A practical introduction to Answer Set Programming

ENSEIRB — 2024-2025

Clémence Frioux
Inria
clemence.frioux@inria.fr

Organisation

- Objectives
- Roadmap
- Resources & Literature
- Systems

What are we going to talk about?

```
takeUmbrella :- rain, not stayAtHome.
rain.
```

What are we going to talk about?

```
takeUmbrella :- rain, not stayAtHome.
rain.
```

```
rain.  takeUmbrella.
```

What are we going to talk about?

```
takeUmbrella :- rain, not stayAtHome.
rain.
```

```
rain.  takeUmbrella.
```

```
s("A",5). s("B",2). s("A",9). s("C",7). s("C",2).
{x(M,N) : s(M,N)}.
:- s(R,_), not x(R,_).
#minimize {B,A:x(A,B) }.
#show x/2.
```

What are we going to talk about?

```
takeUmbrella :- rain, not stayAtHome.
rain.
```

```
rain.  takeUmbrella.
```

```
s("A",5). s("B",2). s("A",9). s("C",7). s("C",2).
{x(M,N) : s(M,N)}.
:- s(R,_), not x(R,_).
#minimize {B,A:x(A,B) }.
#show x/2.
```

```
x("A",5) x("B",2) x("C",2).
```

Objectives of the course

- Discover a logic programming paradigm
- Understand the concept of ASP
- Learn ASP syntax and understand it through examples
- Use Clingo to solve ASP programs
- Model problems
- Design optimisations
- Applications to AI

In practice

Main ASP solver is Clingo

Online solver
https://potassco.org/clingo/run/

Conda installation (compiled with Python)

```
conda install -c conda-forge clingo
```

Python `pip` installation + Jupyter [doc]

```
pip install clyngor_with_clingo
pip install notebook
# and to use clingo in Jupyter notebooks
IPYLOC=$(ipython locate profile)
echo 'alias_magic clingo script -p "clingo
--no-raise-error"' >> $IPYLOC/startup/config.ipy
```

Roadmap

Credits & Teaching resources

This course is a foretaste of the great collection of teaching resources designed by the KRR group (led by Torsten Schaub) at the University of Potsdam. Some contents originate from the course material, some are adapted from the original teaching resources, some are my own.

Potassco is the Potsdam Answer Set Solving Collection.

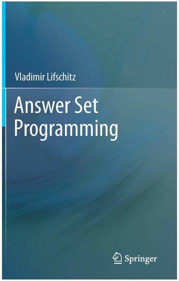
Visit potassco.org/teaching for additional content: videos, tutorials, slides...

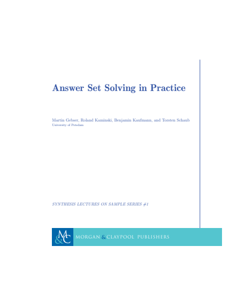
Potassco

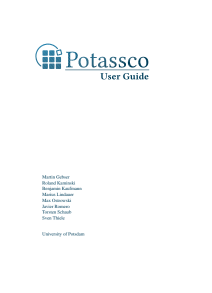
Resources

- A (short) paper on ASP: Lifschitz et al 2008.
- Course material
  - `https://github.com/potassco-asp-course`
  - `https://potassco.org/teaching`
- Videos
  - `https://youtube.com/c/potassco-live`
- Mailing lists
  - `https://sourceforge.net/projects/potassco/lists/potassco-users`
  - `https://sourceforge.net/projects/potassco/lists/potassco-announce`

The ASP and Potassco Book, and Guide







Resources

- http://potassco.org/book
- http://potassco.org/teaching

Literature

- Books [5], [25], [32], [38], [43]
- Surveys [40], [31], [19], [10], [36], [47]
- Magazines [9], [48]
- Articles [35], [34], [7], [45], [44], [39], [33], [23], etc.
- Guide [29]

- + bibliography on Github

Systems

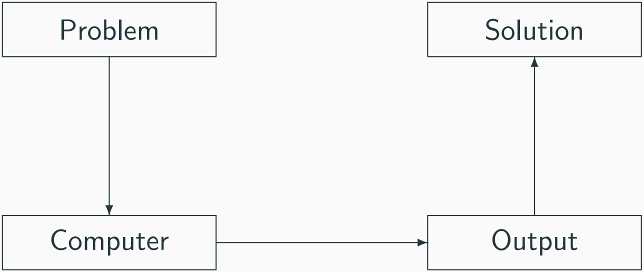
- Systems
  - *clingo* [27] https://potassco.org
  - *dlv* [37, 2] http://www.dlvsystem.com
- Grounders
  - *lparse* [50]
  - *gringo* [22, 28]+[24, 12] https://potassco.org
  - *idlv* [14]+[12] http://www.dlvsystem.com
- Solvers
  - *smodels* [46, 49]
  - *clasp* [26, 21] https://potassco.org
  - *wasp* [4] https://www.mat.unical.it/ricca/wasp
- Encodings
  - *asparagus* [8] https://asparagus.cs.uni-potsdam.de
  - competitions [30, 18, 16, 3, 15]

Motivation

- Traditional and declarative programming
- Nutshell
- Foundation
- Workflow and usage

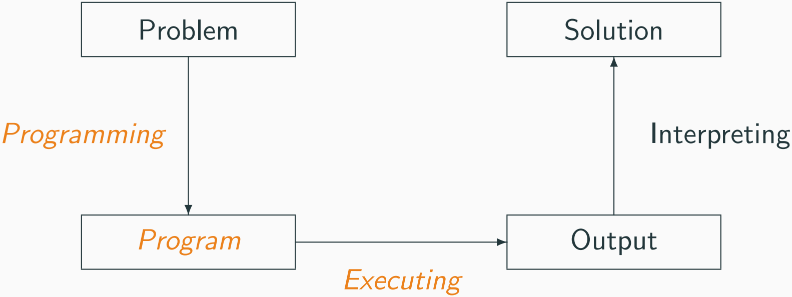
Traditional programming

“How to solve the problem?”



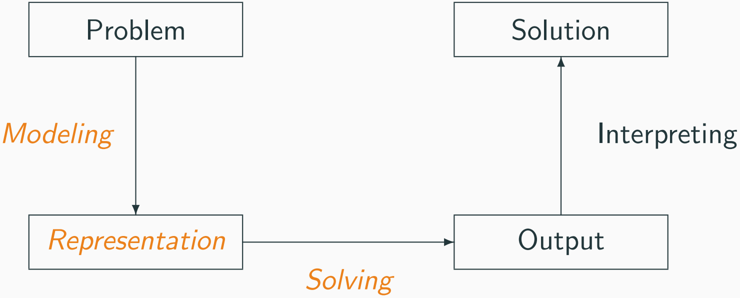
Traditional programming

“How to solve the problem?”

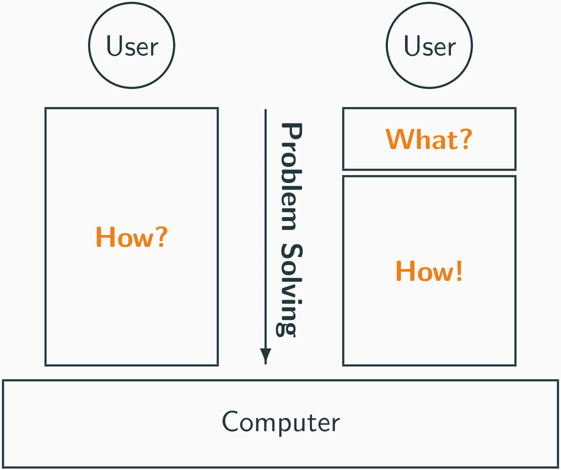


Declarative programming & problem solving

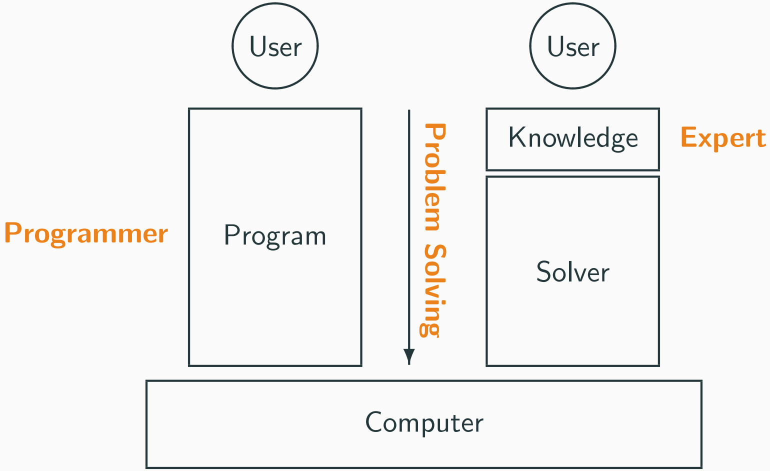
"What is the problem?"



From traditional to knowledge-driven software



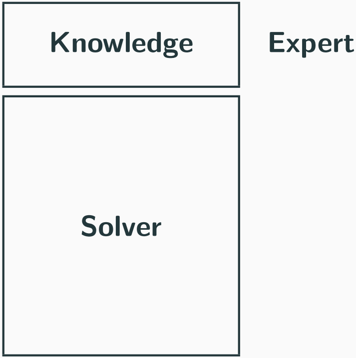
From traditional to knowledge-driven software



What is the benefit?

- \+ Transparency
- \+ Flexibility
- \+ Maintainability
- \+ Reliability

- \+ Generality
- \+ Efficiency
- \+ Optimality
- \+ Availability



Answer Set Programming (ASP)

- What is ASP?

  ASP is an approach for declarative problem solving

Answer Set Programming (ASP)

- What is ASP? **ASP = DB+LP+KR+SAT !**

  ASP is an approach for declarative problem solving
- Where is ASP from?
  - Databases
  - Logic programming
  - Knowledge representation and reasoning
  - Satisfiability solving

Answer Set Programming (ASP)

- What is ASP? **ASP = DB+LP+KR+SAT !**

  ASP is an approach for declarative problem solving

- What is ASP good for?

  Solving knowledge-intense combinatorial (optimization) problems

Answer Set Programming (ASP)

- What is ASP? **ASP = DB+LP+KR+SAT !**

  ASP is an approach for declarative problem solving
- What is ASP good for?

  Solving knowledge-intense combinatorial (optimization) problems
- What problems are these? — **And industrial ones**

  Problems consisting of (many) decisions and constraints

Answer Set Programming (ASP)

- What is ASP? **ASP = DB+LP+KR+SAT !**

  ASP is an approach for declarative problem solving
- What is ASP good for?

  Solving knowledge-intense combinatorial (optimization) problems
- What problems are these? — **And industrial ones**

  Problems consisting of (many) decisions and constraints

  Examples Sudoku, Configuration, Diagnosis, Music composition, Planning, System design, Time tabling, etc.

Answer Set Programming (ASP)

- What is ASP? **ASP = DB+LP+KR+SAT !**

  ASP is an approach for declarative problem solving
- What is ASP good for?

  Solving knowledge-intense combinatorial (optimization) problems
- What problems are these? — **And industrial ones**

  Problems consisting of (many) decisions and constraints
- What are ASP's distinguishing features?
  - High level, versatile modeling language
  - High performance solvers
  - Qualitative and quantitative optimization

Answer Set Programming (ASP)

- What is ASP? **ASP = DB+LP+KR+SAT !**

  ASP is an approach for declarative problem solving
- What is ASP good for?

  Solving knowledge-intense combinatorial (optimization) problems
- What problems are these? — **And industrial ones**

  Problems consisting of (many) decisions and constraints
  - Debian, Ubuntu: Linux package configuration
  - Exeura: Call routing
  - Fcc: Radio frequency auction
  - Gioia Tauro: Workforce management
  - Nasa: Decision support for Space Shuttle
  - Siemens: Partner units configuration
  - Variantum: Product configuration
  - US Navy: risk assessment

Answer Set Programming (ASP)

- What is ASP? **ASP = DB+LP+KR+SAT !**

  ASP is an approach for declarative problem solving
- What is ASP good for?

