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.

\[
\begin{align*}
\text{clrd}(V,1) & :\neg \text{clrd}(V,2), \neg \text{clrd}(V,3), \text{vtx}(V). \\
\text{clrd}(V,2) & :\neg \text{clrd}(V,1), \neg \text{clrd}(V,3), \text{vtx}(V). \\
\text{clrd}(V,3) & :\neg \text{clrd}(V,1), \neg \text{clrd}(V,2), \text{vtx}(V). \\
& : \text{edge}(V,U), \text{clrd}(V,C), \text{clrd}(U,C).
\end{align*}
\]

\text{vtx}(a). \text{vtx}(b). \text{vtx}(c). \text{edge}(a,b). \text{edge}(a,c). \ldots
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)

\[\begin{align*}
p & : \text{not } q. \\
r & : \text{p.} \\
s & : \text{r, not p.}
\end{align*}\]

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

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

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) :$<=>$ $S = \text{least-model}(P^S)$
Properties

- Programs can have multiple answer sets

\[
p_1 :\neg q_1. \quad q_1 :\neg p_1. \\
\vdots \quad \vdots \\
p_n :\neg q_n. \quad q_n :\neg p_n.
\]

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

  stands for

  \[ \text{clrd}(a,1) :- \text{not clrd}(a,2), \text{not clrd}(a,3), \text{vtx}(a). \]
  \[ \text{clrd}(b,1) :- \text{not clrd}(b,2), \text{not clrd}(b,3), \text{vtx}(b). \]
  ...

- Constraint

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

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

clr(V, 1) :- not clrd(V, 2), not clrd(V, 3), vtx(V).
clr(V, 2) :- not clrd(V, 1), not clrd(V, 3), vtx(V).
clr(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:

\[
\begin{align*}
p & :\neg \text{not } q' \\
q' & :\neg \text{not } p \\
& \quad :\neg p, p' \\
& \quad \leftarrow q, q'
\end{align*}
\]

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

\[
\text{clrd}(V,1) \leftarrow \neg \text{clrd}(V,2), \neg \text{clrd}(V,3), \text{vtx}(V).
\]
\[
\text{clrd}(V,2) \leftarrow \neg \text{clrd}(V,1), \neg \text{clrd}(V,3), \text{vtx}(V).
\]
\[
\text{clrd}(V,3) \leftarrow \neg \text{clrd}(V,1), \neg \text{clrd}(V,2), \text{vtx}(V).
\]
\[
\text{:- edge}(V,U), \text{col}(C), \text{clrd}(V,C), \text{clrd}(U,C).
\]
Example: Grounding

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

Then for the clause

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

grounding produces

\[
clrd(v_1,1) :- \neg clrd(v_1,2), \neg clrd(v_1,3).
\]

\[
\ldots
\]

\[
clrd(v_n,1) :- \neg clrd(v_n,2), \neg clrd(v_n,3).
\]
Search

- Backtracking over truth-values for atoms

![Diagram]

- 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 $q ← p_1, \text{not } p_2$ 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 $p \not\in$ least-model ($P'$) derive not $p$

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

\[
p_1 \leftarrow p_2, \ not \ q_1. \quad \quad \quad \quad p_1 \leftarrow p_2.
\]

\[
p_2 \leftarrow not \ q_2.
\]

Derive: not $p_1$, not $p_2$, not $q_1$, not $q_2$
## Schema of Local Propagation Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Candidate</th>
<th>Derive</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(R_1)$</td>
<td>$q \leftarrow p_1, \lnot p_2$</td>
<td>$p_1, \lnot p_2$</td>
</tr>
<tr>
<td>$(R_2)$</td>
<td>$q \leftarrow p_1, \lnot p_2$</td>
<td>$p_2, \lnot p_3$</td>
</tr>
<tr>
<td>$(R_3)$</td>
<td>$q \leftarrow p_1, \lnot p_2$</td>
<td>$q$</td>
</tr>
<tr>
<td>$(R_4)$</td>
<td>$q \leftarrow p_1, \lnot p_2$</td>
<td>$\lnot q, p_1$</td>
</tr>
</tbody>
</table>
Example

\[\begin{align*}
f & : \neg g, \neg h \\
g & : \neg f, \neg h \\
f & : g
\end{align*}\]
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 $\text{not } p$), 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
  - \texttt{vtx(a), ...}
  - \texttt{edge(a,b), ...}
  - \texttt{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 outgoing edge

\[
\begin{align*}
\text{:- } & \text{hc}(X, Y), \text{hc}(X, Z), \text{edge}(X, Y), \text{edge}(X, Z), Y \neq Z. \\
\text{:- } & \text{hc}(Y, X), \text{hc}(Z, X), \text{edge}(Y, X), \text{edge}(Z, X), Y \neq Z.
\end{align*}
\]

