EXPERIMENTAL STUDY ON SUDOKU [*]

S. Maniccam
Computer Science Department
Eastern Michigan University
Ypsilanti, MI 48197, USA
smaniccam@emich.edu

ABSTRACT

This paper presents an experimental study on Sudoku puzzles. Various aspects of Sudoku are studied using computer experiments. The topics include an algorithm to solve the puzzle, test results, run time analysis, performance on difficult puzzles, handling larger size puzzles, finding multiple solutions, and designing puzzles. This study may serve as a source for designing questions, examples, and programming assignments related to Sudoku, recursion, backtracking, and experimentation in courses such as algorithms, data structures, and discrete mathematics.

1. SOLVING THE PUZZLE

Sudoku is a board game. The board is 9×9 with 81 slots. The board is divided into 9 non-overlapping 3×3 regions. Each slot must have an integer between 1 and 9. Some of the slots have initial values and these values cannot change. The problem is to fill the remaining empty slots such that each row, each column, and each 3×3 region has integers between 1 and 9 with no duplicates. There are various algorithms [1, 8] that solve the Sudoku puzzle. These algorithms include backtracking [2-5], graph coloring [4], exact cover [9], constraint programming [10], and stochastic search [11].

Here we describe a simple brute force algorithm. This algorithm uses recursion and backtracking [2-5]. The main idea of the algorithm is the following. Fill the board row by row from top to bottom and left to right. In an empty slot, try to place the numbers from 1 to 9. After placing a number in a slot, check whether that number has a duplicate in the same row, in the same column, or in the same 3×3 region. If there is a duplicate then try the next number. If there is no duplicate then try to fill the rest of the board recursively. If the rest of the board cannot be filled then try the next number. If the rest

[*] Copyright © 2013 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

of the board can be filled then the puzzle is solved. If none of the numbers from 1 to 9 can be placed in a slot then empty that slot and backtrack. The algorithm is described in detail in the fill method shown in Fig.1. The check method is described in the appendix.

2. TEST DATA

We tested the above algorithm on various Sudoku puzzles. As an example, a puzzle and a solution obtained by the algorithm are shown in Fig. 2. We used puzzles from the Daily Sudoku web site [6] for run time analysis. This site provides daily puzzles and maintains an archive of about 3000 puzzles from years 2005 to 2012. It also classifies each puzzle as easy, medium, hard, or very hard. We used 15 puzzles from June 2012. Among these puzzles, there are 5 easy, 5 medium, and 5 very hard puzzles.

3. HOW LONG DOES IT TAKE

One way to measure the Sudoku solving time is to directly measure the run time of the program. The program is written in Java. The System.currentTimeMillis method is used to measure the time in milliseconds. Computer experiments were performed on a Mac 10.7 / Intel Core i5 / 1.6 GHz / 2 GB computer. The set of 15 puzzles was divided into three groups as mentioned before. The run times in each group were measured. Then the average, the minimum, and the maximum run times were computed in each group.

Another way to measure the Sudoku solving time is to count the number of tries a human player will make to solve a puzzle. Specifically, we like to count the number of times a human player will perform write, rewrite, and erase operations on the board if the player executes the fill method by hand. In the fill method, these operations can be counted by defining a global counter and by incrementing that counter whenever a slot changes. The slots change in the lines board[x][y] = value and board[x][y] = 0. The tries were counted in each of the three puzzle groups. Then the average, the minimum, and the maximum tries were computed in each group.

It takes about 123 milliseconds and 390,968 tries to solve the puzzle in Fig. 2. The results of the 15 puzzles are summarized in Fig. 3. The results show that the program takes less than a second to solve any puzzle including the very hard ones. In fact, the average run time is less than 0.1 second. However the number tries the program makes is very large. The average number of tries is in the order of 10^4. Therefore it is not practical for human players to solve a puzzle by strictly following the fill algorithm. Human players usually employ various tricks to reduce the number tries.

