July, 2017

Answer Set Programming Paradigm

Yuliya Lierler

Handout on Answer Set Programming Paradigm

Yuliya Lierler
University of Nebraska Omaha

Introduction

Answer set programming paradigm (ASP) is a form of declarative programming oriented towards difficult combinatorial search problems. It belongs to the group of so called constraint programming languages. The idea of ASP is to represent a given computational problem by a program whose answer sets correspond to solutions, and then use an answer set solver to generate answer sets for this program. In this note we discuss the methodology of answer set programming as well as the use of software systems for computing answer sets. First, a graph coloring problem is utilized to illustrate the use of answer set programming in practice. Then, solutions to Hamiltonian cycle and to n-queens problems are presented. Across the handout you are given problems to solve. This handout is self-contained: you are given all the definitions and links that are required in constructing solutions.

In the text italics is primarily used to identify concepts that are being defined. Some definitions are identified by the word **Definition**.

In this course we will use the answer set system CLINGO[1] that incorporates answer set solver CLASP[1] with its front-end grounder GRINGO[1] (user guide is available online at `https://sourceforge.net/projects/potassco/files/guide/2.0/guide-2.0.pdf/download`). You may access system CLINGO via web interface available at `https://potassco.org/clingo/run/` or download an executable for CLINGO version 5 from the url listed at footnote 1.

Answer set programming practitioners develop applications that rely on ASP languages, which allow variables. Yet, common ASP solvers, including CLASP (a subsystem of CLINGO) process propositional logic programs only. We now re-introduce such programs and restate the definition of an answer set in the form convenient for this part of the course.

[1] `https://potassco.org/clingo/`.

1 Formal preliminaries: Propositional logic programs

A *propositional logic program* is a finite set of *rules* of the form

$$a_0 \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n, not\ not\ a_{n+1}, \ldots, not\ not\ a_p, \qquad (1)$$

where a_0 is a propositional atom or symbol $\bot$; $a_1, \ldots, a_p$ are propositional atoms. Note that rule (1) contains "doubly negated" atoms — a construct that you have not seen before. This construct is used to formalize so called choice rules that are frequently used by the ASP practitioners. Here we present the definition of an answer set for programs composed of rules (1) and then discuss in which sense they capture choice rules.

A rule (1) is *normal* when $n = p$. A program is *normal* when it is composed of normal rules. The left hand side expression of rule (1) is called the *head*. The right hand side is called the *body*. Expression

$$a_1, \ldots, a_m$$

constitutes *positive part* of the body. We call rule (1)

- a *fact* when its body is empty (we then drop $\leftarrow$ from a rule);
- a *constraint* when its head is symbol $\bot$ (we then drop $\bot$ from a rule).

We call a program *definite* when it is composed of rules of the form

$$a_0 \leftarrow a_1, \ldots, a_m. \qquad (2)$$

For instance,

$$\begin{array}{l} p \\ r \leftarrow p, q \end{array} \qquad (3)$$

and

$$\begin{array}{l} p \leftarrow not\ \ q \\ q \leftarrow not\ \ r \end{array} \qquad (4)$$

are both normal programs, where program (3) is also definite.

A set X of atoms *satisfies* a definite rule

$$a_0 \leftarrow a_1, \ldots a_m \qquad (5)$$

when $a_0 \in X$ whenever $\{a_1, \ldots, a_m\} \subseteq X$. We say that a set X of atoms *satisfies* definite program Π if it satisfies every rule of Π. For example, sets $\{p\}$, $\{p, r\}$, and $\{p, q, r\}$ satisfy definite program (3).

Definition 1. *The* reduct Π^X *of a logic program* Π *relative to a set* X *of atoms is the set of rules* (5) *for all rules (1) in* Π *such that* $a_{m+1}, \ldots, a_n \notin X$ *and* $\{a_{n+1}, \ldots, a_p\} \subseteq X$. *A set* X *of atoms is an* answer set *of the program* Π *if* X *is minimal among the sets of atoms that satisfy* Π^X.

For instance, let Π_1 denote program (4) and X_1 denote set $\{q\}$. The reduct $\Pi_1^{X_1}$ consists of a single fact q. Set $\{q\}$ is the minimal set satisfying $\Pi_1^{X_1}$. Consequently, X_1 is an answer set of Π_1. Let X_2 denote set $\{p\}$. The reduct $\Pi_1^{X_2}$ consists of two facts p and q. Set $\{p, q\}$ is the minimal set satisfying $\Pi_1^{X_2}$. Thus, $\{p\}$ is not an answer set of Π_1.

