2001

A Generator of Random Instances of Binary Finite Constraint Satisfaction Problems with Controllable Levels of Interchangeability

Hui Zou
University of Nebraska-Lincoln, hzou@cse.unl.edu

Amy Beckwith
University of Nebraska-Lincoln, abeckwit@cse.unl.edu

Berthe Y. Choueiry
University of Nebraska-Lincoln, choueiry@cse.unl.edu

A Generator of Random Instances of Binary Finite Constraint Satisfaction Problems with Controllable Levels of Interchangeability

Constraint Systems Laboratory
Department of Computer Science and Engineering
University of Nebraska-Lincoln

Technical Report, CONSYSTLAB-01-01

Author: Hui Zou, hzou@cse.unl.edu
Supervisor: Amy Beckwith
Director: Berthe Y. Choueiry

CONTENTS

Abstract

In order to test the performance of algorithms for solving Constraint Satisfaction Problems (CSPs), we must establish a large collection of CSP instances that meet a given set of specifications, such as the number of variables, domain size, constraint density, tightness, etc. The goal of this report is to describe a generator of instances that have a specified degree of interchangeability. An example of such a generator is described in (Freuder and Sabin 1997), which generates non-reflexive constraints and does not allow us to control concurrently the degree of interchangeability and tightness. We have developed a technique and written a program in the C language to generate CSP instances that satisfy the above two conditions at the same time. Our generator removes the restrictions of the generator of (Freuder and Sabin 1997) and yields more general constraints. This paper presents an overview of our technique and our implementation.

1 Introduction

We restrict ourselves to binary CSPs with finite domains. A generator of random CSP instances with a predefined level of interchangeability is presented in (Freuder and Sabin 1997). In this generator, each constraint is defined as the conjunction of two constraints: one that specifies the level of interchangeability and one that dictates the tightness value. The resulting constraint, which is obtained by taking the intersection of the definitions of the two constraints, is not guaranteed to respect the tightness or the degree of interchangeability specified. In this report, we describe a generator for random CSPs that allows us to generate instances that precisely satisfy these values.

This document is structured as follows. Section 2 lists the assumptions and introduces the parameters used in the generator. Section 3 explains the design rationale. Section 4 explains the logical structure of the program as a flow chart. Section 5 discusses the structure of the code and shows how to use it. Section 6 explains the format of the output files and Section 7 lists the shortcomings of our method. Section 8 makes a conclusion and prompts the future work.

2 Preliminaries

In this section, we first state the assumptions that we make for this random generator. Then we describe other design decisions, such as the input parameters and the internal representation of a constraint, and our methods for computing the level of interchangeability in the problem.

2.1 Assumptions

To realize this program, we use the following assumptions:

1. All variables have the same domain.
2. Any particular pair of variables has only one constraint.
3. A constraint is not necessarily symmetrical, unlike (Freuder and Sabin 1997).
4. Any two constraints have a priori distinct definitions.
5. All constraints have the same degree of interchangeability, defined as the number of equivalence classes in the domain of the variable according to a single constraint. Since constraints are not symmetrical, a given constraint is built to induce equivalence classes on one of the variables to which it applies. The

equivalence relation induced on the domain of the remaining variable is not controlled.

6. All constraints have the same tightness.
7. Any two variables are equally likely to be connected with a constraint.

2.2 Input parameters

We adopt these parameters for the input to the program:

- n : the number of variables
- a : domain size
- C : number of constraints
- `IDF:` degree of *induced domain fragmentation* – a measure of the level of interchangeability that indicates the number of equivalence classes in the domain of one of the variables.
- t: tightness of each constraint, the percentage of the number of incompatible tuples to that of all possible tuples between two variables connected by a constraint. For example, t = 0.25, 0.68 etc.

2.3 Constraint representation

We use a binary matrix to represent a constraint connecting two variables. The rows and columns of the matrix represent the values in the domain of each variable. An entry in the matrix is set to 0 when the corresponding values of the two variables are not allowed by the constraint. The entry is set to 1 when the combination is acceptable by the constraint. Each row in a matrix is implemented as a vector. Additionally, we use one vector per matrix to record the information about the matrix. For example, the matrix of Figure 1 represents a constraint where the numbers (1, 2, 3, 4, 5) represent the indices of the row or column as well as the values in the domain of each variable. For this example, let V_1 be the variable whose values index the rows, and V_2 the variable whose values index the columns. The set {(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)} is the list of allowed combinations of the two variables.

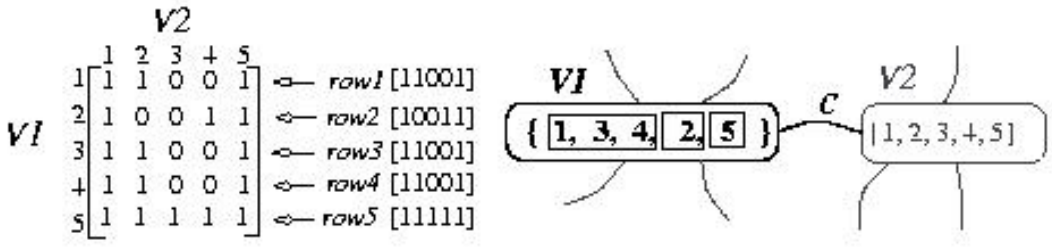


Figure 1: *Fragmentation of the domain of V_1 by the constraint C.*

This matrix is implemented as 5 vectors: *row1*, *row2*, *row3*, *row4*, and *row5*. Thus, the domain size of each variable is a=5. It is easy to check that its tightness is t=0.32. Further, one can check that some vectors are equal, i.e. they have the same definitions. For the CSP, this means that the values in the domain of the CSP variable used to index

the rows are interchangeable. In the above example, we can check that the constraint partitions the values of this variable into three equivalence classes, which correspond to indices (1, 3, 4), (2), (5). So, the constraint partitions the domain of the variable into three equivalence classes, and the degree of interchangeability of the variable with respect to this constraint is 3, `IDF`=3. We have `IDF`=3 by row. However, we note that the number of equivalence classes induced by the constraint of the domain of V_2 is 4. Our generator controls the degree of fragmentation only by rows, not by columns. So we are only controlling the interchangeability for one of the variables, not both.

For each matrix *A* representing a constraint, we use one additional vector to record information about these equivalence sets. The size of this vector is equal to the number of rows in the matrix, i.e. the size of the domain of the variable. Each equivalence set is indexed by a number, any two rows of *A* that are in the same equivalence set (i.e., their vectors are equal) are assigned the same index. This vector is called `IDF` and the one of the matrix *A* above is represented below:

$$\texttt{IDF-A}\ [1\ 2\ 1\ 1\ 3] \quad \begin{bmatrix} 1 \\ 2 \\ 1 \\ 1 \\ 3 \end{bmatrix} \begin{matrix} \leftarrow row1 \\ \leftarrow row2 \\ \leftarrow row3 \\ \leftarrow row4 \\ \leftarrow row5 \end{matrix}$$

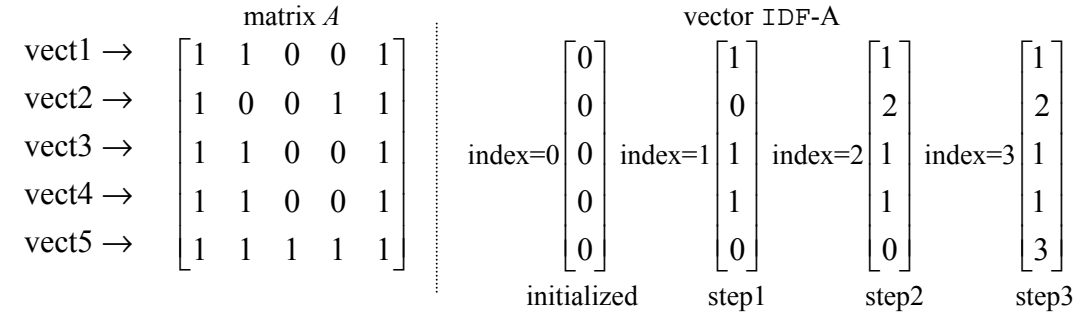
From this vector, we can quickly retrieve the number of equivalence classes (i.e. induced domain fragmentation, `IDF`) by taking the maximum of the entries in the vector. Also we can check if two rows correspond to equal vectors if the corresponding indices are equal. In the example above, it is easy to see that we have 3 equivalence classes. We can also check that rows 1, 3 and 4 belong to the same equivalence class, which means that the corresponding values are interchangeable according to this constraint.

