Computacion evolutiva: Tarea 6

Jordi Colomer

May 13, 2013

Abstract

In this work we will study the performance of different genetic algorithms designed to solve the sudoku problem. We will design different representations and crossover/mutation operators and will test them against our set of problems. We will also tune the parameters to optimize the algorithm.

1 Introduction

Sudoku is a popular puzzle game, consisting of a 9x9 grid of cells, where each cell can either contain a number from 1 to 9 or be missing. The objective of the game is to fill all blank cells with numbers from 1 to 9 in a way that each column, row and 3x3 square contains no repetitions.

Sudoku is an NP-complete problem, which means that solving it involves some kind of search. [1]

Sudoku problems have been successfully solved by using constraint satisfaction algorithms. [7]. But can be solved by using other methodologies as we will see in this work.

The problem of solving a Sudoku can be transformed into an optimization problem, where the value that we are trying to optimize is an heuristic measure that tells us how far away our current solution is from the goal. From this point of view the Sudoku problem can be solved by using any of the optimization techniques, like for instance Genetic Algorithms. In this assignment we are going to implement a GA that is able to solve Sudokus. In the second section we are going to describe the details of the design and implementation of the system. In the third section we are going to evaluate the performance of different configurations. Finally we will conclude our work in the forth section. We also include in the appendix the first design of the algorithm that even though it fails to solve the sudoku problem we think it's worth mentioning as it's been a part of our work.

2 Design and implementation

A sudoku problem can be represented as an array of 9x9=81 integer values representing each one of the cells in the grid. We will only use the numbers from 1 to 9, and 0 to represent the cells in blank. Similarly, we will represent a solution to a problem as an array of 81 integer values. A solution to a problem is valid if the non-blank cells of the problem appear in the solution in the same positions. This constraint is strict and will remain true at any point of the execution. Other constraints for a valid solution will be non-strict, and will be a part of the heuristic, as we will see later in this text.

The fitness value, or the value that we are trying to optimize or minimize, has three components. The first component counts the number of repetitions of each column. Another way to express it is to take the size of the set composed by the elements of each column and subtract that value from 9. In the ideal case where there are no repetitions in any of the columns this value will be 0. The second component is the same value but this time for the rows, and the third and last component would be the same value for the 3x3 squares. We can express this mathematically as such:

$$fitness = columnFitness + rowFitness + squareFitness$$

$$columnFitness = \sum_{i=1}^{9} 9 - |\bigcup_{j=1}^{9} A_{ij}|$$

$$rowFitness = \sum_{j=1}^{9} 9 - |\bigcup_{i=1}^{9} A_{ij}|$$

$$squareFitness = \sum_{a=1}^{3}\sum_{b=1}^{3} 9 - |\bigcup_{i=1}^{3}\bigcup_{j=1}^{3} A_{a*3+i,b*3+j}|$$

Where A_{ij} is a matrix representation of the solution.

We initialize the population randomly with two constraints: the known values of the problem must appear in the solution, and each 3x3 square must contain no repetitions. We can easily construct such individuals by filling in the blanks with values randomly drawn from the set of numbers that don't appear in the square.

We define the crossover as a simple one point crossover, but we only allow the point to be a multiple of 9. This restriction enforces the offspring to inherit each square from one parent only. This operation mantains the fullfillment of the no repetitions in a square restriction. That is, if the parents fullfill the restriction, the offspring generated with this crossover operation will also fullfill it.

We have two mutation operators, that are selected with 50% chances each. The first one is a swap of two randomly selected elements from the same square. The square is selected randomly. We don't allow selecting elements that are known values from the problem. The second one is the same as above but with a three swap or rotation of three elements. The mutation rate is 0.5. It is easy to see that the mutation operator also mantains the satisfaction of the no repetitions square restriction. And since our initial population satisfies the restriction, and all the operations mantain this condition, then all the generated individuals will comply with the rule, and the search algorithm must only focus on the column and row rules. At this point we can actually safely remove the squareFitness part of the equation as we know beforehand that the value will always be 0.