Answer sets semantics by default follows closed world assumption (CWA) – presumption that what is not currently known to be *true* is *false*. Let a program consist of the rule

$$p \leftarrow not\ not\ p. \tag{6}$$

This program has two answer sets $\emptyset$ and $\{p\}$. This rule can be used to "eliminate" CWA for an atom p. Intuitively, it states that an atom p *may* be a part of an answer set. This extension of logic programs is essential. *Choice rules* of the form

$$\{a_0\} \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n, \tag{7}$$

are important constructs of common ASP dialects. A choice rule (7) can be seen as an abbreviation for a rule with doubly negated atoms of the form

$$a_0 \leftarrow\ a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n, not\ not\ a_0.$$

Here we adapt this abbreviation. The simplest choice rule

$$\{p\}$$

corresponds to rule (6).

Consider a constraint $\leftarrow p$. Extending program (3) by this rule will result in a program that has no answer sets. In other words, constraint $\leftarrow p$ eliminates the only answer of (3). It is convenient to view any constraint

$$\leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n, not\ not\ a_{n+1}, \ldots, not\ not\ a_p, \tag{8}$$

as a clause (a disjunction of literals)

$$\neg a_1 \vee \cdots \vee \neg a_m \vee a_{m+1} \vee \cdots \vee a_n \vee \neg a_{n+1} \vee \cdots \vee \neg a_p. \tag{9}$$

Then, we can state the general property about constraints: answer sets of a program satisfy the propositional logic formula composed of its constraints (here (i) the notion of satisfaction is as understood classically in propositional logic and and (ii) an answer set is associated with an interpretation in an intuitive manner). Furthermore, for a program Π and a set Γ of constraints the answer sets of $\Pi \cup \Gamma$ coincide with the answer sets of Π that satisfy Γ. Consequently, constraints can be seen as elements of classical logic in logic programs.

Classical negation in programs The textbook [1] immediately introduces programs that are of more complex form even in propositional case. Indeed, it allows additional connective $\neg$ (classical negation). So that basic entities of a program are literals (a *literal* is either an atom a or an atom proceeded with classical negation $\neg a$) In turn, the concept of an answer set is defined over the consistent sets of literals whereas here we defined an answer set over the sets of atoms. Yet, it is easy to *simulate* classical negation in the simpler form of programs that we consider here.

Classical negation can always be eliminated from a program by means of auxiliary atoms and additional constraints. Indeed, given a program composed of rules of the form

$$l_0 \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n \tag{10}$$

so that l_0 is a literal or $\bot$ and $l_1 \ldots l_n$ are literals, we can replace an occurrence of any literal of the form $\neg a$ with a fresh atom a' and add a constraint to this program in the form

$$\leftarrow a, not\ a'.$$

The answer sets of this new program as defined in this handout are in one to one correspondence with the answer sets as defined in the textbook. In particular, by replacing atoms of the form a' by $\neg a$ we obtain the textbook answer sets.

2 Formalization of Graph coloring problem by means of a Propositional logic program

Consider a *graph coloring* problem *GC*:

> *A* 3*-coloring of a directed graph* (V, E) *is a labeling of its vertexes with at most* 3 *colors (named,* 1*,* 2 *and* 3*) such that no two vertexes sharing an edge have the same color.*

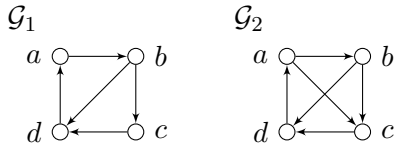

Figure 1: Sample graphs $\mathcal{G}_1$ and $\mathcal{G}_2$.

For instance, see two specific graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ in Figure 1. It is easy to see that there are six distinct 3-colorings for graph $\mathcal{G}_1$ including the following:

> assigning color 1 to vertexes a and c, color 2 to vertex b, and color 3 to vertex d forms a 3-coloring of $\mathcal{G}_1$.

We denote this 3-coloring of graph $\mathcal{G}_1$ by $\mathcal{S}_{\mathcal{G}_1}$. Graph $\mathcal{G}_2$ has no 3-colorings.

A solution to this problem can be described by a set of atoms of the form c_{vi} ($v \in V$ in a given graph (V, E) and $i \in \{1, 2, 3\}$); including c_{vi} in the set indicates that an assertion that vertex v is assigned color i. A solution is a set X satisfying the following conditions:

1. Each vertex must be assigned a color
2. A vertex may not be assigned more that one color
3. Vertexes sharing an edge must be assigned a distinct color.

A following propositional logic program under answer set semantics encodes these specifications so that its answer sets are in one to one correspondence with the 3-colorings of a given graph (V, E):

$$\begin{array}{ll} \{c_{vi}\} & (v \in V, 1 \leq i \leq 3) \\ \leftarrow \mathit{not}\ c_{v1}, \mathit{not}\ c_{v2}, \mathit{not}\ c_{v3}. & (v \in V). \\ \leftarrow c_{vi}, c_{vj} & (v \in V,\ 1 \leq i < j \leq 3), \\ \leftarrow c_{vi}, c_{wi} & (\{v, w\} \in E,\ 1 \leq i \leq 3), \end{array} \tag{11}$$

We now provide an intuitive reading of this program.

