Sudoku solver using Evolutionary Algorithm Techniques

Raushan Sharma (Roll: 18MA20058)
Kaushik Bhartiya (Roll: 18MA20018)

Term Project

COURSE: MT21104 - Genetic Algorithms in Engineering Process Modelling

Contents

Abstract

This term project studies the problems involved in solving, rating and generating Sudoku puzzles with application of evolutionary algorithms. Traditional generate-and-test solution strategies work very well for the classical form of the puzzle, but break down for puzzle sizes larger than 9×9. An alternative approach is the use of genetic algorithms (GAs). GAs have proved effective in attacking a number of NP-complete problems, particularly optimization problems such as the Travelling Salesman Problem.

Sudoku can be regarded as a constraint satisfaction problem which belongs to the class of NP-complete problems. It is therefore natural to explore the possibility whether GAs are also suitable for solving Sudoku puzzles.
When solved with genetic algorithms, it can be handled as a multi-objective optimization problem.

The three objectives of this study were:
1) to test if genetic algorithm optimization is an efficient method for solving Sudoku puzzles
2) whether genetic algorithms can be used to generate new puzzles efficiently
3) whether genetic algorithms can be used as a rating machine that evaluates the difficulty of a given Sudoku puzzle.

The last of these objectives is approached by testing if puzzles that are considered difficult for a human solver are also difficult for the genetic algorithm or not, i.e. take more number of generations to reach the solution or not. The results presented in this project seem to support the conclusion that these objectives are reasonably well met with genetic algorithm optimization.

1 Introduction

Sudoku started in 1783 with the Swiss mathematician Leonhard Euler, who invented Latin Squares, but Sudoku puzzles, as we know them today, were first published in 1979.

A Latin Square problem is not entirely the same as a Sudoku puzzle, but they have many similarities. A Latin Square is a grid of size $n \times n$, which contains all the numbers from 1 through n exactly once in every row and column.

General Sudoku:

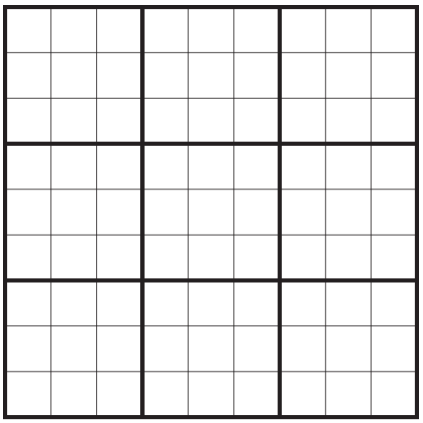
A general Sudoku puzzle consists of a $n^2 \times n^2$ grid, which is divided into $n \times n$ squares, where n is the order of the puzzle.Throughout the paper, the following notations regarding Sudoku will be used:

- A *cell* refers to one of the n^4 entries in the grid.
- The *value* of a cell refers to the number placed in the cell.
- A *row-*, *column-* or *square-domain* refers to the rows, columns and squares of the grid.
- A *candidate* is a number that could be placed in the cell.
- The *order* of the puzzle refers to the size of n. A typical 9×9 grid, will therefore be a puzzle of order 3.

A Sudoku puzzle has the following constraints:

- Each cell must have a value from 1 to n^2.
- Each of the row, column and square domains must contain the values from 1 to n^2 exactly once.

The difference between a Latin Square and a solved Sudoku puzzle is therefore that a Sudoku is more constrained than the Latin Square, as a valid Sudoku must also satisfy the square constraints. The fact that each row and column constraint must be satisfied in a valid Sudoku puzzle shows that every solved Sudoku is also a Latin Square, but not the other way around. Hence this project will also be helpful in solving Latin Square puzzles.



(a) Empty Sudoku grid

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

(b) Sudoku grid with clues

Figure 1: Sudoku problem instances

In Figure 1 a classical order 3 Sudoku grid is displayed, with and without initial values (clues). It is important to note that not every order 3 problem instance is considered to be a real Sudoku puzzle.

In order for a puzzle to be considered a proper Sudoku problem instance, it must have a unique solution, which can be determined by making logical conclusions step-wise based on the values already present in the puzzle. Hence, the importance and difficulty of making own Sudoku puzzle is not limited only to satisfying the Sudoku constraints. Therefore, generating new Sudoku puzzles efficiently is another problem of importance which we will be discussing in our term project.

Rating the Sudoku Puzzle:

A Sudoku puzzle is usually associated with a difficulty rating, which indicates how easy or difficult a given puzzle is to solve.

The obvious factors in determining a Sudoku puzzles rating would be the number of clues and the placement of the clues. One may argue that intuitively a Sudoku with few clues is more difficult than a Sudoku with many clues. Although this is right in many cases, it is not true for all Sudoku puzzles. There exist puzzles that have many clues, but are much more difficult than some puzzles with few clues.

The lowest possible number of starting hints (clues) that can provide a unique solution is till now 17, however, it has not been proven that there are no puzzles with 16 clues, but to date, none have been found. Another prerequisite for a unique solution is the use of atleast 8 of the 9 possible values when giving hints. Since, if only 7 values are used (say, the numbers 1 through 7), then any solution found would not be unique. This is due to the fact that another trivial solution could be found just by exchanging the two unused numbers in clues (the 8 and the 9).

Likewise, the placement of the clues gives rise to different difficulties. A Sudoku puzzle with a certain placement of the clues can be very different in rating compared to a Sudoku puzzle with the same placement of clues, but where the values of the clues are different.
This concludes that neither the number of clues nor the placement is enough to determine the difficulty rating of a puzzle, but instead they are believed to be the main factors in determining if a puzzle has a unique solution. In the Sudoku community, a good rating reflects how difficult a puzzle is for a human to solve.

2 Algorithm

In order to test the proposed algorithm, we decided to use an integer coded elitist Genetic Algorithm (GA). The size of the GA chromosome is 81 integer numbers, implemented as a matrix in the program, which is further divided into nine sub-blocks of nine numbers each.

For solving tougher Sudoku puzzles, it is important that the start of GA should be strong. Hence, firstly we filled the trivial blanks of the puzzle, where only one candidate could have appeared. And then we initialized the population of chromosomes such that each of them contained the initial givens and trivial blanks at the correct positions.

The single-point crossover operations were applied only between same rows of different chromosomes, and the swap mutations only inside the rows. The population size was 8000 , and elitism ratio was 200 *per* 8000 *chromosomes*. All the new individuals were generated by first applying crossover and then mutations to the crossover result.

