Bias

Focus

- Answer Set Programming as Boolean Constraint Satisfaction Problem
- Answer Set Solving as a Boolean Constraint Solving
- Answer Set Systems at http://potassco.sourceforge.net

Further resources

http://potassco.sourceforge.net/teaching.html
Bias

Focus

- Answer Set Programming as Boolean Constraint Satisfaction Problem
- Answer Set Solving as a Boolean Constraint Solving
- Answer Set Systems at http://potassco.sourceforge.net

Further resources

http://potassco.sourceforge.net/teaching.html
Motivation: Overview

1. Motivation
2. Nutshell
3. Shifting paradigms
4. Rooting ASP
5. Problem solving
6. Use
Motivation: Overview

1. Motivation
2. Nutshell
3. Shifting paradigms
4. Rooting ASP
5. Problem solving
6. Use
“What is the problem?” versus “How to solve the problem?”
Motivation

Informatics

“What is the problem?” versus “How to solve the problem?”

Problem

Solution

Computer

Output
Traditional programming

“What is the problem?” versus “How to solve the problem?”
Traditional programming

“What is the problem?” versus “How to solve the problem?”
Declarative problem solving

“What is the problem?” versus “How to solve the problem?”
Declarative problem solving

“What is the problem?” versus “How to solve the problem?”
Declarative problem solving

“What is the problem?” versus “How to solve the problem?”
Motivation: Overview

1. Motivation
2. Nutshell
3. Shifting paradigms
4. Rooting ASP
5. Problem solving
6. Use
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.

ASP is versatile as reflected by the ASP solver clasp, winning first places at ASP’07/09/11, CASC’11, MISC’11, PB’09/11, and SAT’09/11.

ASP embraces many emerging application areas.
Answer Set Programming

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
- ASP is versatile as reflected by the ASP solver clasp, winning first places at ASP’07/09/11, CASC’11, MISC’11, PB’09/11, and SAT’09/11
- ASP embraces many emerging application areas
Answer Set Programming

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

ASP is versatile as reflected by the ASP solver *clasp*, winning first places at ASP’07/09/11, CASC’11, MISC’11, PB’09/11, and SAT’09/11

ASP embraces many emerging application areas
Answer Set Programming

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
- ASP is versatile as reflected by the ASP solver \( clasp \), winning first places at \( ASP'07/09/11, CASC'11, MISC'11, PB'09/11, \) and \( SAT'09/11 \)
- ASP embraces many emerging application areas
Answer Set Programming

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
- ASP is versatile as reflected by the ASP solver *clasp*, winning first places at ASP’07/09/11, CASC’11, MISC’11, PB’09/11, and SAT’09/11
- ASP embraces many emerging application areas
Answer Set Programming

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
- ASP is versatile as reflected by the ASP solver *clasp*, winning first places at ASP’07/09/11, CASC’11, MISC’11, PB’09/11, and SAT’09/11
- ASP embraces many emerging application areas
Answer Set Programming

*in a Peanutshell*

- ASP is an approach to *declarative problem solving*, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
  
  tailored to *Knowledge Representation and Reasoning*
Answer Set Programming

in a Peanutshell

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

tailed to Knowledge Representation and Reasoning

\[
\text{ASP} = \text{KR} + \text{DB} + \text{SAT} + \text{LP}
\]
Motivation: Overview

1. Motivation
2. Nutshell
3. Shifting paradigms
4. Rooting ASP
5. Problem solving
6. Use
KR’s shift of paradigm

**Theorem Proving based approach** (eg. Prolog)

1. Provide a representation of the problem.
2. A solution is given by a derivation of a query.

**Model Generation based approach** (eg. SATisfiability testing)

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

Automated planning, Kautz and Selman (ECAI’92)

Represent planning problems as propositional theories so that models not proofs describe solutions
KR’s shift of paradigm

Theorem Proving based approach \(\text{eg. Prolog}\)

1. Provide a representation of the problem.
2. A solution is given by a derivation of a query.

Model Generation based approach \(\text{eg. SATisfiability testing}\)

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

Automated planning, Kautz and Selman (ECAI’92)

Represent planning problems as propositional theories so that models not proofs describe solutions
KR’s shift of paradigm

Theorem Proving based approach (eg. Prolog)
1. Provide a representation of the problem.
2. A solution is given by a derivation of a query.

Model Generation based approach (eg. SATisfiability testing)
1. Provide a representation of the problem.
2. A solution is given by a model of the representation.

Automated planning, Kautz and Selman (ECAI’92)

Represent planning problems as propositional theories so that models not proofs describe solutions.
Shifting paradigms

KR’s shift of paradigm

Theorem Proving based approach (eg. Prolog)

1. Provide a representation of the problem.
2. A solution is given by a derivation of a query.

Model Generation based approach (eg. SATisfiability testing)

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

Automated planning, Kautz and Selman (ECAI’92)

Represent planning problems as propositional theories so that models not proofs describe solutions.
Model Generation based Problem Solving

<table>
<thead>
<tr>
<th>Representation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint satisfaction problem</td>
<td>assignment</td>
</tr>
<tr>
<td>propositional horn theories</td>
<td>smallest model</td>
</tr>
<tr>
<td>propositional theories</td>
<td>models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>stable models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>supported models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>Herbrand models</td>
</tr>
<tr>
<td>auto-epistemic theories</td>
<td>expansions</td>
</tr>
<tr>
<td>default theories</td>
<td>extensions</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

auto-epistemic theories

default theories

extensions
# Model Generation based Problem Solving

<table>
<thead>
<tr>
<th>Representation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint satisfaction problem</td>
<td>assignment</td>
</tr>
<tr>
<td>propositional horn theories</td>
<td>smallest model</td>
</tr>
<tr>
<td>propositional theories</td>
<td>models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>stable models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>supported models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>Herbrand models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>expansions</td>
</tr>
<tr>
<td>first-order theories</td>
<td>extensions</td>
</tr>
<tr>
<td>auto-epistemic theories</td>
<td>:</td>
</tr>
<tr>
<td>default theories</td>
<td>:</td>
</tr>
<tr>
<td>...</td>
<td>:</td>
</tr>
<tr>
<td>...</td>
<td>:</td>
</tr>
</tbody>
</table>
## Model Generation based Problem Solving

<table>
<thead>
<tr>
<th>Representation</th>
<th>Solution</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint satisfaction problem</td>
<td>assignment</td>
<td></td>
</tr>
<tr>
<td>propositional horn theories</td>
<td>smallest model</td>
<td></td>
</tr>
<tr>
<td>propositional theories</td>
<td>models</td>
<td></td>
</tr>
<tr>
<td>propositional theories</td>
<td>minimal models</td>
<td></td>
</tr>
<tr>
<td>propositional theories</td>
<td>stable models</td>
<td></td>
</tr>
<tr>
<td>propositional programs</td>
<td>supported models</td>
<td></td>
</tr>
<tr>
<td>propositional programs</td>
<td>stable models</td>
<td></td>
</tr>
<tr>
<td>propositional programs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>first-order theories</td>
<td>minimal models</td>
<td></td>
</tr>
<tr>
<td>first-order theories</td>
<td>stable models</td>
<td></td>
</tr>
<tr>
<td>first-order theories</td>
<td>supported models</td>
<td></td>
</tr>
<tr>
<td>first-order theories</td>
<td>stable models</td>
<td></td>
</tr>
<tr>
<td>first-order theories</td>
<td></td>
<td></td>
</tr>
<tr>
<td>auto-epistemic theories</td>
<td>Herbrand models</td>
<td></td>
</tr>
<tr>
<td>default theories</td>
<td>extensions</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

---

Torsten Schaub et al. (KRR@UP)
LP-style playing with blocks

Prolog program

\[
\text{on}(a,b).
\text{on}(b,c).
\text{above}(X,Y) :- \text{on}(X,Y).
\text{above}(X,Y) :- \text{on}(X,Z), \text{above}(Z,Y).
\]

Prolog queries

?– above(a,c).
true.

?– above(c,a).
no.
LP-style playing with blocks

Prolog program

\[
\text{on}(a,b).
\]
\[
\text{on}(b,c).
\]
\[
\text{above}(X,Y) :- \text{on}(X,Y).
\]
\[
\text{above}(X,Y) :- \text{on}(X,Z), \text{above}(Z,Y).
\]

Prolog queries

?- \text{above}(a,c).
\text{true}.

?- \text{above}(c,a).
\text{no}.
LP-style playing with blocks

Another Prolog program

\[
on(a, b).
on(b, c).
\]

\[
\text{above}(X,Y) :- \text{above}(X,Z), \text{on}(Z,Y).
\text{above}(X,Y) :- \text{on}(X,Y).
\]

Prolog queries

\[?- \text{above}(a,c).\]

Fatal Error: local stack overflow.
LP-style playing with blocks

Another Prolog program

\[
\begin{align*}
on(a,b). \\
on(b,c).
\end{align*}
\]

\[
\begin{align*}
\text{above}(X,Y) & \leftarrow \text{above}(X,Z), \ on(Z,Y). \\
\text{above}(X,Y) & \leftarrow \ on(X,Y).
\end{align*}
\]

Prolog queries

\[
\begin{align*}
?\text{-} \text{above}(a,c).
\end{align*}
\]

Fatal Error: local stack overflow.
SAT-style playing with blocks

Formula

\[\begin{align*}
on(a, b) \\
\land \quad & on(b, c) \\
\land \quad & (on(X, Y) \rightarrow above(X, Y)) \\
\land \quad & (on(X, Z) \land above(Z, Y) \rightarrow above(X, Y))
\end{align*}\]

Herbrand model

\[\{ on(b, b), on(a, b), on(b, c), on(a, c), \\
above(b, b), above(c, b), above(a, b), \\
above(b, c), above(c, c), above(a, c) \}\]
SAT-style playing with blocks

Formula

\[\begin{align*}
on(a, b) \\
\land on(b, c) \\
\land (on(X, Y) \rightarrow above(X, Y)) \\
\land (on(X, Z) \land above(Z, Y) \rightarrow above(X, Y))
\end{align*}\]

Herbrand model

\{on(b, b), on(a, b), on(b, c), on(a, c), above(b, b), above(c, b), above(a, b), above(b, c), above(c, c), above(a, c) \}
SAT-style playing with blocks

Formula

\[
\begin{align*}
on(a, b) \\
\land & \quad on(b, c) \\
\land & \quad (on(X, Y) \rightarrow above(X, Y)) \\
\land & \quad (on(X, Z) \land above(Z, Y) \rightarrow above(X, Y))
\end{align*}
\]

Herbrand model (among 426!)

\{on(b, b), on(a, b), on(b, c), on(a, c), above(b, b), above(c, b), above(a, b), above(b, c), above(c, c), above(a, c) \}
Motivation: Overview

1. Motivation
2. Nutshell
3. Shifting paradigms
4. Rooting ASP
5. Problem solving
6. Use
KR’s shift of paradigm

Theorem Proving based approach (eg. Prolog)

1. Provide a representation of the problem.
2. A solution is given by a derivation of a query.

Model Generation based approach (eg. SATisfiability testing)

1. Provide a representation of the problem.
2. A solution is given by a model of the representation.
KR’s shift of paradigm

Theorem Proving based approach  (eg. Prolog)
1. Provide a representation of the problem.
2. A solution is given by a derivation of a query.

Model Generation based approach  (eg. SATisfiability testing)
1. Provide a representation of the problem.
2. A solution is given by a model of the representation.

Answer Set Programming (ASP)
## Model Generation based Problem Solving

<table>
<thead>
<tr>
<th>Representation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint satisfaction problem</td>
<td>assignment</td>
</tr>
<tr>
<td>propositional horn theories</td>
<td>smallest model</td>
</tr>
<tr>
<td>propositional theories</td>
<td>models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>stable models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>supported models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>Herbrand models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>expansions</td>
</tr>
<tr>
<td>auto-epistemic theories</td>
<td>extensions</td>
</tr>
<tr>
<td>default theories</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
</tbody>
</table>
### Answer Set Programming at large

<table>
<thead>
<tr>
<th>Representation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint satisfaction problem</td>
<td>assignment</td>
</tr>
<tr>
<td>propositional horn theories</td>
<td>smallest model</td>
</tr>
<tr>
<td>propositional theories</td>
<td>models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>stable models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>supported models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>Herbrand models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>expansions</td>
</tr>
<tr>
<td>first-order theories</td>
<td>extensions</td>
</tr>
<tr>
<td>auto-epistemic theories</td>
<td></td>
</tr>
<tr>
<td>default theories</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>Representation</td>
<td>Solution</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>constraint satisfaction problem</td>
<td>assignment</td>
</tr>
<tr>
<td>propositional horn theories</td>
<td>smallest model</td>
</tr>
<tr>
<td>propositional theories</td>
<td>models</td>
</tr>
<tr>
<td>minimal models</td>
<td>stable models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>stable models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>supported models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>models</td>
</tr>
<tr>
<td>minimal models</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>Herbrand models</td>
</tr>
<tr>
<td>auto-epistemic theories</td>
<td>expansions</td>
</tr>
<tr>
<td>default theories</td>
<td>extensions</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>
## Answer Set Programming *in practice*

**Representation**

<table>
<thead>
<tr>
<th>Constraint satisfaction problem</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propositional horn theories</td>
<td>Smallest model</td>
</tr>
<tr>
<td>Propositional theories</td>
<td>Models</td>
</tr>
<tr>
<td>Propositional theories</td>
<td>Minimal models</td>
</tr>
<tr>
<td>Propositional programs</td>
<td>Stable models</td>
</tr>
<tr>
<td>Propositional programs</td>
<td>Minimal models</td>
</tr>
<tr>
<td><strong>Propositional programs</strong></td>
<td>Supported models</td>
</tr>
<tr>
<td>First-order theories</td>
<td>Stable models</td>
</tr>
<tr>
<td>First-order theories</td>
<td>Models</td>
</tr>
<tr>
<td>First-order theories</td>
<td>Minimal models</td>
</tr>
<tr>
<td>First-order theories</td>
<td>Stable models</td>
</tr>
<tr>
<td>First-order theories</td>
<td>Herbrand models</td>
</tr>
<tr>
<td>Auto-epistemic theories</td>
<td>Expansions</td>
</tr>
<tr>
<td>Default theories</td>
<td>Extensions</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

**Solution**

<table>
<thead>
<tr>
<th>Constraint satisfaction problem</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propositional horn theories</td>
<td>Smallest model</td>
</tr>
<tr>
<td>Propositional theories</td>
<td>Models</td>
</tr>
<tr>
<td>Propositional theories</td>
<td>Minimal models</td>
</tr>
<tr>
<td>Propositional programs</td>
<td>Stable models</td>
</tr>
<tr>
<td>Propositional programs</td>
<td>Minimal models</td>
</tr>
<tr>
<td><strong>Propositional programs</strong></td>
<td>Supported models</td>
</tr>
<tr>
<td>First-order theories</td>
<td>Stable models</td>
</tr>
<tr>
<td>First-order theories</td>
<td>Herbrand models</td>
</tr>
<tr>
<td>First-order theories</td>
<td>Expansions</td>
</tr>
<tr>
<td>First-order theories</td>
<td>Extensions</td>
</tr>
<tr>
<td>Auto-epistemic theories</td>
<td>:</td>
</tr>
<tr>
<td>Default theories</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>
# Answer Set Programming *in practice*

<table>
<thead>
<tr>
<th>Representation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint satisfaction problem</td>
<td>assignment</td>
</tr>
<tr>
<td>propositional horn theories</td>
<td>smallest model</td>
</tr>
<tr>
<td>propositional theories</td>
<td>models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>stable models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>minimal models</td>
</tr>
<tr>
<td>supported models</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>Herbrand models</td>
</tr>
<tr>
<td>auto-epistemic theories</td>
<td>expansions</td>
</tr>
<tr>
<td>default theories</td>
<td>extensions</td>
</tr>
<tr>
<td>first-order programs</td>
<td>stable Herbrand models</td>
</tr>
</tbody>
</table>
ASP-style playing with blocks

Prolog program

\[
\text{on}(a,b).
\text{on}(b,c).
\]

\[
\text{above}(X,Y) :- \text{on}(X,Y).
\text{above}(X,Y) :- \text{on}(X,Z), \text{above}(Z,Y).
\]

Stable Herbrand model

\[
\{ \text{on}(a,b), \text{on}(b,c), \text{above}(b,c), \text{above}(a,b), \text{above}(a,c) \}
\]
ASP-style playing with blocks

Prolog program

\[
\text{on}(a, b).
\text{on}(b, c).
\]

\[
\text{above}(X, Y) \leftarrow \text{on}(X, Y).
\text{above}(X, Y) \leftarrow \text{on}(X, Z), \text{above}(Z, Y).
\]

Stable Herbrand model

\[
\{ \text{on}(a, b), \text{on}(b, c), \text{above}(b, c), \text{above}(a, b), \text{above}(a, c) \}
\]
ASP-style playing with blocks

Prolog program

\begin{verbatim}
on(a,b).
on(b,c).

above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
\end{verbatim}

Stable Herbrand model (and no others)

\{ on(a,b), on(b,c), above(b,c), above(a,b), above(a,c) \}
### ASP versus LP

<table>
<thead>
<tr>
<th>ASP</th>
<th>Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model generation</td>
<td>Query orientation</td>
</tr>
<tr>
<td>Bottom-up</td>
<td>Top-down</td>
</tr>
<tr>
<td>Modeling language</td>
<td>Programming language</td>
</tr>
</tbody>
</table>

#### Rule-based format

- Instantiation
- Flat terms
- (Turing +) $NP^{(NP)}$

- Unification
- Nested terms
- Turing
### ASP versus SAT

<table>
<thead>
<tr>
<th>ASP</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model generation</td>
<td></td>
</tr>
<tr>
<td>Bottom-up</td>
<td></td>
</tr>
<tr>
<td>Constructive Logic</td>
<td>Classical Logic</td>
</tr>
<tr>
<td>Closed (and open) world reasoning</td>
<td>Open world reasoning</td>
</tr>
<tr>
<td>Modeling language</td>
<td>―</td>
</tr>
<tr>
<td>Complex reasoning modes</td>
<td>Satisfiability testing</td>
</tr>
<tr>
<td>Satisfiability</td>
<td>Satisfiability</td>
</tr>
<tr>
<td>Enumeration/Projection</td>
<td>―</td>
</tr>
<tr>
<td>Optimization</td>
<td>―</td>
</tr>
<tr>
<td>Intersection/Union</td>
<td>―</td>
</tr>
<tr>
<td>(Turing +) $NP(NP)$</td>
<td>$NP$</td>
</tr>
</tbody>
</table>
Motivation: Overview

1 Motivation

2 Nutshell

3 Shifting paradigms

4 Rooting ASP

5 Problem solving

6 Use
Problem solving

ASP solving

Problem

Modeling

Logic Program

Grounder

Solver

Stable Models

Solution

Interpreting

Solving

Torsten Schaub et al. (KRR@UP)  Modeling and Solving in ASP  23 / 226
SAT solving

Problem solving

Problem

Formula (CNF)

Solver

Solution

Classical Models

Programming

Interpreting

Solving
Problem solving

Rooting ASP solving

- Problem
- Logic Program
- Grounder
- Solver
- Stable Models

Modeling

Solving

Interpreting

Torsten Schaub et al. (KRR@UP) Modeling and Solving in ASP
Rooting ASP solving

Modeling

Logic Program

Problem

KR

DB

Grounder

Solver

Solving

SAT

Stable Models

Solution

Interpreting

KR+LP

LP

Modeling and Solving in ASP

Torsten Schaub et al. (KRR@UP)
Motivation: Overview

1. Motivation
2. Nutshell
3. Shifting paradigms
4. Rooting ASP
5. Problem solving
6. Use
What is ASP good for?

- Combinatorial search problems in the realm of $P$, $NP$, and $NP^{NP}$ (some with substantial amount of data), like
  - Automated Planning,
  - Code Optimization,
  - Composition of Renaissance Music,
  - Database Integration,
  - Decision Support for NASA shuttle controllers,
  - Model Checking,
  - Product Configuration,
  - Robotics,
  - System Biology,
  - System Synthesis,
  - (industrial) Team-building,
  - and many many more.
What is ASP good for?

- Combinatorial search problems in the realm of \(P, NP,\) and \(NP^{NP}\) (some with substantial amount of data), like
  - Automated Planning,
  - Code Optimization,
  - Composition of Renaissance Music,
  - Database Integration,
  - Decision Support for NASA shuttle controllers,
  - Model Checking,
  - Product Configuration,
  - Robotics,
  - System Biology,
  - System Synthesis,
  - (industrial) Team-building,
  - and many many more.
What does ASP offer?

- Integration of KR, DB, and SAT techniques
- Succinct, elaboration-tolerant 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 SAT techniques
- Succinct, elaboration-tolerant problem representations
  - Rapid application development tool
- Easy handling of dynamic, knowledge intensive applications
  - including: data, frame axioms, exceptions, defaults, closures, etc.

\[
\text{ASP} = \text{KR} + \text{DB} + \text{SAT} + \text{LP}
\]
Introduction: Overview

7 Syntax

8 Semantics

9 Examples

10 Variables

11 Language Constructs

12 Reasoning Modes
Problem solving in ASP: Syntax

- Problem
- Modeling
- Logic Program
- Solving
- Solution
- Interpreting
- Stable Models
Normal logic programs

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

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

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

  - $\text{head}(r) = A_0$
  - $\text{body}(r) = \{A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n\}$
  - $\text{body}(r)^+ = \{A_1, \ldots, A_m\}$
  - $\text{body}(r)^- = \{A_{m+1}, \ldots, A_n\}$

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

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

\[
A_0 \leftarrow A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n,
\]

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

\[
\begin{align*}
\text{head}(r) &= A_0 \\
\text{body}(r) &= \{A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n\} \\
\text{body}(r)^+ &= \{A_1, \ldots, A_m\} \\
\text{body}(r)^- &= \{A_{m+1}, \ldots, A_n\}
\end{align*}
\]

- A program is called positive if \( \text{body}(r)^- = \emptyset \) for all its rules.
Normal logic programs

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

\[
A_0 \leftarrow A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n,
\]

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

\[
\begin{align*}
\text{head}(r) &= A_0 \\
\text{body}(r) &= \{A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n\} \\
\text{body}(r)^+ &= \{A_1, \ldots, A_m\} \\
\text{body}(r)^- &= \{A_{m+1}, \ldots, A_n\}
\end{align*}
\]

- A program is called positive if \( \text{body}(r)^- = \emptyset \) for all its rules.
(Rough) notational convention

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

<table>
<thead>
<tr>
<th></th>
<th>if</th>
<th>and</th>
<th>or</th>
<th>negation as failure</th>
<th>classical negation</th>
</tr>
</thead>
<tbody>
<tr>
<td>source code</td>
<td>:-</td>
<td>,</td>
<td>;</td>
<td>not</td>
<td>¬</td>
</tr>
<tr>
<td>logic program</td>
<td>←</td>
<td>,</td>
<td>;</td>
<td>not/∼</td>
<td>¬</td>
</tr>
<tr>
<td>formula</td>
<td>→</td>
<td>∧</td>
<td>∨</td>
<td>∼/(¬)</td>
<td>¬</td>
</tr>
</tbody>
</table>
Introduction: Overview
Problem solving in ASP: Semantics

- Problem
  - Modeling
  - Logic Program
- Solution
  - Interpreting
  - Stable Models
- Solving
Answer set: Formal Definition

Positive programs

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

$\bar{X}$ corresponds to a model of $\Pi$ (seen as a formula).

The smallest set of atoms which is closed under a positive program $\Pi$ is denoted by $Cn(\Pi)$.

$Cn(\Pi)$ corresponds to the $\subseteq$-smallest model of $\Pi$ (ditto).

The set $Cn(\Pi)$ of atoms is the answer set of a positive program $\Pi$. 
A set of atoms $X$ is closed under a positive program $\Pi$ iff for any $r \in \Pi$, $\text{head}(r) \in X$ whenever $\text{body}(r)^+ \subseteq X$.

$\bar{X}$ corresponds to a model of $\Pi$ (seen as a formula).

The smallest set of atoms which is closed under a positive program $\Pi$ is denoted by $Cn(\Pi)$.

$\bar{Cn}(\Pi)$ corresponds to the $\subseteq$-smallest model of $\Pi$ (ditto).

The set $Cn(\Pi)$ of atoms is the answer set of a positive program $\Pi$. 
Answer set: Formal Definition

Positive programs

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

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

  \[ A_0 \lor \neg A_1 \lor \cdots \lor \neg A_m \]

- 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.
  - Non-definite Horn clauses can be regarded as integrity constraints.
  - A set of Horn clauses has a smallest model or none.

- This smallest model is the intended semantics of such set of clauses.
  - Given a positive program \( \Pi \), \( Cn(\Pi) \) corresponds to the smallest model of the set of definite clauses corresponding to \( \Pi \).
Some “logical” remarks

- Positive rules are also referred to as definite clauses.
  - Definite clauses are disjunctions with exactly one positive atom:
    \[ A_0 \lor \neg A_1 \lor \cdots \lor \neg A_m \]
  - 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.
  - Non-definite Horn clauses can be regarded as integrity constraints.
  - A set of Horn clauses has a smallest model or none.

- This smallest model is the intended semantics of such set of clauses.
  - Given a positive program \( \Pi \), \( Cn(\Pi) \) corresponds to the smallest model of the set of definite clauses corresponding to \( \Pi \).
Some “logical” remarks

- Positive rules are also referred to as definite clauses.
  - Definite clauses are disjunctions with exactly one positive atom:
    \[ A_0 \lor \neg A_1 \lor \cdots \lor \neg A_m \]
  - 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.
  - Non-definite Horn clauses can be regarded as integrity constraints.
  - A set of Horn clauses has a smallest model or none.