- A collection of choice rules for each vertex v captured by the first line in (11) intuitively says that vertex v may be assigned some colors. *(Condition 1)*

$\{c_{a1}\}$	$\{c_{a2}\}$	$\{c_{a3}\}$	$\{c_{b1}\}$	$\{c_{b2}\}$	$\{c_{b3}\}$	$\{c_{c1}\}$	$\{c_{c2}\}$	$\{c_{c3}\}$	$\{c_{d1}\}$	$\{c_{d2}\}$	$\{c_{d3}\}$
$\leftarrow not\ c_{a1}, not\ c_{a2}, not\ c_{a3}.$											
$\leftarrow not\ c_{b1}, not\ c_{b2}, not\ c_{b3}.$											
$\leftarrow not\ c_{c1}, not\ c_{c2}, not\ c_{c3}.$											
$\leftarrow not\ c_{d1}, not\ c_{d2}, not\ c_{d3}.$											
$\leftarrow c_{a1},\ c_{a2}$			$\leftarrow c_{a1},\ c_{a3}$			$\leftarrow c_{a2},\ c_{a3}$					
$\leftarrow c_{b1},\ c_{b2}$			$\leftarrow c_{b1},\ c_{b3}$			$\leftarrow c_{b2},\ c_{b3}$					
$\leftarrow c_{c1},\ c_{c2}$			$\leftarrow c_{c1},\ c_{c3}$			$\leftarrow c_{c2},\ c_{c3}$					
$\leftarrow c_{d1},\ c_{d2}$			$\leftarrow c_{d1},\ c_{d3}$			$\leftarrow c_{d2},\ c_{d3}$					
$\leftarrow c_{a1},\ c_{b1}$			$\leftarrow c_{a2},\ c_{b2}$			$\leftarrow c_{a3},\ c_{b3}$					
$\leftarrow c_{d1},\ c_{a1}$			$\leftarrow c_{d2},\ c_{a2}$			$\leftarrow c_{d3},\ c_{a3}$					
$\leftarrow c_{b1},\ c_{d1}$			$\leftarrow c_{b2},\ c_{d2}$			$\leftarrow c_{b3},\ c_{d3}$					
$\leftarrow c_{c1},\ c_{d1}$			$\leftarrow c_{c2},\ c_{d2}$			$\leftarrow c_{c3},\ c_{d3}$					
$\leftarrow c_{b1},\ c_{c1}$			$\leftarrow c_{b2},\ c_{c2}$			$\leftarrow c_{b3},\ c_{c3}$					

Figure 2: Logic program for 3-coloring for graph $\mathcal{G}_1$.

- The second line states that it is impossible that a vertex is not assigned a color. *(Condition 1)*
- The third line says that it is impossible that a vertex is assigned two colors. *(Condition 2)*
- The fourth line states that it is impossible that any two adjacent vertexes are assigned the same color. *(Condition 3)*

Recall graph $\mathcal{G}_1$ in Figure 1. Figure 2 presents a logic program in spirit of (11) for $\mathcal{G}_1$. Horizontal lines separate the clauses that come from distinct "schematic rules" in (11). This program has six answer sets including

$$\{c_{a1},\ c_{b2},\ c_{c1},\ c_{d3}\},$$

which captures solution $\mathcal{S}_{\mathcal{G}_1}$. To encode the 3-coloring problem for graph $\mathcal{G}_2$ one has to extend the set of rules in Figure 2 with rules

$$\leftarrow c_{a1}, c_{c1} \qquad \leftarrow c_{a2}, c_{c2} \qquad \leftarrow c_{a3}, c_{c3}.$$

This program has no answer sets, which captures the fact that this graph has no 3-colorings.

The ASP specification (11) illustrates the use of the so-called GENERATE and TEST methodology within answer set programming paradigm. The

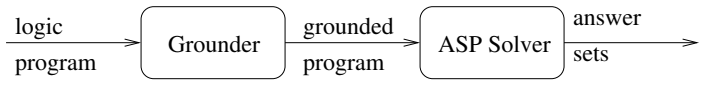

Figure 3: Common Architecture of ASP Systems

GENERATE part of the specification "defines" a collection of "perspective" answer sets that can be seen as potential solutions. The TEST part consists of conditions that eliminate the "perspective" answer sets of the GENERATE part that do not correspond to solutions. The first line in (11) corresponds to GENERATE: saying that any subset of the atoms of the form c_{vi} ($v \in V$ in a given graph (V, E) and $i \in \{1, 2, 3\}$) forms a potential solution. The remaining lines correspond to TEST. Observe, how choice rules provide a convenient tool in ASP for formulating GENERATE, whereas constraints are used to formulate TEST.

