AN INTRODUCTION TO ANSWER SET PROGRAMMING

Paul Vicol
October 14, 2015
(Based on slides by Torsten Schaub)
0. My Research
1. Declarative Problem Solving
2. ASP Syntax and Semantics
3. Modeling Problems in ASP
MY RESEARCH
We have a network of agents
Each agent has some initial beliefs about the state of the world
Agents communicate and share information
**Goal:** Determine what each agent believes after learning as much as possible from other agents
My research deals with ways to do this
DECLARATIVE PROBLEM SOLVING
Convert a problem specification into imperative code that solves instances of the problem
Deal with algorithms and data structures
The focus is on **how to solve the problem**
TRADITIONAL IMPERATIVE PROGRAMMING

- Convert a problem specification into imperative code that solves instances of the problem
- Deal with algorithms and data structures
- The focus is on **how to solve the problem**

Diagram:

```
Problem ----> Programming ----> Program ----> Running ----> Output
           |                          |
           |                          |
           |                          | Solution ----> Interpreting
```

7
- Directly encode the problem specification using a *modeling language*
- *How do we solve the problem?* vs *What is the problem?*
- Focus on **how to describe the problem**
• Write your problem in a formal representation (i.e. using logic)
• The representation defines an *implicit* search space, and gives a description of a solution
• An off-the-shelf *solver* takes the representation and finds its logical models
  • The problem representation should be such that these models represent solutions
WHAT IS ANSWER SET PROGRAMMING?

- ASP is a declarative problem-solving paradigm that combines an *expressive modeling language* with a *high-performance solver*
- It is geared towards solving NP-hard combinatorial search problems
- Originally developed for AI applications dealing with Knowledge Representation and Reasoning (KRR)
  - Led to a rich modeling language, compared to SAT
- Useful for solving combinatorial problems in $P$, $NP$, and $NP^{NP}$, in areas like
  - Bioinformatics
  - Robotics
  - Music Composition
  - Decision Support Systems used by NASA
  - Product Configuration
The best ASP tools are developed by the University of Potsdam, Germany

Download their ASP solver clingo from http://potassco.sourceforge.net/index.html
TWO PARADIGMS

- Theorem-Proving-Based Approach (Prolog)
  - Solution given by the derivation of a query
- Model-Generation-Based Approach (ASP)
  - Solution given by a *model* of the representation
Is Prolog Declarative?

- Not really... shuffling rules in a program can break it
- Prolog program:

```prolog
edge(1,2).
edge(2,3).

reachable(X,Y) :- edge(X,Y).
reachable(X,Y) :- edge(X,Z), reachable(Z,Y).
```

- A query:

```prolog
?- reachable(1,3).
true.
```
If we shuffle the program as follows:

edge(1, 2).
edge(2, 3).

reachable(X, Y) :- reachable(Z, Y), edge(X, Z).
reachable(X, Y) :- edge(X, Y).

Then we get:

?- reachable(1, 3).
Fatal Error: local stack overflow.

This is not a bug in Prolog; it is intrinsic in the fixed execution of its inference algorithm.

Prolog provides constructs to alter program execution

- The cut operator allows you to prune the search space, at the risk of losing solutions
Prolog is a *programming language*; it allows the user to exercise control

- For a programming language, control is good

ASP provides a *representation language*

- Completely decouples *problem specification from problem solving*

---

<table>
<thead>
<tr>
<th>Prolog</th>
<th>ASP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query Derivation</td>
<td>Model Generation</td>
</tr>
<tr>
<td>Top-down</td>
<td>Bottom-up</td>
</tr>
<tr>
<td>Fixed execution order</td>
<td>No fixed execution order</td>
</tr>
<tr>
<td>Programming Language</td>
<td>Modeling Language</td>
</tr>
</tbody>
</table>
ASP SYNTAX AND SEMANTICS
A logic program over a set of atoms $\mathcal{A}$ is a set of rules of the form:

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

where each $a_i \in \mathcal{A}$.

Rules are a way of expressing constraints on a set of atoms.

"$\sim$" represents default negation.

