Handout on Answer Set Programming

Yuliya Lierler
Univresity of Nebraska, Omaha

Tell me and I forget. Show me and I remember. Involve me and I understand. (Chinese proverb)

The Moore method is a deductive manner of instruction used in advanced mathematics courses. It is named after Robert Lee Moore, a famous topologist who first used a stronger version of the method at the University of Pennsylvania when he began teaching there in 1911.

The way the course is conducted varies from instructor to instructor, but the content of the course is usually presented in whole or in part by the students themselves. Instead of using a textbook, the students are given a list of definitions and theorems which they are to prove and present in class, leading them through the subject material. The Moore method typically limits the amount of material that a class is able to cover, but its advocates claim that it induces a depth of understanding that listening to lectures cannot give.
(`http://en.wikipedia.org/wiki/Moore_method`)

... doing mathematical proofs is high art, like playing the violin. You have to practice for years before you make significant progress, and a lot depends on your innate abilities. But in the process you will improve your ability to distinguish between correct and incorrect mathematical arguments and to appreciate clever proofs. This is much easier than inventing proofs, just like it's easier to appreciate good music than to play an instrument.
(Vladimir Lifschitz, personal communication, Oct 10, 2013)

Introduction

Answer set programming (ASP) is a form of declarative programming oriented towards difficult combinatorial search problems. It belongs to the group of so called constraint programming languages. ASP has been applied to a variety of applications including plan generation and product configuration problems in artificial intelligence and graph-theoretic problems arising in VLSI design and in historical linguistics [1].

Syntactically, ASP programs look like logic programs in Prolog, but the computational mechanisms used in ASP are different: they are based on the ideas stemming from the development of satisfiability solvers for propositional logic.

We discuss the concept of an answer set in Section 1. Section 2 is devoted to the methodology of answer set programming as well as the use of software systems for computing answer sets. A graph coloring problem is utilized to illustrate the use of answer set programming in practice. Section 3 presents a solution to n-queens problem. In all mentioned sections 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**.

1 Traditional Programs and their Answer Sets

1.1 Syntax

A *traditional rule* is an expression of the form

$$A_0 \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n. \tag{1}$$

where $n \geq m \geq 0$ and $A_0, \ldots, A_n$ are propositional atoms (propositional symbols). The atom A_0 is called the *head* of the rule, and the list

$$A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n$$

is its *body*. If the body is empty ($n = 0$) then the rule is called a *fact* and identified with its head A_0.

A *traditional program* is a finite set of traditional rules. For instance,

$$\begin{array}{l} p. \\ r \leftarrow p, q. \end{array} \tag{2}$$

and

$$\begin{array}{l} p \leftarrow not\ q. \\ q \leftarrow not\ r. \end{array} \tag{3}$$

are traditional programs.

A traditional rule (1) is *positive* if $m = n$, that is to say, if it has the form

$$A_0 \leftarrow A_1, \ldots, A_m. \tag{4}$$

A traditional program is *positive* if each of its rules is positive. For instance, program (2) is positive, and (3) is not.

1.2 The Answer Set of a Positive Program

We will first define the concept of an answer set for positive traditional programs. To begin, we introduce auxiliary definitions.

Definition 1. *A set X of atoms* satisfies *a positive traditional rule* (4) *when* $A_0 \in X$ *whenever* $\{A_1, \ldots, A_m\} \subseteq X$.

For instance, any positive traditional rule (4) is satisfied by a singleton set $\{A_0\}$.

To interpret Definition 1 recall the truth table of implication in propositional logic:

p	q	$p \rightarrow q$
$true$	$true$	$true$
$true$	$false$	$false$
$false$	$true$	$true$
$false$	$false$	$true$

One can intuitively read it in English as follows condition $p \rightarrow q$ holds if q holds whenever p holds. Expression $A_0 \in X$ plays a role of q whereas $\{A_1, \ldots, A_m\} \subseteq X$ plays a role of p in the definition of a set of atoms satisfying a rule.

Problem 1. *(2pt) Given a set X of atoms and a positive traditional rule* (4)

	Does X satisfies rule (4)*?*
$\{A_1, \ldots, A_m\} \subseteq X$ *and* $A_0 \in X$	Yes
$\{A_1, \ldots, A_m\} \subseteq X$ *and* $A_0 \notin X$	
$\{A_1, \ldots, A_m\} \not\subseteq X$ *and* $A_0 \in X$	Yes
$\{A_1, \ldots, A_m\} \not\subseteq X$ *and* $A_0 \notin X$	

Definition 2. *A set X of atoms* satisfies *a positive traditional program* Π *if X satisfies every rule (4) in* Π.

