomial is in NP.

Motivation

Many real-world applications, having exponential state spaces, like

- bio-informatics,
- planning,
- model checking,
- etc.

have associated PSPACE-decision problems.

☞ For instance, the plan existence problem of deterministic planning is PSPACE-complete.

But the problem of whether there is a plan having a length bounded by a given polynomial is in NP.

Motivation

State of the Art In ASP such problems are dealt with by iterative deepening search.
That is, considering one problem instance after another by gradually increasing the bound on the solution size.

Problem This approach
- is prone to redundancies in grounding and solving, and
- cannot harness modern look-back techniques regarding conflict-driven learning and heuristics.

Goal Avoiding redundancy by gradually processing the extensions to a problem rather than repeatedly re-processing the entire extended problem.

Proposal An incremental approach to both grounding and solving in ASP.

Motivation

State of the Art In ASP such problems are dealt with by iterative deepening search.
That is, considering one problem instance after another by gradually increasing the bound on the solution size.

Problem This approach
- is prone to redundancies in grounding and solving, and
- cannot harness modern look-back techniques regarding conflict-driven learning and heuristics.

Goal Avoiding redundancy by gradually processing the extensions to a problem rather than repeatedly re-processing the entire extended problem.

Proposal An incremental approach to both grounding and solving in ASP.

Motivation

State of the Art In ASP such problems are dealt with by iterative deepening search.
That is, considering one problem instance after another by gradually increasing the bound on the solution size.

Problem This approach

- is prone to redundancies in grounding and solving, and
- cannot harness modern look-back techniques regarding conflict-driven learning and heuristics.

Goal Avoiding redundancy by gradually processing the extensions to a problem rather than repeatedly re-processing the entire extended problem.

Proposal An incremental approach to both grounding and solving in ASP.

Problem Specification

- A (parameterized) domain description is a triple (B, P, Q) of logic programs, among which P and Q contain a (single) parameter k ranging over the natural numbers.
  We sometimes denote P and Q by $P[k]$ and $Q[k]$.
- The base program B is meant to describe static knowledge, independent of parameter k.
  The role of P is to capture knowledge accumulating with increasing k, whereas Q is specific for each value of k.
- One goal is then to decide, for instance, whether the program

$$R[k/i] = B \cup \bigcup_{1 \leq j \leq i} P[k/j] \cup Q[k/i]$$

  has an answer set for some (minimum) integer $i \geq 1$.
  We write $R[i]$ rather than $R[k/i]$ whenever clear from the context.

Problem Specification

- A (parameterized) domain description is a triple (B, P, Q) of logic programs, among which P and Q contain a (single) parameter k ranging over the natural numbers.
  We sometimes denote P and Q by $P[k]$ and $Q[k]$.
- The base program B is meant to describe static knowledge, independent of parameter k.
  The role of P is to capture knowledge accumulating with increasing k, whereas Q is specific for each value of k.
- One goal is then to decide, for instance, whether the program

$$R[k/i] = B \cup \bigcup_{1 \leq j \leq i} P[k/j] \cup Q[k/i]$$

  has an answer set for some (minimum) integer $i \geq 1$.
  We write $R[i]$ rather than $R[k/i]$ whenever clear from the context.

Problem Specification

- A (parameterized) domain description is a triple (B, P, Q) of logic programs, among which P and Q contain a (single) parameter k ranging over the natural numbers.
  We sometimes denote P and Q by $P[k]$ and $Q[k]$.
- The base program B is meant to describe static knowledge, independent of parameter k.
  The role of P is to capture knowledge accumulating with increasing k, whereas Q is specific for each value of k.
- One goal is then to decide, for instance, whether the program

$$R[k/i] = B \cup \bigcup_{1 \leq j \leq i} P[k/j] \cup Q[k/i]$$

  has an answer set for some (minimum) integer $i \geq 1$.
  We write $R[i]$ rather than $R[k/i]$ whenever clear from the context.

Problem Specification

- A (parameterized) domain description is a triple (B, P, Q) of logic programs, among which P and Q contain a (single) parameter k ranging over the natural numbers.
  We sometimes denote P and Q by $P[k]$ and $Q[k]$.
- The base program B is meant to describe static knowledge, independent of parameter k.
  The role of P is to capture knowledge accumulating with increasing k, whereas Q is specific for each value of k.
- One goal is then to decide, for instance, whether the program

$$R[k/i] = B \cup \bigcup_{1 \leq j \leq i} P[k/j] \cup Q[k/i]$$

  has an answer set for some (minimum) integer $i \geq 1$.
  We write $R[i]$ rather than $R[k/i]$ whenever clear from the context.

Grounding and Solving, classically

Input A domain description $R[k] = (B, P[k], Q[k])$.

Output A non-empty set of answer sets of $R[k/i]$, for instance.

1. Ground $B \cup P[1] \cup Q[1]$ and Solve $B \cup P[1] \cup Q[1]$
2. Ground $B \cup P[1] \cup P[2] \cup Q[2]$ and Solve $B \cup P[1] \cup P[2] \cup Q[2]$
3. Ground $B \cup P[1] \cup P[2] \cup P[3] \cup Q[3]$ and Solve $B \cup P[1] \cup P[2] \cup P[3] \cup Q[3]$

i. etc. until an answer set is obtained.

Grounding and Solving, classically

Input A domain description $R[k] = (B, P[k], Q[k])$.

Output A non-empty set of answer sets of $R[k/i]$, for instance.

1. Ground $B \cup P[1] \cup Q[1]$ and Solve $B \cup P[1] \cup Q[1]$
2. Ground $B \cup P[1] \cup P[2] \cup Q[2]$ and Solve $B \cup P[1] \cup P[2] \cup Q[2]$
3. Ground $B \cup P[1] \cup P[2] \cup P[3] \cup Q[3]$ and Solve $B \cup P[1] \cup P[2] \cup P[3] \cup Q[3]$

i. etc. until an answer set is obtained.

Grounding and Solving, classically

Input A domain description $R[k] = (B, P[k], Q[k])$.

Output A non-empty set of answer sets of $R[k/i]$, for instance.

1. Ground $B \cup P[1] \cup Q[1]$ and Solve $B \cup P[1] \cup Q[1]$
2. Ground $B \cup P[1] \cup P[2] \cup Q[2]$ and Solve $B \cup P[1] \cup P[2] \cup Q[2]$
3. Ground $B \cup P[1] \cup P[2] \cup P[3] \cup Q[3]$ and Solve $B \cup P[1] \cup P[2] \cup P[3] \cup Q[3]$

i. etc. until an answer set is obtained.

Grounding and Solving, classically

Input A domain description $R[k] = (B, P[k], Q[k])$.

Output A non-empty set of answer sets of $R[k/i]$, for instance.

1. Ground $B \cup P[1] \cup Q[1]$ and Solve $B \cup P[1] \cup Q[1]$
2. Ground $B \cup P[1] \cup P[2] \cup Q[2]$ and Solve $B \cup P[1] \cup P[2] \cup Q[2]$
3. Ground $B \cup P[1] \cup P[2] \cup P[3] \cup Q[3]$ and Solve $B \cup P[1] \cup P[2] \cup P[3] \cup Q[3]$

i. etc. until an answer set is obtained.

Grounding and Solving, classically

Input A domain description $R[k] = (B, P[k], Q[k])$.

Output A non-empty set of answer sets of $R[k/i]$, for instance.

1. Ground $B \cup P[1] \cup Q[1]$ and Solve $B \cup P[1] \cup Q[1]$
2. Ground $B \cup P[1] \cup P[2] \cup Q[2]$ and Solve $B \cup P[1] \cup P[2] \cup Q[2]$
3. Ground $B \cup P[1] \cup P[2] \cup P[3] \cup Q[3]$ and
   Solve $B \cup P[1] \cup P[2] \cup P[3] \cup Q[3]$

i. etc. until an answer set is obtained.

Grounding and Solving, incrementally

Input A domain description $R[k] = (B, P[k], Q[k])$.

Output A non-empty set of answer sets of $R[k/i]$, for instance.

1. Ground B and Keep B
2. Ground $P[1] \cup Q[1]$, Solve $\underline{B} \cup P[1] \cup Q[1]$,
   Keep $B \cup P[1]$, and Discard $Q[1]$
3. Ground $P[2] \cup Q[2]$, Solve $\underline{B \cup P[1]} \cup P[2] \cup Q[2]$,
   Keep $B \cup P[1] \cup P[2]$, and Discard $Q[2]$
4. Ground $P[3] \cup Q[3]$, Solve $\underline{B \cup P[1] \cup P[2]} \cup P[3] \cup Q[3]$,
   Keep $B \cup P[1] \cup P[2] \cup P[3]$, and Discard $Q[3]$

i. etc. until an answer set is obtained.

Grounding and Solving, incrementally

Input A domain description $R[k] = (B, P[k], Q[k])$.

Output A non-empty set of answer sets of $R[k/i]$, for instance.

1. Ground B and Keep B
2. Ground $P[1] \cup Q[1]$, Solve $\underline{B} \cup P[1] \cup Q[1]$, Keep $B \cup P[1]$, and Discard $Q[1]$
3. Ground $P[2] \cup Q[2]$, Solve $\underline{B \cup P[1]} \cup P[2] \cup Q[2]$, Keep $B \cup P[1] \cup P[2]$, and Discard $Q[2]$
4. Ground $P[3] \cup Q[3]$, Solve $\underline{B \cup P[1] \cup P[2]} \cup P[3] \cup Q[3]$, Keep $B \cup P[1] \cup P[2] \cup P[3]$, and Discard $Q[3]$

i. etc. until an answer set is obtained.