  Solving knowledge-intense combinatorial (optimization) problems
- What problems are these? — **And industrial ones**

  Problems consisting of (many) decisions and constraints
- Any industrial impact?
  - ASP Tech companies: DLV Systems and Potassco Solutions
  - Increasing interest in (large) companies

Answer Set Programming (ASP)

- What is ASP? **ASP = DB+LP+KR+SAT !**
  ASP is an approach for declarative problem solving
- What is ASP good for?
  Solving knowledge-intense combinatorial (optimization) problems
- What problems are these? — **And industrial ones**
  Problems consisting of (many) decisions and constraints
- Any industrial impact?
  - ASP Tech companies: DLV Systems and Potassco Solutions
  - Increasing interest in (large) companies
- Anything not so good for ASP?
  - Number crunching

Logic programs

- A logic program is a set of rules of the form

$$\underbrace{a}_{\text{head}} \leftarrow \underbrace{b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_n}_{\text{body}}$$

where

  - a and all b_i, c_j are atoms (propositional variables)
  - $\leftarrow$, $,$, $\neg$ denote if, and, and negation
  - intuitive reading: head must be true if body holds

- Semantics given by stable models, informally,
  1. (classical) models of the logic program
  2. requiring that each true atom is provable

Closed world assumption

Logic programs

- A logic program is a set of rules of the form

$$\underbrace{a}_{\text{head}} \leftarrow \underbrace{b_1, \dots, b_m, \neg c_1, \dots, \neg c_n}_{\text{body}}$$

where

  - a and all b_i, c_j are atoms (propositional variables)
  - $\leftarrow$, $,$, $\neg$ denote if, and, and negation
  - intuitive reading: head must be true if body holds

- Semantics given by stable models, informally,
  1. (classical) models of the logic program
  2. requiring that each true atom is provable

Closed world assumption

Logic programs

- A logic program is a set of rules of the form

$$\underbrace{a}_{\text{head}} \leftarrow \underbrace{b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_n}_{\text{body}}$$

where

  - a and all b_i, c_j are atoms (propositional variables)
  - $\leftarrow$, $,$, $\neg$ denote if, and, and negation
  - intuitive reading: head must be true if body holds

- Semantics given by stable models, informally,
  1. (classical) models of the logic program
  2. requiring that each true atom is provable

Closed world assumption

Logic programs

- A logic program is a set of rules of the form

$$\underbrace{a}_{\text{head}} \leftarrow \underbrace{b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_n}_{\text{body}}$$

  where

  - a and all b_i, c_j are atoms (propositional variables)
  - $\leftarrow$, $,$, $\neg$ denote if, and, and negation
  - intuitive reading: head must be true if body holds

- Semantics given by stable models, informally,
  1. (classical) models of the logic program
  2. requiring that each true atom is provable

Closed world assumption

Closed world assumption

- Stable model semantics can be seen as a logic for reasoning under the Closed world assumption (CWA)
- In interpretations following the spirit of "classical" logic, an unknown proposition can be true or false
- Here: unknown information is treated as false
- This is a pretty "human" way, also referred to as *commonsense reasoning*
- e.g. bus stop, where a non-listed, and thus unknown timing, is interpreted as "false", in the sense that no bus is supposed to arrive
- Conclusions can change their truth value upon the arrival of new information

Default negation and strong negation

Default negation - Negation as failure
`p` does not not belong to the set and its negation either.

```
cross :- not train.
```

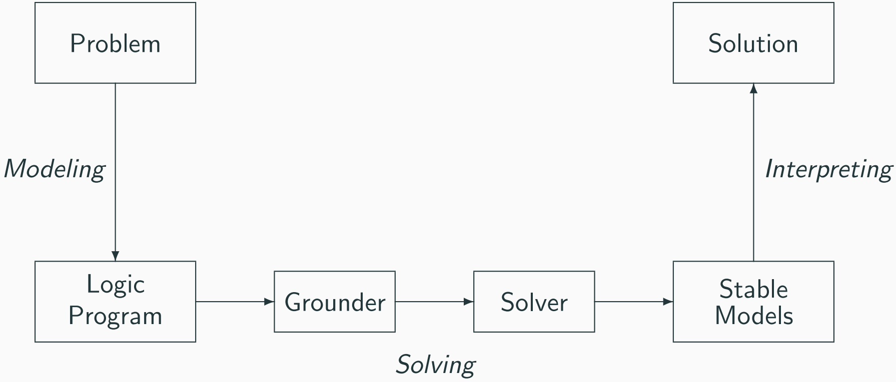
Answer set: `cross`

Strong negation
Also called "classical" negation. Denoted in the language by `-`.

```
cross :- - train.
```

Answer set: $\varnothing$

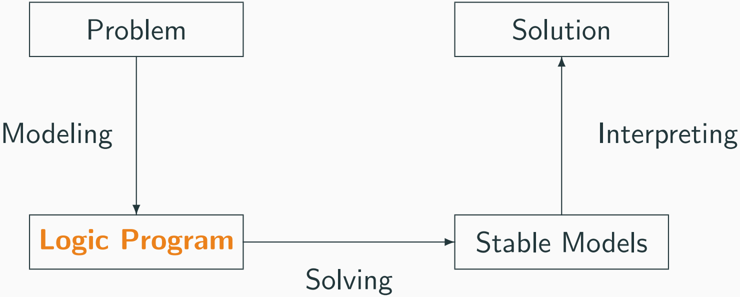
Modeling, grounding, and solving



Introduction to ASP

- Syntax
- Semantics: models and stable models
- Language
- Grounding and variables
- Summary

Syntax


Problem
Solution
Modeling
Interpreting
Logic Program
Stable Models
Solving

Atoms and terms

- Atoms
  - An atom is the elementary construct for representing knowledge
  - An atom represents a relation between objects
  - Examples
    - `answer(42)`
    - `bachelor(friend(joe))`
    - `hot`
  - An atom can be either *true* or *false*
- Terms
  - Terms are the subatomic components of atoms
  - Terms represent objects
  - Examples
    - 42
    - `friend(joe)`, `joe`

Atoms and terms

- Atoms
  - An atom is the elementary construct for representing knowledge
  - An atom represents a relation between objects
  - Examples
    - `answer(42)`
    - `bachelor(friend(joe))`
    - `hot`
  - An atom can be either *true* or *false*
- Terms
  - Terms are the subatomic components of atoms
  - Terms represent objects
  - Examples
    - `42`
    - `friend(joe)`, `joe`

Normal logic programs

- A logic program, P, over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, r, is of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$$

where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$

Normal logic programs

- A logic program, P, over a set $\mathcal{A}$ of atoms is a finite set of rules
  - $\mathcal{A}$ is also called the alphabet (or signature) of P
- A (normal) rule, r, is of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$$

where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$

Normal logic programs

- A logic program, P, over a set $\mathcal{A}$ of atoms is a finite set of rules
  - $\mathcal{A}$ is also called the alphabet (or signature) of P
- A (normal) rule, r, is of the form

$$\underbrace{a_0}_{\text{head}} \leftarrow \underbrace{a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n}_{\text{body}}$$

where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$

Normal logic programs

- A logic program, P, over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, r, is of the form

$$a_0 \leftarrow a_1, \dots, a_m, \neg a_{m+1}, \dots, \neg a_n$$

where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$

Normal logic programs

- A logic program, P, over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, r, is of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$$

  where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$
- A literal is an atom or a negated atom

Normal logic programs

- A logic program, P, over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, r, is of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$$

  where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$
- A literal is an atom or a negated atom
- Note A body is a (finite) set of literals

Normal logic programs

- A logic program, P, over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, r, is of the form

$$\mathtt{a_0\ \text{:-}\ a_1,\ldots,a_m,\ not\ a_{m+1},\ldots,\ not\ a_n.}$$

  where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$
- A literal is an atom or a negated atom
- Note A body is a (finite) set of literals

Normal logic programs

- A logic program, P, over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, r, is of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$$

where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$

Normal logic programs

- A logic program, P, over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, r, is of the form

$$a_0 \leftarrow a_1, \dots, a_m, \neg a_{m+1}, \dots, \neg a_n$$

  where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$

- Notation

$$\begin{aligned} h(r) &= a_0 \\ B(r) &= \{a_1, \dots, a_m, \neg a_{m+1}, \dots, \neg a_n\} \end{aligned}$$

Normal logic programs

- A logic program, P, over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, r, is of the form

$$\underbrace{a_0}_{h(r)} \leftarrow \underbrace{a_1, \dots, a_m, \neg a_{m+1}, \dots, \neg a_n}_{B(r)}$$

  where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$
- Notation

$$\begin{aligned} h(r) &= a_0 \\ B(r) &= \{a_1, \dots, a_m, \neg a_{m+1}, \dots, \neg a_n\} \end{aligned}$$

Normal logic programs

- A logic program, P, over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, r, is of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$$

  where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$

- Notation

$$\begin{aligned} h(r) &= a_0 \\ B(r) &= \{a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n\} \end{aligned}$$

Normal logic programs

- A logic program, P, over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, r, is of the form

$$a_0 \leftarrow a_1, \dots, a_m, \neg a_{m+1}, \dots, \neg a_n$$

  where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$
- Notation

$$\begin{aligned} H(r) &= \{a_0\} \\ B(r) &= \{a_1, \dots, a_m, \neg a_{m+1}, \dots, \neg a_n\} \end{aligned}$$

Normal logic programs

- A logic program, P, over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, r, is of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$$

  where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$
- Notation

$$\begin{aligned} h(r) &= a_0 \\ B(r) &= \{a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n\} \\ H(P) &= \{h(r) \mid r \in P\} \\ B(P) &= \{B(r) \mid r \in P\} \end{aligned}$$

Normal logic programs

- A logic program, P, over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, r, is of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$$

  where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$
- Notation

$$\begin{aligned} h(r) &= a_0 \\ B(r)^+ &= \{a_1, \ldots, a_m\} \\ B(r)^- &= \{a_{m+1}, \ldots, a_n\} \end{aligned}$$

Normal logic programs

- A logic program, P, over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, r, is of the form

$$a_0 \leftarrow a_1, \dots, a_m, \neg a_{m+1}, \dots, \neg a_n$$

where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$

- Notation

$$\begin{aligned} H(r) &= \{a_0\} \\ B(r)^+ &= \{a_1, \dots, a_m\} \\ B(r)^- &= \{a_{m+1}, \dots, a_n\} \\ A(P) &= \bigcup_{r \in P} \left(H(r) \cup B(r)^+ \cup B(r)^- \right) \end{aligned}$$

Normal logic programs

- A logic program, P, over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, r, is of the form

$$a_0 \leftarrow a_1, \dots, a_m, \neg a_{m+1}, \dots, \neg a_n$$

  where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$
- Notation

$$\begin{aligned} H(r) &= \{a_0\} \\ B(r)^+ &= \{a_1, \dots, a_m\} \\ B(r)^- &= \{a_{m+1}, \dots, a_n\} \\ A(P) &= \bigcup_{r \in P} \left(H(r) \cup B(r)^+ \cup B(r)^-\right) \end{aligned}$$

- Note We often assume that $\mathcal{A} = A(P)$

Normal logic programs

- A logic program, P, over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, r, is of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$$

  where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$
- Notation

$$\begin{aligned} h(r) &= a_0 \\ B(r)^+ &= \{a_1, \ldots, a_m\} \\ B(r)^- &= \{a_{m+1}, \ldots, a_n\} \end{aligned}$$

- A program P is positive if $B(r)^- = \emptyset$ for all $r \in P$

Normal logic programs

- A logic program, P, over a set $\mathcal{A}$ of atoms is a finite set of rules
- A (normal) rule, r, is of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$$

  where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$
- A program P is positive if $B(r)^- = \emptyset$ for all $r \in P$

Examples

- Example rules
  - $a \leftarrow b, \neg c$
  - $a \leftarrow \neg c, b$
  - $a \leftarrow$
  - $a \leftarrow b$
  - $a \leftarrow \neg c$
  - $bachelor(joe) \leftarrow male(joe), \neg married(joe)$
- Example literals
  - $a, b, c, bachelor(joe), male(joe), married(joe)$
  - $\neg c, \neg married(joe)$

Examples

- Example rules
  - $a \leftarrow b, \neg c$
  - $a \leftarrow \neg c, b$
  - $a \leftarrow$
  - $a \leftarrow b$
  - $a \leftarrow \neg c$
  - $bachelor(joe) \leftarrow male(joe), \neg married(joe)$
- Example literals
  - $a, b, c, bachelor(joe), male(joe), married(joe)$
  - $\neg c, \neg married(joe)$

Examples

- Example rules
  - $a \leftarrow b, \neg c$
  - $a \leftarrow \neg c, b$
  - $a \leftarrow$
  - $a \leftarrow b$
  - $a \leftarrow \neg c$
  - $bachelor(joe) \leftarrow male(joe), \neg married(joe)$
- Example literals
  - $a, b, c, bachelor(joe), male(joe), married(joe)$
  - $\neg c, \neg married(joe)$

Examples

- Example rules
  - $a \leftarrow b, \neg c$
  - $a \leftarrow \neg c, b$
  - $a \leftarrow$
  - $a \leftarrow b$
  - $a \leftarrow \neg c$
  - $bachelor(joe) \leftarrow male(joe), \neg married(joe)$
- Example literals
  - $a, b, c, bachelor(joe), male(joe), married(joe)$
  - $\neg c, \neg married(joe)$

Literal

- A literal is an atom or a negated atom

  also referred to as positive or negative literal

- Example
  - `answer(42)`, `not answer(42)`
  - `bachelor(friend(joe))`, `not bachelor(friend(joe))`, and
  - `hot`, `not hot`

Rules

- Rules are of the form

$$l_1 \,;\ldots\,; l_m \leftarrow l_{m+1}, \ldots, l_n \tag{1}$$

  where

  - l_i is a conditional literal for $1 \leq i \leq m$ and
  - l_i is a literal for $m+1 \leq i \leq n$

- Note Semicolons ';' must be used in (1) instead of commas ',' whenever some l_i is a (genuine) conditional literal for $1 \leq i \leq n$
- Example `a(X) :- b(X) : c(X), d(X); e(x).`
- Note $l_1 \,;\ldots\,; l_m \leftarrow l_{m+1}, \ldots, l_n$ is the same as

  $l_1 \,;\ldots\,; l_m \leftarrow l_{m+1}; \ldots; l_n$

Rules

- Rules are of the form

$$l_1 \,;\, \dots \,;\, l_m \leftarrow l_{m+1}, \dots, l_n \tag{1}$$

  where

  - l_i is a conditional literal for $1 \leq i \leq m$ and
  - l_i is a literal for $m+1 \leq i \leq n$

- Note Semicolons ';' must be used in (1) instead of commas ',' whenever some l_i is a (genuine) conditional literal for $1 \leq i \leq n$

- Example `a(X) :- b(X) : c(X), d(X); e(x).`

- Note $l_1 \,;\, \dots \,;\, l_m \leftarrow l_{m+1}, \dots, l_n$ is the same as

  $l_1 \,;\, \dots \,;\, l_m \leftarrow l_{m+1}; \dots; l_n$

Rules

- Rules are of the form

$$l_1 \,;\, \dots \,;\, l_m \leftarrow l_{m+1}, \dots, l_n \tag{1}$$

  where
  - l_i is a conditional literal for $1 \leq i \leq m$ and
  - l_i is a literal for $m+1 \leq i \leq n$
- Note Semicolons ';' must be used in (1) instead of commas ',' whenever some l_i is a (genuine) conditional literal for $1 \leq i \leq n$
- Example `a(X) :- b(X) : c(X), d(X); e(x).`
- Note $l_1 \,;\, \dots \,;\, l_m \leftarrow l_{m+1}, \dots, l_n$ is the same as
  $l_1 \,;\, \dots \,;\, l_m \leftarrow l_{m+1}; \dots; l_n$