For the parent and survivor selection we choose a tournament method as it is suggested for this problem in many research papers. We choose k = 5, but we experiment with other values. We choose $\mu = 400, \lambda = 200$ and number of generations 1000. The exit condition is either a) we reach the maximum number of generations, b) we have already found a solution with 0 fitness, or c) we have spent a maximum number of generations without any improvement. We set this number to 100.

In the next section we will run some experiments with this and other configurations and will give measures that will evaluate their performance.

3 Results

In this section we will describe a series of experiments that we have performed. We will try different configurations to test their impact on different performance measures. For each configuration we will run the algorithm for a total of 100 instances. For each instance we will capture the fitness or distance of the best individual found, the number of generations, the time elapsed, and the nodes explored or number of individuals visited. Then we will average all this values over the 100 instances, and we will include the number of successful instances, that is, where the distance equals 0. In table 1 we have summarized all this information. Each row corresponds to a different configuration. The last column contains the parameters that define the configuration. The base configuration described in the previous section is used as the default values. The sudoku problem used is the one given in the file Sudoku_nivelfacil.txt.

The first row of the table corresponds to the base configuration itself, represented by the empty string. It successfully solves the puzzle in 2 occasions out of 100. The mean number of generations per instance is about 156. This number is much smaller than maxGenerations, that we have set to 1000. The reason is that we are exiting most of the time for not being able to improve our best solution found over maxGenerationsWithoutImprovement (100) generations.

In our next experiment we will increase maxGenerationsWithoutImprovement up to 500. As it is expected we observe an increase on the number of generations, time, and nodes, as well as success (reaches 3). The average best distance decreases almost by one.

In the third experiment we choose to reduce the population and λ by half. There are no surprises in the results either, we explore less nodes, so we need less time, and we find worse fitted individuals (only one out of 100 being the

