5. Methodology of Answer Set Programming

Recall that solving a problem using ASP means to write a logic program whose answer sets correspond to solutions, and then find solutions using an answer set solver. The basic approach to writing such programs consists in combining choice rules with constraints. In this handout we discuss a few examples of this "generate-and-test" strategy and its enhancement that involves "defined" atoms.

Generate and Test

Example: N Queens

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

The sets satisfying Conditions 1 and 2 can be described by rules similar to the program from Problem 4.8:

$$1 \leq \{q(1,j),\dots,q(n,j)\}^c \leq 1 \qquad (1 \leq j \leq n)$$

(exactly one queen on each column). These rules form the "generate" part of our program. The "test" part consists of the constraints expressing Condition 3

$$\leftarrow q(i,j), q(i,j') \qquad (1 \leq i,j,j' \leq n;\ j < j')$$

and Condition 4

$$\leftarrow q(i,j), q(i',j') \qquad (1 \le i, i'j, j' \le n;\; j < j';\; |i'-i| = j'-j).$$

Here is a representation of this program in the input language of GRINGO:

```
number(1..n).
#domain number(I;I1;J;J1).

1{q(K,J) : number(K)}1.

:- q(I,J), q(I,J1), J<J1.

:- q(I,J), q(I1,J1), J<J1, abs(I1-I)==J1-J.
```

The expression `number(I;I1;J;J1)` is the GRINGO abbreviation for the list

```
number(I), number(I1), number(J), number(J1).
```

5.1^e Use CLASP to find all solutions to the 8 queens problem that (a) have a queen at (1,1); (b) have no queens in the 4×4 square in the middle of the board.

5.2^e (a) Check how long it takes for CLASP to find one solution to the 16 queens problem using the program above. (b) The constraint corresponding to Condition 3 can be alternatively written using a cardinality expression:

$$\leftarrow 2 \le \{q(i,1), \dots, q(i,n)\} \qquad (1 \le i \le n).$$

Modify the program accordingly and check how this modification affects the computation time for 16 queens.

Example: Schur Numbers

We will show now how to use ASP to estimate the size of Schur numbers. A set A of integers is called *sum-free* if there are no numbers x, y in A such that $x + y$ is in A also (x and y do not need to be different). The *Schur number* $S(k)$ is the largest integer n for which the interval $\{1, \dots, n\}$ can be partitioned into k sum-free sets.[1]

[1] See `http://mathworld.wolfram.com/SchurNumber.html` .

For specific values of k, $S(k)$ can be computed (or estimated) using a program whose answer sets correspond to the partitions of $\{1,\dots,n\}$ into k (possibly empty) sum-free sets. We will use the atoms $a_i(x)$ ($1 \le i \le k$, $1 \le x \le n$) to express that x belongs to the i-th set A_i. The choice rules

$$1 \le \{a_1(x),\dots,a_k(x)\} \le 1 \qquad (1 \le x \le n)$$

describe "potential solutions"—partitions of $\{1,\dots,n\}$ into k sets. The constraints

$$\leftarrow a_i(x), a_i(y), a_i(x+y) \qquad (x, y \ge 1,\ x+y \le n,\ 1 \le i \le k)$$

eliminate the sets A_i that are not sum-free.

Here is how this program can be written in the input language of GRINGO:

```
number(1..n).
#domain number(X;Y).

subset(1..k).

1{a(I,X) : subset(I)}1.

:- a(I,X), a(I,Y), a(I,X+Y), subset(I), X+Y<=n.
```

5.3^e Use this program to find $S(3)$.

The number of answer sets for the ASP program above is greater than the number of partitions of $\{1,\dots,n\}$ into k sum-free sets, because the sets in any such partition can be arranged in a sequence $A_1,\dots,A_k$ in many possible ways. In particular, the subsets in a partition can be always ordered in such a way that

$$\begin{array}{l} 1 \in A_1, \\ 2 \in A_1 \cup A_2, \\ \dots \\ k \in A_1 \cup \cdots \cup A_k. \end{array}$$

Consequently, if we add to the program above a constraint expressing these conditions then the property of the program that we are interested in will be preserved—it will have an answer set iff $\{1,\dots,n\}$ can be partitioned into k sum-free sets. Adding this "symmetry breaking" constraint will actually improve the computation time of CLASP for many values of k and n, because, generally, CLASP is good at exploiting constraints for restricting search.

5.4^e Investigate the effect of this optimization on the computation time of CLASP for

(a) $k = 4$, $n = 44$.

(a) $k = 4$, $n = 45$.

(a) $k = 5$, $n = 130$.

Example: Graph Coloring

An *n-coloring* of a graph G is a function f from its set of vertices to $\{1, \ldots, n\}$ such that $f(x) \neq f(y)$ for every pair of adjacent vertices x, y. The answer sets of the following program are in a 1–1 correspondence with the n-colorings of G.