- This smallest model is the intended semantics of such set of clauses.
  - Given a positive program \( \Pi \), \( Cn(\Pi) \) corresponds to the smallest model of the set of definite clauses corresponding to \( \Pi \).
Answer set: Basic idea

Consider the logical formula $\Phi$ and its three (classical) models:

\[
\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}
\]

Formula $\Phi$ has one stable model, often called answer set:

\[
\{p, q\}
\]

Informally, a set $X$ of atoms is an answer set of a logic program $\Pi$ if $X$ is a (classical) model of $\Pi$ and if all atoms in $X$ are justified by some rule in $\Pi$ (rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))
Consider the logical formula $\Phi$ and its three (classical) models:

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

Formula $\Phi$ has one stable model, often called answer set:

$\{p, q\}$

Informally, a set $X$ of atoms is an answer set of a logic program $\Pi$ if $X$ is a (classical) model of $\Pi$ and if all atoms in $X$ are justified by some rule in $\Pi$ (rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))
Consider the logical formula $\Phi$ and its three (classical) models:

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

Formula $\Phi$ has one stable model, often called answer set:

$$\{p, q\}$$

Informally, a set $X$ of atoms is an answer set of a logic program $\Pi$ if

- $X$ is a (classical) model of $\Pi$ and
- if all atoms in $X$ are justified by some rule in $\Pi$

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))
Answer set: Basic idea

Consider the logical formula $\Phi$ and its three (classical) models:

\[
\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}
\]

Formula $\Phi$ has one stable model, often called answer set:

\[
\{p, q\}
\]

Informally, a set $X$ of atoms is an answer set of a logic program $\Pi$

- if $X$ is a (classical) model of $\Pi$ and
- if all atoms in $X$ are justified by some rule in $\Pi$

(rooted in intuitionistic logics $\text{HT}$ (Heyting, 1930) and $\text{G3}$ (Gödel, 1932))
Answer set: Basic idea

Consider the logical formula $\Phi$ and its three (classical) models:

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

Formula $\Phi$ has one stable model, often called answer set:

$$\{p, q\}$$

Informally, a set $X$ of atoms is an answer set of a logic program $\Pi$ if

- $X$ is a (classical) model of $\Pi$ and
- if all atoms in $X$ are justified by some rule in $\Pi$

(rooted in intuitionistic logics $\text{HT}$ (Heyting, 1930) and $\text{G3}$ (Gödel, 1932))
Answer set: Basic idea

Consider the logical formula $\Phi$ and its three (classical) models:

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

Formula $\Phi$ has one stable model, often called answer set:

$$\{p, q\}$$

Informally, a set $X$ of atoms is an answer set of a logic program $\Pi$
- if $X$ is a (classical) model of $\Pi$ and
- if all atoms in $X$ are justified by some rule in $\Pi$

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))
Consider the logical formula Φ and its three (classical) models:

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

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

\{p, q\}

Informally, a set \(X\) of atoms is an answer set of a logic program \(\Pi\)

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

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))
Consider the logical formula $\Phi$ and its three (classical) models:

\[ \{p, q\}, \{q, r\}, \text{ and } \{p, q, r\} \]

Formula $\Phi$ has one stable model, often called answer set:

\[ \{p, q\} \]

Informally, a set $X$ of atoms is an answer set of a logic program $\Pi$ if

- if $X$ is a (classical) model of $\Pi$ and
- if all atoms in $X$ are justified by some rule in $\Pi$

(rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))
Answer set: Formal Definition

Normal programs

The reduct, $\Pi^X$, of a program $\Pi$ relative to a set $X$ of atoms is defined by

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

A set $X$ of atoms is a stable model of a program $\Pi$, if $\text{Cn}(\Pi^X) = X$.

Note: $\text{Cn}(\Pi^X)$ is the $\subseteq$–smallest (classical) model of $\Pi^X$.

Note: Every atom in $X$ is justified by an “applying rule from $\Pi$”

Sorry, but: We interchangeably use the terms answer set and stable model.
The reduct, $\Pi^X$, of a program $\Pi$ relative to a set $X$ of atoms is defined by

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

A set $X$ of atoms is a stable model of a program $\Pi$, if $Cn(\Pi^X) = X$.

Note: $Cn(\Pi^X)$ is the $\subseteq$–smallest (classical) model of $\Pi^X$.

Note: Every atom in $X$ is justified by an “applying rule from $\Pi$”

Sorry, but: We interchangeably use the terms answer set and stable model.
Answer set: Formal Definition

Normal programs

- The reduct, $\Pi^X$, of a program $\Pi$ relative to a set $X$ of atoms is defined by
  \[
  \Pi^X = \{ \text{head}(r) \leftarrow \text{body}(r)^+ \mid r \in \Pi \text{ and } \text{body}(r)^- \cap X = \emptyset \}.
  \]

- A set $X$ of atoms is a stable model of a program $\Pi$, if $Cn(\Pi^X) = X$.

- Note: $Cn(\Pi^X)$ is the $\subseteq$-smallest (classical) model of $\Pi^X$.

- Note: Every atom in $X$ is justified by an "applying rule from $\Pi$"

- Sorry, but: We interchangeably use the terms answer set and stable model.
Answer set: Formal Definition

Normal programs

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

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

- A set $X$ of atoms is a stable model of a program $\Pi$, if $\text{Cn}(\Pi^X) = X$.

- Note: $\text{Cn}(\Pi^X)$ is the $\subseteq$-smallest (classical) model of $\Pi^X$.

- Note: Every atom in $X$ is justified by an "applying rule from $\Pi$".

- Sorry, but: We interchangeably use the terms answer set and stable model.
A closer look at $\Pi^X$

In other words, given a set $X$ of atoms from $\Pi$,

$\Pi^X$ is obtained from $\Pi$ by deleting

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

Note: Only negative body literals are evaluated wrt $X$
A closer look at $\Pi^X$

- In other words, given a set $X$ of atoms from $\Pi$,

$\Pi^X$ is obtained from $\Pi$ 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.

- Note: Only negative body literals are evaluated wrt $X$
Introduction: Overview

7 Syntax

8 Semantics

9 Examples

10 Variables

11 Language Constructs

12 Reasoning Modes
A first example

\[ \Pi = \{ p \leftarrow p, \ q \leftarrow \neg p \} \]

<table>
<thead>
<tr>
<th>X</th>
<th>( \Pi^X )</th>
<th>( \text{Cn}(\Pi^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>( p \leftarrow p )</td>
<td>{q}</td>
</tr>
<tr>
<td>{p}</td>
<td>( p \leftarrow p )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>{q}</td>
<td>( p \leftarrow p )</td>
<td>{q}</td>
</tr>
<tr>
<td>{p, q}</td>
<td>( p \leftarrow p )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
A first example

\[ \Pi = \{ p \leftarrow p, \; q \leftarrow \text{not } p \} \]

<table>
<thead>
<tr>
<th>\chi \</th>
<th>\ \Pi^X \</th>
<th>\ \text{Cn}(\Pi^X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\emptyset</td>
<td>{ p }</td>
<td>{ q }  \times</td>
</tr>
<tr>
<td>{ p }</td>
<td>{ p }</td>
<td>\emptyset \times</td>
</tr>
<tr>
<td>{ q }</td>
<td>{ q }</td>
<td>{ q }  \checkmark</td>
</tr>
<tr>
<td>{ p, q }</td>
<td>{ p }</td>
<td>\emptyset \times</td>
</tr>
</tbody>
</table>
### A first example

\[ \Pi = \{ p \leftarrow p, \ q \leftarrow \text{not } p \} \]

<table>
<thead>
<tr>
<th>X</th>
<th>( \Pi^X )</th>
<th>( Cn(\Pi^X) )</th>
</tr>
</thead>
</table>
| \( \emptyset \) | \( p \leftarrow p \)
|      | \( q \leftarrow \) | \( \{ q \} \times \) |
| \{ p \} | \( p \leftarrow p \) | \( \emptyset \times \) |
| \{ q \} | \( p \leftarrow p \)
|      | \( q \leftarrow \) | \( \{ q \} \checkmark \) |
| \{ p, q \} | \( p \leftarrow p \) | \( \emptyset \times \) |
### A first example

\[ \Pi = \{ p \leftarrow p, \; q \leftarrow \text{not } p \} \]

<table>
<thead>
<tr>
<th>\chi</th>
<th>\Pi^x</th>
<th>Cn(\Pi^x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\emptyset</td>
<td>p \leftarrow p \quad q \leftarrow \emptyset</td>
<td>{ q } \quad \times</td>
</tr>
<tr>
<td>{ p }</td>
<td>p \leftarrow p</td>
<td>\emptyset \quad \times</td>
</tr>
<tr>
<td>{ q }</td>
<td>p \leftarrow p \quad q \leftarrow \emptyset</td>
<td>{ q } \quad \checkmark</td>
</tr>
<tr>
<td>{ p, q }</td>
<td>p \leftarrow p</td>
<td>\emptyset \quad \times</td>
</tr>
</tbody>
</table>
A first example

\[ \Pi = \{ p \leftarrow p, \ q \leftarrow \text{not } p \} \]

<table>
<thead>
<tr>
<th>(X)</th>
<th>(\Pi^X)</th>
<th>(Cn(\Pi^X))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\emptyset)</td>
<td>(p \leftarrow p)</td>
<td>({q})</td>
</tr>
<tr>
<td></td>
<td>(q \leftarrow )</td>
<td>(\times)</td>
</tr>
<tr>
<td>({p})</td>
<td>(p \leftarrow p)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td></td>
<td>(q \leftarrow )</td>
<td>(\times)</td>
</tr>
<tr>
<td>({q})</td>
<td>(p \leftarrow p)</td>
<td>({q})</td>
</tr>
<tr>
<td></td>
<td>(q \leftarrow )</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>({p, q})</td>
<td>(p \leftarrow p)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td></td>
<td>(q \leftarrow )</td>
<td>(\times)</td>
</tr>
</tbody>
</table>
A first example

\[ \Pi = \{ p \leftarrow p, \ q \leftarrow \text{not } p \} \]

<table>
<thead>
<tr>
<th>( \chi )</th>
<th>( \Pi^\chi )</th>
<th>( \text{Cn}(\Pi^\chi) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>( p \leftarrow p )</td>
<td>{ q }</td>
</tr>
<tr>
<td>{ p }</td>
<td>( p \leftarrow p )</td>
<td>\emptyset</td>
</tr>
<tr>
<td>{ q }</td>
<td>( p \leftarrow p )</td>
<td>{ q }</td>
</tr>
<tr>
<td>{ p, q }</td>
<td>( p \leftarrow p )</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>
### A first example

\[ \Pi = \{ p \leftarrow p, \ q \leftarrow \text{not } p \} \]

<table>
<thead>
<tr>
<th>( \chi )</th>
<th>( \Pi^\chi )</th>
<th>( \text{Cn}(\Pi^\chi) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>( p \leftarrow p ) &lt;br&gt;( q \leftarrow )</td>
<td>( { q } ) \xmark</td>
</tr>
<tr>
<td>( { p } )</td>
<td>( p \leftarrow p )</td>
<td>( \emptyset ) \xmark</td>
</tr>
<tr>
<td>( { q } )</td>
<td>( p \leftarrow p )</td>
<td>( { q } ) \checkmark</td>
</tr>
<tr>
<td>( { p, q } )</td>
<td>( p \leftarrow p )</td>
<td>( \emptyset ) \xmark</td>
</tr>
</tbody>
</table>
A second example

\[ \Pi = \{ p \leftarrow \text{not } q, \ q \leftarrow \text{not } p \} \]

<table>
<thead>
<tr>
<th>( \chi )</th>
<th>( \Pi^X )</th>
<th>( Cn(\Pi^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>( p \leftarrow )</td>
<td>( { p, q } )</td>
</tr>
<tr>
<td>&amp; ( q \leftarrow )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{p}</td>
<td>( p \leftarrow )</td>
<td>( { p } )</td>
</tr>
<tr>
<td>&amp; ( )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{q}</td>
<td>( q \leftarrow )</td>
<td>( { q } )</td>
</tr>
<tr>
<td>&amp; ( )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{p, q}</td>
<td></td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
A second example

\[ \Pi = \{ p \leftarrow \text{not } q, \ q \leftarrow \text{not } p \} \]

<table>
<thead>
<tr>
<th>( \mathcal{X} )</th>
<th>( \Pi^X )</th>
<th>( \text{Cn}(\Pi^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>( p \leftarrow )</td>
<td>{p, q}</td>
</tr>
<tr>
<td>{p}</td>
<td>( p \leftarrow )</td>
<td>{p}</td>
</tr>
<tr>
<td>{q}</td>
<td>( q \leftarrow )</td>
<td>{q}</td>
</tr>
<tr>
<td>{p, q}</td>
<td>( q \leftarrow )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
A second example

\[ \Pi = \{ p \leftarrow \text{not } q, \ q \leftarrow \text{not } p \} \]

<table>
<thead>
<tr>
<th>(X)</th>
<th>(\Pi^X)</th>
<th>(Cn(\Pi^X))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\emptyset)</td>
<td>(p \leftarrow)</td>
<td>({p, q})</td>
</tr>
<tr>
<td>({p})</td>
<td>(p \leftarrow)</td>
<td>({p})</td>
</tr>
<tr>
<td>({q})</td>
<td>(q \leftarrow)</td>
<td>({q})</td>
</tr>
<tr>
<td>({p, q})</td>
<td>(q \leftarrow)</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>
A second example

\[ \Pi = \{ p \leftarrow \text{not } q, \; q \leftarrow \text{not } p \} \]

<table>
<thead>
<tr>
<th>( \chi )</th>
<th>( \Pi^X )</th>
<th>( Cn(\Pi^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>( p \leftarrow )</td>
<td>( { p, q } ) ( \times )</td>
</tr>
<tr>
<td>( { p } )</td>
<td>( p \leftarrow )</td>
<td>( { p } ) ( \checkmark )</td>
</tr>
<tr>
<td>( { q } )</td>
<td>( q \leftarrow )</td>
<td>( { q } ) ( \checkmark )</td>
</tr>
<tr>
<td>( { p, q } )</td>
<td>( q \leftarrow )</td>
<td>( \emptyset ) ( \times )</td>
</tr>
</tbody>
</table>
### Examples

A second example

\[ \Pi = \{ p \leftarrow \text{not } q, \ q \leftarrow \text{not } p \} \]

<table>
<thead>
<tr>
<th>( \chi )</th>
<th>( \Pi^X )</th>
<th>( Cn(\Pi^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>( p \leftarrow )</td>
<td>{( p, q }} \times</td>
</tr>
<tr>
<td>{( p }}</td>
<td>( p \leftarrow )</td>
<td>{( p }} \checkmark</td>
</tr>
<tr>
<td>{( q }}</td>
<td>( q \leftarrow )</td>
<td>{( q }} \checkmark</td>
</tr>
<tr>
<td>{( p, q }}</td>
<td>q \leftarrow</td>
<td>\emptyset \times</td>
</tr>
</tbody>
</table>
A second example

\[ \Pi = \{ p \leftarrow \text{not } q, \ q \leftarrow \text{not } p \} \]

<table>
<thead>
<tr>
<th>(X)</th>
<th>(\Pi^X)</th>
<th>(Cn(\Pi^X))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\emptyset)</td>
<td>(p \leftarrow)</td>
<td>({p, q}) (\times)</td>
</tr>
<tr>
<td>({p})</td>
<td>(p \leftarrow)</td>
<td>({p}) (\checkmark)</td>
</tr>
<tr>
<td>({q})</td>
<td>(q \leftarrow)</td>
<td>({q}) (\checkmark)</td>
</tr>
<tr>
<td>({p, q})</td>
<td>(q \leftarrow)</td>
<td>(\emptyset) (\times)</td>
</tr>
</tbody>
</table>
A third example

\[ \Pi = \{ p \leftarrow \text{not } p \} \]

<table>
<thead>
<tr>
<th>( X )</th>
<th>( \Pi^X )</th>
<th>( \text{Cn}(\Pi^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>( p \leftarrow )</td>
<td>( { p } )</td>
</tr>
<tr>
<td>( { p } )</td>
<td>( )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>
A third example

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

<table>
<thead>
<tr>
<th>$X$</th>
<th>$\Pi^X$</th>
<th>$Cn(\Pi^X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>$p \leftarrow$</td>
<td>${p}$</td>
</tr>
<tr>
<td>${p}$</td>
<td></td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
A third example

\[ \Pi = \{ p \leftarrow \text{not } p \} \]

<table>
<thead>
<tr>
<th>(\chi)</th>
<th>(\Pi^\chi)</th>
<th>(Cn(\Pi^\chi))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\emptyset)</td>
<td>(p \leftarrow)</td>
<td>({p})</td>
</tr>
<tr>
<td>({p})</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>
### A third example

\[ \Pi = \{ p \leftarrow \text{not } p \} \]

<table>
<thead>
<tr>
<th>( X )</th>
<th>( \Pi^X )</th>
<th>( Cn(\Pi^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>( p \leftarrow )</td>
<td>( { p } )</td>
</tr>
<tr>
<td>{ p }</td>
<td></td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

\( \times \) means not in \( Cn(\Pi^X) \).
A logic program may have zero, one, or multiple answer sets!

- If $X$ is an answer set of a logic program $\Pi$, then $X$ is a model of $\Pi$ (seen as a formula).
- If $X$ and $Y$ are answer sets of a normal program $\Pi$, then $X \not\subset Y$. 
Answer set: Some properties

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

7 Syntax

8 Semantics

9 Examples

10 Variables

11 Language Constructs

12 Reasoning Modes
Let $\Pi$ be a logic program.

- Let $\mathcal{T}$ be a set of (variable-free) terms
- Let $\mathcal{A}$ be a set of (variable-free) atoms constructable from $\mathcal{T}$

- Ground Instances of $r \in \Pi$: Set of variable-free rules obtained by replacing all variables in $r$ by elements from $\mathcal{T}$:

$$\text{ground}(r) = \{r\theta \mid \theta : \text{var}(r) \rightarrow \mathcal{T}\}$$

where $\text{var}(r)$ stands for the set of all variables occurring in $r$; $\theta$ is a (ground) substitution.

- Ground Instantiation of $\Pi$: $\text{ground}(\Pi) = \bigcup_{r \in \Pi}\text{ground}(r)$
Let $\Pi$ be a logic program.

- Let $\mathcal{T}$ be a set of (variable-free) terms (also called Herbrand universe).
- Let $\mathcal{A}$ be a set of (variable-free) atoms constructable from $\mathcal{T}$ (also called alphabet or Herbrand base).
- Ground Instances of $r \in \Pi$: Set of variable-free rules obtained by replacing all variables in $r$ by elements from $\mathcal{T}$:

$$\text{ground}(r) = \{ r\theta \mid \theta : \text{var}(r) \rightarrow \mathcal{T} \}$$

where $\text{var}(r)$ stands for the set of all variables occurring in $r$; $\theta$ is a (ground) substitution.

- Ground Instantiation of $\Pi$: $\text{ground}(\Pi) = \bigcup_{r \in \Pi} \text{ground}(r)$
Programs with Variables

Let $\Pi$ be a logic program.

- Let $T$ be a set of (variable-free) terms
- Let $A$ be a set of (variable-free) atoms constructable from $T$

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

  $$ground(r) = \{ r\theta \mid \theta : var(r) \rightarrow T \}$$

  where $var(r)$ stands for the set of all variables occurring in $r$; $\theta$ is a (ground) substitution.

- **Ground Instantiation of $\Pi$:**

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

Let $\Pi$ be a logic program.

- Let $\mathcal{T}$ be a set of (variable-free) terms
- Let $\mathcal{A}$ be a set of (variable-free) atoms constructable from $\mathcal{T}$

- **Ground Instances of** $r \in \Pi$: Set of variable-free rules obtained by replacing all variables in $r$ by elements from $\mathcal{T}$:

$$\text{ground}(r) = \{ r\theta \mid \theta : \text{var}(r) \rightarrow \mathcal{T} \}$$

where $\text{var}(r)$ stands for the set of all variables occurring in $r$; $\theta$ is a (ground) substitution.

- **Ground Instantiation of** $\Pi$: $\text{ground}(\Pi) = \bigcup_{r \in \Pi} \text{ground}(r)$
Variables

An example

\[ \Pi = \{ r(a, b) \leftarrow, r(b, c) \leftarrow, t(X, Y) \leftarrow r(X, Y) \} \]
\[ T = \{ a, b, c \} \]
\[ A = \{ r(a, a), r(a, b), r(a, c), r(b, a), r(b, b), r(b, c), r(c, a), r(c, b), r(c, c), t(a, a), t(a, b), t(a, c), t(b, a), t(b, b), t(b, c), t(c, a), t(c, b), t(c, c) \} \]

\[ \text{ground}(\Pi) = \{ \]
\[ r(a, b) \leftarrow, \]
\[ r(b, c) \leftarrow, \]
\[ t(a, a) \leftarrow r(a, a), t(b, a) \leftarrow r(b, a), t(c, a) \leftarrow r(c, a), \]
\[ t(a, b) \leftarrow r(a, b), t(b, b) \leftarrow r(b, b), t(c, b) \leftarrow r(c, b), \]
\[ t(a, c) \leftarrow r(a, c), t(b, c) \leftarrow r(b, c), t(c, c) \leftarrow r(c, c) \]
\[ \} \]

Intelligent Grounding aims at reducing the ground instantiation.
An example

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

\[ T = \{ a, b, c \} \]

\[ A = \{ r(a, a), r(a, b), r(a, c), r(b, a), r(b, b), r(b, c), r(c, a), r(c, b), r(c, c), \]
\[ t(a, a), t(a, b), t(a, c), t(b, a), t(b, b), t(b, c), t(c, a), t(c, b), t(c, c) \} \]

\[ \text{ground}(\Pi) = \{ r(a, b) \leftarrow , \]
\[ r(b, c) \leftarrow , \]
\[ t(a, a) \leftarrow r(a, a), \ t(b, a) \leftarrow r(b, a), \ t(c, a) \leftarrow r(c, a), \]
\[ t(a, b) \leftarrow r(a, b), \ t(b, b) \leftarrow r(b, b), \ t(c, b) \leftarrow r(c, b), \]
\[ t(a, c) \leftarrow r(a, c), \ t(b, c) \leftarrow r(b, c), \ t(c, c) \leftarrow r(c, c) \} \]

Intelligent Grounding aims at reducing the ground instantiation.
Variables

An example

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

\[ T = \{ a, b, c \} \]

\[ A = \{ r(a, a), r(a, b), r(a, c), r(b, a), r(b, b), r(b, c), r(c, a), r(c, b), r(c, c), \\
  t(a, a), t(a, b), t(a, c), t(b, a), t(b, b), t(b, c), t(c, a), t(c, b), t(c, c) \} \]

\[ \text{ground}(\Pi) = \left\{ \begin{array}{l}
  r(a, b) \leftarrow, \\
  r(b, c) \leftarrow, \\
  t(a, a) \leftarrow r(a, a), t(b, a) \leftarrow r(b, a), t(c, a) \leftarrow r(c, a), \\
  t(a, b) \leftarrow r(a, b), t(b, b) \leftarrow r(b, b), t(c, b) \leftarrow r(c, b), \\
  t(a, c) \leftarrow r(a, c), t(b, c) \leftarrow r(b, c), t(c, c) \leftarrow r(c, c)
\end{array} \right\} \]

- Intelligent Grounding aims at reducing the ground instantiation.
Answer sets of programs with Variables

Let $\Pi$ be a normal logic program with variables.

A set $X$ of (ground) atoms as a stable model of $\Pi$, if $\text{Cn}(\text{ground}(\Pi)^X) = X$. 
Answer sets of programs with Variables

Let $\Pi$ be a normal logic program with variables.