Fig. 3 shows that the run time and the number of tries increase as the difficulty level of puzzles increases. The average number of tries is in the order of 10^2 for easy puzzles whereas it is in the order of 10^5 for very hard puzzles. The average run time increases about 50 fold as the puzzles change from easy to very hard. Another point to note is that there is a wide variation among the run times and the number of tries within the same difficulty level. For example, within the set of very hard problems, the easiest problem takes about 20,000 tries and 27 milliseconds whereas the hardest problem takes about 474,000 tries and 141 milliseconds.

```
main()
{
    //fill the board starting at the first slot
    if (fill(board, 0))
       display board;
    else
       board cannot be filled
}

//fill the board starting at a given location
boolean fill(int[][] board, int location)
{
    //determine the row and the column of the location
    int x = location/9; int y = location%9;

    //if the location exceeds the board then success
    if (location > 80)
        return true;
    //if the location already has an initial number
    //then fill the rest of the board recursively
    else if (board[x][y] != 0)
        return fill(board, location+1);
    else
    {
        //otherwise try numbers from 1 to 9
        for (int value = 1; value <= 9; value++)
        {
             //place the next number at the location
             board[x][y] = value;
             //if the number causes no conflicts with other
             //existing numbers and the rest of the board
             //can be filled recursively then success
             if (check(x, y) && fill(board, location+1))
                 return true;
        }
        //if all numbers from 1 to 9 fail then empty the
        //location and backtrack
        board[x][y] = 0;
        return false;
    }
}
```

Fig. 1. Sudoku solving algorithm

```
0 0 0 2 0 0 8 0 5    4 3 1 2 7 6 8 9 5
2 0 0 0 9 0 1 0 0    2 8 5 4 9 3 1 7 6
0 0 0 0 1 5 4 0 0    9 6 7 8 1 5 4 2 3
5 0 0 0 0 0 0 1 4    5 9 8 6 3 2 7 1 4
0 0 0 0 0 0 0 0 0    7 2 4 1 5 8 6 3 9
3 1 0 0 0 0 0 0 2    3 1 6 7 4 9 5 8 2
0 0 2 9 8 0 0 0 0    6 4 2 9 8 1 3 5 7
0 0 3 0 2 0 0 6 1    8 7 3 5 2 4 9 6 1
1 0 9 0 6 7 0 0 0    1 5 9 3 6 7 2 4 8
```

Fig. 2. A puzzle and its solution

	Easy	Medium	Very hard
Average time	1.2	36.8	70.5
Minimum time	1	2	27
Maximum time	2	63	141
Average tries	561	45467	162500
Minimum tries	188	1462	19836
Maximum tries	1404	98072	473842

Fig. 3. Run time results

```
0 0 0 0 0 0 0 0 0    1 2 0 4 0 0 3 0 0    0 0 0 0 0 0 0 3 9
0 0 0 0 0 3 0 8 5    3 0 0 0 1 0 0 5 0    0 0 0 0 0 1 0 0 5
0 0 1 0 2 0 0 0 0    0 0 6 0 0 0 1 0 0    0 0 3 0 5 0 8 0 0
0 0 0 5 0 7 0 0 0    7 0 0 0 9 0 0 0 0    0 0 8 0 9 0 0 0 6
0 0 4 0 0 0 1 0 0    0 4 0 6 0 3 0 0 0    0 7 0 0 0 2 0 0 0
0 9 0 0 0 0 0 0 0    0 0 3 0 0 2 0 0 0    1 0 0 4 0 0 0 0 0
5 0 0 0 0 0 0 7 3    5 0 0 0 8 0 7 0 0    0 0 9 0 8 0 0 5 0
0 0 2 0 1 0 0 0 0    0 0 7 0 0 0 0 0 5    0 2 0 0 0 0 6 0 0
0 0 0 0 4 0 0 0 9    0 0 0 0 0 0 0 9 8    4 0 0 7 0 0 0 0 0
```

Fig. 4. Three very difficult puzzles from Wikipedia

In theory, the fill method may have to potentially try all possible ways of filling the empty slots before finding a solution. Therefore its worst case run time is exponential in the number of empty slots. There are no known polynomial time algorithms that solve the general Sudoku problem and the problem has been proven to be NP-complete [7].