Grounding and Solving, incrementally

Input A domain description $R[k] = (B, P[k], Q[k])$.

Output A non-empty set of answer sets of $R[k/i]$, for instance.

1. Ground B and Keep B
2. Ground $P[1] \cup Q[1]$, Solve $\underline{B} \cup P[1] \cup Q[1]$,
   Keep $B \cup P[1]$, and Discard $Q[1]$
3. Ground $P[2] \cup Q[2]$, Solve $\underline{B \cup P[1]} \cup P[2] \cup Q[2]$,
   Keep $B \cup P[1] \cup P[2]$, and Discard $Q[2]$
4. Ground $P[3] \cup Q[3]$, Solve $\underline{B \cup P[1] \cup P[2]} \cup P[3] \cup Q[3]$,
   Keep $B \cup P[1] \cup P[2] \cup P[3]$, and Discard $Q[3]$

i. etc. until an answer set is obtained.

Grounding and Solving, incrementally

Input: A domain description $R[k] = (B, P[k], Q[k])$.

Output: A non-empty set of answer sets of $R[k/i]$, for instance.

1. Ground B and Keep B
2. Ground $P[1] \cup Q[1]$, Solve $\underline{B} \cup P[1] \cup Q[1]$, Keep $B \cup P[1]$, and Discard $Q[1]$
3. Ground $P[2] \cup Q[2]$, Solve $\underline{B \cup P[1]} \cup P[2] \cup Q[2]$, Keep $B \cup P[1] \cup P[2]$, and Discard $Q[2]$
4. Ground $P[3] \cup Q[3]$, Solve $\underline{B \cup P[1] \cup P[2]} \cup P[3] \cup Q[3]$, Keep $B \cup P[1] \cup P[2] \cup P[3]$, and Discard $Q[3]$

i. etc. until an answer set is obtained.

Grounding and Solving, incrementally

Input A domain description $R[k] = (B, P[k], Q[k])$.

Output A non-empty set of answer sets of $R[k/i]$, for instance.

1. Ground B and Keep B
2. Ground $P[1] \cup Q[1]$, Solve $\underline{B} \cup P[1] \cup Q[1]$, Keep $B \cup P[1]$, and Discard $Q[1]$
3. Ground $P[2] \cup Q[2]$, Solve $\underline{B \cup P[1]} \cup P[2] \cup Q[2]$, Keep $B \cup P[1] \cup P[2]$, and Discard $Q[2]$
4. Ground $P[3] \cup Q[3]$, Solve $\underline{B \cup P[1] \cup P[2]} \cup P[3] \cup Q[3]$, Keep $B \cup P[1] \cup P[2] \cup P[3]$, and Discard $Q[3]$

i. etc. until an answer set is obtained.

Grounding and Solving, incrementally

Input A domain description $R[k] = (B, P[k], Q[k])$.

Output A non-empty set of answer sets of $R[k/i]$, for instance.

1. Ground B and Keep B
2. Ground $P[1] \cup Q[1]$, Solve $\underline{B} \cup P[1] \cup Q[1]$,
   Keep $B \cup P[1]$, and Discard $Q[1]$
3. Ground $P[2] \cup Q[2]$, Solve $\underline{B \cup P[1]} \cup P[2] \cup Q[2]$,
   Keep $B \cup P[1] \cup P[2]$, and Discard $Q[2]$
4. Ground $P[3] \cup Q[3]$, Solve $\underline{B \cup P[1] \cup P[2]} \cup P[3] \cup Q[3]$,
   Keep $B \cup P[1] \cup P[2] \cup P[3]$, and Discard $Q[3]$

i. etc. until an answer set is obtained.

Grounding and Solving, incrementally

Input: A domain description $R[k] = (B, P[k], Q[k])$.

Output: A non-empty set of answer sets of $R[k/i]$, for instance.

1. Ground B and Keep B
2. Ground $P[1] \cup Q[1]$, Solve $\underline{B} \cup P[1] \cup Q[1]$,
   Keep $B \cup P[1]$, and Discard $Q[1]$
3. Ground $P[2] \cup Q[2]$, Solve $\underline{B \cup P[1]} \cup P[2] \cup Q[2]$,
   Keep $B \cup P[1] \cup P[2]$, and Discard $Q[2]$
4. Ground $P[3] \cup Q[3]$, Solve $\underline{B \cup P[1] \cup P[2]} \cup P[3] \cup Q[3]$,
   Keep $B \cup P[1] \cup P[2] \cup P[3]$, and Discard $Q[3]$

i. etc. until an answer set is obtained.

Grounding and Solving, incrementally

Input A domain description $R[k] = (B, P[k], Q[k])$.

Output A non-empty set of answer sets of $R[k/i]$, for instance.

1. Ground B and Keep B
2. Ground $P[1] \cup Q[1]$, Solve $\underline{B} \cup P[1] \cup Q[1]$, Keep $B \cup P[1]$, and Discard $Q[1]$
3. Ground $P[2] \cup Q[2]$, Solve $\underline{B \cup P[1]} \cup P[2] \cup Q[2]$, Keep $B \cup P[1] \cup P[2]$, and Discard $Q[2]$
4. Ground $P[3] \cup Q[3]$, Solve $\underline{B \cup P[1] \cup P[2]} \cup P[3] \cup Q[3]$, Keep $B \cup P[1] \cup P[2] \cup P[3]$, and Discard $Q[3]$

i. etc. until an answer set is obtained.

Grounding and Solving, incrementally

Input A domain description $R[k] = (B, P[k], Q[k])$.

Output A non-empty set of answer sets of $R[k/i]$, for instance.

1. Ground B and Keep B
2. Ground $P[1] \cup Q[1]$, Solve $\underline{B} \cup P[1] \cup Q[1]$,
   Keep $B \cup P[1]$, and Discard $Q[1]$
3. Ground $P[2] \cup Q[2]$, Solve $\underline{B \cup P[1]} \cup P[2] \cup Q[2]$,
   Keep $B \cup P[1] \cup P[2]$, and Discard $Q[2]$
4. Ground $P[3] \cup Q[3]$, Solve $\underline{B \cup P[1] \cup P[2]} \cup P[3] \cup Q[3]$,
   Keep $B \cup P[1] \cup P[2] \cup P[3]$, and Discard $Q[3]$

i. etc. until an answer set is obtained.

Grounding and Solving, incrementally

Input A domain description $R[k] = (B, P[k], Q[k])$.

Output A non-empty set of answer sets of $R[k/i]$, for instance.

1. Ground B and Keep B
2. Ground $P[1] \cup Q[1]$, Solve $\underline{B} \cup P[1] \cup Q[1]$,
   Keep $B \cup P[1]$, and Discard $Q[1]$
3. Ground $P[2] \cup Q[2]$, Solve $\underline{B \cup P[1]} \cup P[2] \cup Q[2]$,
   Keep $B \cup P[1] \cup P[2]$, and Discard $Q[2]$
4. Ground $P[3] \cup Q[3]$, Solve $\underline{B \cup P[1] \cup P[2]} \cup P[3] \cup Q[3]$,
   Keep $B \cup P[1] \cup P[2] \cup P[3]$, and Discard $Q[3]$

i. etc. until an answer set is obtained.

An Example

$$
\left.\begin{array}{l}
a \textbf{ causes } p \\
\textbf{exogenous } a \\
\textbf{inertial } p
\end{array}\right\} \mapsto
\left\{\begin{array}{l}
B = \left\{\begin{array}{rcl}
p(0) & \leftarrow & \mathit{not}\ \neg p(0) \\
\neg p(0) & \leftarrow & \mathit{not}\ p(0) \\
 & \leftarrow & p(0), \neg p(0)
\end{array}\right\} \\
P[k] = \left\{\begin{array}{rcl}
a(k) & \leftarrow & \mathit{not}\ \neg a(k) \\
\neg a(k) & \leftarrow & \mathit{not}\ a(k) \\
p(k) & \leftarrow & a(k) \\
p(k) & \leftarrow & p(k-1), \mathit{not}\ \neg p(k) \\
\neg p(k) & \leftarrow & \neg p(k-1), \mathit{not}\ p(k) \\
 & \leftarrow & p(k), \neg p(k) \\
 & \leftarrow & a(k), \neg a(k)
\end{array}\right\}
\end{array}\right.
$$

$$
\left.\begin{array}{l}
\neg p \textbf{ holds at } 0 \\
p \textbf{ holds at } n \\
\neg a \textbf{ occurs at } n
\end{array}\right\} \mapsto
\left\{ Q[k] = \left\{\begin{array}{rl}
\leftarrow & \mathit{not}\ \neg p(0) \\
\leftarrow & \mathit{not}\ p(k) \\
\leftarrow & \mathit{not}\ \neg a(k)
\end{array}\right\}\right.
$$

Module

A module $\mathbb{P}$ is a triple (P, I, O) consisting of

- a (ground) program P over $grd(\mathcal{A})$ and
- sets $I, O \subseteq grd(\mathcal{A})$ such that
  - $I \cap O = \emptyset$,
  - $atom(P) \subseteq I \cup O$, and
  - $head(P) \subseteq O$.

The elements of I and O are called input and output atoms,

- also denoted by $I(\mathbb{P})$ and $O(\mathbb{P})$, respectively;
- similarly, we refer to (ground) program P by $P(\mathbb{P})$.