2.4 How to compute the degree of interchangeability

In order to find the degree of interchangeability of a constraint, we use the vector named `IDF` defined in Section 2.3. First, each element of `IDF` is set to 0. We define a counter `index`=0. Starting from the first row,

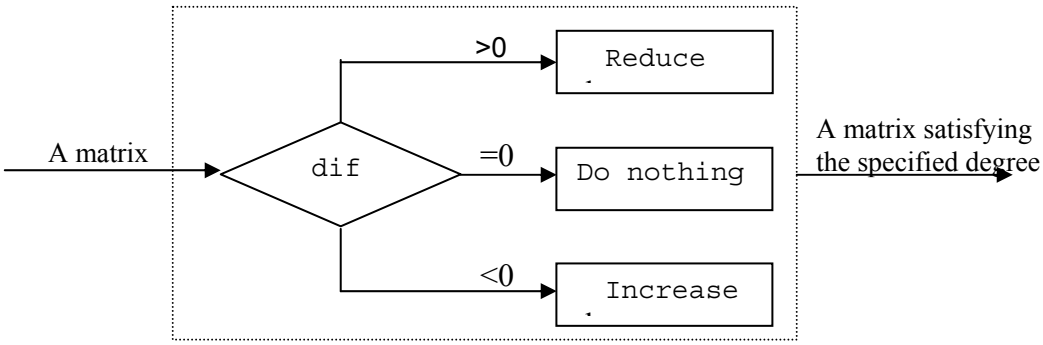
- If `IDF` of the current row `is` 0, then we increment `index` by 1, and set the values of the corresponding elements of `IDF` to `index`. Then we compare current row with all other rows below it.
- If they are equal and the corresponding elements of `IDF` is 0, we set the values of the corresponding elements of `IDF` equal to the `index`.
- We move to the next row in the matrix and do the same thing mentioned above.
- At last, we check `IDF` of the last row. If it still is 0, we increment `index` by 1 and set `IDF` to `index`.

After comparing all rows pair-wise, the value of `index` is the degree of interchangeability of the matrix. For example,



2.5 How to guarantee the degree of interchangeability

We design a subroutine to change the `IDF` of a matrix. It works according to the difference (`dif`) between current degree and specified degree. If the difference is equal 0, then the matrix has the specified degree of interchangeability. If it is bigger than 0, then we need to reduce the degree of interchangeability of current matrix, otherwise we need to increase it.



Suppose we specify `IDF`=2. The IDF of the matrix A defined above is equal to 3, thus `dif`=3-2>0. In order to satisfy the specification, we select any two vectors from two different equivalence classes and set one to be equal to the other. For example, we can set vect2←vect5. When we do so, we obtain a matrix with `IDF`=2. Note that this operation may affect the value of the constraint tightness.

Now, suppose we specify `IDF`=4 while the `IDF` of matrix *A* is equal 3, the `dif`=3-4<0. In this case, we need to select any vector from any equivalence class that has more than one element and modify it to make it different from the other elements in its class. This can be done as follows. We choose two random entries in the vector. If they have different values (one of them is 1, another one is 0) then we swap them. Otherwise, the value of one of them is changed. (The first operation does not affect the value of the constraint tightness, while the second does.) In the example above, we select vect3 and set it to be [1 0 1 0 1], thus yielding a matrix A with `IDF`=4.

3 Design principles

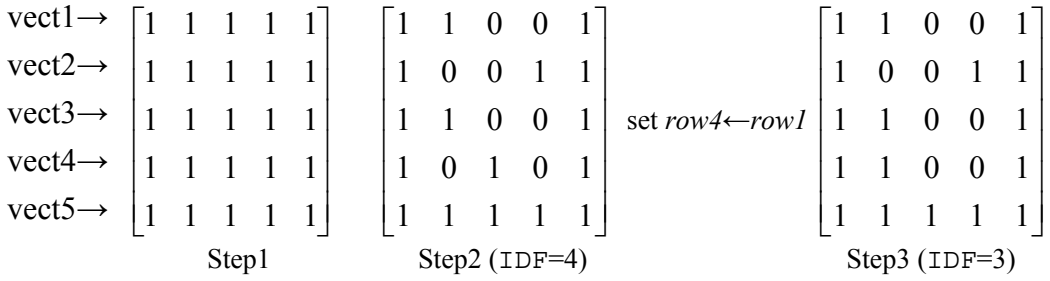
Based on these data structures and assumptions, we now describe to the implementation of the Random CSP generator.

3.1 Constraint generation

It is difficult to generate a constraint that satisfies the degree of interchangeability and tightness concurrently, because these two specifications affect each other. To simplify this problem, we generate a constraint that satisfies the two specifications by the following steps:

- Step1: Create a matrix with the specified domain size *a*.
- Step2: Set the entries in the matrix to realize the given constraint tightness *t*.
- Step3: Modify the matrix to comply with the specified interchangeability degree `IDF`.
- Step4: Check whether the tightness of the resulting matrix is equal to the specified tightness. If yes, then we apply a permutation operation (see Section 3.2) to the constraint obtained so far, store the resulting constraint, and exit this step with a success and continue to generate the next constraint. Otherwise, we just give it up and initialize the matrix, then return to step2 to generate a new constraint.

For example, the steps for generating a constraint with *a*=5, `IDF`=3, *t*=0.32 are illustrated below:



At step3, it is possible that we fail to define a constraint that satisfies the specifications. This may happen when:

1. No solution exists: for example for *a*=5, `IDF`=3, *t*=0.04. When *a*=5 and *t*=0.04, it is easy to check that there only exists solutions with `IDF`=2.
2. Although a solution may exist, the process of modifying interchangeability in the matrix continuously changes tightness.

To avoid this kind of problems, we use a `counter` at the beginning of the process to generate a constraint. When `counter` <50, we generate a constraint with the tightness *t* requested. However, when 50⩽`counter` <100, we instead generate a constraint with *t*=1- *t*. Although we have modified the *t* requested, we still obtain a constraint that meets the specifications through a simple processes `trans()`. The only work of `trans()` is to change 0 to 1 and 1 to 0 of the matrix. By this method, we increase the success rate to generate a CSP. The loop is terminated after trying to get solution 100 times and the generation of the current CSP instance is interrupted, regardless of the number of constraints generated so far.

3.2 Row permutation

To increase our chances of generating random constraints, each constraint that is successfully generated is run through a row permutation process that swaps two randomly

chosen rows a randomly chosen number of times. The input and output matrices of this process are guaranteed to have the same tightness and interchangeability degree: the process does not change the characteristics of the matrix.

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \quad \text{by permuting, it may become} \quad \begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \end{bmatrix}$$

3.3 Constraint assignment and connected CSP

After generating C constraints, we assign each of them to a random pair of variables. This simple procedure is described below.

The total number of combinations of n variables is $\binom{n}{2} = \frac{n(n-1)}{2}$. Of these $n(n\text{-}1)/2$ combinations, we choose C random pairs of variables. The resulting CSP is not guaranteed to be connected. A graph is connected when its number of edges |E| is at least equal to |V|-1, where |V| is the number of nodes in the graph. With respect to the CSP, we have the following implication:

CSP is connected → $(n\text{-}1) \leqslant C \leqslant n(n\text{-}1)/2$.

$C<(n\text{-}1)$ → CSP is not connected

$C> n(n\text{-}1)/2$ → impossible, there are more than one constraint between two nodes.