- A set $X$ of (ground) atoms as a stable model of $\Pi$, if $Cn(\text{ground}(\Pi)^X) = X$. 
Introduction: Overview

7 Syntax

8 Semantics

9 Examples

10 Variables

11 Language Constructs

12 Reasoning Modes
Problem solving in ASP: Extended Syntax

Problem \rightarrow Modeling \rightarrow Logic Program \rightarrow Solving \rightarrow Stable Models

Solution \rightarrow Interpreting
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
\[ p(X) \mid q(X) :- r(X) \]

Integrity Constraints
\[ :- q(X), p(X) \]

Choice
\[ 2 \{ p(X,Y) : q(X) \} 7 :- r(Y) \]

Aggregates
\[ s(Y) :- r(Y), 2 \#\text{count} \{ p(X,Y) : q(X) \} 7 \]
also: \#sum, \#avg, \#min, \#max, \#even, \#odd
Language Constructs

- **Variables (over the Herbrand Universe)**
  - \( p(X) \leftarrow q(X) \) over constants \( \{a, b, c\} \) stands for
    - \( p(a) \leftarrow q(a), p(b) \leftarrow q(b), p(c) \leftarrow q(c) \)

- **Conditional Literals**
  - \( p \leftarrow q(X) : r(X) \) given \( r(a), r(b), r(c) \) stands for
    - \( p \leftarrow q(a), q(b), q(c) \)

- **Disjunction**
  - \( p(X) \mid q(X) \leftarrow r(X) \)

- **Integrity Constraints**
  - \( :- q(X), p(X) \)

- **Choice**
  - \( 2 \{ p(X,Y) : q(X) \} 7 :- r(Y) \)

- **Aggregates**
  - \( s(Y) :- r(Y), 2 \#\text{count} \{ p(X,Y) : q(X) \} 7 \)
    - also: \#\text{sum}, \#\text{avg}, \#\text{min}, \#\text{max}, \#\text{even}, \#\text{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**
  - $p(X) \mid q(X) :- r(X)$

- **Integrity Constraints**
  - $:- q(X), p(X)$

- **Choice**
  - $2 \{ p(X,Y) :- q(X) \} 7 :- r(Y)$

- **Aggregates**
  - $s(Y) :- r(Y), 2 \#\text{count} \{ p(X,Y) :- q(X) \} 7$
  - also: \#\text{sum}, \#\text{avg}, \#\text{min}, \#\text{max}, \#\text{even}, \#\text{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**
  - \( p(X) \mid q(X) :- r(X) \)

- **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**
  - \( p(X) \mid q(X) :- r(X) \)

- **Integrity Constraints**
  - \( :- q(X), p(X) \)

- **Choice**
  - \( 2 \{ p(X,Y) : q(X) \} 7 :- r(Y) \)

- **Aggregates**
  - \( s(Y) :- r(Y), 2 \# \text{count} \{ p(X,Y) : q(X) \} 7 \)
  - also: \#\text{sum}, \#\text{avg}, \#\text{min}, \#\text{max}, \#\text{even}, \#\text{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
  - \( s(Y) :- r(Y), 2 \#\text{count} \{ p(X,Y) : q(X) \} 7 \)
  - 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
  - $p(X) \mid q(X) :- r(X)$

- 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**
  - \( p(X) \mid q(X) :- r(X) \)

- **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
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Syntax</td>
</tr>
<tr>
<td>8</td>
<td>Semantics</td>
</tr>
<tr>
<td>9</td>
<td>Examples</td>
</tr>
<tr>
<td>10</td>
<td>Variables</td>
</tr>
<tr>
<td>11</td>
<td>Language Constructs</td>
</tr>
<tr>
<td>12</td>
<td>Reasoning Modes</td>
</tr>
</tbody>
</table>
Problem solving in ASP: Reasoning Modes

- Problem
  - Modeling
  - Logic Program
  - Solving

- Solution
  - Interpreting
  - Stable Models

Reasoning Modes
Reasoning Modes

- Satisfiability
- Enumeration\(^\dagger\)
- Projection\(^\dagger\)
- Intersection\(^\ddagger\)
- Union\(^\ddagger\)
- Optimization

and combinations of them

\(^\dagger\) without solution recording
\(^\ddagger\) without solution enumeration
Basic Modeling: Overview

13 ASP Solving Process

14 Problems as Logic Programs
   ■ Graph Coloring

15 Methodology
   ■ Satisfiability
   ■ Queens
   ■ Reviewer Assignment
   ■ Planning
Modeling and Interpreting

Modeling

Logic Program

Solving

Stable Models

Interpreting

Problem

Solution
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 $C(I) \cup C(P)$. 
Basic Modeling: Overview

13 ASP Solving Process

14 Problems as Logic Programs
   - Graph Coloring

15 Methodology
   - Satisfiability
   - Queens
   - Reviewer Assignment
   - Planning
ASP Solving Process

- Program
- Grounder
- Solver
- Models
ASP Solving Process

Program → Grounder → Solver → Models
ASP Solving Process

Program → **Grounder** → **Solver** → Models
ASP Solving Process

Program → Grounder → Solver → Models
ASP Solving Process

- Program
- Grounder
- Solver
- Models
ASP Solving Process

Program → Grounder → Solver → Models
Basic Modeling: Overview

13 ASP Solving Process

14 Problems as Logic Programs
   ■ Graph Coloring

15 Methodology
   ■ Satisfiability
   ■ Queens
   ■ Reviewer Assignment
   ■ Planning
ASP Solving Process

- Program
- Grounder
- Solver
- Models
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).
```
ASP Solving Process

Program → Grounder → Solver → Models
Graph Coloring: Grounding

\$ \texttt{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(1,b), color(2,b).
:- color(1,g), color(2,g).
:- color(1,r), color(3,r).
:- color(1,b), color(3,b).
:- color(1,g), color(3,g).
:- color(1,r), color(4,r).
:- color(1,b), color(4,b).
:- color(1,g), color(4,g).
:- color(2,r), color(4,r).
:- color(2,b), color(4,b).
:- color(2,g), color(4,g).
:- color(2,r), color(5,r).
:- color(2,b), color(5,b).
:- color(2,g), color(5,g).
:- color(3,r), color(5,r).
:- color(3,b), color(5,b).
:- color(3,g), color(5,g).
:- color(4,r), color(5,r).
:- color(4,b), color(5,b).
:- color(4,g), color(5,g).
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,g). ... :- 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).
ASP Solving Process

Program → Grounder → Solver → Models
$ 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

clap 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 Modeling: Overview

13 ASP Solving Process

14 Problems as Logic Programs
- Graph Coloring

15 Methodology
- Satisfiability
- Queens
- Reviewer Assignment
- Planning
Methodology

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

Logic program = Data + Generator + Tester
(+ 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**

Logic program  =  Data + Generator + Tester
(+ Optimizer)
Satisfiability

- **Problem Instance:** A propositional formula $\phi$ in CNF.
- **Problem Class:** Is there an assignment of propositional variables to true and false such that a given formula $\phi$ is true.

- **Example:** Consider formula $(a \lor \neg b) \land (\neg a \lor b)$.
- **Logic Program:**

```
Generator  Tester  Stable models
\{a,b\} ← not a, b  \ X_1 = \{a,b\}
← a, not b  \ X_2 = \{\}
```
Satisfiability

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

- Example: Consider formula $(a \lor \neg b) \land (\neg a \lor b)$.

- Logic Program:

  
<table>
<thead>
<tr>
<th>Generator</th>
<th>Tester</th>
<th>Stable models</th>
</tr>
</thead>
<tbody>
<tr>
<td>${a,b}$</td>
<td>$\leftarrow$ not a, b</td>
<td>$X_1 = {a,b}$</td>
</tr>
<tr>
<td>$\leftarrow$ a, not b</td>
<td>$X_2 = {}$</td>
<td></td>
</tr>
</tbody>
</table>
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) \ncol(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

\begin{verbatim}
row(1..n).
col(1..n).
\{ queen(I,J) : row(I) : col(J) \}.
\end{verbatim}

- 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

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
```

1 2 3 4 5
Placing some Queens: Answer 2

Answer 2

A chessboard with two queens placed on it.
Placing some Queens: Answer 3

Answer 3

```
[1 2 3 4 5]
5  
4  
3  
2  
1  
1 2 3 4 5
```
Methodology

Queens

Placing $n$ Queens

queens.lp

```prolog
row(1..n).
col(1..n).
{ queen(I,J) : row(I) : col(J) }.
:- not n { 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) \ncol(1) col(2) col(3) col(4) col(5) \nqueen(5,1) queen(4,1) queen(3,1) \nqueen(2,1) queen(1,1)
Answer: 2
row(1) row(2) row(3) row(4) row(5) \ncol(1) col(2) col(3) col(4) col(5) \nqueen(1,2) queen(4,1) queen(3,1) \nqueen(2,1) queen(1,1)
...

Torsten Schaub et al. (KRR@UP)  Modeling and Solving in ASP 80 / 226
Placing $n$ Queens: Answer 1

Answer 1

```
1 2 3 4 5
1 L0Z0Z
2 QZ0Z0
3 L0Z0Z
4 QZ0Z0
5 L0Z0Z
```
Placing $n$ Queens: Answer 2

Answer 2