n	success	distance	generations	time (s)	nodes	configuration
01	2	6.36	156.88	18.59	31976	
02	3	5.64	622.35	77.38	125062	maxGenerationsWithoutImprovement=500
03	1	7.0	164.0	08.43	16700	mu=200,lambda=100
04	2	6.36	156.88	19.84	31976	maxGenerations=500
05	0	7.02	327.8	14.59	66160	parentSelection=fitnessProportional,survivorSelection=fitnessProportional
06	1	5.95	215.5	17.08	43700	parentSelection=ranking,survivorSelection=ranking
07	1	6.2	161.95	16.93	32990	parentSelectionTournamentk=3,survivorSelectionTournamentk=3
08	1	6.91	145.85	30.42	29770	parentSelectionTournamentk=7,survivorSelectionTournamentk=7
09	1	7.48	160.66	22.95	32732	mutationRate=.3
10	3	6.19	151.99	27.30	30998	mutationRate=.7
11	37	3.19	102.82	10.76	21164	rep=sudoku1a
12	24	3.98	123.89	14.65	25378	rep=sudoku2a
13	0	7.54	154.6	18.33	31520	rep=sudoku3a
14	0	6.98	167.44	19.63	34088	rep=sudoku4a
15	1	6.78	160.21	19.10	32642	rep=sudoku5a
16	50	2.32	86.48	09.44	17896	mutationRate=.7,rep=sudoku1a
17	40	2.55	106.76	12.59	21952	mutationRate=.7,rep=sudoku2a
18	59	1.83	76.06	09.10	15812	mutationRate=.8,rep=sudoku1a
19	42	2.47	101.21	12.55	20842	mutationRate=.8,rep=sudoku2a
20	50	1.98	80.87	09.85	16774	mutationRate=.9,rep=sudoku1a
21	40	2.52	96.13	11.45	19826	mutationRate=.9,rep=sudoku2a
22	56	1.96	80.36	10.40	16672	mutationRate=1,rep=sudoku1a
23	43	2.34	92.03	11.15	19006	mutationRate=1,rep=sudoku2a
24	59	1.83	76.06	09.10	15812	mutationRate=.8,rep=sudoku1a
25	42	2.47	101.21	12.55	20842	mutationRate=.8,rep=sudoku2a
26	1	7.26	153.44	18.90	31288	mutationRate=.8,rep=sudoku3a
27	0	6.67	148.46	20.02	30292	mutationRate=.8,rep=sudoku4a
28	0	5.99	154.23	19.07	31446	mutationRate=.8,rep=sudoku5a
29	2	6.1	159.4	19.26	32480	mutationRate=.8,rep=sudoku1b
30	53	1.52	81.85	10.11	16970	mutationRate=.8,rep=sudoku2b
31	0	7.11	154.15	18.44	31430	mutationRate=.8,rep=sudoku3b
32	1	5.98	148.97	17.83	30394	mutationRate=.8,rep=sudoku4b
33	1	5.6	152.48	17.81	31096	mutationRate=.8,rep=sudoku5b
34	17	6.51	147.6	16.48	30120	mutationRate=.8,rep=sudoku1c
35	1	7.14	155.39	18.57	31678	mutationRate=.8,rep=sudoku2c
36	0	4.72	150.97	17.31	30794	mutationRate=.8,rep=sudoku3c
37	1	6.41	156.01	19.27	31802	mutationRate=.8,rep=sudoku4c
38	1	5.88	148.87	19.28	30374	mutationRate=.8,rep=sudoku5c
39	0	11.18	225.76	33.78	45752	fitnessFunction=sudoku,mutation=swap,recombination=GOX
40	0	10.0	194.5	84.18	39500	fitnessFunction=sudoku,mutation=swap,recombination=GOX,rep=sudoku1a
41	0	6.0	199.0	82.82	40400	fitnessFunction=sudoku,mutation=swap,recombination=GOX,rep=sudoku2a
42	0	12.03	216.97	66.42	43994	fitnessFunction=sudoku,mutation=swap,recombination=GOX,rep=sudoku3a
43	0	12.45	225.2	79.00	45640	fitnessFunction=sudoku,mutation=swap,recombination=GOX,rep=sudoku4a
44	0	11.37	226.38	75.47	45876	fitnessFunction=sudoku,mutation=swap,recombination=GOX,rep=sudoku5a
45	0	13.3488372093	218.697674419	74.15	44339	fitnessFunction=sudoku,mutation=swap,recombination=GOX,rep=sudoku1b
46	0	10.431372549	227.901960784	76.60	46180	fitnessFunction=sudoku,mutation=swap,recombination=GOX,rep=sudoku2b
47	0	13.06	222.74	72.89	45148	fitnessFunction=sudoku,mutation=swap,recombination=GOX,rep=sudoku3b
48	0	10.7	223.2	72.23	45240	fitnessFunction=sudoku,mutation=swap,recombination=GOX,rep=sudoku4b
49	0	10.09	234.81	76.25	47562	fitnessFunction=sudoku,mutation=swap,recombination=GOX,rep=sudoku5b
50	0	14.3846153846	211.884615385	70.47	42976	fitnessFunction=sudoku,mutation=swap,recombination=GOX,rep=sudoku1c
51	0	13.23	232.38	87.01	47076	fitnessFunction=sudoku,mutation=swap,recombination=GOX,rep=sudoku2c
52	0	11.49	222.29	86.19	45058	fitnessFunction=sudoku,mutation=swap,recombination=GOX,rep=sudoku3c
53	0	12.04	222.77	241.42	45154	fitnessFunction=sudoku,mutation=swap,recombination=GOX,rep=sudoku4c
54	0	11.41	223.94	73.06	45388	fitnessFunction=sudoku,mutation=swap,recombination=GOX,rep=sudoku5c

Table 1: Results for different configurations.