For instance, any positive traditional program is satisfied by the set composed of the heads A_0 of all its rules (4).

Problem 2. *(3pt)*

X	*Does* X *satisfies program* (2)*?*
$\emptyset$	No
$\{p\}$	Yes
$\{q\}$	
$\{r\}$	
$\{p\ q\}$	
$\{p\ r\}$	
$\{q\ r\}$	
$\{p\ q\ r\}$	

Proposition 1. *For any positive traditional program* Π*, the intersection of all sets satisfying* Π *satisfies* Π *also.*

Proof. By contradiction. Suppose that this is not the case. Let X denote the intersection of all sets satisfying Π. By definition (of program's satisfiability), there exists a rule

$$A_0 \leftarrow A_1, \ldots, A_m.$$

in Π such that it is not satisfied by X, in other words

$$A_0 \notin X$$

and

$$\{A_1, \ldots, A_m\} \subseteq X.$$

Since X is an intersection of all sets satisfying Π then we conclude that (i) $\{A_1, \ldots, A_m\}$ belongs to each one of the sets satisfying Π and (ii) there is a set Y satisfying Π such that $A_0 \notin Y$. By definition, Y does not satisfy Π. We derive at contradiction.

□

Proposition 1 allows us to talk about the *smallest* set of atoms that satisfies Π.

Definition 3. *The smallest set of atoms that satisfies positive traditional program* Π *is called* the answer set *of* Π.

For instance, the sets of atoms satisfying program (2) are

$$\{p\},\ \{p, r\},\ \{p, q, r\},$$

and its answer set is $\{p\}$.

Proposition 2. *If X is an answer set of a positive traditional program Π, then every element of X is the head of one of the rules of Π.*

Intuitively, we can think of (4) as a rule for generating atoms: once you have generated $A_1, \ldots, A_m$, you are allowed to generate A_0. The answer set is the set of all atoms that can be generated by applying rules of the program in any order. For instance, the first rule of (2) allows us to include p in the answer set. The second rule says that we can add r to the answer set if we have already included p and q. Given these two rules only, we can generate no atoms besides p. If we extend program (2) by adding the rule

$$q \leftarrow p.$$

then the answer set will become $\{p, q, r\}$.

Positive rules may remind you *Horn clauses* or *definite clauses.* One can identify (4) with the following implication

$$A_1 \wedge \cdots \wedge A_m \Rightarrow A_0$$

that is equivalent to the Horn clause

$$\neg A_1 \vee \cdots \vee \neg A_m \vee A_0.$$

Rule (4) is satisfied by a set of atoms if and only if its respective Horn clause is satisfied by this set in propositional logic.

Problem 3. *(Extra credit: 5pt) Prove the claim of Proposition 2.*

1.3 Answer Sets of a Program with Negation

To extend the definition of an answer set to arbitrary traditional programs, we will introduce one more auxiliary definition.

Definition 4. *The* reduct Π^X *of a traditional program* Π *relative to a set X of atoms is the set of rules (4) for all rules (1) in* Π *such that* $A_{m+1}, \ldots, A_n \notin X$.

In other words, Π^X is constructed from Π by (i) dropping all rules (1) such that at least one atom from $A_{m+1}, \ldots, A_n$ is in X, and (ii) eliminating *not* $A_{m+1}, \ldots,$ *not* A_n expression from the rest of the rules.

Thus Π^X is a positive traditional program.

Problem 4. *(5pt) Let* Π *be* (3),

X	*What is* Π^X*?*	*Explanation*
$\emptyset$	$p.$ $q.$	$p \leftarrow$ ~~$not\ q.$~~ $q \leftarrow$ ~~$not\ r.$~~
$\{p\}$	$p.$ $q.$	$p \leftarrow$ ~~$not\ q.$~~ $q \leftarrow$ ~~$not\ r.$~~
$\{q\}$	$q.$	~~$p \leftarrow not\ q.$~~ $q \leftarrow$ ~~$not\ r.$~~
$\{r\}$		
$\{p\ q\}$		
$\{p\ r\}$		
$\{q\ r\}$		
$\{p\ q\ r\}$		

Definition 5. *We say that* X *is an* answer set *of* Π *if* X *is the answer set of* Π^X *(that is, the smallest set of atoms satisfying* Π^X*).*

Problem 5. *(1pt)*

X	*Is* X *an answer set of program* (3)*?*
$\emptyset$	*No*
$\{p\}$	*No*
$\{q\}$	*Yes*
$\{r\}$	
$\{p\ q\}$	
$\{p\ r\}$	
$\{q\ r\}$	
$\{p\ q\ r\}$	

If Π is positive then, for any X, $\Pi^X = \Pi$. It follows that the new definition of an answer set is a generalization of the definition from Section 1.2: for any positive traditional program Π, X is the smallest set of atoms satisfying Π^X iff X is the smallest set of atoms satisfying Π.

