A Generator for Random Non-Binary Finite Constraint Satisfaction Problems

Hui Zou, Berthe Y. Choueiry, and Amy M. Davis
Constraint Systems Laboratory
Department of Computer Science and Engineering
University of Nebraska-Lincoln
[hzou|choueiry|amydavis]@cse.unl.edu

Technical Report CONSYSTLAB-02-02

April 1, 2002

Abstract

The paper describes an implementation of a generator of random instances of non-binary constraint satisfaction problems that meets a given set of specifications. This is a continuation of the work we started in [1].

1 Description

The program is designed to generate random instances of Constraint Satisfaction Problems (CSPs) that meet a set of specified parameters, such as the number of variables, domain size, constraint density, tightness. At the same time, it can generate any combination of binary, ternary, and/or quaternary constraints specifies as percentage of the total number of constraints in the problem.

2 Assumptions

To realize this program, we make the following assumptions:

1. All variables have the same domains.
2. Any particular group of variables has only one constraint of a given arity.
3. All constraints have the same tightness.
4. All variables are equally likely to be connected by a constraint.
5. We guarantee that the resulting CSP is connected.

3 Parameters

The input parameters are the following:

- n: the number of variables
- a: domain size.

- p: constraint probability.
- p_2: the percentage of binary constraints.
- p_3: the percentage of ternary constraints.
- p_4: the percentage of quaternary constraints.
- t: tightness of a constraint, which is the ratio of the number of incompatible tuples over the number of all possible tuples.

Given the input parameters, we compute internally a number of other parameters that we use to generate the CSP.

- C is the total number of constraints, including binary, ternary and quaternary.
- c_2 is the number of binary constraints.
- c_3 is the number of ternary constraints
- c_4 is the number of quaternary constraints

The relations between C, c_2, c_3, and c_4 are as follows:

$$
\begin{aligned}
C &= c_2 + c_3 + c_4 && (1)\\
C &= \binom{n}{2} \cdot p_2 \cdot p + \binom{n}{3} \cdot p_3 \cdot p + \binom{n}{4} \cdot p_4 \cdot p && (2)\\
&= \frac{n!}{(n-2)! \times 2!} \cdot p_2 \cdot p + \frac{n!}{(n-3)! \times 3!} \cdot p_3 \cdot p + \frac{n!}{(n-4)! \times 4!} \cdot p_4 \cdot p \\
&= \frac{n(n-1)}{2} \cdot p_2 \cdot p + \frac{n(n-1)(n-2)}{6} \cdot p_3 \cdot p + \frac{n(n-1)(n-2)(n-3)}{24} \cdot p_4 \cdot p \\
c_2 &= p_2 \cdot C && (3)\\
c_3 &= p_3 \cdot C && (4)\\
c_4 &= p_4 \cdot C && (5)
\end{aligned}
$$

4 Structure of program

4.1 Constraint representation

In general, we represent a constraint as an array whose dimensions equal the arity of the constraint. For example, we represent a binary constraint by using a 2-dimension array, represent a ternary constraint by using a 3-dimension array and so on. Indeed this kind of representation is easy to understand. However, when the arity of a constraint is bigger than 3, we find the structure of a constraint is too complex to be implemented. Although this is doable, the use of such an n-dimensional array implemented in the language `C` is rather cumbersome. So we chose to use an alternative manner to represent a constraint through the use of a vector. The goal is to find a function that can express a multiple dimensions array into a one dimension array. We will discuss this method in detail as follows.

4.1.1 Basic Idea

Binary Constraint: We use a 2-dimension array $m[i, j]$,where $i = 1, 2, 3, \ldots, a$ and $j = 1, 2, 3, \ldots, a$ to represent a constraint connecting two variables V_i and V_j and convert it into a 1-dimension array (i.e., a vector) $v[$*retrieve*$(i, j)]$. We define a function *retrieve*(i, j) that maps the ith and jth position of two variables V_i and V_j into the corresponding position in the 2-dimension array.

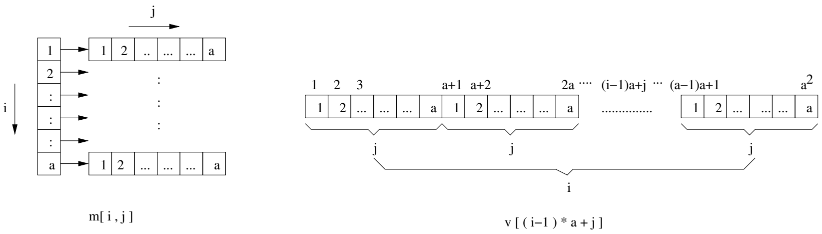

Figure 1: *2-dimension to 1-dimension.*

We find the following relation:

$$i \times j \Longrightarrow retrieve(i, j) = (i - 1) \cdot a + j \tag{6}$$

Ternary Constraint: We use a 3-dimension array $m[i, j, k]$, where $i = 1, 2, 3, \ldots, a$, $j = 1, 2, 3, \ldots, a$ and $k = 1, 2, 3, \ldots, a$ to represent a constraint connecting three variables V_i, V_j, and V_k, then we convert it into a 1-dimension array $v[$*retrieve*$(i, j, k)]$. Here *retrieve*(i, j, k) implies there exists a function forces on i, j and k.