3 Programs with Variables and Grounding

ASP practitioners develop applications that rely on languages, which go beyond propositional/ground atoms. Figure 3 presents a typical architecture of an answer set programming tool that encompasses two parts: a system called *grounder* and a system called *solver*. The former is responsible for eliminating variables in a program. The latter is responsible for finding answer sets of a respective propositional (ground) program. For instance, system GRINGO[2] is a well known grounder that serve as front-ends for many solvers including CLASP. A combination of GRINGO and CLASP is known as system CLINGO.

We recall that given a signature σ consisting of object constants, variables, predicate symbols, and function symbols (where predicate and function symbols are associated with a nonnegative integer n called arity, so that we identify function symbols of arity 0 with object constants),

1. Any object constant or variable in σ is a *term*, and

   an expression of the form $f(t_1, \ldots, t_n)$ is a *term* where f is a function symbol in σ of arity $n > 0$ and $t_1, \ldots t_n$ are terms.

2. Any predicate symbol in σ of arity 0 is an atom, and

[2] `http://potassco.sourceforge.net/` .

an expression of the form $p(t_1, \ldots, t_n)$ is an *atom* where p is a predicate symbol in σ of arity n and $t_1, \ldots t_n$ are terms.

3. Any term that contains no variables is called *ground*. Similarly, any atom that contains no variables is *ground*. Otherwise, we refer to these entities as non-ground.

A *logic program with variables* is a finite set of rules of the form (1), where a_0 is symbol $\bot$ or a non-ground atom; and $a_1, \ldots, a_p$ are non-ground atoms. Grounding a logic program replaces each rule with all its instances obtained by substituting ground terms, formed from the object constants and function symbols occurring in the program, for all variables. For a program Π, by $ground(\Pi)$ we denote the result of its grounding. (We use the convention common in logic programming: variables are represented by capitalized identifiers.) We illustrate this concept on an example. Let Π be a program

$$\begin{array}{l} \{a(1)\} \ \ \{a(2)\} \ \ \{b(1)\} \\ c(X) \leftarrow a(X), b(X), \end{array} \tag{12}$$

$ground(\Pi)$ follows

$$\begin{array}{l} \{a(1)\} \ \ \{a(2)\} \ \ \{b(1)\} \\ c(1) \leftarrow a(1), b(1) \\ c(2) \leftarrow a(2), b(2). \end{array}$$

The answer sets of a program Π with variables are answer sets of $ground(\Pi)$. For instance, there are eight answer sets of program (12) including $\emptyset$ and set $\{a(1)\ b(1)\ c(1)\}$.

Problem 1. *(a) Consider the following program with variables*

$$\begin{array}{l} \{a(1)\}. \ \ \{a(2)\}. \ \ \{b(1)\}. \\ d(X, Y) \leftarrow a(X), b(Y). \end{array} \tag{13}$$

Construct the result of grounding for this program.

(b) Follow the link `https://potassco.org/clingo/run/` *. Replace symbol "$\leftarrow$" by ":-" in the ground program you constructed in (a) and let* CLINGO *run on your program using* reasoning mode *"enumerate all".*

(c) Using the same procedure as in (b), find all answer sets for the program listed in (a). Do answer sets found in (b) coincide with the ones enumerated in this step?

Given a program Π with variables, grounders often produce a variable-free program that is smaller than $ground(\Pi)$, but still has the same answer

sets as $ground(\Pi)$; we call any such program an *image* of Π. For example, program

$$\{a(1)\}\ \ \{a(2)\}\ \ \{b(1)\}$$
$$c(1) \leftarrow a(1), b(1)$$

is an image of (12).

Passing parameter $-t$ in command line when calling CLINGO on a program will force the system to produce ground program in human readable form. Running CLINGO in online interface with checkbox "statistics" allows one to obtain valuable information about the execution of the system. For example, lines titled

- "Rules" provides number of rules in a grounding of the input and suggests the relative size of a ground program,
- "Choices" corresponds to the number of backtracks done by the system in search for the solution (we will learn of this feature in a little bit),
- "Time" reports the execution time of the system.

In command line to obtain statistics while running CLINGO use flag "--stats"

Problem 2. *Let* CLINGO *run on the program* (13) *using* reasoning mode *"enumerate all" and marking* checkbox *"statistics". What is the number of rules that* CLINGO *reports?*

This number corresponds to the size of the image (measured in number of rules) produced by GRINGO *after grounding the input program.*

Think of an image for program (13) *that is of the same size as* GRINGO *computed. List this image.*

When a program Π with variables has at least one function symbol and at least one object constant, grounding results in infinite $ground(\Pi)$. Yet, even for an input program of this kind, grounders often find an image that is a finite propositional program (finite image). For instance, for program

$$\begin{array}{l} p(0) \\ q(f(X)) \leftarrow p(X) \end{array} \tag{14}$$

grounding results in infinite program outlined below

$$\begin{array}{l} p(0) \\ q(f(0)) \leftarrow p(0) \\ q(f(f(0))) \leftarrow p(f(0)) \\ q(f(f(f(0)))) \leftarrow p(f(f(0))) \\ \ldots \end{array}$$

A finite image of program (14) follows

$$p(0)$$
$$q(f(0)) \leftarrow p(0).$$

Program

$$p(0)$$
$$q(f(0))$$

is another image of (14). In fact, given program (14) as an input grounder GRINGO will generate the latter image.

