Notes on Answer Set Programming

CSC 791 Generative Methods for Game Design

Chris Martens

September 20, 2017

1 Introduction

Answer set programming, or ASP, is a technique based on logic programming that allows us to define a generative space in abstract terms, then iteratively place constraints on that generative space to whittle down the output to what we want.

For example, if we consider the problem of dungeon generation, first we specify all the possible ways dungeons could be constructed. Then, we write a logical predicate that defines what it means for the dungeon to be solvable. Then we can run an *answer set solver* to generate a set of dungeons that are solvable.

The programming language that both the constructive rules and constraints are written in is called AnsProlog, and today we are going to dip our toes into AnsProlog.

2 Setup

You will need the Clingo/Potassco answer set solver available here: `http://potassco.sourceforge.net/`

3 Running Clingo

Edit your code in a file `yourfile.lp`, then run:

```
$ clingo -n 0 yourfile.lp
```

The `-n` flag gives Clingo a number of answer sets to generate; `0` tells it to generate all answer sets.

To get one example of *random* output, use:

```
$ clingo -n 1 --rand-freq=1 yourfile.lp
```

But this won't actually generate different output on each run unless you also change the random seed:

```
$ clingo -n 1 --rand-freq=1 --seed=<SEED> yourfile.lp
```

For example, using `‘echo $RANDOM‘` as your seed will generate new output every time you run the command.

4 Facts and Rules

AnsProlog is a style of Prolog, which means we define our program through relations, or logical propositions. Simply by writing

```
p.
q.
```

We declare that `p` and `q` are true facts in the world.

What's more interesting is that we can define how `p` and `q` are logically related to one another. For instance, we can say that `p` *implies* `q` with the syntax:

```
q :- p.
```

This is a *rule*, or a *clause*, where `q` is the head and `p` is the body. The symbol `:-` is sometimes referred to as the "neck" of the rule.

5 Answer Sets

Suppose our complete program is:

```
p.
q :- p.
```

Unlike in Prolog, we don't have to issue a query in order for computation to happen. The answer set solver instead produces *all possible collections* of facts that are consistent with what we've said in the program.

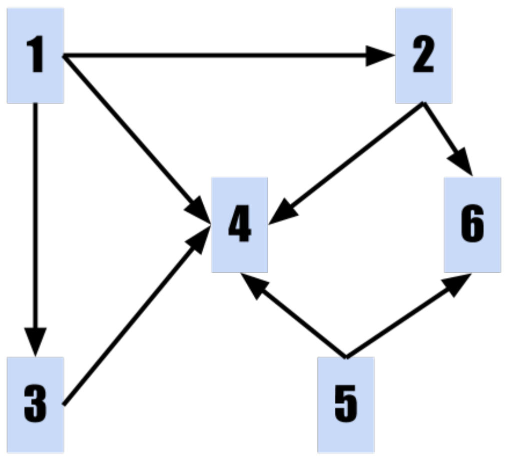
So when we run Clingo on this program, we get a single answer set:

```
p q
```

This set represents all consequences that are derivable from the program. We can use this to do neat recursive searching in a very concise way; for example, we can find all paths between two nodes in a graph.

6 Example: Edges and Paths

Let's say we have a directed graph as follows:



We can represent this graph using a collection of *relations*, or facts with arguments. The first relation we'll need is a unary (one-place) relation, or *predicate*, declaring that something is a node:

```
node(1..6).
```

This notation is syntactic sugar for the equivalent program

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

The 2-place relation `edge(X,Y)` will represent a directed edge from `X` to `Y`. For example, all the edges coming out of node 1 could be written:

```
edge(1, 2).
edge(1, 3).
edge(1, 4).
```

AnsProlog actually has syntactic sugar for writing a collection of facts using the same relation more concisely, using semicolon `;`. We separate out the edge definitions by source node below:

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

Now we can write a rule relating edges to form paths. Actually, this rule will be a *rule schema*, ranging over variables for the nodes in question.

First, we say that paths are *reflexive*: every node has a path to itself.

```
path(X, X) :- node(X).
```

In this rule, `X` is a *logic variable*, which means it stands for *any* value. In logical notation, we can read this as a *quantified* variable, i.e.:

$$\forall x.\mathsf{node}(x) \Rightarrow \mathsf{path}(x, x)$$

By convention, in logic programming, logic variables start with a capital letter to distinguish them from concrete terms.

Run Clingo now to see the output of this partial definition. You should see a `path` fact generated from every node to itself.

Adding the line

```
#show path/2.
```

restricts the answer set just to the path predicate (`/2` is syntax for the number of arguments in the relation, which we have to include).