Therefore, when $C<(n\text{-}1)$, it is impossible to have a connected CSP. When we refer to a connected CSP, we only consider the CSPs with $(n\text{-}1) \leqslant C \leqslant n(n\text{-}1)/2$.

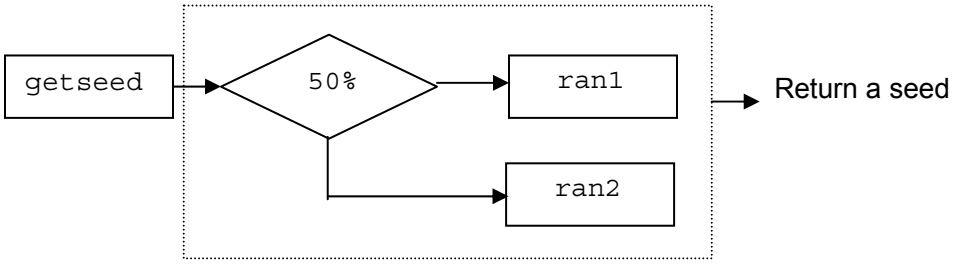
In order to guarantee that the CSP we generate is connected, first we check every CSP that we generate to see if it is connected, if it is, we store the CSP to the output file. If it is not connected, we throw away that CSP and begin again. Below we describe the algorithm to test if a CSP is connected.

1. *Set all variables to be* unvisited;
2. *Choose the first variable* v1*, mark it as* visited;
3. `enqueue(v1);`
4. *while !* `QueueIsEmpty()`
   V=`Dequeue();`
   For each variable that is connected to V *by a constraint*
   If the variable is visited*, then ignore it*;
   otherwise `enqueue(V);`
   mark it as visited;
5. *Check all the variables in the CSP*
   if any variable is unvisited*, then the CSP is disconnected.*
   otherwise, it is connected.

3.4 Generation and use of random numbers

To generate a random CSP instance, we need to generate random numbers. We use two methods to generate a random number. These are as follows:

1. **Method 1:** we set the seed to be the quotient of the id number of the current process and the system time.
2. **Method 2:** we randomly use one of two generators: `ran1` and `ran2`. In order to choose which generator to use, we use the function `getseed`, which uses a system function to generate randomly a number between 0 and 1. Depending on the outcome of `getseed`, we choose either `ran1` or `ran2`. As illustrated in the chart below:



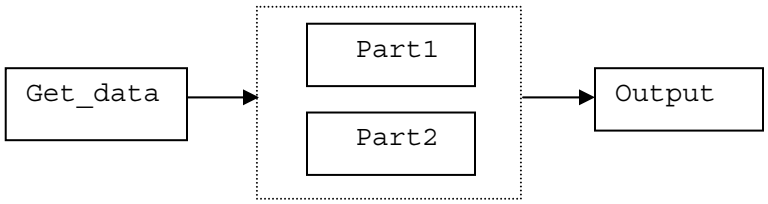
When we enter the program, we use the first method. After that, we use only the second method. The random numbers are used in four cases:

1. Used in `Reduce` (Section2.5) to select any two equivalence classes and select a random vector from the two equivalence classes respectively,
2. Used in `Increase` (Section2.5) to select a equivalence classes and for this given class, to select two elements of the class to be modified by the constraint generation procedure,
3. The number of times the permutation operator is applied in the `permute` function (Section3.2), and
4. The two rows to be swapped within the `permute` function (Section3.2).

4 The structure of our program

Below we describe the program for generating random CSP based on the components described above.

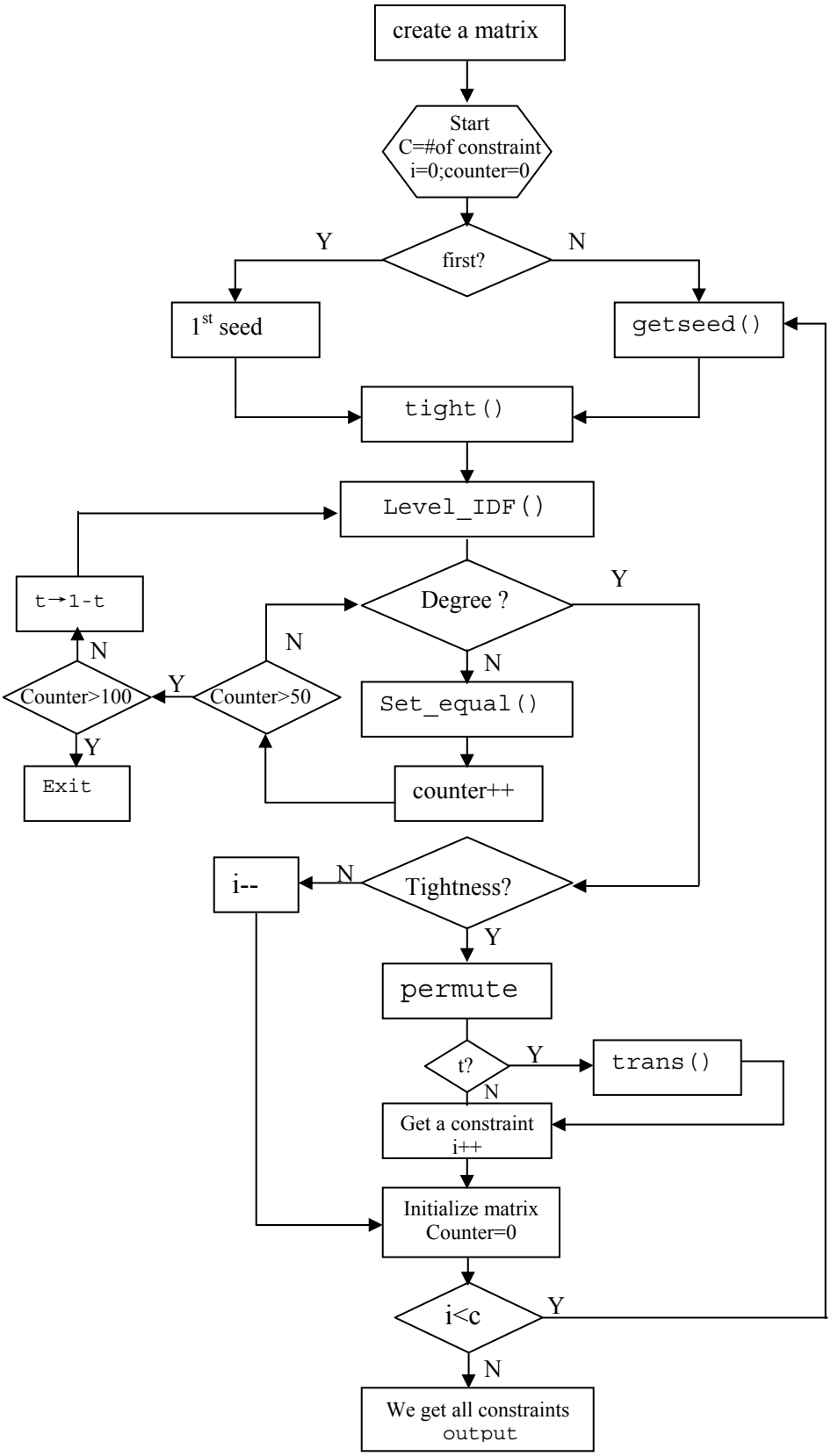
4.1 Main program



The main program has the 3 main components:

1. `Get_data` -- take input data from command line
2. The work will be divided into two main parts.
   `Part1`: Generates *C* distinct constraints.
   `Part2`: Distributes these *C* distinct constraints to any two particular variables.
3. `Output:` Store the result to a file.

