Lecture Notes for Cmput 325: Introduction to Answer Set Programming

Jia-Huai You

University of Alberta

March 31, 2021

Introduction

- Answer set programming (ASP) is a recent programming paradigm for declarative problem solving, constraint solving, and knowledge representation and reasoning (KR).
- Has roots in KR and non-monotonic reasoning.
- Related to propositional satisfiability (SAT) and constraint programming (Constraint Satisfaction Problem (CSP) and Constraint Logic Programming (CLP)).
- Many SAT and ASP solvers built; Solver competitions (see What's hot in the SAT and ASP Competitions)
- Plenty of existing applications.

What is ASP for and not for?

- Not For:
  - ASP is not a general purpose programming language like C, Java, Python, or even Lisp or Prolog (though its syntax is related to that of Prolog)
  - It is not for computing recursive functions as what a Turing Machine does.
- For:
  - Solving computationally hard problems like CLPFD or Boolean Satisfiability.
  - These are NP-complete problems or problems that are even harder.

ASP Paradigm

General Idea: Stable models/answer sets are solutions.

Problem instance Q
$\Rightarrow$ Encoding Q in program P
$\Rightarrow$ Compute stable models of P by an ASP solver
$\Rightarrow$ Extract solutions from stable models

- Problem Q is encoded as a (non-monotonic) logic program P, such that solutions to Q correspond to stable models of P.
- Compute stable models of P using an ASP solver.
- A solution is in a stable model.

A correct encoding ensures the one-to-one correspondence between solutions to the given problem and stable models of its ASP encoding. Thus, if a given problem has no solution, then the corresponding program should have no stable models.

- The most basic form of programs is normal programs. A normal program is a finite set of rules of the form

$$A \leftarrow B_1, ...B_k, not\ C1, ..., not\ C_n$$

  where A, B_i and C_i are atoms in the underlying propositional language. $not\ C_i$ is called a default negation. A_i and $not\ C_i$ are both called literals.
- By convention, the left hand side of a rule is called the head of the rule and the right hand side the body.
- A rule is read intuitively as: If B_i are in a solution and none of the C_i are in the same solution, then A must be in the same solution.

Examples:

% A bird flies if it is not an abnormal bird. Which bird flies?

fly(X) $\leftarrow$ bird(X), not abnormal(X).
bird(X) $\leftarrow$ penguin(X).
abnormal(X) $\leftarrow$ penguin(X).
abnormal(X) $\leftarrow$ bird(X), hungry(X).
bird(b).
penguin(a).

Ground vs. non-ground programs

We can write programs with variables as a compact way to express all instances of non-ground rules.

Suppose we have constants a,b,c in our program, the rule

p(X) $\leftarrow$ q(X), not s(X).

is a compact way to express all instances of the rule wrt the three constants

p(a) $\leftarrow$ q(a), not s(a).
p(b) $\leftarrow$ q(b), not s(b).
p(c) $\leftarrow$ q(c), not s(c).

From now on, we will use identifiers starting with a capital letter as variables.

Ground vs Non-ground

The process of translating a non-ground program to a ground one is called grounding.

If in a non-ground program we only have finite number of constants and no function symbols, then the grounding of the program generates a finite ground program.

Example.

p(X) $\leftarrow$ q(X), not s(Y).	function-free
p(X) $\rightarrow$ q(f(X)), not s(Y).	f is a function symbol

In the case of being function-free, the grounded program may be of exponentially larger than the non-ground one.

Example

bachelor(X) $\leftarrow$ person(X),person(Y), not married(X,Y).
person(john).
person(lily).
person(ken).
married(john,lily).
married(X,Y) $\leftarrow$ married(Y,X).

Intuitively, who is a bachelor?

What is an answer set?

An answer set M (or called stable model) of a program P is a model of P that satisfies the following properties.

- M satisfies all rules in P.
- For any rule $a \leftarrow b_1, ...b_k, not\ c_1, ..., not\ c_n$, if for all b_i, $b_i \in M$ and none of c_i is in M, then $a \in M$.
- For any $h \in M$, there exists a rule $h \leftarrow b_1, ...b_k, not\ c_1, ..., not\ c_n$ in P such that $\forall b_i, b_i \in M$ and $\forall c_i, c_i \notin M$.

  In additional:
  In the last property above, h in M is said to be supported by the rule. Any atom in an answer set must supported by a non-circular derivation. We call such an atom justified.

Examples:

- Let P_1 be:

  a $\leftarrow$ a.

  $M = \{a\}$ is a model of P_1, but it is not an answer set since a is not justified by a non-circular derivation.

- Let P_2 be:

  a $\leftarrow$ not b.
  b $\leftarrow$ not a.

