Completing Latin Square Puzzle With Backtracking Algorithm

Revi Fajar Martha[1], Yogie Adrisatria[2], Syahrul Anwar[3]

Informatics Engineering, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung

E-mail : if13005@students.if.itb.ac.id[1],
if13035@students.if.itb.ac.id[2], if13061@students.if.itb.ac.id[3]

Abstract

Many puzzles can we find in our daily life, they can be game, IQ test question, school subject or many kind of their form. One of the ever popular puzzle is Latin Square Puzzle. Maybe without we realize that we have found that puzzle in our live. It is a simple puzzle to understand, only completing the NxN table with N unique symbols but no same symbol at one row or colum. But in the next level with big number of N we will got problem of it. To solve that some of algorithm is used to completing the puzzle such as Brute Force Algorithm and BackTracking Algorithm.

Keywords*: backtracking algorithm, latin square, puzzle*

1. Preface

Back Tracking Algorithm have used for many core of computation problem solving, one of the problem that excited to explore and completion is Latin Square Puzzle. Many problem in our life that without we realize we have used the concept of completing that puzzle such as scheduling problem, and management problem.With this article/jurnal we hope we can solve problems that have the concept as similar as the Latin Square Problem, especially for software development.

2. History Of Latin Square Name

S	A	T	O	R
A	R	E	P	O
T	E	N	E	T
O	P	E	R	A
R	O	T	A	S

A grafitto from Roman times is shown at the above. This was a kind of "Kilroy Was Here" message. Until recently, only in the Graeco-Roman world was the general public able to read and appreciate grafitti. It has been found from Hadrian's Wall in England to the desert wastes of Arabia. It says: "The sower Arepo holds carefully the wheels" when read in any direction. Pure nonsense, it was apparently designed to excite the superstitious. It is in Latin, and is a Square, but is not a Latin Square, at least not in the strict sense, because the rows and columns contain different letters, not simply the same letters rearranged. However, it may have provided the name Latin Square.

1	i	-1	-i
i	-1	-i	1
-1	-i	1	i
-i	1	i	-1

The table at the above contains the numbers 1, i, -1 and -i, where i is the imaginary unit (i×i = -1). It is a multiplication table, giving the result C of the product AB, where A is the first element in a row, and B the first element in a column. The first row and first column serve as labels as well as table members here, since it is unnecessary to repeat them for this purpose. This is also the "multiplication table" for any *group* whose four members correspond to 1, i, -1 and -i, though they may not be numbers. In fact, group members are usually transformations, and "multiplication" means performing two transformations in a row. A group is a set of elements that is closed under whatever "multiplication" is defined for them. This means that the result of any multiplication is again a member of the group. Furthermore, the group must contain the identity element (corresponding to 1) and the inverse Y to any element X, such that XY = YX = 1. Y is usually written X^{-1}. Quite importantly, the multiplication must be associative. That is, (AB)C = A(BC) for any three members A, B, C of the group. The requirements for the identity and the inverse mean that no element is repeated in any row or column, so that each row or column contains each element once and once only. This is the strict definition of a Latin Square.

3. Definition and History Of Latin Square Completion

Latin square is a Latin rectangle with $k = n$. Specifically, a Latin square consists of n sets of the numbers 1 to n arranged in such a way that no orthogonal (row or column) contains the same number twice. For example, the two Latin squares of order two are given by

$$\begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}, \quad (1)$$

the Latin squares of order three are given by

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 3 & 2 \\ 2 & 1 & 3 \\ 3 & 2 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 3 & 2 \\ 3 & 2 & 1 \\ 2 & 1 & 3 \end{bmatrix},$$

$$\begin{bmatrix} 2 & 1 & 3 \\ 1 & 3 & 2 \\ 3 & 2 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 1 & 3 \\ 3 & 2 & 1 \\ 1 & 3 & 2 \end{bmatrix}, \begin{bmatrix} 2 & 3 & 1 \\ 1 & 2 & 3 \\ 3 & 1 & 2 \end{bmatrix}, \begin{bmatrix} 2 & 3 & 1 \\ 3 & 1 & 2 \\ 1 & 2 & 3 \end{bmatrix},$$

$$\begin{bmatrix} 3 & 2 & 1 \\ 1 & 3 & 2 \\ 2 & 1 & 3 \end{bmatrix}, \begin{bmatrix} 3 & 2 & 1 \\ 2 & 1 & 3 \\ 1 & 3 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 1 & 2 \\ 1 & 2 & 3 \\ 2 & 3 & 1 \end{bmatrix}, \begin{bmatrix} 3 & 1 & 2 \\ 2 & 3 & 1 \\ 1 & 2 & 3 \end{bmatrix}, \quad (2)$$

and two of the whopping 576 Latin squares of order 4 are given by

$$\begin{bmatrix} 2 & 1 & 3 \\ 1 & 3 & 2 \\ 3 & 2 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 1 & 3 \\ 3 & 2 & 1 \\ 1 & 3 & 2 \end{bmatrix}, \begin{bmatrix} 2 & 3 & 1 \\ 1 & 2 & 3 \\ 3 & 1 & 2 \end{bmatrix}, \begin{bmatrix} 2 & 3 & 1 \\ 3 & 1 & 2 \\ 1 & 2 & 3 \end{bmatrix}, \quad (3)$$