4.2 Part1: Generates *C* distinct constraints



1. Create a matrix. *C* is the total number of constraints to generate.
2. The first seed is obtained by the id number of process and by time of system. With using this method, we get a different random sequence each time.
3. Within inner loop, we generate the random seed by `ran1()` and `ran2()` so that we could produce a more general CSP.
4. The function `tight()` will modify the matrix according to a specified tightness.
5. The function level_`IDF()` is to compute the interchangeability present.
6. Verify if the current `IDF` is equal to the specified `IDF`. If yes, then we have a solution that satisfies the specified degree. If not, we continue.
7. According to the difference of current degree and specified degree, we increase and reduce the degree by function `Set_equal()`.
8. We increase counter by 1 each time when we use `Set_equal()`. If `counter`>50 and<100, we will change this problem into its reversed form such that t=1-t then solve it. When `counter`>100, we will terminate the processing.
9. After processing, we test to see if the tightness, t, still meets the specification. If no, we give up this constraint and generate a new one.
10. If yes, we permute the current matrix to get a more general solution.
11. If the matrix generated was made with t=1-t, we transform it by `trans()`.
12. Initialize the matrix, then continue to generate a new constraint if $i< C$
13. If $i=C$, we have C constraints and are finished. Store the result into a file.

5 About the program

This program is implemented as several parts. Below we give a brief description of these parts, and how to use the program.

5.1 Components

There are total 7 files as follows:

1. `doit` – the shell script
2. `vector.h` – the head file
3. `vector.cpp` – the implementation for vector class
4. `Matrix.cpp` – the implementation for matrix class
5. `gen.cpp` – the main program for generating problems individually
6. `gen_batch.cpp` - the main program for generating problems in batch
7. `Makefile`- the make file.

This program was compiled and run under `cse.unl.edu`. It has not been tested on any other machine.

5.2 Usage

The code can be run in two different modes: to generate a batch of instances or to generate an individual instance.

5.2.1 Generating problems in batch

We wrote a shell script to generate large number of instances. All specifications and the number of instances that we generate for each problem are defined in this shell script file. After executing this script one time, all generated CSPs are stored into corresponding files. A file named `fail_record` records the file names of instances that could not be generated by the program.

We initialize a set of specifications in the `doit` file then execute `doit`. The system will generate problem automatically. The contents of `doit` are as follows:

```
R=0
n=10
a=5
NUM=20
for C in 5 9 13 18 22 27 31 36 40 45
 do
  for IDF in 1 2 3 4 5
   do
    for t in  0.04 0.12 0.20 0.28 0.36 0.44 0.52 0.60 0.68 0.76 0.84 0.92
     do
      i=1
      while [ $i -le NUM ]
      do
        i=`expr $i + 1`
        R=`expr $R + 1`
        gen_batch $n $a $C $IDF  $t $R irand-$C-$I-$T.$R
      done
      R=0
     done
   done
 done
```

NUM - the number of instances of each problem, which determine the total problems generated by the program.

We define different CSPs by changing the value of n, a, C, `IDF`, t, NUM. From above code we know it could generate at most 10*5*12*20=12,000 problems (in this example). The result will be stored into an output file named "irand-C-`IDF`-t.R" C, `IDF` and t are respectively corresponding to their actual value, and R is the series number.

5.2.2 Generating problems individually

This program takes 6 command arguments as follows: `gen n a C IDF t outfile`, where `outfile` is the filename where results are stored.

5.3 Appearance

During the running of this program, "." or " /" will be displayed on the screen to show the status of the processing:

- "." indicates a successful generation.
- "/" indicates a failed generation.

6 Output file format

- Line 1. A string (no white space characters) specifying the name of the problem (for the log file). Such that CSP-n-a-C-`IDF`-t, for example CSP-10-5-13-3-0.68
- Line 2. n a -1 (all integers).
- The next entries specify the constraints.
  1. First specify the number of different constraints C followed by each of the constraints in the format

```
C
const # sizei sizej
0 1 ...
1 0 ...
const # sizei sizej
0 1  ...
1 0  ...
```

  Where C is the number of constraints. Each constraint is specified by first giving three integers: the constraint number (each must be distinct and numbered in the range 1 to C), the domain size of the first variable and the domain size of the second variable.

  2. The next set of entries specifies the rows of the constraint matrix, (thus we have `sizei` "rows" with `sizej` entries of 0 and 1 each). These entries must all be 0 or 1.

- After each of the constraints is specified we specify the constraints that hold between particular variables. This is specified by a sequence of triples in the format:

```
i1 j1 h1
...
il jl hl
...
im jm hm
```

In Appendix 2, we provide an example of output file.

7 Shortcomings of this program

1. We do not guarantee that any two generated constraints are distinct, because instances are generated independently. This problem can be fixed, but this would be too costly and require lots of time to compare and reject or accept.
2. At step3 mentioned in section 3.1, we don't distinguish the two cases that cause CSP generation to fail. We solve that problem simply by a `counter`. This method is time consuming, because we mat have to abandon near solutions and begin a new generation from scratsch.

8 Conclusions and future work

- We used this program to generate a large batch of CSPs, as given in the example in Section 5.2.1. Our success rate was 97% under batch mode. Some problems that failed when run in batch mode succeeded when run individually (see Section 5.2.2).
- The performance of this program mainly depends on the distribution of random numbers. The performance is better when it is used to generate CSPs individually than when it is used CSPs in batch. This is because the system gives a better distribution of random numbers when the process differs every time.
- Since we don't distinguish the cases that lead to a failed generation (Section7), this method increases the times of failure generation. This problem can be addressed as follows:
  1. Before generating a CSP, we first check whether an instance with the specified conditions(C, `IDF`, t) can at all be generated. If this is not possible, then we don't need to proceed for this case and just exclude it.
  2. If there exist solutions to that problem, then we need to improve our algorithm to effectively generate the solution instead of giving up after a counter has reaches a threshold value, as we currently do.

9 Appendix 1: Source Code

1. gen.cpp

```
//A CSP Problem Generator (Ver1.03)
//-----------------------------------------------------------------------------------------
// Name : Hui Zou
// Date : June 28, 2001
// Description: This program is a CSP problem generator. It takes 6 command arguments as
  follows,
//            gen n a C IDF t outfile
//              n - the number of variables
//              a - domain size
//              C - distinct constraints number
//              IDF- the degree of interchangeability
//              t - tightness of each constrain,that is the percentage of zero
//              outfile - filename you want to store the results
//              we assume all values of above parameters ,given by user,are appropriate.
//              A detailed description will be provided with a readme file.
//-----------------------------------------------------------------------------------------
#include "vector.h"
//-----------------------------------------------------------------------------------------

int main(int argc, char *argv[]) {
   int n,a,C,IDF;float t, seed;
   int deg, first_time=0,fail=0;
   if (argc !=7 )
      usage();

   //reads data from character strings
   sscanf(argv[1],"%d",&n);
   sscanf(argv[2],"%d",&a);
   sscanf(argv[3],"%d",&C);
   sscanf(argv[4],"%d",&IDF);
   sscanf(argv[5],"%f",&t);

   //if the degree of interchangeability=1, then the number of zero should be multiple of a(domain
  size)
   //Otherwise, it is impossible to get solution at this case
   if ((int(a*a*t)%a !=0) && (IDF==1))
      exit(0);

   //open output fule
   ofstream fout;
   fout.open(argv[6]);
   if (fout.fail()) {
      cout << "Output file opening failed.\n";
      exit(1);
   }
   fout<<"CSP-"<<n<<"-"<<a<<"-"<<C<<"-"<<IDF<<"-"<<t<<endl;
```

```
fout<<n<<" "<<a<<" "<<"-1"<<"\n\n";
fout<<C<<endl;

//define a matrix class object
matrix m(a);

//generate the first seed
time_t T;
T=time(&T);
pid_t pid;
pid=getpid();
unsigned seed1;
seed1=T/pid;

//generate constraints
float T_temp=t;
for (int i=0;i<C;i++) {
   if (first_time==0) {
      seed=seed1;
      first_time=1;
   }
   else
      seed=getseed();//seed will be generated by random generators after first time
   srandom(seed);
   vector degree(m.getrow(),0);
   m.tight(t,seed);
   deg=m.Degree_Inter(degree);
   bool done=false;
   while (deg!=IDF){
      fail++;
  if (fail>=50 && !done) {
         t=1-t;
         m.tight(t,seed);
         done=true;
      }
      m.set_equal(deg,IDF,degree);
      degree.ini(m.getrow());
      deg=m.Degree_Inter(degree);
      if (fail>=100) {
     //after try it 50 times, if still no solution, then give up
         cout<<"Failure, please try again\n";
       //remove the outfile
     unlink(argv[6]);
     exit(0);
  }
   }

   int flag=0;
   int tmp=m.tightness();
```

```
        // give up the matrix whose tightness does not satisfy the specification
        if( tmp != int(a*a*t) ){
            flag=1;
            i--;
        }
        //select the constraint that satisfies both tightness& the degree of interchangeability
        if (flag != 1) {
       //in order to get more general problem, we permute the matrix
            m.permute(a);
       if (t != T_temp ) {
           m.need_change();
           t=T_temp;
           }
            fout<<i+1<<" "<<a<<" "<<a<<endl;
            fout<<m<<endl;
            fail=0;
        }
   //put the matix into original status
       m.initialization();
    }//end of for

    //assign constraints to any two variables
    assign_CS(fout,C,n);
    cout<<"Success!\n";
    fout.close();
    return 0;
}
```