```
  5  
 4   3   2   1
 1   2   3   4   5
```
Horizontal and vertical Attack

queens.lp

\[
\begin{align*}
\text{row}(1..n). \\
\text{col}(1..n).
\end{align*}
\]

\[
\{ \text{queen}(I,J) : \text{row}(I) : \text{col}(J) \}. \\
\text{:- not } n \{ \text{queen}(I,J) \} \ n. \\
\text{:- queen}(I,J), \text{queen}(I,JJ), J != JJ. \\
\text{:- queen}(I,J), \text{queen}(II,J), I != II.
\]

- Forbid horizontal attacks
- Forbid vertical attacks
Horizontal and vertical Attack

queens.lp

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

- 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) \ncol(1) col(2) col(3) col(4) col(5) \nqueen(5,5) queen(4,4) queen(3,3) \nqueen(2,2) queen(1,1)
...
Horizontal and vertical Attack: Answer 1

Answer 1

[Diagram of a chessboard with queens placed in positions to avoid horizontal and vertical attacks]
Diagonal Attack

queens.lp

\text{row}(1..n).
\text{col}(1..n).
\{ \text{queen}(I,J) : \text{row}(I) : \text{col}(J) \}.
:- \text{not} \ n \ \{ \text{queen}(I,J) \} \ n.
:- \text{queen}(I,J), \text{queen}(I,JJ), J != JJ.
:- \text{queen}(I,J), \text{queen}(II,J), I != II.
:- \text{queen}(I,J), \text{queen}(II,JJ), (I,J) != (II,JJ),
    I-J == II-JJ.
:- \text{queen}(I,J), \text{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
   Prepare : 0.000
   Prepro. : 0.000
   Solving : 0.000
Diagonal Attack: Answer 1

Answer 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) }.

Torsten Schaub et al. (KRR@UP)
Modeling and Solving in ASP
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).
:- 6 { assigned(P,R) : paper(P) } , 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(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).
pree(a,p).
add(a,q).
del(a,p).

action(b).
pree(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

\[
\begin{align*}
\text{fluent}(p). & \quad \text{fluent}(q). & \quad \text{fluent}(r). \\
\text{action}(a). & \quad \text{pre}(a,p). & \quad \text{add}(a,q). & \quad \text{del}(a,p). \\
\text{action}(b). & \quad \text{pre}(b,q). & \quad \text{add}(b,r). & \quad \text{del}(b,q). \\
\text{init}(p). & \quad \text{query}(r). \\
\text{time}(1..k). & \quad \text{lasttime}(T) :- \text{time}(T), \neg \text{time}(T+1). \\
\text{holds}(P,0) :- \text{init}(P). \\
1 \{ \text{occ}(A,T) : \text{action}(A) \} 1 & :- \text{time}(T). \\
& :- \text{occ}(A,T), \text{pre}(A,F), \neg \text{holds}(F,T-1). \\
\text{ocdel}(F,T) :- \text{occ}(A,T), \text{del}(A,F). \\
\text{holds}(F,T) :- \text{occ}(A,T), \text{add}(A,F). \\
\text{holds}(F,T) :- \text{holds}(F,T-1), \neg \text{ocdel}(F,T), \text{time}(T). \\
& :- \text{query}(F), \neg \text{holds}(F,T), \text{lasttime}(T). 
\end{align*}
\]
Language Extensions: Overview

16 Motivation
17 Integrity Constraints
18 Choice Rules
19 Cardinality Constraints
20 Weight Constraints
21 Optimization statements
22 Conditional literals
23 smodels format
Language Extensions: Overview

16  Motivation

17  Integrity Constraints

18  Choice Rules

19  Cardinality Constraints

20  Weight Constraints

21  Optimization statements

22  Conditional literals

23  smodels format
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, e.g., classical negation.
- This translation might also be used for implementing the language extension.
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, e.g., classical negation.
- This translation might also be used for implementing the language extension.
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, e.g., classical negation.
- This translation might also be used for implementing the language extension.
Language Extensions: Overview

16 Motivation

17 Integrity Constraints

18 Choice Rules

19 Cardinality Constraints

20 Weight Constraints

21 Optimization statements

22 Conditional literals

23 smodels format
Integrity Constraints

- **Idea**: Eliminate unwanted solution candidates
- **Syntax**: An integrity constraint is of the form

\[ \leftarrow A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n, \]

where \( n \geq m \geq 1 \), and each \( A_i \) (\( 1 \leq i \leq n \)) is an atom

- **Example**: \(-\text{edge}(X,Y), \text{color}(X,C), \text{color}(Y,C).\)

- **Embedding**: For a new symbol \( x \), map

\[ \leftarrow A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n \]

\[ \mapsto x \leftarrow A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n, \text{not } x \]

- **Another example**: \( \Pi = \{ p \leftarrow \text{not } q, q \leftarrow \text{not } p \} \)

versus \( \Pi' = \Pi \cup \{ p \leftarrow p \} \) and \( \Pi'' = \Pi \cup \{ \text{not } p \leftarrow p \} \)
Integrity Constraints

- **Idea** Eliminate unwanted solution candidates
- **Syntax** An integrity constraint is of the form

\[
\leftarrow A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n,
\]

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).
- **Embedding** For a new symbol \( x \), map

\[
\leftarrow A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n
\]

\[
\mapsto x \leftarrow A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n, \text{not } x
\]

- Another example \( \Pi = \{p \leftarrow \text{not } q, \ q \leftarrow \text{not } p\} \)
  versus \( \Pi' = \Pi \cup \{\leftarrow p\} \) and \( \Pi'' = \Pi \cup \{\leftarrow \text{not } p\} \)
Integrity Constraints

- **Idea** Eliminate unwanted solution candidates
- **Syntax** An integrity constraint is of the form

  \[ \leftarrow A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n, \]

  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).
- **Embedding** For a new symbol \( x \), map

  \[ \leftarrow A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n \]

  \[ x \leftarrow A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n, \text{not } x \]

- Another example \( \Pi = \{ p \leftarrow \text{not } q, \ q \leftarrow \text{not } p \} \)
  
  versus \( \Pi' = \Pi \cup \{ \leftarrow p \} \) and \( \Pi'' = \Pi \cup \{ \leftarrow \text{not } p \} \)
Language Extensions: Overview

16 Motivation

17 Integrity Constraints

18 Choice Rules

19 Cardinality Constraints

20 Weight Constraints

21 Optimization statements

22 Conditional literals

23 smodels format
Choice rules

- **Idea**: Choices over subsets
- **Syntax**: \[ \{A_1, \ldots, A_m\} \leftarrow A_{m+1}, \ldots, A_n, \text{not } A_{n+1}, \ldots, \text{not } A_o, \]

- **Informal meaning**: If the body is satisfied in an answer set, then any subset of \( \{A_1, \ldots, A_m\} \) can be included in the answer set.

- **Example 1**: \( \{\text{color}(X,C) : \text{col}(C)\} 1 \leftarrow \text{node}(X) \).

- **Another Example**: Program \( \Pi = \{ \{a\} \leftarrow b, b \leftarrow \} \) has two answer sets: \( \{b\} \) and \( \{a, b\} \).
Choice rules

- **Idea** Choices over subsets
- **Syntax**

  \[
  \{A_1, \ldots, A_m\} \leftarrow A_{m+1}, \ldots, A_n, \text{not } A_{n+1}, \ldots, \text{not } A_o,
  \]

- **Informal meaning** If the body is satisfied in an answer set, then any subset of \(\{A_1, \ldots, A_m\}\) can be included in the answer set.
- **Example**

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

- **Another Example** Program \(\Pi = \{\{a\} \leftarrow b, b \leftarrow\}\) has two answer sets: \(\{b\}\) and \(\{a, b\}\).
Choice rules

- **Idea** Choices over subsets
- **Syntax**

\[
\{A_1, \ldots, A_m\} \leftarrow A_{m+1}, \ldots, A_n, \text{not} \ A_{n+1}, \ldots, \text{not} \ A_o,
\]

- **Informal meaning** If the body is satisfied in an answer set, then any subset of \(\{A_1, \ldots, A_m\}\) can be included in the answer set.
- **Example** 1 \(\{\text{color}(X,C) : \text{col}(C)\} \leftarrow \text{node}(X).\)
- **Another Example** Program \(\Pi = \{ \{a\} \leftarrow b, \ b \leftarrow \}\) has two answer sets: \(\{b\}\) and \(\{a, b\}\)
Embedding in normal logic programs

- A choice rule of form

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

can be translated into $2m + 1$ rules

$$A \leftarrow A_{m+1}, \ldots, A_n, \text{not } A_{n+1}, \ldots, \text{not } A_o$$

$$A_1 \leftarrow A, \text{not } A_1 \quad \ldots \quad A_m \leftarrow A, \text{not } A_m$$

$$\overline{A}_1 \leftarrow \text{not } A_1 \quad \ldots \quad \overline{A}_m \leftarrow \text{not } A_m$$

by introducing new atoms $A, \overline{A}_1, \ldots, \overline{A}_m$
Language Extensions: Overview

16 Motivation

17 Integrity Constraints

18 Choice Rules

19 Cardinality Constraints

20 Weight Constraints

21 Optimization statements

22 Conditional literals

23 smodels format
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 \( l \leq |\{ A_1, \ldots, A_m \} \cap X| \leq u \)

- **Example 2** \( \{ \text{hd}(a), \ldots, \text{hd}(m) \} \) 4

- **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 \( A_1, \ldots, A_m \)
Cardinality constraints

- **Syntax** A (positive) cardinality constraint is of the form

  \[ l \{A_1, \ldots, A_m\} \leq u \]

- **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 \( l \leq |\{A_1, \ldots, A_m\} \cap X| \leq u \)

- **Example 2** \( \{\text{hd}(a), \ldots, \text{hd}(m)\} \leq 4 \)

- **Conditions** \[ l \{A_1 : B_1, \ldots, A_m : B_m\} \leq u \]

  where \( B_1, \ldots, B_m \) are used for restricting instantiations of variables occurring in \( A_1, \ldots, A_m \)
Cardinality rules

- **Syntax**
  
  \[ A_0 \leftarrow l \{A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n\} \]

- **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** Program \( \Pi = \{ a \leftarrow l\{b, c\}, \ b \leftarrow \} \) has answer set \( \{ a, b \} \)
Embedding in normal logic programs

- Replace each cardinality rule

\[ A_0 \leftarrow l \{A_1, \ldots, A_m\} \]

by

\[ A_0 \leftarrow cc(A_1, l) \]

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

\[
\begin{align*}
cc(A_i, j + 1) & \leftarrow cc(A_{i+1}, j), A_i \\
cc(A_i, j) & \leftarrow cc(A_{i+1}, j) \\
cc(A_{m+1}, 0) & \leftarrow
\end{align*}
\]
Embedding in normal logic programs

- Replace each cardinality rule

\[ A_0 \leftarrow l \{A_1, \ldots, A_m\} \quad \text{by} \quad A_0 \leftarrow cc(A_1, l) \]

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

\[
\begin{align*}
cc(A_i, j+1) & \leftarrow cc(A_{i+1}, j), A_i \\
cc(A_i, j) & \leftarrow cc(A_{i+1}, j) \\
cc(A_{m+1}, 0) & \leftarrow \\
\end{align*}
\]
Cardinality Constraints

... and vice versa

- A normal rule

\[
A_0 \leftarrow A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n,
\]

can be represented by the cardinality rule

\[
A_0 \leftarrow n \{A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n\}
\]
Cardinality rules with upper bounds

A rule of the form

\[ A_0 \leftarrow l \{ A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n \} \ u \]

stands for

\[ A_0 \leftarrow B, \text{not } C \]
\[ B \leftarrow l \{ A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n \} \]
\[ C \leftarrow u+1 \{ A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n \} \]
Cardinality constraints as heads

A rule of the form

\[ l \{A_1, \ldots, A_m\} \ u \leftarrow A_{m+1}, \ldots, A_n, \text{not } A_{n+1}, \ldots, \text{not } A_o, \]

stands for

\[ B \leftarrow A_{m+1}, \ldots, A_n, \text{not } A_{n+1}, \ldots, \text{not } A_o \]

\[ \{A_1, \ldots, A_m\} \leftarrow B \]

\[ C \leftarrow l \{A_1, \ldots, A_m\} \ u \leftarrow B, \text{not } C \]
Full-fledged cardinality rules

A rule of the form

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

stands for \(0 \leq i \leq n\)

\[ B_i \leftarrow l_i \ S_i \]

\[ C_i \leftarrow u_i + 1 \ S_i \]

\[ A \leftarrow B_1, \ldots, B_n, \text{not } C_1, \ldots, \text{not } C_n \]

\[ \leftarrow A, \text{not } B_0 \]

\[ \leftarrow A, C_0 \]

\[ S_0 \cap A \leftarrow A \]

where \(A\) is the underlying alphabet
Cardinality Constraints

Full-fledged cardinality rules

- A rule of the form

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

stands for \( 0 \leq i \leq n \)

\[ B_i \leftarrow l_i S_i \]
\[ C_i \leftarrow u_i + 1 S_i \]
\[ A \leftarrow B_1, \ldots, B_n, not C_1, \ldots, not C_n \]
\[ \leftarrow A, not B_0 \]
\[ \leftarrow A, C_0 \]
\[ S_0 \cap A \leftarrow A \]

where \( A \) is the underlying alphabet
Language Extensions: Overview

16 Motivation

17 Integrity Constraints

18 Choice Rules

19 Cardinality Constraints

20 Weight Constraints

21 Optimization statements

22 Conditional literals

23 smodels format
Weight constraints

- **Syntax**: $l \left[ A_1 = w_1, \ldots, A_m = w_m, \
    \text{not } A_{m+1} = w_{m+1}, \ldots, \text{not } A_n = w_n \right] u$

- **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
Language Extensions: Overview

16 Motivation
17 Integrity Constraints
18 Choice Rules
19 Cardinality Constraints
20 Weight Constraints
21 Optimization statements
22 Conditional literals
23 smodels format
Optimization statements

- **Idea** Compute optimal answer sets by minimizing (or maximizing) a weighted sum of given elements
- **Syntax**
  - `#minimize [A_1 = w_1, \ldots, A_m = w_m, not A_{m+1} = w_{m+1}, \ldots, not A_n = w_n]`
  - `#maximize [A_1 = w_1, \ldots, A_m = w_m, not A_{m+1} = w_{m+1}, \ldots, not A_n = w_n]`
- **Example**
  - `#minimize [hd(a)=30, \ldots, hd(m)=50]`
  - `#minimize [road(X,Y) : length(X,Y,L) = L]`

- Multi-criteria optimization can be accomplished by adding priority levels to weighted literals, that is, by replacing $L_i = w_i$ by $L_i = w_i \oplus P_i$
Optimization statements

■ Idea Compute optimal answer sets by minimizing (or maximizing) a weighted sum of given elements

■ Syntax

  ■ \#minimize $[A_1 = w_1, \ldots, A_m = w_m,$
    \hspace{1cm} \text{not } A_{m+1} = w_{m+1}, \ldots, \text{not } A_n = w_n]$

  ■ \#maximize $[A_1 = w_1, \ldots, A_m = w_m,$
    \hspace{1cm} \text{not } A_{m+1} = w_{m+1}, \ldots, \text{not } A_n = w_n]$

■ Example

  ■ \#minimize $[\text{hd}(a)=30, \ldots, \text{hd}(m)=50]$
  ■ \#minimize $[\text{road}(X,Y) : \text{length}(X,Y,L) = L]$

■ Multi-criteria optimization can be accomplished by adding priority levels to weighted literals, that is, by replacing $L_i = w_i$ by $L_i = w_i @ P_i$
Language Extensions: Overview

16 Motivation
17 Integrity Constraints
18 Choice Rules
19 Cardinality Constraints
20 Weight Constraints
21 Optimization statements
22 Conditional literals
23 smodels format
Conditional literals

- **Idea** Encode the contents of a (multi-)set without enumerating its elements
- **Syntax** \( A_0 : A_1 : \ldots : A_m : \text{not} \ A_{m+1} : \ldots : \text{not} \ A_n \)
- **Informal meaning** List all ground instances of \( A_0 \) such that corresponding instances of \( A_1, \ldots, A_m, \text{not} \ A_{m+1}, \ldots, \text{not} \ A_n \) are true
- **Example** Given ‘\( p(1). \ p(2). \ p(3). \ q(2). \)’ the choice
  \[ \{r(X) : p(X) : \text{not} \ q(X)\}. \]
  is instantiated to
  \[ \{r(1), r(3)\}. \]
Language Extensions: Overview

16  Motivation
17  Integrity Constraints
18  Choice Rules
19  Cardinality Constraints
20  Weight Constraints
21  Optimization statements
22  Conditional literals
23  smodels format
smodels format

- Logic programs in *smodels* format consist of:
  - normal rules
  - choice rules
  - cardinality rules
  - weight rules
  - optimization statements

- Such a format is obtained by grounders *lpars* and *gringo*
Conflict-Driven Answer Set Solving: Overview

24 Motivation

25 Boolean Constraints

26 Nogoods from Logic Programs
   - Nogoods from program completion
   - Nogoods from loop formulas

27 Conflict-Driven Nogood Learning
   - CDNL-ASP Algorithm
   - Nogood Propagation
   - Conflict Analysis
Conflict-Driven Answer Set Solving: Overview

24 Motivation

25 Boolean Constraints

26 Nogoods from Logic Programs

- Nogoods from program completion
- Nogoods from loop formulas

27 Conflict-Driven Nogood Learning

- CDNL-ASP Algorithm
- Nogood Propagation
- Conflict Analysis
Motivation

- **Goal**: Approach to computing answer sets of logic programs, based on concepts from
  - Constraint Processing (CP) and
  - Satisfiability Checking (SAT)

- **Idea**: View inferences in 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 CP and SAT
  - Highly competitive implementation
Conflict-Driven Answer Set Solving: Overview

Motivation

Boolean Constraints

Nogoods from Logic Programs
- Nogoods from program completion
- Nogoods from loop formulas

Conflict-Driven Nogood Learning
- CDNL-ASP Algorithm
- Nogood Propagation
- Conflict Analysis
Assignments

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

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

of signed literals $\sigma_i$ of form $T_p$ or $F_p$ for $p \in \text{dom}(A)$ and $1 \leq i \leq n$.

$T_p$ expresses that $p$ is true and $F_p$ that it is false.

- The complement, $\bar{\sigma}$, of a literal $\sigma$ is defined as $\overline{T_p} = F_p$ and $\overline{F_p} = T_p$.

- $A \circ B$ 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

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

- An assignment $A$ over $\text{dom}(A) = \text{atom}(\Pi) \cup \text{body}(\Pi)$ is a sequence $(\sigma_1, \ldots, \sigma_n)$ of signed literals $\sigma_i$ of form $T_p$ or $F_p$ for $p \in \text{dom}(A)$ and $1 \leq i \leq n$.
  - $T_p$ expresses that $p$ is true and $F_p$ that it is false.
- The complement, $\overline{\sigma}$, of a literal $\sigma$ is defined as $\overline{T_p} = F_p$ and $\overline{F_p} = T_p$.
- $A \circ B$ 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

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

- An assignment $A$ over $\text{dom}(A) = \text{atom}(\Pi) \cup \text{body}(\Pi)$ is a sequence $(\sigma_1, \ldots, \sigma_n)$ of signed literals $\sigma_i$ of form $T_p$ or $F_p$ for $p \in \text{dom}(A)$ and $1 \leq i \leq n$. $T_p$ expresses that $p$ is true and $F_p$ that it is false.

- The complement, $\overline{\sigma}$, of a literal $\sigma$ is defined as $\overline{T_p} = F_p$ and $\overline{F_p} = T_p$.

- $A \circ B$ 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

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

- An assignment $A$ over $\text{dom}(A) = \text{atom}(\Pi) \cup \text{body}(\Pi)$ is a sequence $$(\sigma_1, \ldots, \sigma_n)$$

  of signed literals $\sigma_i$ of form $T_p$ or $F_p$ for $p \in \text{dom}(A)$ and $1 \leq i \leq n$.  
  - $T_p$ expresses that $p$ is true and $F_p$ that it is false.

- The complement, $\bar{\sigma}$, of a literal $\sigma$ is defined as $\bar{T}p = Fp$ and $\bar{F}p = Tp$.

- $A \circ B$ 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

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

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

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

of signed literals $\sigma_i$ of form $T_p$ or $F_p$ for $p \in \text{dom}(A)$ and $1 \leq i \leq n$. $T_p$ expresses that $p$ is true and $F_p$ that it is false.

- The complement, $\overline{\sigma}$, of a literal $\sigma$ is defined as $\overline{T_p} = F_p$ and $\overline{F_p} = T_p$.

- $A \circ B$ 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

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

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

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

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

- $T\ p$ expresses that $p$ is true and $F\ p$ that it is false.

- The complement, $\overline{\sigma}$, of a literal $\sigma$ is defined as $\overline{T\ p} = F\ p$ and $\overline{F\ p} = T\ p$.

- $A \circ B$ 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

$$A^T = \{ p \in \text{dom}(A) \mid T\ p \in A \} \quad \text{and} \quad A^F = \{ p \in \text{dom}(A) \mid F\ p \in A \} .$$
Nogoods, Solutions, and Unit Propagation

- A **nogood** is a set \( \{\sigma_1, \ldots, \sigma_n\} \) of signed literals, expressing a constraint violated by any assignment containing \( \sigma_1, \ldots, \sigma_n \).

- An assignment \( A \) such that \( A^T \cup A^F = \text{dom}(A) \) and \( A^T \cap A^F = \emptyset \) is a solution for a set \( \Delta \) of nogoods, if \( \delta \not\subseteq A \) for all \( \delta \in \Delta \).

- For a nogood \( \delta \), a literal \( \sigma \in \delta \), and an assignment \( A \), we say that \( \overline{\sigma} \) is unit-resulting for \( \delta \) wrt \( A \), if
  1. \( \delta \setminus A = \{\sigma\} \) and
  2. \( \overline{\sigma} \not\in A \).

- For a set \( \Delta \) 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 \( \Delta \).
A nogood is a set \( \{\sigma_1, \ldots, \sigma_n\} \) of signed literals, expressing a constraint violated by any assignment containing \( \sigma_1, \ldots, \sigma_n \).

An assignment \( A \) such that \( A^T \cup A^F = \text{dom}(A) \) and \( A^T \cap A^F = \emptyset \) is a solution for a set \( \Delta \) of nogoods, if \( \delta \not\subseteq A \) for all \( \delta \in \Delta \).

For a nogood \( \delta \), a literal \( \sigma \in \delta \), and an assignment \( A \), we say that \( \overline{\sigma} \) is unit-resulting for \( \delta \) wrt \( A \), if
\[
1 \quad \delta \setminus A = \{\sigma\} \quad \text{and} \\
2 \quad \overline{\sigma} \notin A.
\]

For a set \( \Delta \) 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 \( \Delta \).
Nogoods, Solutions, and Unit Propagation

- A nogood is a set \( \{\sigma_1, \ldots, \sigma_n\} \) of signed literals, expressing a constraint violated by any assignment containing \( \sigma_1, \ldots, \sigma_n \).

- An assignment \( A \) such that \( A^T \cup A^F = \text{dom}(A) \) and \( A^T \cap A^F = \emptyset \) is a solution for a set \( \Delta \) of nogoods, if \( \delta \not\subseteq A \) for all \( \delta \in \Delta \).

- For a nogood \( \delta \), a literal \( \sigma \in \delta \), and an assignment \( A \), we say that \( \sigma \) is unit-resulting for \( \delta \) wrt \( A \), if
  1. \( \delta \\setminus A = \{\sigma\} \) and
  2. \( \overline{\sigma} \not\in A \).

- For a set \( \Delta \) 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 \( \Delta \).
Nogoods, Solutions, and Unit Propagation

- A **nogood** is a set \(\{\sigma_1, \ldots, \sigma_n\}\) of signed literals, expressing a constraint violated by any assignment containing \(\sigma_1, \ldots, \sigma_n\).

- An assignment \(A\) such that \(A^T \cup A^F = \text{dom}(A)\) and \(A^T \cap A^F = \emptyset\) is a solution for a set \(\Delta\) of nogoods, if \(\delta \not\subseteq A\) for all \(\delta \in \Delta\).

- For a nogood \(\delta\), a literal \(\sigma \in \delta\), and an assignment \(A\), we say that \(\overline{\sigma}\) is unit-resulting for \(\delta\) wrt \(A\), if
  1. \(\delta \setminus A = \{\sigma\}\) and
  2. \(\overline{\sigma} \not\in A\).

- For a set \(\Delta\) 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 \(\Delta\).
Conflict-Driven Answer Set Solving:
Overview

24 Motivation

25 Boolean Constraints

26 Nogoods from Logic Programs
   - Nogoods from program completion
   - Nogoods from loop formulas

27 Conflict-Driven Nogood Learning
   - CDNL-ASP Algorithm
   - Nogood Propagation
   - Conflict Analysis
Nogoods from logic programs
via program completion

The completion of a logic program $\Pi$ can be defined as follows:

$$\begin{align*}
\{ p_\beta & \leftrightarrow p_1 \land \cdots \land p_m \land \neg p_{m+1} \land \cdots \land \neg p_n \mid \\
& \beta \in body(\Pi), \beta = \{p_1, \ldots, p_m, not \ p_{m+1}, \ldots, not \ p_n\}\} \\
\cup \{ p & \leftrightarrow p_{\beta_1} \lor \cdots \lor p_{\beta_k} \mid \\
& p \in atom(\Pi), body(p) = \{\beta_1, \ldots, \beta_k\}\}
\end{align*}$$

where $body(p) = \{body(r) \mid r \in \Pi, head(r) = p\}$. 
Let $\beta = \{p_1, \ldots, p_m, \neg p_{m+1}, \ldots, \neg p_n\}$ be a body.

The equivalence

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

can be decomposed into two implications.

1. We get

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

which is equivalent to the conjunction of

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

This set of clauses expresses the following set of nogoods:

$$\Delta(\beta) = \{ \{T_\beta, F_{p_1}\}, \ldots, \{T_\beta, F_{p_m}\}, \{T_\beta, T_{p_{m+1}}\}, \ldots, \{T_\beta, T_{p_n}\} \}$$
Nogoods from logic programs via program completion

Let $\beta = \{p_1, \ldots, p_m, \neg p_{m+1}, \ldots, \neg p_n\}$ be a body.

The equivalence

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

can be decomposed into two implications.

1. We get

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

which is equivalent to the conjunction of

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

This set of clauses expresses the following set of nogoods:

$$\Delta(\beta) = \{ \{T_\beta, F_p_1\}, \ldots, \{T_\beta, F_p_m\}, \{T_\beta, T_p_{m+1}\}, \ldots, \{T_\beta, T_p_n\}\}$$
Let $\beta = \{p_1, \ldots, p_m, \neg p_{m+1}, \ldots, \neg p_n\}$ be a body.

The equivalence

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

can be decomposed into two implications.

The converse of the previous implication, viz.

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

gives rise to the nogood

$$\delta(\beta) = \{F_\beta, T_{p_1}, \ldots, T_{p_m}, F_{p_{m+1}}, \ldots, F_{p_n}\}$$

Intuitively, $\delta(\beta)$ is a constraint enforcing the truth of body $\beta$, or the falsity of a contained literal.
Nogoods from logic programs
via program completion

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

The equivalence

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

can be decomposed into two implications.

1. The converse of the previous implication, viz.

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

gives rise to the nogood

$$\delta(\beta) = \{Fp_\beta, Tp_1, \ldots, Tp_m, Fp_{m+1}, \ldots, Fp_n\}$$

Intuitively, $\delta(\beta)$ is a constraint enforcing the truth of body $\beta$, or the falsity of a contained literal.
Nogoods from logic programs via program completion

Proceeding analogously with the atom-based equivalences, viz.

\[ p \leftrightarrow p_{\beta_1} \lor \cdots \lor p_{\beta_k} \]

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

\[ \Delta(p) = \left\{ \{Fp, T_{\beta_1}\}, \ldots, \{Fp, T_{\beta_k}\} \right\} \quad \text{and} \quad \delta(p) = \{Tp, F_{\beta_1}, \ldots, F_{\beta_k}\}. \]
For an atom $p$ where $\text{body}(p) = \{\beta_1, \ldots, \beta_k\}$, recall that
\[
\delta(p) = \{T_p, F_{\beta_1}, \ldots, F_{\beta_k}\}
\]
\[
\Delta(p) = \{\{F_p, T_{\beta_1}\}, \ldots, \{F_p, T_{\beta_k}\}\}.
\]

For example, for atom $x$ with $\text{body}(x) = \{\{y\}, \{\text{not } z\}\}$, we obtain
\[
\begin{align*}
x & \leftarrow y \\
x & \leftarrow \text{not } z
\end{align*}
\]
\[
\delta(x) = \{T_x, F\{y\}, F\{\text{not } z\}\}
\]
\[
\Delta(x) = \{\{F_x, T\{y\}\}, \{F_x, T\{\text{not } z\}\}\}
\]

For nogood $\delta(x) = \{T_x, F\{y\}, F\{\text{not } z\}\}$, the signed literal $F_x$ is unit-resulting wrt assignment $(F\{y\}, F\{\text{not } z\})$ and $T\{\text{not } z\}$ is unit-resulting wrt assignment $(T_x, F\{y\})$.
Nogoods from logic programs
atom-oriented nogoods

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

$$\delta(p) = \{T_p, F_{\beta_1}, \ldots, F_{\beta_k}\}$$
$$\Delta(p) = \{\{F_p, T_{\beta_1}\}, \ldots, \{F_p, T_{\beta_k}\}\}.$$  

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

$$\begin{align*}
x &\leftarrow y \\
x &\leftarrow \text{not } z
\end{align*}$$

$$\begin{align*}
\delta(x) &= \{T_x, F\{y\}, F\{\text{not } z\}\} \\
\Delta(x) &= \{\{F_x, T\{y\}\}, \{F_x, T\{\text{not } z\}\}\}
\end{align*}$$

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

$$F_x$$ is unit-resulting wrt assignment $(F\{y\}, F\{\text{not } z\})$ and

$$T\{\text{not } z\}$$ is unit-resulting wrt assignment $(T_x, F\{y\})$.
Nogoods from logic programs
atom-oriented nogoods

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

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

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

$$\begin{align*}
x & \leftarrow y \\
x & \leftarrow \text{not } z
\end{align*}$$

$$\begin{align*}
\delta(x) & = \{Tx, F\{y\}, F\{\text{not } z\}\} \\
\Delta(x) & = \{\{Fx, T\{y\}\}, \{Fx, T\{\text{not } z\}\}\}
\end{align*}$$

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

$$Fx$$

is unit-resulting wrt assignment $(F\{y\}, F\{\text{not } z\})$ and

$$T\{\text{not } z\}$$

is unit-resulting wrt assignment $(Tx, F\{y\})$. 
For an atom $p$ where $\text{body}(p) = \{\beta_1, \ldots, \beta_k\}$, recall that

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

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

$$
\begin{align*}
\delta(x) &= \{Tx, F\{y\}, F\{\text{not } z\}\} \\
\Delta(x) &= \{\{Fx, T\{y\}\}, \{Fx, T\{\text{not } z\}\}\}
\end{align*}
$$

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

- $Fx$ is unit-resulting wrt assignment $(F\{y\}, F\{\text{not } z\})$ and
- $T\{\text{not } z\}$ is unit-resulting wrt assignment $(Tx, F\{y\})$
For an atom $p$ where $\text{body}(p) = \{\beta_1, \ldots, \beta_k\}$, recall that
\[
\delta(p) = \{Tp, F\beta_1, \ldots, F\beta_k\}
\]
\[
\Delta(p) = \{\{Fp, T\beta_1\}, \ldots, \{Fp, T\beta_k\}\}.
\]

For example, for atom $x$ with $\text{body}(x) = \{\{y\}, \{\text{not } z\}\}$, we obtain
\[
\begin{align*}
x & \leftarrow y & \delta(x) & = \{Tx, F\{y\}, F\{\text{not } z\}\} \\
x & \leftarrow \text{not } z & \Delta(x) & = \{\{Fx, T\{y\}\}, \{Fx, T\{\text{not } z\}\}\}
\end{align*}
\]

For nogood $\delta(x) = \{Tx, F\{y\}, F\{\text{not } z\}\}$, the signed literal
- $Fx$ is unit-resulting wrt assignment $(F\{y\}, F\{\text{not } z\})$ and
- $T\{\text{not } z\}$ is unit-resulting wrt assignment $(Tx, F\{y\})$.
Nogoods from logic programs

atom-oriented nogoods

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

\[
\delta(p) = \{ Tp, F\beta_1, \ldots, F\beta_k \}
\]

\[
\Delta(p) = \{ \{ Fp, T\beta_1 \}, \ldots, \{ Fp, T\beta_k \} \}.
\]

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

\[
x \leftarrow y \quad \delta(x) = \{ Tx, F\{ y \}, F\{ \text{not } z \} \}
\]

\[
x \leftarrow \text{not } z \quad \Delta(x) = \{ \{ Fx, T\{ y \} \}, \{ Fx, T\{ \text{not } z \} \} \}