A found solution or the number of generations not exceeding 2000 were the stop conditions. However, a restart condition as proposed in [5] was also put in order to tackle the problem of getting stuck in some local optimum. The GA when restarted, was initialised with the population containing the best individuals of previous generations. This ensured a good start for the next GA run.

Also, while solving the puzzle it was important to keep a check that the given clues must stay in their original positions. This was taken care of with the use of helper arrays.

Therefore, a Sudoku solution obeys four conditions:
1) Each row has to contain each integer from 1 to 9
2) Each column has to contain each integer from 1 to 9
3) Each 3×3 subgrid must contain each integer from 1 to 9
4) The clues must stay in their original positions

By an appropriate solving approach, one of the conditions from 1) to 3) can be controlled. Hence only three remaining conditions are subject to optimization. We chose to program the GA so that conditions 1) and 4) are automatically fulfilled and only the conditions 1) and 2) are optimized.

This GA does not use direct mutations or crossovers that could generate illegal situations for rows. On the contrary, columns and 3×3 subgrids can contain integers more than once in non-optimal situation. The genetic operators[1] are not allowed to move the initial values (clues). This is guaranteed by the help of a helper array, which indicates whether a number is fixed or not.

2.1 The Selection Operation

We used the classical k-way tournament selection, where we chose $k = 100$, i.e. each time 100 individuals from the population were selected and a tournament was run and the fittest among those was selected for the population for mating.
Also, each individual was declared as a structure variable in the code, where with each individual, its fitness and helper arrays to count the number of

[1] crossover and mutation operators

occurrences of each number from 1 to 9 in every row, column and sub-grid. This was used in the fitness calculation of each individual, and has been discussed later in the section discussing about the fitness function.

Pseudo Code

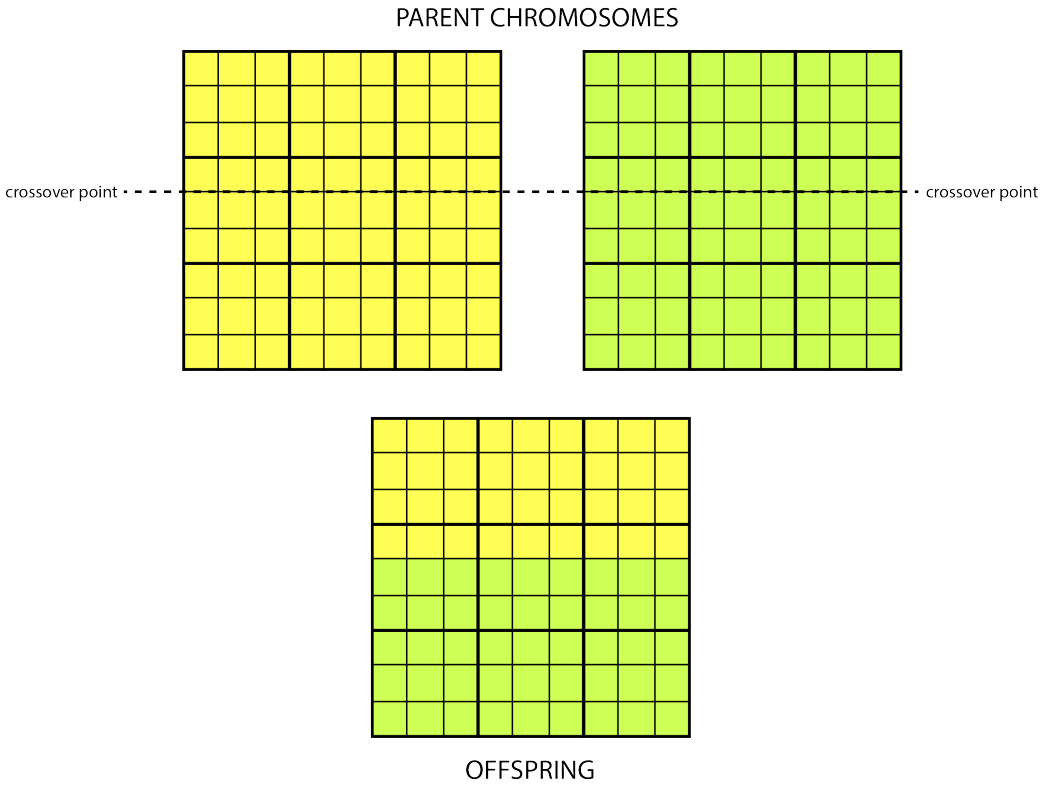
```
struct chromosome{    N=9
                      int ans[N][N];
                      int helprow[N][N],helpcol[N][N],helpsubgrid[N][N];
                      double fitness;
                  }
POP=8000, K=100

tournament ( chromosome p[POP] )
      { for ( i=0 ; i ≤ K; i++)
            r = rand()% POP
            temp[i]=p[r]
        sort temp[ ] in increasing order
        return temp[0]
      }
```

2.2 The Crossover Operation

Single-point crossover operation was applied only between rows of two different individuals since this ensured that the given initial clues are not misplaced, as the rows are always checked for being a permutation of $\{1, 2, ..., 9\}$.

Uniform crossover was also tested where one of each row of two parent solutions was randomly selected and assigned that row to the offspring. However, results obtained showed that both- the single-point crossover and the uniform crossover worked equally well and both ensured that the initial clues are not misplaced by this operation. The crossover probability was taken to be high (0.8 here).



Pseudo Code

```
crossover ( chromosome a , chromosome b )
        { r=rand()%N
          for ( i=0 , j=0 ; i < r , j < N ; i++ , j++ )
                offspring.ans[ i ][ j ] = a.ans[ i ][ j ]
          otherwise
                offspring.ans[ i ][ j ] = b.ans[ i ][ j ]
          calculate_fitness( offspring )
          return offspring
        }
```

2.3 The Mutation Operation

Mutations are applied only within a row. We used the swap mutation strategy in the GA. In swap mutation, the values of two positions in a row are exchanged. Each time mutation is tried inside a row, the help array of clues is checked. If it is illegal to change the randomly chosen position, mutation is omitted. The swap mutation probability was chosen to be small (0.4 here) so that good and potential chromosomes don't change drastically and lose their fitness.

The test runs with using different number of sequential swaps showed that the GA with just one swap mutation performed almost as well as GA with 1 to 5 swap mutation sequences.