To produce images for input programs, grounders follow techniques exemplified by intelligent grounding. Different grounders implement distinct procedures so that they may generate different images for the same input program. One can intuitively measure the quality of a produced image by its size so that the smaller the image is the better. A common syntactic restriction that grounders pose on input programs is "safety". A program Π is *safe* if every variable occurring in a rule of Π also occurs in positive body of that rule. For instance, programs (12) and (14) are safe. The safety requirement suggests that positive body of a rule must contain information on the values that should be substituted for a variable in the process of grounding. Safety is instrumental in designing grounding techniques that utilize knowledge about the structure of a program for constructing smaller images. The GRINGO grounder and the grounder of the DLV system expect programs to be safe. For programs with function symbols, to guarantee that the grounding process terminates, grounders pose additional syntactic restrictions (in other words, to guarantee that a grounder is able to construct a finite image).

4 Formalization of Graph coloring problem by means of a logic program with variables

We now revisit our graph coloring example and illustrate how often a set of propositional rules that follow a simple pattern can be represented concisely by means of logic programs with variables. Recall program (11). We now capture atoms of the form c_{vi} by expressions $c(v, i)$, where c is a predicate symbol and v, i are object constants denoting a vertex v and color i respectively. Atom of the form $vtx(v)$, intuitively, states that an object constant v is a vertex, while atom $e(v, w)$ states that there is an edge from vertex v to vertex w in a given graph. Atom $color(i)$ states that an object

constant i represents a color. Recall graph coloring problem GC for an input graph (V, E). We now present a program with variables that encodes a solution to this problem. First, this program consists of facts that encode graph (V, E):

$$\begin{array}{ll} vtx(v) & (v \in V) \\ e(v, w) & (\{v, w\} \in E) \end{array} \tag{15}$$

Second, facts

$$color(c) \qquad (c \in \{1, 2, 3\}) \tag{16}$$

enumerate three colors of the problem. The following rules conclude the description of the program:

$$\{c(V, I)\} \leftarrow vtx(V),\ color(I) \tag{17}$$

$$\leftarrow not\ c(V, 1), not\ c(V, 2), not\ c(V, 3),\ vtx(V) \tag{18}$$

$$\leftarrow c(V, I),\ c(V, J),\ I < J,\ vtx(V),\ color(I),\ color(J) \tag{19}$$

$$\leftarrow c(V, I),\ c(W, I),\ vtx(V),\ vtx(W),\ color(I),\ e(V, W) \tag{20}$$

These rules are the counterparts of groups of rules in propositional program (11). Indeed,

- rule (17) states that every vertex may be assigned some colors;
- the second rule (18) states that it is impossible that a vertex is not assigned a color;
- rule (19) says that it is impossible that a vertex is assigned two colors; and
- rule (20) says that it is impossible that any two adjacent vertexes are assigned the same color.

Programs with variables permit for a concise encoding of an instance of a search problem. Indeed, size of a program composed of rules (15-18) is almost identical to the size of a given graph (V, E). There are $|V| + |E| + 7$ rules in this program. On the other hand, the line

$$\leftarrow c_{vi}, c_{wi} \qquad (\{v, w\} \in E,\ 1 \leq i \leq 3)$$

of program (11) alone encapsulates $3|E|$ rules.

5 Modeling of Search Problems in ASP

Answer set programming provides a general purpose modeling language that supports elaboration tolerant solutions for search problems. We now define a search problem abstractly. A *search problem* P consists of a set of instances with each *instance* I assigned a finite set $S_P(I)$ of solutions. In answer set programming to solve a search problem P, we construct a program Π_P that captures problem specifications so that when extended with facts D_I representing an instance I of the problem, the answer sets of $\Pi_P \cup D_I$ are in one to one correspondence with members in $S_P(I)$. In other words, answer sets describe all solutions of problem P for the instance I. Thus solving of a search problem is reduced to finding a uniform encoding of its specifications by means of a logic program with variables.