\]

For nogood \( \delta(x) = \{ Tx, F\{ y \}, F\{ \text{not } z \} \} \), the signed literal

- \( Fx \) is unit-resulting wrt assignment \((F\{ y \}, F\{ \text{not } z \})\) and
- \( T\{ \text{not } z \} \) is unit-resulting wrt assignment \((Tx, F\{ y \})\)
Nogoods from logic programs
atom-oriented nogoods

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

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

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

$x \leftarrow y$
$x \leftarrow \text{not } z$

$$\delta(x) = \{Tx, F\{y\}, F\{\text{not } z\}\}$$
$$\Delta(x) = \{\{Fx, T\{y\}\}, \{Fx, T\{\text{not } z\}\}\}.$$  

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

- $Fx$ is unit-resulting wrt assignment $(F\{y\}, F\{\text{not } z\})$ and
- $T\{\text{not } z\}$ is unit-resulting wrt assignment $(Tx, F\{y\})$
For an atom $p$ where $\text{body}(p) = \{\beta_1, \ldots, \beta_k\}$, recall that

$$\delta(p) = \{Tp, F\beta_1, \ldots, F\beta_k\}$$

$$\Delta(p) = \{\{Fp, T\beta_1\}, \ldots, \{Fp, T\beta_k\}\}.$$

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

$$x \leftarrow y \quad \delta(x) = \{Tx, F\{y\}, F\{\text{not } z\}\}$$

$$x \leftarrow \text{not } z \quad \Delta(x) = \{\{Fx, T\{y\}\}, \{Fx, T\{\text{not } z\}\}\}$$

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

- $Fx$ is unit-resulting wrt assignment $(F\{y\}, F\{\text{not } z\})$ and
- $T\{\text{not } z\}$ is unit-resulting wrt assignment $(Tx, F\{y\})$
Nogoods from logic programs

atom-oriented nogoods

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

\[
\delta(p) = \{T\beta_1, \ldots, F\beta_k\}
\]

\[
\Delta(p) = \{\{Fp, T\beta_1\}, \ldots, \{Fp, T\beta_k\}\}.
\]

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

\[
\begin{align*}
x & \leftarrow y \\
x & \leftarrow \text{not } z
\end{align*}
\]

\[
\delta(x) = \{T\beta, F\{y\}, F\{\text{not } z\}\}
\]

\[
\Delta(x) = \{\{Fp, T\{y\}\}, \{Fp, T\{\text{not } z\}\}\}.
\]

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

- \( F\beta \) is unit-resulting wrt assignment \( (F\{y\}, F\{\text{not } z\}) \) and
- \( T\{\text{not } z\} \) is unit-resulting wrt assignment \( (T\beta, F\{y\}) \)
Nogoods from logic programs
atom-oriented nogoods

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

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

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

$$x \leftarrow y \quad x \leftarrow \text{not } z$$

$$\delta(x) = \{Tx, F\{y\}, F\{\text{not } z\}\}$$
$$\Delta(x) = \{\{Fx, T\{y\}\}, \{Fx, T\{\text{not } z\}\}\}.$$  

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

- $Fx$ is unit-resulting wrt assignment $(F\{y\}, F\{\text{not } z\})$ and
- $T\{\text{not } z\}$ is unit-resulting wrt assignment $(Tx, F\{y\})$.
Nogoods from logic programs
atom-oriented nogoods

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

$$\delta(p) = \{T_p, F_{\beta_1}, \ldots, F_{\beta_k}\}$$
$$\Delta(p) = \{\{F_p, T_{\beta_1}\}, \ldots, \{F_p, T_{\beta_k}\}\}.$$ 

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

$$x \leftarrow y \quad \delta(x) = \{T_x, F\{y\}, F\{\text{not } z\}\}$$
$$x \leftarrow \text{not } z \quad \Delta(x) = \{\{F_x, T\{y\}\}, \{F_x, T\{\text{not } z\}\}\}.$$ 

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

- $F_x$ is unit-resulting wrt assignment $(F\{y\}, F\{\text{not } z\})$ and
- $T\{\text{not } z\}$ is unit-resulting wrt assignment $(T_x, F\{y\})$
atom-oriented nogoods

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

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

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

$$\delta(x) = \{ Tx, F\{y\}, F\{not\ z\}\}$$
$$\Delta(x) = \{ \{ Fx, T\{y\}\}, \{ Fx, T\{not\ z\}\} \}.$$

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

- $Fx$ is unit-resulting wrt assignment $(F\{y\}, F\{not\ z\})$ and
- $T\{not\ z\}$ is unit-resulting wrt assignment $(Tx, F\{y\})$
Nogoods from logic programs
atom-oriented nogoods

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

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

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

\[
\begin{align*}
x & \leftarrow y \\
x & \leftarrow \text{not } z
\end{align*}
\]

$$\delta(x) = \{T_x, F\{y\}, F\{\text{not } z\}\}$$
$$\Delta(x) = \{\{F_x, T\{y\}\}, \{F_x, T\{\text{not } z\}\}\}.$$
Nogoods from logic programs
atom-oriented nogoods

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

\[
\delta(p) = \{T p, F\beta_1, \ldots, F\beta_k\}
\]
\[
\Delta(p) = \{\{F p, T\beta_1\}, \ldots, \{F p, T\beta_k\}\}.
\]

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

\[
\begin{align*}
x & \leftarrow y \\
x & \leftarrow \text{not } z
\end{align*}
\]
\[
\delta(x) = \{T x, F\{y\}, F\{\text{not } z\}\}
\]
\[
\Delta(x) = \{\{F x, T\{y\}\}, \{F x, T\{\text{not } z\}\}\}.
\]

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

- \( F x \) is unit-resulting wrt assignment \((F\{y\}, F\{\text{not } z\})\) and

- \( T\{\text{not } z\} \) is unit-resulting wrt assignment \((T x, F\{y\})\).
Nogoods from logic programs

body-oriented nogoods

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

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

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

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

$$\delta(\{x, not \ y\}) = \{F \{x, not \ y\}, T x, F y\}$$

$$\Delta(\{x, not \ y\}) = \{\{T \{x, not \ y\}, F x\}, \{T \{x, not \ y\}, T y\}\}.$$

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

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

body-oriented nogoods

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

$$\delta(\beta) = \{F\beta, Tp_1, \ldots, Tp_m, Fp_{m+1}, \ldots, Fp_n\}$$

$$\Delta(\beta) = \{\{T\beta, Fp_1\}, \ldots, \{T\beta, Fp_m\}, \{T\beta, Tp_{m+1}\}, \ldots, \{T\beta, Tp_n\}\}.$$  

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

$$\delta(\{x, \text{not } y\}) = \{F\{x, \text{not } y\}, Tx, Fy\}$$

$$\Delta(\{x, \text{not } y\}) = \{\{Tx, Fy\}, \{Tx, \text{not } y\}, Fx\}. $$

For nogood $\delta(\{x, \text{not } y\}) = \{F\{x, \text{not } y\}, Tx, Fy\}$, the signed literal

- $T\{x, \text{not } y\}$ is unit-resulting wrt assignment $(Tx, Fy)$ and
- $Ty$ is unit-resulting wrt assignment $(F\{x, \text{not } y\}, Tx)$. 
Nogoods from logic programs

body-oriented nogoods

For a body \( \beta = \{ p_1, \ldots, p_m, \text{not } p_{m+1}, \ldots, \text{not } p_n \} \), recall that

\[
\delta(\beta) = \{ F\beta, Tp_1, \ldots, Tp_m, Fp_{m+1}, \ldots, Fp_n \}
\]

\[
\Delta(\beta) = \{ \{ T\beta, Fp_1 \}, \ldots, \{ T\beta, Fp_m \}, \{ T\beta, Tp_{m+1} \}, \ldots, \{ T\beta, Tp_n \} \} .
\]

For example, for body \( \{ x, \text{not } y \} \), we obtain

\[
\delta(\{ x, \text{not } y \}) = \{ F\{ x, \text{not } y \}, Tx, Fy \}
\]

\[
\Delta(\{ x, \text{not } y \}) = \{ \{ Tx, \text{not } y \}, Fx \}, \{ Tx, \text{not } y \}, Ty \} .
\]

For nogood \( \delta(\{ x, \text{not } y \}) = \{ F\{ x, \text{not } y \}, Tx, Fy \} \), the signed literal

- \( T\{ x, \text{not } y \} \) is unit-resulting wrt assignment \((Tx, Fy)\) and
- \( Ty \) is unit-resulting wrt assignment \((Fx, Tx)\).
Nogoods from logic programs
body-oriented nogoods

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

$$\delta(\beta) = \{F\beta, Tp_1, \ldots, Tp_m, Fp_{m+1}, \ldots, Fp_n\}$$
$$\Delta(\beta) = \{\{T\beta, Fp_1\}, \ldots, \{T\beta, Fp_m\}, \{T\beta, Tp_{m+1}\}, \ldots, \{T\beta, Tp_n\}\}.$$

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

$$\delta(\{x, \text{not } y\}) = \{F\{x, \text{not } y\}, Tx, Fy\}$$
$$\Delta(\{x, \text{not } y\}) = \{\{T\{x, \text{not } y\},Fx\}, \{T\{x, \text{not } y\}, Ty\}\}$$

For nogood $\delta(\{x, \text{not } y\}) = \{F\{x, \text{not } y\}, Tx, Fy\}$, the signed literal

- $T\{x, \text{not } y\}$ is unit-resulting wrt assignment $(Tx, Fy)$ and
- $Ty$ is unit-resulting wrt assignment $(F\{x, \text{not } y\}, Tx)$. 
Nogoods from logic programs

body-oriented nogoods

For a body \( \beta = \{p_1, \ldots, p_m, \neg p_{m+1}, \ldots, \neg p_n\} \), recall that

\[
\delta(\beta) = \{F_\beta, T_{p_1}, \ldots, T_{p_m}, F_{p_{m+1}}, \ldots, F_{p_n}\}
\]

\[
\Delta(\beta) = \{\{T_\beta, F_{p_1}\}, \ldots, \{T_\beta, F_{p_m}\}, \{T_\beta, T_{p_{m+1}}\}, \ldots, \{T_\beta, T_{p_n}\}\}.
\]

For example, for body \( \{x, \neg y\} \), we obtain

\[
\delta(\{x, \neg y\}) = \{F\{x, \neg y\}, T_x, F_y\}
\]

\[
\Delta(\{x, \neg y\}) = \{\{T\{x, \neg y\}, F_x\}, \{T\{x, \neg y\}, T_y\}\}.
\]

For nogood \( \delta(\{x, \neg y\}) = \{F\{x, \neg y\}, T_x, F_y\} \), the signed literal

- \( T\{x, \neg y\} \) is unit-resulting wrt assignment \( (T_x, F_y) \) and
- \( T_y \) is unit-resulting wrt assignment \( (F\{x, \neg y\}, T_x) \).
Characterization of answer sets
for tight logic programs

Let $\Pi$ be a logic program and

$$\Delta_\Pi = \{\delta(p) \mid p \in \text{atom}(\Pi)\} \cup \{\delta \in \Delta(p) \mid p \in \text{atom}(\Pi)\}$$
$$\cup \{\delta(\beta) \mid \beta \in \text{body}(\Pi)\} \cup \{\delta \in \Delta(\beta) \mid \beta \in \text{body}(\Pi)\}.$$ 

**Theorem**

Let $\Pi$ be a tight logic program. Then,

$X \subseteq \text{atom}(\Pi)$ is an answer set of $\Pi$ iff

$X = A^T \cap \text{atom}(\Pi)$ for a (unique) solution $A$ for $\Delta_\Pi$. 
Characterization of answer sets
for tight logic programs

Let \( \Pi \) be a logic program and

\[
\Delta_\Pi = \{ \delta(p) \mid p \in \text{atom}(\Pi) \} \cup \{ \delta \in \Delta(p) \mid p \in \text{atom}(\Pi) \} \\
\cup \{ \delta(\beta) \mid \beta \in \text{body}(\Pi) \} \cup \{ \delta \in \Delta(\beta) \mid \beta \in \text{body}(\Pi) \} .
\]

**Theorem**

*Let \( \Pi \) be a tight logic program. Then,*

- \( X \subseteq \text{atom}(\Pi) \) is an answer set of \( \Pi \) *iff*
- \( X = A^T \cap \text{atom}(\Pi) \) for a (unique) solution \( A \) for \( \Delta_\Pi \).*
Characterization of answer sets
for tight logic programs, ie. free of positive recursion

Let $\Pi$ be a logic program and

$$\Delta_\Pi = \{\delta(p) \mid p \in \text{atom}(\Pi)\} \cup \{\delta \in \Delta(p) \mid p \in \text{atom}(\Pi)\}$$
$$\quad \cup \{\delta(\beta) \mid \beta \in \text{body}(\Pi)\} \cup \{\delta \in \Delta(\beta) \mid \beta \in \text{body}(\Pi)\}.$$  

**Theorem**

Let $\Pi$ be a tight logic program. Then,

$X \subseteq \text{atom}(\Pi)$ is an answer set of $\Pi$ iff

$X = A^T \cap \text{atom}(\Pi)$ for a (unique) solution $A$ for $\Delta_\Pi$. 

Torsten Schaub et al. (KRR@UP)  Modeling and Solving in ASP 129 / 226
Let $\Pi$ be a normal logic program and recall that:

- For $L \subseteq \text{atom}(\Pi)$, the external supports of $L$ for $\Pi$ are
  \[ ES_\Pi(L) = \{ r \in \Pi \mid \text{head}(r) \in L, \text{body}(r)^+ \cap L = \emptyset \}. \]

- The (disjunctive) loop formula of $L$ for $\Pi$ is
  \[ LF_\Pi(L) = (\bigvee_{A \in L} A) \rightarrow (\bigvee_{r \in ES_\Pi(L)} \text{body}(r)) \]
  \[ \equiv (\bigwedge_{r \in ES_\Pi(L)} \neg \text{body}(r)) \rightarrow (\bigwedge_{A \in L} \neg A) \]

  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 $\Pi$ are
  \[ EB(L) = \{ \text{body}(r) \mid r \in ES_\Pi(L) \} \]
Nogoods from logic programs
via loop formulas

Let $\Pi$ be a normal logic program and recall that:

- For $L \subseteq \text{atom}(\Pi)$, the external supports of $L$ for $\Pi$ are
  \[ ES_{\Pi}(L) = \{ r \in \Pi \mid \text{head}(r) \in L, \text{body}(r)^+ \cap L = \emptyset \} \].

- The (disjunctive) loop formula of $L$ for $\Pi$ is
  \[ LF_{\Pi}(L) = \left( \bigvee_{A \in L} A \right) \rightarrow \left( \bigvee_{r \in ES_{\Pi}(L)} \text{body}(r) \right) \]
  \[ \equiv \left( \bigwedge_{r \in ES_{\Pi}(L)} \neg \text{body}(r) \right) \rightarrow \left( \bigwedge_{A \in L} \neg A \right) \]

  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 $\Pi$ are
  \[ EB(L) = \{ \text{body}(r) \mid r \in ES_{\Pi}(L) \} \]
Let $\Pi$ be a normal logic program and recall that:

- For $L \subseteq \text{atom}(\Pi)$, the external supports of $L$ for $\Pi$ are
  
  \[ ES_\Pi(L) = \{ r \in \Pi \mid \text{head}(r) \in L, \text{body}(r)^+ \cap L = \emptyset \} \]

- The (disjunctive) loop formula of $L$ for $\Pi$ is
  
  \[ LF_\Pi(L) = (\bigvee_{A \in L} A) \rightarrow (\bigvee_{r \in ES_\Pi(L)} \text{body}(r)) \]
  \[ \equiv (\bigwedge_{r \in ES_\Pi(L)} \neg \text{body}(r)) \rightarrow (\bigwedge_{A \in L} \neg A) \]

  The loop formula of $L$ enforces all atoms in $L$ to be \textit{false} whenever $L$ is not externally supported.

- The external bodies of $L$ for $\Pi$ are
  
  \[ EB(L) = \{ \text{body}(r) \mid r \in ES_\Pi(L) \} \]
For a logic program \( \Pi \) and some \( \emptyset \subset U \subseteq \text{atom}(\Pi) \), define the \textit{loop nogood} of an atom \( p \in U \) as

\[
\lambda(p, U) = \{ Tp, F\beta_1, \ldots, F\beta_k \}
\]

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

In all, we get the following set of loop nogoods for \( \Pi \):

\[
\Lambda_\Pi = \bigcup_{\emptyset \subset U \subseteq \text{atom}(\Pi)} \{ \lambda(p, U) \mid p \in U \}
\]

The set \( \Lambda_\Pi \) of loop nogoods denies cyclic support among \textit{true} atoms.
Nogoods from logic programs

loop nogoods

- For a logic program $\Pi$ and some $\emptyset \subseteq U \subseteq \text{atom}(\Pi)$, define the loop nogood of an atom $p \in U$ as
  \[ \lambda(p, U) = \{ Tp, F\beta_1, \ldots, F\beta_k \} \]
  where $EB(U) = \{ \beta_1, \ldots, \beta_k \}$.

- In all, we get the following set of loop nogoods for $\Pi$:
  \[ \Lambda_{\Pi} = \bigcup_{\emptyset \subseteq U \subseteq \text{atom}(\Pi)} \{ \lambda(p, U) \mid p \in U \} \]

- The set $\Lambda_{\Pi}$ of loop nogoods denies cyclic support among true atoms.
For a logic program $\Pi$ and some $\emptyset \subset U \subseteq \text{atom}(\Pi)$, define the loop nogood of an atom $p \in U$ as

$$\lambda(p, U) = \{ Tp, F_{\beta_1}, \ldots, F_{\beta_k} \}$$

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

In all, we get the following set of loop nogoods for $\Pi$:

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

The set $\Lambda_{\Pi}$ of loop nogoods denies cyclic support among true atoms.
Consider

\[ \Pi = \left\{ \begin{array}{l}
x \leftarrow \text{not } y \\
y \leftarrow \text{not } x \\
u \leftarrow x \\
u \leftarrow v \\
v \leftarrow u, y \\
\end{array} \right\} \]

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

\[ \lambda(u, \{ u, v \}) = \{ T_u, F\{x\} \} \]

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

\[ \lambda(v, \{ u, v \}) = \{ T_v, F\{x\} \} \]
Example

Consider

\[ \Pi = \{ \begin{array}{ll}
  x & \leftarrow \text{not } y \\
  y & \leftarrow \text{not } x \\
  u & \leftarrow x \\
  u & \leftarrow v \\
  v & \leftarrow u, y \\
\end{array} \} \]

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

\[ \lambda(u, \{u, v\}) = \{ T u, F\{x}\} \]

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

\[ \lambda(v, \{u, v\}) = \{ T v, F\{x}\} \]
Consider

\[ \Pi = \left\{ \begin{array}{c}
  x \leftarrow \text{not } y \\
  y \leftarrow \text{not } x \\
  u \leftarrow x \\
  u \leftarrow v \\
  v \leftarrow u, y
\end{array} \right\} \]

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

\[ \lambda(u, \{u, v\}) = \{T u, F\{x\}\} \]

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

\[ \lambda(v, \{u, v\}) = \{T v, F\{x\}\} \]
Characterization of answer sets

**Theorem**

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

**Some remarks**

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

**Theorem**

Let $\Pi$ be a logic program. Then,

- $X \subseteq \text{atom}(\Pi)$ is an answer set of $\Pi$ iff
- $X = A^T \cap \text{atom}(\Pi)$ for a (unique) solution $A$ for $\Delta_\Pi \cup \Lambda_\Pi$.

**Some remarks**

- Nogoods in $\Lambda_\Pi$ augment $\Delta_\Pi$ with conditions checking for unfounded sets, in particular, those being loops.
- While $|\Delta_\Pi|$ is linear in the size of $\Pi$, $\Lambda_\Pi$ may contain exponentially many (non-redundant) loop nogoods.
Conflict-Driven Answer Set Solving: Overview

Motivation

Boolean Constraints

Nogoods from Logic Programs
- Nogoods from program completion
- Nogoods from loop formulas

Conflict-Driven Nogood Learning
- CDNL-ASP Algorithm
- Nogood Propagation
- Conflict Analysis
Conflict-driven search

Boolean constraint solving algorithms pioneered for SAT led to:

- Traditional DPLL-style approach
  - (Unit) propagation
  - (Chronological) backtracking

- Modern CDCL-style approach
  - (Unit) propagation
  - Conflict analysis (via resolution)
  - Learning + Backjumping + Assertion
DPLL-style solving

loop

propagate  // compute deterministic consequences

if no conflict then
 if all variables assigned then return variable assignment
 else decide  // non-deterministically assign some literal
else
 if top-level conflict then return unsatisfiable
 else
 backtrack  // undo assignments made after last decision
 flip  // assign complement of last decision literal
Conflict-Driven Nogood Learning

CDCL-style solving

loop

propagate  // compute deterministic consequences

if no conflict then

  if all variables assigned then return variable assignment
  else decide  // non-deterministically assign some literal

else

  if top-level conflict then return unsatisfiable
  else

    analyze  // analyze conflict and add a conflict constraint
    backjump  // undo assignments until conflict constraint is unit
Outline of CDNL-ASP algorithm

- Keep track of deterministic consequences by unit propagation on:
  - Program completion \([\Delta_\Pi]\)
  - Loop nogoods, determined and recorded on demand \([\Lambda_\Pi]\)
    - Dedicated unfounded set detection
  - Dynamic nogoods, derived from conflicts and unfounded sets \([\nabla]\)

- When a nogood in \(\Delta_\Pi \cup \nabla\) becomes violated:
  - Analyze the conflict by resolution (until reaching a Unique Implication Point, short: UIP)
  - Learn the derived conflict nogood \(\delta\)
  - Backjump to the earliest (heuristic) choice such that the complement of the UIP is unit-resulting for \(\delta\)
  - Assert the complement of the 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:
  - Program completion $[\Delta_\Pi]$  
  - Loop nogoods, determined and recorded on demand $[\Lambda_\Pi]$
    - Dedicated unfounded set detection !
  - Dynamic nogoods, derived from conflicts and unfounded sets $[\nabla]$

- When a nogood in $\Delta_\Pi \cup \nabla$ becomes violated:
  - Analyze the conflict by resolution (until reaching a Unique Implication Point, short: UIP)
  - Learn the derived conflict nogood $\delta$
  - Backjump to the earliest (heuristic) choice such that the complement of the UIP is unit-resulting for $\delta$
  - Assert the complement of the 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:
  - Program completion  \([\Delta_\Pi]\)
  - Loop nogoods, determined and recorded on demand  \([\Lambda_\Pi]\)
    - Dedicated unfounded set detection !
  - Dynamic nogoods, derived from conflicts and unfounded sets  \([\nabla]\)

- When a nogood in \(\Delta_\Pi \cup \nabla\) becomes violated:
  - Analyze the conflict by resolution
    (until reaching a Unique Implication Point, short: UIP)
  - Learn the derived conflict nogood \(\delta\)
  - Backjump to the earliest (heuristic) choice such that the complement of the UIP is unit-resulting for \(\delta\)
  - Assert the complement of the 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 $\Pi$.

**Output**: An answer set of $\Pi$ or “no answer set”.

A $\leftarrow \emptyset$  
// assignment over $\text{atom}(\Pi) \cup \text{body}(\Pi)$

$\nabla \leftarrow \emptyset$  
// set of (dynamic) nogoods

dl $\leftarrow 0$  
// decision level

**Loop**

\[
(A, \nabla) \leftarrow \text{NogoodPropagation}(\Pi, \nabla, A)
\]

**if** $\varepsilon \subseteq A$ for some $\varepsilon \in \Delta_\Pi \cup \nabla$ **then**

**if** $\text{dl} = 0$ **then return** no answer set

$(\delta, k) \leftarrow \text{ConflictAnalysis}(\varepsilon, \Pi, \nabla, A)$

$\nabla \leftarrow \nabla \cup \{\delta\}$  
// learning

$A \leftarrow (A \setminus \{\sigma \in A \mid k < \text{dl}(\sigma)\})$  
// backjumping

$\text{dl} \leftarrow k$

**else if** $A^T \cup A^F = \text{atom}(\Pi) \cup \text{body}(\Pi)$ **then**

**return** $A^T \cap \text{atom}(\Pi)$  
// answer set

**else**

$\sigma_d \leftarrow \text{Select}(\Pi, \nabla, A)$  
// heuristic choice of $\sigma_d \notin A$

$\text{dl} \leftarrow \text{dl} + 1$

$A \leftarrow A \circ (\sigma_d)$  
// $\text{dl}(\sigma_d) = \text{dl}$
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 $\sigma_d = Tp$ or $\sigma_d = Fp$, respectively, we require $p \in (\text{atom}(\Pi) \cup \text{body}(\Pi)) \setminus (A^T \cup A^F)$.

- For any literal $\sigma \in A$, $dl(\sigma)$ denotes the decision level of $\sigma$, viz. the value $dl$ had when $\sigma$ 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 $\delta$ derived by conflict analysis is asserting, that is, some literal is unit-resulting for $\delta$ at a decision level $k < dl$.
  - After learning $\delta$ 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 $\sigma_d = Tp$ or $\sigma_d = Fp$, respectively, we require $p \in (\text{atom} (\Pi) \cup \text{body} (\Pi)) \setminus (A^T \cup A^F)$.

- For any literal $\sigma \in A$, $dl(\sigma)$ denotes the decision level of $\sigma$, viz. the value $dl$ had when $\sigma$ 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 $\delta$ derived by conflict analysis is asserting, that is, some literal is unit-resulting for $\delta$ at a decision level $k < dl$.
  - After learning $\delta$ 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 $\sigma_d = Tp$ or $\sigma_d = Fp$, respectively, we require $p \in (\text{atom}(\Pi) \cup \text{body}(\Pi)) \setminus (A^T \cup A^F)$.
- For any literal $\sigma \in A$, $dl(\sigma)$ denotes the decision level of $\sigma$, viz. the value $dl$ had when $\sigma$ was assigned.
- A conflict is detected from violation of a nogood $\epsilon \subseteq \Delta_\Pi \cup \nabla$.
- A conflict at decision level 0 (where $A$ contains no heuristically chosen literals) indicates non-existence of answer sets.
- A nogood $\delta$ derived by conflict analysis is asserting, that is, some literal is unit-resulting for $\delta$ at a decision level $k < dl$.
  - After learning $\delta$ 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

\[ \Pi = \{ x \leftarrow \text{not } y, y \leftarrow \text{not } x, u \leftarrow x, y, v \leftarrow x, w \leftarrow \text{not } x, \text{not } y \} \]

<table>
<thead>
<tr>
<th>dl</th>
<th>( \sigma_d )</th>
<th>( \overline{\sigma} )</th>
<th>( \delta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( Tu )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( F{\text{not } x, \text{not } y} )</td>
<td>( Fw )</td>
<td>( {Tw, F{\text{not } x, \text{not } y}} = \delta(w) )</td>
</tr>
<tr>
<td>3</td>
<td>( F{\text{not } y} )</td>
<td>( Fx )</td>
<td>( {Tx, F{\text{not } y}} = \delta(x) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( F{x} )</td>
<td>( {T{x}, Fx} \in \Delta({x}) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( F{x, y} )</td>
<td>( {T{x, y}, Fx} \in \Delta({x, y}) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( {Tu, F{x}, F{x, y}} = \lambda(u, {u, v}) )</td>
</tr>
</tbody>
</table>
Example: CDNL-ASP

Consider

\[ \Pi = \begin{cases} x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\ y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y \end{cases} \]

<table>
<thead>
<tr>
<th>( dl )</th>
<th>( \sigma_d )</th>
<th>( \overline{\sigma} )</th>
<th>( \delta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( T_u )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( F{\neg x, \neg y} )</td>
<td>( F_w )</td>
<td>( {T_w, F{\neg x, \neg y}} = \delta(w) )</td>
</tr>
<tr>
<td>3</td>
<td>( F{\neg y} )</td>
<td>( F_x )</td>
<td>( {T_x, F{\neg y}} = \delta(x) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( F{x} )</td>
<td>( {T{x}, F_x} \in \Delta({x}) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( F{x, y} )</td>
<td>( {T{x, y}, F_x} \in \Delta({x, y}) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( {T_u, F{x}, F{x, y}} = \lambda(u, {u, v}) )</td>
<td></td>
</tr>
</tbody>
</table>
Example: CDNL-ASP

Consider

\[ \Pi = \{ x \leftarrow \text{not} \ y, \ y \leftarrow \text{not} \ x, \ u \leftarrow \text{not} \ y, \ u \leftarrow x, y, \ v \leftarrow x, \ w \leftarrow \text{not} \ x, \text{not} \ y, \ v \leftarrow u, y \} \]

<table>
<thead>
<tr>
<th>dl</th>
<th>\sigma_d</th>
<th>\sigma</th>
<th>\delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\textbf{T}u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>\textbf{F}{\text{not} \ x, \text{not} \ y}}</td>
<td>\textbf{F}w</td>
<td>{\textbf{T}w, \textbf{F}{\text{not} \ x, \text{not} \ y}}} = \delta(w)</td>
</tr>
<tr>
<td>3</td>
<td>\textbf{F}{\text{not} \ y}}</td>
<td>\textbf{F}x, \textbf{F}{\text{x}}</td>
<td>{\textbf{T}x, \textbf{F}{\text{not} \ y}}} = \delta(x), {\textbf{T}{\text{x}}, \textbf{F}x} \in \Delta({\text{x}}), {\textbf{T}{\text{x}, y}, \textbf{F}x} \in \Delta({\text{x}, y})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\textbf{F}{x, y}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>\vdots</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{\textbf{T}u, \textbf{F}{x}, \textbf{F}{x, y}} = \lambda(u, {u, v})</td>
<td></td>
</tr>
</tbody>
</table>
### Example: CDNL-ASP

Consider

\[ \Pi = \begin{cases} 
  x \leftarrow \text{not} \ y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \text{not} \ x, \text{not} \ y \\
  \ y \leftarrow \text{not} \ x & u \leftarrow v & v \leftarrow u, y 
\end{cases} \]

<table>
<thead>
<tr>
<th>dl</th>
<th>( \sigma_d )</th>
<th>( \bar{\sigma} )</th>
<th>( \delta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( Tu )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( F{\text{not} \ x, \text{not} \ y} )</td>
<td>( Fw )</td>
<td>( { Tw, F{\text{not} \ x, \text{not} \ y}} = \delta(w) )</td>
</tr>
<tr>
<td>3</td>
<td>( F{\text{not} \ y} )</td>
<td>( Fx )</td>
<td>( { Tx, F{\text{not} \ y}} = \delta(x) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( F{x} )</td>
<td>( { T{x}, Fx} \in \Delta({x}) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( F{x, y} )</td>
<td>( { T{x, y}, Fx} \in \Delta({x, y}) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( { Tu, F{x}, F{x, y}} = \lambda(u, {u, v}) )</td>
<td></td>
</tr>
</tbody>
</table>
Example: CDNL-ASP

Consider

\[ \Pi = \begin{cases} 
  x & \leftarrow \text{not } y \\
  u & \leftarrow x, y \\
  v & \leftarrow x \\
  w & \leftarrow \text{not } x, \text{not } y \\
  y & \leftarrow \text{not } x \\
  u & \leftarrow v \\
  v & \leftarrow u, y 
\end{cases} \]

<table>
<thead>
<tr>
<th>(dl)</th>
<th>(\sigma_d)</th>
<th>(\overline{\sigma})</th>
<th>(\delta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(T u)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(F{\text{not } x, \text{not } y})</td>
<td>(F w)</td>
<td>({T w, F{\text{not } x, \text{not } y}} = \delta(w))</td>
</tr>
<tr>
<td>3</td>
<td>(F{\text{not } y})</td>
<td>(F x)</td>
<td>({T x, F{\text{not } y}} = \delta(x))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(F{x})</td>
<td>({T{x}, F x} \in \Delta({x}))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(F{x, y})</td>
<td>({T{x, y}, F x} \in \Delta({x, y}))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\vdots)</td>
<td>(\vdots)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: CDNL-ASP

Consider

\[ \Pi = \{ x \leftarrow \text{not } y, u \leftarrow x, y, v \leftarrow x, w \leftarrow \text{not } x, \text{not } y, y \leftarrow \text{not } x, u \leftarrow v, v \leftarrow u, y \} \]

<table>
<thead>
<tr>
<th>dl</th>
<th>( \sigma_d )</th>
<th>( \overline{\sigma} )</th>
<th>( \delta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( T_u )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( F{ \text{not } x, \text{not } y } )</td>
<td>( F_w )</td>
<td>( { T_w, F{ \text{not } x, \text{not } y } } = \delta(w) )</td>
</tr>
<tr>
<td>3</td>
<td>( F{ \text{not } y } )</td>
<td>( F_x )</td>
<td>( { T_x, F{ \text{not } y } } = \delta(x) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( F{ x } )</td>
<td>( { T{ x }, F_x } \in \Delta({ x }) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( F{ x, y } )</td>
<td>( { T{ x, y }, F_x } \in \Delta({ x, y }) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( { T_u, F{ x }, F{ x, y } } = \lambda(u, { u, v }) )</td>
<td></td>
</tr>
</tbody>
</table>
Example: CDNL-ASP

Consider

\[ \Pi = \begin{cases} 
  x \leftarrow \neg y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \neg x, \neg y \\
  y \leftarrow \neg x & u \leftarrow v & v \leftarrow u, y 
\end{cases} \]

<table>
<thead>
<tr>
<th>dl</th>
<th>( \sigma_d )</th>
<th>( \bar{\sigma} )</th>
<th>( \delta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( Tu )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( F{ \neg x, \neg y } )</td>
<td>( Fw )</td>
<td>{ ( Tw, F{ \neg x, \neg y } ) } = \delta(w)</td>
</tr>
<tr>
<td>3</td>
<td>( F{ \neg y } )</td>
<td>( Fx )</td>
<td>{ ( Tx, F{ \neg y } ) } = \delta(x)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( F{ x } )</td>
<td>{ ( T{ x }, Fx ) } \in \Delta({x})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( F{ x, y } )</td>
<td>{ ( T{ x, y }, Fx ) } \in \Delta({x, y})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ ( Tu, F{ x }, F{ x, y } ) } = \lambda(u, {u, v})</td>
<td></td>
</tr>
</tbody>
</table>

Torsten Schaub et al. (KRR@UP) Modeling and Solving in ASP
Example: CDNL-ASP

Consider

\[ \Pi = \begin{cases} 
    x \leftarrow \text{not } y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \text{not } x, \text{not } y \\
    y \leftarrow \text{not } x & u \leftarrow v & v \leftarrow u, y 
\end{cases} \]

<table>
<thead>
<tr>
<th>( dl )</th>
<th>( \sigma_d )</th>
<th>( \overline{\sigma} )</th>
