PROBLEM SOLVING AND SEARCH IN ARTIFICIAL INTELLIGENCE

Lecture 5 Answer-Set Programming Motivation and Introduction
* slides adapted from Torsten Schaub [Gebser et al.(2012)]

Sarah Gaggl

Dresden, 14th November 2019
Agenda

1. Introduction
2. Uninformed Search versus Informed Search (Best First Search, A* Search, Heuristics)
3. Local Search, Stochastic Hill Climbing, Simulated Annealing
4. Tabu Search
5. Answer-set Programming (ASP)
6. Constraint Satisfaction (CSP)
7. Evolutionary Algorithms/ Genetic Algorithms
8. Structural Decomposition Techniques (Tree/Hypertree Decompositions)
Outline

1 Motivation
   - Declarative Problem Solving
   - ASP in a Nutshell
   - ASP Paradigm

2 Introduction
   - Syntax
   - Semantics
   - Examples
   - Language Constructs
   - Modeling
Informatics

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

Problem

Computer

Solution

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

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

Diagram:
- Problem -> Computer -> Solution
- Computer -> Output
Traditional programming

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

Problem

Program

Solution

Output

Programming

Interpreting

Executing
Declarative problem solving

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

Diagram:
- **Problem** → **Computer** → **Output**
- **Solution** → **Interpreting**
Declarative problem solving

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

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

Problem \rightarrow Representation \rightarrow Solution \rightarrow Output

Modeling \rightarrow Solving \rightarrow Interpreting
Answer Set Programming
in a Nutshell

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

- ASP is an approach to **declarative problem solving**, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (deductive) databases
  - logic programming (with negation)
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - constraint solving (in particular, SATisfiability testing)
Answer Set Programming in a Nutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (deductive) databases
  - logic programming (with negation)
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - constraint solving (in particular, SATisfiability testing)
- ASP allows for solving all search problems in $NP$ (and $NP^{NP}$) in a uniform way
Answer Set Programming
in a Nutshell

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

• ASP has its roots in
  – (deductive) databases
  – logic programming (with negation)
  – (logic-based) knowledge representation and (nonmonotonic) reasoning
  – constraint solving (in particular, SATisfiability testing)

• ASP allows for solving all search problems in $NP$ (and $NP^{NP}$) in a uniform way

• ASP is versatile as reflected by the ASP solver clasp, winning first places at ASP, CASC, MISC, PB, and SAT competitions
Answer Set Programming
in a Nutshell

- ASP is an approach to **declarative problem solving**, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (deductive) databases
  - logic programming (with negation)
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - constraint solving (in particular, SATisfiability testing)
- ASP allows for solving all search problems in \( NP \) (and \( NP^{NP} \)) in a uniform way
- ASP is versatile as reflected by the ASP solver **clasp**, winning first places at ASP, CASC, MISC, PB, and SAT competitions
- ASP embraces many emerging application areas
Answer Set Programming
in a Hazelnutshell

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

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

\[ \text{ASP} = \text{DB} + \text{LP} + \text{KR} + \text{SAT} \]
KR’s shift of paradigm

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

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

Model Generation based approach (eg. SATisfiability testing)
1. Provide a representation of the problem
2. A solution is given by a model of the representation
<table>
<thead>
<tr>
<th>Representation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint satisfaction problem</td>
<td>assignment</td>
</tr>
<tr>
<td>propositional horn theories</td>
<td>smallest model</td>
</tr>
<tr>
<td>propositional theories</td>
<td>models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>stable models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>Supported models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>Herbrand models</td>
</tr>
<tr>
<td>auto-epistemic theories</td>
<td>expansions</td>
</tr>
<tr>
<td>default theories</td>
<td>extensions</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Representation</td>
<td>Solution</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>constraint satisfaction problem</td>
<td>assignment</td>
</tr>
<tr>
<td>propositional horn theories</td>
<td>smallest model</td>
</tr>
<tr>
<td>propositional theories</td>
<td>models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional theories</td>
<td>stable models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>minimal models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>supported models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>minimal models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>Herbrand models</td>
</tr>
<tr>
<td>auto-epistemic theories</td>
<td>expansions</td>
</tr>
<tr>
<td>default theories</td>
<td>extensions</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Model Generation based Problem Solving

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

Prolog program

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

Prolog program

on(a,b).
o (b,c).
above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).