Module

A module $\mathbb{P}$ is a triple (P, I, O) consisting of

- a (ground) program P over $grd(\mathcal{A})$ and
- sets $I, O \subseteq grd(\mathcal{A})$ such that
  - $I \cap O = \emptyset$,
  - $atom(P) \subseteq I \cup O$, and
  - $head(P) \subseteq O$.

The elements of I and O are called input and output atoms,

- also denoted by $I(\mathbb{P})$ and $O(\mathbb{P})$, respectively;
- similarly, we refer to (ground) program P by $P(\mathbb{P})$.

Recall: Ground Instantiation

- The ground instantiation of a program P is defined as

$$grd(P) = \{r\theta \mid r \in P, \theta : var(r) \to \mathcal{U}\} \text{ , where}$$
$$\mathcal{U} = \{t \in \mathcal{T} \mid var(t) = \emptyset\} \text{ .}$$

- Analogously, $grd(\mathcal{A}) = \{a \in \mathcal{A} \mid var(a) = \emptyset\}$.

- Note that the set $\mathcal{T}$ of terms includes the natural numbers !

Recall: Ground Instantiation

- The ground instantiation of a program P is defined as

$$grd(P) = \{r\theta \mid r \in P, \theta : var(r) \to \mathcal{U}\} \text{ , where}$$
$$\mathcal{U} = \{t \in \mathcal{T} \mid var(t) = \emptyset\} \text{ .}$$

- Analogously, $grd(\mathcal{A}) = \{a \in \mathcal{A} \mid var(a) = \emptyset\}$.

- Note that the set $\mathcal{T}$ of terms includes the natural numbers !

Formal Setting

- For a program P over $grd(\mathcal{A})$ and a set $X \subseteq grd(\mathcal{A})$,

$$P|_X = \{head(r) \leftarrow body^+(r) \cup L \mid \\ r \in P, body^+(r) \subseteq X, L = \{not\ c \mid c \in body^-(r) \cap X\}\} .$$

☞ $P|_X$ projects the bodies of rules in P to the atoms of X.

- For a program P over $\mathcal{A}$ and $I \subseteq grd(\mathcal{A})$, define $\mathbb{P}(I)$ as the module

$$(\, grd(P)|_Y, I, head(grd(P)|_X)\,) \ ,$$

where $X = I \cup head(grd(P))$ and $Y = I \cup head(grd(P)|_X)$.

- Let $\mathbb{P}(I) = (P', I, O)$. Then, we have

$$O \subseteq grd(\mathcal{A}) \quad \text{and} \quad atom(P') \subseteq I \cup O \ .$$

Formal Setting

- For a program P over $grd(\mathcal{A})$ and a set $X \subseteq grd(\mathcal{A})$,

$$P|_X = \{head(r) \leftarrow body^+(r) \cup L \mid r \in P, body^+(r) \subseteq X, L = \{not\ c \mid c \in body^-(r) \cap X\}\} \ .$$

  ☞ $P|_X$ projects the bodies of rules in P to the atoms of X.

- For a program P over $\mathcal{A}$ and $I \subseteq grd(\mathcal{A})$, define $\mathbb{P}(I)$ as the module

$$(\, grd(P)|_Y, I, head(grd(P)|_X)\,)\ ,$$

  where $X = I \cup head(grd(P))$ and $Y = I \cup head(grd(P)|_X)$.

- Let $\mathbb{P}(I) = (P', I, O)$. Then, we have

$$O \subseteq grd(\mathcal{A}) \quad \text{and} \quad atom(P') \subseteq I \cup O\ .$$

Formal Setting

- For a program P over $grd(\mathcal{A})$ and a set $X \subseteq grd(\mathcal{A})$,

$$P|_X = \{head(r) \leftarrow body^+(r) \cup L \mid \\ r \in P, body^+(r) \subseteq X, L = \{not\ c \mid c \in body^-(r) \cap X\}\} \ .$$

  ☞ $P|_X$ projects the bodies of rules in P to the atoms of X.

- For a program P over $\mathcal{A}$ and $I \subseteq grd(\mathcal{A})$, define $\mathbb{P}(I)$ as the module

$$(\, grd(P)|_Y, I, head(grd(P)|_X)\,)\ ,$$

  where $X = I \cup head(grd(P))$ and $Y = I \cup head(grd(P)|_X)$.

- Let $\mathbb{P}(I) = (P', I, O)$. Then, we have

$$O \subseteq grd(\mathcal{A}) \quad \text{and} \quad atom(P') \subseteq I \cup O\ .$$

A Simple Example

Consider

$$P[k] = \{\ p(k) \leftarrow p(Y), \mathit{not}\ p(2) \qquad p(k) \leftarrow p(2)\ \}$$

and note that $grd(P[1])$ is infinite !

For $P[1]$ and $I = \{\ p(0)\ \}$, we get the module

$$(\ grd(P[1])|_{\{p(0),p(1)\}}\ ,\ \{p(0)\}\ ,\ \{p(1)\}\ )$$

where $grd(P[1])|_{\{p(0),p(1)\}} = \{p(1) \leftarrow p(0) \qquad p(1) \leftarrow p(1)\}$.

A Simple Example

Consider

$$P[k] = \{\ p(k) \leftarrow p(Y), not\ p(2) \qquad p(k) \leftarrow p(2)\ \}$$

and note that $grd(P[1])$ is infinite !
For $P[1]$ and $I = \{\ p(0)\ \}$, we get the module

$$(\ grd(P[1])|_{\{p(0),p(1)\}}\ ,\ \{p(0)\}\ ,\ \{p(1)\}\ )$$

where $grd(P[1])|_{\{p(0),p(1)\}} = \{p(1) \leftarrow p(0) \qquad p(1) \leftarrow p(1)\}$.

Modular Domain Description

- Define the join of two modules $\mathbb{P}$ and $\mathbb{Q}$, $\mathbb{P} \sqcup \mathbb{Q}$, as the module

$$(P(\mathbb{P}) \cup P(\mathbb{Q}) , \; I(\mathbb{P}) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P})) , \; O(\mathbb{P}) \cup O(\mathbb{Q})) ,$$

provided that $(I(\mathbb{P}) \cup O(\mathbb{P})) \cap O(\mathbb{Q}) = \emptyset$.
  - ☞ Recursion between two modules to be joined is disallowed.
  - ☞ Recursion is allowed within each module.
- A domain description $(B, P[k], Q[k])$ is modular, if the modules

$$\mathbb{P}_i = \mathbb{P}_{i-1} \sqcup \mathbb{P}[i](O(\mathbb{P}_{i-1})) \quad \text{and} \quad \mathbb{Q}_i = \mathbb{P}_i \sqcup \mathbb{Q}[i](O(\mathbb{P}_i))$$

are defined for $i \geq 1$, where $\mathbb{P}_0 = \mathbb{B}(\emptyset)$.

Modular Domain Description

- Define the join of two modules $\mathbb{P}$ and $\mathbb{Q}$, $\mathbb{P} \sqcup \mathbb{Q}$, as the module

$$(\, P(\mathbb{P}) \cup P(\mathbb{Q}) \, , \; I(\mathbb{P}) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P})) \, , \; O(\mathbb{P}) \cup O(\mathbb{Q}) \,) \; ,$$

  provided that $(I(\mathbb{P}) \cup O(\mathbb{P})) \cap O(\mathbb{Q}) = \emptyset$.
  - ☞ Recursion between two modules to be joined is disallowed.
  - ☞ Recursion is allowed within each module.
- A domain description $(B, P[k], Q[k])$ is modular, if the modules

$$\mathbb{P}_i = \mathbb{P}_{i-1} \sqcup \mathbb{P}[i](O(\mathbb{P}_{i-1})) \quad \text{and} \quad \mathbb{Q}_i = \mathbb{P}_i \sqcup \mathbb{Q}[i](O(\mathbb{P}_i))$$

  are defined for $i \geq 1$, where $\mathbb{P}_0 = \mathbb{B}(\emptyset)$.

Modular Domain Description

- Define the join of two modules $\mathbb{P}$ and $\mathbb{Q}$, $\mathbb{P} \sqcup \mathbb{Q}$, as the module

$$(\ P(\mathbb{P}) \cup P(\mathbb{Q}) \,, \ I(\mathbb{P}) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P})) \,, \ O(\mathbb{P}) \cup O(\mathbb{Q}) \ ) \ ,$$

  provided that $(I(\mathbb{P}) \cup O(\mathbb{P})) \cap O(\mathbb{Q}) = \emptyset$.
  - ☞ Recursion between two modules to be joined is disallowed.
  - ☞ Recursion is allowed within each module.
- A domain description $(B, P[k], Q[k])$ is modular, if the modules

$$\mathbb{P}_i = \mathbb{P}_{i-1} \sqcup \mathbb{P}[i](O(\mathbb{P}_{i-1})) \quad \text{and} \quad \mathbb{Q}_i = \mathbb{P}_i \sqcup \mathbb{Q}[i](O(\mathbb{P}_i))$$

  are defined for $i \geq 1$, where $\mathbb{P}_0 = \mathbb{B}(\emptyset)$.

A Pragmatic Approach

A domain description $(B, P[k], Q[k])$ is modular, if

- atoms defined in B comprise dedicated predicates or 0 as argument,
- atoms defined in $P[k]$ comprise k as argument, and
- atoms defined in $Q[k]$ comprise dedicated predicates and k as argument.