<th>( \delta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( T_u )</td>
<td>( T_x )</td>
<td>( { T_u, F_x } \in \nabla )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td></td>
<td>( T_v )</td>
<td>( { F_v, T{x} } \in \Delta(v) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( F_y )</td>
<td>( { T_y, F{\text{not } x} } = \delta(y) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( F_w )</td>
<td>( { T_w, F{\text{not } x, \text{not } y} } = \delta(w) )</td>
<td></td>
</tr>
</tbody>
</table>
Example: CDNL-ASP

Consider

\[
\Pi = \left\{ \begin{array}{l}
x \leftarrow \text{not } y \\
y \leftarrow \text{not } x \\
u \leftarrow x, y \\
v \leftarrow x \\
w \leftarrow \text{not } x, \text{not } y \\
u \leftarrow v \\
v \leftarrow u, y
\end{array} \right\}
\]

<table>
<thead>
<tr>
<th>dl</th>
<th>(\sigma_d)</th>
<th>(\bar{\sigma})</th>
<th>(\delta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(T_u)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(T_x)</td>
<td>({T_u, F_x} \in \nabla)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>:</td>
<td>:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(T_v)</td>
<td>({F_v, T{x}} \in \Delta(v))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(F_y)</td>
<td>({T_y, F{\text{not } x}} = \delta(y))</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(F_w)</td>
<td>({T_w, F{\text{not } x, \text{not } y}} = \delta(w))</td>
<td></td>
</tr>
</tbody>
</table>
Consider

\[ \Pi = \{ x \leftarrow \text{not } y \quad u \leftarrow x, y \quad v \leftarrow x \quad w \leftarrow \text{not } x, \text{not } y \} \]
\[ y \leftarrow \text{not } x \quad u \leftarrow v \quad v \leftarrow u, y \]
Consider

\[ \Pi = \begin{cases} 
  x \leftarrow \text{not } y & u \leftarrow x, y & v \leftarrow x & w \leftarrow \text{not } x, \text{not } y \\
  y \leftarrow \text{not } x & u \leftarrow v & v \leftarrow u, y 
\end{cases} \]

<table>
<thead>
<tr>
<th>\text{dl}</th>
<th>\sigma_d</th>
<th>\overline{\sigma}</th>
<th>\delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\text{T}u</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>\text{T}x</td>
<td>{\text{T}u, \text{F}x} \in \nabla</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\vdots</td>
<td>{\text{F}v, \text{T}{x}} \in \Delta(v)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\text{T}v</td>
<td>{\text{T}y, \text{F}{\text{not } x}} = \delta(y)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\text{F}y</td>
<td>{\text{T}w, \text{F}{\text{not } x, \text{not } y}} = \delta(w)</td>
<td></td>
</tr>
</tbody>
</table>
Outline of NogoodPropogation

- Derive deterministic consequences via:
  - Unit propagation on $\Delta_\Pi$ and $\nabla$;
  - Unfounded sets $U \subseteq \text{atom}(\Pi)$.

- Note that $U$ is unfounded if $EB(U) \subseteq A^F$.
  - For any $p \in U$, we have $(\lambda(p, U) \setminus \{T\, p\}) \subseteq A$.

- An “interesting” unfounded set $U$ satisfies:
  $$\emptyset \subset U \subseteq (\text{atom}(\Pi) \setminus A^F)$$.

- Wrt a fixpoint of unit propagation, such an unfounded set contains some loop of $\Pi$.
  - Tight programs do not yield “interesting” unfounded sets!

- Given an unfounded set $U$ and some $p \in U$, adding $\lambda(p, U)$ to $\nabla$ 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 $\Delta_\Pi$ and $\nabla$;
  - Unfounded sets $U \subseteq \text{atom}(\Pi)$.

- Note that $U$ is unfounded if $EB(U) \subseteq A^F$.
  - For any $p \in U$, we have $(\lambda(p, U) \setminus \{T\ p\}) \subseteq A$.

- An “interesting” unfounded set $U$ satisfies:

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

- Wrt a fixpoint of unit propagation, such an unfounded set contains some loop of $\Pi$.
  - Tight programs do not yield “interesting” unfounded sets!

- Given an unfounded set $U$ and some $p \in U$, adding $\lambda(p, U)$ to $\nabla$ triggers a conflict or further derivations by unit propagation.
  - Add loop nogoods atom by atom to eventually falsify all $p \in U$. 
Conflict-Driven Nogood Learning

Nogood Propagation

Outline of NogoodPropagation

- Derive deterministic consequences via:
  - Unit propagation on $\Delta_\Pi$ and $\nabla$;
  - Unfounded sets $U \subseteq \text{atom}(\Pi)$.

- Note that $U$ is unfounded if $EB(U) \subseteq A^F$.
  - For any $p \in U$, we have $(\lambda(p, U) \setminus \{T \cdot p\}) \subseteq A$.

- An “interesting” unfounded set $U$ satisfies:
  \[ \emptyset \subset U \subseteq (\text{atom}(\Pi) \setminus A^F) \, . \]

- Wrt a fixpoint of unit propagation, such an unfounded set contains some loop of $\Pi$.
  - Tight programs do not yield “interesting” unfounded sets!

- Given an unfounded set $U$ and some $p \in U$, adding $\lambda(p, U)$ to $\nabla$ triggers a conflict or further derivations by unit propagation.
  - Add loop nogoods atom by atom to eventually falsify all $p \in U$. 

Torsten Schaub et al. (KRR@UP)
Modeling and Solving in ASP

143 / 226
Outline of NogoodPropagation

- Derive deterministic consequences via:
  - Unit propagation on $\Delta_\Pi$ and $\nabla$;
  - Unfounded sets $U \subseteq \text{atom}(\Pi)$.
- Note that $U$ is unfounded if $EB(U) \subseteq A^F$.
  - For any $p \in U$, we have $(\lambda(p, U) \setminus \{T p\}) \subseteq A$.
- An “interesting” unfounded set $U$ satisfies:
  $$\emptyset \subset U \subseteq (\text{atom}(\Pi) \setminus A^F).$$
- Wrt a fixpoint of unit propagation, such an unfounded set contains some loop of $\Pi$.
  - Tight programs do not yield “interesting” unfounded sets!
- Given an unfounded set $U$ and some $p \in U$, adding $\lambda(p, U)$ to $\nabla$ triggers a conflict or further derivations by unit propagation.
  - Add loop nogoods atom by atom to eventually falsify all $p \in U$. 

Torsten Schaub et al. (KRR@UP)  
Modeling and Solving in ASP
Algorithm 2: NogoodPropagation

Input: A logic program $\Pi$, a set $\nabla$ 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 $(A, \nabla)$ // conflict
    $\Sigma \leftarrow \{\delta \in \Delta_\Pi \cup \nabla \mid (\delta \setminus A) = \{\sigma\}, \bar{\sigma} \notin A\}$ // unit-resulting nogoods
    if $\Sigma \neq \emptyset$ then
      let $\sigma \in (\delta \setminus A)$ for some $\delta \in \Sigma$ in
      $A \leftarrow A \circ (\bar{\sigma})$ // $dl(\bar{\sigma}) = \max(\{dl(\rho) \mid \rho \in (\delta \setminus \{\sigma\})\}) \cup \{0\}$
    until $\Sigma = \emptyset$

if $\Pi$ is tight then return $(A, \nabla)$ // no unfounded set $\emptyset \subset U \subseteq (\text{atom}(\Pi) \setminus A^F)$
else
  $U \leftarrow (U \setminus A^F)$
  if $U = \emptyset$ then $U \leftarrow \text{UnfoundedSet}(\Pi, A)$
  if $U = \emptyset$ then return $(A, \nabla)$ // no unfounded set $\emptyset \subset U \subseteq (\text{atom}(\Pi) \setminus A^F)$
  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. $U \subseteq (\text{atom}(\Pi) \setminus A^F)$
  2. $E_B(U) \subseteq A^F$
  3. $U = \emptyset$ iff there is no nonempty unfounded subset of $(\text{atom}(\Pi) \setminus A^F)$

- 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 $\Pi$
  - Usually, the latter option is implemented in ASP solvers
Requirements for UnfoundedSet

- Implementations of UnfoundedSet must guarantee the following for a result $U$
  1. $U \subseteq (\text{atom}(\Pi) \setminus A^F)$
  2. $EB(U) \subseteq A^F$
  3. $U = \emptyset$ iff there is no nonempty unfounded subset of $(\text{atom}(\Pi) \setminus A^F)$

- 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 $\Pi$
  - Usually, the latter option is implemented in ASP solvers
Example: NogoodPropagation

Consider

\[ \Pi = \{ x \leftarrow \text{not } y, \ u \leftarrow x, y, \ v \leftarrow x, \ w \leftarrow \text{not } x, \text{not } y, \ y \leftarrow \text{not } x, \ u \leftarrow v, \ v \leftarrow u, y \} \]

<table>
<thead>
<tr>
<th>(dl)</th>
<th>(\sigma_d)</th>
<th>(\sigma)</th>
<th>(\delta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(Tu)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(F{\text{not } x, \text{not } y})</td>
<td>(Fw)</td>
<td>({Tw, F{\text{not } x, \text{not } y}} = \delta(w))</td>
</tr>
<tr>
<td>3</td>
<td>(F{\text{not } y})</td>
<td>(Fx, F{x}, F{x, y}, F{\text{not } x}, Ty, F{v}, F{u, y}, F{u, v, y})</td>
<td>({Fx, F{\text{not } y}} = \delta(x)), ({T{x}, Fx} \in \Delta({x})), ({T{x, y}, Fx} \in \Delta({x, y})), ({F{\text{not } x}, Fx} = \delta({\text{not } x})), ({F{\text{not } y}, Fy} = \delta({\text{not } y})), ({Tu, F{x, y}, F{v}} = \delta(u)), ({F{u, y}, Tu, Ty} = \delta({u, y})), ({Fv, T{u, y}} \in \Delta(v)), ({Tu, F{x}, F{x, y}} = \lambda(u, {u, v}))</td>
</tr>
</tbody>
</table>
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 \( \varepsilon \), we obtain a new violated nogood by resolving \( \delta \) and \( \varepsilon \) as follows:

\[
(\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\sigma\}).
\]

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 \( \delta \) containing exactly one literal \( \sigma \) assigned at decision level \( dl \).
- This literal \( \sigma \) is called First Unique Implication Point (First-UIP).
- All literals in \( (\delta \setminus \{\sigma\}) \) are assigned at decision levels smaller than \( dl \).
Conflict-Driven Nogood Learning

Conflict Analysis

Outline of Conflict Analysis

- 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 $\varepsilon$, we obtain a new violated nogood by resolving $\delta$ and $\varepsilon$ 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 $\delta$ containing exactly one literal $\sigma$ assigned at decision level $dl$.
    - This literal $\sigma$ is called First Unique Implication Point (First-UIP).
    - All literals in $(\delta \setminus \{\sigma\})$ are assigned at decision levels smaller than $dl$. 
Outline of Conflict Analysis

- 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 $\varepsilon$, we obtain a new violated nogood by resolving $\delta$ and $\varepsilon$ 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 $\delta$ containing exactly one literal $\sigma$ assigned at decision level $dl$.
  - This literal $\sigma$ 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 $\delta$, a logic program $\Pi$, a set $\nabla$ of nogoods, and an assignment $A$.

**Output**
A derived nogood and a decision level.

**loop**

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

Example: ConflictAnalysis

Consider

\[ \Pi = \left\{ \begin{array}{l} x \leftarrow \text{not } y \\ u \leftarrow x, y \\ v \leftarrow x \\ w \leftarrow \text{not } x, \text{not } y \\ y \leftarrow \text{not } x \\ u \leftarrow v \\ v \leftarrow u, y \end{array} \right\} \]

<table>
<thead>
<tr>
<th>dl</th>
<th>\sigma_d</th>
<th>\bar{\sigma}</th>
<th>\delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tu</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>F{not x, not y}</td>
<td>Fw</td>
<td>{Tw, F{not x, not y}} = \delta(w)</td>
</tr>
<tr>
<td>3</td>
<td>F{not y}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fx</td>
<td></td>
<td>{Tx, F{not y}} = \delta(x)</td>
</tr>
<tr>
<td></td>
<td>F{x}</td>
<td></td>
<td>{Tx, Fx} \in \Delta({x})</td>
</tr>
<tr>
<td></td>
<td>F{x, y}</td>
<td></td>
<td>{Tx, y}, Fx \in \Delta({x, y})</td>
</tr>
<tr>
<td></td>
<td>T{not x}</td>
<td></td>
<td>{F{not x}, Fx} = \delta({not x})</td>
</tr>
<tr>
<td></td>
<td>Ty</td>
<td></td>
<td>{F{not y}, Fy} = \delta({not y})</td>
</tr>
<tr>
<td></td>
<td>T{v}</td>
<td></td>
<td>{Tu, F{x, y}, F{v}} = \delta(u)</td>
</tr>
<tr>
<td></td>
<td>T{u, y}</td>
<td></td>
<td>{Fx, Tu, Ty} = \delta({u, y})</td>
</tr>
<tr>
<td></td>
<td>Tv</td>
<td></td>
<td>{Fv, T{u, y}} \in \Delta(v)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>{Tu, F{x}, F{x, y}} = \lambda(u, {u, v})</td>
</tr>
</tbody>
</table>
Example: ConflictAnalysis

Consider

\[ \Pi = \{ x \leftarrow \neg y \quad u \leftarrow x, y \quad v \leftarrow x \quad w \leftarrow \neg x, \neg y \\
\quad y \leftarrow \neg x \quad u \leftarrow v \quad v \leftarrow u, y \} \]

<table>
<thead>
<tr>
<th>dl</th>
<th>( \sigma_d )</th>
<th>( \bar{\sigma} )</th>
<th>( \delta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T( u )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>F{\neg x, \neg y}</td>
<td>F( w )</td>
<td>{T( w ), F{\neg x, \neg y}} = \delta(w)</td>
</tr>
<tr>
<td>3</td>
<td>F{\neg y}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>F( x )</td>
<td>{T( x ), F{\neg y}} = \delta({\neg y})</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F{x}</td>
<td>{T{x}, F( x )} \in \Delta({x})</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F{x, y}</td>
<td>{T{x, y}, F( x )} \in \Delta({x, y})</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T{\neg x}</td>
<td>F{\neg x}, F( x ) = \delta({\neg x})</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T( y )</td>
<td>F{\neg y}, F( y ) = \delta({\neg y})</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T{v}</td>
<td>{T( u ), F{x, y}, F{v}} = \delta(u)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T{u, y}</td>
<td>{F{u, y}, T( u ), T( y )} = \delta({u, y})</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T( v )</td>
<td>F( v ), T{u, y} \in \Delta(v)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{T( u ), F{x}, F{x, y}} = \lambda(u, {u, v})</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( \{T\( u \), F\( x \\} \)
\( \{T\( u \), F\( x \), F\{x\}\} \)
Example: ConflictAnalysis

Consider

\[ \Pi = \begin{cases} 
  x \leftarrow \neg y \\
  y \leftarrow \neg x \\
  u \leftarrow x, y \\
  v \leftarrow x \\
  w \leftarrow \neg x, \neg y \\
  u \leftarrow v \\
  v \leftarrow u, y
\end{cases} \]

<table>
<thead>
<tr>
<th>dl</th>
<th>( \sigma_d )</th>
<th>( \overline{\sigma} )</th>
<th>( \delta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( Tu )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( F{\neg x, \neg y} )</td>
<td>( Fw )</td>
<td>( {Tw, F{\neg x, \neg y}} = \delta(w) )</td>
</tr>
<tr>
<td>3</td>
<td>( F{\neg y} )</td>
<td>( Fx, {x}, {x, y} )</td>
<td>( {Tu, Fx} )</td>
</tr>
<tr>
<td></td>
<td>( {Fx, {x}, {x, y}} \in \Delta({x}) )</td>
<td>( {Fx, {x}, {x, y}} \in \Delta({x, y}) )</td>
<td>( {Tu, Fx, F{x}} )</td>
</tr>
<tr>
<td></td>
<td>( {Fx} )</td>
<td>( {Fx, {x}} \in \Delta({x}) )</td>
<td>( {Tu, F{x, y}, F{v}} = \delta(u) )</td>
</tr>
<tr>
<td></td>
<td>( {F{not x}, Fx} = \delta({not x}) )</td>
<td>( {F{not y}, Fy} = \delta({not y}) )</td>
<td>( {Tu, F{x, y}, F{v}} = \delta(u) )</td>
</tr>
<tr>
<td></td>
<td>( {Fx} )</td>
<td>( {Fx, {x}} \in \Delta({x}) )</td>
<td>( {Fx} )</td>
</tr>
<tr>
<td></td>
<td>( {F{not y}} )</td>
<td>( {F{not y}} )</td>
<td>( {Tu, F{x, y}, F{v}} = \delta(u) )</td>
</tr>
<tr>
<td></td>
<td>( {Fx} )</td>
<td>( {Fx, {x}} \in \Delta({x}) )</td>
<td>( {Fx} )</td>
</tr>
<tr>
<td></td>
<td>( {F{not y}} )</td>
<td>( {F{not y}} )</td>
<td>( {Tu, F{x, y}, F{v}} = \delta(u) )</td>
</tr>
<tr>
<td></td>
<td>( {Fx} )</td>
<td>( {Fx, {x}} \in \Delta({x}) )</td>
<td>( {Fx} )</td>
</tr>
<tr>
<td></td>
<td>( {F{not y}} )</td>
<td>( {F{not y}} )</td>
<td>( {Tu, F{x, y}, F{v}} = \delta(u) )</td>
</tr>
<tr>
<td></td>
<td>( {Fx} )</td>
<td>( {Fx, {x}} \in \Delta({x}) )</td>
<td>( {Fx} )</td>
</tr>
<tr>
<td></td>
<td>( {F{not y}} )</td>
<td>( {F{not y}} )</td>
<td>( {Tu, F{x, y}, F{v}} = \delta(u) )</td>
</tr>
</tbody>
</table>

\[
\text{Torsten Schaub et al. (KRR@UP)}
\]
Example: ConflictAnalysis

Consider

\[ \Pi = \{ x \leftarrow \neg y \quad u \leftarrow x, y \quad v \leftarrow x \quad w \leftarrow \neg x, \neg y \quad y \leftarrow \neg x \quad u \leftarrow v \quad v \leftarrow u, y \} \]

<table>
<thead>
<tr>
<th>dl</th>
<th>( \sigma_d )</th>
<th>( \bar{\sigma} )</th>
<th>( \delta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( Tu )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( F{\neg x, \neg y} )</td>
<td>( Fw )</td>
<td>( {Tw, F{\neg x, \neg y}} = \delta(w) )</td>
</tr>
<tr>
<td>3</td>
<td>( F{\neg y} )</td>
<td>( Fx )</td>
<td>( {Tx, F{\neg y}} = \delta(x) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( F{x} )</td>
<td>( {T{x}, Fx} \in \Delta({x}) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( F{x,y} )</td>
<td>( {T{x,y}, Fx} \in \Delta({x,y}) )</td>
</tr>
<tr>
<td></td>
<td>( T{not\ x} )</td>
<td>( F{not\ x}, Fx } = \delta({not\ x}) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( Ty )</td>
<td>( F{not\ y}, Fy } = \delta({not\ y}) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( T{v} )</td>
<td>( {Tu, F{x,y}, F{v}} = \delta(u) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( T{u,y} )</td>
<td>( F{u,y}, Tu, Ty } = \delta({u,y}) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( Tv )</td>
<td>( Fv, T{u,y}} \in \Delta(v) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( {Tu, F{x}, F{x,y}} = \lambda(u, {u, v}) )</td>
<td>( X )</td>
<td></td>
</tr>
</tbody>
</table>
Example: Conflict Analysis

Consider

\[ \Pi = \{ \begin{align*}
  x &\leftarrow \text{not } y \\
  u &\leftarrow x, y \\
  v &\leftarrow x \\
  w &\leftarrow \text{not } x, \text{not } y \\
  y &\leftarrow \text{not } x \\ 
  u &\leftarrow v \\
  v &\leftarrow u, y 
\end{align*} \} \]

<table>
<thead>
<tr>
<th>dl</th>
<th>( \sigma_d )</th>
<th>( \overline{\sigma} )</th>
<th>( \delta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( T_u )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( F{\text{not } x, \text{not } y} )</td>
<td>( F_w )</td>
<td>( { T_w, F{\text{not } x, \text{not } y} } = \delta(w) )</td>
</tr>
<tr>
<td>3</td>
<td>( F{\text{not } y} )</td>
<td>( F_x )</td>
<td>( { T_x, F{\text{not } y} } = \delta(x) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( F{x} )</td>
<td>( { T{x}, F_x } \in \Delta({x}) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( F{x, y} )</td>
<td>( { T{x, y}, F_x } \in \Delta({x, y}) )</td>
</tr>
<tr>
<td></td>
<td>( T{\text{not } x} )</td>
<td>( F{\text{not } x}, F_x )</td>
<td>( \delta({\text{not } x}) )</td>
</tr>
<tr>
<td></td>
<td>( Ty )</td>
<td>( F{\text{not } y}, F_y )</td>
<td>( \delta({\text{not } y}) )</td>
</tr>
<tr>
<td></td>
<td>( T{v} )</td>
<td>( T_u, F{x, y}, F{v} )</td>
<td>( \delta(u) )</td>
</tr>
<tr>
<td></td>
<td>( T{u, y} )</td>
<td>( F{u, y}, T_u, Ty )</td>
<td>( \delta({u, y}) )</td>
</tr>
<tr>
<td></td>
<td>( Tv )</td>
<td>( F_{v}, T{u, y} )</td>
<td>( \in \Delta(v) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( T_u, F{x}, F{x, y} )</td>
<td>( \lambda(u, {u, v}) )</td>
</tr>
</tbody>
</table>
### Example: ConflictAnalysis

Consider

\[ \Pi = \{ \begin{align*} x & \leftarrow \text{not } y \\ u & \leftarrow x, y \\ v & \leftarrow x \\ w & \leftarrow \text{not } x, \text{not } y \\ y & \leftarrow \text{not } x \\ u & \leftarrow v \\ v & \leftarrow u, y \end{align*} \} \]

<table>
<thead>
<tr>
<th>dl</th>
<th>(\sigma_d)</th>
<th>(\overline{\sigma})</th>
<th>(\delta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(Tu)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(F{\text{not } x, \text{not } y})</td>
<td>(Fw)</td>
<td>({Tw, F{\text{not } x, \text{not } y}} = \delta(w))</td>
</tr>
<tr>
<td>3</td>
<td>(F{\text{not } y})</td>
<td>(Fx, F{x}, F{x, y}, T{\text{not } x}, Ty, T{v}, T{u, y}, Tv)</td>
<td>({Tu, F{x}, F{x, y}} = \lambda(u, {u, v}))</td>
</tr>
</tbody>
</table>

\(\{Tu, Fx, F\{x\}\}\)
Example: ConflictAnalysis

Consider

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

<table>
<thead>
<tr>
<th>dl</th>
<th>$\sigma_d$</th>
<th>$\overline{\sigma}$</th>
<th>$\delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$Tu$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$F{\text{not } x, \text{not } y}$</td>
<td>$Fw$</td>
<td>${Tw, F{\text{not } x, \text{not } y}} = \delta(w)$</td>
</tr>
<tr>
<td>3</td>
<td>$F{\text{not } y}$</td>
<td>$Fx$, ${x}$, $F{x}$</td>
<td>$\Delta({x})$</td>
</tr>
<tr>
<td></td>
<td>$F{x}$</td>
<td>${T{x}, Fx}$</td>
<td>$\Delta({x})$</td>
</tr>
<tr>
<td></td>
<td>$F{x, y}$</td>
<td>${T{x, y}, Fx}$</td>
<td>$\Delta({x, y})$</td>
</tr>
<tr>
<td></td>
<td>$T{\text{not } x}$</td>
<td>$F{\text{not } x}, Fx$</td>
<td>$\delta({\text{not } x})$</td>
</tr>
<tr>
<td></td>
<td>$Ty$</td>
<td>$F{\text{not } y}, Fy$</td>
<td>$\delta({\text{not } y})$</td>
</tr>
<tr>
<td></td>
<td>$T{v}$</td>
<td>$Tu, F{x, y}, F{v}$</td>
<td>$\delta(u)$</td>
</tr>
<tr>
<td></td>
<td>$T{u, y}$</td>
<td>$F{u, y}, Tu, Ty$</td>
<td>$\delta({u, y})$</td>
</tr>
<tr>
<td></td>
<td>$Tv$</td>
<td>$Fv, T{u, y}$</td>
<td>$\Delta(v)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>${Tu, F{x}, F{x, y}} = \lambda(u, {u, v})$</td>
<td></td>
</tr>
</tbody>
</table>
Example: ConflictAnalysis

Consider

\[ \Pi = \left\{ \begin{array}{l}
x \leftarrow \text{not } y \\
u \leftarrow x, y \\
v \leftarrow x \\
w \leftarrow \text{not } x, \text{not } y \\
y \leftarrow \text{not } x \\
u \leftarrow v \\
v \leftarrow u, y \\
\end{array} \right\} \]

<table>
<thead>
<tr>
<th>dl</th>
<th>(\sigma_d)</th>
<th>(\bar{\sigma})</th>
<th>(\delta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(Tu)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(F{\text{not } x, \text{not } y})</td>
<td>(Fw)</td>
<td>({Tw, F{\text{not } x, \text{not } y}} = \delta(w))</td>
</tr>
<tr>
<td>3</td>
<td>(F{\text{not } y})</td>
<td>(Fx)</td>
<td>({Tx, F{\text{not } y}} = \delta(x))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(F{x})</td>
<td>({Tx, Fx} \in \Delta({x}))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(F{x, y})</td>
<td>({Tx, y, Fx} \in \Delta({x, y}))</td>
</tr>
<tr>
<td></td>
<td>(T{\text{not } x})</td>
<td>(F{\text{not } x}, Fx)</td>
<td>(\delta({\text{not } x}))</td>
</tr>
<tr>
<td></td>
<td>(Ty)</td>
<td>(F{\text{not } y}, Fy)</td>
<td>(\delta({\text{not } y}))</td>
</tr>
<tr>
<td></td>
<td>(T{v})</td>
<td>(Tu, F{x, y}, F{v})</td>
<td>(\delta(u))</td>
</tr>
<tr>
<td></td>
<td>(T{u, y})</td>
<td>(F{u, y}, Tu, Ty)</td>
<td>(\delta({u, y}))</td>
</tr>
<tr>
<td></td>
<td>(Tv)</td>
<td>(Fv, T{u, y})</td>
<td>(\in \Delta(v))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Tu, F{x}, F{x, y})</td>
<td>(= \lambda(u, {u, v}))</td>
</tr>
</tbody>
</table>

\(X\)
Example: Conflict Analysis

Consider

\[ \Pi = \{ \begin{align*}
    x & \leftarrow \text{not } y & u & \leftarrow x, y & v & \leftarrow x & w & \leftarrow \text{not } x, \text{not } y \\
    y & \leftarrow \text{not } x & u & \leftarrow v & v & \leftarrow u, y
\end{align*} \} \]

<table>
<thead>
<tr>
<th>dl</th>
<th>( \sigma_\delta )</th>
<th>( \overline{\sigma} )</th>
<th>( \delta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( Tu )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( F{\text{not } x, \text{not } y} )</td>
<td>( Fw )</td>
<td>( {Tw, F{\text{not } x, \text{not } y}} = \delta(w) )</td>
</tr>
<tr>
<td>3</td>
<td>( F{\text{not } y} )</td>
<td>( Fx )</td>
<td>( {Tx, F{\text{not } y}} = \delta(x) )</td>
</tr>
<tr>
<td></td>
<td>( F{x} )</td>
<td>( {T{x}, Fx} \in \Delta({x}) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( F{x, y} )</td>
<td>( {T{x, y}, Fx} \in \Delta({x, y}) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( T{\text{not } x} )</td>
<td>( {F{\text{not } x}, Fx} = \delta({\text{not } x}) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( Ty )</td>
<td>( {F{\text{not } y}, Fy} = \delta({\text{not } y}) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( T{v} )</td>
<td>( {Tu, F{x, y}, F{v}} = \delta(u) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( T{u, y} )</td>
<td>( {F{u, y}, Tu, Ty} = \delta({u, y}) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( Tv )</td>
<td>( {Fv, T{u, y}} \in \Delta(v) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( {Tu, F{x}, F{x, y}} = \lambda(u, {u, v}) )</td>
<td>( X )</td>
<td></td>
</tr>
</tbody>
</table>
Consider

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

<table>
<thead>
<tr>
<th>$dl$</th>
<th>$\sigma_d$</th>
<th>$\bar{\sigma}$</th>
<th>$\delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$Tu$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$F{\text{not } x, \text{not } y}$</td>
<td>$Fw$</td>
<td>${Tw, F{\text{not } x, \text{not } y}} = \delta(w)$</td>
</tr>
<tr>
<td>3</td>
<td>$F{\text{not } y}$</td>
<td>$Fx, F{x}, F{x, y}, T{\text{not } x}, Ty, T{v}, T{u, y}, T\nu$</td>
<td>${Tu, F{x}}$, ${Tu, Fx, F{x}}$</td>
</tr>
</tbody>
</table>

$\Delta(\{x\})$, $\Delta(\{x, y\})$, $\Delta(v)$, $\lambda(u, \{u, v\})$
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 $\delta$ containing First-UIP $\sigma$ 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 $\delta$ in $\nabla$ and backjumping to decision level $k$, $\sigma$ is unit-resulting for $\delta$.
  - Such a nogood $\delta$ 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 $\delta$ containing First-UIP $\sigma$ 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 $\delta$ in $\nabla$ and backjumping to decision level $k$,
    $\overline{\sigma}$ is unit-resulting for $\delta$ !
  - Such a nogood $\delta$ 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 $\delta$ containing First-UIP $\sigma$ 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 $\delta$ in $\nabla$ and backjumping to decision level $k$, $\overline{\sigma}$ is unit-resulting for $\delta$.
  - Such a nogood $\delta$ 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 $\delta$ containing First-UIP $\sigma$ 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 $\delta$ in $\nabla$ and backjumping to decision level $k$, $\overline{\sigma}$ is unit-resulting for $\delta$ !
  - Such a nogood $\delta$ 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 !
Effective Modeling: Overview

28 Problems as Logic Programs (Revisited)
- Graph Coloring
- Hamiltonian Cycle
- Traveling Salesperson

29 Encoding Methodology
- Tweaking $N$-Queens
- Do’s and Dont’s

30 Hints
Effective Modeling: Overview

Problems as Logic Programs (Revisited)
- Graph Coloring
- Hamiltonian Cycle
- Traveling Salesperson

Encoding Methodology
- Tweaking $N$-Queens
- Do’s and Don’t’s

Hints
Problems as Logic Programs (Revisited)

Modeling and Interpreting

Problem

Modeling

Logic Program

Solving

Solution

Interpreting

Stable Models

Torsten Schaub et al. (KRR@UP)
Modeling and Interpreting

Problem

Logic Program

Modeling

Solution

Stable Models

Interpreting

Solving
Problems as 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 $C(I) \cup C(P)$.

Uniform encoding

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

**Uniform encoding**

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 $G$

\[
G = \left( \begin{array}{l}
V = \{1, 2, 3, 4, 5, 6\}, \\
E = \{(1, 2), (1, 3), (1, 4), (2, 4), (2, 5), (2, 6), (3, 1), (3, 4), (3, 5), (4, 1), (4, 2), (5, 3), (5, 4), (5, 6), (6, 2), (6, 3), (6, 5)\} \end{array} \right)
\]
N-Colorability

Problem Instance as Facts

Given: a (directed) graph $G$

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

<table>
<thead>
<tr>
<th>Natural Language</th>
<th>Logical Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Each node has a unique color.</td>
<td>1. <code>color(X,C) :- iscol(C), node(X), not other(X,C).</code></td>
</tr>
<tr>
<td>2. Any two connected nodes must not have the same color.</td>
<td>2. <code>other(X,C) :- iscol(C), color(X,D), D != C.</code></td>
</tr>
<tr>
<td>3. Let there be three colors.</td>
<td>3. <code>:- color(X,C), color(Y,C), edge(X,Y).</code></td>
</tr>
<tr>
<td>4. A solution is a coloring.</td>
<td>4. <code>#const n=3. iscol(1..n).</code></td>
</tr>
<tr>
<td></td>
<td><code>#hide. #show color/2.</code></td>
</tr>
</tbody>
</table>
### 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.`
### 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. \( \text{color}(X,C) :- \text{iscol}(C), \newline \quad \text{node}(X), \text{not other}(X,C). \)
2. \( \text{other}(X,C) :- \text{iscol}(C), \newline \quad \text{color}(X,D), D \neq C. \)
3. \( :- \text{color}(X,C), \text{color}(Y,C), \newline \quad \text{edge}(X,Y). \)
4. \( \#\text{const} \ n=3. \newline \quad \text{iscol}(1..n). \)
5. \( \#\text{hide}. \newline \quad \#\text{show} \ \text{color}/2. \)
### N-Colorability

**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. \( \text{color}(X,C) :- \text{iscol}(C), \text{node}(X), \text{not other}(X,C). \)
   
   \( \text{other}(X,C) :- \text{iscol}(C), \text{color}(X,D), D \neq C. \)

2. \( \text{:- color}(X,C), \text{color}(Y,C), \text{edge}(X,Y). \)

3. \#const n=3.
   
   \#hide.

4. \#show color/2.
### N-Colorability

#### (Extended) Problem Encoding

<table>
<thead>
<tr>
<th>Natural Language</th>
<th>Logical Language</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong> Each node has a unique color.</td>
<td><strong>1</strong> <code>color(X,C) :- iscol(C), node(X), not other(X,C).</code></td>
</tr>
<tr>
<td><strong>2</strong> Any two connected nodes must not have the same color.</td>
<td><code>other(X,C) :- iscol(C), color(X,D), D != C.</code></td>
</tr>
<tr>
<td><strong>3</strong> Let there be three colors.</td>
<td><strong>2</strong> <code>:- color(X,C), color(Y,C), edge(X,Y).</code></td>
</tr>
<tr>
<td><strong>4</strong> A solution is a coloring.</td>
<td><strong>3</strong> <code>#const n=3. iscol(1..n).</code></td>
</tr>
<tr>
<td></td>
<td><strong>4</strong> <code>#hide. #show color/2.</code></td>
</tr>
</tbody>
</table>
**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. \[ \text{color}(X,C) :- \text{iscol}(C), \text{node}(X), \text{not other}(X,C). \]
2. \[ \text{other}(X,C) :- \text{iscol}(C), \text{color}(X,D), D \neq C. \]
3. \[ :- \text{color}(X,C), \text{color}(Y,C), \text{edge}(X,Y). \]
4. \[ \#\text{const} n=3. \]
5. \[ \text{iscol}(1..n). \]
6. #hide.
7. #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. \[ \text{color}(X,C) :\neg \text{iscol}(C), \text{node}(X), \text{not other}(X,C). \]
   \[ \text{other}(X,C) :\neg \text{iscol}(C), \text{color}(X,D), D \neq C. \]
2. \[ \text{:- color}(X,C), \text{color}(Y,C), \text{edge}(X,Y). \]
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.
### Instance as Facts (in graph.lp)