Prolog queries

?- above(a,c).
true.
LP-style playing with blocks

Prolog program

on(a,b).
on(b,c).

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

Prolog queries

?- above(a,c).
true.

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

**Prolog program**

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

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

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

**Prolog queries (testing entailment)**

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

true.

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

no.
LP-style playing with blocks

Shuffled Prolog program

```
on(a,b).
on(b,c).

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

Prolog queries

```
?- above(a,c).
```

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

<table>
<thead>
<tr>
<th>Shuffled Prolog program</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>on(a,b).</code></td>
</tr>
<tr>
<td><code>on(b,c).</code></td>
</tr>
<tr>
<td><code>above(X,Y) :- above(X,Z), on(Z,Y).</code></td>
</tr>
<tr>
<td><code>above(X,Y) :- on(X,Y).</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prolog queries</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>?- above(a,c).</code></td>
</tr>
</tbody>
</table>
LP-style playing with blocks

Shuffled Prolog program

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

Prolog queries (answered via fixed execution)

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

Fatal Error: local stack overflow.

TU Dresden, 14th November 2019   PSSAI   slide 29 of 141
SAT-style playing with blocks

**Formula**

\[
\begin{align*}
on(a, b) \\
\land on(b, c) \\
\land (on(X, Y) \rightarrow above(X, Y)) \\
\land (on(X, Z) \land above(Z, Y) \rightarrow above(X, Y))
\end{align*}
\]
SAT-style playing with blocks

Formula

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

Herbrand model

\[
\{ \begin{array}{ccccc}
on(a, b), & on(b, c), & on(a, c), & on(b, b), \\
above(a, b), & above(b, c), & above(a, c), & above(b, b), & above(c, b)
\end{array} \}
\]
SAT-style playing with blocks

Formula

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

Herbrand model (among 426!)

\[ \begin{align*} \{ & on(a, b), \quad on(b, c), \quad on(a, c), \quad on(b, b), \\
& above(a, b), \quad above(b, c), \quad above(a, c), \quad above(b, b), \quad above(c, b) \} \end{align*} \]
SAT-style playing with blocks

Formula

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

Herbrand model (among 426!)

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

Formula

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

Herbrand model (among 426!)

\[
\{ \text{on}(a, b), \text{on}(b, c), \text{on}(a, c), \text{on}(b, b), \\
\text{above}(a, b), \text{above}(b, c), \text{above}(a, c), \text{above}(b, b), \text{above}(c, b) \}
\]
KR’s shift of paradigm

Theorem Proving based approach (eg. Prolog)

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

Model Generation based approach (eg. SATisfiability testing)

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

Model Generation based approach (eg. SATisfiability testing)

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

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

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

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

TU Dresden, 14th November 2019
### Answer Set Programming commonly

<table>
<thead>
<tr>
<th>Representation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>propositional theories</td>
<td>stable models</td>
</tr>
<tr>
<td>propositional programs</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order theories</td>
<td>stable models</td>
</tr>
</tbody>
</table>

...
<table>
<thead>
<tr>
<th>Representation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>propositional programs</td>
<td>stable models</td>
</tr>
<tr>
<td>Representation</td>
<td>Solution</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>propositional programs</td>
<td>stable models</td>
</tr>
<tr>
<td>first-order programs</td>
<td>stable Herbrand models</td>
</tr>
</tbody>
</table>
ASP-style playing with blocks

Logic program

\[
\begin{align*}
on(a,b) . \\
on(b,c) . \\
above(X,Y) & : \neg on(X,Y) . \\
above(X,Y) & : \neg on(X,Z), above(Z,Y) .
\end{align*}
\]
ASP-style playing with blocks

Logic program

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

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

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

Stable Herbrand model

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

Logic program

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

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

Stable Herbrand model (and no others)

\begin{verbatim}