  $M_1 = \{a\}$ is an answer set, so is $M_2 = \{b\}$.

- The head of a rule can be empty (false) - a constraint. E.g., In a graph, suppose we want to represent: if any nodes A and B that share an edge cannot be colored with the same color

  $\leftarrow$ node(A), node(B), color(C), edge(A,B), colored(A,C), colored(B,C).

We can write a program to enumerate all subsets, say of {a, b, c}.

```
a ← not a'.
a' ← not a.
b ← not b'.
b' ← not b.
c ← not c'.
c' ← not c.
```

where we use x' to represent "x is false". So, e.g., {a, b', c} is an answer set representing subset {a,c}.

We can further express: any subset where *c* is not in it by adding a constraint

```
 ← c.
```

Does tweety fly?

fly(X) $\leftarrow$ bird(X), not ab(X).
ab(X) $\leftarrow$ penguin(X).
bird(X) $\leftarrow$ penguin(X).
bird(tweety).

There is precisely one answer set of the program which contains fly(tweety). What if we now know tweety is a penguin and add this knowledge to our program:

penguin(tweety).

Will tweety still fly?

Example

person(joe). person(ann). married(ann).
male(X) $\leftarrow$ not female(X).
female(X) $\leftarrow$ not male(X).
bachelor(X) $\leftarrow$ male(X), not married(X).

What are the answer sets?

Since we never said whether joe is a male or female (the same applies to ann), all four (exclusive) cases are possible: both male, both female, and of the opposite sex. All these can be justifed by our rules. Note that our rules do not support that someone is both male and female. So, there are 4 answer sets. Here are two of them (predicate names are abbreviated)

{p(joe),p(ann),m(ann), male(joe), f(ann), b(joe)}
{p(joe),p(ann),m(ann), f(joe), male(ann)}

What if we add the fact $\mathrm{male(joe)}$ to our program?

Answer sets may not always exist

Just like a problem may not have a solution, a program may not have an answer set. E.g. the single-rule program

f $\leftarrow$ not f.

does not have any answer set. $\emptyset$ cannot be an answer set since it does not satisfy the rule. $\{f\}$ cannot be an answer set, since under it f is not supported by any rule. Intuitively, in this case f cannot be settled down with a truth value. It cannot be true neither false.

As another example, the following program has no answer sets

f $\leftarrow$ not f, a, not b.
a $\leftarrow$ not b.

Why?

Example: Barber's Paradox

A barber named John put up a poster, which says: I will shave all those who don't shave themselves.

Consider a normal program:

person(joe).   person(ken).   person(john).
shave(john,Y) $\leftarrow$ person(Y), not shave(Y,Y).

Does this program have an answer set? No. Our barber forgot to make it precise that "those" do not include himself. An instance of the last rule is:

shave(john,john) $\leftarrow$ person(john), not shave(john,john).

Our barber should have said:

shave(john,Y) $\leftarrow$ person(Y), Y $\neq$ john, not shave(john,Y).

Generate and Constrain

Or, you don't want to use unintuitive primed symbols, then you can represent a subset by a predicate *in*(m) meaning m is in such a subset, and *out*(m) that m is not in.

element(a). element(b). element(c).
in(X) $\leftarrow$ element(X), not out(X).
out(X) $\leftarrow$ element(X), not in(X).

E.g., {in(a),in(b),out(c)} is an answer set, representing the subset {a,b}.

How do you specify the subsets of {a,b,c} such that whenever a is in it, so is b?

3-colorability

Whether 3 colors, say red, blue, and yellow, are sufficient to color a map, where a map is represented by a graph, with facts about nodes, arcs and available colors as given, e.g,

```
vertex(a).
vertex(b).
arc(a,b).
......
col(r).
col(b).
col(y).
```

3-colorability

Every vertex must be colored with exactly one color:

color(V,r) $\leftarrow$ vertex(V), not color(V,b), not color(V,y).
color(V,b) $\leftarrow$ vertex(V), not color(V,r), not color(V,y).
color(V,y) $\leftarrow$ vertex(V), not color(V,b), not color(V,r).

No adjacent vertices may be colored with the same color:

$\leftarrow$ vertex(V), vertex(U), arc(V,U), V $\neq$ U, col(C),
color(V,C),color(U,C).

N-colorability

A vertex is either colored with a color, or not, using auxiliary predicate ncolor/2

color(V,C) $\leftarrow$ vertex(V), col(C), not ncolor(V,C).
ncolor(V,C) $\leftarrow$ vertex(V), col(C), not color(V,C).

No vertex may be colored with more than one color

$\leftarrow$ vertex(V), col(C1), col(C2), color(V,C1), color(V,C2), C1 $\neq$ C2.