The above conditions can be formalized as follows:

- $atom(grd(B)) \cap \left(\bigcup_{1 \leq i} head(grd(P[i] \cup Q[i]))\right) = \emptyset$,
- $\left(\bigcup_{1 \leq i} atom(grd(P[i]))\right) \cap \left(\bigcup_{1 \leq j} head(grd(Q[j]))\right) = \emptyset$,
- $atom(grd(P[i])) \cap \left(\bigcup_{i < j} head(grd(P[j]))\right) = \emptyset$ for all $1 \leq i$, and
- $atom(grd(Q[i])) \cap \left(\bigcup_{i < j} head(grd(Q[j]))\right) = \emptyset$ for all $1 \leq i$.

A Pragmatic Approach

A domain description $(B, P[k], Q[k])$ is modular, if

- atoms defined in B comprise dedicated predicates or 0 as argument,
- atoms defined in $P[k]$ comprise k as argument, and
- atoms defined in $Q[k]$ comprise dedicated predicates and k as argument.

The above conditions can be formalized as follows:

- $\mathit{atom}(\mathit{grd}(B)) \cap \left(\bigcup_{1 \leq i} \mathit{head}(\mathit{grd}(P[i] \cup Q[i]))\right) = \emptyset$,
- $\left(\bigcup_{1 \leq i} \mathit{atom}(\mathit{grd}(P[i]))\right) \cap \left(\bigcup_{1 \leq j} \mathit{head}(\mathit{grd}(Q[j]))\right) = \emptyset$,
- $\mathit{atom}(\mathit{grd}(P[i])) \cap \left(\bigcup_{i < j} \mathit{head}(\mathit{grd}(P[j]))\right) = \emptyset$ for all $1 \leq i$, and
- $\mathit{atom}(\mathit{grd}(Q[i])) \cap \left(\bigcup_{i < j} \mathit{head}(\mathit{grd}(Q[j]))\right) = \emptyset$ for all $1 \leq i$.

Incremental ASP Solving (made very easy)

See [28] for formal details!

Grounding For a program P over $\mathcal{A}$ and $I \subseteq grd(\mathcal{A})$, an incremental grounder is a partial function

$$\texttt{ground} : (P, I) \mapsto (P', O) \ ,$$

where P' is a program over $grd(\mathcal{A})$ and $O \subseteq grd(\mathcal{A})$.

Solving For programs R, R' over $grd(\mathcal{A})$ and a set L of literals over $grd(\mathcal{A})$, an incremental solver is a pair of total functions

$$\texttt{add} : R \mapsto R' \quad \text{and} \quad \texttt{solve} : L \mapsto \chi,$$

where χ is a subset of the power set of $grd(\mathcal{A})$.

Incremental ASP Solving (made very easy)

See [28] for formal details!

Grounding For a program P over $\mathcal{A}$ and $I \subseteq grd(\mathcal{A})$, an incremental grounder is a partial function

$$\texttt{ground} : (P, I) \mapsto (P', O) \ ,$$

where P' is a program over $grd(\mathcal{A})$ and $O \subseteq grd(\mathcal{A})$.

Solving For programs R, R' over $grd(\mathcal{A})$ and a set L of literals over $grd(\mathcal{A})$, an incremental solver is a pair of total functions

$$\texttt{add} : R \mapsto R' \quad \text{and} \quad \texttt{solve} : L \mapsto \chi,$$

where χ is a subset of the power set of $grd(\mathcal{A})$.

Incremental ASP Solving (made very easy)

See [28] for formal details!

Grounding For a program P over $\mathcal{A}$ and $I \subseteq grd(\mathcal{A})$, an incremental grounder is a partial function

$$\texttt{ground} : (P, I) \mapsto (P', O) \ ,$$

where P' is a program over $grd(\mathcal{A})$ and $O \subseteq grd(\mathcal{A})$.

Solving For programs R, R' over $grd(\mathcal{A})$ and a set L of literals over $grd(\mathcal{A})$, an incremental solver is a pair of total functions

$$\texttt{add} : R \mapsto R' \quad \text{and} \quad \texttt{solve} : L \mapsto \chi,$$

where χ is a subset of the power set of $grd(\mathcal{A})$.

Algorithm 4: `isolve`

Input : A domain description $(B, P[k], Q[k])$.
Output : A nonempty set of answer sets.
Internal : A grounder $\textsc{Grounder}$.
Internal : A solver $\textsc{Solver}$.

$i \leftarrow 0$
$(P_0, O) \leftarrow \textsc{Grounder}.\mathtt{ground}(B, \emptyset)$
$\textsc{Solver}.\mathtt{add}(P_0)$
loop
  $i \leftarrow i + 1$
  $(P_i, O_i) \leftarrow \textsc{Grounder}.\mathtt{ground}(P[i], O)$
  $\textsc{Solver}.\mathtt{add}(P_i)$
  $O \leftarrow O \cup O_i$
  $(Q_i, O'_i) \leftarrow \textsc{Grounder}.\mathtt{ground}(Q[i], O)$
  $\textsc{Solver}.\mathtt{add}(Q_i(\alpha_i) \cup \{\{\alpha_i\} \leftarrow\} \cup \{\leftarrow \alpha_{i-1}\})$
  $\mathcal{X} \leftarrow \textsc{Solver}.\mathtt{solve}(\{\alpha_i\})$
  if $\mathcal{X} \neq \emptyset$ **then return** $\{X \setminus \{\alpha_i\} \mid X \in \mathcal{X}\}$

Example Reloaded

$$
\left.\begin{array}{l}
a \textbf{ causes } p \\
\textbf{exogenous } a \\
\textbf{inertial } p
\end{array}\right\}
\mapsto
\left\{\begin{array}{rcl}
B & = & \left\{\begin{array}{rcl}
p(0) & \leftarrow & \mathit{not}\ \neg p(0) \\
\neg p(0) & \leftarrow & \mathit{not}\ p(0) \\
 & \leftarrow & p(0), \neg p(0)
\end{array}\right\} \\
P[k] & = & \left\{\begin{array}{rcl}
a(k) & \leftarrow & \mathit{not}\ \neg a(k) \\
\neg a(k) & \leftarrow & \mathit{not}\ a(k) \\
p(k) & \leftarrow & a(k) \\
p(k) & \leftarrow & p(k-1), \mathit{not}\ \neg p(k) \\
\neg p(k) & \leftarrow & \neg p(k-1), \mathit{not}\ p(k) \\
 & \leftarrow & p(k), \neg p(k) \\
 & \leftarrow & a(k), \neg a(k)
\end{array}\right\}
\end{array}\right.
$$

$$
\left.\begin{array}{r}
\neg p \textbf{ holds at } 0 \\
p \textbf{ holds at } n \\
\neg a \textbf{ occurs at } n
\end{array}\right\}
\mapsto
\left\{\ Q[k] = \left\{\begin{array}{rl}
\leftarrow & \mathit{not}\ \neg p(0) \\
\leftarrow & \mathit{not}\ p(k) \\
\leftarrow & \mathit{not}\ \neg a(k)
\end{array}\right\}\right.
$$

Example Reloaded

i	Rules				L
0	B	$p(0)$	$\leftarrow$	$not\ \neg p(0)$	
		$\neg p(0)$	$\leftarrow$	$not\ p(0)$	
			$\leftarrow$	$p(0), \neg p(0)$	
1	$P[1]$	$a(1)$	$\leftarrow$	$not\ \neg a(1)$	
		$\neg a(1)$	$\leftarrow$	$not\ a(1)$	
		$p(1)$	$\leftarrow$	$a(1)$	
		$p(1)$	$\leftarrow$	$p(0), not\ \neg p(1)$	
		$\neg p(1)$	$\leftarrow$	$\neg p(0), not\ p(1)$	
			$\leftarrow$	$p(1), \neg p(1)$	
			$\leftarrow$	$a(1), \neg a(1)$	
	$Q[1](\alpha_1)$		$\leftarrow$	$not\ \neg p(0), \alpha_1$	α_1
			$\leftarrow$	$not\ p(1), \alpha_1$	
			$\leftarrow$	$not\ \neg a(1), \alpha_1$	
		$\{\alpha_1\}$	$\leftarrow$		
			$\leftarrow$	α_0	

Example Reloaded

i	Rules				L
0	B	$p(0)$	$\leftarrow$	$not\ \neg p(0)$	
		$\neg p(0)$	$\leftarrow$	$not\ p(0)$	
			$\leftarrow$	$p(0), \neg p(0)$	
1			$\vdots$		
2	$P[2]$	$a(2)$	$\leftarrow$	$not\ \neg a(2)$	
		$\neg a(2)$	$\leftarrow$	$not\ a(2)$	
		$p(2)$	$\leftarrow$	$a(2)$	
		$p(2)$	$\leftarrow$	$p(1), not\ \neg p(2)$	
		$\neg p(2)$	$\leftarrow$	$\neg p(1), not\ p(2)$	
			$\leftarrow$	$p(2), \neg p(2)$	
			$\leftarrow$	$a(2), \neg a(2)$	
	$Q[2](\alpha_2)$		$\leftarrow$	$not\ \neg p(0), \alpha_2$	α_2
			$\leftarrow$	$not\ p(2), \alpha_2$	
			$\leftarrow$	$not\ \neg a(2), \alpha_2$	
		$\{\alpha_2\}$	$\leftarrow$		
			$\leftarrow$	α_1	

Example with iclingo

incremental.lp

```
#base.
p(0)  :- not -p(0).
-p(0) :- not p(0).
:- p(0), -p(0).

#cumulative k.
a(k)  :- not -a(k).
-a(k) :- not a(k).
p(k)  :- a(k).
p(k)  :- p(k-1), not -p(k).
-p(k) :- -p(k-1), not p(k).
:- p(k), -p(k).
:- a(k), -a(k).

#volatile k.
:- not -p(0).
:- not p(k).
:- not -a(k).
```

```
$ iclingo -V[erbose] incremental.lp

iclingo version 2.0.2 (clasp 1.1.1)
Reading from incremental.lp...
=============== step 1 ===============
Grounding...
Preprocessing...
Solving...
=============== step 2 ===============
Grounding...
Preprocessing...
Solving...
Answer: 1
-p(0) a(1) p(1) -a(2) p(2)
=============== Summary ===============

Models      : 1
Total Steps : 2
Time        : 0.000
```

Experiments

We consider iclingo in four settings, keeping over successive solving steps

1. learned constraints,
2. learned constraints and heuristic values,
3. heuristic values only, and
4. neither.

