Answer Set Programming

- Answer Set Programs
- Answer Set Semantics
- Implementation Techniques
- Using Answer Set Programming
Example ASP: 3-Coloring

Problem: For a graph \((V, E)\) find an assignment of one of 3 colors to each vertex such that no adjacent vertices share a color.

\[
\begin{align*}
\text{clrd}(V,1) & :­ \ not \ clrd(V,2), \ not \ clrd(V,3), \ \text{vtx}(V). \\
\text{clrd}(V,2) & :­ \ not \ clrd(V,1), \ not \ clrd(V,3), \ \text{vtx}(V). \\
\text{clrd}(V,3) & :­ \ not \ clrd(V,1), \ not \ clrd(V,2), \ \text{vtx}(V). \\
& :­ \ \text{edge}(V,U), \ \text{clrd}(V,C), \ \text{clrd}(U,C).
\end{align*}
\]

\[
\text{vtx}(a). \ \text{vtx}(b). \ \text{vtx}(c). \ \text{edge}(a,b). \ \text{edge}(a,c). \ ...
\]
ASP in Practice

- Compact, easily maintainable representation
- Roots: logic programming
- Solutions = Answer sets to logic program
Some Applications

- Constraint satisfaction
- Planning, Routing
- Computer-aided verification
- Security analysis
- Configuration
- Diagnosis
ASP vs. Prolog

- Prolog not directly suitable for ASP
  - Models vs. proofs + answer substitutions
  - Prolog not entirely declarative

- Answer set semantics: alternative semantics for negation-as-failure

- Existing ASP Systems: CLINGO, SMODELS, DLV and others
Answer Set Semantic

- A logic program clause
  \[ A \leftarrow B_1, \ldots, B_m, \text{not} \ C_1, \ldots, \text{not} \ C_n \quad (m \geq 0, n \geq 0) \]
  
is seen as constraint on an answer (model): if \( B_1, \ldots, B_m \) are in the answer and none of \( C_1, \ldots, C_m \) is, then must \( A \) be included in the answer.

- Answer sets should be **minimal**
- Answer sets should be **justified**
Answer Sets: Example (1)

\[
p :- \text{not } q.
\]
\[
r :- p.
\]
\[
s :- r, \text{not } p.
\]

The answer set is \{p, r\}

- \{p\} is not an answer (because it's not a model)

- \{r, s\} is not an answer (because \(r\) included for no reason)
Answer Sets: Example (2)

\[ p \leftarrow q. \]
\[ p \leftarrow r. \]
\[ q \leftarrow \text{not } r. \]
\[ r \leftarrow \text{not } q. \]

There are two answers: \{p, q\} and \{p, r\}.

Note that in Prolog, \( p \) is not derivable.
Consider a program $P$ of ground clauses

$$A \leftarrow B_1, \ldots, B_m, \text{not } C_1, \ldots, \text{not } C_n \quad (m \geq 0, n \geq 0)$$

Let $S$ be a set of ground atoms.

- **Reduct $P^S$:**
  - delete each clause with some $\text{not } C_i$ such that $C_i \in S$
  - delete each $\text{not } C_i$ such that $C_i \notin S$

- $S$ answer set (also called stable model) $:\iff S = \text{least-model}(P^S)$
Properties

- Programs can have multiple answer sets
  
  \[
  p_1 : \neg q_1. \quad q_1 : \neg p_1. \\
  \vdots \\
  p_n : \neg q_n. \quad q_n : \neg p_n. 
  \]

  This program has \(2^n\) answers

- Programs can have no answers
  
  \[
  p : \neg q. \\
  q : p. 
  \]
Properties (ctd)

- A stratified program has a unique answer (= the standard model).
- Checking whether a set of atoms is a stable model can be done in linear time.
- Deciding whether a program has a stable model is NP-complete.
Programs with Variables and Functions

- Semantics: Herbrand models

- Clause seen as shorthand for all its ground instances

  \[ \text{clrd}(V,1) :- \neg \text{clrd}(V,2), \neg \text{clrd}(V,3), \text{vtx}(V). \]

  stands for

  \[ \text{clrd}(a,1) :- \neg \text{clrd}(a,2), \neg \text{clrd}(a,3), \text{vtx}(a). \]
  \[ \text{clrd}(b,1) :- \neg \text{clrd}(b,2), \neg \text{clrd}(b,3), \text{vtx}(b). \]
  \[ \ldots \]