The following file `color` is a GRINGO encoding of the n coloring problem.

```
% File: color

c(1..n).
1 {color(X,I) : c(I)} 1 :- v(X).
:- color(X,I), color(Y,I), e(X,Y), c(I).
```

The following file `graph` encodes a graph instance.

```
% File: graph

v(0..7).

e(0,1).  e(1,2).  e(2,3).  e(3,0).
e(4,5).  e(5,6).  e(6,7).  e(7,4).
e(0,4).  e(1,5).  e(2,6).  e(3,7).
```

One can find all 2-colorings using the following command:

```
% gringo -c n=2 -d none graph color | clasp 0
```

Example: Cliques

A *clique* in a graph G is a set of pairwise adjacent vertices of G. The answer sets of the following program are in a 1–1 correspondence with cliques of cardinalities $\geq n$.

```
% File: clique

n {in(X) : v(X)}.
:- in(X), in(Y), v(X;Y), X!=Y, not e(X,Y), not e(Y,X).
```

Generate-Define-Test

In more complex uses of ASP, constraints in the test part of the program use auxiliary atoms that are defined in terms of the atoms occurring in the generate part. The definitions of the auxiliary atoms form then a third component of the program, besides the "generate" and "test" parts—its "define" part.

Consider, for instance, the problem of finding a Hamiltonian circuit in a given graph, that is, a closed path that visits every vertex of the graph exactly once. A Hamiltonian circuit in a graph G can be thought of as a subgraph C of G that has the same set V of vertices as G and satisfies two conditions:

1. Every vertex is adjacent in C to exactly two vertices.

2. Every vertex is reachable in C from some fixed vertex u_0.

The program below represents an edge $\{u,v\}$ of G by the atom $in(u,v)$, where $u < v$ ($<$ is a fixed total order on V). The presence of this atom in an answer set indicates that the edge $\{u,v\}$ is included in C. The generate part of the program consists of the rules

$$\{in(u,v)\}^c \tag{1}$$

for all edges $\{u,v\}$ of G ($u < v$). Thus any set of edges is a potential solution. To encode the two conditions that characterize Hamiltonian circuits, we introduce the auxiliary atoms $adj(u,v)$ ($u < v$) that represent adjacency in C. These atoms are "defined" in terms of $in(u,v)$ by the rules

$$adj(u,v), adj(v,u) \leftarrow in(u,v) \tag{2}$$

for all edges $\{u,v\}$ of G ($u < v$). Then Condition 1 above can be expressed by the constraints

$$\begin{array}{l} \leftarrow \{adj(u,v)\ :\ v \in V\} \leq 1 \\ \leftarrow 3 \leq \{adj(u,v)\ :\ v \in V\} \end{array} \tag{3}$$

for all $u \in V$. For Condition 2, we introduce the auxiliary atoms $r(u)$ that express the reachability of u from u_0. They are "defined recursively" by the rules

$$\begin{array}{l} r(u_0) \\ r(u) \leftarrow r(v), adj(u,v) \end{array} \tag{4}$$

for all edges $\{u, v\}$ of G. Using these atoms, we express Condition 2 by the constraints

$$\leftarrow not\ r(u) \tag{5}$$

for all $u \in V$.

Answer sets for program (1)–(5) are in a 1–1 correspondence with the Hamiltonian circuits in G. Rules (2) and (4) form the define part of the program, and rules (3) and (5) form the test part.

The following file `hc` is a GRINGO encoding of program (1)–(5):

```
{in(U,V)} :- edge(U,V).

adj(U,V) :- in(U,V), vertex(U;V).
adj(V,U) :- in(U,V), vertex(U;V).

:- {adj(U,V) : vertex(V)}1, vertex(U).
:- 3{adj(U,V) : vertex(V)}, vertex(U).

r(0).
r(U) :- r(V), adj(U,V), vertex(U;V).

:- not r(U), vertex(U).

#hide.
#show in/2.
```

The last two lines tell CLASP to display the `in` atoms only. They make the use of the option `-d none` to suppress information on the domain predicates redundant.

This file needs to be appended to a description of G in the form of a definition of the domain predicates `vertex` and `edge`. For instance, the file

```
vertex(u0;u1;u2;u3;v0;v1;v2;v3).
edge(u0,u1). edge(u1,u2). edge(u2,u3). edge(u3,u0).
edge(v0,v1). edge(v1,v2). edge(v2,v3). edge(v3,v0).
edge(u0,v0). edge(u1,v1). edge(u2,v2). edge(u3,v3).
```

describes the vertices and edges of a cube. Assuming that this file is called `cube`, CLASP can be instructed to find a Hamiltonian circuit by the command

```
    % gringo cube hc | clasp
```

Syntactically, rules in the generate and test parts of an ASP program are somewhat different from Prolog rules: Prolog has neither choice rules nor constraints. On the other hand, the define part of an ASP program does look similar to a Prolog program, and writing define rules in ASP is based on the same intuition as writing Prolog programs.