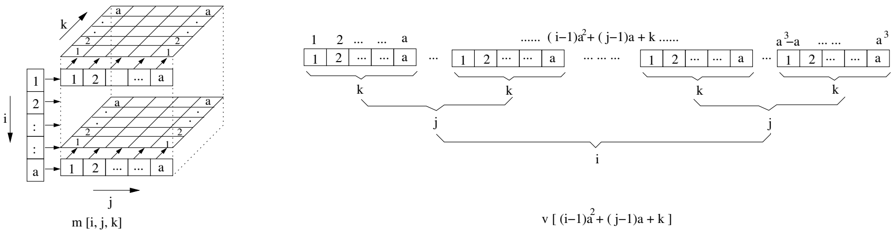

Figure 2: *3-dimension to 1-dimension.*

We find the following relation:

$$i \times j \times k \Longrightarrow retrieve(i, j, k) = (i - 1) \cdot a^2 + (j - 1) \times a + k \tag{7}$$

Quaternary Constraint: We use a 4-dimension array $m[i, j, k, l]$, where i =1, 2, 3, $\ldots, a$, j =1, 2, 3, $\ldots, a$, $k = 1$, 2, 3, $\ldots, a$, and l =1, 2, 3, $\ldots, a$ to represent a constraint connecting four variables V_i, V_j, V_k, and V_l, then we convert it into a 1-dimension array $v[$*retrieve*$(i, j, k, l)]$. Here *retrieve*(i, j, k, l) implies there exists a function forces on i, j, k and l. Because it is difficult to draw a 4-dimension array, we use a tree to express it.

We find the following relation:

$$i \times j \times k \times l \Longrightarrow retrieve(i, j, k, l) = (i - 1) \cdot a^3 + (j - 1) \cdot a^2 + (k - 1) \cdot a + l \tag{8}$$

General Constraint: From observing the above particular instances, we can find a uniform expression of the function by which we can use a 1-dimension array to represent any multiple dimension array. We call this expression as *constraint's conversion function* (`CCF`) and we define it as follows,

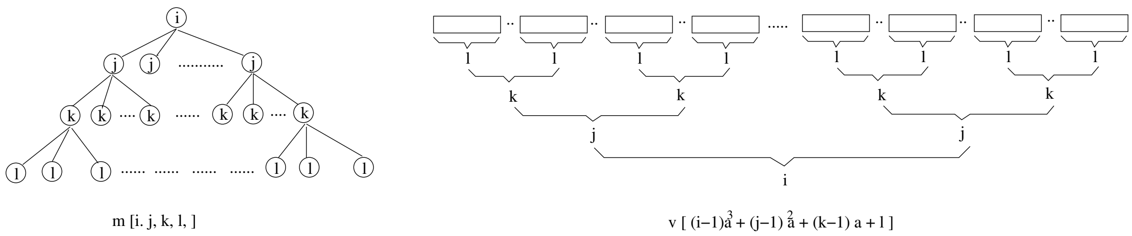

Figure 3: *4-dimension to 1-dimension.*

For a given array $m[i_1, i_2, i_3, \dots, i_\alpha]$, here α is the arity of the constraint, we have a 1-dimension array $v[\texttt{CCF}]$ responding to it.

$$\texttt{CCF} = (i_1 - 1) \cdot a^{\alpha-1} + (i_2 - 1) \cdot a^{\alpha-2} + \ldots + (i_{\alpha-1} - 1) \cdot a + (i_\alpha - 1) + 1 \tag{9}$$

here $i_1, i_2, \dots, i_\alpha = 1, 2, 3, \dots, a$

For example, we have a constraint with a=5, arity=3, then CCF$= (i_1-1)\cdot 5^2 + (i_2-1)\cdot 5 + (i_3-1)+1$, and i_1, i_2, i_3 in [1, 2, 3, 4, 5].

