

Answer Set Programming

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

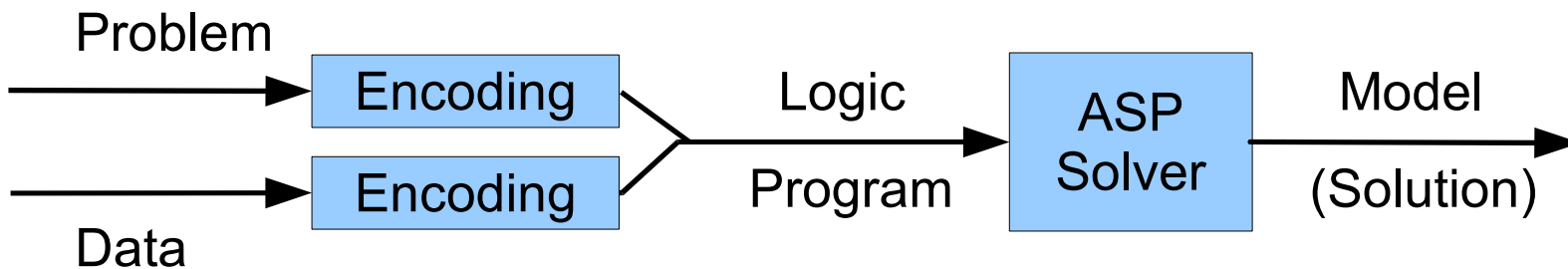
Example ASP: 3-Coloring

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.

```
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). ...
```

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

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

is seen as constraint on an answer (model): if B_1, \dots, B_m are in the answer and none of C_1, \dots, 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 :- not q.`

`r :- p.`

`s :- r, 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 :- q.`

`p :- r.`

`q :- not r.`

`r :- not q.`

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

Note that in Prolog, `p` is not derivable.

Answer Sets: Definition

Consider a program P of **ground** clauses

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

Let S be a set of ground atoms.

- **Reduct** P^S : \Leftrightarrow
 - 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**) : \Leftrightarrow $S = \text{least-model}(P^S)$

Properties

- Programs can have multiple answer sets

$$\begin{array}{ll} p_1 \text{ :- not } q_1. & q_1 \text{ :- not } p_1. \\ \vdots & \vdots \\ p_n \text{ :- not } q_n. & q_n \text{ :- not } p_n. \end{array}$$

This program has 2^n answers

- Programs can have no answers

$$\begin{array}{l} p \text{ :- not } q. \\ q \text{ :- } p. \end{array}$$

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

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

stands for

`clrd(a,1) :- not clrd(a,2), not clrd(a,3), vtx(a).`

`clrd(b,1) :- not clrd(b,2), not clrd(b,3), vtx(b).`

...

- **Constraint**

$\leftarrow B_1, \dots, B_m, \text{not } C_1, \dots, \text{not } C_n$

shorthand for **false** $\leftarrow B_1, \dots, B_m, \text{not } C_1, \dots, \text{not } C_n$, **not 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 :- not ¬q  
¬q :- 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:

```
p   :- not q'  
q'  :- not p  
    :- p, p'  
    :- q, q'
```

has the answer {q' }

Generalization: Disjunction

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

- Example:

$p \vee q \text{ :- not } p$

has the only answer $\{q\}$

- Another example:

$p \vee q \text{ :- not } p$
 $p \text{ :- } q$

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

```
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), col(C), clrd(V,C), clrd(U,C).
```

Example: Grounding

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

Then for the clause

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

grounding produces

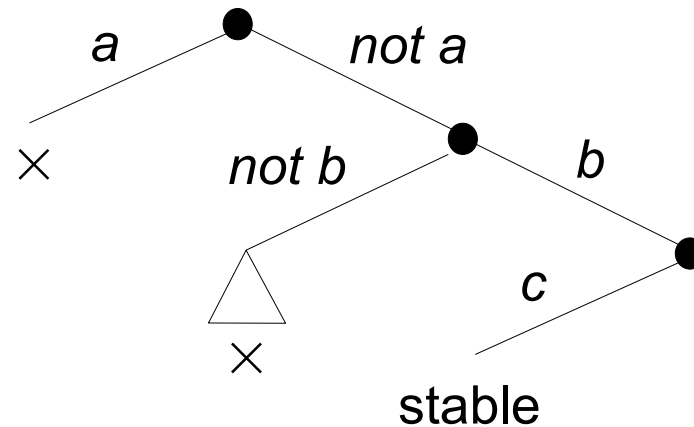
```
clrd(v1,1) :- not clrd(v1,2), not clrd(v1,3).
```