For "generate-and-test" programs, such as N queens and Schur numbers examples, proving correctness is usually not difficult: it requires no theory of answer sets beyond the basic properties of choice formulas and constraints discussed in the previous handout. But if a program contains definitions, as the Hamiltonian circuit example, then more complex mathematical tools are required for a correctness proof. We skip this discussion.

System F2LP

The input languages of answer set solvers allows a limited form of modern answer set programs. System F2LP is a front-end to GRINGO to allow modern answer set programs as input. The system is available at

`http://reasoning.eas.asu.edu/f2lp` .

The language of F2LP is a superset of the language of GRINGO. To encode a rule

$$F \leftarrow G$$

that is not accepted by GRINGO, the current version of F2LP requires us to encode it as a formula

$$G \rightarrow F.$$

It also allows quantifiers to be used, which are understood as multiple conjunctions or multiple disjunctions ranging over the set of object constants.

Formulas can be encoded in the language of F2LP using the following ASCII characters.

Symbol	$\neg$	$\wedge$	$\vee$	$\rightarrow$	$\bot$	$\top$	$\forall xyz$	$\exists xyz$
ASCII	`-`	`&`	`|`	`->`	`false`	`true`	`![X,Y,Z]:`	`?[X,Y,Z]:`

For instance,

$$p \leftarrow \neg\neg p$$

can be encoded in the language of F2LP as

```
        --p -> p.
```

Rule

$$r \leftarrow \exists x(p(x) \wedge q(x))$$

can be encoded as

```
     ?[X]: (p(X) & q(X)) -> r.
```

In view of the Problem 4.5 statement, F2LP can be used for computing models of propositional logic and Herbrand models of first-order formulas. For instance in order to compute the models of

$$(p_1 \wedge q_1) \vee (p_2 \wedge q_2) \vee (p_3 \wedge q_3).$$

we write the following file `f1`.

```
{p1}.
{q1}.
{p2}.
{q2}.
{p3}.
{q3}.

(p1 & q1) | (p2 & q2) | (p3 & q3).
```

One can find all 37 models of the formula using command

```
f2lp f1 | gringo | claspD 0
```

(`claspD` is an extension of `clasp` to handle disjunctive programs.)

5.5^e Use F2LP to find all models of

$$(\neg p \vee q) \wedge (\neg p \vee r) \wedge (q \vee r) \wedge (\neg q \vee \neg r)$$

5.6^e Consider the following statement.

If the train arrives late and there are no taxis at the station then John is late for his meeting. John is not late for his meeting. The train did arrive late.

It can be encoded as a set of formulas in propositional logic as follows:

$$\begin{gathered} \mathit{train\_late} \land \neg \mathit{taxi} \rightarrow \mathit{john\_late} \\ \neg \mathit{john\_late} \qquad\qquad l \\ \mathit{train\_late} \end{gathered}$$

Use F2LP to show that there was a taxi at the station. In other words, show that the set of formulas entails

$$\mathit{taxi}.$$

Programming Projects

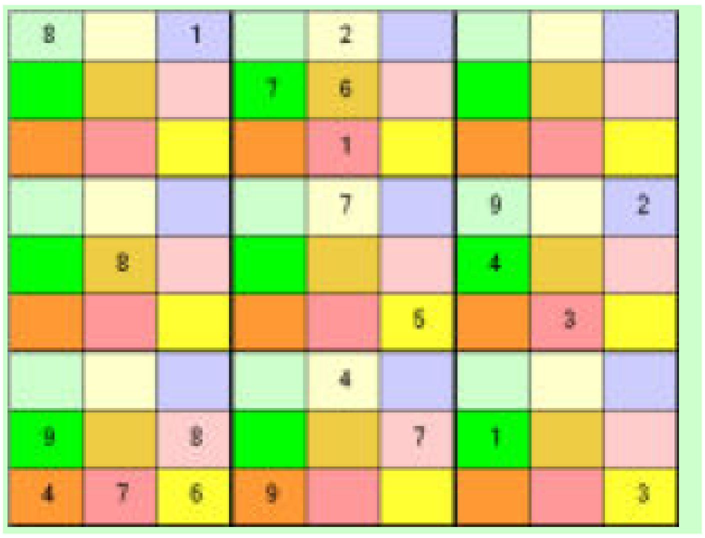
In each of the following exercises, show how to solve the given computational problem using CLASP.

5.7 Sudoku (a.k.a. Number Place) is a game to fill in the grid so that every row, every column, and every 3×3 box contains all the digits 1 through 9.