- Constraint

  \[ \leftarrow B_1, \ldots, B_m, \neg C_1, \ldots, \neg C_n \]

  shorthand for \[ \text{false} \leftarrow B_1, \ldots, B_m, \neg C_1, \ldots, \neg C_n, \not\text{false} \]
Example ASP: 3-Coloring

```
clrd(V,1) :- not clrd(V,2), not clrd(V,3), vtx(V).
clrd(V,2) :- not clrd(V,1), not clrd(V,3), vtx(V).
clrd(V,3) :- not clrd(V,1), not clrd(V,2), vtx(V).
:- edge(V,U), clrd(V,C), clrd(U,C).

vtx(a). vtx(b). vtx(c). edge(a,b). edge(a,c).
```

Each answer set is a valid coloring, for example:

```
{clrd(a,1), clrd(b,2), clrd(c,2)}
```
Generalization: Classical Negation

- Rules built using classical literals (not just atoms)
- Answers are sets of literals
- Example:
  
  \[ p \leftarrow \text{not} \neg q \]
  \[ \neg q \leftarrow \text{not} \ p \]

  An answer is \{\neg q\}
Generalization: Classical Negation (ctd)

- Classical negation can be handled by normal programs:
  - treat $\neg A$ as a new atom (renaming)
  - add the constraint $\leftarrow A, \neg A$

Example:

\[
\begin{align*}
p & :\neg q' \\
q' & :\neg p \\
& : p, p' \\
& : q, q'
\end{align*}
\]

has the answer \{q'\}
Generalization: Disjunction

- Rules can have disjunctions in the head
- Direct generalization of answer set semantics

Example:

\[ p \lor q : \neg p \]

has the only answer \{q\}

Another example:

\[ p \lor q : \neg p \]
\[ p : \neg q \]

has no answer
ASP Solver: Architecture

Two challenging tasks: handle complex data; search

Two-layer architecture:

- **Grounding** handles complex data: A set of ground clauses is generated which preserves the models

- **Model search** uses special-purpose search procedures
Grounding: Domain Restrictions

- Domain-restricted programs guarantee decidability.

- Domain-restricted programs consist of two parts:
  1. Domain predicate definitions (a stratified clause set), where each variable occurs in a positive domain predicate defined in an earlier stratum;
  2. Clauses where each variable occurs in a positive domain predicate in the body.

- The domain predicate definitions have a unique answer, which is subset of every solution to the program.

- Only those ground instances of clauses need to be generated where the domain predicates in the body are true.
Example: Domain Predicate Definitions

col(1). col(2). col(3).

r(a,b). r(a,c). ...

d(U) :- r(V,U).

tr(V,U) :- r(V,U).

tr(V,U) :- r(V,Z), tr(Z,U), d(U).

edge(t(V), t(U)) :- tr(V,U), not tr(U,U), not tr(V,V).

vtx(V) :- edge(V,U).

vtx(U) :- edge(V,U).
Example: Domain-Restricted Clauses

\[
\begin{align*}
\text{clrd}(V,1) & : - \text{not clrd}(V,2), \text{not clrd}(V,3), \text{vtx}(V). \\
\text{clrd}(V,2) & : - \text{not clrd}(V,1), \text{not clrd}(V,3), \text{vtx}(V). \\
\text{clrd}(V,3) & : - \text{not clrd}(V,1), \text{not clrd}(V,2), \text{vtx}(V). \\
& : - \text{edge}(V, U), \text{col}(C), \text{clrd}(V,C), \text{clrd}(U,C).
\end{align*}
\]
Example: Grounding

Suppose that the unique stable model for the definition of the domain predicate $\text{vtx}(V)$ contains $\text{vtx}(v_1), \ldots, \text{vtx}(v_n)$

Then for the clause

$$\text{clrd}(V,1) :- \text{not clrd}(V,2), \text{not clrd}(V,3), \text{vtx}(V).$$

grounding produces

$$\text{clrd}(v_1,1) :- \text{not clrd}(v_1,2), \text{not clrd}(v_1,3).$$

$$\ldots$$

$$\text{clrd}(v_n,1) :- \text{not clrd}(v_n,2), \text{not clrd}(v_n,3).$$
Search

• Backtracking over truth-values for atoms

• Each node consists of a model candidate (set of literals)

• Propagation rules are applied after each choice
Propagation Rules

- A propagation rule extends a model candidate by one or more new literals.

