An Implementation of Sudoku-Solver using Randomized Algorithm

Parag H. Dave
H. B. Dave

July 15, 2007

Abstract: The popular Sudoku puzzle gives an opportunity to algorithm developers to try out various approaches. The problem is known to be NP-hard. We have designed and implemented a Randomized, Las Vegas type algorithm to solve a valid 9×9 Sudoku puzzle. The paper describes its design and performance.

Key words: Sudoku, randomized algorithm, Las Vegas.

1 Introduction

In recent years a number game or puzzle named "Sudoku' has become very popular with young and old alike. For lay persons its attraction is in its easy to understand rules coupled with intellectual challenge from very elementary to very high level. One can find commuters busy solving these puzzles while commuting to or from their work-place, in almost all countries of the world.

To a mathematician and computer scientist, the puzzle provides a base to investigate various algorithms to solve it. Basically it is an assignment problem and the problem of finding solution to a general Sudoku is known to be a NP-hard. Various approaches to the solution are tried, including pure logical reasoning, backtracking search of the solution space (using Donald Knuths Dancing Links), Simulated Annealing, Diff algorithm, etc. There is at least one web-based group of serious "Sudoku'ers", discussing thread-bare various issues, using their own jargon.

Writing a computer program to solve a general Sudoku is a non-trivial project. A number of "Sudoku generators" are also announced on the Internet.

The standard 9×9 Sudoku is quite well-known – given a partially filled 9×9 board, the player is required to fill in empty squares with digits 1 to 9, such that certain constraints are satisfied. Each row column and the 9 3×3 sub-squares should have exactly one instance of digits 1 to 9, each square containing, of course, exactly one digit.

In this paper we describe our implementation of a 9×9 Sudoku Solver, written in C language. It works in two stages – stage 1 uses purely logical rules, which solves most of the simpler puzzles; stage 2, which takes over if stage 1 is not successful, uses a Randomized (Las Vegas type) algorithm, with search

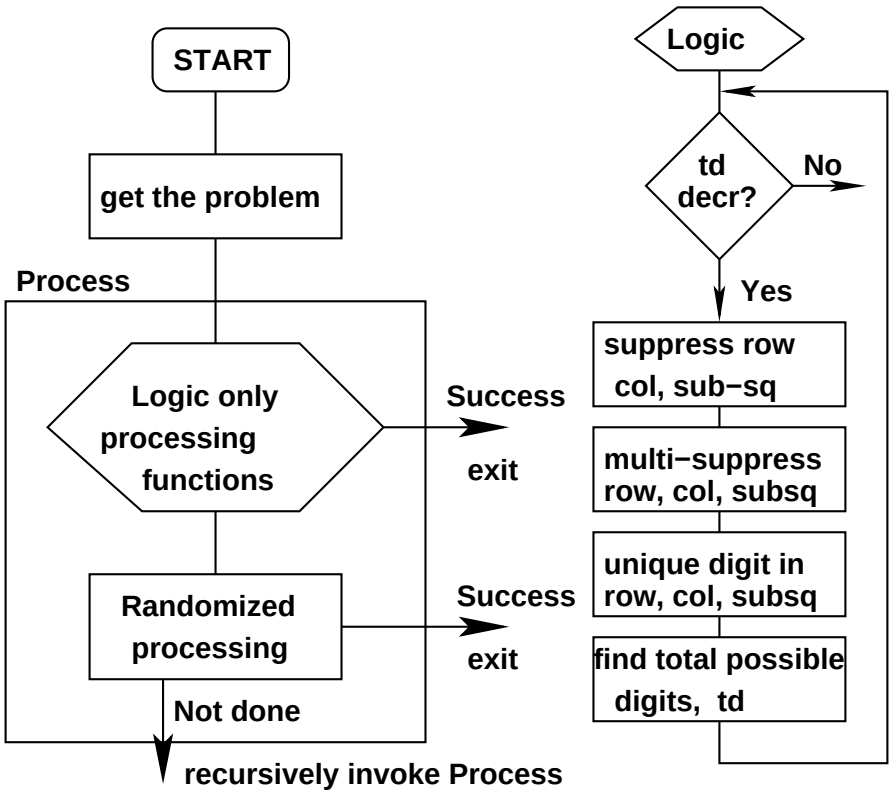

Figure 1: Flow-chart - 1

pruning, for cracking the tougher nuts. There are some interesting aspects of our program, which are described subsequently.