Intuitively, rule (1) allows us to generate A_0 as soon as we generated the atoms $A_1, \ldots, A_m$ *provided that none of the atoms* $A_{m+1}, \ldots, A_n$ *can be* *generated using the rules of the program.* There is a vicious circle in this sentence: to decide whether a rule of Π can be used to generate a new atom, we need to know which atoms can be generated using the rules of Π. The definition of an answer set overcomes this difficulty by employing a "fixpoint construction." Take a set X that you suspect may be exactly the set of atoms that can be generated using the rules of Π. Under this assumption, Π has the same meaning as the positive program Π^X. Consider the answer set of Π^X, as defined in Section 1.2. If this set is exactly identical to the set X that you started with then X was a "good guess"; it is indeed an answer set of Π.

In Problem 5, to find all answer sets of program (3) we constructed its reduct for each subset of $\{p, q, r\}$ to establish whether these sets are answer sets of (3). The following general properties of answer sets of traditional programs allow us to sometime establish that a set is not an answer set in a trivial way by inspecting its elements rather than constructing the reduct of a given program.

Proposition 3. *If X is an answer set of a traditional program Π then every element of X is the head of one of the rules of Π.*

Proposition 4. *For any traditional program Π and any sets X, Y of atoms, if $X \subseteq Y$ then $\Pi^Y \subseteq \Pi^X$.*

Proposition 5. *If X is an answer set for a traditional program Π then no proper subset of X can be an answer set of Π.*

In application to program (3), Proposition 3 tells us that its answer sets do not contain r, so that we only need to check $\emptyset$ and $\{p, q\}$. By Proposition 3, $\emptyset$ cannot be an answer set because it is a proper subset of the answer set $\{q\}$, and $\{p, q\}$ cannot be an answer set because the answer set $\{q\}$ is its proper subset. Consequently, $\{q\}$ is the only answer set of (3).

Program (3) has a unique answer set. On the other hand, the program

$$\begin{array}{l} p \leftarrow \mathit{not}\ q. \\ q \leftarrow \mathit{not}\ p. \end{array} \tag{5}$$

has two answer sets: $\{p\}$ and $\{q\}$. The one-rule program

$$r \leftarrow \mathit{not}\ r. \tag{6}$$

has no answer sets.

Problem 6. *(5pt) Prove that if X is an answer set of a traditional program Π so that for some rule (1), it holds that $\{A_1, \ldots, A_m\} \subseteq X$ and $\{A_{m+1}, \ldots, A_n\} \cap X = \emptyset$, then $A_0 \in X$.*

Problem 7. *(5pt) Find all answer sets of the following program, which extends (5) by two additional rules:*

$$
\begin{array}{l}
p \leftarrow not\ q. \\
q \leftarrow not\ p. \\
r \leftarrow p. \\
r \leftarrow q.
\end{array}
$$

Problem 8. *(5pt) Find all answer sets of the following combination of programs (5) and (6):*

$$
\begin{array}{l}
p \leftarrow not\ q. \\
q \leftarrow not\ p. \\
r \leftarrow not\ r. \\
r \leftarrow p.
\end{array}
$$

Problem 9. *(Extra credit: 5pt) Prove the claim of Proposition 3.*

Problem 10. *(Extra credit: 5pt) Prove the claim of Proposition 4.*

Problem 11. *(Extra credit: 5pt) Prove the claim of Proposition 5.*

2 Answer Set Programming

Answer set programming (ASP) [1] is a declarative programming formalism based on the answer set semantics of logic programs. 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 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.

A common methodology to solve a problem in ASP is to design GENERATE, DEFINE, and TEST parts of a program. The GENERATE part defines a

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

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

In addition to Prolog-like rules such as rule (1), GRINGO also accepts rules of other kinds — "choice rules" and "constraints". For example, rule

$$\{p, q, r\}.$$

is a choice rule. Answer sets of this one-rule program are:

$$\emptyset,\ \{p\},\ \{q\},\ \{r\},\ \{p,q\},\ \{p,r\},\ \{q,r\},\ \{p,q,r\}.$$

Choice rules are typically the main members of the GENERATE part of the program. Constraints often form the TEST section of a program. Syntactically, a constraint is the rule with an empty head (we can, intuitively, understand empty head as $false$). It encodes the conditions on the answer sets that have to be met. For instance, the constraint

$$\leftarrow p,\ not\ q.$$

eliminates the answer sets of a program that include p and do not include q. Or, in other words, the situations when p holds while q does not hold lead to contradiction ($false$).