{ on(a,b), on(b,c), above(b,c), above(a,b), above(a,c) }
\end{verbatim}
ASP-style playing with blocks

Logic program

on(a, b).
on(b, c).

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

Stable Herbrand model (and no others)

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

<table>
<thead>
<tr>
<th></th>
<th>ASP</th>
<th>Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model generation</td>
<td></td>
<td>Query orientation</td>
</tr>
<tr>
<td>Bottom-up</td>
<td></td>
<td>Top-down</td>
</tr>
<tr>
<td>Modeling language</td>
<td></td>
<td>Programming language</td>
</tr>
<tr>
<td>Rule-based format</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>NP((NP]))</td>
<td>Unification</td>
</tr>
<tr>
<td>Flat terms</td>
<td></td>
<td>Nested terms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Turing</td>
</tr>
</tbody>
</table>
### ASP versus SAT

<table>
<thead>
<tr>
<th>ASP</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model generation</strong></td>
<td><strong>Bottom-up</strong></td>
</tr>
<tr>
<td><strong>Constructive Logic</strong></td>
<td><strong>Classical Logic</strong></td>
</tr>
<tr>
<td>Closed (and open) world reasoning</td>
<td>Open world reasoning</td>
</tr>
<tr>
<td>Modeling language</td>
<td>—</td>
</tr>
<tr>
<td>Complex reasoning modes</td>
<td>Satisfiability testing</td>
</tr>
<tr>
<td>Satisfiability</td>
<td>Satisfiability</td>
</tr>
<tr>
<td>Enumeration/Projection</td>
<td>—</td>
</tr>
<tr>
<td>Optimization</td>
<td>—</td>
</tr>
<tr>
<td>Intersection/Union</td>
<td>—</td>
</tr>
<tr>
<td>((Turing +) NP^{NP})</td>
<td>(NP)</td>
</tr>
</tbody>
</table>
ASP solving

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

Modeling

Solving

Solution

Interpreting
SAT solving

Problem → Formula (CNF) → Solver → Classical Models

Programming → Formula (CNF) → Solver → Classical Models

Interpreting → Classical Models
Rooting ASP solving

Problem

Modeling

Logic Program

Solving

Grounder

Solver

Stable Models

Solution

Interpreting
Rooting ASP solving

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

Modeling → KR → LP → DB → Solving → SAT → DB+KR+LP

Interpreting
Two sides of a coin

- **ASP as High-level Language**
  - Express problem instance(s) as sets of facts
  - Encode problem (class) as a set of rules
  - Read off solutions from stable models of facts and rules

- **ASP as Low-level Language**
  - Compile a problem into a logic program
  - Solve the original problem by solving its compilation
What is ASP good for?

- Combinatorial search problems in the realm of $P$, $NP$, and $NP^{NP}$
  (some with substantial amount of data), like
What is ASP good for?

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

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

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

**ASP = DB+LP+KR+SAT**
Agenda

1 Motivation
   - Declarative Problem Solving
   - ASP in a Nutshell
   - ASP Paradigm

2 Introduction
   - Syntax
   - Semantics
   - Examples
   - Language Constructs
   - Modeling
Problem solving in ASP: Syntax

Problem -> Logic Program
Modeling

Logic Program -> Stable Models
Solving

Stable Models -> Solution
Interpreting
Normal logic programs

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

$$a_0 \leftarrow a_1, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n$$

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

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

$$a_0 \leftarrow a_1, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n$$

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

- Notation

\[
\begin{align*}
\text{head}(r) & = a_0 \\
\text{body}(r) & = \{a_1, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n\} \\
\text{body}(r)^+ & = \{a_1, \ldots, a_m\} \\
\text{body}(r)^- & = \{a_{m+1}, \ldots, a_n\}
\end{align*}
\]
Normal logic programs

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

$$a_0 \leftarrow a_1, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n$$

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

$$\text{head}(r) = a_0$$
$$\text{body}(r) = \{a_1, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n\}$$
$$\text{body}(r)^+ = \{a_1, \ldots, a_m\}$$
$$\text{body}(r)^- = \{a_{m+1}, \ldots, a_n\}$$

• A program is called positive if $\text{body}(r)^- = \emptyset$ for all its rules
We sometimes use the following notation interchangeably in order to stress the respective view:

<table>
<thead>
<tr>
<th>Source code logic program formula</th>
<th>true, false</th>
<th>if</th>
<th>and</th>
<th>or</th>
<th>iff</th>
<th>default negation</th>
<th>classical negation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>:- ,</td>
<td></td>
<td></td>
<td></td>
<td>not</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>← , ;</td>
<td></td>
<td></td>
<td></td>
<td>not</td>