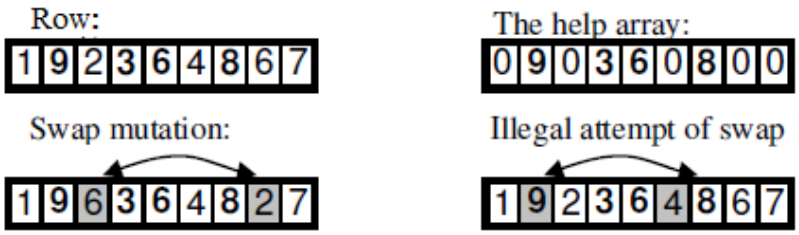


Figure 2: The swap mutation used in Sudoku optimization. Up left is one row and up right the static values of that row (9,3,6 and 8 are given clues). The mutation is applied so that we randomly select positions inside the row, and then compare the selected positions in the help array to see if the positions are free to change

There was one more special rule controlling whether the mutation was performed or abandoned. In this rule, we have another help table that tells how many times each digit appears in each row and column. If after the attempted mutation some digits appear three times or less, the mutation is done, otherwise not.

These rules take some time to calculate, but the overall performance was enhanced as the convergence became faster. These rules also decrease the real mutation probability which was taken to be 0.4.

Pseudo Code

```
mutate ( chromosome offspring )
        { r = random number in (0,1)
          if ( r < pm )
              r1, r2, r3 = rand() % N
              x = offspring.ans[ r3 ][ r1 ], y = offspring.ans[ r3 ][ r2 ]
              if ( arr[ r3 ][ r1 ] = 0 & arr[ r3 ][ r2 ] = 0 )
                  if( count of x and y in columns r2 and r1 respectively ≤ 3 )
                        swap ( offspring.ans[ r3 ][ r1 ], offspring.ans[ r3 ][ r2 ] )
        }
```

2.4 Fitness Function

To design a fitness function that would aid the GA search is often difficult in combinatorial problems. Here, the conditions that every row contains integers from 1 to 9 and that the fixed numbers stay in place were guaranteed intrinsically by keeping a check. And, the fitness function was designed such that whenever the other two conditions were violated, a penalty was added in the fitness of that chromosome. The fitness function used was as follows:

For each column and subgrid, the number of distinct numerals (*i.e. between 1 to 9*) was calculated. Let us denote any column and subgrid as an *array* named arr[ ]. Then the above number is found and denoted as unique(arr). Certainly, we would want to maximize this quantity for each *array*, and for the final solution of Sudoku puzzle, this quantity must equal 9 for each *array*. Then for each such *array* (column and subgrid), we calculated the quantity $\frac{9}{unique(arr)}$ and multiplied them all.

The resultant value was assigned as the fitness of the chromosome (potential solution). Clearly, this fitness function was to be minimized. Note that for every *array*,

$$1 \leq unique(arr) \leq 9$$

$$\Rightarrow \frac{9}{unique(arr)} \geq 1$$

So, as the fitness value of a chromosome is the product of quantities, each of which is ≥ 1, the minimum possible value of fitness is 1, which is achieved when all columns and subgrids contain 9 distinct numerals, which is nothing but the definition of a proper solution of a Sudoku puzzle.

Hence the termination condition of the algorithm was that whenever the fittest individual of a population in any generation had fitness value = 1, the program was terminated.

Pseudo Code

```
calculate_fitness ( chromosome a )
        { a.fitness = 1.0
          for ( i=0 ; i<N ; i++ )
              uniquecol[ i ] = 0
              for ( j=0 ; j<N ; j++ )
                  if ( a.helpcol[ i ][ j ] ≠ 0 )
                      uniquecol[ i ] += 1
              a.fitness = a.fitness*( 9.0/uniquecol[ i ] )

          for ( i=0 ; i<N ; i++ )
              uniquesub[ i ] = 0
              for ( j=0 ; j<N ; j++ )
                  if ( a.helpsubgrid[ i ][ j ] ≠ 0 )
                      uniquesub[ i ] += 1
              a.fitness = a.fitness*( 9.0/uniquesub[ i ] )

          return a.fitness
        }
```

---

3 Experimental Results

3.1 Effectiveness in Solving Sudoku Puzzles

In order to generate a strong and good initial population, as mentioned earlier, we firstly filled the trivial blanks of the Sudoku puzzle, where only one candidate could have appeared. For easy Sudoku puzzles, in many cases, this approach was found to be sufficient for finding a solution. Even for harder puzzles, it was worth filling in the trivially determined cells, in order to reduce the dimensionality of the problem.

We tried to test our algorithm by running it on various Sudoku puzzles collected from various sources like newspapers: *The Telegraph*, and Hindi newspaper *Sanmarg*, and websites like www.memory-improvement-tips.com. The results of running the Genetic Algorithm showed that it is very effective in solving Sudokus upto *Hard* level, however in many cases of *Very Hard* Sudoku puzzles, it gets stuck in a local optimal, and the GA has to restart. However, in these cases also the solution obtained was very near to the correct solution. But, once stuck in a local minimum (as the objective was to minimise the fitness function), the GA was seldom able to jump out and find the best solution, though restarting the GA was somewhat effective. Also, as expected, larger population size of solutions was helpful as it increased the search space for the optimal solution.

The final solution must have a fitness of 1.0. This was achieved in many cases. However, as mentioned earlier, mainly in the *Very Hard* and certain *Hard* puzzles, when it got stuck in some local minima, even after 2000 generations and one restart, the best solution we found was generally having a fitness of 1.265625 or 1.4328 or 1.6018. But, knowing that $(\frac{9}{8})^2 = 1.265625$, we can say that a fitness of 1.265625 means the obtained solution has a repetition of one digit in only 2 out of the total 18 columns and subgrids. Similarly, $(\frac{9}{8})^3 = 1.4328$ and $(\frac{9}{8})^4 = 1.6018$. Hence the obtained best solution was always close to the final solution.

Here are some results obtained for certain Sudoku puzzles of different difficulty levels from our code :