- An atom is assumed to be false until it is proven to be true.

Let $X$ be the set of atoms representing a solution.

This rule says: “If $a_1, \ldots, a_m$ are all in $X$, and none of $a_{m+1}, \ldots, a_n$ are in $X$, then $a_0$ should be in $X$.”
\[ \varphi = q \land (q \land \neg r \rightarrow p) \]

- \( \varphi \) has three classical models: \( \{p, q\} \), \( \{q, r\} \), and \( \{p, q, r\} \)
  - \( \{p, q\} \) represents the model where:
    \( p \mapsto 1, \ q \mapsto 1, \ r \mapsto 0 \)
- The logic program representation of \( \varphi \) is \( P_\varphi \):
  
  \[
  \begin{align*}
  q & \leftarrow \\
  p & \leftarrow q, \sim r
  \end{align*}
  \]
- This logic program has one stable model (a.k.a answer set): \( \{p, q\} \)
- A set \( X \) of atoms is a stable model of a logic program \( P \) if \( X \) is a (classical) model of \( P \) and all atoms in \( X \) are justified by some rule in \( P \)
MODELING PROBLEMS IN ASP
Model your problem as a logic program
- The ASP solver only deals with *propositional* logic programs
- But it’s more convenient (and much more flexible) to write first-order programs with variables
- Thus, the ASP solving process consists of two steps:
  1. A ** grounder ** converts a first-order program into a propositional program, by systematically replacing variables with concrete values from some domain
  2. A ** solver ** takes the *ground program* and assigns truth values to atoms to obtain the stable models of the program
ASP SOLVING STEPS

Problem → Modeling → Grounding → Solving → Output → Interpreting → Solution
- **Facts**
  - a.
  - person(bill).
  - person(alice;bob;james).
  - num(1..10).
    - Is shorthand for num(1). num(2). num(3). num(4). etc.

- **Rules**
  - a :- b.
  - reachable(X,Z) :- edge(X,Y), reachable(Y,Z).
If we have a logic program

\[
\begin{align*}
  r(a, b). \\
  r(b, c). \\
  t(X, Y) & :- r(X, Y).
\end{align*}
\]

Then the full _ground instantiation_ is:

\[
\begin{align*}
  r(a, b). \\
  r(b, c). \\
  t(a, a) & :- r(a, a). \\
  t(b, a) & :- r(b, a). \\
  t(c, a) & :- r(c, a). \\
  \ldots
\end{align*}
\]

Which is trivially reduced to:

\[
\begin{align*}
  r(a, b). \\
  r(b, c). \\
  t(a, b) & :- r(a, b). \\
  t(b, c) & :- r(b, c).
\end{align*}
\]
General methodology: *generate and test* (or “guess and check”)

1. **Generate** candidate solutions through non-deterministic constructs (like choice rules)
2. **Test** them to eliminate invalid candidate solutions
ASP MODELING CONSTRUCTS

- **Choice Rules**
  - 1 { has_property(X,C) : property(C) } 1 :- item(X).

- **Integrity Constraints**
  - :- in_clique(2), in_clique(3), not edge(2,3).
  - “It cannot be the case that nodes 2 and 3 are in a clique, and there is no edge between 2 and 3.”

- **Aggregates**
  - within_budget :- 10 #sum { Amount : paid(Amount) } 100.

- **Optimization Statements**
  - #maximize { 1,X:in_clique(X),node(X) }.
**N-QUEENS PROBLEM**

- **Goal:** Place $n$ queens on an $n \times n$ chess board such that no queens attack each other
Define the board:

\[
\begin{align*}
\text{row}(1..n). \\
\text{col}(1..n).
\end{align*}
\]

\[
\begin{align*}
\text{Answer: 1} \\
\text{row(1) row(2) row(3) row(4) \} \} \\
\text{col(1) col(2) col(3) col(4) \} \} \\
\text{SATISFIABLE}
\end{align*}
\]

$ $ clingo queens.lp --const n=4$ $
**Generate**: Place any number of queens on the board:

\[
\{ \text{queen}(I,J) : \text{row}(I), \text{col}(J) \}.
\]

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

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

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

SATISFIABLE
N-QUEENS - PLACING QUEENS

1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
We need to say that there should only be \( n \) queens

Expressed by an integrity constraint using *double negation*

- “It should not be the case that there are not \( n \) queens.”

\[
:- \text{not } n \{ \text{queen}(I,J) \} \text{ n.}
\]

$ clingo queens.lp --const n=4$

Solving...