2. gen_batch.cpp

```
//A CSP Problem Generator in batch (Ver1.02)
//-------------------------------------------------------------------------------------------
// Name : Hui Zou
// Date : May 29, 2001
// Description: This program is a CSP problem generator used for generating CSPs in batch.
//              It takes 7 command arguments as follows,
//              gen_batch n a C IDF t R outfile
//                n - the number of variables
//                a - domain size
//                C - distinct constraints number
//                IDF - the degree of interchangeability
//                t - tightness of each constrain,that is the percentage of zero
//                R - the series number of instance
//                outfile - filename you want to store the results
//                we assume all values of above parameters ,given by user,are appropriate.
//                A detailed description will be provided with a readme file.
//                After a succefull generating, "." is displayed. otherwise,
//                "/" will be displayed.
//-------------------------------------------------------------------------------------------
#include "vector.h"
//-------------------------------------------------------------------------------------------
```

```
int main(int argc, char *argv[]) {
   int n,a,C,IDF;float t, fn,seed;
   int deg, first_time=0,fail=0;
   if (argc !=8 )
      usage();

   //reads data from character strings
   sscanf(argv[1],"%d",&n);
   sscanf(argv[2],"%d",&a);
   sscanf(argv[3],"%d",&C);
   sscanf(argv[4],"%d",&IDF);
   sscanf(argv[5],"%f",&t);
   sscanf(argv[6],"%f",&fn);

   //if the degree of interchangeability=1, then the number of zero should be multiple of a(domain
  size)
   //Otherwise, it is impossible to get solution at this case
   if ((int(a*a*t)%a !=0) && (IDF==1))
      exit(0);

   //open output fule
   ofstream fout;
   ofstream fout1;
   fout.open(argv[7]);
   if (fout.fail()) {
      cout << "Output file opening failed.\n";
      exit(1);
   }
   fout<<"CSP-"<<n<<"-"<<a<<"-"<<C<<"-"<<IDF<<"-"<<t<<endl;
   fout<<n<<" "<<a<<" "<<-1"<<"\n\n";
   fout<<C<<endl;

   //define a matrix class object
   matrix m(a);

   //generate the first seed
   time_t T;
   T=time(&T);
   pid_t pid;
   pid=getpid();
   unsigned seed1;
   seed1=T/pid;

   //generate constraints
   for (int i=0;i<C;i++) {
      if (first_time==0) {
         seed=seed1;
         first_time=1;
```

```
        }
        else
           seed=getseed();//seed will be generated by random generators after first time
        srandom(seed);

        vector degree(m.getrow(),0);
        m.tight(t,seed);
        deg=m.Degree_Inter(degree);
        while (deg!=IDF){
           m.set_equal(deg,IDF,degree);
           degree.ini(m.getrow());
           deg=m.Degree_Inter(degree);
       fail++;
       if (fail>=50) {
          //after try it 50 times, if still no solution, then give up
              cout<<"/";
           //remove the outfile
          unlink(argv[7]);
              fout1.open("fail_record",ios::app);
              fout1<<"irand-"<<C<<"-"<<IDF<<"-"<<t<<"."<<fn<<endl;
              fout1.close();
          exit(0);
           }
        }
        int flag=0;
        int tmp=m.tightness();
        // give up the matrix whose tightness does not satisfy the specification
        if( tmp != int(a*a*t) ){
           flag=1;
           i--;
        }
        //select the constraint that satisfies both tightness& the degree of interchangeability
        if (flag != 1) {
       //in order to get more general problem, we permute the matrix
           m.permute(a);
           fout<<i+1<<" "<<a<<" "<<a<<endl;
           fout<<m<<endl;
           fail=0;
        }
     //put the matix into original status
        m.initialization();
     }//end of for

     //assign constraints to any two variables
     assign_CS(fout,C,n);
     cout<<".";
     fout.close();
     return 0;
}
```