System GRINGO allows the user to specify large programs in a compact way, using rules with *(schematic) variables* and other abbreviations. *A detailed description of its input language can be found in the online Guide* (see url listed earlier). Grounder GRINGO takes a program "with abbreviations and variables" as an input and produces its propositional counterpart that is then processed by CLASP. The system CLINGO can be used as a shortcut for invoking both of these systems at once.

In this handout, we use the convention common in logic programming: variables are represented by capitalized identifiers. For a program Π with variables, by $ground(\Pi)$ we denote the result of its grounding – a program that contains no variables (such as programs we discussed in Section 1). We bypass formal definition of grounding process while illustrating it using examples. Let Π be a program with variables

$$\begin{array}{l} \{a(1)\}.\ \ \{a(2)\}.\ \ \{b(1)\}. \\ c(X) \leftarrow a(X), b(X). \end{array} \tag{7}$$

ground(Π) follows

$$\begin{array}{l} \{a(1)\}.\ \ \{a(2)\}.\ \ \{b(1)\}. \\ c(1) \leftarrow a(1), b(1). \\ c(2) \leftarrow a(2), b(2). \end{array} \tag{8}$$

Symbols 1 and 2 are the "constants" that occur in (7). These constants are used to instantiate the only rule

$$c(X) \leftarrow a(X), b(X) \tag{9}$$

with variables in this program so that it results in two ground instances

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

Substituting rule (9) in program (7) by rule

$$d(X, Y) \leftarrow a(X), b(Y).$$

results in the following grounding:

$$\begin{array}{l} \{a(1)\}.\ \ \{a(2)\}.\ \ \{b(1)\}. \\ d(1,1) \leftarrow a(1), b(1). \\ d(1,2) \leftarrow a(1), b(2). \\ d(2,1) \leftarrow a(2), b(1). \\ d(2,2) \leftarrow a(2), b(2). \end{array} \tag{10}$$

The answer sets of a program Π with variables are answer sets of *ground*(Π). For instance, there are eight answer sets of program (7) including ∅ and set $\{a(1)\ b(1)\ c(1)\}$.

Problem 12. *(2pt) (a) Follow the link* ***https://potassco.org/clingo/run/*** *. Replace symbol "←" by ":-" in* (10) *and let* CLINGO *run on the program* (10) *using* reasoning mode *"enumerate all".*

(b) Recall that (10) *is the result of grounding the following program with variables*

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

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

Given a program Π with variables, CLINGO often produces 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

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

is an image of (7). In fact, given program (7) as an input grounder GRINGO of CLINGO will generate this image. To produce images for input programs, grounders follow techniques exemplified by intelligent grounding [2]. 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 (7) is 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. System CLINGO can process only safe programs. Passing parameter $-t$ in command line when calling CLINGO on a program will force the system to produce ground program in human readable form.

Problem 13. *(1pt) Let* CLINGO *run on the program* (10) *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.*

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,
- "Time" reports the execution time of the system.

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

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 in Section 1 such as any element of answer set must appear in the head of some rule in a program (see Problem 3). For the general definition of an answer set, it is more difficult to develop intuitions on what answer sets conceptually are and unnecessary.

To illustrate discussed concepts, we look at the formalization of 3-coloring problem GC in answer set programming.

> *A* 3*-coloring of a graph is a labeling of its vertexes with at most* 3 *colors such that no two vertexes sharing the same edge have the same color.*

Logic programs with variables can often represent the statements of such combinatorial search problems as GC concisely. Atoms of the form $c(v,i)$, where c is a predicate symbol and v, i are constants denoting a vertex v and color i, respectively, assert that vertex v is assigned a color i. Atom of the form $vtx(v)$ 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 a constant i represents a color. A program with variables presented in the second column of the following chart

$$
\begin{array}{c|ll}
D_{(V,E)} & vtx(v). & (v \in V) \\
 & e(v,w). & (\{v,w\} \in E) \\
\hline
\Pi_{gc} & color(1). & \\
 & color(2). & \\
 & color(3). & \\
 & \{c(V,I)\} \leftarrow vtx(V),\ color(I). & \\
 & \leftarrow c(V,I),\ c(V,J),\ I<J,\ vtx(V),\ color(I),\ color(J). & \\
 & \leftarrow c(V,I),\ c(W,I),\ vtx(V),\ vtx(W),\ color(I),\ e(V,W). & \\
 & \leftarrow notc(V,1), notc(V,2), notc(V,3),\ vtx(V). &
\end{array}
\tag{11}
$$

encodes a solution to the graph coloring problem GC for an input graph (V,E). The facts in the first five lines of (11) form the DEFINE part of the program and encode the definition of an input graph as well as enumerate available colors. The rule