We compare these variants with iterative deepening search using

- clingo, the direct combination of gringo and clasp via an internal interface, as well as
- gringo and clasp via a textual interface (using the output language of lparse).

Experiments

Name	n	*iclingo* (1)	*iclingo* (2)	*iclingo* (3)	*iclingo* (4)	*clingo*	*gringo*|*clasp*
Blocks-world	20	2.61	2.61	2.62	2.62	37.09	42.41
	25	6.78	6.84	6.80	6.80	124.35	138.68
	30	15.68	15.80	15.71	15.81	330.15	362.39
	35	32.43	32.36	32.29	32.31	753.90	821.96
	40	60.99	60.75	60.71	61.04	-	-
	Σ	118.49	118.36	118.13	118.58	2445.49	2565.44
Queens	80	19.46	65.83	39.98	47.79	144.28	153.61
	90	36.72	135.19	70.81	81.70	249.13	264.21
	100	49.25	227.69	111.99	128.62	409.69	431.23
	110	64.05	424.03	176.16	201.67	636.91	669.75
	120	99.54	612.76	274.29	354.00	958.34	1003.67
	Σ	269.02	1465.50	673.23	813.78	2398.35	2522.47

Experiments

Name	n	*iclingo (1)*	*iclingo (2)*	*iclingo (3)*	*iclingo (4)*	*clingo*	*gringo | clasp*
Sokoban	16	243.22	287.46	320.07	334.08	376.74	384.41
	12	26.50	37.55	50.61	28.19	27.83	28.43
	16	124.26	124.44	320.97	341.94	189.48	194.12
	16	135.72	164.70	128.66	183.74	120.60	123.57
	18	140.80	145.07	233.71	275.12	236.60	242.19
	16	26.86	40.60	29.41	27.88	45.94	47.04
	17	1165.67	906.00	734.44	730.09	887.26	904.75
	14	119.95	140.11	106.40	213.22	96.26	98.10
	14	35.42	42.74	58.79	46.81	70.16	71.81
	21	286.46	200.43	600.19	777.68	278.97	285.09
	17	120.33	140.44	139.19	156.85	171.01	174.90
	14	39.09	36.21	36.00	47.48	66.12	67.43
	Σ	2464.28	2265.75	2758.44	3163.08	2566.97	2621.84

Experiments

Name	n	*iclingo (1)*	*iclingo (2)*	*iclingo (3)*	*iclingo (4)*	*clingo*	*gringo* | *clasp*
Sokoban	16	-	-	-	-	-	-
back	12	51.23	44.62	98.09	57.42	72.59	74.30
	16	264.81	201.48	265.21	359.38	296.45	302.46
	16	148.19	121.19	150.06	145.40	148.25	151.43
	18	723.07	-	-	-	1059.02	1081.34
	16	243.81	185.00	340.97	190.32	402.27	410.72
	17	599.74	714.40	1051.60	825.61	-	-
	14	149.37	126.04	164.98	191.33	170.36	173.74
	14	29.73	69.46	73.03	28.04	43.06	43.89
	21	346.56	428.43	400.81	295.69	402.78	411.70
	17	181.00	143.20	172.83	317.82	234.21	239.56
	14	15.06	58.45	39.27	17.50	59.63	60.78
	Σ	3952.57	4492.27	5156.85	4828.51	5288.62	5349.92

Experiments

Name	n	*iclingo (1)*	*iclingo (2)*	*iclingo (3)*	*iclingo (4)*	*clingo*	*gringo*|*clasp*
Towers	33	38.00	42.96	48.46	27.15	31.98	32.76
	34	61.40	36.78	47.09	45.95	61.77	63.39
	36	81.26	60.77	88.52	131.29	86.56	88.46
	39	223.46	155.76	184.63	204.13	216.89	222.74
	41	429.82	327.74	392.47	342.11	459.97	471.22
	Σ	833.94	624.01	761.17	750.63	857.17	878.57
Towers back	33	4.62	6.42	5.68	5.80	12.59	12.79
	34	55.79	33.42	56.27	42.39	52.80	54.00
	36	16.66	16.46	14.69	17.11	24.81	25.38
	39	27.88	25.43	28.60	32.83	46.01	46.85
	41	48.20	36.38	62.75	40.62	83.78	85.60
	Σ	153.15	118.11	167.99	138.75	219.99	224.62
	$\Sigma\Sigma$	7791.45	9084.00	9635.81	9813.33	13776.59	14162.86

Conclusion

- Tackling bounded problems in ASP, paving the way for more ambitious real-world applications.
- Module theory provides us with
  - a natural semantics for non-ground, parameterized program slices and
  - makes precise their composition by appeal to input/output interfaces.
- First experimental results indicate the computational impact of our incremental approach, *but* more needs to be done !

☞ Incremental problems differ from traditional ones !

Conclusion

- Tackling bounded problems in ASP, paving the way for more ambitious real-world applications.
- Module theory provides us with
  - a natural semantics for non-ground, parameterized program slices and
  - makes precise their composition by appeal to input/output interfaces.
- First experimental results indicate the computational impact of our incremental approach, *but* more needs to be done !

☞ Incremental problems differ from traditional ones !

Constraint Answer Set Programming
Overview

Motivation

Observation

Certain applications are more naturally modeled by mixing Boolean with non-Boolean constructs, eg., accounting for

- resources,
- fine timings, or
- functions over finite domains.

Introduction

Groundbreaking Work in ASP [5, 64, 65]

- semantics for multi-sorted, first-order language
- algorithms using DPLL-style backtracking

SAT Modulo Theories [69]

- no modelling language
- algorithms using CDCL-style backjumping and learning

Our ASP approach [40]

- propositional semantics
- algorithms using CDCL-style backjumping and learning
- use off-the-shelf CP solvers

SAT Modulo Theories

SAT Modulo Theories (SMT)

- logical formulas with respect to combinations of background theories
- real numbers, integers, lists, arrays, bit vectors ...

SAT Modulo Theories

SAT Modulo Theories (SMT)

- logical formulas with respect to combinations of background theories
- real numbers, integers, lists, arrays, bit vectors . . .

SAT

$$(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$$

SAT Modulo Theories

SAT Modulo Theories (SMT)

- logical formulas with respect to combinations of background theories
- real numbers, integers, lists, arrays, bit vectors ...

SMT

$$(\sin(x)^3 = \cos(\log(y) \cdot x) \qquad \vee\ b \qquad \vee\ -x^2 \geq 2.3y)$$

SAT Modulo Theories

SAT Modulo Theories (SMT)

- logical formulas with respect to combinations of background theories
- real numbers, integers, lists, arrays, bit vectors ...

SMT

$$(\quad a \quad \lor b \quad \lor -x^2 \geq 2.3y)$$

SAT Modulo Theories

SAT Modulo Theories (SMT)

- logical formulas with respect to combinations of background theories
- real numbers, integers, lists, arrays, bit vectors ...

SMT

$$(\quad a \quad \vee b \quad \vee \quad c \quad)$$

Outline

Constraint Satisfaction Problem

Definition

A *Constraint Satisfaction Problem* (CSP) consists of

- a set V of variables,
- a set D of domains, and
- a set C of constraints

such that

- each *variable* $v \in V$ has an associated *domain* $dom(v) \in D$;
- a *constraint* c is a pair (S, R) consisting of a k-ary *relation* R on a vector $S \subseteq V^k$ of variables, called the *scope* of R.
  - ☞ For $S = (v_1, \ldots, v_k)$, we have $R \subseteq dom(v_1) \times \cdots \times dom(v_k)$.

Constraint Satisfaction Problem

Definition

A *Constraint Satisfaction Problem* (CSP) consists of

- a set V of variables,
- a set D of domains, and
- a set C of constraints

such that

- each *variable* $v \in V$ has an associated *domain* $dom(v) \in D$;
- a *constraint* c is a pair (S, R) consisting of a k-ary *relation* R on a vector $S \subseteq V^k$ of variables, called the *scope* of R.
  - ☞ For $S = (v_1, \ldots, v_k)$, we have $R \subseteq dom(v_1) \times \cdots \times dom(v_k)$.

Example

$$
\begin{array}{ccccc}
 & s & e & n & d \\
+ & m & o & r & e \\
\hline
m & o & n & e & y
\end{array}
$$

Each letter corresponds exactly to one digit and all variables have to be pairwisely distinct.