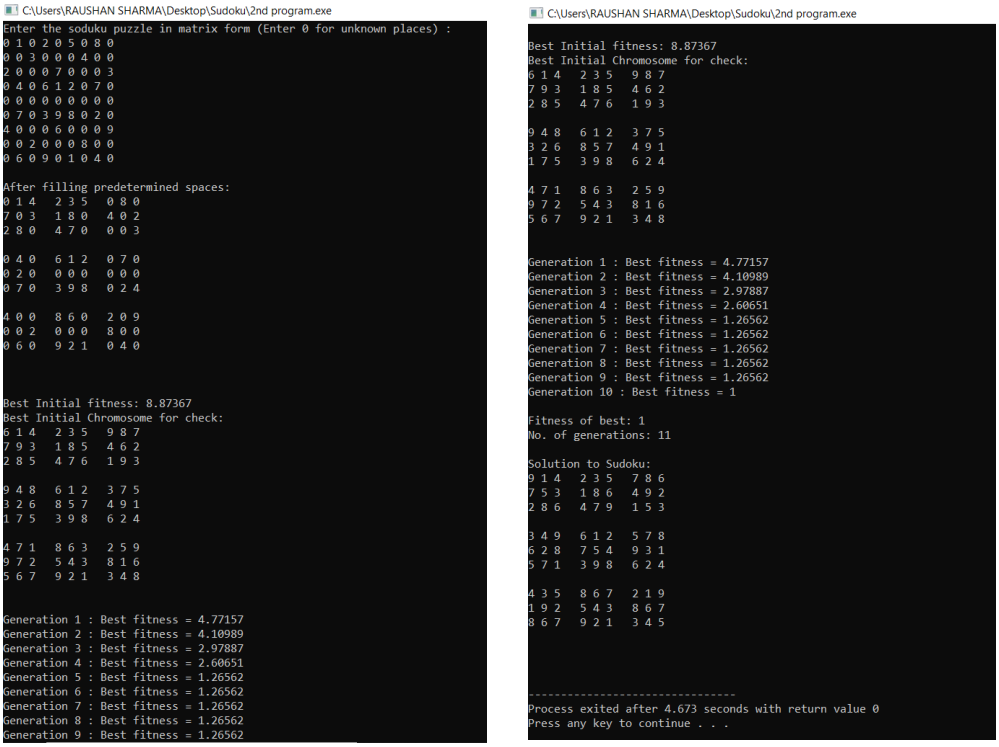


Figure 3: This puzzle has been taken from [5]

The above Sudoku is of moderate difficulty, and the GA is quite effective in solving this one.

Next we tried to test the Sudoku puzzle from the newspaper *The Telegraph-t2 on Sunday* dated 31st May 2020. It contained two Sudoku puzzles rated as Moderate and Gentle. Since our algorithm had a strong initialization of population by filling the trivial blanks, and as stated eariler, in many cases this was sufficient for solving many Easy to Moderately rated puzzles, both the puzzles from this newspaper were solved in just the first generation.

Here is the result obtained after testing the Moderately rated Sudoku puzzle from the same newspaper *(Sudoku No. 8116)* :

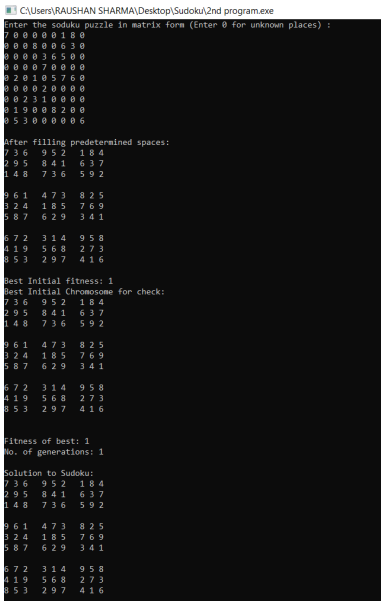
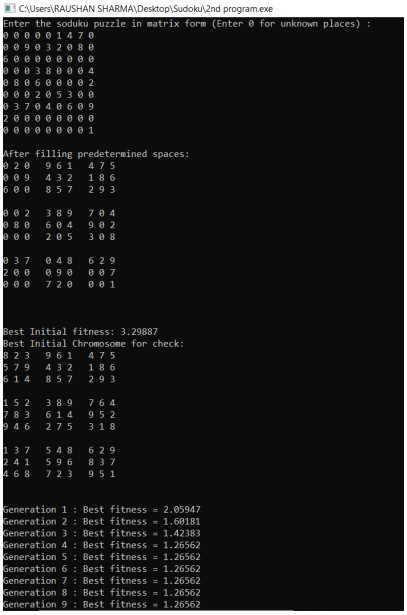


Figure 4: A Moderate Sudoku taken from newspaper *The Telegraph*



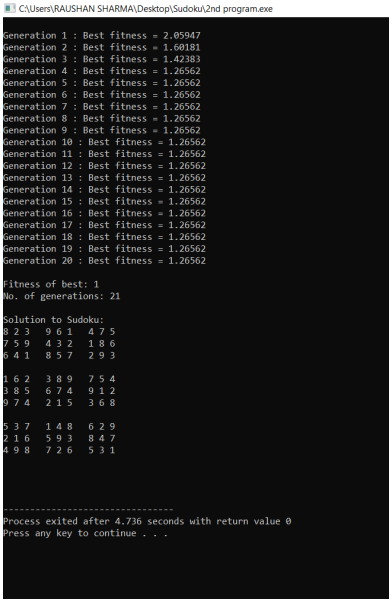


Figure 5: A Hard rated Sudoku taken from a Sudoku game mobile app

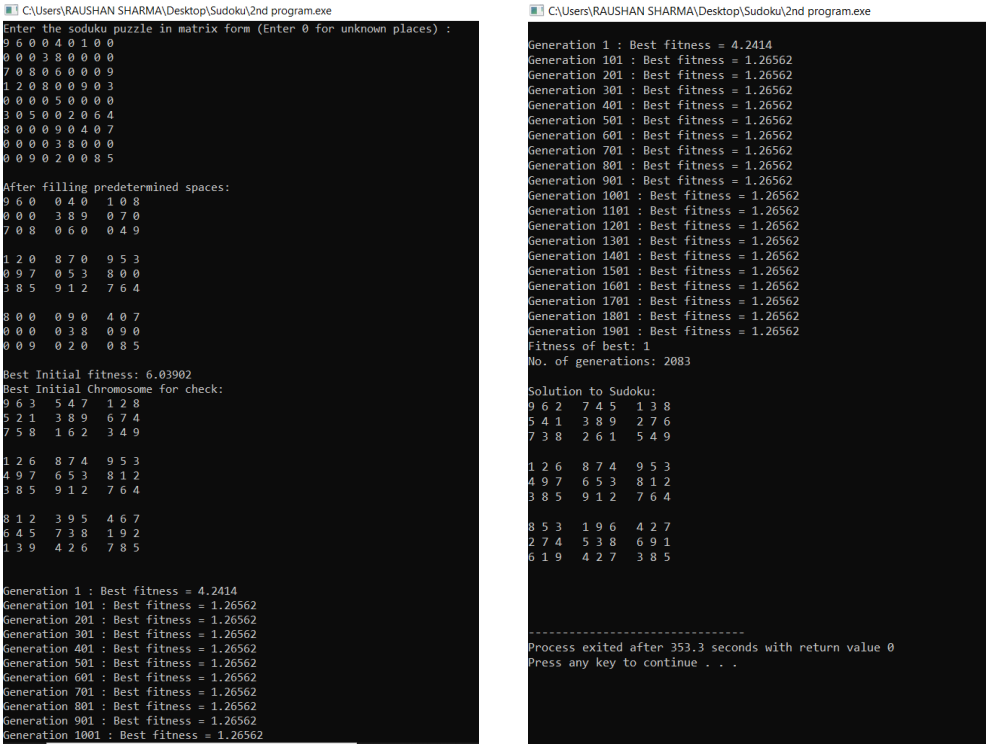


Figure 6: A Very Hard rated Sudoku taken from www.memory-improvement-tips.com

The puzzle above required the GA to be restarted with the new population containing the best solutions from the previous generations. This showed the efficiency of the restart algorithm used. However, still in the case of many Very Hard Sudoku puzzles, even after restarting, the solution was unable to converge to the correct solution and got stuck in some local minima. Though, the obtained best solution was always very close to the correct solution as discussed earlier.