Rules

- Rules are of the form

$$l_1 \,;\, \dots \,;\, l_m \leftarrow l_{m+1}, \dots, l_n \tag{1}$$

  where

  - l_i is a conditional literal for $1 \leq i \leq m$ and
  - l_i is a literal for $m+1 \leq i \leq n$

- Note Semicolons ';' must be used in (1) instead of commas ',' whenever some l_i is a (genuine) conditional literal for $1 \leq i \leq n$
- Example `a(X) :- b(X) : c(X), d(X); e(x).`
- Note $l_1 \,;\, \dots \,;\, l_m \leftarrow l_{m+1}, \dots, l_n$ is the same as
  $l_1 \,;\, \dots \,;\, l_m \leftarrow l_{m+1}; \dots; l_n$

Notational conventions

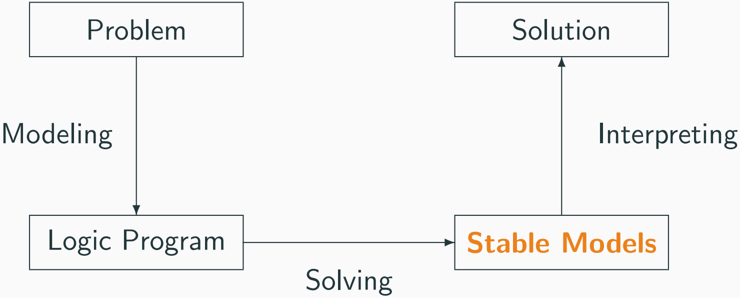
	false, true	if	and	or	iff	(default) negation	strong negation
source code		`:-`	`,`	`;`		`not`	`-`
logic program		$\leftarrow$	$,$	$;$		$\neg$	$\sim$
formula	$\bot, \top$	$\rightarrow$	$\wedge$	$\vee$	$\leftrightarrow$	$\neg$	$\sim$

Let's get practical

Identify stable models from given programs

- Programs without variables
  - positive programs
  - positive recursion
  - normal programs
  - negative recursion

Semantics



Problem
Solution
Modeling
Interpreting
Logic Program
Stable Models
Solving

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

What is a model?

- **Assignment** A function mapping variables to values
- **Solution** An assignment satisfying a set of constraints

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

What is a model?

- Assignment A function mapping variables to values
  - Example $A : \{x, y, z\} \rightarrow \mathbb{N}$ such that
    $A = \{x \mapsto 3, y \mapsto 1, z \mapsto 7\}$
- Solution An assignment satisfying a set of constraints

What is a model?

- Assignment A function mapping variables to values
  - Example $A : \{x, y, z\} \to \mathbb{N}$ such that
    $A = \{x \mapsto 3, y \mapsto 1, z \mapsto 7\}$
- Solution An assignment satisfying a set of constraints
  - Example A is a solution of $\{2x < z, x + y < 2z\}$

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values
  - Truth values $\boldsymbol{T}$ and $\boldsymbol{F}$ stand for true and false

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values
  - Truth values $\boldsymbol{T}$ and $\boldsymbol{F}$ stand for true and false
  - Example $B : \{a, b, c\} \rightarrow \{\boldsymbol{T}, \boldsymbol{F}\}$ such that $B = \{a \mapsto \boldsymbol{T}, b \mapsto \boldsymbol{T}, c \mapsto \boldsymbol{F}\}$

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values
  - Truth values $\boldsymbol{T}$ and $\boldsymbol{F}$ stand for true and false
  - Example $B : \{a, b, c\} \rightarrow \{\boldsymbol{T}, \boldsymbol{F}\}$ such that $B = \{a \mapsto \boldsymbol{T}, b \mapsto \boldsymbol{T}, c \mapsto \boldsymbol{F}\}$
  - An interpretation satisfies a formula if it evaluates the formula to $\boldsymbol{T}$

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values
  - Truth values $\boldsymbol{T}$ and $\boldsymbol{F}$ stand for true and false
  - Example $B : \{a, b, c\} \rightarrow \{\boldsymbol{T}, \boldsymbol{F}\}$ such that $B = \{a \mapsto \boldsymbol{T}, b \mapsto \boldsymbol{T}, c \mapsto \boldsymbol{F}\}$
  - An interpretation satisfies a formula if it evaluates the formula to $\boldsymbol{T}$
- Model An interpretation satisfying a set of formulas (or rules)

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values
  - Truth values $\boldsymbol{T}$ and $\boldsymbol{F}$ stand for true and false
  - Example $B : \{a, b, c\} \rightarrow \{\boldsymbol{T}, \boldsymbol{F}\}$ such that $B = \{a \mapsto \boldsymbol{T}, b \mapsto \boldsymbol{T}, c \mapsto \boldsymbol{F}\}$
  - An interpretation satisfies a formula if it evaluates the formula to $\boldsymbol{T}$
- Model An interpretation satisfying a set of formulas (or rules)
  - Example B is a model of $\{a \wedge b, a \vee c\}$

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values
- Model An interpretation satisfying a set of formulas (or rules)

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values
- Model An interpretation satisfying a set of formulas (or rules)

- Representation

  We often denote interpretations by the set of their true atoms

What is a model?

- Assignment A function mapping variables to values
- Solution An assignment satisfying a set of constraints

- Interpretation An assignment mapping variables to truth values
- Model An interpretation satisfying a set of formulas (or rules)

- Representation

  We often denote interpretations by the set of their true atoms

  - Example We use $\{a, b\}$ to represent $\{a \mapsto \boldsymbol{T}, b \mapsto \boldsymbol{T}, c \mapsto \boldsymbol{F}\}$

Some "logical" remarks

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

$$a_0 \vee \neg a_1 \vee \cdots \vee \neg a_m$$

  - A set of definite clauses has a (unique) smallest model

- This smallest model is the intended semantics of such sets of clauses
  - Given a positive program P, $Cn(P)$ ("consequences") corresponds to the smallest model of the set of definite clauses corresponding to P

Some "logical" remarks

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

$$a_0 \vee \neg a_1 \vee \cdots \vee \neg a_m$$

  - A set of definite clauses has a (unique) smallest model
- This smallest model is the intended semantics of such sets of clauses
  - Given a positive program P, $Cn(P)$ ("consequences") corresponds to the smallest model of the set of definite clauses corresponding to P

Some "logical" remarks

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

$$a_0 \vee \neg a_1 \vee \cdots \vee \neg a_m$$

  - A set of definite clauses has a (unique) smallest model
- This smallest model is the intended semantics of such sets of clauses
  - Given a positive program P, $Cn(P)$ ("consequences") corresponds to the smallest model of the set of definite clauses corresponding to P

What is the meaning of a logic program?

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest set closed under P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest set closed under P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof
  - Example $P = \{a \leftarrow, b \leftarrow a, d \leftarrow c\}$ yields $\{a, b\}$ only
    - a is justified by $a \leftarrow$
    - b is justified by $b \leftarrow a$ and $a \leftarrow$

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest set closed under P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof
  - Example $P = \{a \leftarrow, b \leftarrow a, d \leftarrow c\}$ yields $\{a, b\}$ only
    - a is justified by $a \leftarrow$
    - b is justified by $b \leftarrow a$ and $a \leftarrow$

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest set closed under P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest model of P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof
- Logical attempt
  - Hypothesis The smallest model of P ?

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest model of P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof
- Logical attempt
  - Hypothesis The smallest model of P ?

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest model of P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof

- Logical attempt
  - Hypothesis The smallest model of P ?
  - Example $P = \{a \leftarrow \neg b\}$ yields (stable model) $\{a\}$

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest model of P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof
- Logical attempt
  - Hypothesis The smallest model of P ?
  - Example $P = \{a \leftarrow \neg b\}$ yields (stable model) $\{a\}$
    but P has three models, $\{a\}$, $\{b\}$, and $\{a, b\}$

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest model of P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof

- Logical attempt
  - Hypothesis The smallest model of P?
  - Example $P = \{a \leftarrow \neg b\}$ yields (stable model) $\{a\}$
    but P has two minimal models, $\{a\}$ and $\{b\}$

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest model of P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof
- Logical attempt — failed
  - Hypothesis The smallest model of P ?
  - Example $P = \{a \leftarrow \neg b\}$ yields (stable model) $\{a\}$
    but P has two minimal models, $\{a\}$ and $\{b\}$

What is the meaning of a logic program?

- Lessons learned from positive programs
  - $Cn(P)$ is the smallest set closed under P, eliminating all others
  - Every atom in $Cn(P)$ is justified by a proof

What is the meaning of a logic program?

- The reduct, P^X, of a program P relative to a set X of atoms is

$$P^X = \{h(r) \leftarrow B(r)^+ \mid r \in P,\ B(r)^- \cap X = \emptyset\}$$

What is the meaning of a logic program?

- The reduct, P^X, of a program P relative to a set X of atoms is

$$P^X = \{h(r) \leftarrow B(r)^+ \mid r \in P,\ B(r)^- \cap X = \emptyset\}$$

What is the meaning of a logic program?

- The reduct, P^X, of a program P relative to a set X of atoms is

$$P^X = \{h(r) \leftarrow B(r)^+ \mid r \in P,\ B(r)^- \cap X = \emptyset\}$$

- Note P^X can be obtained from P by
  1. deleting each rule r satisfying $B(r)^- \cap X \neq \emptyset$ and then
  2. deleting all negative body literals from the remaining rules

What is the meaning of a logic program?

- The reduct, P^X, of a program P relative to a set X of atoms is

$$P^X = \{h(r) \leftarrow B(r)^+ \mid r \in P,\ B(r)^- \cap X = \emptyset\}$$

- Note P^X can be obtained from P by
  1. deleting each rule r satisfying $B(r)^- \cap X \neq \emptyset$
     and then
  2. deleting all negative body literals from the remaining rules

  Only negative body literals are evaluated!

What is the meaning of a logic program?

- The reduct, P^X, of a program P relative to a set X of atoms is

$$P^X = \{h(r) \leftarrow B(r)^+ \mid r \in P,\ B(r)^- \cap X = \emptyset\}$$

What is the meaning of a logic program?
Stable models!

- The reduct, P^X, of a program P relative to a set X of atoms is

$$P^X = \{h(r) \leftarrow B(r)^+ \mid r \in P,\ B(r)^- \cap X = \emptyset\}$$

What is the meaning of a logic program?
Stable models!

- The reduct, P^X, of a program P relative to a set X of atoms is

$$P^X = \{h(r) \leftarrow B(r)^+ \mid r \in P,\ B(r)^- \cap X = \emptyset\}$$

- A set X of atoms is a stable model of a program P if $Cn(P^X) = X$

Stable models

- The reduct, P^X, of a program P relative to a set X of atoms is

$$P^X = \{h(r) \leftarrow B(r)^+ \mid r \in P,\ B(r)^- \cap X = \emptyset\}$$

- A set X of atoms is a stable model of a program P if $Cn(P^X) = X$

Stable models

- The reduct, P^X, of a program P relative to a set X of atoms is

$$P^X = \{h(r) \leftarrow B(r)^+ \mid r \in P,\ B(r)^- \cap X = \emptyset\}$$

- A set X of atoms is a stable model of a program P if $Cn(P^X) = X$
- Note
  - $Cn(P^X)$ is the $\subseteq$-smallest set closed under P^X
  - Each atom in X is justified by a proof from P^X

Stable models

- The reduct, P^X, of a program P relative to a set X of atoms is

$$P^X = \{h(r) \leftarrow B(r)^+ \mid r \in P,\ B(r)^- \cap X = \emptyset\}$$

- A set X of atoms is a stable model of a program P if $Cn(P^X) = X$
- Note
  - $Cn(P^X)$ is the $\subseteq$-smallest model of P^X
  - Each atom in X is justified by a proof from P^X

Example one

- $P = \{p \leftarrow p,\ q \leftarrow \neg p\}$

X	P^X	$Cn(P^X)$	
$\{\ \}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✘
$\{p\}$	$p \leftarrow p$	$\emptyset$	✘
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✘

✔ stable model ✘ no stable model

Example one

- $P = \{p \leftarrow p,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example one

- $P = \{p \leftarrow p,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example one

- $P = \{p \leftarrow p,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example one

- $P = \{p \leftarrow p,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example one

- $P = \{p \leftarrow p,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example one

- $P = \{p \leftarrow p,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example one

- $P = \{p \leftarrow p,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example one

- $P = \{p \leftarrow p,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example one

- $P = \{p \leftarrow p,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example one

- $P = \{p \leftarrow p,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example one

- $P = \{p \leftarrow p,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example one

- $P = \{p \leftarrow p,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example one

- $P = \{p \leftarrow p,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example one

- $P = \{p \leftarrow p,\ q \leftarrow \neg p\}$

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

✔ model ✘ no model

Example two

- $P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$

X	P^X	$Cn(P^X)$	
$\{\ \}$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✘
$\{p\}$	$p \leftarrow$	$\{p\}$	
$\{q\}$	$q \leftarrow$	$\{q\}$	
$\{p, q\}$		$\emptyset$	✘

✔ stable model ✘ no stable model

Example two

- $P = \{p \leftarrow \neg q,\ \ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example two

- $P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example two

- $P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example two

- $P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example two

- $P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example two

- $P = \{p \leftarrow \neg q, \; q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example two

- $P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example two

- $P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example two

- $P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example two

- $P = \{p \leftarrow \neg q, \; q \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example two

- $P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$

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

✔ model ✘ no model

Example three

- $P = \{p \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example three

- $P = \{p \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example three

- $P = \{p \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example three

- $P = \{p \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example three

- $P = \{p \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example three

- $P = \{p \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example three

- $P = \{p \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example three

- $P = \{p \leftarrow \neg p\}$

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

✔ stable model ✘ no stable model

Example three

- $P = \{p \leftarrow \neg p\}$

X	P^X	$Cn(P^X)$	
$\{\ \}$	$p \leftarrow$	$\{p\}$	✘
$\{p\}$		$\emptyset$	✔

✔ model ✘ no model

Stable model or Answer Set

- Theory - *cf* Potassco resources (esp. videos): consequences of positive programs, reducts of normal logic programs.
- In practice - some properties + examples.

