Genetic Algorithms with Local Optima Handling to Solve Sudoku Puzzles

Firas Gerges
Department of Computer Science and Mathematics
Lebanese American University
Byblos, 1h401 2010, Lebanon
+961-76-029164
firas.gerges@lau.edu

Germain Zouein
Department of Computer Science and Mathematics
Lebanese American University
Byblos, 1h401 2010, Lebanon
+961-76-700554
Germain.zouein@lau.edu

Danielle Azar
Department of Computer Science and Mathematics
Lebanese American University
Byblos, 1h401 2010, Lebanon
+961-9-547254#2408
Danielle.azar@lau.edu.lb

ABSTRACT

Sudoku is a popular combinatorial number puzzle game and is widely spread on online blogs and in newspapers worldwide. However, the game is very complex in nature and solving it gives rise to an NP-Complete problem. In this paper, we introduce a heuristic to tackle the problem. The heuristic is a genetic algorithm with modified crossover and mutation operators. In addition, we present a new approach to prevent the genetic algorithm from getting stuck in local optima. We refer to this approach as the "purge approach". We test our algorithm on different puzzles of different difficulty levels. Results show that our algorithm outperforms several existing methods.

CCS Concepts

•Mathematics of computing → Combinatorial optimization • Theory of computation → Evolutionary algorithms • Computing methodologies → Heuristic function construction • Computing methodologies → Genetic algorithms • Computing methodologies → Learning paradigms • Applied computing → Enterprise computing

Keywords

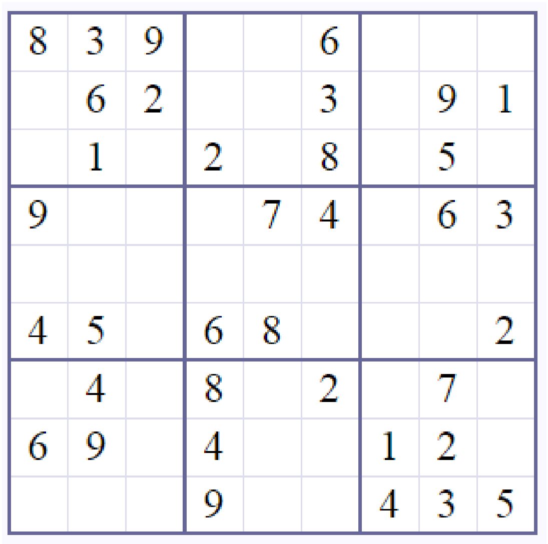
Combinatorial Problems; Genetic Algorithm; NP-Complete Problem; Sudoku; Puzzles; Games.

1. INTRODUCTION

The Sudoku is a puzzle game based on a 9x9 grid in which a player inserts numbers from 1 to 9 in empty cells. The grid is divided into nine 3x3 sub-grid (each called a block). The game starts with a given Sudoku grid containing some filled cells (called clues). The player has to fill the rest of the cells in such a way that no row, column or sub-grid contains duplicates. The number of clues determines the difficulty level of the Sudoku instance. Figure 1 shows an example of a Sudoku puzzle generated via www.websudoku.com.

ICCAI 2018, March 12–14, 2018, Chengdu, China
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6419-5/18/03…$15.00
https://doi.org/10.1145/3194452.3194463

Nowadays, Sudoku puzzles are widely spread as a single player logic game online and in newspapers. Solving this game can be seen as a combinatorial optimization problem [1]. Given its growing popularity of the problem, there have been many attempts to develop algorithms to solve it. In this paper, we present a genetic algorithm to solve Sudoku puzzles in an efficient way.



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

Figure 1. An example of an instance of a Sudoku puzzle.