```prolog
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).
```
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!

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

```text
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)
Graph Coloring

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

- Problem Instance
- Grounding
- Propositional Logic Program
- Stable Models
- Solving
- Grounder
- Problem Encoding
- Solver
Interlude: Answer Set(s) Computation

Problem Instance

Grounder

Grounding

Propositional Logic Program

Solver

Solving

Stable Models
Graph Coloring

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). ...

isco1(1). iscoop(2). isoco3(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
```

---

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Models</td>
<td>6</td>
</tr>
<tr>
<td>Time</td>
<td>0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)</td>
</tr>
<tr>
<td>CPU Time</td>
<td>0.000s</td>
</tr>
<tr>
<td>Choices</td>
<td>5</td>
</tr>
<tr>
<td>Conflicts</td>
<td>0</td>
</tr>
<tr>
<td>Restarts</td>
<td>0</td>
</tr>
<tr>
<td>Atoms</td>
<td>63</td>
</tr>
<tr>
<td>Rules</td>
<td>113 (1: 95 2: 12 3: 6)</td>
</tr>
<tr>
<td>Bodies</td>
<td>64</td>
</tr>
<tr>
<td>Equivalences</td>
<td>106 (Atom=Atom: 31 Body=Body: 6 Other: 69)</td>
</tr>
<tr>
<td>Tight</td>
<td>Yes</td>
</tr>
<tr>
<td>Variables</td>
<td>63 (Eliminated: 0 Frozen: 30)</td>
</tr>
<tr>
<td>Constraints</td>
<td>45 (Binary: 73.3% Ternary: 0.0% Other: 26.7%)</td>
</tr>
<tr>
<td>Lemmas</td>
<td>0 (Binary: 0.0% Ternary: 0.0% Other: 0.0%)</td>
</tr>
</tbody>
</table>
Problems as Logic Programs (Revisited)

Graph Coloring

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

```bash
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


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%)
```
Recall: a directed graph $G$

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

type{edge}(1,2). edge(1,3). edge(1,4).
type{edge}(2,4). edge(2,5). edge(2,6).
type{edge}(3,1). edge(3,4). edge(3,5).
type{edge}(4,1). edge(4,2). edge(5,3).
type{edge}(5,4). edge(5,6). edge(6,2).
type{edge}(6,3). edge(6,5).
**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).
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 \( \#\text{count}\{ \text{cycle}(X,Y) : \text{edge}(X,Y) \} \) 1 :- \text{node}(Y).
1 \( \#\text{count}\{ \text{cycle}(X,Y) : \text{edge}(X,Y) \} \) 1 :- \text{node}(X).
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).
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 $\#\text{count}\{ \text{cycle}(X,Y) : \text{edge}(X,Y) \} \ 1 \ :- \ \text{node}(Y)$.
1 $\#\text{count}\{ \text{cycle}(X,Y) : \text{edge}(X,Y) \} \ 1 \ :- \ \text{node}(X)$.
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).
```
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 ].
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).
```
Uniform Encoding (in \texttt{cycle.lp})

\begin{verbatim}
% 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.
\end{verbatim}
Problems as Logic Programs (Revisited)

Hamiltonian Cycle

Let's Run it!

```sh
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!

```bash
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)
Found: Hamiltonian cycle

Answer: 1

cycle(1,4)
cycle(4,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,1)
Found: Hamiltonian cycle

Answer: 1

cycle(1,4)
cycle(4,2)
cycle(2,6)
cycle(6,5)
cycle(5,3)
cycle(3,1)
Found: Hamiltonian cycle

Answer: 1

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

\[
\begin{align*}
\text{cost}(1,2,2). & \\
\text{cost}(1,3,3). & \text{ cost}(3,1,3). \\
\text{cost}(1,4,1). & \text{ cost}(4,1,1). \\
\text{cost}(2,4,2). & \text{ cost}(4,2,2). \\
\text{cost}(2,5,2). & \\
\text{cost}(2,6,4). & \text{ cost}(6,2,4). \\
\text{cost}(3,4,2). & \\
\text{cost}(3,5,2). & \text{ cost}(5,3,2). \\
\text{cost}(5,4,2). & \\
\text{cost}(5,6,1). & \text{ cost}(6,5,1). \\
\text{cost}(6,3,3). & \\
\end{align*}
\]
Mr Hamilton as Traveling Salesperson

Solution Optimization

Optimization Objective

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

Use \texttt{#minimize} (and/or \texttt{#maximize}) to associate each answer set with objective value(s).

Optimization Encoding

\%
OPTIMIZE

\texttt{#minimize[ cycle(X,Y) : cost(X,Y,C) = C@1 ]}.

Target: minimal sum of costs \( C \) (at priority level 1) associated with instances of \texttt{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 \texttt{#minimize} (and/or \texttt{#maximize}) to associate each answer set with objective value(s).

Optimization Encoding

\begin{verbatim}
% OPTIMIZE
#minimize[ cycle(X,Y) : cost(X,Y,C) = C@1 ].
\end{verbatim}

Target: \textbf{minimal sum} of costs \( C \) (at priority level 1) associated with instances of \texttt{cycle} in an answer set
Optimization Objective

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

Use \#\text{minimize} (and/or \#\text{maximize}) to associate each answer set with objective value(s).

Optimization Encoding

\%
\text{OPTIMIZE}
\#\text{minimize}\left[ \text{cycle}(X,Y) : \text{cost}(X,Y,C) = C@1 \right].

\textbf{Target: minimal sum of costs} \(C\) (at priority level 1) associated with instances of \text{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 \texttt{#minimize} (and/or \texttt{#maximize}) to associate each answer set with objective value(s).

Optimization Encoding

\begin{verbatim}
% OPTIMIZE
#minimize[ cycle(X,Y) : cost(X,Y,C) = C@1 ].
\end{verbatim}

Target: minimal sum of costs \( C \) \textbf{(at priority level 1)} associated with instances of \texttt{cycle} in an answer set.
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!

```
grindo 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 \(\subseteq (\text{Data + Deduction}) + (\text{Generation + Analysis}) + \text{Selection + Projection} \ [+ \text{Optimization}]\)
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
Problems as Logic Programs (Revisited)

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 \( \subseteq (\text{Data} + \text{Deduction}) + (\text{Generation} + \text{Analysis}) + \text{Selection} + \text{Projection} \) [\(+ \text{Optimization}\)]
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 \( \subseteq (\text{Data} + \text{Deduction}) + (\text{Generation} + \text{Analysis}) + \text{Selection} + \text{Projection} \) [+ 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]
Problems as Logic Programs (Revisited)

Traveling Salesperson

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]
Effective Modeling: Overview

28 Problems as Logic Programs (Revisited)
- Graph Coloring
- Hamiltonian Cycle
- Traveling Salesperson

29 Encoding Methodology
- Tweaking $N$-Queens
- Do’s and Don’t’s

30 Hints
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) \( \sim 8(\log_28) = 16 \) “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 \(\sim \frac{8!}{2} = 20,160\) “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) \(~ 8(\log_2 8) = 16 \) “operations”
- permutation guessing \(~ 8!/2 = 20,160 \) “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”
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
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

\[
\begin{array}{cccc}
4 & 3 & 2 & 1 \\
\hline \\
1 & 2 & 3 & 4 \\
\end{array}
\]

Placement

\[
\begin{array}{cccc}
4 & 3 & 2 & 1 \\
\hline \\
1 & 2 & 3 & 4 \\
\end{array}
\]
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.

```lp
 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.

```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.

```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, abs(X2-X1) = abs(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.
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 \texttt{queen} in an answer set.
4. We have to place (at least) \( N \) queens.

\begin{verbatim}
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.
\end{verbatim}
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!

```prolog
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 causes the problem!
**A First Refinement**

**Where Time Has Gone**

```
gringo -c n=122 queens_1.lp | wc
```

```
1241358 7402724 24950848
```

```
real 1m15.468s
user 1m15.980s
sys 0m0.090s
```

Grounding causes 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 causes the problem!
A First Refinement
Where Time Has Gone

\texttt{time(gringo \ -c n=122 \ queens\_1.lp \ | \ wc)}

1241358 7402724 24950848

real 1m15.468s
user 1m15.980s
sys 0m0.090s

Grounding causes 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 causes the problem!
Encoding Methodology  Tweaking $N$-Queens

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 causes the problem!
A First Refinement

Grounding Time $\sim$ Space

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

Grounding Time \sim Space

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.
Encoding Methodology  Tweaking $N$-Queens

A First Refinement
Grounding Time $\sim$ Space

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

Grounding Time ~ Space

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

Grounding Time $\sim$ Space

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.  \quad O(n \times n)
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.  \quad O(n \times n)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.  \quad O(n^2 \times n^2)

% DISPLAY
#hide. #show queen/2.
A First Refinement
Grounding Time \sim Space

\texttt{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

Grounding Time $\sim$ Space

```lp
queens_1.lp

% DOMAIN
#const n=4. square(1..n,1..n).  \( O(n \times n) \)

% GENERATE
0 #count{ queen(X,Y) } 1 :- square(X,Y).  \( O(n \times n) \)

% TEST
:- X = 1..n, not 1 #count{ queen(X,Y) } 1.  \( O(n \times n) \)
:- Y = 1..n, not 1 #count{ queen(X,Y) } 1.  \( O(n \times n) \)
:- queen(X1,Y1), queen(X2,Y2), X1 < X2, X2-X1 == |Y2-Y1|.  \( O(n^2 \times n^2) \)

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

Diagonals cause trouble!
Encoding Methodology

A First Refinement

Grounding Time $\sim$ Space

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

Grounding Time $\sim$ Space

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
Grounding Time \sim Space

\textbf{queens\_1.lp}

\begin{verbatim}
\% 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.
\end{verbatim}
A First Refinement

Grounding Time $\sim$ Space

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.

Diagonals cause trouble!
A Nomenclature for Diagonals

\[ N = 4 \]

\[ \#\text{diagonal}_1 = (\#\text{row} + \#\text{column}) - 1 \]

\[ \#\text{diagonal}_2 = (\#\text{row} - \#\text{column}) + N \]

#diagonal\(_{1/2}\) can be determined in this way for arbitrary \( N \).
A Nomenclature for Diagonals

\[ N = 4 \]

#diagonal_1 = 
(#row + #column) – 1

#diagonal_2 = 
(#row – #column) + N

#diagonal_1/2 can be determined in this way for arbitrary \( N \).
A Nomenclature for Diagonals

\[ N = 4 \]

#diagonal_1 = 
\[(\#\text{row} + \#\text{column}) - 1\]

#diagonal_2 = 
\[(\#\text{row} - \#\text{column}) + N\]

#diagonal_{1/2} can be determined in this way for arbitrary \( N \).
A Nomenclature for Diagonals

\(N = 4\)

\[
\begin{array}{cccc}
4 & 5 & 6 & 7 \\
3 & 4 & 5 & 6 \\
2 & 3 & 4 & 5 \\
1 & 2 & 3 & 4 \\
\end{array}
\]

\[
\begin{array}{cccc}
4 & 7 & 6 & 5 \\
3 & 6 & 5 & 4 \\
2 & 5 & 4 & 3 \\
1 & 4 & 3 & 2 \\
\end{array}
\]

\[
\begin{align*}
\text{#diagonal}_1 &= \frac{(\text{#row} + \text{#column}) - 1}{2} \\
\text{#diagonal}_2 &= \frac{(\text{#row} - \text{#column}) + N}{2}
\end{align*}
\]

\(\text{#diagonal}_{1/2}\) can be determined in this way for arbitrary \(N\).
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!

```bash
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!

```plaintext
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!

```lp
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!

```sh
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

```
geg(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  \(\times\) obsolete if properties change
2. use variable-sized conjunction (via ‘:’) \(\checkmark\) adapts to changing facts
3. use negation of complement \(\checkmark\) 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

\[
\begin{align*}
\text{veg}(\text{asparagus}). & \quad \text{veg}(\text{cucumber}). \\
\text{pro}(\text{asparagus}, \text{cheap}). & \quad \text{pro}(\text{cucumber}, \text{cheap}). \quad \text{pre}(\text{cheap}). \\
\text{pro}(\text{asparagus}, \text{fresh}). & \quad \text{pro}(\text{cucumber}, \text{fresh}). \quad \text{pre}(\text{fresh}). \\
\text{pro}(\text{asparagus}, \text{tasty}). & \quad \text{pro}(\text{cucumber}, \text{tasty}). \quad \text{pre}(\text{tasty}). \\
\text{pro}(\text{asparagus}, \text{clean}). & \quad \text{pre}(\text{clean}). \\
\end{align*}
\]

\[
\begin{align*}
\text{buy}(X) & :- \text{veg}(X), \text{not bye}(X). & \quad \text{bye}(X) & :- \text{veg}(X), \text{pre}(P), \text{not pro}(X,P).
\end{align*}
\]
Implementing Universal Quantification

**Goal:** identify objects such that ALL properties from a “list” hold

1. check all properties explicitly \(\times\) obsolete if properties change
2. use variable-sized conjunction (via `'\(')` \(\checkmark\) adapts to changing facts
3. use negation of complement \(\checkmark\) adapts to changing facts

**Example:** vegetables to buy

\[
\begin{align*}
\text{veg(} & \text{asparagus), veg(} \text{cucumber).} \\
\text{pro(} & \text{asparagus,cheap), pro(} \text{cucumber,cheap), pre(} \text{cheap).} \\
\text{pro(} & \text{asparagus,fresh), pro(} \text{cucumber,fresh), pre(} \text{fresh).} \\
\text{pro(} & \text{asparagus,tasty), pro(} \text{cucumber,tasty), pre(} \text{tasty).} \\
\text{pro(} & \text{asparagus,clean).} \\
\text{buy(} & \text{X) :- veg(} \text{X), not bye(} \text{X).} \\
\text{bye (} & \text{X) :- veg(} \text{X), pre(} \text{P), not pro(} \text{X,P).}
\end{align*}
\]
Running Example: Latin Square

**Given:** an $N \times N$ board

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

represented by facts:

- `square(1,1).`
- `square(2,1).`
- `square(3,1).`
- `square(4,1).`
- `square(5,1).`
- `square(6,1).`
- `square(1,2).`
- `square(2,2).`
- `square(3,2).`
- `square(4,2).`
- `square(5,2).`
- `square(6,2).`
- `square(1,3).`
- `square(2,3).`
- `square(3,3).`
- `square(4,3).`
- `square(5,3).`
- `square(6,3).`
- `square(1,4).`
- `square(2,4).`
- `square(3,4).`
- `square(4,4).`
- `square(5,4).`
- `square(6,4).`
- `square(1,5).`
- `square(2,5).`
- `square(3,5).`
- `square(4,5).`
- `square(5,5).`
- `square(6,5).`
- `square(1,6).`
- `square(2,6).`
- `square(3,6).`
- `square(4,6).`
- `square(5,6).`
- `square(6,6).`

**Wanted:** assignment of $1, \ldots, N$

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

represented by atoms:

- `num(1,1,1)`
- `num(1,2,2)`
- `num(1,3,3)`
- `num(1,4,4)`
- `num(1,5,5)`
- `num(1,6,6)`
- `num(2,1,2)`
- `num(2,2,3)`
- `num(2,3,4)`
- `num(2,4,5)`
- `num(2,5,6)`
- `num(2,6,1)`
- `num(3,1,3)`
- `num(3,2,4)`
- `num(3,3,5)`
- `num(3,4,6)`
- `num(3,5,1)`
- `num(3,6,2)`
- `num(4,1,4)`
- `num(4,2,5)`
- `num(4,3,6)`
- `num(4,4,1)`
- `num(4,5,2)`
- `num(4,6,3)`
- `num(5,1,5)`
- `num(5,2,6)`
- `num(5,3,1)`
- `num(5,4,2)`
- `num(5,5,3)`
- `num(5,6,4)`
- `num(6,1,6)`
- `num(6,2,1)`
- `num(6,3,2)`
- `num(6,4,3)`
- `num(6,5,4)`
- `num(6,6,5)`
Running Example: Latin Square

**Given:** an $N \times N$ board

\begin{align*}
\begin{array}{cccccc}
1 & & & & & \\
2 & & & & & \\
3 & & & & & \\
4 & & & & & \\
5 & & & & & \\
6 & & & & & \\
\end{array}
\end{align*}

represented by facts:

\begin{align*}
\text{square}(1,1). & \ldots \text{square}(1,6). \\
\text{square}(2,1). & \ldots \text{square}(2,6). \\
\text{square}(3,1). & \ldots \text{square}(3,6). \\
\text{square}(4,1). & \ldots \text{square}(4,6). \\
\text{square}(5,1). & \ldots \text{square}(5,6). \\
\text{square}(6,1). & \ldots \text{square}(6,6).
\end{align*}

**Wanted:** assignment of 1, \ldots, $N$

\begin{align*}
\begin{array}{cccccc}
1 & 1 & 2 & 3 & 4 & 5 & 6 \\
2 & 2 & 3 & 4 & 5 & 6 & 1 \\
3 & 3 & 4 & 5 & 6 & 1 & 2 \\
4 & 4 & 5 & 6 & 1 & 2 & 3 \\
5 & 5 & 6 & 1 & 2 & 3 & 4 \\
6 & 6 & 1 & 2 & 3 & 4 & 5 \\
\end{array}
\end{align*}

represented by atoms:

\begin{align*}
\text{num}(1,1,1) & \text{num}(1,2,2) \ldots \text{num}(1,6,6) \\
\text{num}(2,1,2) & \text{num}(2,2,3) \ldots \text{num}(2,6,1) \\
\text{num}(3,1,3) & \text{num}(3,2,4) \ldots \text{num}(3,6,2) \\
\text{num}(4,1,4) & \text{num}(4,2,5) \ldots \text{num}(4,6,3) \\
\text{num}(5,1,5) & \text{num}(5,2,6) \ldots \text{num}(5,6,4) \\
\text{num}(6,1,6) & \text{num}(6,2,1) \ldots \text{num}(6,6,5)
\end{align*}
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”

griingo latin_0.lp | wc
griingo 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

Torsten Schaub et al. (KRR@UP)  Modeling and Solving in ASP
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
grego 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”)
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
#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”)

gingo latin_2.lp | wc
2071560 12389384 40906946

gingo 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”)

g ringso latin_2.lp | wc
2071560 12389384 40906946

g ringso 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

grringo latin_3.lp | wc
1055752 6294536 21099558
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!
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!
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.