Stable models vs open world reasoning

- $P = \{p \leftarrow p,\ q \leftarrow \neg p\}$

X	*Classical logic*	*ASP*
$\{\quad\}$	✘	✘
$\{p\quad\}$	✔	✘
$\{\quad q\}$	✔	✔
$\{p, q\}$	✔	✘

Stable models vs open world reasoning

- $P = \{p \leftarrow p,\ q \leftarrow \neg p\}$

X	*Classical logic*	*ASP*
$\{\ \ \}$	✘	✘
$\{p\ \}$	✔	✘
$\{\ q\}$	✔	✔
$\{p, q\}$	✔	✘

Some properties

- A logic program may have zero, one, or multiple stable models

- If X is a stable model of a logic program P, then $X \subseteq H(P)$

- If X is a stable model of a logic program P, then X is a model of P

- If X and Y are stable models of a normal program P, then $X \not\subset Y$

Some properties

- A logic program may have zero, one, or multiple stable models

- If X is a stable model of a logic program P, then $X \subseteq H(P)$

- If X is a stable model of a logic program P, then X is a model of P

- If X and Y are stable models of a normal program P, then $X \not\subset Y$

Some properties

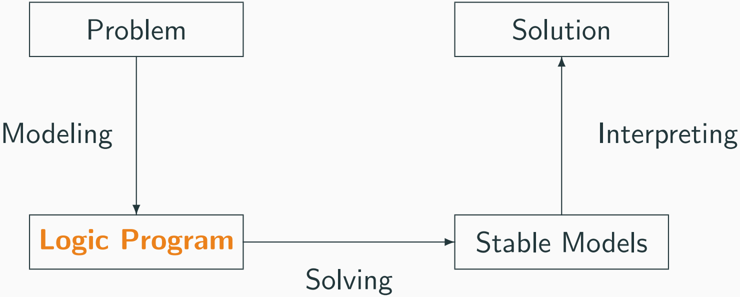
- A logic program may have zero, one, or multiple stable models

- If X is a stable model of a logic program P, then $X \subseteq H(P)$
- If X is a stable model of a logic program P, then X is a model of P
- If X and Y are stable models of a normal program P, then $X \not\subset Y$

Exemplars

Logic program	Stable models
`a.`	{a}
`a :- b.`	{}
`a :- b.      b.`	{a,b}
`a :- b.      b :- a.`	{}
`a :- not c.`	{a}
`a :- not c.  c.`	{c}
`a :- not c.  c :- not a.`	{a}, {c}
`a :- not a.`	

Syntax


Problem
Solution
Modeling
Interpreting
Logic Program
Stable Models
Solving

Language constructs

- Facts `q(42).`
- Rules `p(X) :- q(X), not r(X).`
- Conditional literals `p :- q(X) : r(X).`
- 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 #sum{ X : p(X,Y), q(X) } 7.`
- Multi-objective optimization `:~ q(X), p(X,C). [C@42]`

  `#minimize { C@42 : q(X), p(X,C) }`

Language constructs

- Facts `q(42).`
- Rules `p(X) :- q(X), not r(X).`
- Conditional literals `p :- q(X) : r(X).`
- 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 #sum{ X : p(X,Y), q(X) } 7.`
- Multi-objective optimization `:~ q(X), p(X,C). [C@42]`

  `#minimize { C@42 : q(X), p(X,C) }`

Language constructs

- Facts `q(42).`
- Rules `p(42) :- q(42), not r(42).`
- Conditional literals `p :- q(X) : r(X).`
- 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 #sum{ X : p(X,Y), q(X) } 7.`
- Multi-objective optimization `:~ q(X), p(X,C). [C@42]`

  `#minimize { C@42 : q(X), p(X,C) }`

Language constructs

- Facts `q(42).`
- Rules `p(X) :- q(X), not r(X).`
- Conditional literals `p :- q(X) : r(X).`
- 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 #sum{ X : p(X,Y), q(X) } 7.`
- Multi-objective optimization `:~ q(X), p(X,C). [C@42]`

  `#minimize { C@42 : q(X), p(X,C) }`

Language constructs

- Facts `q(42).`
- Rules `p(X) :- q(X), not r(X).`
- Conditional literals `p :- q(X) : r(X).`
- 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 #sum{ X : p(X,Y), q(X) } 7.`
- Multi-objective optimization `:~ q(X), p(X,C). [C@42]`

  `#minimize { C@42 : q(X), p(X,C) }`

Language constructs

- Facts `q(42).`
- Rules `p(X) :- q(X), not r(X).`
- Conditional literals `p :- q(X) : r(X).`
- 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 #sum{ X : p(X,Y), q(X) } 7.`
- Multi-objective optimization `:~ q(X), p(X,C). [C@42]`

  `#minimize { C@42 : q(X), p(X,C) }`

Language constructs

- Facts `q(42).`
- Rules `p(X) :- q(X), not r(X).`
- Conditional literals `p :- q(X) : r(X).`
- 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 #sum{ X : p(X,Y), q(X) } 7.`
- Multi-objective optimization `:~ q(X), p(X,C). [C@42]`

  `#minimize { C@42 : q(X), p(X,C) }`

Language constructs

- Facts `q(42).`
- Rules `p(X) :- q(X), not r(X).`
- Conditional literals `p :- q(X) : r(X).`
- 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 #sum{ X : p(X,Y), q(X) } 7.`

- Multi-objective optimization `:~ q(X), p(X,C). [C@42]`

  `#minimize { C@42 : q(X), p(X,C) }`

Language constructs

- Facts `q(42).`
- Rules `p(X) :- q(X), not r(X).`
- Conditional literals `p :- q(X) : r(X).`
- 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 #sum{ X : p(X,Y), q(X) } 7.`
- Multi-objective optimization `:~ q(X), p(X,C). [C@42]`

  `#minimize { C@42 : q(X), p(X,C) }`

Language constructs

- Facts `q(42).`
- Rules `p(X) :- q(X), not r(X).`
- Conditional literals `p :- q(X) : r(X).`
- 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 #sum{ X : p(X,Y), q(X) } 7.`
- Multi-objective optimization `:~ q(X), p(X,C). [C@42]`

  `#minimize { C@42 : q(X), p(X,C) }`

Grounding

- Ground instances of a rule r are obtained by replacing all variables in r by terms.
- Instanciation
- Grounding aims at reducing the ground instanciations by applying semantic principles
- More details in Potassco resources

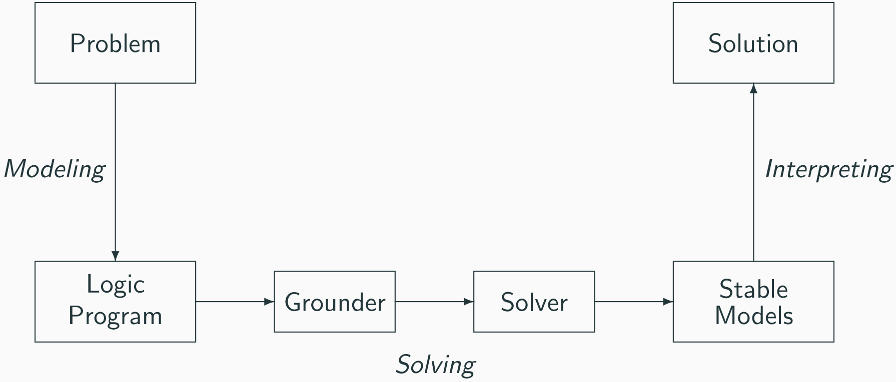
Stable models of programs with Variables

- Let P be a normal logic program with variables
- A set X of (ground) atoms is a stable model of P,
  if X is a stable model of $\mathit{ground}(P)$

Stable models of programs with Variables

- Let P be a normal logic program with variables
- A set X of (ground) atoms is a stable model of P,
  if X is a stable model of $ground(P)$

Modeling, grounding, and solving



Safety

- A normal rule is safe, if all its variables occur in its positive body
- Examples
  - $p(a) \leftarrow$
  - $p(X) \leftarrow$
  - $p(X) \leftarrow q(X)$
  - $p(X) \leftarrow \neg q(X)$
  - $p(X) \leftarrow \neg q(X), r(X)$
- A normal program is safe, if all of its rules are safe

Safety

- A normal rule is safe, if all its variables occur in its positive body
- Examples
  - $p(a) \leftarrow$
  - $p(X) \leftarrow$
  - $p(X) \leftarrow q(X)$
  - $p(X) \leftarrow \neg q(X)$
  - $p(X) \leftarrow \neg q(X), r(X)$
- A normal program is safe, if all of its rules are safe

Safety

- A normal rule is safe, if all its variables occur in its positive body
- Examples
  - $p(a) \leftarrow$
  - $p(X) \leftarrow$
  - $p(X) \leftarrow q(X)$
  - $p(X) \leftarrow \neg q(X)$
  - $p(X) \leftarrow \neg q(X), r(X)$
- A normal program is safe, if all of its rules are safe

Safety

- A normal rule is safe, if all its variables occur in its positive body
- Examples
  - $p(a) \leftarrow$ ✔
  - $p(X) \leftarrow$
  - $p(X) \leftarrow q(X)$
  - $p(X) \leftarrow \neg q(X)$
  - $p(X) \leftarrow \neg q(X), r(X)$
- A normal program is safe, if all of its rules are safe

Safety

- A normal rule is safe, if all its variables occur in its positive body
- Examples
  - $p(a) \leftarrow$ ✔
  - $p(X) \leftarrow$ ✘
  - $p(X) \leftarrow q(X)$
  - $p(X) \leftarrow \neg q(X)$
  - $p(X) \leftarrow \neg q(X), r(X)$
- A normal program is safe, if all of its rules are safe

Safety

- A normal rule is safe, if all its variables occur in its positive body
- Examples
  - $p(a) \leftarrow$ ✔
  - $p(X) \leftarrow$ ✘
  - $p(X) \leftarrow q(X)$ ✔
  - $p(X) \leftarrow \neg q(X)$
  - $p(X) \leftarrow \neg q(X), r(X)$
- A normal program is safe, if all of its rules are safe

Safety

- A normal rule is safe, if all its variables occur in its positive body
- Examples
  - $p(a) \leftarrow$ ✔
  - $p(X) \leftarrow$ ✘
  - $p(X) \leftarrow q(X)$ ✔
  - $p(X) \leftarrow \neg q(X)$ ✘
  - $p(X) \leftarrow \neg q(X), r(X)$
- A normal program is safe, if all of its rules are safe

Safety

- A normal rule is safe, if all its variables occur in its positive body
- Examples
  - $p(a) \leftarrow$ ✔
  - $p(X) \leftarrow$ ✘
  - $p(X) \leftarrow q(X)$ ✔
  - $p(X) \leftarrow \neg q(X)$ ✘
  - $p(X) \leftarrow \neg q(X), r(X)$ ✔
- A normal program is safe, if all of its rules are safe

Safety

- A normal rule is safe, if all its variables occur in its positive body
- Examples
  - $p(a) \leftarrow$ ✔
  - $p(X) \leftarrow$ ✘
  - $p(X) \leftarrow q(X)$ ✔
  - $p(X) \leftarrow \neg q(X)$ ✘
  - $p(X) \leftarrow \neg q(X), r(X)$ ✔
- A normal program is safe, if all of its rules are safe

ASP's syntax and semantics in a nutshell

- Syntax
  - A logic program, P, over a set $\mathcal{A}$ of atoms is a finite set of rules
  - A (normal) rule, r, is of the form

    $$a_0 \leftarrow a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$$

    where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is an atom for $0 \leq i \leq n$

- Semantics
  - The reduct, P^X, of a program P relative to a set X of atoms is

    $$P^X = \{h(r) \leftarrow B(r)^+ \mid r \in P,\ B(r)^- \cap X = \emptyset\}$$

  - A set X of atoms is a stable model of a program P if $Cn(P^X) = X$

Exercises - Basic ASP

Let's get practical

Identify stable models from given programs

- Programs without variables
  - positive programs
  - positive recursion
  - normal programs
  - negative recursion
  - choice rules
  - integrity constraints
  - cardinality rules
- Programs with variables
  - normal logic programs
  - choices and constraints

Language

- Integrity constraint
- Choices
- Cardinality and weight
- Conditional literal
- Optimization
- Reasoning modes
- Exercises

Integrity constraint

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

$$\leftarrow a_1, \dots, a_m, \neg a_{m+1}, \dots, \neg a_n$$

  where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$

- Example

```
:- edge(3,7), color(3,red), color(7,red).
```

Choice rule

- Purpose Provide choices over subsets of atoms
- Syntax A choice rule is of the form

$$\{a_1, \ldots, a_m\} \leftarrow a_{m+1}, \ldots, a_n, \neg a_{n+1}, \ldots, \neg a_o$$

  where $0 \leq m \leq n \leq o$ and each a_i is an atom for $1 \leq i \leq o$

- Informal meaning If the body is satisfied by the stable model, any subset of $\{a_1, \ldots, a_m\}$ can be included in the stable model
- Example

```
{ buy(pizza);buy(wine);buy(corn) } :- at(grocery).
```