$V = \{s, e, n, d, m, o, r, y\}$ $\qquad$ $dom(v) = 0..9$ for all $v \in V$

$$
\begin{aligned}
C = \{ & (V, \quad allDistinct(V)), \\
& (V, \quad s \times 1000 + e \times 100 + n \times 10 + d + \\
& \qquad m \times 1000 + o \times 100 + r \times 10 + e == \\
& \qquad m \times 10000 + o \times 1000 + n \times 100 + e \times 10 + y), \\
& ((m), m == 1)\}
\end{aligned}
$$

Example

$$\begin{array}{ccccc} & s & e & n & d \\ + & m & o & r & e \\ \hline m & o & n & e & y \end{array}$$

Each letter corresponds exactly to one digit and all variables have to be pairwisely distinct.
The example has exactly one solution.

$$\{ s \mapsto 9, e \mapsto 5, n \mapsto 6, d \mapsto 7, m \mapsto 1, o \mapsto 0, r \mapsto 8, y \mapsto 2 \}$$

$$\begin{array}{ccccc} & 9 & 5 & 6 & 7 \\ + & 1 & 0 & 8 & 5 \\ \hline 1 & 0 & 6 & 5 & 2 \end{array}$$

Constraint Satisfaction Problem

Notation

We use $S(c) = S$ and $R(c) = R$ to access the scope and the relation of a constraint $c = (S, R)$.

Definition

For an assignment $A : V \to \bigcup_{v \in V} dom(v)$ and a constraint (S, R) with scope $S = (v_1, \ldots, v_k)$, define

$$sat_C(A) = \{c \in C \mid A(S(c)) \in R(c)\}$$

where $A(S) = (A(v_1), \ldots, A(v_k))$.

Constraint Satisfaction Problem

Notation

We use $S(c) = S$ and $R(c) = R$ to access the scope and the relation of a constraint $c = (S, R)$.

Definition

For an assignment $A : V \rightarrow \bigcup_{v \in V} dom(v)$ and a constraint (S, R) with scope $S = (v_1, \ldots, v_k)$, define

$$sat_C(A) = \{c \in C \mid A(S(c)) \in R(c)\}$$

where $A(S) = (A(v_1), \ldots, A(v_k))$.

Constraint Answer Set Programming

Definition

A *Constraint Logic Program* P is a logic program over an extended alphabet $\mathcal{A} \cup \mathcal{C}$ where

- $\mathcal{A}$ is a set of *regular atoms* and
- $\mathcal{C}$ is a set of *constraint atoms*,

such that $head(r) \in \mathcal{A}$ for each $r \in P$.

Auxiliary Definition

Given a set of literals B and some set $\mathcal{B}$ of atoms, we define
$B|_{\mathcal{B}} = (B^+ \cap \mathcal{B}) \cup \{not\ a \mid a \in B^- \cap \mathcal{B}\}$.

Constraint Answer Set Programming

Definition

A *Constraint Logic Program* P is a logic program over an extended alphabet $\mathcal{A} \cup \mathcal{C}$ where

- $\mathcal{A}$ is a set of *regular atoms* and
- $\mathcal{C}$ is a set of *constraint atoms*,

such that $head(r) \in \mathcal{A}$ for each $r \in P$.

Auxiliary Definition

Given a set of literals B and some set $\mathcal{B}$ of atoms, we define
$B|_{\mathcal{B}} = (B^+ \cap \mathcal{B}) \cup \{not\ a \mid a \in B^- \cap \mathcal{B}\}$.

Constraint Answer Set Programming

Definition

We identify constraint atoms with constraints via a function

$$\gamma : \mathcal{C} \rightarrow C$$

furthermore, $\gamma(Y) = \{\gamma(c) \mid c \in Y\}$ for any $Y \subseteq \mathcal{C}$.

Note

- Unlike regular atoms $\mathcal{A}$, constraint atoms $\mathcal{C}$ are not subject to the unique names assumption, eg.

  $$\gamma(x < y) \;=\; \gamma(((-y - 1) \leq -(x + 1)) \wedge (x \neq y))$$

- A constraint logic program P is associated with a CSP as follows
  - $C[P] = \gamma(atom(P) \cap \mathcal{C})$,
  - $V[P]$ is obtained from the constraint scopes in $C[P]$,
  - $D[P]$ is provided by a declaration.

Constraint Answer Set Programming

Definition

We identify constraint atoms with constraints via a function

$$\gamma : \mathcal{C} \rightarrow C$$

furthermore, $\gamma(Y) = \{\gamma(c) \mid c \in Y\}$ for any $Y \subseteq \mathcal{C}$.

Note

- Unlike regular atoms $\mathcal{A}$, constraint atoms $\mathcal{C}$ are not subject to the unique names assumption, eg.

  $$\gamma(x < y) \;=\; \gamma(((-y - 1) \leq -(x + 1)) \wedge (x \neq y))$$

- A constraint logic program P is associated with a CSP as follows
  - $C[P] = \gamma(atom(P) \cap \mathcal{C})$,
  - $V[P]$ is obtained from the constraint scopes in $C[P]$,
  - $D[P]$ is provided by a declaration.

Constraint Answer Set Programming

Definition

We identify constraint atoms with constraints via a function

$$\gamma : \mathcal{C} \to C$$

furthermore, $\gamma(Y) = \{\gamma(c) \mid c \in Y\}$ for any $Y \subseteq \mathcal{C}$.

Note

- Unlike regular atoms $\mathcal{A}$, constraint atoms $\mathcal{C}$ are not subject to the unique names assumption, eg.

  $$\gamma(x < y) \ = \ \gamma(((-y - 1) \leq -(x + 1)) \wedge (x \neq y))$$

- A constraint logic program P is associated with a CSP as follows
  - $C[P] = \gamma(atom(P) \cap \mathcal{C})$,
  - $V[P]$ is obtained from the constraint scopes in $C[P]$,
  - $D[P]$ is provided by a declaration.

Constraint Answer Set Programming

Definition

Let P be a constraint logic program over $\mathcal{A} \cup \mathcal{C}$ and let $A : V[P] \rightarrow D[P]$ be an assignment.
We define the *constraint reduct* of as P wrt A as follows.

$$\begin{aligned} P^A \quad = \quad \{ \; & head(r) \leftarrow body(r)|_{\mathcal{A}} \mid r \in P, \\ & \gamma(body(r)|_{\mathcal{C}}{}^{+}) \subseteq sat_{C[P]}(A), \\ & \gamma(body(r)|_{\mathcal{C}}{}^{-}) \cap sat_{C[P]}(A) = \emptyset \; \} \end{aligned}$$

Definition

A set $X \subseteq \mathcal{A}$ of (regular) atoms is a *constraint answer set* of P wrt A, if X is an answer set of P^A.

☞ That is, if X is the $\subseteq$-smallest model of $(P^A)^X$.

Constraint Answer Set Programming

Definition

Let P be a constraint logic program over $\mathcal{A} \cup \mathcal{C}$ and
let $A : V[P] \to D[P]$ be an assignment.
We define the *constraint reduct* of as P wrt A as follows.

$$
\begin{aligned}
P^A \quad = \quad \{\, & head(r) \leftarrow body(r)|_{\mathcal{A}} \mid r \in P, \\
& \gamma(body(r)|_{\mathcal{C}}{}^{+}) \subseteq sat_{C[P]}(A), \\
& \gamma(body(r)|_{\mathcal{C}}{}^{-}) \cap sat_{C[P]}(A) = \emptyset \,\}
\end{aligned}
$$

Definition

A set $X \subseteq \mathcal{A}$ of (regular) atoms is a *constraint answer set* of P wrt A, if X is an answer set of P^A.

☞ That is, if X is the $\subseteq$-smallest model of $(P^A)^X$.

Constraint Answer Set Programming

Definition

Let P be a constraint logic program over $\mathcal{A} \cup \mathcal{C}$ and let $A : V[P] \rightarrow D[P]$ be an assignment.
We define the *constraint reduct* of as P wrt A as follows.

$$\begin{array}{rcl} P^A & = & \{\ head(r) \leftarrow body(r)|_{\mathcal{A}} \mid r \in P, \\ & & \qquad \gamma(body(r)|_{\mathcal{C}}{}^{+}) \subseteq sat_{C[P]}(A), \\ & & \qquad \gamma(body(r)|_{\mathcal{C}}{}^{-}) \cap sat_{C[P]}(A) = \emptyset\ \} \end{array}$$

Definition

A set $X \subseteq \mathcal{A}$ of (regular) atoms is a *constraint answer set* of P wrt A, if X is an answer set of P^A.

- ☞ That is, if X is the $\subseteq$-smallest model of $(P^A)^X$.

Outline

Modeling Language

Note

Although our semantics is propositional, the atoms in $\mathcal{A}$ and $\mathcal{C}$ are constructible from a multi-sorted, first-order signature given by:

- a set $\mathcal{P}_\mathcal{A} \cup \mathcal{P}_\mathcal{C}$ of *predicate symbols* such that $\mathcal{P}_\mathcal{A} \cap \mathcal{P}_\mathcal{C} = \emptyset$,
- a set $\mathcal{F}_\mathcal{A} \cup \mathcal{F}_\mathcal{C}$ of *function symbols* (including constant symbols),
- a set $\mathcal{V}_\mathcal{A}$ of *regular variable symbols*, and
- a set $\mathcal{V}_\mathcal{C} \subseteq \mathcal{T}(\mathcal{F}_\mathcal{A})$ of *constraint variable symbols*, where $\mathcal{T}(\mathcal{F}_\mathcal{A})$ denotes the set of all ground terms over $\mathcal{F}_\mathcal{A}$.