For example, an instance of the graph coloring search problem GC is a graph. All 3-colorings for a given graph form its solutions set. Consider any graph (V, E). By $D_{(V,E)}$ we denote facts in (15) that encode graph (V, E). By Π_{gc} we denote a program composed of rules in (16-18). This program captures specifications of 3-coloring problem so that answer sets of $\Pi_{gc} \cup D_{(V,E)}$ correspond to solutions to instance graph (V, E) of a problem. Recall graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ presented in Figure 1. Facts $D_{\mathcal{G}_1}$ representing $\mathcal{G}_1$ follow

$$vtx(a)\ \ vtx(b)\ \ vtx(c)\ \ vtx(d)\ \ e(a,b)\ \ e(b,c)\ \ e(c,d)\ \ e(d,a)\ \ e(b,d).$$

Program $\Pi_{gc} \cup D_{\mathcal{G}_1}$ has six answer sets, including

$$\begin{array}{l}\{vtx(a)\ vtx(b)\ vtx(c)\ vtx(d)\\ e(a,b)\ e(b,c)\ e(c,d)\ e(d,a)\ e(b,d)\\ color(1)\ color(2)\ color(3)\\ c(a,1)\ c(b,2)\ c(c,1)\ c(d,3)\},\end{array}$$

which captures solution $\mathcal{S}_{\mathcal{G}_1}$. Similarly, we can use encoding Π_{gc} to establish whether 3-colorings exist for graph $\mathcal{G}_2$. Facts $D_{\mathcal{G}_2}$ representing $\mathcal{G}_2$ consists of facts in $D_{\mathcal{G}_1}$ and an additional fact $e(a, c)$. Program $\Pi_{gc} \cup D_{\mathcal{G}_2}$ has no answer sets suggesting that no 3-colorings exist for graph $\mathcal{G}_2$.

It is important to mention that the languages supported by ASP grounders and solvers go beyond rules presented here. For instance, GRINGO versions 4.5+ support such constructs as aggregates, cardinality expressions, intervals, pools. It is beyond the scope of this lecture to formally discuss these constructs, but it is worth mentioning that they generally allow us more concise, intuitive, and elaboration tolerant encodings of problems.

Also, they often permit to utilize more sophisticated and efficient procedures in solving.

For instance, a single rule

$$\leftarrow \mathit{not}\ 1\#\mathit{count}\{V, I : c(V, I)\}1,\ \mathit{vtx}(V). \tag{21}$$

can replace two rules (18) and (19) in program Π_{gc}. This rule states that

it must be the case that a vertex is assigned exactly one color. (22)

This shorter program will result in smaller groundings for instances of the *GC* problem paving the way to more efficient solving. Cardinality construct

$$1\#\mathit{count}\{V, I : c(V, I)\}1$$

intuitively suggests us to count, for a given value of V the tuples (V, I) for which atom of the form $c(V, I)$ belongs to the answer set. Number 1 to the right and to the left of this aggregate expression tells us a specific condition on the count, in particular, that it has to be exactly 1. Indeed, the number to the right suggests at most count whereas the number to the left suggest at least count. Cardinality expressions form only one example of multitude of constructs that GRINGO language offers for effective modeling of problem specifications.

It is worth noting that the GRINGO language also provides *syntactic sugar*, i.e., convenient abbreviations for groups of rules. For example, expression

$$\mathit{edge}(a, b; b, c; c, d).$$

abbreviates the set of facts

$$\mathit{edge}(a, b).\ \mathit{edge}(b, c).\ \mathit{edge}(c, d).$$

Expression

$$1\{c(V, I) : \mathit{color}(I)\} \leftarrow \mathit{vtx}(V).$$

abbreviates rules (17) and (18). So that we can encode Condition 1 listed in Section 2 by one rule.

Expression

$$1\{c(V, I) : \mathit{color}(I)\}1 \leftarrow \mathit{vtx}(V). \tag{23}$$

abbreviates a collection of two rules (17) and (21). Intuitively, we can read the meaning of this rule as stated in (22) that we used to characterize an intuitive meaning of constraint (21). Although English statements turn out

to be the same for rules (21) and (23), the formal meaning of mathematical expression (23) obviously extends that of (21). In addition to a constraint on solutions captured by (21) [which is one of the rules abbreviated by expression (23)], choice rule (17) [the other rule abbreviated by expression (23)] provides ground for atoms of the form $c(\cdot,\cdot)$ be part of solutions. Or, in other words, choice rule (17) removes closed world assumption from $c(\cdot,\cdot)$ atoms.

Problem 3. *Note that directive #show $c/2$. added to a* CLINGO *program Π_{gc} allows one to instruct* CLINGO *to only output these atoms in the computed answer sets that have the form $c(\cdot,\cdot)$.*

(a) Recall the statement of graph coloring problem. Imagine the following extension to that statement: there is at most one node in a given graph colored by color named 1. *Extend the program Π_{gc} so that this new statement is respected.*

(b) For graph $\mathcal{G}_1$, how many solutions to the new graph coloring problem stated in (a) are there?

(c) Use CLINGO *to test your solution in by running it on the program $\Pi_{gc} \cup D_{\mathcal{G}_1}$ extended with the code you developed in (a). List the answer sets that* CLINGO *computes (list only the atoms of the form $c(\cdot,\cdot)$).*

6 The Generate-Define-and-Test Modeling Methodology of ASP

Previously, we presented how the GENERATE and TEST methodology is applicable within answer set programming. Yet, an essential feature of logic programs is their ability to elegantly and concisely "define" predicates. Logic programs provide a convenient language for expressing inductive/recursive definitions.