4. CRACKING HARD NUTS

Some Sudoku puzzles are considered especially difficult. Three such puzzles are shown in Fig. 4. We call these puzzles A, B, and C from to left to right. These puzzles were taken from the Wikipedia [8]. The Wikipedia states that puzzle A is a near worst case puzzle for the brute force algorithm used in this paper. Here the difficulty is measured in terms of the number of tries the algorithm makes. The Wikipedia also states that puzzles B and C are exceptionally difficult for human solvers. Here the difficulty is measured subjectively by expert players.

We ran the program on these three puzzles. We measured the run times and counted the number of tries. The puzzle A takes about 15 seconds and 691,752,849 tries. This puzzle does take an unusually large number of tries, almost 691 million. This number is significantly larger than the average number of tries in Fig. 3. In fact, the largest number of tries there is less than 0.5 million. The puzzle B takes about 0.28 seconds and 3,004,926 tries. The puzzle C takes about 0.3 seconds and 3,040,223 tries. These two puzzles take about 3 million tries each. This number is also much larger than the average number of tries in Fig. 3. All these results seem to support the Wikipedia's claim that the puzzles A, B, and C are exceptionally difficult.

5. DEALING WITH MONSTERS

The classic 9×9 puzzle may be extended to 16×16 puzzle or more generally to $n^2 \times n^2$ puzzle. The $n^2 \times n^2$ puzzle has an $n^2 \times n^2$ board with n^4 slots. The board is divided into n^2 non-overlapping n×n regions. Each slot must have an integer between 1 and n^2. The problem is to fill the slots such that each row, each column, and each n×n region has integers between 1 and n^2 with no duplicates. These larger puzzles are often called monsters.

The $n^2 \times n^2$ puzzle can be solved by modifying the fill method as follows. Divide by n^2 when computing the x and y coordinates. Stop when the location reaches n^4. Try the values from 1 to n^2 in the loop. The check method also needs to be modified appropriately.

We tested the modified algorithm on various 16×16 puzzles. As an example, a puzzle and a solution obtained by the modified algorithm are shown in Fig. 5. The program takes about 194 seconds and 4,036,832,547 tries to solve this puzzle. The number of tries in this puzzle is in the order of 10^9 whereas the average number of tries in a 9×9 puzzle in Fig. 3 is in the order of 10^4. The time to solve this puzzle is about 5000 times longer than the average time to solve a 9×9 puzzle in Fig. 3. Although the run time and the number of tries increase drastically in 16×16 puzzles, the program solves these puzzles in minutes.

```
 0  0  3  0  9  0  0  0  1  0  2  6  0  0  0  8
 0  6  0  5  0  0  0  0  0  8  0 15  7  0  3 16
 0  9  7 16  3  8  0 15  0  0  0  5  1  2  0  0
 4  0  0  0 14  6  1  0  7  3  0 10  0 11  0  0
 0  0  0  4  0 13  0  0  0  0  6 14  0  1  0  0
 0  0 11  9  0  3  0  0  0 16  0  0  8  0  0  4
 3  8 13  0  0  0  0  0 10  4  1 12  0  0  0 14
 0 10  2 14  0  4  0  1  0  0  0  8  5  0  7  0
 0  0 14  0  0  0  0  0  0 10 13  0  0  3  0  2
 0 13  0  6  4 14  9  0  2  0  5  0  0  0 16 12
11  5  9  0  0 16  0 10  6  0  8  4 13  0 14  0
 0 15  8  1  0  0  0  0 14  0  0 16  9  6  4  0
14  0 16 13  0  0  8  0  0  2  0  3  6  0  0  0
 0  1  0 15 16 11  0  0  0  6  0  7  2  9  0 13
 8  3  0  0  0  2 13  7  0  1  0  0  0  0 12 15
 0  7  0  2  0  0  0  6  0  5  0 13  0 16  8  0
```

```
12 14  3 11  9  7  5 16  1 13  2  6  4 10 15  8
13  6  1  5 10 12  4  2  9  8 11 15  7 14  3 16
10  9  7 16  3  8 11 15  4 14 12  5  1  2 13  6
 4  2 15  8 14  6  1 13  7  3 16 10 12 11  9  5
15 16  5  4  8 13  7 12  3 11  6 14 10  1  2  9
 1 12 11  9 15  3 10 14  5 16  7  2  8 13  6  4
 3  8 13  7  6  9  2  5 10  4  1 12 16 15 11 14
 6 10  2 14 11  4 16  1 13  9 15  8  5 12  7  3
16  4 14 12  7  1  6  8 11 10 13  9 15  3  5  2
 7 13 10  6  4 14  9  3  2 15  5  1 11  8 16 12
11  5  9  3  2 16 15 10  6 12  8  4 13  7 14  1
 2 15  8  1 13  5 12 11 14  7  3 16  9  6  4 10
14 11 16 13 12 10  8  9 15  2  4  3  6  5  1  7
 5  1 12 15 16 11  3  4  8  6 14  7  2  9 10 13
 8  3  6 10  5  2 13  7 16  1  9 11 14  4 12 15
 9  7  4  2  1 15 14  6 12  5 10 13  3 16  8 11
```