The numbers $N(n,n)$ of Latin squares of order n = 1, 2, ... are 1, 2, 12, 576, 161280, ...[3]

A pair of Latin squares is said to be orthogonal if the n^2 pairs formed by juxtaposing the two arrays are all distinct. For example, the two Latin squares

$$\begin{bmatrix} 3 & 2 & 1 \\ 2 & 1 & 3 \\ 1 & 3 & 2 \end{bmatrix} \qquad \begin{bmatrix} 2 & 3 & 1 \\ 1 & 2 & 3 \\ 3 & 1 & 2 \end{bmatrix} \quad (4)$$

are orthogonal. The number of pairs of orthogonal Latin squares of order n = 1, 2, ... are 0, 0, 36, 3456, ... [3]

A normalized, or reduced, Latin square is a Latin square with the first row and column given by $\{1, 2, \ldots, n\}$. General formulas for the number of *normalized* Latin squares $L(n,n)$ are given by Nechvatal (1981), Gessel (1987), and Shao and Wei (1992). The *total* number of Latin squares $N(n,n)$ of order n can then be computed from

$$N(n,n) = n!(n-1)!L(n,n). \quad (5)$$

The numbers of normalized Latin squares of order n = 1, 2, ..., are 1, 1, 1, 4, 56, 9408, ... [3]. McKay and Rogoyski (1995) give the number of normalized Latin rectangles $L(k,n)$ for n = 1, ..., 10, as well as estimates for $L(n,n)$ with n = 11, 12, ..., 15.

n	$L(n,n)$
11	$N(n,n) = n!(n-1)!L(n,n).$
12	1.62×10^{44}
13	2.51×10^{56}
14	2.33×10^{70}
15	1.5×10^{86}

.

4. Back Tracking Implementation and Source Sample

One of source that use backtracking to solve 9x9 Latin Square:

----------source begin----------

```
#include <stdio.h>
#define SIZE 20

int sq[SIZE][SIZE];
int allowed[SIZE][SIZE];
int row[SIZE];
int col[SIZE];
int n;
int allvec;

int nrbitset(int x)
{
      int r = 0;
      for (int i = 0; i < n; i++)
            if ((1 << i) & x)
                  r++;
      return r;
}

void set(int i, int j, int k)
{
      if (sq[i][j] != 0)
      {
```

```
            printf("%d %d %d %d\n",
i, j, sq[i][j], k);
            exit(1);
      }
      sq[i][j] = k+1;
      row[i] &= ~(1 << k);
      col[j] &= ~(1 << k);
}
void iset(int i, int j, int k)
{     set(i-1,j-1,k-1);
}
void reset(int i, int j)
{
      int k = sq[i][j]-1;
      sq[i][j] = 0;
      row[i] |= (1 << k);
      col[j] |= (1 << k);
}
int possiblemoves()
{
      int p = 0;

      for (int i = 0; i < n; i++)
            for (int j = 0; j < n;
j++)
                  if (sq[i][j] ==
0)
                  {
                        int v =
nrbitset(row[i] & col[j] &
allowed[i][j]);
                        if (v == 0)

      return 0;
                        p += v; //
v*v - (v < 2 ? 0 : v-2);
                  }

      return p;
}

int count = 0;

bool solve(int depth)
{
      //printf("%*.*s%d\n", depth,
depth, "", possiblemoves());
      struct
      {     int pm;
            int i;
            int j;
            int k;
      } sols;

      sols.pm = 0;

      int v;
      for (int i = 0; i < n; i++)
            for (int j = 0; j < n;
j++)
                  if (sq[i][j] == 0
&& nrbitset(v = row[i] & col[j] &
allowed[i][j]) == 1)
                  {
                        for (int k
= 0; k < n; k++)
                              if
((1 << k) & v)
                              {

      set(i,j,k);

      if (solve(depth+1))

            return true;

      reset(i,j);



      return false;
                              }
                  }

      int o = 0;
      for (int i = 0; i < n; i++)
            for (int j = 0; j < n;
j++)
                  if (sq[i][j] ==
0)
                  {
                        o++;
                        int x =
row[i] & col[j] & allowed[i][j];
                        for (int k
= 0; k < n; k++)
                              if
((1 << k) & x)
                              {

      set(i,j,k);

      int pm = possiblemoves();

      reset(i,j);


      if (pm > sols.pm)

      {


            sols.pm = pm;

            sols.i = i;

            sols.j = j;

            sols.k = k;

      }
```