The GENERATE, DEFINE, and TEST is a typical methodology used by ASP practitioners in designing programs. It generalizes the GENERATE and TEST methodology discussed earlier. The roles of the GENERATE and TEST parts of a program stay the same so that, informally,

- GENERATE defines a large collection of answer sets that could be seen as potential solutions, while
- TEST consists of rules that eliminate the answer sets of the GENERATE part that do not correspond to solutions.
- The DEFINE section expresses additional, auxiliary concepts and connects the GENERATE and TEST parts.

To illustrate the essence of DEFINE, consider a *Hamiltonian cycle* search problem:

> Given a directed graph (V, E), the goal is to find a *Hamiltonian cycle* — a set of edges that induce in (V, E) a directed cycle going through each vertex exactly once.

This is an important combinatorial search problem related to Traveling Salesperson problem. A solution can be described by a set of atoms of the form $in(v, w)$, $(v, w) \in E$ of the given graph (V, E); including $in(v, w)$ in the set indicates that an edge from vertex v to vertex w is part of a found Hamiltonian cycle. A solution is a set X satisfying the following conditions

1. the Hamiltonian cycle is formed by the edges of a given graph (V, E), or, in other words, the extension of predicate in is a subset of E,
2. X does not contain a pair of different atoms of the form $in(u, v)$, $in(u', v)$ (two selected edges end at the same vertex; thus we visit node u only once),
3. For each pair u, v of vertexes, X is such that (u, v) is a part of the transitive closure of in relation defined by X. (Thus, a found subset of edges of the graph is indeed a cycle.) Recall that
   - the *transitive closure* of a binary relation R (in our case relation in) on a set Y of elements (in our case the set of the vertexes of the given graph) is the smallest relation on Y that contains R and is transitive.
   - a *transitive* relation R on set Y of elements is such that for any three elements a, b, c (not necessarily distinct elements) in Y the following property holds if a is in relation R with b and b is in relation R with c then a is in relation R with c.

We now formalize the specifications of Hamiltonian cycle problem by means of GENERATE, DEFINE, and TEST methodology. As in the encoding Π_{gc} of the graph coloring problem, we use expressions of the form $vtx(v)$ and $e(v, w)$ to encode an input graph.

Choice rule

$$\{in(X, Y)\} \leftarrow e(X, Y) \tag{24}$$

forms the GENERATE part of the problem. This rule states that any subset of edges of a given graph may form a Hamiltonian cycle (*Condition 1*). Answer

sets of a program composed of this rule and a set of facts encoding an input graph will correspond to all subsets of edges of the graph. For instance, program composed of facts $D_{\mathcal{G}_1}$ that encode directed graph $\mathcal{G}_1$ introduced in Figure 1 extended by rule (24) has 32 answer sets each representing a different subset of its edges.

The remaining *Conditions 2-3* are captured in the TEST part. To formulate Condition 3, an auxiliary concept of *reachable* (a transitive closure of *in*) is required so that we can capture the restriction that a found subset of edges of the graph is also a cycle. The DEFINE part follows

$$\begin{array}{l} reachable(V,V) \leftarrow vtx(V) \\ reachable(U,W) \leftarrow in(U,V),\ reachable(V,W), \\ \qquad\qquad\qquad\qquad vtx(U),\ vtx(V),\ vtx(W) \end{array} \tag{25}$$

These rules define *transitive closure* of the predicate *in*: all pairs of vertexes (u, v) such that v can be reached from u by following zero or more edges that are *in*.

We are now ready to state the TEST part composed of three rules

$$\begin{array}{l|l} \text{Condition 2} & \leftarrow in(U,V),\ in(W,V),\ U \neq W,\ vtx(U;V;W) \\ \text{Condition 3} & \leftarrow not\ reachable(U,V),\ vtx(U;V). \end{array} \tag{26}$$

Rules (24), (25), and (26) form a program Π_{hc} that captures specifications of Hamiltonian cycle search problem. Extending Π_{hc} with facts representing a directed graph results in a program whose answer sets describe all Hamiltonian cycles of this graph. For example, program $\Pi_{hc} \cup D_{\mathcal{G}_1}$ has only one answer set. Set

$$\{in(a,b)\ in(b,c)\ in(c,d)\ in(d,a)\}$$

contains all *in*-edges of that answer set stating that edges (a, b), (b, c), (c, d), and (d, a) form the only Hamiltonian cycle for graph $\mathcal{G}_1$.

Concise encoding of transitive closure is a feature of answer set programming that constitutes an essential difference between ASP and formalisms based on classical logic. For example, transitive closure is not expressible by first-order formulas. Thus any subset of first-order logic taken as the language with variables for modeling search problems declaratively will fail at defining (directly) concepts that rely on transitive closure.