- Example: Given $q \leftarrow p_1, \text{not } p_2$ and candidate $\{p_1, \text{not } q\}$: derive $p_2$

- Propagation rules need to be correct: If $L$ is derived from model candidate $A$ then $L$ holds in every stable model compatible with $A$. 
Example: Propagation Rule “Upper Bound”

Consider program $P$ and candidate model $A$

Let $P'$ be all clauses in $P$

- whose body is not false under $A$
- without negative body literals

If $p \not\in \text{least-model}(P')$ derive not $p$

\[
P: \quad p_2 \leftarrow p_1, \text{ not } q_1. \quad A: \{q_2\} \quad P': p_2 \leftarrow p_1.
\]
\[
p_1 \leftarrow p_2, \text{ not } q_1. \quad \quad \quad \quad p_1 \leftarrow p_2.
\]
\[
p_2 \leftarrow \text{ not } q_2.
\]

Derive: \text{not } p_1, \text{ not } p_2, \text{ not } q_1, \text{ not } q_2
## Schema of Local Propagation Rules

<table>
<thead>
<tr>
<th></th>
<th>Only clauses for $q$</th>
<th>Candidate</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(R_1)$</td>
<td>$q \leftarrow p_1, \text{ not } p_2$</td>
<td>$p_1, \text{ not } p_2$</td>
<td>$q$</td>
</tr>
<tr>
<td>$(R_2)$</td>
<td>$q \leftarrow p_1, \text{ not } p_2$</td>
<td>$p_2, \text{ not } p_3$</td>
<td>not $q$</td>
</tr>
<tr>
<td>$(R_3)$</td>
<td>$q \leftarrow p_1, \text{ not } p_2$</td>
<td>$q$</td>
<td>$p_1, \text{ not } p_2$</td>
</tr>
<tr>
<td>$(R_4)$</td>
<td>$q \leftarrow p_1, \text{ not } p_2$</td>
<td>not $q, p_1$</td>
<td>$p_2$</td>
</tr>
</tbody>
</table>
Example

\[ f : \neg g, \neg h \]
\[ g : \neg f, \neg h \]
\[ f : g \]

\[ \neg f \]
\[ \neg g \]

\[ (R_2) \]

\[ \neg h \]

\[ (R_2) \]

\[ (R_4) \]
\[ g \]
\[ (R_1) \]
\[ f \]

\[ \times \]

\[ \text{stable} \]
Lookahead

Given a program $P$ and a candidate model $A$.

If, for a literal $L$, $\text{propagate}(P, A \cup \{L\})$ contains a conflict (some $p$ together with $\neg p$), derive the complement of $L$. 
Search Heuristics

Heuristics to select the next atom for splitting the search tree:

- an atom with the maximal number of occurrences in clauses of minimal size
- an atom with the maximal number of propagations after the split
- an atom with the smallest remaining search space after split + propagation
Using ASPs (Example 1): Hamiltonian Cycles

- **A Hamiltonian cycle**: a closed path that visits all vertices of a graph exactly once
- **Input**: a graph
  - \( \text{vtx}(a), \ldots \)
  - \( \text{edge}(a, b), \ldots \)
  - \( \text{initialvtx}(a) \)
- **Weight** atoms in ASP:

\[
m \{ p : d(x) \} n
\]

means that an answer contains at least \( m \) and at most \( n \) different \( p \)-instances which satisfy \( d(x) \). If \( m \) is omitted, there is no lower bound; if \( n \) is omitted, there is no upper bound.
Hamiltonian Cycles (ctd)