As common in ASP, the atoms in $\mathcal{A} \cup \mathcal{C}$ are obtained by a grounding process.

An Example

$$
\begin{array}{rcl}
time(0..t_{max}) & & \\
bucket(a) \quad bucket(b) & & \\
1\ \{pour(B,T) : bucket(B)\}\ 1 & \leftarrow & time(T), T < t_{max} \\
1 \leq^{\$} amt(B,T) & \leftarrow & pour(B,T), T < t_{max} \\
amt(B,T) \leq^{\$} 3 & \leftarrow & pour(B,T), T < t_{max} \\
amt(B,T) =^{\$} 0 & \leftarrow & not\ pour(B,T), T < t_{max} \\
vol(B,T{+}1) =^{\$} vol(B,T) + amt(B,T) & \leftarrow & time(T) < t_{max} \\
down(B,T) & \leftarrow & vol(C,T) <^{\$} vol(B,T) \\
up(B,T) & \leftarrow & not\ down(B,T) \\
vol(a,0) =^{\$} 0 \quad vol(b,0) =^{\$} 1 & & \\
 & \leftarrow & up(a, t_{max})
\end{array}
$$

An Example

- Consider the signature of our exemplary program:

$$
\begin{aligned}
\{B, C, T\} &\subseteq \mathcal{V}_{\mathcal{A}} \\
\{0, \dots, t_{max}, +, a, b, amt, vol\} &\subseteq \mathcal{F}_{\mathcal{A}} \\
\{<, time, bucket, pour, up, down\} &\subseteq \mathcal{P}_{\mathcal{A}}
\end{aligned}
$$

$$
\begin{aligned}
\{0, 1, 3, +\} &\subseteq \mathcal{F}_{\mathcal{C}} \\
\{=^{\$}, <^{\$}, \leq^{\$}\} &\subseteq \mathcal{P}_{\mathcal{C}}
\end{aligned}
$$

- With substitution $\{B \mapsto b, T \mapsto 1, t_{max} \mapsto 2\}$, we get:

$$
\begin{aligned}
amt(b,1) =^{\$} 0 &\leftarrow not\ pour(b,1) \\
vol(b,2) =^{\$} vol(b,1) + amt(b,1) &\leftarrow
\end{aligned}
$$

- and, among others, our signature hence contains

$$
\begin{aligned}
\{amt(b,1), vol(b,1), vol(b,2)\} &\subseteq \mathcal{V}_{\mathcal{C}} \\
\{pour(b,1)\} &\subseteq \mathcal{A} \\
\{amt(b,1) =^{\$} 0,\ vol(b,2) =^{\$} vol(b,1) + amt(b,1)\} &\subseteq \mathcal{C}
\end{aligned}
$$

An Example

- Consider the signature of our exemplary program:

$$\begin{array}{rcl}
\{B, C, T\} & \subseteq & \mathcal{V}_{\mathcal{A}} \\
\{0, \dots, t_{max}, +, a, b, amt, vol\} & \subseteq & \mathcal{F}_{\mathcal{A}} \\
\{<, time, bucket, pour, up, down\} & \subseteq & \mathcal{P}_{\mathcal{A}} \\
\{0, 1, 3, +\} & \subseteq & \mathcal{F}_{\mathcal{C}} \\
\{=^{\$}, <^{\$}, \leq^{\$}\} & \subseteq & \mathcal{P}_{\mathcal{C}}
\end{array}$$

- With substitution $\{B \mapsto b, T \mapsto 1, t_{max} \mapsto 2\}$, we get:

$$\begin{array}{rcl}
amt(b, 1) =^{\$} 0 & \leftarrow & not\ pour(b, 1) \\
vol(b, 2) =^{\$} vol(b, 1) + amt(b, 1) & \leftarrow &
\end{array}$$

- and, among others, our signature hence contains

$$\begin{array}{rcl}
\{amt(b, 1), vol(b, 1), vol(b, 2)\} & \subseteq & \mathcal{V}_{\mathcal{C}} \\
\{pour(b, 1)\} & \subseteq & \mathcal{A} \\
\{amt(b, 1) =^{\$} 0,\ vol(b, 2) =^{\$} vol(b, 1) + amt(b, 1)\} & \subseteq & \mathcal{C}
\end{array}$$

An Example

- Consider the signature of our exemplary program:

$$\begin{array}{rcl} \{B, C, T\} & \subseteq & \mathcal{V}_{\mathcal{A}} \\ \{0, \dots, t_{max}, +, a, b, amt, vol\} & \subseteq & \mathcal{F}_{\mathcal{A}} \\ \{<, time, bucket, pour, up, down\} & \subseteq & \mathcal{P}_{\mathcal{A}} \\ \{0, 1, 3, +\} & \subseteq & \mathcal{F}_{\mathcal{C}} \\ \{=^{\$}, <^{\$}, \leq^{\$}\} & \subseteq & \mathcal{P}_{\mathcal{C}} \end{array}$$

- With substitution $\{B \mapsto b, T \mapsto 1, t_{max} \mapsto 2\}$, we get:

$$\begin{array}{rcl} amt(b,1) =^{\$} 0 & \leftarrow & not\ pour(b,1) \\ vol(b,2) =^{\$} vol(b,1) + amt(b,1) & \leftarrow & \end{array}$$

- and, among others, our signature hence contains

$$\begin{array}{rcl} \{amt(b,1), vol(b,1), vol(b,2)\} & \subseteq & \mathcal{V}_{\mathcal{C}} \\ \{pour(b,1)\} & \subseteq & \mathcal{A} \\ \{amt(b,1) =^{\$} 0,\ vol(b,2) =^{\$} vol(b,1) + amt(b,1)\} & \subseteq & \mathcal{C} \end{array}$$

An Example

- Consider the signature of our exemplary program:

$$
\begin{array}{rcl}
\{B, C, T\} & \subseteq & \mathcal{V}_{\mathcal{A}} \\
\{0, \dots, t_{max}, +, a, b, amt, vol\} & \subseteq & \mathcal{F}_{\mathcal{A}} \\
\{<, time, bucket, pour, up, down\} & \subseteq & \mathcal{P}_{\mathcal{A}} \\
\{0, 1, 3, +\} & \subseteq & \mathcal{F}_{\mathcal{C}} \\
\{=^{\$}, <^{\$}, \leq^{\$}\} & \subseteq & \mathcal{P}_{\mathcal{C}}
\end{array}
$$

- With substitution $\{B \mapsto b, T \mapsto 1, t_{max} \mapsto 2\}$, we get:

$$
\begin{array}{rcl}
amt(b,1) =^{\$} 0 & \leftarrow & not\ pour(b,1) \\
vol(b,2) =^{\$} vol(b,1) + amt(b,1) & \leftarrow &
\end{array}
$$

- and, among others, our signature hence contains

$$
\begin{array}{rcl}
\{amt(b,1), vol(b,1), vol(b,2)\} & \subseteq & \mathcal{V}_{\mathcal{C}} \\
\{pour(b,1)\} & \subseteq & \mathcal{A} \\
\{amt(b,1) =^{\$} 0,\ vol(b,2) =^{\$} vol(b,1) + amt(b,1)\} & \subseteq & \mathcal{C}
\end{array}
$$

An Example

- Consider the signature of our exemplary program:

$$
\begin{array}{rcl}
\{B, C, T\} & \subseteq & \mathcal{V}_{\mathcal{A}} \\
\{0, \dots, t_{max}, +, a, b, amt, vol\} & \subseteq & \mathcal{F}_{\mathcal{A}} \\
\{<, time, bucket, pour, up, down\} & \subseteq & \mathcal{P}_{\mathcal{A}} \\
\{0, 1, 3, +\} & \subseteq & \mathcal{F}_{\mathcal{C}} \\
\{=^{\$}, <^{\$}, \leq^{\$}\} & \subseteq & \mathcal{P}_{\mathcal{C}}
\end{array}
$$

- With substitution $\{B \mapsto b, T \mapsto 1, t_{max} \mapsto 2\}$, we get:

$$
\begin{array}{rcl}
amt(b,1) =^{\$} 0 & \leftarrow & not\ pour(b,1) \\
vol(b,2) =^{\$} vol(b,1) + amt(b,1) & \leftarrow &
\end{array}
$$

- and, among others, our signature hence contains

$$
\begin{array}{rcl}
\{amt(b,1), vol(b,1), vol(b,2)\} & \subseteq & \mathcal{V}_{\mathcal{C}} \\
\{pour(b,1)\} & \subseteq & \mathcal{A} \\
\{amt(b,1) =^{\$} 0,\ \ vol(b,2) =^{\$} vol(b,1) + amt(b,1)\} & \subseteq & \mathcal{C}
\end{array}
$$

An Example

For $t_{max} = 2$, our program has eleven constraint answer sets, summarized as follows

$up(a,0)$	$pour(a,0)$	$amt(a,0)$	$up(a,1)$	$pour(a,1)$	$amt(a,1)$	$up(a,2)$
T	**T**	1	**T**	**T**	$1,2,3$	**F**
T	**T**	$2,3$	**F**	**T**	$1,2,3$	**F**
T	**T**	3	**F**	**F**	0	**F**
T	**F**	0	**T**	**T**	3	**F**