The above results also show that the harder rated the puzzle was, the GA took more number of generations to reach the correct solution. Hence, the difficulty ratings given for Sudoku puzzles in newspapers and websites which we referred to were consistent with their difficulty in GA optimization.

3.2 New Sudoku Puzzle Creation

As discussed earlier regarding the importance of generation of Sudoku puzzles, a puzzle is considered a proper Sudoku problem instance if and only if it has a unique solution. We tried generating new Sudoku puzzles by first searching a new open Sudoku solution, giving the solution as input (no given numbers in the beginning), and then randomly picking 20-50 givens and removing the rest. In this process, we also ensured that there were no two such numbers who didn't appear in the givens at all. The reason for this was already discussed here, as then, the Sudoku can't have a unique solution. The new puzzle was then solved at max. 10 times in order to test that only a unique solution to the puzzle exists.

The preliminary tests seemed to suggest that if the same solution was found more than five times in successive solving runs, it was likely to be the only solution. We ran the Sudoku generation program about 50-60 times with each open Sudoku solution, because in randomly drawing certain number of givens, it was difficult to get a puzzle with a unique solution quickly. And in each round, some 20-50 givens were drawn randomly and the process was carried out as mentioned above.

Results obtained were not too satisfactory, as from an input Sudoku solution, generating a puzzle with around $50 - 60$ givens was successful, but such a Sudoku could be rated Very Easy to Easy. We were unable to generate Moderate to Hard level Sudoku, which contain ≤ 30 givens generally. Most often when only $20 - 30$ numbers were picked as initial clues, the Sudoku was not unique.

However, the above mentioned way, by no means guarantees the uniqueness of the Sudoku puzzle. In fact, some deterministic algorithm can better check the uniqueness. But, we wanted to implement and find out if Genetic Algorithms can be effective in uniqueness testing or not. To confirm our results, we also tested half of our GA generated Sudokus with the Sudoku Explainer software[1] and it confirmed that each of the Easy or so-Sudokus generated by our program had a unique solution.

[1]Link to the software used: www.sudokuwiki.org/sudoku.htm

4 Conclusion

In this term project, we tried to study the effectiveness of combinatorial Genetic Algorithm in solving, rating and generating Sudoku puzzles. Sudoku is a good laboratory for algorithm design, and it is based on one of the hardest unsolved problems in computer science - the NP complete problems.

The results showed that Sudoku, being an NP-complete problem, can be tackled by GA more effectively than certain deterministic algorithms like that of backtracking[1], especially when the size or order of Sudoku is large. The GA results can of course be enhanced by adding or implementing more problem related rules. However, if one adds too much problem specific logic to the Sudoku solving, there will be nothing left to optimize, therefore we decided to omit most of the problem specific logic/constraints and try a more straight forward GA optimization approach.

The other goal in this project was to study whether the difficulty ratings given for Sudoku puzzles in newspapers are consistent with their difficulty in GA optimization. The answer to this question seems to be positive with the results obtained. This meant that GA can be used for rating the difficulty of a new Sudoku puzzle.

Also, new puzzles were generated by finding one possible open Sudoku solution and then removing numbers randomly and testing if only one solution exists. It seemed that we can suppose the solution was unique if the same solution was found more than five times in a row, since each our GA generated puzzles had a unique solution according the Sudoku Explainer software.

5 Scope of Further Improvements

The problem of getting stuck in local optimum is something which has been discussed in many papers, and we would also like to study more on how the GA results could be enhanced by handling this problem more efficiently.

The fitness function choice in this project worked satisfactorily, but in future we might study more whether or not this is a proper GA fitness function for the Sudoku problem and try to improve on our results.

[1] Backtracking algorithm for Sudoku is a type of brute force using Depth-first search.

References

1. Christian Agerbeck, Mikael O. Hansen : *A Multi-Agent Approach to Solving NP-Complete Problems*. Kongens Lyngby, 2008. IMM-Masters Thesis 2008

2. Wikipedia : https://en.wikipedia.org/wiki/Sudoku

3. Wikipedia : http://en.wikipedia.org/wiki/Latin_square

4. Yuji Sato, Hazuki Inoue : *Solving Sudoku with Genetic Operations that Preserve Building Blocks*. September 2010

5. Dr. John M. Weiss : *Genetic Algorithms and Sudoku*. South Dakota School of Mines and Technology (SDSM&T). MICS 2009

6. Sudoku Explainer : www.sudokuwiki.org/sudoku.htm

7. Kalyanmoy Deb : *Multi-Objective Optimization using Evolutionary Algorithms*. Indian Institute of Technology, Kanpur

8. Mike Gold : *Genetic Algorithms to come up with Sudoku Puzzles* : www.c-sharpcorner.com, September 2012

9. David Isaac Waters : *Sudokube - Using Genetic Algorithms to simultaneously solve multiple combinatorial problems* : semanticscholar.org, Oklahoma State University, 2005