We have tested our program with a large number of Sudoku puzzles. It gives a solution to newspaper-last-page variety of Sudoku very fast, using mostly stage 1 processing only. Most of the "pro" variety are hard and our program uses both stage 1 and stage 2 to give the answer. As a typical Las Vegas algorithm, it always gives correct answer, but running time is variable, with a very reasonable average time. The section on Testing and Results gives details.

2 Overview of the Program

The program consists of a number of C functions, each implementing some specific operation. In stage 1 (logical processing) basic idea used is: if there is any cell rows[r][c], whose digit d_{rc} is fixed-up, either from initial input or intermediate processing, then d_{rc} can not appear anywhere else in row r and column c and also in the 3×3 sub-square. During all its operation, the program keeps track of which digits can be currently allowed in each of the 81 squares. It does this by systematically canceling digits which are not allowed in various squares. See the code listing for the details.

The program starts with a board will all digits allowed in all the 81 squares.

After reading in the problem, the given "starter" digits are *fixed* in their respective squares. It then applies stage 1 processing. In relatively simple puzzles, this is sufficient to get a solution. If logic processing can not *fix* all the 81 squares, then stage 2 takes over and uses a Randomized algorithm to assign trial digit values in un-fixed cells, (hopefully, stage 1 has introduced a few more fixed squares.) In case stage 2 find that it can not go ahead and fix all the 81 square, the whole process is repeated recursively, because we can view the board with its currently fixed up squares as if it a simpler puzzle input. See Fig. 1 and 2.

3 Major Functions

The following are major functions used.

int init_problem(int flag, char *problem) – initialize the board from the problem file.

int supress_row_fixed() – Cancel in each row digits which are fixed in some square in that row.

int supress_col_fixed() – Similar to above, for a column.

int supress_ssq_fixed() – Similar to above, for a 3×3 sub-square.

int digit_count(CELL c) – find number of allowed digits in this square.

int totdigit_count() – find total number of allowed digits in the whole board. This is used as a measure of efficacy of logical processing.

int mark_fixed() – If a previous pass has left any square with only one allowed digit, fix it.

int check_all_fixed() – find how many squares are fixed in the board.

int unique_cell_row() – if there is only one square having possible a particular digit in a row, then it should be fixed there.

int unique_cell_col() – Similar to above, for column.

int unique_cell_ssq() – Similar to above, for sub-square.

int multi_suppress_col() – If there are exactly two squares in a col, having the same allowed two digits (2-D), those can not appear anywhere else in the col.

int multi_suppress_row() – Similar to above, for row.

int validate() – check the number of duplicate single digits in rows and columns, it should be 0.

int list_pairs(CELL rows, NODE trylist) – prepare a list of 2-D squares and their contents, for use by the randomized portion of the program.

int logic() – a higher level function implementing the logical rules method for possible solution.