3.vector.h

```
//-------------------------------------------------------------------------------------------
// Name : Hui Zou
// Date: June 27, 2001
// Description: This is the header file of vector ,matrix and Queue class.
// ver 1.02
//-------------------------------------------------------------------------------------------
#include <iostream.h>
#include <math.h>
#include <iomanip.h>
#include <stdlib.h>
#include <fstream.h>
#include <sys/types.h>
#include <unistd.h>
#include <time.h>
#include <float.h>
#include <stdio.h>
#include <signal.h>
//-------------------------------------------------------------------------------------------
//a structure CS_vars to record each constraint assigned to any two varibles
struct CS_vars {
   int var1;
   int var2;
};

class matrix;
//-------------------------------------------------------------------------------------------
//vector class prototype
//-------------------------------------------------------------------------------------------
class vector {
friend class matrix;
public:
   // constructor
   //to initialize each element to be zero
   //Precondition: the size of vector shall be given.
   //Postcondition: each element in vector is initialized to be 1.
   vector(int);

   //constructor
   //to initialize each element to be the specified value
   //Precondition:the size of vector and the specified value shall be given
   //Postcondition: each element in vector is initialized to be the specified
   //             value
   vector(int,int);

   //constructor
   // to initialize vector by a given array
   //Precondition: an given array and the size of vector is avalable
```

```
    //Postcondition:the vector copys each element of the given array
    vector(const vector*,int);

    //copy constructor
    //make a copy of an object
    //Precondition:an object of class existing
    //Postcondition: the vector is copied
    vector(const vector&);

    //distructor
    //deallocate the memory used by an object
    //Precondition: none
    //Postcondition:memory is free
    ~vector();

    //assign =operator
    //assign an object to another
    //Precondition: an object existing
    //Postcondition:the object is assigned with the given object
    vector& operator= (const vector&);

    //set each element of the vector to be 0
    //Precondition:the size of vector shall be given
    //Postcondition:each element=0
    void ini(int);

    //the [] opreator
    //return the value of element indexed by int
    int& operator[](int);

    //return the size of a vector
    int getsize();

    //return the value of one element of an object
    //Precondition: the index of an existing object given
    //Postcondition:return the value of the element pointed by the index
    int getvalue(int);

    //to swap any two elements in vector
    void swap(int,int);


    //the output operator
    friend ostream& operator<<(ostream&,const vector&);


private:
    int size;
    int *vec;
```

```
};

//-----------------------------------------------------------------------------------------
//matrix class prototype
//-----------------------------------------------------------------------------------------
class matrix {
public:
   //contructor
   //Precondition : two integers should be given
   //Postcondition: a matrix size of integer1*interger2 will be defined
   matrix(int,int);

   //contructor
   //Precondition : an integer should be given
   //Postcondition: a matrix size of integer*interger will be defined
   matrix(int);

   //copy constructor
   //Precondition :a marix class object is existing
   //Postcondition:to make a copy of the existing object
   matrix(const matrix&);

   //destructor
   //Precondition : none
   //Postcondition: deallocate memory
   ~matrix();

   //[] operator
   //return a complete row vector
   vector& operator[](int);

   //assign operator
   //Precondition :a marix class object is existing
   //Postcondition:to assign the existing object to another object
   matrix& operator=(const matrix&);

   //to test if two vetors(such that two rows) are equal
   //Precondition: two vector class objects is existing
   //Postcondition:if equal,then return 1,otherwise 0
   bool Isequal(const vector&,const vector&);

   //to calculate the tightness of a matrix
   //Precondition :a matrix class object is existing
   //Postcondition:return the number of zero in matrix
   int tightness();

   //to calculate the degree of interchangeaqbility of a matrix
   //return the degree
```

```
    int Degree_Inter(vector&);

    //according to the tightness, to fix the matrix
    void tight(double,unsigned);

    //according to the degree of interchangeability, to fix a matrix
    void set_equal(int,int,vector);

    //set the matrix to its orignal status
    void initialization();

    //return row
    int getsize();
    int getrow();

    //return col
    int getcol();

    //to swap any two vectors of the matrix
    void swap(int,int);

    //to exchange two elements of some vector of the matrix
    //the 1st numner is the row#, and the 2rd&3rd are the indexs of the elements of the row
    void change(int,int,int);

    void need_change();

    //to get a transpose matrix of the original matirx
    void transpose();

    //in order to get a more general solution and improve the rate of distinct constraints
    //after we get a solution, we permute this matrix in random
    void permute(int);

    //to test if two matrixs are equal
    //return 1 if equal, otherwise return 0;
    friend bool operator==(const matrix&,const matrix&);

    //the output operator
    friend ostream& operator<<(ostream&,const matrix&);
  private:
    int row;
    int col;
    vector **mat;
};

//-------------------------------------------------------------------------------------------
//Queue class prototype
//-------------------------------------------------------------------------------------------
```

```
class Queue;                // forward declaration
//---------------------------------------------------------------------
// Class for a node to be used in a linked list:creates and empty element
// Programming note: no implementation coding is needed.
// The queue is implemented as a linked list with two external pointers.
// One points to the front of the queue, another one points to the rear
//of the queue.
class QueueNode
{
   private:

   // Data members
   int element;         // Queue element
   QueueNode *next;     // Pointer to the next element

   friend class Queue;
};

//---------------------------------------------------------------------
class Queue
{
  public:

   Queue ();                           // Constructor
   // The constructor. creates an empty queue
   // Precondition : none
   // Postcondition: the variable front and rear has an appropriate
   //              value assigned.

   ~Queue ();                          // Destructor
   // The destructor.  Deallocated the memory assigned to pointers
   //Precondition: a Queue object has been created
   //Postcondition: Memory used by the queue is freed

   Queue(const Queue& TheQ);           // Copy constructor
   //Initializes Queue by making a copy of an existing Queue object
   //Precondition: object to be copied to is empty or can be emptied
   //             and object to be copied from exists
   //Postcondition: a copy of the object is made

   Queue& operator=(const Queue & TheQ);      // Assignment operator
   //Initializes Queue by making a copy of an existing Queue object
   //Precondition: object to be assigned to is empty or can be emptied
   //             and object to be assigned from exists
   //Postcondition: makes a copy of the Queue TheQ

   void enqueue (int newElement );          // Enqueue element
   //Adds a new item to the rear of the Queue
   //Precondition: A previously created Queue.  The parameter newElement
```

```
   //            is added to the rear of the Queue.
   //            There is memory available to add the new item in the Queue.
   //Postcondition:If the Queue is not full, the item is added to the rear
   //            of the Queue.  Otherwise, the Queue is unchanged.

   int dequeue ();                    // Dequeue element
   //Removes an item from the front of the Queue.
   //Precondition: A previously created Queue.
   //Postcondition: If the Queue is not empty, it has its front item removed
   //and returned. Otherwise (the Queue is empty) 0 is returned.

   void make_empty();                    // Clear queue
   //To make an existing Queue empty.
   //Precondition: A Queue object has been created.
   //Postcondition: The Queue is cleared.

   bool empty () const;                  // Queue is empty
   //Class instance tester
   //Precondition: A previously created Queue.
   //Postcondition: returns true if empty, false if not.


  private:

   // Data members
   QueueNode *front,              // Pointer to the front element
          *rear;              // Pointer to the rear element
};

//--------------------------------------------------------------------------------------
//other functions in main program
//--------------------------------------------------------------------------------------

//two random number generators to generate seed in random
float ran1();
float ran2();

//return seed in random with ran1() or ran2()
float getseed();

//assign constraints to any two variables
void assign_CS(ofstream&,int,int);

//to display the usage
void usage();
```

4.vector.cpp

```
//--------------------------------------------------------------------------------------
```

```
// Name : Hui Zou
// Date: May 8, 2001
// Description: This is the implementation of vector class.
// Ver 1.0
//-----------------------------------------------------------------------------------------
#include "vector.h"
//-----------------------------------------------------------------------------------------
vector::vector(int n) {
   size=n;
   vec=new int[size];
   for (int i=0; i<size;i++)
    vec[i]=1;
}

vector::vector(int n,int value){
   size=n;
   vec=new int[size];
   for (int i=0;i<size;i++)
       vec[i]=value;
}

void vector::ini(int n){
   size=n;
   vec=new int[size];
   for (int i=0; i<size;i++)
    vec[i]=0;
}


vector::vector(const vector *a, int n){
   size=n;
   vec=new int[size];
   for (int i=0;i<size;i++)
    vec[i]=a->vec[i];
}

vector::vector(const vector& vect) {
   size=vect.size;
   vec=new int[size];
   for(int i=0;i<size;i++)
    vec[i]=vect.vec[i];
}

vector::~vector() {
   delete vec;
}

vector& vector::operator=(const vector& vect) {
   if (size != vect.size) {
```

```
     delete vec;
     size=vect.size;
     vec=new int[size];
    }
    for (int i=0;i<size;i++)
        vec[i]=vect.vec[i];
    return *this;
}


int vector::getsize() {
    return (size);
}

int vector::getvalue(int index) {
    return vec[index];
}

int& vector::operator[](int i){
    return vec[i];
}

void vector::swap(int x,int y) {
    int temp=vec[x];
    vec[x]=vec[y];
    vec[y]=temp;
}

ostream& operator<<(ostream& out,const vector &vect) {
    for(int i=0;i<vect.size;i++)
        out<<vect.vec[i]<<" ";
    out<<endl;
    return out;
}
```