Cardinality rule

- Purpose Control (lower) cardinality of subsets of literals
- Syntax A cardinality rule is the form

$$a_0 \leftarrow l \; \{ \; a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n \; \}$$

  where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$

  and l is a non-negative integer called lower bound

- Informal meaning The head belongs to the stable model, if at least

  l positive/negative body literals are in/excluded in the stable model

- Example

```
pass(c42) :- 2 { pass(a1); pass(a2); pass(a3) }.
```

Cardinality rules with upper bounds

- A rule of the form

$$a_0 \leftarrow l \; \{ \; a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n \; \} \; u$$

  where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$

  and l and u are non-negative integers

- Note The expression in the body of the cardinality rule is referred to as a cardinality constraint with lower and upper bound l and u

Cardinality rules with upper bounds

- A rule of the form

$$a_0 \leftarrow l \; \{ \; a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n \; \} \; u$$

  where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$

  and l and u are non-negative integers

- Note The expression in the body of the cardinality rule is referred to as a cardinality constraint with lower and upper bound l and u

Cardinality rules with upper bounds

- A rule of the form

$$a_0 \leftarrow l \; \{ \; a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n \; \} \; u$$

where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$ and l and u are non-negative integers

- Note The expression in the body of the cardinality rule is referred to as a cardinality constraint with lower and upper bound l and u

Cardinality constraints as heads

- A rule of the form

  $$l \, \{a_1, \dots, a_m, \neg a_{m+1}, \dots, \neg a_n\} \, u \leftarrow a_{n+1}, \dots, a_o, \neg a_{o+1}, \dots, \neg a_p$$

  where $0 \le m \le n \le o \le p$ and each a_i is an atom for $1 \le i \le p$

  and l and u are non-negative integers

- Example

```
1 {color(2,red); color(2,green); color(2,blue)}
```

Conditional literals

- Syntax A conditional literal is of the form

$$l : l_1, \dots, l_n$$

  where l and l_i are literals for $0 \leq i \leq n$

- Informal meaning A (non-ground) conditional literal can be regarded as the collection of elements in the set $\{l \mid l_1, \dots, l_n\}$
- Example Assume '`p(1..3). q(2).`'
  - `r(X) : p(X), not q(X)` yields `r(1)` and `r(3)`

Optimization statement

- Purpose Express (multiple) cost functions subject to minimization
(and/or maximization)
- Syntax A minimize statement is of the form

$$minimize\ \{\ w_1@p_1 : l_{1_1}, \dots, l_{m_1}; \dots; w_n@p_n : l_{1_n}, \dots, l_{m_n}\ \}.$$

where each l_{j_i} is a literal and w_i and p_i are integers for $1 \leq i \leq n$

priority levels, p_i, allow for representing lexicographically ordered minimization objectives
- Meaning A minimize statement is a directive that instructs the
ASP solver to compute optimal stable models by minimizing a sum of weights (by descending levels)

Optimization statement

- **Purpose** Express (multiple) cost functions subject to minimization
(and/or maximization)
- **Syntax** A minimize statement is of the form

$$minimize \ \{ \ w_1@p_1 : l_{1_1}, \ldots, l_{m_1}; \ldots; w_n@p_n : l_{1_n}, \ldots, l_{m_n} \ \}.$$

where each l_{j_i} is a literal and w_i and p_i are integers for $1 \leq i \leq n$

priority levels, p_i, allow for representing lexicographically ordered minimization objectives

- **Meaning** A minimize statement is a directive that instructs the
ASP solver to compute optimal stable models by minimizing a sum of weights (by descending levels)

Optimization statement

- Purpose Express (multiple) cost functions subject to minimization (and/or maximization)
- Syntax A minimize statement is of the form

  $$minimize \ \{ \ w_1@p_1 : l_{1_1}, \ldots, l_{m_1}; \ldots; w_n@p_n : l_{1_n}, \ldots, l_{m_n} \ \}.$$

  where each l_{j_i} is a literal and w_i and p_i are integers for $1 \leq i \leq n$

  priority levels, p_i, allow for representing lexicographically ordered minimization objectives
- Meaning A minimize statement is a directive that instructs the ASP solver to compute optimal stable models by minimizing a sum of weights (by descending levels)

Optimization statement

- A maximize statement of the form

$$maximize \; \{ \; w_1@p_1 : l_1, \ldots, w_n@p_n : l_n \; \}$$

stands for $minimize \; \{ \; -w_1@p_1 : l_1, \ldots, -w_n@p_n : l_n \; \}$

- Example When configuring a computer, we may want to maximize hard disk capacity, while minimizing price

```
#maximize { 250@1:hd(1); 500@1:hd(2); 750@1:hd(3) }.
#minimize { 30@2:hd(1);  40@2:hd(2);  60@2:hd(3) }.
```

The priority levels indicate that (minimizing) price is more important than (maximizing) capacity

Optimization statement

- A maximize statement of the form

$$maximize \; \{ \; w_1@p_1 : l_1, \ldots, w_n@p_n : l_n \; \}$$

stands for $minimize \; \{ \; -w_1@p_1 : l_1, \ldots, -w_n@p_n : l_n \; \}$

- Example When configuring a computer, we may want to maximize hard disk capacity, while minimizing price

```
#maximize { 250@1:hd(1); 500@1:hd(2); 750@1:hd(3) }.
#minimize { 30@2:hd(1);  40@2:hd(2);  60@2:hd(3) }.
```

The priority levels indicate that (minimizing) price is more important than (maximizing) capacity

Optimization statement

- A maximize statement of the form

$$maximize \{ \ w_1@p_1 : l_1, \ldots, w_n@p_n : l_n \ \}$$

stands for $minimize \{ \ -w_1@p_1 : l_1, \ldots, -w_n@p_n : l_n \ \}$

- Example When configuring a computer, we may want to maximize hard disk capacity, while minimizing price

```
#maximize { C@1:hd(I,P,C) }.
#minimize { P@2:hd(I,P,C) }.
```

The priority levels indicate that (minimizing) price is more important than (maximizing) capacity

Reasoning modes

- Satisfiability
- Enumeration†
- Projection†
- Intersection‡
- Union‡
- Optimization
- and combinations of them

† without solution recording

‡ without solution enumeration

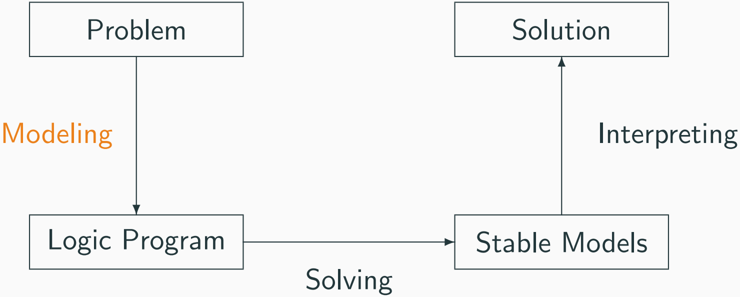
Let's get practical

- Programs with variables
  - count aggregates
  - sum aggregates
  - conditional litterals
- Going further with ASP
  - numbers
  - Python
  - Booleans
  - constants
  - intervals
  - show statements
  - projections
  - output

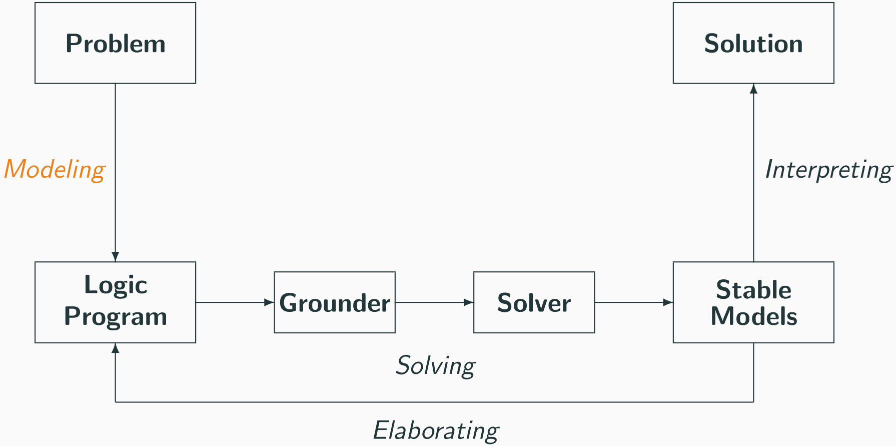
Modeling

- Example of graph coloring

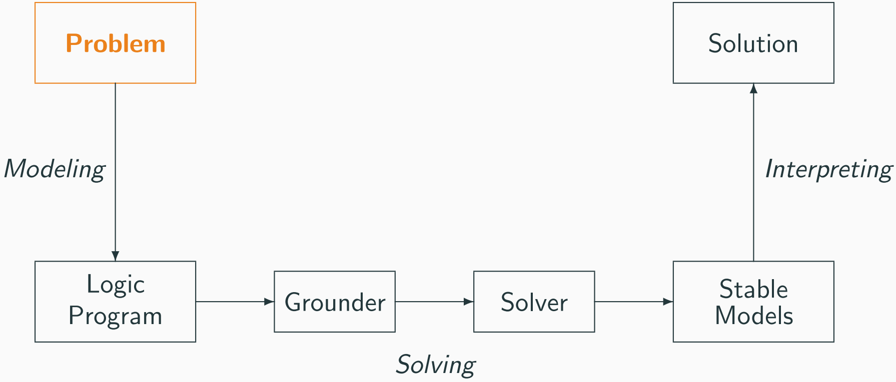
Modeling


Problem
Solution
Modeling
Interpreting
Logic Program
Stable Models
Solving

ASP workflow



ASP workflow: Problem



A case-study: Graph coloring

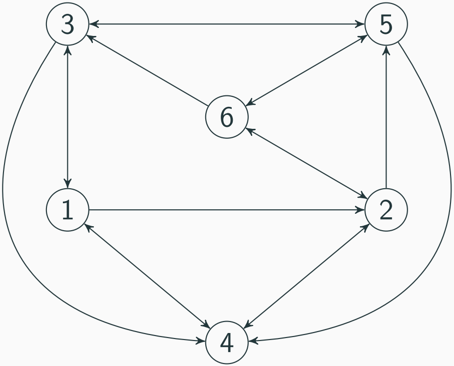
- Problem instance A graph consisting of nodes and edges

A case-study: Graph coloring

- Problem instance A graph consisting of nodes and edges

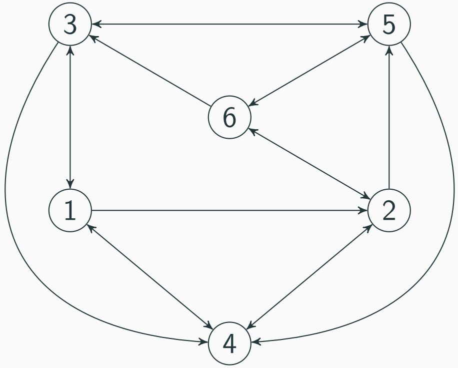
A case-study: Graph coloring

- Problem instance A graph consisting of nodes and edges



A case-study: Graph coloring

- Problem instance A graph consisting of nodes and edges
  - facts formed by predicates `node/1` and `edge/2`



A case-study: Graph coloring

- Problem instance A graph consisting of nodes and edges
  - facts formed by predicates `node/1` and `edge/2`
  - facts formed by predicate `color/1`

A case-study: Graph coloring

- Problem instance A graph consisting of nodes and edges
  - facts formed by predicates `node/1` and `edge/2`
  - facts formed by predicate `color/1`

- Problem class Assign each node one color such that no two nodes connected by an edge have the same color

A case-study: Graph coloring

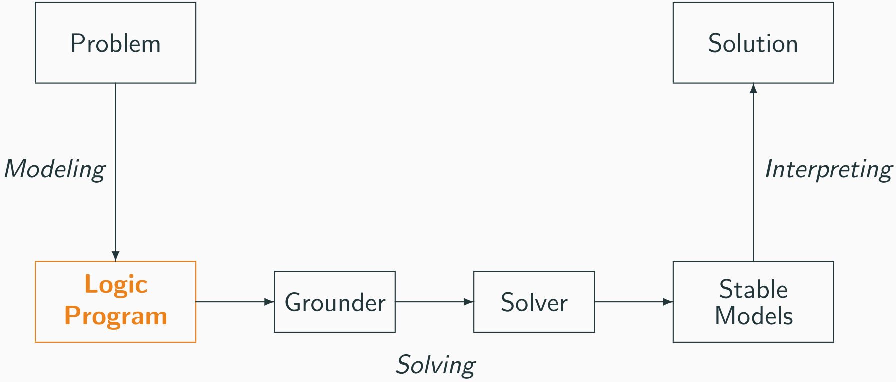
- Problem instance A graph consisting of nodes and edges
  - facts formed by predicates `node/1` and `edge/2`
  - facts formed by predicate `color/1`

- Problem class Assign each node one color such that no two nodes connected by an edge have the same color

  In other words,

  1. Each node has one color
  2. Two connected nodes must not have the same color

ASP workflow: Problem representation



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).

color(r).   color(b).   color(g).
```

Problem instance

```
{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,C).
```

Problem encoding

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).

color(r).   color(b).   color(g).

{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,C).
```

Problem instanc

Problem encoding

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).

color(r).   color(b).   color(g).
```

Problem instance

```
{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,C).
```

Problem encoding

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).

color(r).   color(b).   color(g).
```

Problem instance

```
{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,C).
```

Problem encoding

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).

color(r).   color(b).   color(g).
```

Problem instance

```
{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,C).
```

Problem encoding

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).

color(r).   color(b).   color(g).

{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,C).
```

Problem instance

Problem encoding

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).

color(r).   color(b).   color(g).
```

Problem instance

```
{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,C).
```

Problem encoding

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).

color(r).   color(b).   color(g).
```

Problem instance

```
{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,C).
```

Problem encoding

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).

color(r).   color(b).   color(g).
```

Problem instance

```
{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,C).
```

Problem encoding

Graph coloring

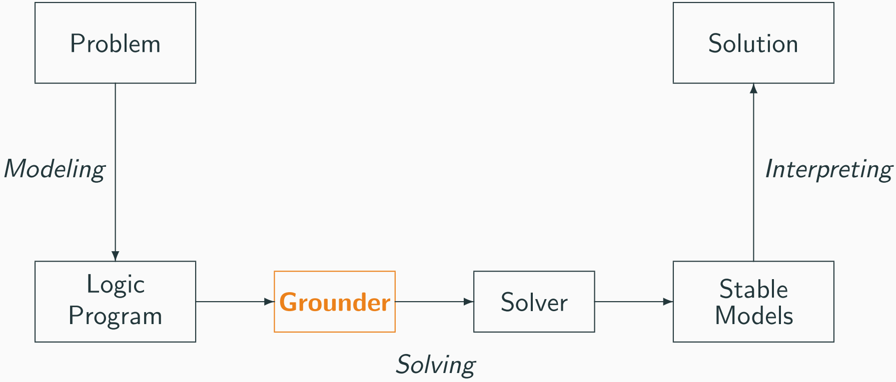
```
node(1..6).