<td>¬</td>
<td></td>
</tr>
<tr>
<td>⊥, T</td>
<td>→</td>
<td>∧</td>
<td>∨</td>
<td>↔</td>
<td>~</td>
<td>¬</td>
<td></td>
</tr>
</tbody>
</table>
Problem solving in ASP: Semantics

Problem -> Logic Program -> Stable Models

Modeling

Solution

Interpreting

Solving
Formal Definition
Stable models of positive programs

- A set of atoms $X$ is **closed under** a positive program $P$ iff for any $r \in P$, $\text{head}(r) \in X$ whenever $\text{body}(r)^+ \subseteq X$
  - $X$ corresponds to a model of $P$ (seen as a formula)
Formal Definition

Stable models of positive programs

- A set of atoms $X$ is **closed under** a positive program $P$ iff for any $r \in P$, $head(r) \in X$ whenever $body(r)^+ \subseteq X$
  - $X$ corresponds to a model of $P$ (seen as a formula)

- The **smallest** set of atoms which is closed under a positive program $P$ is denoted by $Cn(P)$
  - $Cn(P)$ corresponds to the $\subseteq$-smallest model of $P$ (ditto)
Formal Definition

Stable models of positive programs

- A set of atoms $X$ is closed under a positive program $P$ iff for any $r \in P$, $\text{head}(r) \in X$ whenever $\text{body}(r)^+ \subseteq X$
  - $X$ corresponds to a model of $P$ (seen as a formula)

- The smallest set of atoms which is closed under a positive program $P$ is denoted by $Cn(P)$
  - $Cn(P)$ corresponds to the $\subseteq$-smallest model of $P$ (ditto)

- The set $Cn(P)$ of atoms is the stable model of a positive program $P$
Some “logical” remarks

- Positive rules are also referred to as **definite clauses**
  - Definite clauses are disjunctions with **exactly one** positive atom:
    
    $$a_0 \lor \neg a_1 \lor \cdots \lor \neg a_m$$

  - A set of definite clauses has a (unique) smallest model
Some “logical” remarks

- Positive rules are also referred to as **definite clauses**
  - Definite clauses are disjunctions with exactly one positive atom:
    \[ a_0 \lor \neg a_1 \lor \cdots \lor \neg a_m \]
  - A set of definite clauses has a (unique) smallest model

- **Horn clauses** are clauses with at most one positive atom
  - Every definite clause is a Horn clause but not vice versa
  - Non-definite Horn clauses can be regarded as integrity constraints
  - A set of Horn clauses has a smallest model or none
Some “logical” remarks

• Positive rules are also referred to as definite clauses
  – Definite clauses are disjunctions with exactly one positive atom:
    \[ a_0 \lor \neg a_1 \lor \cdots \lor \neg a_m \]
  – A set of definite clauses has a (unique) smallest model

• Horn clauses are clauses with at most one positive atom
  – Every definite clause is a Horn clause but not vice versa
  – Non-definite Horn clauses can be regarded as integrity constraints
  – A set of Horn clauses has a smallest model or none

• This smallest model is the intended semantics of such sets of clauses
  – Given a positive program \( P \), \( Cn(P) \) corresponds to the smallest model of the set of definite clauses corresponding to \( P \)
Basic idea

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

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

$\Phi \equiv q \land (q \land \neg r \rightarrow p)$
Basic idea

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

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

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

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

$\Phi \quad q \land (q \land \neg r \rightarrow p)$

$p \leftrightarrow 1 
q \leftrightarrow 1 
r \leftrightarrow 0$
Basic idea

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

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

\[
\Phi \equiv q \land (q \land \neg r \rightarrow p)
\]
Basic idea

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

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

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

$$\{p, q\}$$
Basic idea

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

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

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

$$\{p, q\}$$
Basic idea

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

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

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

\[
\{p, q\}
\]

Informally, a set $X$ of atoms is a stable model of a logic program $P$

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

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

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

$$\{p, q\}$$

Informally, a set $X$ of atoms is a **stable model** of a logic program $P$ if

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

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

Stable model of normal programs