Alias	Difficulty	Representation
sudoku1a	very easy	.....93246.3.7..9.4...856...4.79...62.1...4.75...42.1...612...5.5..6.8.28724.....
sudoku2a	easy	.6...2.7.3.94.5..87.5..61.94...29....86...73....36...11.82..3.66..9.38.4.4.6...2.
sudoku3a	medium	.9.3...6....9...87..64.195..25......7.......9......63..528.41..81...5....3...6.2.
sudoku4a	hard	..1.78.....79..2.....2...34..63..4.7.3.....6.7.4..25..64...5.....9..41.....78.6..
sudoku5a	fiendish	...36...45.....798..9....6....89...5.7.6.1.4.9...42....2....9..198.....67...36...
sudoku1b	very easy	.493.2..1..8694....2.1.......4...876.........591...2.......1.5....4863..1..7.592.
sudoku2b	easy	..58.12...715...4.9...6..851...7..93..49.67..59..4...861..2...9.4...831...37.98..
sudoku3b	medium	4.6.5.......2...6..8..7.3..1.87.39...6.....7...36.28.4..2.3..1..9...5.......6.2.8
sudoku4b	hard	48.....27...6.4...3.5...9.6..75.21...5.....7...39.16..6.2...4.1...8.6...17.....65
sudoku5b	fiendish	..4...8.......3.4193.71...........25572...13614...........67.5442.9.......6...3..
sudoku1c	very easy	..91.56..3.8...75.....82.......642.32.......48.729.......73.....92...3.1..64.95..
sudoku2c	easy	4..1.....57..9......8...23..8.3.7..56..8.9..13..5.1.7..12...4......5..89.....4..6
sudoku3c	medium	13.....45...2.6...7.8...3.9..39.74...5.....8...45.17..4.1...5.3...6.4...27.....14
sudoku4c	hard	4.......6.6.895.3..8.....2...35.46...4.....1...12.87...2.....7..1.746.9.7.......5
sudoku5c	fiendish	9..7.6..2.7.3.9.1...5...7....94.78..7.......4..39.21....2...5...9.8.5.2.8..1.3..6

Table 2: Representation for the sudokus used for the evaluation.

correct answer).

Our selection method for both the parents and the survivors is the tournament method as suggested by the literature. We are going to try two different methods in the next two experiments. These are the fitnessProportional, and the ranking methods. The former selects the individuals with a probability that is proportional to the normalized fitness of the individual. The later selects the individuals with a probability that is a function of the position of the individual in a ranking by fitness, therefore in this method the absolute value of the fitness is not important, what is important is the relative position in relation to the other individuals. Both methods perform worse than the tournament method, thereby confirming what has been written in the literature. The ranking method (with one successful run) performs a little better than fitnessProportional (with none).

At this point we are certain that we should use tournament as the selection mode in our algorithm. We will experiment with different values of k (3 and 7). We find that in neither of the two experiments we outperform the base configuration of (k=5).

The next two experiments (9 and 10) will test the effects of changing the mutation rate, we try the values 0.3 and 0.7. We get a bigger success ratio for the value 0.7 (3% success).

In order to complete our evaluation we will generate a total of 15 new sudoku puzzles and we will test the algorithms against them. The sudoku problems have been generated automatically by using the software *sudoku* of the ubuntu linux distribution. This software allows to generate puzzles of 5 different levels of difficulty (very easy, easy, medium, hard and fiendish). We are going to generate 3 puzzles of each kind. Therefore we will obtain 3 panels with 5 puzzles of different difficulties each. In table 2 we show each one of the puzzles along with the difficulty command switch used to generate it, and its representation in the format used in the configuration file. The representation is the sequence of integers of the puzzle row by row from top to bottom. Emtpy cells are represented with a dot. Table 2 includes an alias for easy reference of the sudokus, and these alias are used in table 1.

The experiments 11 to 15 correspond to the tests of the base configuration against the first panel of sudokus. We notice pretty high success rates for the two easiest puzzles (37% and 24%). The success rate drops down to zero for the medium and hard problems, but surprisingly the algorithm has been able to solve the hardest puzzle (fiendish) with a 1% success rate.