10. Free Printable Sudoku Puzzles : www.memory-improvement-tips.com

Appendix: Code in C++

```
#include<bits/stdc++.h>
using namespace std;

#define pc 0.8
#define N 9
#define sqrtN 3
#define POP 8000
#define elite 200
#define GEN 2000

double pm=0.4;
int arr[N][N],helperR[N][N],helperC[N][N],helperS[N][N];

struct chromosome{
        int ans[N][N];
        int helprow[N][N],helpcol[N][N],helpsubgrid[N][N];
        double fitness;
}c[POP],best,parent[POP],coparent[POP],newgen[POP],offspring;


bool compare(chromosome a, chromosome b)
{
        return (a.fitness<b.fitness);
}

void calculate_fitness(chromosome& a)
{
        int uniquecol[N],uniquesub[N];
        int i,j;
        a.fitness=1.0;
        for(i=0;i<N;i++)
          { uniquecol[i]=0;
                for(j=0;j<N;j++)
                  { if(a.helpcol[i][j]!=0)
                      uniquecol[i]+=1;
                  }
                a.fitness=a.fitness*(9.0/(double)uniquecol[i]);
                //Calculating fitness for column repetitions
          }

        for(i=0;i<N;i++)
          { uniquesub[i]=0;
                for(j=0;j<N;j++)
                  { if(a.helpsubgrid[i][j]!=0)
                      uniquesub[i]+=1;
                  }
                a.fitness=a.fitness*(9.0/(double)uniquesub[i]);
                //Calculating fitness for sub-grid repetitions
          }
}
```

```
void generate_pop()
{
        int i,j,k,l,x,y;
        helperC[N][N]={0};helperS[N][N]={0};
        for(k=0;k<POP;k++)
          for(i=0;i<N;i++)
                for(j=0;j<N;j++)
                  {
                        c[k].helprow[i][j]=helperR[i][j];
                        c[k].helpcol[i][j]=0;
                        c[k].helpsubgrid[i][j]=0;
                  }

        for(k=0;k<POP;k++)
          {
                for(j=0;j<N;j++)
                  for(i=0;i<N;i++)
                    {
                          if(arr[i][j]!=0)
                            { x=arr[i][j];
                              helperC[j][x-1]=1;
                              c[k].helpcol[j][x-1]+=1;
                              //Initializing helpcol with givens
                            }
                    }
          }

    for(k=0;k<POP;k++)
          {
                for(i=0;i<N;i++)
                  { for(j=0;j<N;j++)
                          {
                            x=arr[i][j];
                            if(x!=0)
                              { y=sqrtN*(i/sqrtN)+(j/sqrtN);
                                helperS[y][x-1]=1;
                                c[k].helpsubgrid[y][x-1]+=1;
                                //Initializing helpsubgrid with givens
                              }
                          }
                  }
          }

        int possible[N][N][N],countposs[N][N];
        for(i=0;i<N;i++)
          {
                for(j=0;j<N;j++)
                  { countposs[i][j]=0;
//Countposs array stores the number of possible values at a place
                    for(l=0;l<N;l++)
                       possible[i][j][l]=0;
//Possible array stores the possible values at a place
                  }
          }
        for(i=0;i<N;i++)
          {
                for(j=0;j<N;j++)
```

```
                { if(arr[i][j]==0)
                    {
                      y=sqrtN*(i/sqrtN)+(j/sqrtN);
                      for(l=0;l<N;l++)
                        {if(helperR[i][l]==0 && helperC[j][l]==0
                              && helperS[y][l]==0)
                            { possible[i][j][l]=1;
                              countposs[i][j]++;
                            }
                        }
                    }
                }
        }

       int times=10;
       while(times--)
       {

       for(i=0;i<N;i++)
         { for(j=0;j<N;j++)
             {
               if(countposs[i][j]==1)
               /* Those places with only 1 possible value will
                  now be filled with that value*/
                 { for(l=0;l<N;l++)
                     { if(possible[i][j][l]==1)
                         { arr[i][j]=l+1;
                           countposs[i][j]=0;
                           possible[i][j][l]=0;
                         }
                                 }
                         }
                 }
         }


for(i=0;i<N;i++)
    for(j=0;j<N;j++)
      {
               x=arr[i][j];
               if(x!=0)
                 helperR[i][x-1]=1;
                 /* Updating all the helpers after the update in
                    arr by filling obvious blanks */
               }
  for(j=0;j<N;j++)
        for(i=0;i<N;i++)
               {
                 if(arr[i][j]!=0)
                       { x=arr[i][j];
                         helperC[j][x-1]=1;
                       }
               }
  for(i=0;i<N;i++)
        { for(j=0;j<N;j++)
               {
                       x=arr[i][j];
```

```
                if(x!=0)
                  { y=sqrtN*(i/sqrtN)+(j/sqrtN);
                    helperS[y][x-1]=1;
                  }
            }
  }

for(i=0;i<N;i++)
  {
        for(j=0;j<N;j++)
          { countposs[i][j]=0;
          /* Count array will now store no. of possible
             values at a place after updating arr */
                for(l=0;l<N;l++)
                  possible[i][j][l]=0;
          /* Possible array will now store the possible
             values at a place after updating arr */
          }
  }
for(i=0;i<N;i++)
  {
    for(j=0;j<N;j++)
      { if(arr[i][j]==0)
          {
            y=sqrtN*(i/sqrtN)+(j/sqrtN);
            for(l=0;l<N;l++)
              {if(helperR[i][l]==0 && helperC[j][l]==0
                            && helperS[y][l]==0)
                          { possible[i][j][l]=1;
                            countposs[i][j]++;
                          }
                    }
                 }
          }
  }

/* We'll now update obvious numbers w.r.t each row */

int cnt;
for(int num=1;num<=N;num++)
  { for(i=0;i<N;i++)
      { cnt=0;
          for(j=0;j<N;j++)
            { if(possible[i][j][num-1]==1)
                { cnt++;
                }
            }
          if(cnt==1)
                { for(j=0;j<N;j++)
                    if(possible[i][j][num-1]==1)
                      { arr[i][j]=num;
                        countposs[i][j]=0;
                        possible[i][j][num-1]=0;
                      }
                }
          }
  }
```

```
    /* Updating helpers and possible array after filling
       obvious numbers with respect to each row */

for(i=0;i<N;i++)
   for(j=0;j<N;j++)
     {
             x=arr[i][j];
             if(x!=0)
               helperR[i][x-1]=1;
               /* Updating all the helpers after the update
                   in arr by filling obvious blanks */
     }
 for(j=0;j<N;j++)
       for(i=0;i<N;i++)
             {
               if(arr[i][j]!=0)
                     { x=arr[i][j];