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

states that every vertex may be assigned some colors. This rule also forms the GENERATE part of the encoding. The last three rule form the TEST part, where the rule

$$\leftarrow c(V,I),\ c(V,J),\ I < J,\ vtx(V),\ color(I),\ color(J).$$

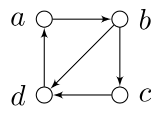
says that it is impossible that a vertex is assigned two colors; the rule

$$\leftarrow c(V,I),\ c(W,I),\ vtx(V),\ vtx(W),\ color(I),\ e(V,W).$$

says that it is impossible that any two adjacent vertexes are assigned the same color; and the last rule in (11) states that it is impossible that a vertex is not assigned a color.

We note that answer set programming provides a general purpose modeling language that supports elaboration tolerant solutions for search problems. We now follow the lines of [1] in defining 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 ASP, to solve a search problem P, we construct a program Π_P that captures problem's 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, program Π_{gc} in (11) is an example of such a uniform encoding for the 3-coloring problem GC, where any graph is an instance of this search problem. As a result, for any graph (V,E) answer sets of $\Pi_{gc} \cup D_{(V,E)}$ correspond to solutions of 3-coloring search problem to instance graph (V,E).

Consider a specific graph $\mathcal{G}_1$



Facts $D_{\mathcal{G}_1}$ representing $\mathcal{G}_1$ follow

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

One of the answer sets of program $\Pi_{gc} \cup D_{\mathcal{G}_1}$ follows

$$\begin{array}{l} \{vtx(a)\ vtx(b)\ vtx(c)\ vtx(d) \\ e(a,b)\ e(b,c)\ e(c,d)\ e(d,a)\ e(b,d) \\ color(1)\ color(2)\ color(3) \\ c(a,1)\ c(b,2)\ c(c,1)\ c(d,3)\}. \end{array} \qquad (12)$$

This answer set specifies that coloring vertex a and c with color 1, vertex b with color 2, and vertex d with color 3 is a 3-coloring of $\mathcal{G}_1$. If we add a constraint

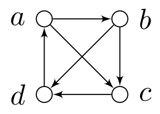
$$\leftarrow c(a, 1). \tag{13}$$

to $\Pi_{gc} \cup D_{\mathcal{G}_1}$, it forbids accepting solutions that assign color 1 to vertex a. Thus the set (12) is not an answer set of the resulting program.

Problem 14. *(3pt) (a) Write program $\Pi_{gc} \cup D_{\mathcal{G}_1}$ in the language of* CLINGO*. Run* CLINGO *on the resulting program instructing the system to enumerate all solutions. How many solutions to this instance of* 3*-coloring problem are there, or in other words, how many answer sets does* CLINGO *find?*

(b) Run your program with the constraint (13) *instructing the system to enumerate all solutions. How many answer sets do you have? Which* 3*-colorings do these answer sets encode.*

(c) Consider another graph, called $\mathcal{G}_2$,



How many 3*-colorings exist for this graph? Encode this graph as a set $D_{\mathcal{G}_2}$ of facts (similarly as we encoded graph $\mathcal{G}_1$ by set $D_{\mathcal{G}_1}$ of facts). Run* CLINGO *on program $\Pi_{gc} \cup D_{\mathcal{G}_2}$ (remember to replace symbol $\leftarrow$ by :-) instructing the system to enumerate all solutions. How many answer sets does* CLINGO *find?*

3 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 15. *(3pt) (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 16. *(7pt) (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 these notes follow the lecture notes on *Answer Sets; and Methodology of Answer Set Programming; course Answer set programming: CS395T, Spring 2005*[2] by Vladimir Lifschitz.

References

[1] Gerhard Brewka, Thomas Eiter, and Miros law Truszczynski. Answer set programming at a glance. Communications of the ACM, 54(12):92-103, 2011.

[2] Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable functions in ASP: theory and implementation. In Proceedings of International Conference on Logic Programming (ICLP), pages 407-424, 2008.

[2] `http://www.cs.utexas.edu/~vl/teaching/asp.html`