From previous experiments we know that probably the value 0.5 for the mutationRate wasn't the optimal one, and the results obtained suggest that the optimal value might be higher. The experiments 16 to 23 are targeted to address this question. We have selected the two easiest problems of the first panel. The success rate in those problems are much greater than 0 and therefore we will be able to measure the effects of changes in the configuration in a way that is more robust and less affected by random chance (it would be difficult to make a conclusion when all parameter changes yield results close to 0). We choose to test the algorithm with mutationRate set to values 0.7, 0.8, 0.9 and 1.0. The value that obtains the best ratios is 0.8 (59% and 42%) although it doesn't dominate the results obtained for the value 1.0 (56%,43%). Since the sum of success ratios is greater for the value 0.8 we will consider this to be the optimal value and we will use it from now on.

The next 15 (24 to 38) experiments correspond to the tests for each one of the generated sudokus using the optimal configuration found so far, that consists on the base configuration with mutationRate set to 0.8. We find the correct answer for 11 problems and we fail to solve it for 4. In all three panels we find solutions for the very easy and easy difficulty levels, and often (but not always) with relatively high success ratios. We fail to solve two mediums, one hard and one fiendish problems. We find a correlation between difficulty level and success ratio, but the correlation seems to be far from perfect. This could be due a poor implementation of the difficulty command switch of the sudoku generator.

The following experiment shown in the list (39) actually corresponds to the first experiment, and corresponds to a different representation, crossover and mutation operators. The performance is the worst of all previous experiments, and the details of the implementation are given in the first appendix.

The last 15 experiments (40-54) correspond to tests with the first implementation against our sample of 15 sudokus. The results confirm that the approach used is not a good one, since none of the tests are able to correcty solve the puzzle.

4 Conclusions

In this work we have seen how genetic algorithms are capable of solving any problem that can be transformed into and optimization problem. In the particular case of the sudoku puzzle, GA has proven to be capable of solving some problems. It seems though that this might not be the best available methodology for this particular problem and constraint satisfaction algorithms might perform better [7]. GA is a general problem solver, and although better solutions might exist it performs reasonably well in many cases.

We have also learned that choosing the "right" representation and operators is a mandatory step, and not doing so might lead to very poor results. In the first Appendix we will give an example of a bad design. It doesn't seem to exist a methodology for finding a good representation and operators, which makes this task seem more like an art than a science.

It is also an important step to tweak the parameters to optimize the algorithm. In order to accomplish that we need to measure the goodness of an algorithm. It is usually enough to generate a test case with a few examples of the problems that we want to solve, and take the success ratio as a measure of goodness for the algorithm.

The average best distance given in our test experiments could be an indicator of the difficulty of the sudoku puzzle. Therefore the algorithm could be used to measure the difficulty of a sudoku puzzle automatically. It could even be used to generate puzzles of a certain difficulty by generating them first using the generator and filtering them later by using this measure.

5 Appendix A: The first try

Our initial idea was to represent the solution to a sudoku problem as an array of integers with all the numbers of the sudoku that were unknown beforehand. The idea was to reduce the representaton size, and hence the search space, by not including in the representation the numbers that where already known. Each element of the array would be mapped to a position of the grid in a sequential manner, but skipping the positions of the grid already occupied by previously known numbers.

The initial state is a random sequence of the missing numbers in the sudoku. This is easy to construct given the known numbers, and the fact that each complete sudoku must have each number from 1 to 9 repeated exactly 9 times.

5.1 GOX crossover

In order to maintan the number of repetitions of each number, we need to choose special operators. In [4] a crossover for permutations with repetition is proposed that "respects the repetition-structure of a certain permutation and preserves the coded characteristics of solutions as far as possible to guarantee the inheritance-feature". We have two parents parent1 and parent2.

```
parent1:BABBCACCBA
parent2:ABBACABCBC
```

We randomly select a substring of the chromosome of p2 by choosing a random point in the chromosome string and a random length of the substring with values between a third and a half of the total length. The substring might actually overlap the string.

```
parent1:BABBCACCBA
parent2:ABBACABCBC
mark2:    ----
```