...

```
clrd(vn,1) :- not clrd(vn,2), not clrd(vn,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 $p \notin$ least-model (P') derive not p

$P:$	$p_2 :- p_1, \text{ not } q_1.$	$A: \{q_2\}$	$P':$	$p_2 :- p_1.$
	$p_1 :- p_2, \text{ not } q_1.$			$p_1 :- p_2.$
	$p_2 :- \text{ not } q_2.$			

Derive: not p_1 , not p_2 , not q_1 , not q_2

Schema of Local Propagation Rules

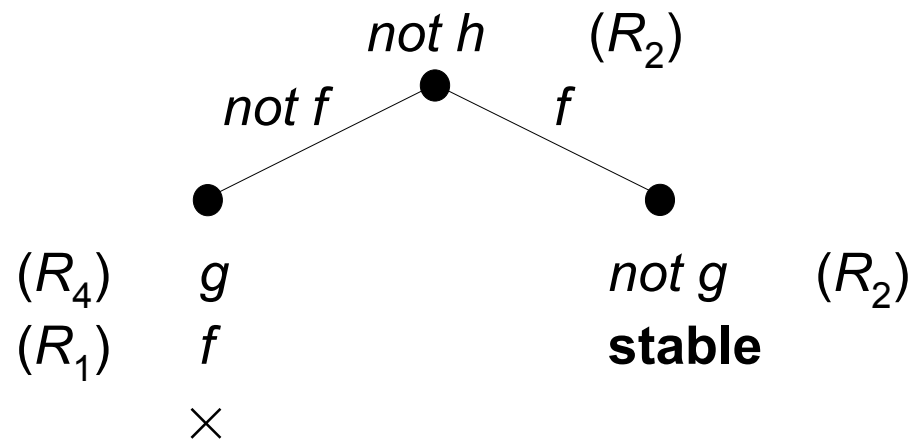
	Only clauses for q	Candidate	Derive
(R_1)	$q \leftarrow p_1, \text{ not } p_2$	$p_1, \text{ not } p_2$	q
(R_2)	$q \leftarrow p_1, \text{ not } p_2$ $q \leftarrow p_3, \text{ not } p_4$	$p_2, \text{ not } p_3$	$\text{not } q$
(R_3)	$q \leftarrow p_1, \text{ not } p_2$	q	$p_1, \text{ not } p_2$
(R_4)	$q \leftarrow p_1, \text{ not } p_2$	$\text{not } q, p_1$	p_2

Example

$f :- \text{not } g, \text{not } h$

$g :- \text{not } f, \text{not } h$

$f :- g$



Lookahead

Given a program P and a candidate model A .

If, for a literal L , **propagate**($P, A \cup \{ L \}$) contains a conflict (some p together with *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

- `vtx(a), ...`
- `edge(a,b), ...`
- `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
 - :- $hc(X,Y), hc(X,Z), edge(X,Y), edge(X,Z), Y \neq Z.$
 - :- $hc(Y,X), hc(Z,X), edge(Y,X), edge(Z,X), Y \neq Z.$
- Only subsets of chosen edges $hc(a,b)$ 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

```
:- vtx(X), not r(X).
```

```
r(Y) :- hc(X,Y), edge(X,Y), initialvtx(X).
```

```
r(Y) :- hc(X,Y), edge(X,Y), r(X), not initialvtx(X).
```

- Only Hamiltonian cycles pass both tests

Hamiltonian Cycles (ctd)

- Using more weight atoms enables even more compact encoding
- Tester(1) using 2 variables:

```
:- 2 { hc(X,Y) : edge(X,Y) }, vtx(X).  
:- 2 { hc(X,Y) : edge(X,Y) }, vtx(Y).
```

Hamiltonian Cycles (ctd): Undirected Cycles

- Instance (V,E) :

```
vtx(v).  
edge(v,u).      % one fact for each edge in E
```

- Generator:

```
2 { hc(V,U) : edge(V,U),  
    hc(W,V) : edge(W,V) } 2 :- vtx(V).
```

- Tester:

```
r(V) :- initialvtx(V).  
r(V) :- hc(V,U), edge(V,U), r(U).  
r(V) :- hv(U,V), edge(U,V), r(U).  
:- vtx(V), not r(V).
```

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

```
legal(P,mark(X,Y)) :- true(cell(X,Y,b)),  
                      true(control(P)).
```

```
legal(xplayer,noop) :- true(cell(X,Y,b)),  
                      true(control(ooplayer)).
```

```
legal(ooplayer,noop) :- true(cell(X,Y,b)),  
                      true(control(xplayer)).
```

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:

```
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:

```
coordinate(1..3).
```

```
symbol(x). symbol(o). symbol(b).
```

```
tdomain(cell(X,Y,C)) :- coordinate(X), coordinate(Y),  
                          symbol(C).
```

```
tdomain(control(X)) :- player(X).
```

```
{ true(T) : tdomain(T) }.
```

- Transition generator for the induction step:

```
ddomain(mark(X,Y)) :- coordinate(X), coordinate(Y).
```

```
ddomain(noop).
```

```
1 { does(P,M) : ddomain(M) } 1 :- player(P).
```


Verification (ctd)

- Tester(1): Every transition must be legal

```
:- does(P,M), not legal(P,M).
```

- Tester(2): Induction hypothesis

```
t0 :- 1 { true(control(X)) : player(X) } 1.  
:- not t0.
```

- Induction step

```
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:

```
t0(X,Y) :- 1 { init(cell(X,Y,Z)) : symbol(Z) } 1.  
t0 :- not t0(X,Y).  
:- not t0.
```

- This program has **no** answer set, which proves the claim.

Verification (ctd)

- Induction hypothesis

```
t0(X,Y) :- 1 { true(cell(X,Y,Z)) : symbol(Z) } 1.  
t0 :- not t0(X,Y).  
:- t0.
```

- Induction step to prove that cells have unique contents

```
t(X,Y) :- 1 { next(cell(X,Y,Z)) : symbol(Z) } 1.  
t :- not t(X,Y).  
:- not t.
```

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

```
p :- 1 { true(control(X)) : player(X) } 1.  
:- 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