	6		1		4		5	
		8	3		5	6		
2								1
8			4		7			6
		6				3		
7			9		1			4
5								2
		7	2		6	9		
	4		5		8		7	

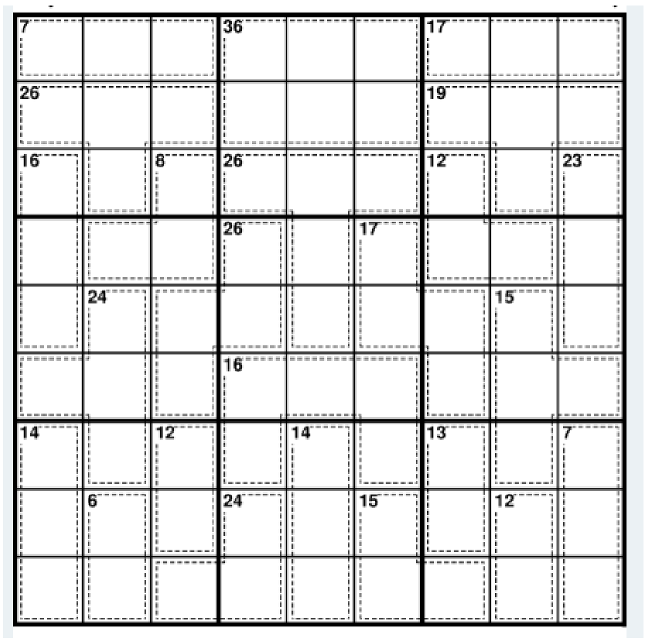
Problem: fill in the numbers.

5.8 Consider the following variant of Sudoku. The same position in each of 3×3 boxes form a "region," yielding 9 total regions (a region is represented by the same color below). In addition to the requirement of Sudoku, every region must contain all the digits 1 through 9.



Problem: fill in the numbers.

5.9 Consider the following variant of Sudoku. Each cage ("dotted area") is associated with a number. In addition to the requirement of Sudoku, the sum of the cells in a cage must be equal to the number given for the cage. Each digit in the cage must be unique.



Problem: fill in the numbers.

5.10 Each of four men owns a different species of exotic pet. Here is what we know about them:

1. Mr Engels (whose pet is named Sparky), Abner and Mr. Foster all belong to a club for owners of unusual pets.
2. The iguana isn't owned by either Chuck or Duane.
3. Neither the jackal nor the king cobra is owned by Mr. Foster.
4. The llama doesn't belong to Duane (whose pet is named Waggles).
5. Abner, who doesn't own the king cobra, isn't Mr. Gunter.
6. Bruce and Mr. Foster are neighbors.
7. Mr. Halevy is afraid of iguanas.

Problem: Find each man's full name and determine what kind of pet he owns.

5.11 You are organizing a large New Year's Eve party. There will be n tables in the room, with m chairs around each table. You need to select a table for each of the guests, who are assigned numbers from 1 to mn, so that two conditions are satisfied. First, some guests like each other and want to sit together; accordingly, you are given a set A of two-element subsets of $\{1, \dots, mn\}$, and, for every $\{i, j\}$ in A, guests i and j should be assigned the same table. Second, some guests dislike each other and want to sit at different tables; accordingly, you are given a set B of two-element subsets of $\{1, \dots, mn\}$, and, for every $\{i, j\}$ in B, guests i and j should be assigned different tables. Problem: find such a seating arrangement or determine that this is impossible.

5.12 You are in charge of assigning referees to the papers submitted to a conference. Each of the n submissions needs to be assigned to a few referees from among the m members of the Program Committee. The PC members have read the abstracts of all submissions, and each of them gave you the numbers of the submissions that he is qualified to referee; let A_i ($1 \leq i \leq m$) be the subset of $\{1, \dots, n\}$ given to you by the i-th committee member. You need to select, for each i, a set X_i of papers to be assigned to the i-th committee member so that three conditions are satisfied. First, each X_i should be a subset of A_i. Second, the cardinality of each X_i should be between a lower bound l and an upper bound u. Third, each paper should be assigned to exactly k referees. Problem: find such an assignment of papers to referees or determine that this is impossible.

5.13 By G_n we denote the $n \times n$ grid of unit squares (that is, the graph whose vertices are the points (x, y) with coordinates $x, y \in \{0, \ldots, n\}$, with two vertices considered adjacent if the distance between them is 1). A *wire routing problem* is determined by a list

$$\{v_1, v_1'\}, \ldots, \{v_k, v_k'\}$$

of disjoint pairs of vertices of G_n. A *solution* to a wire routing problem is a list of paths $\alpha_1, \ldots, \alpha_k$ in G_n such that

- α_i connects v_i with v_i' $(i = 1, \ldots, k)$, and
- no two paths on the list pass through a common vertex.

Problem: find a solution such that the length of each path in it does not exceed a given upper bound u, or determine that this is impossible.