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).

color(r).   color(b).   color(g).
```

graph.lp

```
{ assign(N,C) : color(C) } = 1 :- node(N).

:- edge(N,M), assign(N,C), assign(M,C).
```

color.lp

ASP workflow: Grounding



Graph coloring: Grounding

```
$ gringo --text graph.lp color.lp

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

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

color(r).  color(b).  color(g).

{ assign(1,r); assign(1,b); assign(1,g) } = 1. { assign(4,r); assign(4,b); assign(4,g) } = 1.
{ assign(2,r); assign(2,b); assign(2,g) } = 1. { assign(5,r); assign(5,b); assign(5,g) } = 1.
{ assign(3,r); assign(3,b); assign(3,g) } = 1. { assign(6,r); assign(6,b); assign(6,g) } = 1.

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

Graph coloring: Grounding

```
$ gringo --text graph.lp color.lp
```

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

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

color(r).  color(b).  color(g).

{ assign(1,r); assign(1,b); assign(1,g) } = 1. { assign(4,r); assign(4,b); assign(4,g) } = 1.
{ assign(2,r); assign(2,b); assign(2,g) } = 1. { assign(5,r); assign(5,b); assign(5,g) } = 1.
{ assign(3,r); assign(3,b); assign(3,g) } = 1. { assign(6,r); assign(6,b); assign(6,g) } = 1.

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

Graph coloring: Grounding

```
$ gringo --text graph.lp color.lp
```

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

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

color(r).  color(b).  color(g).

{assign(1,r);assign(1,b);assign(1,g)} = 1. {assign(4,r);assign(4,b);assign(4,g)} = 1.
{assign(2,r);assign(2,b);assign(2,g)} = 1. {assign(5,r);assign(5,b);assign(5,g)} = 1.
{assign(3,r);assign(3,b);assign(3,g)} = 1. {assign(6,r);assign(6,b);assign(6,g)} = 1.

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

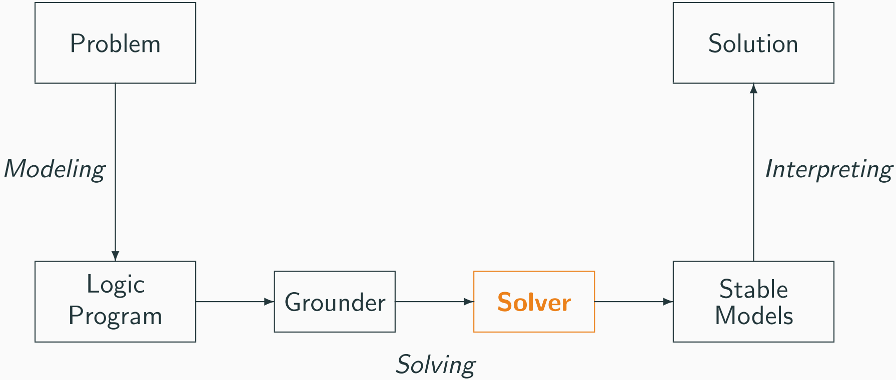
Graph coloring: Grounding

```
$ gringo --text graph.lp color.lp

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

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

color(r).  color(b).  color(g).

{ assign(1,r); assign(1,b); assign(1,g) } = 1. { assign(4,r); assign(4,b); assign(4,g) } = 1.
{ assign(2,r); assign(2,b); assign(2,g) } = 1. { assign(5,r); assign(5,b); assign(5,g) } = 1.
{ assign(3,r); assign(3,b); assign(3,g) } = 1. { assign(6,r); assign(6,b); assign(6,g) } = 1.

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

Graph coloring: Grounding

```
$ clingo --text graph.lp color.lp

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

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

color(r).  color(b).  color(g).

{ assign(1,r); assign(1,b); assign(1,g) } = 1. { assign(4,r); assign(4,b); assign(4,g) } = 1.
{ assign(2,r); assign(2,b); assign(2,g) } = 1. { assign(5,r); assign(5,b); assign(5,g) } = 1.
{ assign(3,r); assign(3,b); assign(3,g) } = 1. { assign(6,r); assign(6,b); assign(6,g) } = 1.

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

ASP workflow: Solving



Graph coloring: Solving

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

```
clasp version 2.1.0
Reading from stdin
Solving...
Answer: 1
node(1) [...] assign(6,b) assign(5,g) assign(4,b) assign(3,r) assign(2,r) assign(1,g)
Answer: 2
node(1) [...] assign(6,r) assign(5,g) assign(4,r) assign(3,b) assign(2,b) assign(1,g)
Answer: 3
node(1) [...] assign(6,g) assign(5,b) assign(4,g) assign(3,r) assign(2,r) assign(1,b)
Answer: 4
node(1) [...] assign(6,r) assign(5,b) assign(4,r) assign(3,g) assign(2,g) assign(1,b)
Answer: 5
node(1) [...] assign(6,g) assign(5,r) assign(4,g) assign(3,b) assign(2,b) assign(1,r)
Answer: 6
node(1) [...] assign(6,b) assign(5,r) assign(4,b) assign(3,g) assign(2,g) assign(1,r)
SATISFIABLE

Models      : 6
Time        : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
```

Graph coloring: Solving

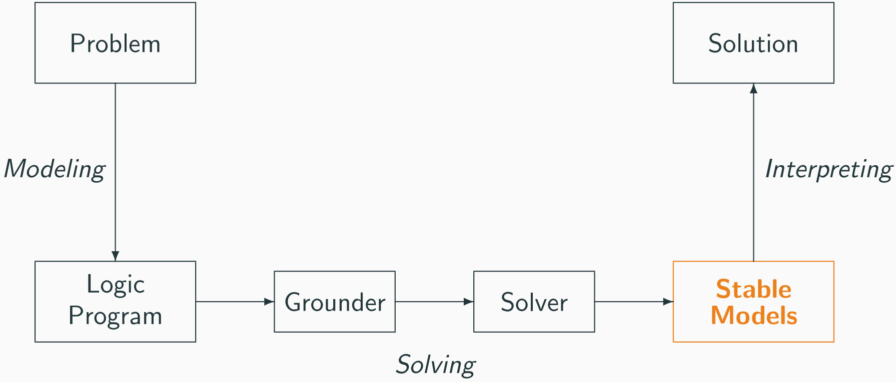
```
$ gringo graph.lp color.lp | clasp 0
```

```
clasp version 2.1.0
Reading from stdin
Solving...
Answer: 1
node(1) [...] assign(6,b) assign(5,g) assign(4,b) assign(3,r) assign(2,r) assign(1,g)
Answer: 2
node(1) [...] assign(6,r) assign(5,g) assign(4,r) assign(3,b) assign(2,b) assign(1,g)
Answer: 3
node(1) [...] assign(6,g) assign(5,b) assign(4,g) assign(3,r) assign(2,r) assign(1,b)
Answer: 4
node(1) [...] assign(6,r) assign(5,b) assign(4,r) assign(3,g) assign(2,g) assign(1,b)
Answer: 5
node(1) [...] assign(6,g) assign(5,r) assign(4,g) assign(3,b) assign(2,b) assign(1,r)
Answer: 6
node(1) [...] assign(6,b) assign(5,r) assign(4,b) assign(3,g) assign(2,g) assign(1,r)
SATISFIABLE

Models      : 6
Time        : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
```

Graph coloring: Solving

```
$ clingo graph.lp color.lp 0

clasp version 2.1.0
Reading from stdin
Solving...
Answer: 1
node(1) [...] assign(6,b) assign(5,g) assign(4,b) assign(3,r) assign(2,r) assign(1,g)
Answer: 2
node(1) [...] assign(6,r) assign(5,g) assign(4,r) assign(3,b) assign(2,b) assign(1,g)
Answer: 3
node(1) [...] assign(6,g) assign(5,b) assign(4,g) assign(3,r) assign(2,r) assign(1,b)
Answer: 4
node(1) [...] assign(6,r) assign(5,b) assign(4,r) assign(3,g) assign(2,g) assign(1,b)
Answer: 5
node(1) [...] assign(6,g) assign(5,r) assign(4,g) assign(3,b) assign(2,b) assign(1,r)
Answer: 6
node(1) [...] assign(6,b) assign(5,r) assign(4,b) assign(3,g) assign(2,g) assign(1,r)
SATISFIABLE

Models      : 6
Time        : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
```

ASP workflow: Stable models

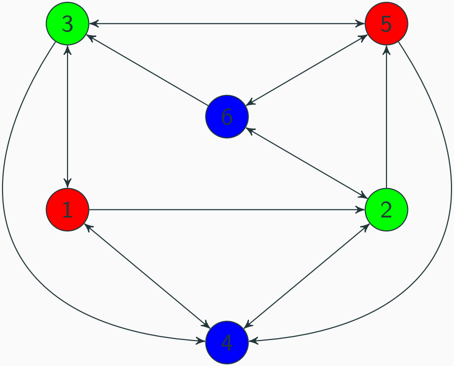


A coloring

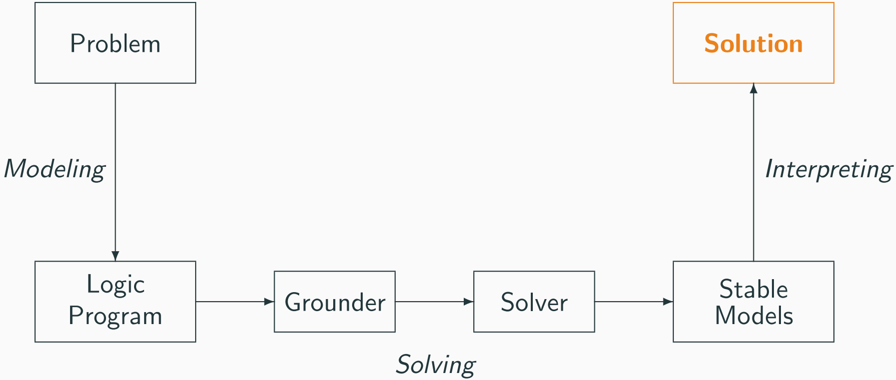
```
Answer: 6
node(1)   [...]   \
assign(6,b) assign(5,r) assign(4,b) assign(3,g) assign(2,g) assign(1,r)
```

A coloring

```
Answer: 6
node(1)   [...]    \
assign(6,b) assign(5,r) assign(4,b) assign(3,g) assign(2,g) assign(1,r)
```



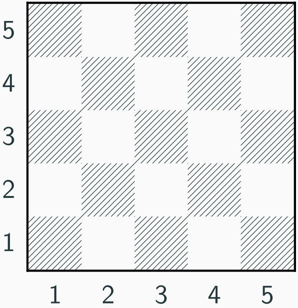
ASP workflow: Solutions



Hands-on

- n-queens problem
- Reviewer assignment
- Sudoku
- Traveling sales person

The n-queens problem

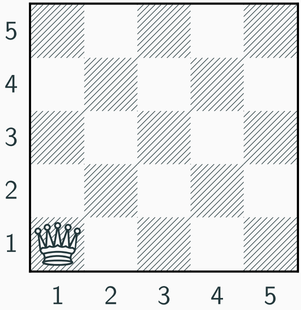


- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

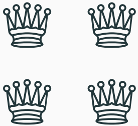
♕ ♕ ♕

♕ ♕

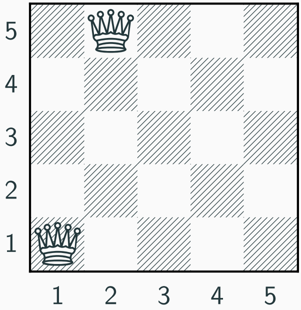
The n-queens problem



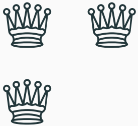
- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$



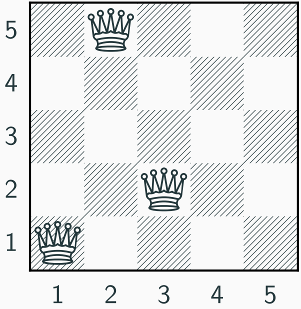
The n-queens problem



- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

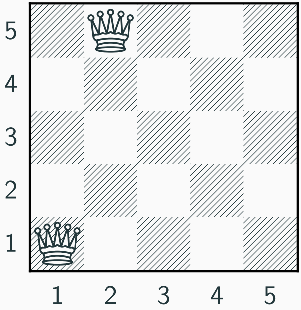


The n-queens problem

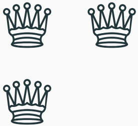


- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

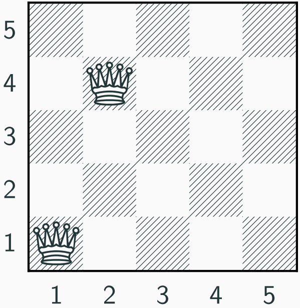
The n-queens problem



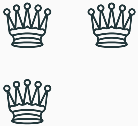
- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$



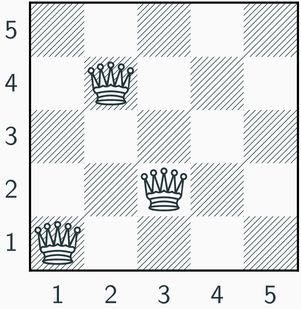
The n-queens problem



- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

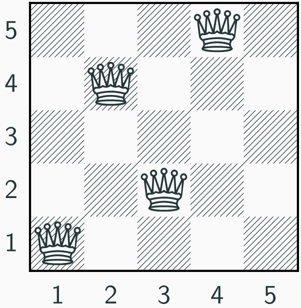


The n-queens problem



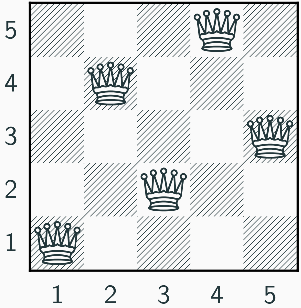
- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

The n-queens problem



- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

The n-queens problem



- Place n queens on an $n \times n$ chess board
- Queens must not attack one another
- Example $n = 5$

Defining the field

queens.lp

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

- Define the field
  - n rows
  - n columns

Defining the field

queens.lp

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

➥ Define the field

- n rows
- n columns

Defining the field

Running ...