- The Gelfond-Lifschitz Reduct [Gelfond and Lifschitz (1991)], $P^X$, of a program $P$ relative to a set $X$ of atoms is defined by

$$P^X = \{head(r) \leftarrow body(r)^+ \mid r \in P \text{ and } body(r)^- \cap X = \emptyset\}$$
Formal Definition
Stable model of normal programs

- The **Gelfond-Lifschitz Reduct** [Gelfond and Lifschitz(1991)], $P^X$, of a program $P$ relative to a set $X$ of atoms is defined by

\[
P^X = \{ \text{head}(r) \leftarrow \text{body}(r)^+ \mid r \in P \text{ and } \text{body}(r)^- \cap X = \emptyset \}\]

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

Stable model of normal programs

- The Gelfond-Lifschitz Reduct [Gelfond and Lifschitz (1991)], $P^X$, of a program $P$ relative to a set $X$ of atoms is defined by

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

- 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 (classical) model of $P^X$
- Note: Every atom in $X$ is justified by an “applying rule from $P$”
A closer look at $P^X$

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

$P^X$ is obtained from $P$ by deleting

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

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

  $P^X$ is obtained from $P$ by deleting

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

- Note: Only negative body literals are evaluated w.r.t. $X$
A first example

\[ P = \{ p \leftarrow p, \ q \leftarrow \text{not } p \} \]
A first example

\[ P = \{ p \leftrightarrow p, \; q \leftrightarrow \text{not } p \} \]

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

Let $P = \{p \leftarrow p, \ q \leftarrow not \ p\}$

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

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

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

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

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

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

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

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

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

\[ P = \{ p \leftarrow \text{not } q, \ q \leftarrow \text{not } p \} \]
A second example

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

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

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

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

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

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

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

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

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

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

\[ P = \{ p \leftarrow \text{not } p \} \]
A third example

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

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

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

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

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

<table>
<thead>
<tr>
<th>( X )</th>
<th>( P^X )</th>
<th>( \text{Cn}(P^X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>( p \leftarrow )</td>
<td>( {p} ) ( \times )</td>
</tr>
<tr>
<td>{p}</td>
<td>( \emptyset )</td>
<td>( \times )</td>
</tr>
</tbody>
</table>
Some properties

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

- A logic program may have zero, one, or multiple stable models!
- If $X$ is an stable model of a logic program $P$, then $X$ is a model of $P$ (seen as a formula)
- If $X$ and $Y$ are stable models of a normal program $P$, then $X \not\subseteq Y$
Let $P$ be a logic program

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

- Let $\mathcal{T}$ be a set of variable-free terms (also called Herbrand universe)
- Let $\mathcal{A}$ be a set of (variable-free) atoms constructable from $\mathcal{T}$ (also called alphabet or Herbrand base)
Let $P$ be a logic program

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

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

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

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

Let $P$ be a logic program

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

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

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

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

- **Ground Instantiation of $P$:** $\text{ground}(P) = \bigcup_{r \in P} \text{ground}(r)$
An example

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

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

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

\[
P = \{ \ r(a, b) \leftarrow, \ r(b, c) \leftarrow, \ t(X, Y) \leftarrow r(X, Y) \ \} \\
\mathcal{T} = \{a, b, c\} \\