- Candidate answer sets: subsets of edges
- Generator (using a weight atom):

  \[
  \{ \text{hc} (X, Y) \} \ :- \ \text{edge} (X, Y)
  \]

- Answer sets for the generator given a graph:
  
  input graph
  + a subset of the ground facts \( \text{hc}(a, b) \) for which there is \( \text{edge}(a, b) \)
Hamiltonian Cycles (ctd)

- Tester(1): Each vertex has at most one chosen incoming and one outcoming edge

  \[\neg \text{hc}(X, Y), \neg \text{hc}(X, Z), \text{edge}(X, Y), \text{edge}(X, Z), Y \neq Z.\]
  \[\neg \text{hc}(Y, X), \neg \text{hc}(Z, X), \text{edge}(Y, X), \text{edge}(Z, X), Y \neq Z.\]

- Only subsets of chosen edges \(\text{hc}(a, b)\) forming paths (possibly closed) pass this test
Hamiltonian Cycles (ctd)

- **Tester(2):** Every vertex is reachable from a given initial vertex through chosen $hc(a, b)$ edges

  $$\begin{align*}
  :- \text{vtx}(X), \text{not } r(X). \\
  r(Y) :- hc(X, Y), \text{edge}(X, Y), \text{initialvtx}(X). \\
  r(Y) :- hc(X, Y), \text{edge}(X, Y), r(X), \text{not initialvtx}(X).
  \end{align*}$$

- Only Hamiltonian cycles pass both tests
Hamiltonian Cycles (ctd)

- Using more weight atoms enables even more compact encoding

- Tester(1) using 2 variables:

  \[
  :- 2 \{ \text{hc}(X,Y) : \text{edge}(X,Y) \}, \text{vtx}(X).
  
  :- 2 \{ \text{hc}(X,Y) : \text{edge}(X,Y) \}, \text{vtx}(Y).
  \]
Hamiltonian Cycles (ctd): Undirected Cycles

- **Instance \((V,E)\):**
  
  \[
  \begin{align*}
  &\text{vtx}(v), \\
  &\text{edge}(v,u). \quad \text{% one fact for each edge in } E
  \end{align*}
  \]

- **Generator:**
  
  \[
  2 \{ \text{hc}(V,U) : \text{edge}(V,U), \\
  \quad \text{hc}(W,V) : \text{edge}(W,V) \} 2 :- \text{vtx}(V).
  \]

- **Tester:**
  
  \[
  \begin{align*}
  &r(V) :- \text{initialvtx}(V). \\
  &r(V) :- \text{hc}(V,U), \text{edge}(V,U), r(U). \\
  &r(V) :- \text{hv}(U,V), \text{edge}(U,V), r(U). \\
  &:- \text{vtx}(V), \text{not } r(V).
  \end{align*}
  \]
Using ASPs (Example 2): Verification

- Verify, on the basis of a given formal specification, that a dynamic system satisfies desirable properties
- Example:

```
X |
---|
O |
---|
X |
```

Given a formal specification of Tic-Tac-Toe, ASP can be used to verify that it is a turn-taking game and that no cell ever contains two symbols.
Formal Specification: Initial State

init(cell(1,1,b)).
init(cell(1,2,b)).
init(cell(1,3,b)).
init(cell(2,1,b)).
init(cell(2,2,b)).
init(cell(2,3,b)).
init(cell(3,1,b)).
init(cell(3,2,b)).
init(cell(3,3,b)).
init(control(xplayer)).
Formal Specification: State Transitions

\[
\text{legal}(P, \text{mark}(X,Y)) \leftarrow \text{true}(\text{cell}(X,Y,b)), \\
\quad \text{true}(\text{control}(P)).
\]

\[
\text{legal}(\text{xplayer}, \text{noop}) \leftarrow \text{true}(\text{cell}(X,Y,b)), \\
\quad \text{true}(\text{control}(\text{oplayer})).
\]

\[
\text{legal}(\text{oplayer}, \text{noop}) \leftarrow \text{true}(\text{cell}(X,Y,b)), \\
\quad \text{true}(\text{control}(\text{xplayer})).
\]
**Formal Specification: State Change**

\[
\text{next(cell}(M,N,x)) :­ \text{does(xplayer,mark}(M,N)) .
\]