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

Models       : 1
Time         : 0.000
```

Placing some queens

queens.lp

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

➥ Guess a solution candidate

by placing some queens on the board

Placing some queens

queens.lp

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

➥ Guess a solution candidate

by placing some queens on the board

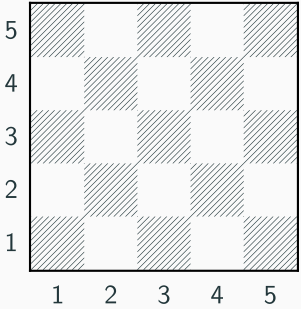
Placing some queens

Running ...

```
$ clingo queens.lp --const 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

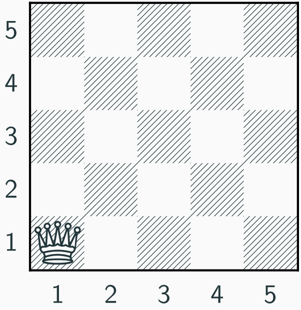
Answer: 1




```
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5)
```

Placing some queens

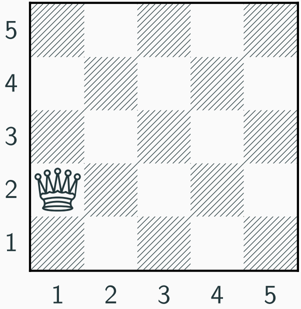
```
Answer: 2
```




```
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)
```

Placing some queens

```
Answer: 3
```



```
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)
```

Placing *n* queens

queens.lp

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

➥ Place exactly *n* queens on the board

Placing *n* queens

queens.lp

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

➥ Place exactly *n* queens on the board

Placing n queens directly

queens.lp

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

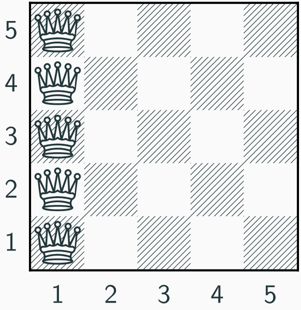
➥ Place exactly n queens on the board

Running . . .

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

Placing *n* queens

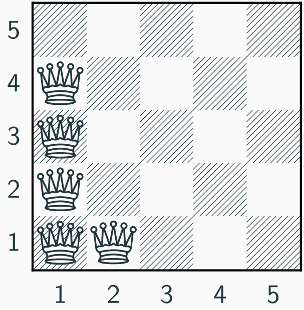
```
Answer: 1
```




```
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(5,1) queen(4,1) queen(3,1) \
queen(2,1) queen(1,1)
```

Placing *n* queens

Answer: 2




```
Answer: 2
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(1,2) queen(4,1) queen(3,1) \
queen(2,1) queen(1,1)
```

Horizontal and vertical attack

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I), col(J) }.
:- { queen(I,J) } != n.
:- queen(I,J), queen(I,J'), J != J'.
:- queen(I,J), queen(I',J), I != I'.
```

➥ Forbid horizontal and vertical attacks

Horizontal and vertical attack

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I), col(J) }.
:- { queen(I,J) } != n.
:- queen(I,J), queen(I,J'), J != J'.
:- queen(I,J), queen(I',J), I != I'.
```

➥ Forbid horizontal and vertical attacks

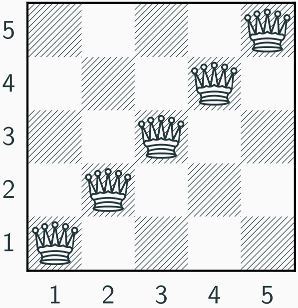
Horizontal and vertical attack

Running ...

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

Horizontal and vertical attack

```
Answer: 1
```




```
Answer: 1
row(1) row(2) row(3) row(4) row(5) \
col(1) col(2) col(3) col(4) col(5) \
queen(5,5) queen(4,4) queen(3,3) \
queen(2,2) queen(1,1)
```

Diagonal attack

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I), col(J) }.
:- { queen(I,J) } != n.
:- queen(I,J), queen(I,J'), J != J'.
:- queen(I,J), queen(I',J), I != I'.
:- queen(I,J), queen(I',J'), (I,J) != (I',J'), I-J == I'-
:- queen(I,J), queen(I',J'), (I,J) != (I',J'), I+J == I'+
```

➥ Forbid diagonal attacks

Diagonal attack

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I), col(J) }.
:- { queen(I,J) } != n.
:- queen(I,J), queen(I,J'), J != J'.
:- queen(I,J), queen(I',J), I != I'.
:- queen(I,J), queen(I',J'), (I,J) != (I',J'), I-J == I'-
:- queen(I,J), queen(I',J'), (I,J) != (I',J'), I+J == I'+
```

➥ Forbid diagonal attacks

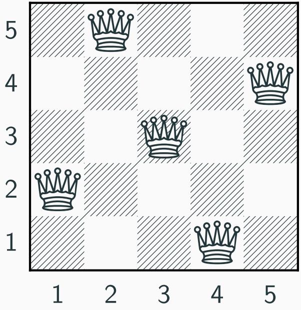
Diagonal attack

Running ...

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

Models       : 1+
Time         : 0.000
```

Diagonal attack

```
Answer: 1
```



```
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)
```

Optimizing

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I), col(J) }.
:- { queen(I,J) } != n.
:- queen(I,J), queen(I,J'), J != J'.
:- queen(I,J), queen(I',J), I != I'.
:- queen(I,J), queen(I',J'), (I,J) != (I',J'), I-J == I'-
:- queen(I,J), queen(I',J'), (I,J) != (I',J'), I+J == I'+
```

- Encoding can be optimized
- Much faster to solve

Optimizing

queens.lp

```
row(1..n).
col(1..n).
{ queen(I,J) : row(I), col(J) }.
:- { queen(I,J) } != n.
:- queen(I,J), queen(I,J'), J != J'.
:- queen(I,J), queen(I',J), I != I'.
:- queen(I,J), queen(I',J'), (I,J) != (I',J'), I-J == I'-
:- queen(I,J), queen(I',J'), (I,J) != (I',J'), I+J == I'+
```

- Encoding can be optimized
- Much faster to solve

Optimizing

queens-opt.lp

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

- Encoding can be optimized
- Much faster to solve

And sometimes it rocks

```
$ clingo -c n=5000 queens-opt-diag.lp --config=jumpy -q --stats=2

clingo version 4.1.0
Solving...
SATISFIABLE

Models       : 1+
Time         : 3758.143s (Solving: 1905.22s 1st Model: 1896.20s Unsat: 0.00s)
CPU Time     : 3758.320s

Choices      : 288594554
Conflicts    : 3442   (Analyzed: 3442)
Restarts     : 17     (Average: 202.47 Last: 3442)
Model-Level  : 7594728.0
Problems     : 1      (Average Length: 0.00 Splits: 0)
Lemmas       : 3442   (Deleted: 0)
  Binary     : 0      (Ratio:   0.00%)
  Ternary    : 0      (Ratio:   0.00%)
  Conflict   : 3442   (Average Length: 229056.5 Ratio: 100.00%)
  Loop       : 0      (Average Length:    0.0 Ratio:   0.00%)
  Other      : 0      (Average Length:    0.0 Ratio:   0.00%)

Atoms        : 75084857 (Original: 75069989 Auxiliary: 14868)
Rules        : 100129956 (1: 50059992/100090100 2: 39990/29856 3: 10000/10000)
Bodies       : 25090103
Equivalences : 125029999 (Atom=Atom: 50009999 Body=Body: 0 Other: 75020000)
Tight        : Yes
Variables    : 25024868 (Eliminated: 11781 Frozen: 25000000)
Constraints  : 66664  (Binary: 35.6% Ternary:  0.0% Other: 64.4%)

Backjumps    : 3442   (Average: 681.19 Max: 169512 Sum: 2344658)
  Executed   : 3442   (Average: 681.19 Max: 169512 Sum: 2344658 Ratio: 100.00%)
  Bounded    : 0      (Average:  0.00 Max:    0 Sum:      0 Ratio:   0.00%)
```

And sometimes it rocks

```
$ clingo -c n=5000 queens-opt-diag.lp --config=jumpy -q --stats=2
```

```
clingo version 4.1.0
Solving...
SATISFIABLE

Models       : 1+
Time         : 3758.143s (Solving: 1905.22s 1st Model: 1896.20s Unsat: 0.00s)
CPU Time     : 3758.320s

Choices      : 288594554
Conflicts    : 3442   (Analyzed: 3442)
Restarts     : 17     (Average: 202.47 Last: 3442)
Model-Level  : 7594728.0
Problems     : 1      (Average Length: 0.00 Splits: 0)
Lemmas       : 3442   (Deleted: 0)
  Binary     : 0      (Ratio:   0.00%)
  Ternary    : 0      (Ratio:   0.00%)
  Conflict   : 3442   (Average Length: 229056.5 Ratio: 100.00%)
  Loop       : 0      (Average Length:    0.0 Ratio:   0.00%)
  Other      : 0      (Average Length:    0.0 Ratio:   0.00%)

Atoms        : 75084857 (Original: 75069989 Auxiliary: 14868)
Rules        : 100129956 (1: 50059992/100090100 2: 39990/29856 3: 10000/10000)
Bodies       : 25090103
Equivalences : 125029999 (Atom=Atom: 50009999 Body=Body: 0 Other: 75020000)
Tight        : Yes
Variables    : 25024868 (Eliminated: 11781 Frozen: 25000000)
Constraints  : 66664  (Binary: 35.6% Ternary:  0.0% Other: 64.4%)

Backjumps    : 3442   (Average: 681.19 Max: 169512 Sum: 2344658)
  Executed   : 3442   (Average: 681.19 Max: 169512 Sum: 2344658 Ratio: 100.00%)
  Bounded    : 0      (Average:  0.00 Max:   0 Sum:      0 Ratio:   0.00%)
```

Reviewer Assignment

- Problem Instance A set of papers and a set of reviewers along with their first and second choices of papers and conflict of interests
- Problem Class An assignment of three reviewers to each paper

Reviewer Assignment - by Ilkka Niemelä

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

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

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

#minimize { 1,P,R : assignedB(P,R), paper(P), reviewer(R) }.
```

Reviewer Assignment - by Ilkka Niemelä

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

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

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

#minimize { 1,P,R : assignedB(P,R), paper(P), reviewer(R) }.
```

Reviewer Assignment - by Ilkka Niemelä

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

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

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

#minimize { 1,P,R : assignedB(P,R), paper(P), reviewer(R) }.
```

Reviewer Assignment - by Ilkka Niemelä

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

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

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

#minimize { 1,P,R : assignedB(P,R), paper(P), reviewer(R) }.
```

Reviewer Assignment - by Ilkka Niemelä

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

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

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

#minimize { 1,P,R : assignedB(P,R), paper(P), reviewer(R) }.
```

Reviewer Assignment - by Ilkka Niemelä

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

#count { P,R : 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).
 :- not 6 <= #count { P,R : assigned(P,R), paper(P) } <= 9, reviewer(R).

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

#minimize { 1,P,R : assignedB(P,R), paper(P), reviewer(R) }.
```

Reviewer Assignment - by Ilkka Niemelä

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

#count { P,R : 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).
 :- not 6 <= #count { P,R : assigned(P,R), paper(P) } <= 9, reviewer(R).