5. Marix.cpp

```
//-------------------------------------------------------------------------------------------
// Name : Hui Zou
// Date: June 27, 2001
// Description: This is the implementation file of matrix and Queue class. The header file is
//              in vector.h
// Ver 1.03
//-------------------------------------------------------------------------------------------
#include "vector.h"
//-------------------------------------------------------------------------------------------
//Implementation of matrix class
//-------------------------------------------------------------------------------------------
matrix::matrix(int n) {
```

```
    row=n;col=n;
    mat=new vector*[row];
    for (int i=0; i<row;i++)
     mat[i]=new vector(col);
}

void matrix::initialization() {
    for(int i=0;i<row;i++)
     for(int j=0;j<col;j++)
         mat[i]->vec[j]=1;
}

matrix::matrix(int r, int c){
    row=r;col=c;
    mat=new vector*[row];
    for (int i=0;i<row;i++)
     mat[i]=new vector(col);
}

matrix::matrix(const matrix& m) {
    row=m.row;col=m.col;
    mat=new vector*[row];
    for (int i=0;i<row;i++)
     mat[i]=new vector(col);
    for (i=0;i<row;i++)
     *mat[i]=*m.mat[i];

}

matrix::~matrix() {
    for (int i=row;i>0;i--)
     delete mat[i-1];
    delete mat;
}

matrix& matrix::operator=(const matrix& m) {
    if (m.row != row || m.col != col) {
     for (int i=row;i>0;i--)
           delete mat[i-1];
     row=m.row;col=m.col;
     mat=new vector*[row];
     for (i=0;i<row;i++)
         mat[i]=new vector(col);
    }//end of if
    for (int i=0;i<row;i++)
     *mat[i]=*m.mat[i];
    return *this;
}
```

```
vector& matrix::operator[](int i) {
   return *mat[i];
}

int matrix::getsize() {
   return row;
}

int matrix::getrow() {
   return row;
}

int matrix::getcol() {
   return col;
}

void matrix::swap(int i, int j) {
   vector *temp=mat[i];
   mat[i]=mat[j];
   mat[j]=temp;
}

void matrix::change(int num, int x, int y) {
   int temp;
   temp=mat[num]->vec[x];
   mat[num]->vec[x]=mat[num]->vec[y];
   mat[num]->vec[y]=temp;
}

bool matrix::Isequal(const vector &v1,const vector &v2) {
   for (int i=0;i<col;i++)
      if (v1.vec[i] != v2.vec[i])
       return 0;
   return 1;
}

void matrix::transpose() {
   int x=row; int y=col;
   matrix mt(y,x); //create a transpose matrix;
   for (int i=0;i<y;i++) {
    for(int j=0;j<x;j++)
       mt.mat[i]->vec[j]=mat[j]->vec[i];
   }
}

bool operator==(const matrix &m1,const matrix&m2){
   if ((m1.row != m2.row) || (m1.col != m2.col) )
      return false;
   for (int i=0;i<m1.row;i++) {
```

```
        vector v1(m1.mat[i],m1.row);vector v2(m2.mat[i],m2.row);
     for(int j=0;j<m1.col;j++) {
          if(v1.getvalue(j) != v2.getvalue(j))
              return false;
        }
    }
    return true;
}

ostream& operator<<(ostream& out,const matrix &m) {
    for(int i=0;i<m.row;i++)
        out<<*m.mat[i];
    return out;
}


int matrix::tightness() {
    int sum=0;
    for(int i=0;i<row;i++)
     for(int j=0;j<col;j++)
         sum=sum+mat[i]->vec[j];//get the sum of all 1s in matrix
    return (row*col-sum);//return the number of zeros
}

int matrix::Degree_Inter(vector &degree){
    //we use the vector class object- degree to record the information of a matrix
    int max_index=0;
    for(int i=0;i<row;i++) {
     for(int j=i+1;j<row;j++){
          vector v1(mat[i],row);vector v2(mat[j],row);
          if (degree.vec[i]==0){
              max_index=max_index+1;
              degree.vec[i]=max_index;
          }
         if(Isequal(v1,v2)) {
              if(degree.vec[j]==0)
                  degree.vec[j]=max_index;
           }
        }
    }
    if(degree.vec[row-1]==0) {
        max_index++;
        degree.vec[row-1]=max_index;
    }
    return max_index;
}

void matrix::tight(double tight,unsigned seed) {
    int tmp;
```

```
    do {
     int num1=random()% row;
     int num2=random()% col;
     mat[num1]->vec[num2]=0;
     tmp=tightness();
    }while (tmp <int(tight*row*col));

}

void matrix::set_equal(int current_dg,int desired_dg, vector degree) {
    int index=0;
    int num1,num2,num3,num4;
    int dif=current_dg - desired_dg;//we need dif pairs of vectors to be equal
    int *A=new int[current_dg+1];
        for(int i=1;i<=current_dg;i++)
           A[i]=0;
        //A[j] contains the number of elements whose value=j (j>=1)
        for(int j=0;j<row;j++)
            A[degree.vec[j]]=A[degree.vec[j]]+1;
        //to classify each vector into its responding equivalence class
        int **B;
        B=new int*[current_dg];
        for(i=0;i<current_dg;i++)
            B[i]=new int[A[i+1]];
        for(j=1;j<=current_dg;j++) {
         for (int k=0;k<row;k++) {
             if (degree.vec[k]==j) {
                 B[j-1][index]=k;
                 index++;
             } // end of if
         } //end of inter for
            index=0;
        }
    if (dif>0) {//if the current degree > the desired degree, we need to set some pairs of vectors
                //to be equal.

     //to select any two vectors to be equal
        for (i=0;i<dif;i++) {
         num1=random()%current_dg;
            num2=random()%current_dg;
         num3=random()%A[num1+1];
         num4=random()%A[num2+1];
            if (num1>num2)
             mat[B[num1][num3]]=mat[B[num2][num4]];
            else
                mat[B[num2][num4]]=mat[B[num1][num3]];
        } //end of for
     //free memory
        delete A;
```

```
      delete []B;
    }
   else if (dif<0){// if the current degree < the desired degree, we need reduce the number of
    equivalent class
      for(int i=0;i<(-1)*dif;i++) {
       //take any one vectors out of the equivalent classes in random
         num1=random()% current_dg;
         while (A[num1+1] == 1)
          num1=random()% current_dg;
         num2=random()% A[num1+1];
         num3=random()%col;
       num4=random()%col;
         int count=0;
         while((count<=current_dg)&&(mat[B[num1][num2]]->vec[num3]==mat[B[num1][num2]]-
    >vec[num4])) {
            num3=random()% col;
            num4=random()% col;
            count++;
         }
         //cout<<"ok"<<endl;
         if (mat[B[num1][num2]]->vec[num3] != mat[B[num1][num2]]->vec[num4])
            change(B[num1][num2],num3,num4);
         else {
          if (((random()% 10) /10.0)>0.5){
               if (mat[B[num1][num2]]->vec[num3]==0)
                  mat[B[num1][num2]]->vec[num3]=1;
               else
                  mat[B[num1][num2]]->vec[num3]=0;
               }
            else {
             if (mat[B[num1][num2]]->vec[num4]==0)
                  mat[B[num1][num2]]->vec[num4]=1;
               else
                  mat[B[num1][num2]]->vec[num4]=0;
               }
         }
      }//end of for
   }//end of if
   else {//if dif=0, then do nothing
      }
   delete A;
   delete []B;
}//end of function

void matrix::permute(int K){
    int i,num, v1,v2,fail=0;
    num=random()%(K*(K-1)/2);
    int **V;
    V=new int*[K];
```

```
    for(i=0;i<K;i++)
        V[i]=new int[K];
    for(i=0;i<K;i++)
     for(int j=0;j<K;j++)
         V[i][j]=0;
    for(i=0;i<=num;i++) {
        v1=random()%K;
        v2=random()%K;
        while(v1==v2 && fail<=num) {
            v1=random()%K;
            v2=random()%K;
            fail++;
        }
        swap(v1,v2);
    }
    delete []V;
}

void matrix::need_change() {
    matrix n(row);
    for(int i=0;i<row;i++) {
     for(int j=0;j<col;j++) {
         if (mat[i]->vec[j]==1)
             n.mat[i]->vec[j]=0;
            else
             n.mat[i]->vec[j]=1;
      }
    }
    *this=n;
}

//-----------------------------------------------------------------------------------------------
// Implementation of Queue class
//-----------------------------------------------------------------------------------------------

Queue::Queue() {
    front=NULL;
    rear=NULL;
} // end constructer

Queue::~Queue() {
    make_empty();
} // end destructor

Queue::Queue(const Queue& TheQ) {
    if ( TheQ.empty() ) {
     front=NULL;
     rear=NULL; //original list is empty
     }
```

```
   else {
    QueueNode *x=TheQ.front;// x is temporary pointer, which traverses TheQ.

    // copy the first Queuenode
       front=new QueueNode;
       front->element=x->element;
       rear=front;// The new Queue just has a node.

       // copy the rest element of the list.
       x=x->next;
       while ( x != NULL) {
        rear->next=new QueueNode;
           rear=rear->next;
           rear->element=x->element;
        x=x->next;
        }
       rear->next=NULL;
    }
} // end copy constructor

Queue& Queue::operator=(const Queue& TheQ) {

   // Deallocated memory.
   if ( this != & TheQ )
      make_empty();

   // Special case, the Queue is empty
   if ( TheQ.front == NULL && TheQ.rear == NULL) {
    front=NULL;
    rear=NULL; // The original list is empty
          }

   else {
    QueueNode *x=TheQ.front;// x is temporary pointer, which traverses TheQ.

    // copy the first Queuenode
       front=new QueueNode;
       front->element=x->element;
       rear=front;// The new Queue just has a node.

       // copy the rest element of the list.
       x=x->next;
       while ( x != NULL) {
        rear->next=new QueueNode;
           rear=rear->next;
           rear->element=x->element;
        x=x->next;
        }
       rear->next=NULL;
```

```
    }
   return *this;
} // end assignment operator

//Uses stdlib.h and iostream.h
void Queue::enqueue(int newElement) {
   // creat a new node
   QueueNode *newptr;
   newptr=new QueueNode;

   if (newptr !=NULL ) { // check allocation and allocation successful,
                         // set data portion of new node
    newptr->element=newElement;

       // insert the new node
       // special case, the Queue is empty
       if ( front==NULL && rear==NULL) {
        front=newptr;
        rear=front;// The new Queue just has one node.
           }

       // the queue is not empty
    else {
          rear->next=newptr;
          rear=newptr;
          }
       }
   else {
    cout<<"The Queue is full, insuffucient memory. \n";
    exit(1);
    }
} //end enqueue

//Uses stdlib.h and iostream.h
int Queue::dequeue() {
   if ( !empty() ) { // the Queue is not empty.
    int front_item=front->element;

       // special case: the Queue just has one node.
       if (front==rear) {
        delete front;
           front=NULL;
           rear=NULL;
           }

       // the queue has more than one node
    else {
          QueueNode *x=front;// x is temporary pointer.
        front=front->next;
```

```
        delete x;
          x=NULL;
          }
        return front_item;
        }
    else { // the Queue is empty
     cout<<"The Queue is empty. \n";
     return 0;
     }
} // end dequeue

void Queue::make_empty() {
    QueueNode *x=front;  // x is temporary pointer, which traverses the Queue.
    while (! empty() ) {
        // special case: the Queue just has one node.
        if (front==rear) {
         delete front;
            front=NULL;
            rear=NULL;
            }
        // the queue has more than one node
        else {
        front=front->next;
        delete x;
        x=front;
      } //end if
    } // end while loop
} //end make_empty

bool Queue::empty() const {
    if (front==NULL && rear==NULL )
     return true;
    else
     return false;
} // end empty

//-----------------------------------------------------------------------------------------------
//Implementation of other functions in main program
//-----------------------------------------------------------------------------------------------
float ran1() {
    static long int a=100001;
    a=(a*random())%2796203;
    return a;
}

float ran2() {
    static long int a=1;
    a=(a*32719+random())%32749;
    return a;
```

```
}

float getseed() {
   float x=(random()%10/10.0);
   if (x>0.5)
      return ran1();
   else
      return ran2();
}

void assign_CS(ofstream& fout,int C,int N){
   int i, j,k,v1,v2;bool Connected;
   int size=N*(N-1)/2;
   int *V=new int[size+1];//indicates which two variables are constrained
   CS_vars *constraint=new CS_vars[C+1];//keep the record of two variables limited by a
    constraint
   int *visited=new int[N+1]; //to mark a variable if it is visited

   do {
      Connected=1;
      for(i=1;i<=size;i++)
         V[i]=0;

      //assign a constraint to any two variables
      for (k=1;k<=C;k++) {
         int num=random()%(size+1-k);
         int count=1;
         bool finish=false;
         for (j=1; (j<=size) && (!finish);j++) {
            if ( (count==(num+1)) && (V[j]==0) ) {
               V[j]=1;
               finish=true;
            }
            else if (V[j]==0)
               count++;
         } //end of for
      } //end of for
      int count=1;
      int index=1;

      //pick up the two constrainted variables, and store them.
      for (i=1;i<=N;i++) {
         for (j=i+1;j<=N;j++) {
            if (V[index]==1) {
               constraint[count].var1=i;
               constraint[count].var2=j;
               count++;
             }//end of if
             index++;
```

```
            }
        }

        // to check if the CSP is connected
        Queue Q;
        for(i=1;i<=N;i++)
            visited[i]=0;
        visited[1]=1;
        Q.enqueue(1);
        while ( !Q.empty() ) {
            int v=Q.dequeue();
            for(int j=1;j<=C;j++) {
                v1=constraint[j].var1;
                v2=constraint[j].var2;
                if (v1==v && visited[v2]==0) {
                    Q.enqueue(v2);
                    visited[v2]=1;
                }
                if (v2==v && visited[v1]==0) {
                    Q.enqueue(v1);
                    visited[v1]=1;
                }
            } //end of for
        } //end of while

        //if C<(N-1), it is impossible to be connected
        if (C >= (N-1)) {
            //if there exists any a variable that is not visited, then this CSP is not connected
            for(i=1;i<=N;i++) {
                if (visited[i]==0) {
                    Connected=0;
                    break;
                    }
            } //end of if
        } //end of if
    } while (!Connected); //if not connected, then select againt
    //store the connected CSP into output file
    for (i=1;i<=C;i++) {
        v1=constraint[i].var1;
        v2=constraint[i].var2;
        fout<<v1<<" "<<setw(3)<<v2<<" "<<setw(3)<<i<<endl;
    }
    //deallocate memory
    delete visited;
    delete constraint;
    delete V;
}
```

```
void usage(){
   cout << "Usage: gen N K C I T outfile"<< endl;
   exit(1);
}
```