4.1.2 Constraint representation and retrieval

Based on the above statement, we use a 1-dimension array to represent a constraint with any arity. The entry of the array is 1 or 0. The entry is set to 1 when the tuple is compatible, otherwise it is set to 0. So the number of zero in a array is equal to the number of incompatible tuples. we have the following relation,

$$t = \frac{the\ number\ of\ zero}{a^\alpha} \tag{10}$$

$$the\ number\ of\ zero = t \times a^\alpha \tag{11}$$

For example, the array below represents the following constraint (with $a = 5, t = 0.32$), where the numbers represent the indices of the row and column $\{$(1,1), (1,2), (1,5), (2,1), (2,4), (2,5), (3,1), (3,2), (3,5), (4,1), (4,2), (4,5), (5,1), (5,2), (5,3), (5,4), (5,5)$\}$. This is the list of all compatible tuples.

1	1	0	0	1	1	0	0	1	1	1	1	0	0	1	1	1	0	0	1	1	1	1	1	1

v [i * j]

Figure 4: *example.*

How do we retrieve this array to find all compatible tuples that it represents after we establish such a 1-dimension array? The way is very easy just same as the way we retrieve a multiple dimension array. Please see the following examples, we want to pick up all tuples of a constraint with arity=3. We achieve it using for-loop structure as follows,

for $(i = 1; i <= a; i++)$
    for $(j = 1; j <= a; j++)$
        for $(k = 1; k <= a; k++)$
            if $(v[\texttt{CCF}] == 1)$
                (i, j, k) *is a tuple;*

Another advantage of this method is that a better distribution of 0s and 1s in the array can be obtained among the whole space of the array. However we can only control the distribution of 0s and 1s among a slot of the array by using multiple-dimension method. Thus we can generate more general instances by applying the alternative method mentioned above.

4.2 Processing

We have three main stages of the generation processing, such as preparation, generating and assignment stages. We describe them as follows,

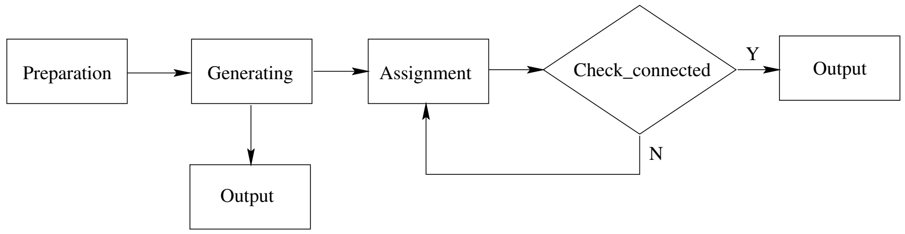

Figure 5: *Processing.*

Preparation stage: At this stage, the main task is to get the initial data and finish relative computations.

- `get data`: $n, a, p, p_2, p_3, p_4, t, outfile$
- `computation`: To compute C, c_2, c_3, c_4 and $the\ number\ of\ incompatible\ tuples$ according to Expressions (2), (3), (4),(5) and (11)

Generating stage: We generate constraints into 3 sets – binary set, ternary set and quaternary set by calling a function gen() in main program as follows,

gen(arity=2);

gen(arity=3);

gen(arity=4);

The function gen() is defined by the following way,

```
gen(){
    initialize array v[]; //set all entry to be 1
    set_ tightness(); //set the tightness with specified value
    retrieve() ; //pick up all compatible tuples and output
    }
```

Assignment stage: After we generate all constraints we need to distribute them to variables. Several issues we should focus on,

1. Constraints with different arity shall be assigned to an appropriate number of variables. That means a constraint with $arity = 2$ should be assigned to two variables. A constraint with $arity = 3$ should be assigned to three variables.
2. No two constraint of the same arity have exactly the same scope.

3. the constraint network should be connected. To guarantee the connectivity, we need to check if the constraint network is connected after each assignment stage. If it is, then output the generated result. Otherwise we will execute the assignment stage again utill a connected result is generated.

4.3 Format of input

The executable file is named `gen2`. It takes 6 command arguments as follows: $gen2\ n\ a\ p\ p_2\ p_3\ p_4\ outfile$. The parameters being the ones defined in Section 3.

4.4 Format of output

```
CSP-n-a-p-p2-p3-p4-t
1 a a
((1 1) (1 2) (1 3) ... (55))
2 a a a
((1 1 1) (1 1 3) ... (5 5 3))
.
.
.
C a a a a
((1 1 1 2) ( 1 1 2 3) ... (5 5 3 5))
```

References

[1] Hui Zou, Amy M. Beckwith, and Berthe Y. Choueiry. A Generator of Random Instances of Binary Finite Constraint Satisfaction Problems with Controllable Levels of Interchangeability. Working note, Department of Computer Science and Engineering, University of Nebraska-Lincoln, 2001. July, 2001.