\mathcal{A} = \{ \ r(a, a), r(a, b), r(a, c), r(b, a), r(b, b), r(b, c), r(c, a), r(c, b), r(c, c), \ t(a, a), t(a, b), t(a, c), t(b, a), t(b, b), t(b, c), t(c, a), t(c, b), t(c, c) \ \} \\
\text{ground}(P) = \left\{ \begin{array}{c}
  r(a, b) \leftarrow, \\
  r(b, c) \leftarrow, \\
  t(a, a) \leftarrow r(a, a), \ t(b, a) \leftarrow r(b, a), \ t(c, a) \leftarrow r(c, a), \\
  t(a, b) \leftarrow r(a, b), \ t(b, b) \leftarrow r(b, b), \ t(c, b) \leftarrow r(c, b), \\
  t(a, c) \leftarrow r(a, c), \ t(b, c) \leftarrow r(b, c), \ t(c, c) \leftarrow r(c, c) \\
\end{array} \right\}
\]
An example

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

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

\[ \mathcal{A} = \left\{ r(a,a), r(a,b), r(a,c), r(b,a), r(b,b), r(b,c), r(c,a), r(c,b), r(c,c), \right. \]
\[ \left. \begin{array}{l}
\ t(a,a), t(a,b), t(a,c), t(b,a), t(b,b), t(b,c), t(c,a), t(c,b), t(c,c)
\end{array} \right\} \]

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

- Intelligent Grounding aims at reducing the ground instantiation
Stable models of programs with Variables

Let $P$ be a normal logic program with variables
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 $\text{Cn}(\text{ground}(P)^X) = X$
Problem solving in ASP: Extended Syntax

- Problem
  - Modeling
  - Logic Program

- Solution
  - Interpreting
  - Stable Models

- Solving
Language Constructs

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

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

- Disjunction
  - p(X) | q(X) :- r(X)

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

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

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

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

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

- Disjunction
  - \( p(X) | q(X) :- r(X) \)

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

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

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

- **Conditional Literals**
  - $p :- q(X) : r(X)$
  - given $r(a), r(b), r(c)$ stands for
  - $p :- q(a), q(b), q(c)$
Language Constructs

- **Disjunction**
  
  \[- p(X) \mid q(X) : r(X)\]
Language Constructs

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

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

- **Disjunction**
  
  \[ p(X) | q(X) :- r(X) \]

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

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

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

- **Variables (over the Herbrand Universe)**
  
  \[ p(X) \leftarrow q(X) \]

  over constants \{a, b, c\}

  stands for

  \[ p(a) \leftarrow q(a), p(b) \leftarrow q(b), p(c) \leftarrow q(c) \]

- **Conditional Literals**

  \[ p \leftarrow q(X) : r(X) \]

  given \( r(a), r(b), r(c) \)

  stands for

  \[ p \leftarrow q(a), q(b), q(c) \]

- **Disjunction**

  \[ p(X) \lor q(X) \leftarrow r(X) \]

- **Integrity Constraints**

  \[ \leftarrow q(X), p(X) \]

- **Choice**

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

- **Aggregates**

  \[ s(Y) \leftarrow r(Y), 2 \#\text{count} \{ p(X,Y) : q(X) \} 7 : r(Y) \]

  – also:

  \#\text{sum}, \#\text{avg}, \#\text{min}, \#\text{max}, \#\text{even}, \#\text{odd}
Language Constructs

- **Variables (over the Herbrand Universe)**

  - `p(X) :- q(X)` over constants `{a, b, c}` stands for `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`

- **Conditional Literals**

  - `p :- q(X) : r(X)` given `r(a), r(b), r(c)` stands for `p :- q(a), q(b), q(c)`

- **Disjunction**

  - `p(X) | q(X) :- r(X)`

- **Integrity Constraints**

  - `:- q(X), p(X)`

- **Choice**

  - `2 { p(X,Y) : q(X) } 7 :- r(Y)`

- **Aggregates**

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

- **Variables (over the Herbrand Universe)**
  
  \[ p(X) :- q(X) \]

  over constants \{a, b, c\} stands for
  
  \[ p(a) :- q(a), p(b) :- q(b), p(c) :- q(c) \]

- **Conditional Literals**

  \[ p :- q(X) : r(X) \]

  given \( r(a), r(b), r(c) \) stands for
  
  \[ p :- q(a), q(b), q(c) \]

- **Integrity Constraints**

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

- **Choice**

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

- **Aggregates**

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

  also: \#sum, \#avg, \#min, \#max, \#even, \#odd
Modeling

• For solving a problem class $C$ for a problem instance $I$, encode
  1. the problem instance $I$ as a set $P_I$ of facts and
  2. the problem class $C$ as a set $P_C$ of rules
such that the solutions to $C$ for $I$ can be (polynomially) extracted from the stable models of $P_I \cup P_C$
Modeling

• For solving a problem class $C$ for a problem instance $I$, encode
  1. the problem instance $I$ as a set $P_I$ of facts and