We assign to each element of the string a number corresponding to 1 + the number of repetitions of the element seen so far.

```
parent1:BABBCACCBA
index1: 1123122343
parent2:ABBACABCBC
index2: 1122133243
mark2:    ----
```

We mark in parent1 the letter-number pairs that are marked in parent2.

```
parent1:BABBCACCBA
index1: 1123122343
mark1:    ---   -
parent2:ABBACABCBC
index2: 1122133243
mark2:    ----
```

A new string is formed by inserting the selected substring of parent2 into parent1. The insertion point is right after the position in parent1 of the first element (letter-number pair) of the substring in parent2.

```
child: BABBCA ACAB CCBA
index: 112312      2343
mark:     ---         -
```

The last step consists in eliminating the marked elements. The resulting child will always be a permutation of any of the parents.

```
child: BABACABCCB
```

In the case that the substring wraps around the string, the process is even simpler, lets consider that we have the following two parents with the indexes already calculated. The mark for parent2 is an example of a substring wrapping around the string. Mark 1 is calculated as seen before.

```
parent1:BABBCACCBA
index1: 1123122343
mark1:  --     --
parent2:ABBACABCBC
index2: 1122133243
mark2:  --      --
```

The last step is to create the child by copying parent 1, eliminating the elements marked in parent1 and inserting the elements marked in parent 2 in the same position as they appear.

```
child: ABBBCACABC
```

5.2 Fitness

The fitness is the same as the one seen before in this work. The fitness is composed by three elements (columns, rows and squares), and we said in the previous algorithm that the third part of the equation (the part of the squares) was not necessary, because the condition of each square of not having any repetition was already being enforced at any time during the search. This is no longer true, and therefore we need all three elements of the equation for the fitness value.

5.3 Conclusion bis

This algorithm has been shown to give pretty bad results in comparison to the new algorithm. The reason might be that the fitness value is much more complex and therefore much more difficult to fullfill. In the newer version is designed so that all the generated individuals will fullfill the squares part of the fitness expression, simplifying in a great deal the fitness function and the search. And even though the search space is bigger in the new design, there are restrictions (like not being able to change known numbers) that might make this difference less important.

6 Appendix B: Running the program

The following command will run the main program.

```
java main.Main100
```

It will read the parameters from the file default.properties. There are two lines in the properties file that reference to the input sudoku file (insudokufile) and the output sudoku file that will contain the solution of the problem in the case that it is found (outsudokufile). The formats of the input and output files is the one used in the files *Sudoku_nivelfacil.txt* and *Sudoku_nivelfacil_Soluc.txt*. The configuration file also accepts specifing a sudoku problem directly by using the parameter *rep*.

The program runs the algorithm a total of 100 times with different seeds and gives for each one of the instances the following measures: fitness or distance of the best individual found, the number of generations, the time elapsed in millisecods, and the nodes explored or number of individuals visited. This is a very easy to implement strategy to mantain the diversity during a search, and should be considered as just one big diversified search.

References

[1] Das, Kedar Nath, et al. “A Retrievable GA for Solving Sudoku Puzzles”. Technical Report, http://www. cse. psu. edu/ sub194/papers/sudokuTechReport. pdf, 2012.

[2] Moraglio, Alberto, Julian Togelius, and Simon Lucas. “Product geometric crossover for the sudoku puzzle.” Evolutionary Computation, 2006. CEC 2006. IEEE Congress on. IEEE, 2006.

[3] Mantere, Timo, and Janne Koljonen. “Solving and rating sudoku puzzles with genetic algorithms.” New Developments in Artificial Intelligence and the Semantic Web, Proceedings of the 12th Finnish Artificial Intelligence Conference STeP. 2006.

[4] Bierwirth, Christian. “A generalized permutation approach to job shop scheduling with genetic algorithms.” Operations-Research-Spektrum 17.2-3 (1995): 87-92.

[5] http://fakeguido.blogspot.de/2010/05/solving-sudoku-with-genetic-algorithms.html

[6] http://blog.refu.co/?p=915

[7] http://norvig.com/sudoku.html