An Example

For $t_{max} = 2$, our program has eleven constraint answer sets, summarized as follows

$up(a,0)$	$pour(a,0)$	$amt(a,0)$	$up(a,1)$	$pour(a,1)$	$amt(a,1)$	$up(a,2)$
T	**T**	1	**T**	**T**	$1,2,3$	**F**
T	**T**	$2,3$	**F**	**T**	$1,2,3$	**F**
T	**T**	3	**F**	**F**	0	**F**
T	**F**	0	**T**	**T**	3	**F**

Outline

CDNL

Main Algorithm

loop

PROPAGATE

if no conflict **then**

if partial assignment **then** DECIDE
else return solution

else if some decisions made **then**

ANALYZE CONFLICT
RECORD REASON
BACKJUMP

else exit

CDNL

Main Algorithm

loop

PROPAGATE

if no conflict **then**

if partial assignment **then** DECIDE
else return solution

else if some decisions made **then**

ANALYZE CONFLICT
RECORD REASON
BACKJUMP

else exit

ASP Propagation

Propagation Algorithm

loop

  UNIT-PROPAGATION

  if conflict **then return**

  else if UNFOUNDED-SET **then**

    RECORD LOOP-NOGOOD

    if conflict **then return**

  else return

Constraint ASP Propagation

Propagation Algorithm

loop
- UNIT-PROPAGATION
- **if** conflict **then return**
- **else if** UNFOUNDED-SET **then**
  - RECORD LOOP-NOGOOD
  - **if** conflict **then return**
- **else if** CONSTRAINT-PROPAGATION **then**
  - **if** conflict **then return**
- **else return**

CDNL

Main Algorithm

loop

    PROPAGATE
    if no conflict **then**
        if partial assignment **then** DECIDE
        else return solution
    else if some decisions made **then**
        ANALYZE CONFLICT
        RECORD REASON
        BACKJUMP
    else exit

Constraint CDNL

Main Algorithm

loop

PROPAGATE

if no conflict **then**

if partial assignment **then** DECIDE
else if CSP-SOLVE **then return** solution
else

ANALYZE CONFLICT
RECORD REASON
BACKJUMP

else if some decisions made **then**

ANALYZE CONFLICT
RECORD REASON
BACKJUMP

else exit

Outline

Example on Grounding

John goes to work either by car (30-40 minutes), or by bus (at least 60 minutes).
Fred goes to work either by car (20-30 minutes), or in a car pool (40-50 minutes).
Today John left home between 7:10 and 7:20, and Fred arrived between 8:00 and 8:10.
We also know that John arrived at work about 10-20 minutes after Fred left home.

We wish to answer queries such as:

- Is the information in the story consistent?
- Is it possible that John took the bus, and Fred used the carpool?
- What are the possible times at which Fred left home?

Example on Grounding

- Number of Lines Output

maxtime	gringo	clingcon
10	626	19
100	47132	19
500	1137332	19
1000	4525082	19

NASA Advisor: 40 seconds realtime

	clingo	*adsolver*				*clingcon*				
Benchmark	5	5	7	11	13	5	7	11	13	20
3-0/025	162.84	14.74	51.42	460.57	365.37	1.19	1.97	4.21	5.99	17.84
3-0/050	173.28	31.39	108.21	471.41	—	1.26	2.32	6.80	11.85	27.36
3-0/100	175.94	448.90	188.33	—	—	1.32	2.35	10.11	12.04	38.78
3-0/125	165.64	19.78	60.07	224.60	—	1.18	1.94	4.05	10.00	133.99
5-0/025	174.12	28.78	107.41	—	—	1.28	2.90	5.87	14.27	66.55
5-0/050	163.25	13.57	42.00	204.34	497.64	1.18	1.97	4.71	10.04	241.59
5-0/100	168.16	21.50	66.10	282.36	514.08	1.20	1.98	4.13	6.45	25.32
5-0/125	174.38	32.02	104.32	429.72	—	1.34	2.95	6.39	9.70	81.17
8-0/025	177.82	41.57	140.93	—	—	1.30	2.73	11.00	12.69	222.49
8-0/050	167.72	18.83	54.76	215.43	—	1.18	1.93	4.02	7.76	457.86
8-0/100	165.55	13.72	41.03	208.74	—	1.21	2.00	5.05	6.10	26.17
8-0/125	162.29	16.81	53.40	246.64	519.59	1.20	1.99	4.15	6.69	17.82
∅	169.25	58.47	84.83	378.65	558.06	1.24	2.25	5.87	9.47	113.08

clingo standard ASP grounder and ASP solver
adsolver ASPmCSP solver based on smodels [Mellarkod and Gelfond, '08]
clingcon ASPmCSP solver based on clingo and gecode

http://potassco.sourceforge.net

Potassco, the Potsdam Answer Set Solving Collection,
bundles tools for ASP developed at the University of Potsdam,
for instance:

- *Grounder*: Gringo, pyngo
- *Solver*: clasp, claspD, claspar
- *Grounder+Solver*: Clingo, iClingo, oClingo, Clingcon
- *Further Tools*: claspre, claspfolio, coala, inca, plasp, sbass, xorro

Benchmarking: http://asparagus.cs.uni-potsdam.de

http://potassco.sourceforge.net

Potassco, the Potsdam Answer Set Solving Collection,
bundles tools for ASP developed at the University of Potsdam,
for instance:

- *Grounder*: Gringo, pyngo
- *Solver*: clasp, claspD, claspar
- *Grounder+Solver*: Clingo, iClingo, oClingo, Clingcon
- *Further Tools*: claspre, claspfolio, coala, inca, plasp, sbass, xorro
- *Benchmarking*: `http://asparagus.cs.uni-potsdam.de`

http://potassco.sourceforge.net

Potassco, the Potsdam Answer Set Solving Collection,
bundles tools for ASP developed at the University of Potsdam,
for instance:

- *Grounder*: Gringo, pyngo
- *Solver*: clasp, claspD, claspar
- *Grounder+Solver*: Clingo, iClingo, oClingo, Clingcon
- *Further Tools*: claspre, claspfolio, coala, inca, plasp, sbass, xorro
- *Benchmarking*: `http://asparagus.cs.uni-potsdam.de`

Summary

- ASP is emerging as a viable tool for Knowledge Representation and Reasoning
- ASP offers efficient and versatile off-the-shelf solving technology
  - `http://potassco.sourceforge.net`
  - ASP'09, PB'09, and SAT'09
- ASP offers an expanding functionality and ease of use
  - Rapid application development tool
- ASP has a growing range of applications

Summary

- ASP is emerging as a viable tool for Knowledge Representation and Reasoning
- ASP offers efficient and versatile off-the-shelf solving technology
  - `http://potassco.sourceforge.net`
  - ASP'09, PB'09, and SAT'09
- ASP offers an expanding functionality and ease of use
  - Rapid application development tool
- ASP has a growing range of applications

$$\text{ASP} = \text{KR} + \text{DB} + \text{Search}$$

Acknowledgments

- Fahiem Bacchus
- Thomas Eiter
- Wolfgang Faber
- Jinbo Huang
- Roland Kaminski
- Benjamin Kaufmann
- Tomi Janhunen
- Joohyung Lee
- Vladimir Lifschitz
- Ilkka Niemelä
- Stefan Woltran

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.

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.

S. Baselice, P. Bonatti, and M. Gelfond.
Towards an integration of answer set and constraint solving.
In M. Gabbrielli and G. Gupta, editors, *Proceedings of the Twenty-first International Conference on Logic Programming (ICLP'05)*, volume 3668 of *Lecture Notes in Computer Science*, pages 52–66. Springer-Verlag, 2005.

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. [4], 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.

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 Garcia de la Banda and Pontelli [25], pages 724–728.

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 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. 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*. Springer-Verlag, 2008.

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, M. Ostrowski, T. Schaub, and S. Thiele.
Engineering an incremental ASP solver.
In Garcia de la Banda and Pontelli [25], pages 190–205.

M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub.
On the implementation of weight constraint rules in conflict-driven ASP solvers.
In Hill and Warren [49], pages 250–264.

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.
In D. Le Berre and A. Van Gelder, editors, *Proceedings of the Second Workshop on Pragmatics of SAT (PoS'11)*, 2011.
To appear.

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. [4], pages 260–265.

M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Conflict-driven answer set enumeration.
In Baral et al. [4], pages 136–148.

M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Conflict-driven answer set solving.
In Veloso [77], 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, M. Ostrowski, and T. Schaub.
Constraint answer set solving.
In Hill and Warren [49], pages 235–249.

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. Gelfond.
Answer sets.
In V. Lifschitz, F. van Hermelen, 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.

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*. Springer-Verlag, 2009.

J. Huang.
The effect of restarts on the efficiency of clause learning.
In Veloso [77], 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 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.
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. [8], 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.

V. Mellarkod and M. Gelfond.
Integrating answer set reasoning with constraint solving techniques.
In J. Garrigue and M. Hermenegildo, editors, *Proceedings of the Ninth International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *Lecture Notes in Computer Science*, pages 15–31. Springer-Verlag, 2008.

V. Mellarkod, M. Gelfond, and Y. Zhang.
Integrating answer set programming and constraint logic programming.

Annals of Mathematics and Artificial Intelligence, 53(1-4):251–287, 2008.

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.

R. Nieuwenhuis, A. Oliveras, and C. Tinelli.

Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T).
Journal of the ACM, 53(6):937–977, 2006.

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.

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.

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.