\[
\text{next(cell}(M,N,o)) :­ \text{does(oplayer,mark}(M,N)) .
\]

\[
\text{next(cell}(M,N,W)) :­ \text{true(cell}(M,N,W)), \ W!=b .
\]

\[
\text{next(cell}(M,N,b)) :­ \text{true(cell}(M,N,b)), \\
\text{does(P,mark}(J,K)), \ M!=J .
\]

\[
\text{next(cell}(M,N,b)) :­ \text{true(cell}(M,N,b)), \\
\text{does(P,mark}(J,K)), \ N!=K .
\]

\[
\text{next(control}(xplayer)) :­ \text{true(control}(oplayer)) .
\]

\[
\text{next(control}(oplayer)) :­ \text{true(control}(xplayer)) .
\]
Verification (ctd)

- Properties of dynamic systems are verified inductively
- Induction base:

```prolog
player(xplayer).
player(oplayer).
t0 :- 1 { init(control(X)) : player(X) } 1.
    :- t0.
```

- This program has no answer set, which proves the fact that initially exactly one player has the control.
Verification (ctd)

- State generator for the induction step:

  \[
  \text{coordinate}(1..3).
  \text{symbol}(x). \text{symbol}(o). \text{symbol}(b).
  \]

  \[
  \text{tdomain}(\text{cell}(X,Y,C)) :\quad \text{coordinate}(X), \text{coordinate}(Y), \text{symbol}(C).
  \]

  \[
  \text{tdomain}(\text{control}(X)) :\quad \text{player}(X).
  \]

  \[
  \{ \text{true}(T) : \text{tdomain}(T) \}.
  \]

- Transition generator for the induction step:

  \[
  \text{ddomain}(\text{mark}(X,Y)) :\quad \text{coordinate}(X), \text{coordinate}(Y).
  \]

  \[
  \text{ddomain}(\text{noop}).
  \]

  \[
  1 \{ \text{does}(P,M) : \text{ddomain}(M) \} 1 :\quad \text{player}(P).
  \]
Verification (ctd)

- Tester(1): Every transition must be legal
  
  \[-\text{does}(P,M), \text{not legal}(P,M).\]

- Tester(2): Induction hypothesis
  
  \[t_0 :\mathord{-} 1 \{ \text{true(control}(X)) : \text{player}(X) \} 1.\]
  
  \[-\text{not} \ t_0.\]

- Induction step
  
  \[t :\mathord{-} 1 \{ \text{next(control}(X)) : \text{player}(X) \} 1.\]
  
  \[-t.\]

- This program has no answer, which proves the claim that in every reachable state exactly one player has the control.
Verification (ctd)

- Induction base to prove that cells have unique contents:

  \[
  t_0(X,Y) \leftarrow 1 \{ \text{init(cell}(X,Y,Z)) : \text{symbol}(Z) \} 1.
  \]

  \[
  t_0 \leftarrow \neg t_0(X,Y).
  \]

  \[
  \neg t_0 \leftarrow \neg t_0.
  \]

- This program has no answer set, which proves the claim.
Verification (ctd)

- Induction hypothesis

\[
\begin{align*}
t_0(X,Y) & : - 1 \{ \text{true(cell}(X,Y,Z)) : \text{symbol}(Z) \} \ 1. \\
t_0 & : - \text{not } t_0(X,Y). \\
& : - t_0.
\end{align*}
\]

- Induction step to prove that cells have unique contents

\[
\begin{align*}
t(X,Y) & : - 1 \{ \text{next(cell}(X,Y,Z)) : \text{symbol}(Z) \} \ 1. \\
t & : - \text{not } t(X,Y). \\
& : - \text{not } t.
\end{align*}
\]

- This program has an answer set! Need to add uniqueness-of-control:

\[
\begin{align*}
p & : - 1 \{ \text{true(control}(X)) : \text{player}(X) \} \ 1. \\
& : - \text{not } p.
\end{align*}
\]

Now the program has no answer set, which proves the claim.
Objectives

- Answer Set Programs
- Answer Set Semantics
- Implementation Techniques
- Using Answer Set Programming