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

#minimize { 1,P,R : assignedB(P,R), paper(P), reviewer(R) }.
```

Sudoku

Solve a Sudoku puzzle using ASP.

- Fill a 9x9 grid with digits
- Each column, each row and each of the nine 3x3 sub-grids that compose the grid contains all numbers from 1 to 9.
- Partially filled grid as an input

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Representation in ASP

The initial state of the grid is represented by facts of predicate `initial/3`:

```
initial(X,Y,N). % initially cell [X,Y] contains number
N
```

The solution is represented by atoms of predicate `sudoku/3`:

```
sudoku(X,Y,N). % the cell [X,Y] contains number N
```

First tests

Start with a 4x4 grid

9x9 grid

```
initial(1,1,5). initial(1,2,3). initial(1,5,7). initial(2,1,6).
initial(2,4,1). initial(2,5,9). initial(2,6,5). initial(3,2,9).
initial(3,3,8). initial(3,8,6). initial(4,1,8). initial(4,5,6).
initial(4,9,3). initial(5,1,4). initial(5,4,8). initial(5,6,3).
initial(5,9,1). initial(6,1,7). initial(6,5,2). initial(6,9,6).
initial(7,2,6). initial(7,7,2). initial(7,8,8). initial(8,4,4).
initial(8,5,1). initial(8,6,9). initial(8,9,5). initial(9,5,8).
initial(9,8,7). initial(9,9,9).
```

A 4x4 grid example

```
1,_, _,_,
_,4, _,_,

_,_, _,3,
2,_, _,1,
```

```
initial(1,1,1).
initial(2,2,4).
initial(3,4,3).
initial(4,1,2). initial(4,4,1).
```

The traveling salesperson problem (TSP)

- Problem Instance A set of cities and distances among them, or simply a weighted graph
- Problem Class What is the shortest possible route visiting each city once and returning to the city of origin?
- Note
  - TSP extends the Hamiltonian cycle problem:
    Is there a cycle in a graph visiting each node exactly once
  - TSP is relevant to applications in logistics, planning, chip design,
    and the core of the vehicle routing problem

Traveling salesperson

```
start(a).

city(a). city(b). city(c). city(d).

road(a,b,10). road(b,c,20). road(c,d,25). road(d,a,40
road(b,d,30). road(d,c,25). road(c,a,35).
```

Traveling salesperson

```
{ travel(X,Y) } :- road(X,Y,_).

visited(Y) :- travel(X,Y), start(X).
visited(Y) :- travel(X,Y), visited(X).

:- city(X), not visited(X).

:- city(X), 2 { travel(X,Y) }.
:- city(X), 2 { travel(Y,X) }.
```

Traveling salesperson

```
{ travel(X,Y) } :- road(X,Y,_).

visited(Y) :- travel(X,Y), start(X).
visited(Y) :- travel(X,Y), visited(X).

:- city(X), not visited(X).

:- city(X), 2 { travel(X,Y) }.
:- city(X), 2 { travel(Y,X) }.

#minimize { D,X,Y : travel(X,Y), road(X,Y,D) }.
```

Running salesperson

```
$ clingo tsp.lp cities.lp
clingo version 5.3.1
Reading...
Solving...
Answer: 1
start(a) [...] road(c,a,35)
travel(a,b) travel(b,d) travel(d,c) travel(c,a)
visited(b) visited(c) visited(d) visited(a)
Optimization: 100

Answer: 2
start(a) [...] road(c,a,35)
travel(a,b) travel(b,c) travel(c,d) travel(d,a)
visited(b) visited(c) visited(d) visited(a)
Optimization: 95
OPTIMUM FOUND

Models       : 2
  Optimum    : yes
Optimization : 95
Calls        : 1
Time         : 0.005s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.002s
```

Traveling salesperson - Alternative problem encoding

```
{ travel(X,Y) : road(X,Y,_) } = 1 :- city(X).
{ travel(X,Y) : road(X,Y,_) } = 1 :- city(Y).

visited(Y) :- travel(X,Y), start(X).
visited(Y) :- travel(X,Y), visited(X).

:- city(X), not visited(X).

#minimize { D,X,Y : travel(X,Y), road(X,Y,D) }.
```

Going further

- Python and ASP
- Preferences

Python & ASP

2 possibilities

- Use Python code directly in ASP e.g. to make calculations
- Hide ASP in Python (cf Clyngor)
- Completely hide the use of ASP from the user by installing clingo binaries in a Python environment (cf Clyngor-with-Clingo))

Preferences with Asprin

- Going further than classical optimisations
- Asprin on Github
- Various types of preferences already implemented (weight, cardinality, pareto...)
- Possibility to implement new ones

Applications

- Bioinformatics
- Planning, scheduling, timetabling
- Classification (e.g. of customers based on their preferences)
- Systems configuration

Bibliography

Bibliography

- The following list of references is compiled from the open source bibliography available at

  `https://github.com/krr-up/bibliography`

Potassco

[1] S. Abiteboul, R. Hull, and V. Vianu. **Foundations of Databases.** Addison-Wesley, 1995.

[2] M. Alviano et al. **"The ASP System DLV2".** In: *Proceedings of the Fourteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'17)*. Ed. by M. Balduccini and T. Janhunen. Vol. 10377. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2017, pp. 215–221.

[3] M. Alviano et al. **"The Fourth Answer Set Programming Competition: Preliminary Report".** In: *Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*. Ed. by P. Cabalar and T. Son. Vol. 8148. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2013, pp. 42–53.

[4] M. Alviano et al. **"WASP: A Native ASP Solver Based on Constraint Learning".** In: *Proceedings of the Twelfth International Conference on Logic Programming and*

Potassco

Nonmonotonic Reasoning (LPNMR'13). Ed. by P. Cabalar and T. Son. Vol. 8148. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2013, pp. 54–66.

[5] C. Baral. **Knowledge Representation, Reasoning and Declarative Problem Solving.** Cambridge University Press, 2003.

[6] C. Baral, G. Brewka, and J. Schlipf, eds. **Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07).** Vol. 4483. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2007.

[7] C. Baral and M. Gelfond. **"Logic Programming and Knowledge Representation".** In: *Journal of Logic Programming* 12 (1994), pp. 1–80.

[8] P. Borchert et al. **"Towards Systematic Benchmarking in Answer Set Programming: The Dagstuhl Initiative".** In:

Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04). Ed. by V. Lifschitz and I. Niemelä. Vol. 2923. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2004, pp. 3–7.

[9] G. Brewka, T. Eiter, and M. Truszczynski. **"Answer Set Programming: An Introduction to the Special Issue".** In: *AI Magazine* 37.3 (2016), pp. 5–6.

[10] G. Brewka, T. Eiter, and M. Truszczyński. **"Answer set programming at a glance".** In: *Communications of the ACM* 54.12 (2011), pp. 92–103.

[11] P. Cabalar and T. Son, eds. **Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13).** Vol. 8148. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2013.

[12] F. Calimeri et al. **"ASP-Core-2 Input Language Format".** In: *Theory and Practice of Logic Programming* 20.2 (2019), pp. 294–309.

[13] F. Calimeri et al. **"ASP-Core-2 Input Language Format".** In: *Theory and Practice of Logic Programming* 20.2 (2020), pp. 294–309.

[14] F. Calimeri et al. **"I-DLV: The new intelligent grounder of DLV".** In: *Intelligenza Artificiale* 11.1 (2017), pp. 5–20.

[15] F. Calimeri et al. **"The Design of the Fifth Answer Set Programming Competition".** In: *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*. Ed. by M. Leuschel and T. Schrijvers. Vol. arXiv:1405.3710v4. Theory and Practice of Logic Programming, Online Supplement. 2014. URL: http://arxiv.org/abs/1405.3710v4.

[16] F. Calimeri et al. **“The Third Answer Set Programming Competition: Preliminary Report of the System Competition Track”.** In: *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR’11)*. Ed. by J. Delgrande and W. Faber. Vol. 6645. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2011, pp. 388–403.

[17] J. Delgrande and W. Faber, eds. **Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR’11).** Vol. 6645. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2011.

[18] M. Denecker et al. **“The Second Answer Set Programming Competition”.** In: *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR’09)*. Ed. by E. Erdem, F. Lin,

and T. Schaub. Vol. 5753. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2009, pp. 637–654.

[19] T. Eiter, G. Ianni, and T. Krennwallner. **"Answer Set Programming: A Primer".** In: *Fifth International Reasoning Web Summer School (RW'09)*. Ed. by S. Tessaris et al. Vol. 5689. Lecture Notes in Computer Science. Slides at http://www.kr.tuwien.ac.at/staff/tkren/pub/2009/rw2009-lecture.zip. Springer-Verlag, 2009, pp. 40–110. URL: http://www.kr.tuwien.ac.at/staff/tkren/pub/2009/rw2009-asp.pdf.

[20] E. Erdem, F. Lin, and T. Schaub, eds. **Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09).** Vol. 5753. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2009.

[21] M. Gebser, B. Kaufmann, and T. Schaub. **"Conflict-Driven Answer Set Solving: From Theory to Practice".** In: *Artificial Intelligence* 187-188 (2012), pp. 52–89.

[22] M. Gebser, T. Schaub, and S. Thiele. **"Gringo: A New Grounder for Answer Set Programming".** In: *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*. Ed. by C. Baral, G. Brewka, and J. Schlipf. Vol. 4483. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2007, pp. 266–271.

[23] M. Gebser et al. **"Abstract Gringo".** In: *Theory and Practice of Logic Programming* 15.4-5 (2015), pp. 449–463.

[24] M. Gebser et al. **"Advances in gringo Series 3".** In: *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*. Ed. by J. Delgrande and W. Faber. Vol. 6645. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2011, pp. 345–351.

[25] M. Gebser et al. **Answer Set Solving in Practice.** Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012.

[26] M. Gebser et al. **“Conflict-Driven Answer Set Solving”.** In: *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI’07)*. Ed. by M. Veloso. AAAI/MIT Press, 2007, pp. 386–392.

[27] M. Gebser et al. **“Multi-shot ASP solving with clingo”.** In: *Theory and Practice of Logic Programming* 19.1 (2019), pp. 27–82.

[28] M. Gebser et al. **“On the Input Language of ASP Grounder Gringo”.** In: *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR’09)*. Ed. by E. Erdem, F. Lin, and T. Schaub. Vol. 5753. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2009, pp. 502–508.

[29] M. Gebser et al. **Potassco User Guide.** 2nd ed. University of Potsdam. 2015. URL: http://potassco.org.

[30] M. Gebser et al. **"The First Answer Set Programming System Competition".** In: *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*. Ed. by C. Baral, G. Brewka, and J. Schlipf. Vol. 4483. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2007, pp. 3–17.

[31] M. Gelfond. **"Answer Sets".** In: *Handbook of Knowledge Representation*. Ed. by V. Lifschitz, F. van Harmelen, and B. Porter. Elsevier Science, 2008. Chap. 7, pp. 285–316.

[32] M. Gelfond and Y. Kahl. **Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach.** Cambridge University Press, 2014.

[33] M. Gelfond and N. Leone. **"Logic programming and knowledge representation — the A-Prolog perspective".** In: *Artificial Intelligence* 138.1-2 (2002), pp. 3–38.

[34] M. Gelfond and V. Lifschitz. **"Logic Programs with Classical Negation".** In: *Proceedings of the Seventh International Conference on Logic Programming (ICLP'90)*. Ed. by D. Warren and P. Szeredi. MIT Press, 1990, pp. 579–597.

[35] M. Gelfond and V. Lifschitz. **"The Stable Model Semantics for Logic Programming".** In: *Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*. Ed. by R. Kowalski and K. Bowen. MIT Press, 1988, pp. 1070–1080.

[36] R. Kaminski, T. Schaub, and P. Wanko. **"A Tutorial on Hybrid Answer Set Solving with clingo".** In: *Proceedings of the Thirteenth International Summer School of the Reasoning*

Web. Ed. by G. Ianni et al. Vol. 10370. Lecture Notes in Computer Science. Springer-Verlag, 2017, pp. 167–203.

[37] N. Leone et al. **“The DLV System for Knowledge Representation and Reasoning”.** In: *ACM Transactions on Computational Logic* 7.3 (2006), pp. 499–562.

[38] V. Lifschitz. **Answer Set Programming.** Springer-Verlag, 2019.

[39] V. Lifschitz. **“Answer set programming and plan generation”.** In: *Artificial Intelligence* 138.1-2 (2002), pp. 39–54.

[40] V. Lifschitz. **“Introduction to answer set programming”.** Unpublished draft. 2004. URL: `http://www.cs.utexas.edu/users/vl/papers/esslli.ps`.

[41] V. Lifschitz. **“Thirteen Definitions of a Stable Model”.** In: *Fields of Logic and Computation, Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*. Ed. by A. Blass,

N. Dershowitz, and W. Reisig. Vol. 6300. Lecture Notes in Computer Science. Springer-Verlag, 2010, pp. 488–503.

[42] V. Lifschitz. **"Twelve Definitions of a Stable Model".** In: *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*. Ed. by M. Garcia de la Banda and E. Pontelli. Vol. 5366. Lecture Notes in Computer Science. Springer-Verlag, 2008, pp. 37–51.

[43] V. Marek and M. Truszczyński. **Nonmonotonic logic: context-dependent reasoning.** Artifical Intelligence. Springer-Verlag, 1993.

[44] V. Marek and M. Truszczyński. **"Stable models and an alternative logic programming paradigm".** In: *The Logic Programming Paradigm: a 25-Year Perspective*. Ed. by K. Apt et al. Springer-Verlag, 1999, pp. 375–398.

[45] I. Niemelä. **"Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm".** In:

Annals of Mathematics and Artificial Intelligence 25.3-4 (1999), pp. 241–273.

[46] I. Niemelä and P. Simons. **“Efficient Implementation of the Well-founded and Stable Model Semantics”.** In: *Proceedings of the Joint International Conference and Symposium on Logic Programming*. Ed. by M. Maher. MIT Press, 1996, pp. 289–303.

[47] T. Schaub and S. Woltran. **“Answer set programming unleashed!”** In: *Künstliche Intelligenz* 32.2-3 (2018), pp. 105–108.

[48] T. Schaub and S. Woltran. **“Special Issue on Answer Set Programming”.** In: *Künstliche Intelligenz* 32.2-3 (2018), pp. 101–103.

[49] P. Simons, I. Niemelä, and T. Soininen. **“Extending and implementing the stable model semantics”.** In: *Artificial Intelligence* 138.1-2 (2002), pp. 181–234.

[50] T. Syrjänen. **“Omega-Restricted Logic Programs”.** In: *Proceedings of the Sixth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR’01)*. Ed. by T. Eiter, W. Faber, and M. Truszczyński. Vol. 2173. Lecture Notes in Computer Science. Springer-Verlag, 2001, pp. 267–279.