  2. the problem class $C$ as a set $P_C$ of rules
such that the solutions to $C$ for $I$ can be (polynomially) extracted from the stable models of $P_I \cup P_C$

• $P_I$ is (still) called problem instance
• $P_C$ is often called the problem encoding
Modeling

- For solving a problem class \( C \) for a problem instance \( I \), encode
  1. the problem instance \( I \) as a set \( P_I \) of facts and
  2. the problem class \( C \) as a set \( P_C \) of rules

such that the solutions to \( C \) for \( I \) can be (polynomially) extracted from the stable models of \( P_I \cup P_C \)

- \( P_I \) is (still) called problem instance
- \( P_C \) is often called the problem encoding

- An encoding \( P_C \) is uniform, if it can be used to solve all its problem instances
  That is, \( P_C \) encodes the solutions to \( C \) for any set \( P_I \) of facts
Example 3-Colorability

- Vertices are represented with predicates \texttt{node}(X);
- Edges are represented with predicates \texttt{edge}(X, Y).

Question: Is there a valid assignment of three colors for an input graph $G$ such that no two adjacent vertices have the same color?
Graph coloring

\[ \text{node}(1 \ldots 6). \]
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).
Graph coloring

node(1..6).

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

col(r). col(b). col(g).
Graph coloring

node(1..6).

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

col(r).  col(b).  col(g).
Graph coloring

node(1..6).

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

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

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

node(1..6).

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

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

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

:- edge(X,Y), color(X,C), color(Y,C).
Graph coloring

node(1..6).

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

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

1 { color(X,C) : col(C) } 1 :- node(X).
:- edge(X,Y), color(X,C), color(Y,C).
Graph coloring

node(1..6).

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

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

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

node(1..6).

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

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

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

:- edge(X,Y), color(X,C), color(Y,C).
ASP solving process

Modeling

Logic Program → Grounder → Solver → Stable Models

Solving

Problem

Solution

Interpreting

TU Dresden, 14th November 2019
Graph coloring: Grounding

$ gringo --text color.lp
Graph coloring: Grounding

$\text{gringo --text color.lp}$

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

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

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

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

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

Problem

Logic Program

Grounder

Solver

Stable Models

Solution

Modeling

Interpreting

Solving
Graph coloring: Solving

$ gringo color.lp | clasp 0
Graph coloring: Solving

$ gringo color.lp | clasp 0

class version 2.1.0
Reading from stdin
Solving...
Answer: 1
division(1,2) ... col(r) ... node(1) ... color(6,b) color(5,g) color(4,b) color(3,r) color(2,r) color(1,g)
Answer: 2
division(1,2) ... col(r) ... node(1) ... color(6,r) color(5,g) color(4,r) color(3,b) color(2,b) color(1,g)
Answer: 3
division(1,2) ... col(r) ... node(1) ... color(6,g) color(5,b) color(4,g) color(3,r) color(2,r) color(1,b)
Answer: 4
division(1,2) ... col(r) ... node(1) ... color(6,r) color(5,b) color(4,r) color(3,g) color(2,g) color(1,b)
Answer: 5
division(1,2) ... col(r) ... node(1) ... color(6,g) color(5,r) color(4,g) color(3,b) color(2,b) color(1,r)
Answer: 6
division(1,2) ... col(r) ... node(1) ... color(6,b) color(5,r) color(4,b) color(3,g) color(2,g) color(1,r)
SATISFIABLE

Models : 6
Time : 0.002s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time : 0.000s
Problem solving in ASP: Reasoning Modes

- Problem
  - Modeling
    - Logic Program
  - Solving
- Solution
  - Interpreting
    - Stable Models

Reasoning Modes

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

and combinations of them

† without solution recording
‡ without solution enumeration
References

Martin Gebser, Benjamin Kaufmann Roland Kaminski, and Torsten Schaub.
Answer Set Solving in Practice.
Synthesis Lectures on Artificial Intelligence and Machine Learning.
doi=10.2200/S00457ED1V01Y201211AIM019.

Michael Gelfond and Vladimir Lifschitz.
Classical negation in logic programs and disjunctive databases.

• See also: http://potassco.sourceforge.net