- 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

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

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

Using more weight atoms enables even more compact encoding

Tester(1) using 2 variables:

\[
\begin{align*}
&:- 2 \{ \text{hc}(X,Y) : \text{edge}(X,Y) \}, \text{vtx}(X). \\
&:- 2 \{ \text{hc}(X,Y) : \text{edge}(X,Y) \}, \text{vtx}(Y).
\end{align*}
\]
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**:
  
  \[
  2 \{ \text{hc}(V,U) : \text{edge}(V,U), \\
  \text{hc}(W,V) : \text{edge}(W,V) \} 2 :\!-\! \text{vtx}(V).
  \]

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

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

- Example:

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

\begin{verbatim}
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)).
\end{verbatim}
Formal Specification: State Transitions

\begin{align*}
\text{legal}(P, \text{mark}(X,Y)) & :\neg \text{true}(\text{cell}(X,Y,b)), \\
& \quad \text{true}(\text{control}(P)).
\end{align*}

\begin{align*}
\text{legal}(\text{xplayer}, \text{noop}) & :\neg \text{true}(\text{cell}(X,Y,b)), \\
& \quad \text{true}(\text{control}(\text{oplayer})).
\end{align*}

\begin{align*}
\text{legal}(\text{oplayer}, \text{noop}) & :\neg \text{true}(\text{cell}(X,Y,b)), \\
& \quad \text{true}(\text{control}(\text{xplayer})).
\end{align*}
Formal Specification: State Change

next(cell(M,N,x)) :- does(xplayer,mark(M,N)).

next(cell(M,N,o)) :- does(oplayer,mark(M,N)).

next(cell(M,N,W)) :- true(cell(M,N,W)), W!=b.

next(cell(M,N,b)) :- true(cell(M,N,b)),
                  does(P,mark(J,K)),
                  M!=J.

next(cell(M,N,b)) :- true(cell(M,N,b)),
                  does(P,mark(J,K)),
                  N!=K.

next(control(xplayer)) :- true(control(oplayer)).

next(control(oplayer)) :- true(control(xplayer)).
Verification (ctd)

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

\[
\begin{align*}
\text{player(xplayer).} \\
\text{player(oplayer).} \\
\text{t0 :- 1 \{ init(control(X)) : player(X) \} 1.} \\
\text{:- t0.}
\end{align*}
\]

- 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:

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

  \[
  \begin{align*}
  \text{tdomain} \left( \text{cell}(X,Y,C) \right) :& \quad \text{coordinate}(X), \text{coordinate}(Y), \\
  & \quad \text{symbol}(C).
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{tdomain} \left( \text{control}(X) \right) :& \quad \text{player}(X).
  \end{align*}
  \]

  \[
  \begin{align*}
  \{ \text{true}(T) : \text{tdomain}(T) \}. 
  \end{align*}
  \]

- Transition generator for the induction step:

  \[
  \begin{align*}
  \text{ddomain} \left( \text{mark}(X,Y) \right) :& \quad \text{coordinate}(X), \text{coordinate}(Y). \\
  \text{ddomain} \left( \text{noop} \right).
  \end{align*}
  \]

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

- **Tester(1):** Every transition must be legal
  
  ```prolog
  :- does(P,M), not legal(P,M).
  ```

- **Tester(2):** Induction hypothesis
  
  ```prolog
  t0 :- 1 { true(control(X)) : player(X) } 1.
  :- not t0.
  ```

- **Induction step**
  
  ```prolog
  t :- 1 { next(control(X)) : 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:

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

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

- Induction hypothesis

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

- Induction step to prove that cells have unique contents

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

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

\[
p :- 1 \{ \text{true}(\text{control}(X)) : \text{player}(X) \} 1.
:- \text{not } p.
\]

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

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