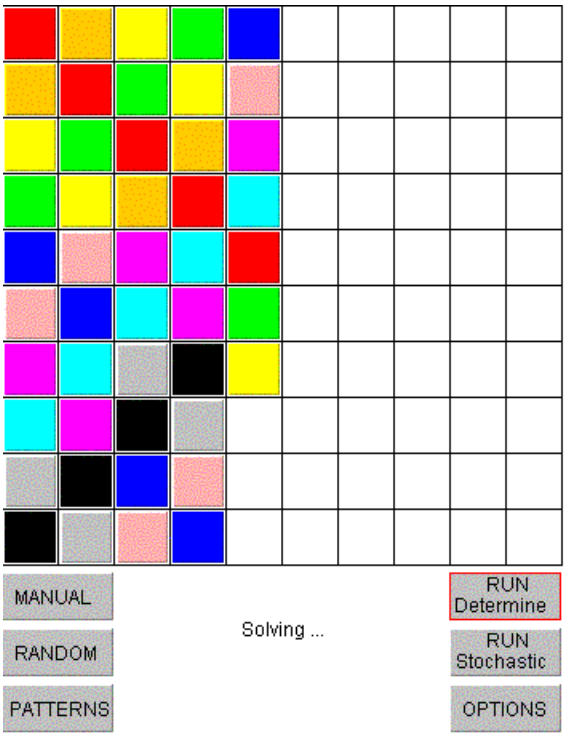
```
                              }
                        }

      //printf("nrsols = %d\n", o);

      if (o == 0)
      {
#if 0
            for (int j = n-1; j >=
0; j--)
            {
                  for (int i = 0; i
< n; i++)

      printf("%3d", sq[i][j]);
                  printf("\n");
            }
#else
            for (int j = n-1; j >=
0; j--)
            {
                  for (int i = 0; i
< n; i++)

      printf("%1d", sq[i][j]);
            }
            printf("\n");
#endif
            if (count++ > 100000)
                  return true;
            //return true;
      }

      if (sols.pm > 0)
      {
            set(sols.i, sols.j,
sols.k);
            if (solve(depth+1))
                  return true;
            reset(sols.i, sols.j);
            allowed[sols.i][sols.j]
&= ~(1 << sols.k);
            if (solve(depth+1))
                  return true;
            allowed[sols.i][sols.j]
|= (1 << sols.k);
      }

      return false;
}

int main()
{
      n = 9;
      allvec = (2 << n) - 1;
      for (int i = 0; i < n; i++)
      {
            row[i] = allvec;
            col[i] = allvec;
            for (int j = 0; j < n;
j++)
            {
                  sq[i][j] = 0;
                  allowed[i][j] =
allvec;
            }
      }

      iset(1,1,1);
      iset(2,3,1);
      iset(4,3,4);
      iset(4,5,1);
      iset(4,8,5);
      iset(6,4,2);
      iset(6,7,6);
      iset(7,6,2);
      iset(8,3,3);
      solve(9);

      return 0;
}
```

----------end of source----------

There is a good demo of this puzzle completion, its called Quasigroup completion :

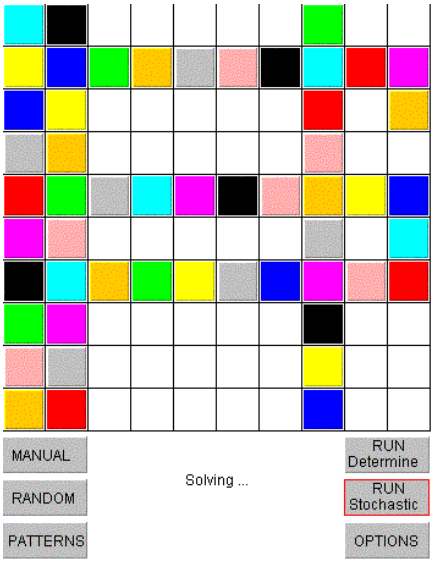
a. **completion with deterministic step**

start from first row and column and end at last cell.



b. completion with stochastic step

ramdomize the first cell and next completion



5. Conclusion

Back Tracking Algorithm is one of the most popular algorithm for many problem solving, so does for Ltin Square completion, if we use brute force it will spend more time because each component will be tried for all cell so if we use the backtrack we can decrease a large of step because every a stuck step it only goes back to last valid step.

References

1. C. E. Weatherburn, *A First Course in Mathmatical Statistics* (Cambridge: C.U.P., 1968), Chapter XI. One of the best of many texts in statistics.

2. R. S. Burington and D. C. May, *Handbook of Probability and Statistics with Tables* (Sandusky, Ohio: Handbook Publishers, Inc., 1958), pp. 276-279. There are many books of statistical tables; all contain the F-test.

3. mathworld.wolfram.com
4. iwriteit.com