Fig. 5. A 16×16 monster and its solution

```
p = new int[9][9][9];
for (x = 0; x < 9; x++)
     for (y = 0; y < 9; y++)
     {
          for (i = 0; i < 9; i++)
               p[x][y][i] = i+1;

          for (i = 0; i < 9; i++)
          {
               j = i + generator.nextInt(9-i);
               temp = p[x][y][i];
               p[x][y][i] = p[x][y][j];
               p[x][y][j] = temp;
          }
      }
```

Fig. 6. Creating 9×9×9 array of random permutations

```
0 0 0 0 8 0 7 0 3
0 0 0 0 0 7 6 0 0
6 0 0 0 0 0 0 0 2
0 0 0 0 0 0 0 0 0
9 0 0 0 6 0 0 0 0
0 0 0 0 2 0 4 0 0
4 0 8 0 5 0 2 0 0
0 0 0 0 0 0 0 0 0
0 0 0 1 0 4 0 0 6
```

```
0 5 8 4 0 0 6 3 0
3 0 4 0 0 0 7 0 0
0 6 7 3 0 5 0 0 0
2 3 1 5 9 0 0 4 0
0 0 0 0 1 8 0 0 0
0 0 0 0 3 4 2 0 9
8 1 3 6 5 7 4 9 2
6 0 5 8 2 9 1 0 0
7 0 2 0 0 0 5 0 0
```

```
0 4 7 1 6 9 5 8 2
5 6 8 3 2 4 1 0 7
1 9 0 5 8 7 3 6 4
0 0 5 2 9 0 4 0 6
4 3 1 7 5 6 9 2 8
9 0 6 0 4 0 7 5 0
6 5 3 4 0 0 2 1 9
8 0 4 0 0 2 6 0 5
2 7 0 6 1 5 0 4 0
```

Fig. 7. Computer generated puzzles

```
0 9 0 0 5 1 0 0 0
0 4 0 0 2 0 0 0 0
0 0 0 0 0 0 0 0 0
0 2 0 0 0 0 1 5 0
0 0 0 0 0 9 6 0 0
0 0 0 0 0 0 0 0 0
9 0 3 0 0 0 0 0 0
0 0 4 0 0 0 0 0 0
0 0 0 0 4 2 3 0 6
```

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

Fig. 8. A very difficult computer generated puzzle and its solution

6. FINDING MULTIPLE SOLUTIONS

Some Sudoku puzzles have multiple solutions. In such puzzles, the empty slots can be filled in more than one ways. The fill method described earlier will only find one solution in such puzzles. The fill method can be modified so that it will find all solutions. The main modification is to return false when a solution is found in the line if (location > 80). This forces the algorithm to backtrack even after finding a solution. Backtracking forces the algorithm to fill the empty slots in more than one ways. This results in finding all solutions. Every time a solution is found in the line if (location > 80), one can print the solution and increment a global counter to keep track of the number of solutions.