:- 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!

```bash
gringo latin_3.lp | wc
1055752 6294536 21099558
```

```bash
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.
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.
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.
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.
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

Torsten Schaub et al. (KRR@UP) Modeling and Solving in ASP
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.

griego latin_5.lp | wc
griego latin_6.lp | wc
304136 5778440 30252505 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.

grego latin_5.lp | wc
304136 5778440 30252505
grego 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.

 difficulté facile pour corriger une ligne/colonnes pleines !
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
Effective Modeling: Overview

28 Problems as Logic Programs (Revisited)
- Graph Coloring
- Hamiltonian Cycle
- Traveling Salesperson

29 Encoding Methodology
- Tweaking $N$-Queens
- Do’s and Dont’s

30 Hints
Hints

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) identical 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.
Hints

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) identical 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.
Hints

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) identical 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.
Hints

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) identical 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.
Hints

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) identical 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) identical ground rules?
   Q4: Do you enumerate pairs of values (to test uniqueness)?
   Q5: Do you assign dynamic aggregate values (to check a fixed bound)?
   Q6: Do you admit (obvious) symmetric solutions?
   Q7: Do you have additional domain knowledge simplifying the problem?
   Q8: Are you aware of anything else that, if encoded, would reduce grounding and/or solving efforts?

2. Revise until no “Yes” answer!
   - If the format of facts makes encoding painful (for instance, abusing grounding for “scientific calculations”), revise the fact format as well.
Some Hints on (Preventing) Debugging

Kinds of errors

- syntactic  
  ... follow error messages by the grounder
- semantic  
  ... (most likely) encoding/intention mismatch

Ways to identify semantic errors (early)

1. develop and test incrementally
   prepare toy instances with “interesting features”
   build the encoding bottom-up and verify additions (eg. new predicates)
2. compare the encoded to the intended meaning
   check whether the grounding fits (use gringo -t)
   if answer sets are unintended, investigate conditions that fail to hold
   if answer sets are missing, examine integrity constraints (add heads)
3. ask tools (eg. http://www.kr.tuwien.ac.at/research/projects/mmdasp/)
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 (e.g. 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 (e.g. 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 (e.g. 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 (e.g. http://www.kr.tuwien.ac.at/research/projects/mmdasp/)
Hints

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 (e.g., 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 (e.g., 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)

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/)
Hints

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 (e.g., `time(gringo [...] | wc)`)  
  2. profiling info (e.g., `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 (e.g. `time(gringo [...] | wc)`)  
  2. profiling info (e.g. `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 (e.g., `time(gringo [...] | wc)`) 
  2. profiling info (e.g., `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
Systems: Overview

31 Potassco
32 gringo
33 clasp
34 Siblings
   - claspfolio
   - clingcon
   - iclingo
   - oclingo
35 Book
Systems: Overview

Potassco

gringo

clasp

siblings
  - claspfolio
  - clingcon
  - iclingo
  - oclingo

Book
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**: claspfolio, coala, inca, plasp, sbass, xorro

**Benchmark repository**: [http://asparagus.cs.uni-potsdam.de](http://asparagus.cs.uni-potsdam.de)
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**: claspfolio, coala, inca, plasp, sbass, xorro

**Benchmark repository**: [http://asparagus.cs.uni-potsdam.de](http://asparagus.cs.uni-potsdam.de)
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**: claspfolio, coala, inca, plasp, sbass, xorro
- **Benchmark repository**: [http://asparagus.cs.uni-potsdam.de](http://asparagus.cs.uni-potsdam.de)
Systems: Overview

31 Potassco

32 gringo

33 clasp

34 Siblings
   ■ claspfolio
   ■ clingcon
   ■ iclingo
   ■ oclingo

35 Book
- Accepts safe programs with aggregates
- Tolerates unrestricted use of function symbols (as long as it yields a finite ground instantiation :)
- Expressive power of a Turing machine

Basic architecture of `gringo`:

```
Parser → Preprocessor → Grounder → Output

--ground

--lparse
--text
--reify
```
Systems: Overview

31 Potassco
32 gringo
33 clasp
34 Siblings
   - claspfolio
   - clingcon
   - iclingo
   - oclingo
35 Book
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
- Multi-threaded search
- ...
Reasoning modes of \textit{clasp}

Beyond deciding answer set existence, \textit{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

- and combinations thereof

\textit{clasp} also allows for solving

- propositional CNF formulas (extended \textit{dimacs})
- pseudo-Boolean formulas (\textit{opb} and \textit{wbo})
Reasoning modes of \textit{clasp}

Beyond deciding answer set existence, \textit{clasp} allows for:

- Optimization
- Enumeration \hspace{1cm} [without solution recording]
- Projective Enumeration \hspace{1cm} [without solution recording]
- Brave and Cautious Reasoning determining the
  - union or
  - intersection
  of all answer sets by computing only linearly many of them
- and combinations thereof

\textit{clasp} also allows for solving

- propositional CNF formulas (extended \textit{dimacs})
- pseudo-Boolean formulas (\textit{opb} and \textit{wbo})
Multi-threaded architecture of *clasp*

Diagram showing the architectural components and their interactions:

- **Preprocessing**
  - Program Builder
  - Preprocessor

- **Solver 1...n**
  - Conflict Resolution
  - Decision Heuristic
  - Assignment Atoms/Bodies

- **Coordination**
  - Shared Context
    - Propositional Variables
    - Atoms
    - Bodies
    - Static Nogoods
    - Implication Graph

- **Parallel Context**
  - Threads: $S_1, S_2, \ldots, S_n$
  - Counter: $T, W, \ldots, S$
  - Queue: $P_1, P_2, \ldots, P_n$

- **Enumerator**
  - Shared Nogoods

- **Recorded Nogoods**

- **Post Propagation**
  - Unit Propagation
  - Propagation

**Logic Program**
Multi-threaded architecture of *clasp*

- **Preprocessing**
  - Program Builder
  - Preprocessor

- **Logic Program**

- **Solver 1...n**
  - Decision Heuristic
  - Assignment Atoms/Bodies
  - Conflict Resolution
  - Recorded Nogoods
  - Propagation
    - Unit Propagation
    - Post Propagation

- **Coordination**
  - SharedContext
    - Propositional Variables
    - Atoms
    - Bodies
    - Static Nogoods
    - Implication Graph
  - Enumerator
    - ParallelContext
      - Threads: $S_1, S_2, \ldots, S_n$
      - Counter: $T, W, \ldots, S$
      - Queue: $P_1, P_2, \ldots, P_n$
  - Nogood Distributor
    - Shared Nogoods

- **Torsten Schaub et al. (KRR@UP)**
Multi-threaded architecture of clasp

Preprocessing
- Program Builder
- Preprocessor

Coordination
- SharedContext
  - Propositional Variables
  - Atoms
  - Bodies
  - Static Nogoods
  - Implication Graph
- Enumerator
- Nogood Distributor

ParallelContext
- Threads $S_1 S_2 \ldots S_n$
- Counter $T W \ldots S$
- Queue $P_1 P_2 \ldots P_n$

Solver 1...n
- Decision Heuristic
- Assignment Atoms/Bodies
- Conflict Resolution
- Recorded Nogoods
- Propagation
  - Unit Propagation
  - Post Propagation
Multi-threaded architecture of \textit{clasp}
Systems: Overview

- Potassco
- gringo
- clasp
- claspfolio
- clingcon
- iclingo
- oclingo

Book
Automatic selection of clasp configuration among 22 configuration via (learned) classifiers

Basic architecture of claspfolio:
Hybrid grounding and solving

Solving in hybrid domains, like Bio-Informatics

Basic architecture of clingcon:
Pouring Water into Buckets on a Scale

\[
\begin{align*}
\text{time}(0..t). & & \text{domain}(0..500). \\
\text{bucket}(a). & & \text{volume}(a,0) \leqslant 0. \\
\text{bucket}(b). & & \text{volume}(b,0) \leqslant 100. \\
\end{align*}
\]

\[
\begin{align*}
1 \{ \text{pour}(B,T) : \text{bucket}(B) \} & \leqslant 1 :- \text{time}(T), T < t. \\
1 \leqslant \text{amount}(B,T) & :- \text{pour}(B,T), T < t. \\
\text{amount}(B,T) \leqslant 30 & :- \text{pour}(B,T), T < t. \\
\text{amount}(B,T) \leqslant 0 & :- \text{not pour}(B,T), \text{bucket}(B), \text{time}(T), T < t. \\
\text{volume}(B,T+1) \leqslant \text{volume}(B,T) + \text{amount}(B,T) & :- \text{bucket}(B), \text{time}(T), T < t. \\
\text{down}(B,T) & :- \text{volume}(C,T) < \text{volume}(B,T), \text{bucket}(B;C), \text{time}(T). \\
\text{up}(B,T) & :- \text{not down}(B,T), \text{bucket}(B), \text{time}(T). \\
\end{align*}
\]
Pouring Water into Buckets on a Scale

time(0..t).
$domain(0..500).
bucket(a).
volume(a,0) $== 0.
bucket(b).
volume(b,0) $== 100.

\[
1 \{ \text{pour}(B,T) : \text{bucket}(B) \} \ 1 : \text{time}(T), T < t.
\]

\[
1 \leq \text{amount}(B,T) : \text{pour}(B,T), T < t.
\]

\[
\text{amount}(B,T) \leq 30 : \text{pour}(B,T), T < t.
\]

\[
\text{amount}(B,T) == 0 : \text{not pour}(B,T), \text{bucket}(B), \text{time}(T), T < t.
\]

\[
\text{volume}(B,T+1) == \text{volume}(B,T) + \text{amount}(B,T) : \text{bucket}(B), \text{time}(T), T < t.
\]

down(B,T) :- \text{volume}(C,T) < \text{volume}(B,T), \text{bucket}(B;C), \text{time}(T).
up(B,T) :- \text{not down}(B,T), \text{bucket}(B), \text{time}(T).

:- \text{up}(a,t).
iclingo

- Incremental grounding and solving
- Offline solving in dynamic domains, like Automated Planning
- Basic architecture of iclingo:

```
<table>
<thead>
<tr>
<th>gringo</th>
<th>clasp</th>
</tr>
</thead>
</table>
```

iclingo
Incremental ASP Solving Process

Logic Program → Grounder → Solver → Answer Sets

Modeling
Incremental ASP Solving Process

- Logic Program
- Grounder
- Solver
- Answer Sets

Modeling
Incremental ASP Solving Process

\[ \begin{align*}
B \\
Q_k \\
P_k
\end{align*} \rightarrow \text{Grounder} \rightarrow \text{Solver} \rightarrow \text{Answer Sets} \]

Modeling
Incremental ASP Solving Process
Incremental ASP Solving Process

Grounder → \( B \) → \( P_k \) → \( Q_k \) → Solver → Answer Sets
Incremental ASP Solving Process

- Grounder
  - $B$
  - $P_1$
  - $Q_1$

- Solver

- Answer Sets
Incremental ASP Solving Process

Grounder

\[ \begin{align*} B \\ P_1 \\ Q_1 \end{align*} \]

Solver

Answer Sets

- \( Q \)
- \( P \)
- \( B \)
Incremental ASP Solving Process

- **Grounder**
  - $B$
  - $P_1$
  - $Q_1$

- **Solver**

- **Answer Sets**
Incremental ASP Solving Process

Grounder → Solver

Grounder

Solver

Answer Sets

Modeling and Solving in ASP
Incremental ASP Solving Process

Grounder

-B
-P₁
-P₂
-Q₂

Solver

Answer Sets
Incremental ASP Solving Process

1. Grounder
   - $B$
   - $P_1$
   - $P_2$
   - $Q_2$

2. Solver

Answer Sets

Torsten Schaub et al. (KRR@UP) Modeling and Solving in ASP
Incremental ASP Solving Process

- Grounder
  - \( B \)
  - \( P_1 \)
  - \( P_2 \)

- Solver

- Answer Sets
Incremental ASP Solving Process

Grounder

\[ B \]
\[ P_1 \]
\[ P_2 \]
\[ P_3 \]
\[ Q_3 \]

Solver

Answer Sets

Torsten Schaub et al. (KRR@UP) Modeling and Solving in ASP
Incremental ASP Solving Process

- Grounder
  - $B$
  - $P_1$
  - $P_2$
  - $P_3$
  - $Q_3$

- Solver

Answer Sets
Incremental ASP Solving Process

- **Grounder**
  - \( B \)
  - \( P_1 \)
  - \( P_2 \)
  - \( P_3 \)

- **Solver**

- **Answer Sets**
Incremental ASP Solving Process

**Grounder**

\[ B \]

\[ P_1 \]

\[ P_2 \]

\[ P_3 \]

\[ \vdots \]

\[ P_n \]

\[ Q_n \]

**Solver**

**Answer Sets**
Incremental ASP Solving Process

1. **Grounder**
   - $B$
   - $P_1$
   - $P_2$
   - $P_3$
   - ... $P_n$
   - $Q_n$

2. **Solver**

3. **Answer Sets**
Simplistic STRIPS Planning

#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

#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).
oclingo

- Reactive grounding and solving
- Online solving in dynamic domains, like Robotics

- Basic architecture of oclingo:

  ![Diagram](image)

  - gringo
  - clasp
  - oclingo
  - Controller
Reactive ASP Solving Process
Reactive ASP Solving Process

1. Logic Program
2. Grounder
3. Solver
4. Answer Sets

Modeling
Reactive ASP Solving Process

Modeling

\[ B, P_k, Q_k \rightarrow \text{Grounder} \rightarrow \text{Solver} \rightarrow \text{Answer Sets} \]
Reactive ASP Solving Process

\[ B \quad P_k \quad Q_k \quad \text{Grounder} \quad \text{Solver} \quad \text{Answer Sets} \]
Reactive ASP Solving Process

- **Grounder**
  - $B$
  - $P_k$
  - $Q_k$

- **Solver**

- **Answer Sets**

The process involves grounding the input to the Solver, which then processes the grounded program to produce the answer sets.
Reacting ASP Solving Process

- Grounder
  - $B$
  - $P_k$
  - $Q_k$
- Solver
- Answer Sets
Reactive ASP Solving Process

Grounder

Solver

Answer Sets

$E_1$

$F_1$

$B$

$P_k$

$Q_k$

Update

Query

Erasure
Reactive ASP Solving Process

Grounder

Solver

Answer Sets

E₁, F₁

F₁

E₁

B

Pₖ

Qₖ

Update, Query, Erasure

Torsten Schaub et al. (KRR@UP)  Modeling and Solving in ASP
Reactive ASP Solving Process

Grounder

Solver

Answer Sets

E₁
F₁

F₁
E₁
B
P₁
...
Pₙ₁
Qₙ₁
Reactive ASP Solving Process

**Grounder**

- $F_1$
- $E_1$
- $B$
- $P_1$
- $P_{n_1}$
- $Q_{n_1}$

**Solver**

**Answer Sets**

- $E_2$
- $F_2$
- $E_1$
- $F_1$
Reactive ASP Solving Process
Reactive ASP Solving Process

Grounder

Solver

Answer Sets

E₁, F₁, E₂, F₂, ... Eₙ₂, Fₙ₂, P₁, B

F₂, E₂, E₁, F₁

Qₙ₂, Pₙ₂

Torsten Schaub et al. (KRR@UP) Modeling and Solving in ASP
Reactive ASP Solving Process

Grounder

Solver

Answer Sets
Reactive ASP Solving Process

Grounder

Solver

Answer Sets

\[ E_3 \]
\[ E_2 \]
\[ E_1 \]
\[ B \]
\[ P_1 \]
\[ \ldots \]
\[ P_{n_2} \]
\[ Q_{n_2} \]
Reactive ASP Solving Process

Grounder

Solver

Answer Sets

$E_3$ $E_2$ $E_1$

$F_3$ $F_2$ $F_1$

$P_1$

$B$

$Q_{n_3}$ $P_{n_3}$

$F_3$ $E_3$

$F_2$ $E_2$

$F_1$ $E_1$

$E_1$ $E_2$ $E_3$
Reactive ASP Solving Process

Grounder

Solver

Answer Sets

$P_{n41}$

$Q_{n41}$

$E_{42}$

$F_{42}$

$E_{41}$

$F_{41}$

$E_{40}$

$F_{40}$

$E_{39}$

$F_{39}$

$E_{38}$

$F_{38}$

$E_{37}$

$F_{37}$

$E_{36}$

$F_{36}$

$E_{35}$

$F_{35}$

$E_{34}$

$F_{34}$

$E_{33}$

$F_{33}$

$E_{32}$

$F_{32}$

$E_{31}$

$F_{31}$

$E_{30}$

$F_{30}$

$E_{29}$

$F_{29}$

$E_{28}$

$F_{28}$

$E_{27}$

$F_{27}$

$E_{26}$

$F_{26}$

$E_{25}$

$F_{25}$

$E_{24}$

$F_{24}$

$E_{23}$

$F_{23}$

$E_{22}$

$F_{22}$

$E_{21}$

$F_{21}$

$E_{20}$

$F_{20}$

$E_{19}$

$F_{19}$

$E_{18}$

$F_{18}$

$E_{17}$

$F_{17}$

$E_{16}$

$F_{16}$

$E_{15}$

$F_{15}$

$E_{14}$

$F_{14}$

$E_{13}$

$F_{13}$

$E_{12}$

$F_{12}$

$E_{11}$

$F_{11}$

$E_{10}$

$F_{10}$

$E_{9}$

$F_{9}$

$E_{8}$

$F_{8}$

$E_{7}$

$F_{7}$

$E_{6}$

$F_{6}$

$E_{5}$

$F_{5}$

$E_{4}$

$F_{4}$

$E_{3}$

$F_{3}$

$E_{2}$

$F_{2}$

$E_{1}$

$F_{1}$

$E_{0}$

$F_{0}$

$E_{42}$

$F_{42}$

$E_{41}$

$F_{41}$

$E_{40}$

$F_{40}$

$E_{39}$

$F_{39}$

$E_{38}$

$F_{38}$

$E_{37}$

$F_{37}$

$E_{36}$

$F_{36}$

$E_{35}$

$F_{35}$

$E_{34}$

$F_{34}$

$E_{33}$

$F_{33}$

$E_{32}$

$F_{32}$

$E_{31}$

$F_{31}$

$E_{30}$

$F_{30}$

$E_{29}$

$F_{29}$

$E_{28}$

$F_{28}$

$E_{27}$

$F_{27}$

$E_{26}$

$F_{26}$

$E_{25}$

$F_{25}$

$E_{24}$

$F_{24}$

$E_{23}$

$F_{23}$

$E_{22}$

$F_{22}$

$E_{21}$

$F_{21}$

$E_{20}$

$F_{20}$

$E_{19}$

$F_{19}$

$E_{18}$

$F_{18}$

$E_{17}$

$F_{17}$

$E_{16}$

$F_{16}$

$E_{15}$

$F_{15}$

$E_{14}$

$F_{14}$

$E_{13}$

$F_{13}$

$E_{12}$

$F_{12}$

$E_{11}$

$F_{11}$

$E_{10}$

$F_{10}$

$E_{9}$

$F_{9}$

$E_{8}$

$F_{8}$

$E_{7}$

$F_{7}$

$E_{6}$

$F_{6}$

$E_{5}$

$F_{5}$

$E_{4}$

$F_{4}$

$E_{3}$

$F_{3}$

$E_{2}$

$F_{2}$

$E_{1}$

$F_{1}$

$E_{0}$

$F_{0}
Reactive ASP Solving Process

Grounder

Solver

Answer Sets

$E_1$

$B$

$P_1$

$P_{n42}$

$Q_{n42}$

$E_{42}$

$E_{41}$

$E_{40}$

$E_{39}$

$E_{38}$

$E_{37}$

$E_{36}$

$B$

$F_{42}$

$F_{41}$

$F_{40}$

$F_{39}$

$F_{38}$

$F_{37}$

$F_{36}$

$F$

$E$

$F$
Reactive ASP Solving Process

Grounder

Solver

Update

Answer Sets

Torsten Schaub et al. (KRR@UP)  Modeling and Solving in ASP
Reactive ASP Solving Process

Grounder

Solver

Answer Sets

Query

Modeling and Solving in ASP

Torsten Schaub et al. (KRR@UP)
Reactive ASP Solving Process

Grounder

Solver

Answer Sets

Erasure

$E_{42}$
$F_{42}$

$E_{41}$
$F_{41}$

$E_{40}$
$F_{40}$

$E_{39}$
$F_{39}$

$E_{38}$
$F_{38}$

$E_{37}$
$F_{37}$

$E_{36}$
$F_{36}$

$B$
Elevator Control

#base.

floor(1..3).
atFloor(1,0).

#cumulative t.
#external request(F,t) : floor(F).

1 { atFloor(F-1,F+1,t) } 1 :- atFloor(F,t-1), floor(F).
:- atFloor(F,t), not floor(F).
requested(F,t) :- request(F,t), floor(F), not atFloor(F,t).
requested(F,t) :- requested(F,t-1), floor(F), not atFloor(F,t).
goal(t) :- not requested(F,t) : floor(F).

#volatile t.

:- not goal(t).
Elevator Control

#base.

floor(1..3).
atFloor(1,0).

#cumulative t.
#external request(F,t) : floor(F).

1 { atFloor(F-1;F+1,t) } 1 :- atFloor(F,t-1), floor(F).
:- atFloor(F,t), not floor(F).
requested(F,t) :- request(F,t), floor(F), not atFloor(F,t).
requested(F,t) :- requested(F,t-1), floor(F), not atFloor(F,t).
goal(t) :- not requested(F,t) : floor(F).

#volatile t.

:- not goal(t).
Pushing a button

- oClingo acts as a server listening on a port waiting for client requests
- To issue such requests, a separate controller program sends online progressions using network sockets
- For instance,
  
  #step 1.
  request(3,1).
  #endstep.
- This process terminates when the client sends #stop.
Pushing a button

- oClingo acts as a server listening on a port waiting for client requests.
- To issue such requests, a separate controller program sends online progressions using network sockets.
- For instance,
  
  #step 1.
  request(3,1).
  #endstep.
- This process terminates when the client sends #stop.
Pushing a button

- oClingo acts as a server listening on a port waiting for client requests
- To issue such requests, a separate controller program sends online progressions using network sockets
- For instance,
  
  ```
  #step 1.
  request(3,1).
  #endstep.
  ```
- This process terminates when the client sends
  
  ```
  #stop.
  ```
Pushing a button

- oClingo acts as a server listening on a port waiting for client requests.
- To issue such requests, a separate controller program sends online progressions using network sockets.
- For instance,
  
  ```
  #step 1.
  request(3, 1).
  #endstep.
  ```

- This process terminates when the client sends #stop.
Systems: Overview

31 Potassco
32 gringo
33 clasp
34 Siblings
   - claspfolio
   - clingcon
   - iclingo
   - oclingo
35 Book
The (forthcoming) Potassco Book

1. Motivation
2. Introduction
3. Basic modeling
4. Grounding
5. Characterizations
6. Solving
7. Systems
8. Advanced modeling
9. Conclusions

http://potassco.sourceforge.net/teaching.html
Summary

- ASP is emerging as a viable tool for Knowledge Representation and Reasoning
- ASP offers efficient and versatile off-the-shelf solving technology
  - ASP’07/09/11, CASC’11, MISC’11, PB’09/11, and SAT’09/11
- ASP offers an expanding functionality and ease of use
  - Rapid application development tool
- ASP has a growing range of applications

\[
ASP = KR + DB + SAT + LP
\]


C. Baral, G. Brewka, and J. Schlipf, editors.
S. Baselice, P. Bonatti, and M. Gelfond.
Towards an integration of answer set and constraint solving.

A. Biere.
Adaptive restart strategies for conflict driven SAT solvers.

A. Biere.
PicoSAT essentials.


M. Brain, O. Cliffe, and M. de Vos.
A pragmatic programmer’s guide to answer set programming.

Debugging ASP programs by means of ASP.
In Baral et al. , pages 31–43.

That is illogical captain! — the debugging support tool spock for answer-set programs: System description.

K. Clark.

Negation as failure.

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 , pages 724–728.

M. D’Agostino, D. Gabbay, R. Hähnle, and J. Posegga, editors.

Handbook of Tableau Methods.
E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov.  
Complexity and expressive power of logic programming.  

M. Davis, G. Logemann, and D. Loveland.  
A machine program for theorem-proving.  

M. Davis and H. Putnam.  
A computing procedure for quantification theory.  

Conflict-driven disjunctive answer set solving.
C. Drescher, M. Gebser, B. Kaufmann, and T. Schaub. 
**Heuristics in conflict resolution.**

N. Eén and N. Sörensson. 
**An extensible SAT-solver.**

T. Eiter and G. Gottlob.
On the computational cost of disjunctive logic programming: Propositional case.

F. Fages.
Consistency of Clark’s completion and the existence of stable models.

P. Ferraris.
Answer sets for propositional theories.

M. Fitting.
A Kripke-Kleene semantics for logic programs.

M. Garcia de la Banda and E. Pontelli, editors.

Repair and prediction (under inconsistency) in large biological networks with answer set programming.


A user’s guide to gringo, clasp, clingo, and iclingo.


Engineering an incremental ASP solver.
In Garcia de la Banda and Pontelli, pages 190–205.


A portfolio solver for answer set programming: Preliminary report.

M. Gebser, R. Kaminski, and T. Schaub.
Complex optimization in answer set programming.

M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
clap: A conflict-driven answer set solver.
In Baral et al., pages 260–265.

M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.
Conflict-driven answer set enumeration.
In Baral et al., pages 136–148.


M. Gebser, B. Kaufmann, and T. Schaub. Solution enumeration for projected Boolean search problems.
M. Gebser, M. Ostrowski, and T. Schaub.
Constraint answer set solving.
In Hill and Warren, pages 235–249.

A meta-programming technique for debugging answer-set programs.

M. Gebser and T. Schaub.
Tableau calculi for answer set programming.
In S. Etalle and M. Truszczynski, editors, Proceedings of the Twenty-second International Conference on Logic Programming...
M. Gebser and T. Schaub.
**Generic tableaux for answer set programming.**

M. Gelfond.
**Answer sets.**

M. Gelfond and N. Leone.
**Logic programming and knowledge representation — the A-Prolog perspective.**
M. Gelfond and V. Lifschitz.
The stable model semantics for logic programming.

M. Gelfond and V. Lifschitz.
Logic programs with classical negation.

E. Giunchiglia, Y. Lierler, and M. Maratea.
Answer set programming based on propositional satisfiability.

P. Hill and D. Warren, editors.
J. Huang.
The effect of restarts on the efficiency of clause learning.
In Veloso, pages 2318–2323.

H. Kautz and B. Selman.
Planning as satisfiability.

K. Konczak, T. Linke, and T. Schaub.
Graphs and colorings for answer set programming.

R. Kowalski.
Logic for data description.

J. Lee.
A model-theoretic counterpart of loop formulas.

N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello.
The DLV system for knowledge representation and reasoning.

V. Lifschitz.
Answer set programming and plan generation.

V. Lifschitz and A. Razborov.
Why are there so many loop formulas?

V. Lifschitz, L. Tang, and H. Turner.
Nested expressions in logic programs.

F. Lin and Y. Zhao. 
**ASSAT**: computing answer sets of a logic program by SAT solvers. 

J. Lloyd. 
*Foundations of Logic Programming*. 

V. Marek and M. Truszczyński. 
Stable models and an alternative logic programming paradigm. 

Conflict-driven clause learning SAT solvers. 
In Biere et al., chapter 4, pages 131–153.
J. Marques-Silva and K. Sakallah.
GRASP: A search algorithm for propositional satisfiability.

V. Mellarkod and M. Gelfond.
Integrating answer set reasoning with constraint solving techniques.

V. Mellarkod, M. Gelfond, and Y. Zhang.
Integrating answer set programming and constraint logic programming.

D. Mitchell.
A SAT solver primer.
M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik.
Chaff: Engineering an efficient SAT solver.

I. Niemelä.
Logic programs with stable model semantics as a constraint programming paradigm.

Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T).

J. Oetsch, J. Pührer, and H. Tompits.
Catching the ouroboros: On debugging non-ground answer-set programs.

K. Pipatsrisawat and A. Darwiche.
A lightweight component caching scheme for satisfiability solvers.

L. Ryan.
Efficient algorithms for clause-learning SAT solvers.

J. Schlipf.
The expressive powers of the logic programming semantics.
Extending and implementing the stable model semantics.

T. Syrjänen.
Lparse 1.0 user’s manual.

A. van Gelder, K. Ross, and J. Schlipf.
The well-founded semantics for general logic programs.

M. Veloso, editor.

L. Zhang, C. Madigan, M. Moskewicz, and S. Malik.
Efficient conflict driven learning in a Boolean satisfiability solver.