10 Appendix 2: example of an output file

```
CSP-10-5-13-3-0.36
10 5 -1

13
1 5 5
1 0 1 1 0
1 1 1 1 1
1 0 1 1 0
1 0 1 1 0
1 1 0 0 0

2 5 5
1 1 0 1 0
1 1 0 1 1
1 0 0 1 1
1 0 0 1 1
1 0 0 1 1

3 5 5
1 0 0 1 1
1 0 0 1 1
1 1 1 1 0
1 0 0 1 1
0 1 1 0 1

4 5 5
1 1 0 1 0
0 1 0 1 1
0 1 0 1 1
0 1 0 1 1
1 1 0 1 1

5 5 5
0 1 0 1 1
0 1 0 1 1
0 1 0 1 1
1 1 1 1 0
1 0 1 1 0

6 5 5
0 1 1 1 0
0 1 1 1 0
1 1 1 0 1
1 1 0 0 1
0 1 1 1 0

7 5 5
```

```
0 1 1 1 0
0 1 1 1 0
0 1 1 1 0
1 1 1 0 0
1 1 1 0 1
8 5 5
0 1 0 1 1
0 1 0 1 1
0 1 0 1 1
1 1 1 0 1
1 0 1 1 0

9 5 5
0 1 1 1 1
1 1 0 1 0
1 1 0 1 0
1 1 0 1 0
1 1 0 0 1

10 5 5
1 0 0 1 1
0 1 1 1 1
0 1 0 1 1
0 1 0 1 1
0 1 0 1 1

11 5 5
0 1 1 0 1
1 1 0 1 1
0 1 0 1 1
0 1 1 0 1
0 1 1 0 1

12 5 5
1 1 0 0 1
1 1 0 0 1
1 1 1 0 0
1 1 1 1 0
1 1 0 0 1

13 5 5
1 1 0 0 1
1 1 0 0 1
1 1 0 0 1
0 1 1 0 1
1 1 1 0 1

4  8  1
1  9  2
2 10  3
2  9  4
```

7 10 5
3 4 6
4 6 7
8 10 8
1 3 9
1 6 10
6 10 11
9 10 12

11 Biobliography

Freuder, E. C. and D. Sabin (1997). Interchangeability Supports Abstraction and Reformulation for Multi-Dimensional Constraint Satisfaction. Proc. of AAAI-97, Providence, Rhode Island.