We did some experiments with the modified algorithm. The puzzle in Fig. 2 has 109 solutions. If the value 2 is removed from the first row then the modified puzzle will have 401 solutions. If the values 2 and 8 are removed from the first row then the modified puzzle will have 1158 solutions. If the values 2, 8, 9, and 1 are removed from the top two rows then the modified puzzle will have 5153 solutions. The program finds the multiple solutions reasonably fast. For example, it finds and prints all 109 solutions to the console in about 0.5 seconds, and all 5153 solutions in about 15 seconds.

The number of solutions increases as more values are removed from the board. In the extreme case, all values are removed and the initial board is empty. How many solutions does an empty Sudoku puzzle have? The answer is approximately 6.67×10^{21} [5]. The program will take years to find such a large number of solutions. Therefore we wanted to determine the number of solutions the program is able to find within a given time period. The program was run for 1, 5, and 10 minutes. The program is able to find and print about 216,000 solutions in 1 minute, about 1,174,000 solutions in 5 minutes, and about 2,230,000 solutions in 10 minutes.

7. DESIGNING PUZZLES

One approach to creating Sudoku puzzles is the following. Start with an empty board. Solve the empty board and find a solution. Then erase some slots of the solution. The resulting board becomes a Sudoku puzzle. In order to prevent a player from knowing how the puzzle was created, some sort of randomness must be included in this procedure.

The fill method can be modified so that it will find a random solution of the empty board. Recall that the fill method described earlier fills each slot with numbers 1, 2, 3, . . . 9 in the ascending order. Instead of filling each slot in the ascending order, one can fill each slot in a random order. Each slot can use its own random order. One can associate a random order with each slot in the board by creating 81 random permutations of 1, 2, 3, . . . 9 and storing them in a 9×9×9 array. This array is named p and it is a global array. The details are shown in Fig. 6. The fill method needs two modifications. The loop will be (index = 0; index < 9; index++). Each slot will be filled with values from the p array as board[x][y] = p[x][y][index]. Note that a random solution is ultimately determined by the seed value of the random number generator. The puzzle designer can obtain different solutions by changing the seed value.

After a random solution of the empty board is obtained, some slots are erased. The number of erased slots is determined by the density value, which is chosen by the puzzle designer. The density means the percentage of occupied slots in the puzzle and its value ranges between 0 and 1. For example, density = 0.6 means that approximately 60% of slots are occupied and 40% of slots are empty. The slots are erased probabilistically as follows. Visit each slot and generate a random number between 0 and 1. If the number is greater than the density then erase the slot, otherwise do not erase the slot. The puzzle designer can obtain different puzzles by changing the density value.

We created various puzzles using the program. Three such puzzles are shown in Fig. 7. The puzzles from left to right were created using density values 0.25, 0.5, 0.75 and seed values 10, 20, 30 respectively. The average time to create a puzzle is less than 0.1 second. Note that a puzzle created by the program always has a solution and some puzzles may have more than one solution. The puzzles in Fig. 7 all have unique solutions.

One can use the number of tries needed to solve a puzzle as an objective measure of difficulty. Difficulty levels can be defined in terms of ranges of tries. A puzzle with a certain difficulty level can be designed by randomly generating puzzles until finding one that requires the number of tries that is within the desired range. Out of curiosity, we wanted to find a puzzle that is harder than the puzzle A in Fig. 4. It requires 691,752,849 tries and about 15 seconds to solve it. We randomly generated puzzles and searched for a puzzle that requires more than 691,752,849 tries. We did find such a puzzle. The puzzle and its solution are shown in Fig. 8. This puzzle requires 2,313,917,402 tries and about 57 seconds to solve it. This puzzle is harder than all three puzzles in Fig. 4, in terms of the number of tries made by the brute force algorithm.