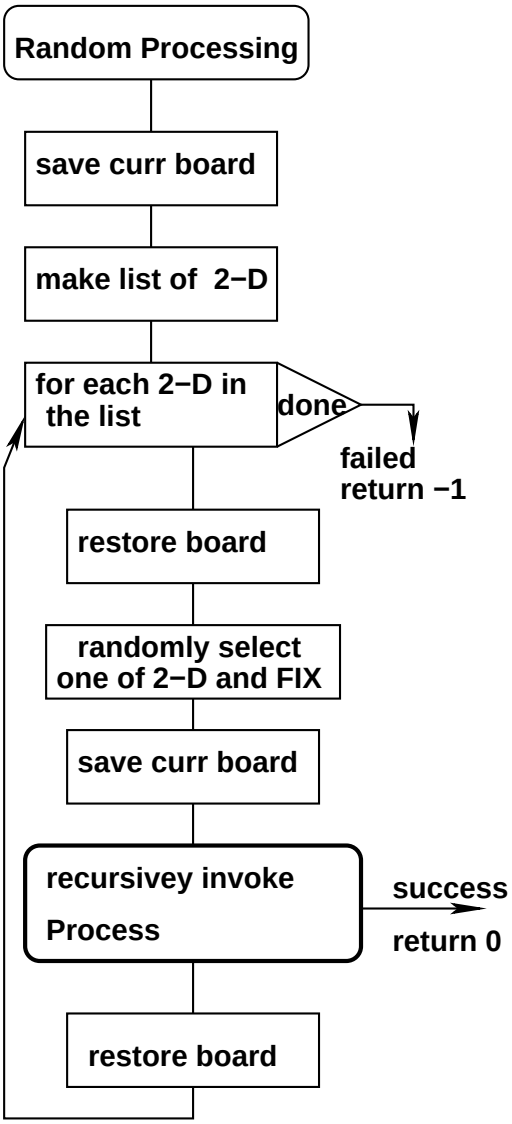

Figure 2: Flow-chart - 2

int process() – a higher level recursive function implementing the overall algorithm.

int main(int argc, char argv[ )] – sets up random number generator, reads problem file name and outputs final results.

4 Testing and Results

We tested our program with a large number of usual puzzles which appear in newspapers, magazines, etc. Also, we tested further with puzzles generated by a sudoku generator being written by us (under further development and testing). Also, we obtained 9 sudoku puzzles said to be really hard, and tested the program with them. The results are given below. Each puzzle was solved 20 times and the running times were recorded. The detailed histogram of really hard puzzles out of them (**hard1** and **mostdiff1**) show a characteristic Las Vegas behaviour.

4.1 Test Puzzles

In the following, we give results of 20 runs of some known hard Sudoku puzzles. They are given in a single line format, with a 0 representing an empty square. Times are in seconds. The average is over 20 runs.

4.1.1 hard1

```
700000019460190000000682704090000007000300405006700000001000000200740000002003 00
```

Solution Times (seconds):
max = 0.886007, min = 0.001161, avg = 0.166623

4.1.2 hard2

```
001080604037600000500000000000005000006010800000400000000000030000075208020907 00
```

Solution Times (seconds):
max = 0.086961, min = 0.001442, avg = 0.019691

4.1.3 hard3

```
000002040007040030006000109080009000309060805000700010605000400010080200070300000
```

Solution Times (seconds):
max = 0.001825, min = 0.001037, avg = 0.001390

4.1.4 hard4

```
080560010073000000400000002000200300008070900002003000100000009000000170060035080
```

Solution Times (seconds):
max = 0.019728, min = 0.000854, avg = 0.005863

4.1.5 hard5

```
000700050029050000057000840500004000006090300000500008084000710000040560090006000
```

Solution Times (seconds):
max = 0.016627, min = 0.000893, avg = 0.003149

4.1.6 hard6

```
070200000003590200008001040400000030200030007060000009090300800006054100000009060
```

Solution Times (seconds):
max = 0.001620, min = 0.000752, avg = 0.000884

4.1.7 hard7

```
000230050000905400000080093030000049100003007520010008960300000004000000000792000
```

Solution Times (seconds):
max = 0.001452, min = 0.000833, avg = 0.001004

4.1.8 hard8

```
000230050000905400000080003830020049109803507500010068960340000004000000000792000
```

Solution Times (seconds):
max = 0.002051, min = 0.000645, avg = 0.000778

4.1.9 mostdiff1

```
370600000009000000060020180000005000020010090000400000016090070000000500000007042
```

Solution Times (seconds):
max = 1.460041, min = 0.001661, avg = 0.396540

4.1.10 A typical simple puzzle

```
000065003002400000000100840090850100001002004065001000406000200000090076030000000
```

Solution Times (seconds):
max = 0.004, min = 0.0039, average = 0.00397.

4.1.11 A suite of 250 puzzples

We also run about 250 puzzles with, what are known as "naked triplets only". The histogram of timing is shown in the Fig. 7. As seen from the histogram, more than 208 of the 250 puzzles were solved in time less than 0.067 sec, in fact most of them were solved within a few milliseconds.