Any node must be colored with at least one color

$\leftarrow$ vertex(V), col(C), not color(V,C).

No adjacent vertices may be colored with the same color:

$\leftarrow$ vertex(V), vertex(U), arc(V,U), V $\neq$ U, col(C),
color(V,C),color(U,C).

Program

Executable code:

```
color(V,C) :- vertex(V), col(C), not ncolor(V,C).
ncolor(V,C) :- vertex(V), col(C), not color(V,C).

:- vertex(V), col(C1), col(C2), color(V,C1),
   color(V,C2), C1 != C2.
:- vertex(V), col(C), not color(V,C).
:- vertex(V), vertex(U), arc(V,U), V != U, col(C),
     color(V,C),color(U,C).
```

Ooops

The previous program has a bug - if you run it with a graph, it says no answer set. By

```
:- vertex(V), col(C), not color(V,C).
```

We want to say each vertex should be colored with at least one color, which should have been said as:

```
:- not coloring(V).
coloring(V) :- vertex(V), col(C), color(V,C).
```

What is the difference between the two?

ASP solvers

Research products that are reasonably maintained.

- Smodels (formally Helsinki Univ. of Tech., now Aalto University); Lparse is the grounder and has become the standard for syntax of ASP language (latesst syntax is called ASP-Core-2-Encoding).
- DLV (Vienna Univ. of Tech.)
- Cmodel (U. of Texas at Austin)
- Clasp (Universitat Potsdam)

There are a number of other solvers around.

ASP is different from Prolog, though similar in syntax

- The common standard of syntax is called Lparse, the first implemented program that converts a function-free program to a ground program before calling an ASP solver.
- Syntax follows that of Prolog. But ASP represents a very different programming paradigm.
- In Prolog, one writes a program and query whether a goal follows from a program; but in ASP, we compute answer sets as solutions.
- Even for (pure) function-free Horn clause programs, ASP is different from Prolog. Prolog is not a faithful implementation, while ASP is. Prolog may not terminate while ASP does.

Language Constructs are added

Enumerating combinations by a normal program is inconvenient. To address this problem, The language is extended.

Cardinality constraint

l {a1,...,an} u

where l and u are non-negative integers. It specifies all subsets of {a1,...,an} whose size is between l and u, inclusive. When l and u are omitted, then it says the same as

0 {a1,...,an} n

which specifies any subset of {a1,...,an} (i.e., any subset satisfies the constraint) and it is called a choice constraint.

Example

E.g. All subsets of {a,b,c,d,e} whose size is between 2 and 3 such that a and b cannot be together.

2 {a,b,c,d,e} 3 $\leftarrow$.
$\leftarrow$ a, b.

Answer sets are: {a,c}, {a,c,d}, ... But {a,b} is not an answer set, neither {a,c,d,e}, among others.

Note: In latest versions of solvers, "," is replaced by ";". So, write
2 {a;b;c;d;e} 3 $\leftarrow$.
$\leftarrow$ a, b.

Conditional Literals in Lparse Syntax

A number of language constructs are provided, mostly for convenience. One is conditional literal, a shorthand to express a set. It takes the form

l : d

where l is an atom and d a domain predicate.

E.g. Color a vertex v with exactly one color among red, blue and yellow:

```
1 {setColor(v,C): color(C)}1.
color(red).
color(blue).
color(yellow).
```

The first rule above is a shorthand for

```
1{setColor(v,red); setColor(v,blue); setColor(v,yellow)} 1.
```

N-colorability

Every vertex is colored with exactly one color:

```
  1 {setColor(V,C) : col(C) } 1 :- vertex(V).
```

Facts representing colors (a short hand)

```
  col(1..colors).
```

No adjacent vertices are colored with the same color:

```
  :- vertex(V), vertex(U), arc(V,U), col(C ), U != V,
     setColor(V,C), setColor(U,C).
```

At the command line input, a number should be provided to colors. E.g. given a graph in graph.lp and the above code in color.lp, we can run

```
clingo graph.lp color.lp -c colors=3
```

N-queens problem

```
#show q/2.

d(1..queens).
1 {q(X,Y):d(Y)} 1 :-d(X).
:- d(X), d(Y), d(X1), q(X,Y), q(X1,Y), X1 != X.
:- d(X), d(Y), d(Y1), q(X,Y), q(X,Y1), Y1 != Y.
       % the above is not needed logically
:- d(X), d(Y), d(X1), d(Y1), q(X,Y), q(X1,Y1),
   X != X1, Y != Y1, abs(X -X1) == abs(Y -Y1).
:-d(Y), not hasq(Y).
hasq(Y) :-d(X), d(Y), q(X,Y).
```

Assume the code is in the file nqueens,lp. Run it by
clingo -c queens=4 nqueens.lp