                       helperC[j][x-1]=1;
                     }
             }
 for(i=0;i<N;i++)
       { for(j=0;j<N;j++)
             {
                     x=arr[i][j];
                     if(x!=0)
                       { y=sqrtN*(i/sqrtN)+(j/sqrtN);
                         helperS[y][x-1]=1;
                       }
             }
       }

     for(i=0;i<N;i++)
       {
             for(j=0;j<N;j++)
               { countposs[i][j]=0;
               /* Count array will now store no. of possible
                  values at a place after updating arr */
                  for(l=0;l<N;l++)
                    possible[i][j][l]=0;
                    /* Possible array will now store the
                       possible values at a place after
                       updating arr */
             }
       }
     for(i=0;i<N;i++)
       {
         for(j=0;j<N;j++)
           { if(arr[i][j]==0)
             {
               y=sqrtN*(i/sqrtN)+(j/sqrtN);
               for(l=0;l<N;l++)
                 {if(helperR[i][l]==0 && helperC[j][l]==0
                             && helperS[y][l]==0)
                             { possible[i][j][l]=1;
                               countposs[i][j]++;
                             }
```

```
                        }
                    }
                }
        }

    /* Now we'll update obvious numbers w.r.t each column */

    for(int num=1;num<=N;num++)
      { for(j=0;j<N;j++)
            { cnt=0;
              for(i=0;i<N;i++)
            { if(possible[i][j][num-1]==1)
                { cnt++;
                        }
                }
              if(cnt==1)
                { for(i=0;i<N;i++)
                    if(possible[i][j][num-1]==1)
                      { arr[i][j]=num;
                        countposs[i][j]=0;
                        possible[i][j][num-1]=0;
                      }
                }
            }
      }

    /* Updating helpers and possible array after filling
       obvious numbers with respect to each column */

for(i=0;i<N;i++)
   for(j=0;j<N;j++)
     {
            x=arr[i][j];
            if(x!=0)
              helperR[i][x-1]=1;
              /* Updating all the helpers after the update
                 in arr by filling obvious blanks */
     }
 for(j=0;j<N;j++)
       for(i=0;i<N;i++)
            {
              if(arr[i][j]!=0)
                    { x=arr[i][j];
                      helperC[j][x-1]=1;
                    }
            }
 for(i=0;i<N;i++)
       { for(j=0;j<N;j++)
            {
                    x=arr[i][j];
                    if(x!=0)
                      { y=sqrtN*(i/sqrtN)+(j/sqrtN);
                        helperS[y][x-1]=1;
                      }
            }
      }
```

```
for(i=0;i<N;i++)
  {
        for(j=0;j<N;j++)
          { countposs[i][j]=0;
          /* Count array will now store no. of possible
             values at a place after updating arr */
                for(l=0;l<N;l++)
                  possible[i][j][l]=0;
                   /* Possible array will now store the
                     possible values at a place after
                     updating arr */
          }
  }
for(i=0;i<N;i++)
  {
    for(j=0;j<N;j++)
      { if(arr[i][j]==0)
          {
            y=sqrtN*(i/sqrtN)+(j/sqrtN);
            for(l=0;l<N;l++)
              {if(helperR[i][l]==0 && helperC[j][l]==0
                          && helperS[y][l]==0)
                          { possible[i][j][l]=1;
                            countposs[i][j]++;
                          }
                      }
                    }
          }
  }

/* Now we'll update obvious numbers w.r.t each subgrid */

for(int num=1;num<=N;num++)
  { int ct[N]={0};
    for(i=0;i<N;i++)
      { for(j=0;j<N;j++)
          { y=sqrtN*(i/sqrtN)+(j/sqrtN);
            if(possible[i][j][num-1]==1)
              ct[y]++;
                }
          }
        for(l=0;l<N;l++)
          { if(ct[l]==1)
              { for(i=0;i<N;i++)
                  { for(j=0;j<N;j++)
                      { y=sqrtN*(i/sqrtN)+(j/sqrtN);
                        if(y==l)
                          { if(possible[i][j][num-1]==1)
                              { arr[i][j]=num;
                                countposs[i][j]=0;
                                possible[i][j][num-1]=0;
                              }
                          }
                      }
                  }
              }
```

```
      }

    for(i=0;i<N;i++)
  for(j=0;j<N;j++)
    {
            x=arr[i][j];
            if(x!=0)
              helperR[i][x-1]=1;
              /* Updating all the helpers after the update
                 in arr by filling obvious blanks */
            }
for(j=0;j<N;j++)
      for(i=0;i<N;i++)
            {
              if(arr[i][j]!=0)
                    { x=arr[i][j];
                      helperC[j][x-1]=1;
                    }
            }
for(i=0;i<N;i++)
      { for(j=0;j<N;j++)
            {
                    x=arr[i][j];
                    if(x!=0)
                      { y=sqrtN*(i/sqrtN)+(j/sqrtN);
                        helperS[y][x-1]=1;
                      }
            }
      }

    for(i=0;i<N;i++)
      {
            for(j=0;j<N;j++)
              { countposs[i][j]=0;
              /* Count array will now store no. of possible
                 values at a place after updating arr */
                 for(l=0;l<N;l++)
                   possible[i][j][l]=0;
                   /* Possible array will now store the
                      possible values at a place after
                      updating arr */
            }
      }
    for(i=0;i<N;i++)
      {
        for(j=0;j<N;j++)
          { if(arr[i][j]==0)
            {
              y=sqrtN*(i/sqrtN)+(j/sqrtN);
              for(l=0;l<N;l++)
                { if(helperR[i][l]==0 && helperC[j][l]==0
                             && helperS[y][l]==0)
                           { possible[i][j][l]=1;
                             countposs[i][j]++;
                           }
                  }
                }
```

```
                }
        }

}

cout<<"\nAfter filling predetermined spaces:\n";
for(i=0;i<N;i++)
  { for(j=0;j<N;j++)
      cout<<arr[i][j]<<" ";
    cout<<"\n";
      }
    cout<<"\n";

    for(k=0;k<POP;k++)
      for(i=0;i<N;i++)
        for(j=0;j<N;j++)
              { c[k].helprow[i][j]=helperR[i][j];
                c[k].helpcol[i][j]=helperC[i][j];
                c[k].helpsubgrid[i][j]=helperS[i][j];
              }

    /* Now we will initialize the population using the possible
       values at a place */

for(k=0;k<POP;k++)           //Initialization of Population
  {
    for(i=0;i<N;i++)
      {
        for(j=0;j<N;j++)
          {
            if(arr[i][j]!=0)
              c[k].ans[i][j]=arr[i][j];
            else
              {
                x=1+rand()%N;
                if(c[k].helprow[i][x-1]==0
                   && possible[i][j][x-1]==1)