REFERENCES

[1] Rosenhouse, J., Taalman, L., *Taking Sudoku Seriously*, Oxford University Press, 2012

[2] Levitin, A., *Design and Analysis of Algorithms*, Addison Wesley, 2007.

[3] Koffman, E., Wolfgang, P., *Data Structures*, Wiley, 2010.

[4] Rosen, K., *Discrete Mathematics and Its Applications*, McGraw Hill, 2006.

[5] Weiss, M., *Data Structures and Problem Solving Using Java*, Addison Wesley, 2005.

[6] www.dailysudoku.com/sudoku/archive/2012/06/index.html

[7] en.wikipedia.org/w/index.php?title=Mathematics_of_Sudoku&oldid=517379543

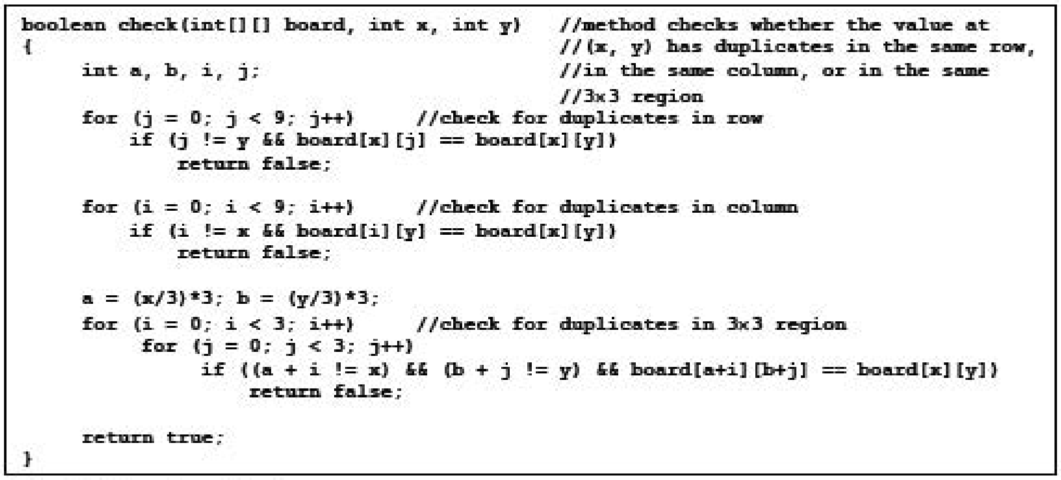
[8] en.wikipedia.org/w/index.php?title=Sudoku_solving_algorithms&oldid=517261520

[9] Cormen, T., Leiserson, C., Rivest, R., *Introduction to Algorithms*, MIT Press, 2002.

[10] Russell, S., Norvig, P., *Artificial Intelligence*, Prentice Hall, 2009.

[11] Michalewicz, Z., Fogel, D., *How to Solve It: Modern Heuristics*, Springer, 2004.

APPENDIX



```
boolean check(int[][] board, int x, int y)   //method checks whether the value at
{                                            //(x, y) has duplicates in the same row,
     int a, b, i, j;                         //in the same column, or in the same
                                             //3x3 region
     for (j = 0; j < 9; j++)     //check for duplicates in row
         if (j != y && board[x][j] == board[x][y])
             return false;

     for (i = 0; i < 9; i++)     //check for duplicates in column
         if (i != x && board[i][y] == board[x][y])
             return false;

     a = (x/3)*3; b = (y/3)*3;
     for (i = 0; i < 3; i++)     //check for duplicates in 3x3 region
          for (j = 0; j < 3; j++)
               if ((a + i != x) && (b + j != y) && board[a+i][b+j] == board[x][y])
                   return false;

     return true;
}
```

Fig. 9. Check method