To generate a `path` fact for all *transitive* traversal of the edges, we write a recursive rule:

```
path(Source, Dest)
:-  edge(Source, Target),
        path(Target, Dest).
```

Running this program should produce the following answer set:

```
path(1,1) path(2,2) path(3,3) path(4,4) path(5,5) path(6,6)
path(1,2) path(1,3) path(1,4) path(2,4) path(2,6) path(3,4)
path(5,4) path(5,6) path(1,6)
```

6.1 Exercise

Modify the path relation to be a 3-place relation that counts the length of the path. You can add numbers together with `+`.

7 Choice Rules

A choice rule allows us to use disjunction in the head of a rule, i.e. to say that either *A or B* may hold. Choice rules allow us to define a generative space and are what lead to the possibility of multiple answer sets.

Choice rules are written with curly braces. For example, the choice rule

```
{p}.
```

declares simply that `p` may be true or not. This produces two answer sets:

```
Answer: 1

Answer: 2
p
```

If we declare *two* choices, the number of answer sets increases combinatorially:

```
{p}. {q}.
```

produces

```
Answer: 1

Answer: 2
q
Answer: 3
p
Answer: 4
p q
```

Just like we could define multiple facts with syntactic sugar, we can do so inside of choice rules: `{p; q}` is equivalent to the program above.

Choice rules let us do fun things like make sense of recursive definitions. For example, the program

```
p :- q.
q :- p.
```

has a circular definition—`p` and `q`'s truth depends mutually on the other's. But if we give ASP a choice for the truth of these facts with `{p; q}`, it generates two answer sets:

```
Answer: 1

Answer: 2
q p
```

Both of these answers are *possible worlds* that satisfy the truth model posited by the program's rules.

7.1 Graph Coloring

The fun starts when we begin using logic variables inside choice rules. Let's say we want to assign a random color to every node in our graph. First let's define some colors.

```
color(red). color(green). color(blue).
```

It may seem at first glance that we could write a choice rule

```
{colornode(1, C)}.
```

to simply choose any color for the node 1. But recall that ASP *expands* rules to their possible instantiations. In order to do that, it has to know all valid instantiations of `C`. So we actually need to tell it that `C` ranges over colors, which we do with a new piece of syntax, the colon `:`:

```
{colornode(1, C) : color(C)}.
```

This syntax says that *for each instance* of the predicate `color(C)`, possibly assign that color to the node 1. It can be seen as expanding to:

```
{colornode(1, red)}.
{colornode(1, green)}.
{colornode(1, blue)}.
```

However, this program means that the answer sets produced include multiple color assignments for the node, as well as no color assignment:

```
Answer: 1

Answer: 2
colornode(1,blue)
Answer: 3
colornode(1,green)
Answer: 4
colornode(1,blue) colornode(1,green)
Answer: 5
colornode(1,red)
Answer: 6
colornode(1,red) colornode(1,green)
Answer: 7
colornode(1,red) colornode(1,blue)
Answer: 8
colornode(1,red) colornode(1,blue) colornode(1,green)
```

To prevent this under- and over-generation of assignments, we can use AnsProlog's upper and lower bound notation:

```
l { ...some predicates... } u
```

where `l` is a lower bound on the number of facts generated from the choice and `u` is an upper bound. So if we write

```
1 {colornode(1, C) : color(C)} 1.
```

it will generate exactly one color assignment for node 1. To do this for all nodes, we want to replicate the choice for each instantiation of the `node` predicate. We do this using a rule schema:

```
1 {colornode(X, C) : color(C)} 1 :- node(X).
```

This code produces 729 answers representing all possible colorings of the graph.

8 Integrity Constraints

Finally, in addition to setting mutual exclusion constraints through bounds on choice rules as above, we can introduce constraints that say "this condition is illegal." The answer sets produced will be those that do not have the illegal condition.

We do this in AnsProlog by writing a *headless rule*:

```
:- condition.
```

In logical terms, this says "condition implies FALSE," and FALSE inherently creates an inconsistent world. Since ASP generates just those worlds that are consistent, this means that any worlds containing the condition will be ruled out.

8.1 Exercise

See if you can create a condition ensuring that no two nodes with an edge between them has the same color. If you finish that, try *generating* 3-colorable graphs.

9 Resources

The definitive manual for the Potassco ASP toolkit (Clingo, etc.) is `http://wp.doc.ic.ac.uk/arusso/wp-content/uploads/sites/47/2015/01/clingo_guide.pdf`.

Adam Smith has a "map generation speed-run": `https://eis-blog.soe.ucsc.edu/2011/10/map-generation-speedrun/`