Answer: 1

queen(1,1) queen(2,1) queen(3,1) queen(4,1)
N-QUEENS - RESTRICTING THE NUMBER OF QUEENS

- The last solution looks like this:

```
 4
 3
 2
 1
```

```
1 2 3 4
```
- Prevent attacks by adding *integrity constraints*
- Forbid horizontal attacks (two queens in the same row):
  \[
  \text{:-}\ \text{queen}(I, J_1), \text{queen}(I, J_2), J_1 \neq J_2.
  \]
- Forbid vertical attacks (two queens in the same column):
  \[
  \text{:-}\ \text{queen}(I_1, J), \text{queen}(I_2, J), I_1 \neq I_2.
  \]
- And forbid diagonal attacks:
  \[
  \text{:-}\ \text{queen}(I, J), \text{queen}(II, JJ), (I, J) \neq (II, JJ), I+J = II+JJ.
  \]
  \[
  \text{:-}\ \text{queen}(I, J), \text{queen}(II, JJ), (I, J) \neq (II, JJ), I-J = II-JJ.
  \]
queens.lp

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

% Generate
n { queen(I,J) : row(I), col(J) } n.

% Test
:- queen(I,J1), queen(I,J2), J1 != J2.
:- queen(I1,J), queen(I2,J), I1 != I2.
:- queen(I,J), queen(II,JJ), (I,J) != (II,JJ), I+J == II+JJ.
:- queen(I,J), queen(II,JJ), (I,J) != (II,JJ), I-J == II-JJ.

#show queen/2.
$ clingo queens.lp --const n=4
Solving...
Answer: 1
queen(3,1) queen(1,2) queen(4,3) queen(2,4)
• ASP is very tolerant to elaboration in the problem specification
• Start with a large search space, and keep adding constraints to whittle down results
• Helps you to understand your problem, and is useful for prototyping
**Problem instance:** A graph $G = \langle V, E \rangle$.

**Goal:** Assign one colour to each node, such that no two nodes connected by an edge have the same colour.
Represent the graph using node/1 and edge/2 predicates:

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

col(red;green;blue).
```
• **Choice Rules**
  
  1 { has_property(X,C) : property(C) } 1 :- item(X).

• **Integrity Constraints**
  
  :- in_clique(2), in_clique(3), not edge(2,3).
  
  “It *cannot* be the case that nodes 2 and 3 are in a clique, *and* there is no edge between 2 and 3.”

• **Aggregates**
  
  within_budget :- 10 #sum { Amount : paid(Amount) } 100.

• **Optimization Statements**
  
  #maximize { 1,X:in_clique(X),node(X) }.


- **Generate:** Assign one colour to each node using a *choice rule*

\[
\text{1 \{ node\_col}(X,C) : \text{col}(C) \}\text{ 1 :- node}(X).
\]

- **Test:** Eliminate candidate solutions where two nodes connected by an edge get the same colour, using an *integrity constraint*

\[
\text{:- edge}(X,Y), \text{node\_col}(X,C), \text{node\_col}(Y,C).
\]
col.lp

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

col(red;green;blue).
1 { node_col(X,C) : col(C) } 1 :- node(X).
:- edge(X,Y), node_col(X,C), node_col(Y,C).

#show node_col/2.
$ clingo col.lp
Answer: 1
node_col(2,green) node_col(1,blue) node_col(3,red) \nnode_col(5,green) node_col(4,blue) node_col(6,red)
SATISFIABLE

Models : 1+
Calls : 1
Time : 0.003s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00s)
CPU Time : 0.000s
GRAPH 3-COLOURING - INTERPRETING
We also can enumerate all possible solutions:

```
$ clingo col_encoding.lp col_instance.lp 0
Answer: 1
node_col(2,green) node_col(1,blue) node_col(3,red) \nnode_col(5,green) node_col(4,blue) node_col(6,red)
Answer: 2
node_col(2,green) node_col(1,blue) node_col(3,red) \nnode_col(5,green) node_col(4,blue) node_col(6,blue)
Answer: 3
node_col(1,red) node_col(2,green) node_col(3,blue) \nnode_col(5,green) node_col(4,red) node_col(6,red)
Answer: 4
node_col(1,red) node_col(2,green) node_col(3,blue) \nnode_col(5,green) node_col(4,red) node_col(6,blue)
Answer: 5
node_col(1,red) node_col(2,blue) node_col(3,green) \nnode_col(5,blue) node_col(4,red) node_col(6,red)
```

SATISFIABLE
MY HOBBY:
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS

CHOTCHKIES RESTAURANT

APPETIZERS
MIXED FRUIT 2.15
FRENCH FRIES 2.75
SIDE SALAD 3.35
HOT WINGS 3.55
MOZZARELLA STICKS 4.20
SAMPLER PLATE 5.80

SANDWICHES
BARBECUE 6.55

WE’D LIKE EXACTLY $15.05 WORTH OF APPETIZERS, PLEASE.

...EXACTLY? UHH...

HERE, THESE PAPERS ON THE KNAPSACK PROBLEM MIGHT HELP YOU OUT.

LISTEN, I HAVE SIX OTHER TABLES TO GET TO...

--AS FAST AS POSSIBLE, OF COURSE. WANT SOMETHING ON TRAVELING SALESMAN?
Order some amount (possibly 0) of each food
Such that the sum of the costs of the foods times the number ordered is *exactly* some desired amount
We can encode the foods and costs as follows:

food(fruit;fries;salad;wings;mozz_sticks;sampler).

cost(fruit,215).
cost(fries,275).
cost(salad,335).
cost(wings,355).
cost(mozz_sticks,420).
cost(sampler,580).
#const total = 1505.
#const max_order = 10.

food(fruit; fries; salad; wings; mozz_sticks; sampler).

cost(fruit, 215). cost(fries, 275). cost(salad, 335).

% Have to set an upper bound on the orders for a specific food
num(0..max_order).

% Order some amount (possibly 0) of each type of food
1 { order(Food, Number) : num(Number) } 1 :- food(Food).

% We want the prices to sum to the desired total
#sum{(Cost*N), F : order(F,N) : cost(F, Cost), num(N)} == total.

47
clingo --const total=1505 xkcd.lp
order(fruit,7) order(fries,0) order(salad,0)
order(wings,0) order(mozz_sticks,0) order(sampler,0)
clingo --const total=19000 xkcd.lp 0
Is ASP Declarative?

- In many ways, yes:
  - You provide a specification of the problem, and a problem instance, and you get a result
  - The order of rules doesn't matter
  - You don’t have to think about how your problem is solved (algorithm, data structures), just what your problem is

- However…
  - Different problem encodings can yield different solving times
  - Efficiency still depends on how you specify your problem
• Performance generally depends on the size of the ground instantiation
  • This is what the solver has to look at
• *Intelligent grounding* techniques attempt to automatically reduce the size of the ground program by eliminating unnecessary rules
• But still, you can never recover from a bad encoding
The previous encoding of n-Queens becomes slow at \( n \approx 15 \). The encoding below is much better (gets to \( n \approx 250 \) in the same amount of time):

\[
1 \{ \text{queen}(I, 1..n) \} 1 :- I = 1..n. \\
1 \{ \text{queen}(1..n, J) \} 1 :- J = 1..n. \\
:- 2 \{ \text{queen}(D-J, J) \}, D = 2..2*n. \\
:- 2 \{ \text{queen}(D+J, J) \}, D = 1-n..n-1.
\]

A version of this encoding was used to go to \( n = 5000 \):

- Solving took about 1 hour.