In mastering the art of answer set programming it is enough to develop intuitions about answer sets of the programs with variables that are formed in accordance with the GENERATE, DEFINE, and TEST methodology. Some of these intuitions will stem from the general properties you encountered earlier

such as any element of answer set must appear in the head of some rule in a program. For the general definition of an answer set, it is more difficult to develop intuitions on what answer sets conceptually are and unnecessary.

Problem 4. *Recall that the rule*

$$\leftarrow in(U,V),\ in(W,V),\ U \neq W,\ vtx(U;V;W)$$

in Π_{hc} *states that no two selected edges end at the same node. Rewrite this rule using aggregate #count exemplified in rule* (21)*. You may wish to consult Clingo-Gringo manual Section 3.1.12 for more details on aggregates.*

7 ASP Formulation of n-Queens

We now turn our attention to another combinatorial search problem: n-queens problem.

> The goal is to place n queens on an $n \times n$ chessboard so that no two queens would be placed on the same row, column, and diagonal.

A solution can be described by a set of atoms of the form $q(i,j)$ $(1 \leq i,j \leq n)$; including $q(i,j)$ in the set indicates that there is a queen at position (i,j). A solution is a set X satisfying the following conditions:

1. the cardinality of X is n,
2. X does not contain a pair of different atoms of the form $q(i,j)$, $q(i',j)$ (two queens on the same row),
3. X does not contain a pair of different atoms of the form $q(i,j)$, $q(i,j')$ (two queens on the same column),
4. X does not contain a pair of different atoms of the form $q(i,j)$, $q(i',j')$ with $|i'-i| = |j'-j|$ (two queens on the same diagonal).

Here is the representation of this program in the input language of CLINGO:

```
number(1..n).

%Condition 1 and 2
1{q(K,J): number(K)}1:- number(J).
```

```
%Condition 3
:-q(I,J), q(I,J1), J<J1.

%Condition 4
:-q(I,J), q(I1,J1), J<J1, |I1-I|==J1-J.
```

We name this program *queens.clingo*.

Appending the line

```
# const n=8.
```

to the code in *queens.clingo* will instruct answer set system CLINGO to search for solution for 8-queens problem. Alternatively, the command line

```
clingo -c n=8 queens.clingo
```

instructs the answer set system CLINGO to find a single solution for 8-queens problem, whereas the command line

```
clingo -c n=8 queens.clingo 0
```

instructs CLINGO to find all solutions to 8-queens program. The command line

```
gringo -c n=8 queens.clingo >  queens.8.grounded
```

instructs the grounder GRINGO to ground 8-queens problem; the ground problem (ready for processing with CLASP) is stored in file *queens.8.grounded*. The command lines

```
gringo -t -c n=8 queens.clingo
```

or

```
clingo -t -c n=8 queens.clingo
```

will produce human-readable grounded 8-queens problem.

The command line

```
clasp < queens.8.grounded
```

will instruct the answer set solver CLASP to look for answer sets of a program in *queens.8.grounded*.

An extract from the output of the last command line follows

```
...
Answer: 92
number(1) number(2) number(3) number(4)
number(5) number(6) number(7) number(8)
q(5,8) q(7,7) q(2,6) q(6,5) q(3,4) q(1,3) q(8,2) q(4,1)
SATISFIABLE
```

This 92nd solution found by the solver encodes the following valid configuration of queens on the board

```
 1 2 3 4 5 6 7 8
1    Q
2          Q
3      Q
4Q
5              Q
6        Q
7            Q
8  Q
```

Similarly, appending the line

```
# const n=4.
```

to the code in *queens.clingo* will instruct CLINGO to solve 4-queens problem. The command line

```
clingo -c n=4 queens.clingo 0
```

instructs CLINGO to find all solutions for 4-queens problem.

Problem 5. *(a) Use* CLINGO *to find all solutions to the* 8*-queens problem that have a queen at* $(1, 1)$*. How many solutions of the kind are there?*

(b) Use CLINGO *to find all solutions to the* 12*-queens problem that have a queen at* $(1, 1)$*. How many solutions of the kind are there?*

Submit the lines of code that you wrote to solve these problems.

Problem 6. *(a) Use* CLINGO *to find all solutions to the* 8*-queens problem that have no queens in the* 4×4 *square in the middle of the board. How many solutions of the kind are there?*

(b) Use CLINGO *to find all solutions to the* 10*-queens problem that have no queens in the* 4×4 *square in the middle of the board. How many solutions of the kind are there?*

Submit the lines of code that you wrote to solve these problems.

Acknowledgments

Parts of this handout follow *What is answer set programming to propositional satisfiability*, Yuliya Lierler, Constraints, July 2017, Volume 22, Issue 3, pp 307337 available at `https://link.springer.com/article/10.1007/s10601-016-9257-7`.

References

[1] Michael Gelfond and Yulia Kahl. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach.* Cambridge University Press, 2014.