                   {
                    c[k].ans[i][j]=x;
                    c[k].helprow[i][x-1]+=1;
                   }
              else if(c[k].helprow[i][x-1]!=0
                            || possible[i][j][x-1]==0)
                       {
                         x=1+rand()%N;
                         while(1)
                         /* Ensuring that helprow array for any
                            chromosome is an all 1s array */
                            {
                              if(c[k].helprow[i][x-1]==0)
                                {
                                  c[k].ans[i][j]=x;
                                  c[k].helprow[i][x-1]+=1;
                                  break;
                                }
                             x=x%N + 1;
                            }
```

```
                        }
                    }
                }
            }
        }

    for(k=0;k<POP;k++)
        {
            for(j=0;j<N;j++)
                {
                    for(i=0;i<N;i++)
                        {
                            x=c[k].ans[i][j];
                            c[k].helpcol[j][x-1]+=1;
                            //Updating helpcol
                        }
                }
        }

    for(k=0;k<POP;k++)
        {
          for(i=0;i<N;i++)
            { for(j=0;j<N;j++)
                    {
                      x=c[k].ans[i][j];
                      y=sqrtN*(i/sqrtN)+(j/sqrtN);
                      c[k].helpsubgrid[y][x-1]+=1;
                      //Updating helpsubgrid
                    }
            }
        }

    for(k=0;k<POP;k++)
        {
            c[k].fitness=1.0;
            calculate_fitness(c[k]);
        }

}  //End of generate_pop() function


chromosome tournament(chromosome p[POP])
{
    int i,j,k,M=10;
    /* M is for this tournament being an M-way tournament */
    chromosome temp[M];
    for(k=0;k<M;k++)
      { int r=rand()%POP;
        temp[k]=p[r];
      }
    sort(temp,temp+M,compare);
    return temp[0];
}

void mutate(chromosome& offspring,double pm)
{
    double r= (double)rand()/((double)RAND_MAX + 1.0);
```

```
        if(r<pm)
          {
                int rnd=rand()%N;
                int r1=rand()%N;
                int r2=rand()%N;
                if(arr[rnd][r1]==0 && arr[rnd][r2]==0)
                  { int temp = offspring.ans[rnd][r1];
                    offspring.ans[rnd][r1] = offspring.ans[rnd][r2];
                    offspring.ans[rnd][r2]=temp;
                    //Swap mutation
                  }
          }
}

chromosome crossover(chromosome a, chromosome b,double pm)
{
        chromosome offspring;
        int i,j,r,x,y;
        r=rand()%N;
        for(i=0;i<r;i++)
          for(j=0;j<N;j++)
            offspring.ans[i][j]=a.ans[i][j];
        for(i=r;i<N;i++)
          for(j=0;j<N;j++)
            offspring.ans[i][j]=b.ans[i][j];

        for(i=0;i<N;i++)
          { for(j=0;j<N;j++)
                { offspring.helprow[i][j]=1;
                  offspring.helpcol[i][j]=0;
                  offspring.helpsubgrid[i][j]=0;
            }
          }

        for(j=0;j<N;j++)
          {
                for(i=0;i<N;i++)
            {
                        x=offspring.ans[i][j];
                        offspring.helpcol[j][x-1]+=1;
                }
          }

        for(i=0;i<N;i++)
          { for(j=0;j<N;j++)
                {
                        x=offspring.ans[i][j];
                        y=sqrtN*(i/sqrtN)+(j/sqrtN);
                        offspring.helpsubgrid[y][x-1]+=1;
                }
          }
        offspring.fitness=1.0;
        calculate_fitness(offspring);

        chromosome mu=offspring;
        mutate(mu,pm);    // Mutation
        for(i=0;i<N;i++)
```

```
        { for(j=0;j<N;j++)
                { mu.helprow[i][j]=1;
                  mu.helpcol[i][j]=0;
                  mu.helpsubgrid[i][j]=0;
            }
        }

      for(j=0;j<N;j++)
        {
              for(i=0;i<N;i++)
                {
                      x=mu.ans[i][j];
                      mu.helpcol[j][x-1]+=1;
                }
        }

      for(i=0;i<N;i++)
        { for(j=0;j<N;j++)
                {
                      x=mu.ans[i][j];
                      y=sqrtN*(i/sqrtN)+(j/sqrtN);
                      mu.helpsubgrid[y][x-1]+=1;
                }
        }
      mu.fitness=1.0;
      calculate_fitness(mu);

      if(mu.fitness <= offspring.fitness)
        return mu;
      else
        return offspring;
}


int main()
{
   srand(time(0));
   int i,j,k,x,y;

      for(i=0;i<N;i++)
        for(j=0;j<N;j++)
          helperR[i][j]=0;
      cout<<"Enter the soduku puzzle in matrix form ";
      cout<<"(Enter 0 for unknown places) :\n";
      for(i=0;i<N;i++)
        {
          for(j=0;j<N;j++)
            {
              cin>>arr[i][j];
              x=arr[i][j];
              if(arr[i][j]!=0)
                helperR[i][x-1]+=1;
                /* helper array contains 1 if the value is a
                   given clue, otherwise 0 */
             }
         }
```

```
generate_pop();

/* Till now population has been initialized and fitness
  values of the chromosomes of the initial population
  have been calculated */

sort(c,c+POP,compare);
for(i=0;i<POP;i++)
  { parent[i]=c[i];
  }
best=c[0];

cout<<"\nBest Initial fitness: "<<best.fitness<<"\n";
cout<<"Best Initial Chromosome for check:\n";
for(i=0;i<N;i++)
  { for(j=0;j<N;j++)
      cout<<best.ans[i][j]<<" ";
    cout<<"\n";
  }
cout<<"\n";

int gen=1;
while(gen<=GEN && best.fitness>1.0)
  {
    for(k=0;k<elite;k++)
      { coparent[k]=parent[k];
        newgen[k]=parent[k];
      }
    for(k=elite;k<POP;k++)
      {
        coparent[k]=tournament(parent);
      }
    k=elite;
    while(k<POP)
      {
        int r1=rand()%POP;
        int r2=rand()%POP;
        double r=(double)rand()/((double)RAND_MAX+1.0);
        if(r<pc)
          {
            newgen[k++] =
            crossover(coparent[r1],coparent[r2],pm);
            /* Mutation function is called inside
               the crossover function */
          }
       }

   /* Now the new generation population is ready */

   sort(newgen,newgen+POP,compare);

   if(newgen[0].fitness<best.fitness)
        best=newgen[0];

   for(k=0;k<POP;k++)
        parent[k]=newgen[k];
```

```
            if(gen%100==1)
              { cout<<"Generation "<<gen<<" : Best fitness = "
                    <<best.fitness<<"\n";
              }
            gen++;

          }

        cout<<"\nFitness of best : "<<best.fitness<<"\n";
        cout<<"No. of generations : "<<gen<<"\n";
        cout<<"\nSolution to Sudoku:\n";

          for(i=0;i<N;i++)
            {
                    for(j=0;j<N;j++)
                      cout<<best.ans[i][j]<<" ";
                    cout<<"\n";
            }

          cout<<"\n\n";

    return 0;
}
```