4.1.12 A suite of 95 puzzples

We run 95 known difficult puzzles, what are known as "top95", four times. The histogram of timing is shown in the Fig. 8. As seen from the histogram, more than 320 of the 380 puzzles were solved in time less than 6 sec, in fact most of them were solved within a small fraction of a second.

Solution Times (seconds):
max = 291.99081, min = 0.000939, average = 6.6307567.

5 Discussion and Conclusion

For really hard problems, the algorithm shows characteristic behaviour of a Las Vegas algorithm, where many solutions are obtained on relatively small times and a few take longer times. It will be interesting to compare algorithm presented in this paper with others to solve sudoku problem.

A C implemented Randomized algorithm is presented to solve 9×9 sudoku puzzles. As a supporting activity, sudoku generator is being developed and tested, to be presented in future.

[Esler2006] http://www.news.cornell.edu/stories/Feb06/Elser.sudoku.lg.html

Figure 3: Histogram of hard1 runs

Figure 4: Histogram of hard2 runs

Figure 5: Histogram of hard4 runs

Figure 6: Histogram of mostdiff1 runs

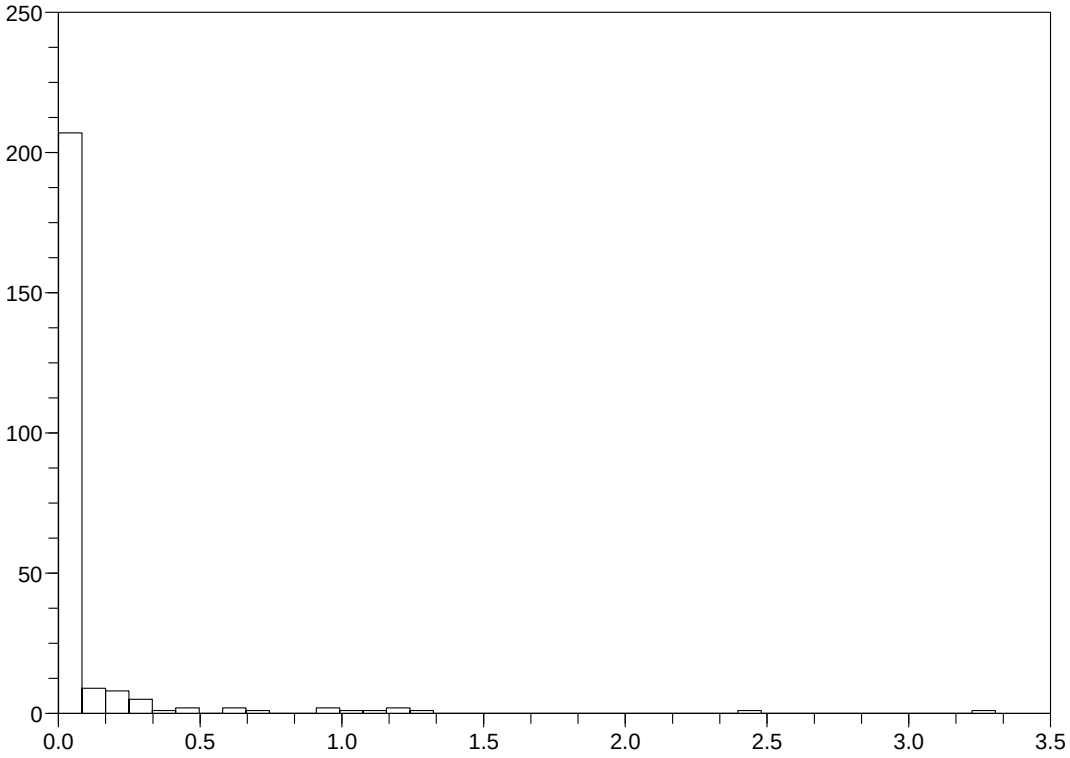

Figure 7: Histogram of 250 runs

Figure 8: Histogram of top95 runs