2. LITERATURE REVIEW

Many papers tried to tackle the game through different techniques. In [1], Douglas presents a genetic algorithm to solve the game. The algorithm is tested on three problems. It solves two and fails to solve the harder one. Results show a big standard deviation in terms of generations needed to solve the problem. The work is an improvement trial over [2] where Mantere and Koljonen introduce another Genetic Algorithm which is tested on 10 difficulty ratings (including one "empty" Sudoku board). Results show that GA can always solve an empty puzzle with an average of 1400 iterations. Easy puzzles were all solved while higher difficulty levels had lower success. The number of iterations required also varied greatly, from 770 for the easiest puzzles to 25,000 for the hardest. The results also suggest that GA can be used to rate Sudoku puzzles, by checking the percentage of puzzles solved as well as the number of iterations used. Another genetic algorithm (GA) is presented in [3] where crossover is used to avoid collisions. In [4], the authors propose a GA with intensified search to tackle the problem. Results show that the proposed GA outperforms the generic GA in terms of the number of generations needed to solve the game. In [5], the authors acquire real-time images of Sudoku instances and performs image processing on them prior to solving them with GAs, simulated annealing and backtracking. When tested on thirty different instances of the problem, backtracking required the least time and successfully solved all instances. GA and simulated annealing also failed to solve complex examples. In [6], Mantere presents an improved hybrid combination of a Genetic Algorithm(GA) and ant colony optimization (ACO). Mantere compares his algorithm to a previous work of the same author using genetic algorithms, genetic algorithm-based cultural algorithm and hybrid GA/ACO [7]. The new approach shows to be faster than the previous one. In [8], the author tries to solve the Sudoku problem using the Harmony Search (HS) algorithm. The author applies the HS approach on two Sudoku puzzles (easy and hard) and is able to solve the easy example with different parameters but a different time range as well. In [9], the authors present another version of harmony search to solve the problem. The achieved improvement is in the number of iterations required to solve a puzzle. However, the authors based their comparisons on a single Sudoku puzzle. It is uncertain how their proposed method would perform on other puzzles.

Another meta-heuristic is presented in [10] where the authors used a CUDA parallelized version of particle swarm optimization. Experiments showed that results are very sensitive to the used parameters. Although the technique was able to solve multiple Sudoku examples, it needed 1.6 million iterations with a time in the order of 30 minutes. In [11], the authors consider using the Artificial Bee Colony (ABC) algorithm to solve Sudoku puzzles. The algorithm enforced uniqueness on a block-level when building the random population (during neighborhood search, block-level violations are dealt with by swapping the older violating cell with the previous correct value). Results show that this method can solve Sudoku puzzles in around 200-6000 cycles or 4-270 seconds on average depending on the level of difficulty. A comparison with GA also shows that ABC outperforms it on one particularly hard puzzle, though the comparison was not carried out on further ones. In [12], the authors propose a hybrid algorithm that combines Tabu Search (TS) and constraint programming. The proposed algorithm is compared to AC3-TS (a hybrid combining AC3 and Tabu Search), AC3-CS (which combines AC3 and cuckoo search) and Genetic Algorithms which are all implemented in [13]. AC3 is an arc consistency algorithm used in order to reduce the domain of Tabu Search. AC3 implements local consistency by defining properties that have to be satisfied by constraint programing. The comparison is based on the number of iterations for 13 difficulty subgroups, each consisting of 3 puzzles. The algorithm outperforms AC3-TS in needed iterations, though the comparison with AC3-CS and GA is incomplete: the comparison with GA only considers 1 out of 13 difficulty subgroups, for which the proposed algorithm and GA perform equally well; the comparison with AC3-CS covers 8 out of 13 difficulty subgroups, in which the proposed algorithm outperforms AC3-CS. A Backtracking approach is presented in [14]. The work used brute force to iterate over all possible number arrangements in a search space. The authors tested their developed algorithm on 30 Sudoku puzzles with different difficulty levels (in term of number of given clues). The reported average time needed to solve a Sudoku puzzle is 20.365 seconds.

3. GENETIC ALGORITHMS

Genetic algorithms are one type of evolutionary algorithms that mimics the natural selection process leading to biological evolution [15].The basic idea behind GA is to generate populations of chromosomes from previous ones until the desired solution is found. A chromosome encodes a solution to a problem. The evolution process happens through three major operators: selection, crossover and mutation. Selection consists of probabilistically choosing two chromosomes from the current population. These undergo crossover i.e. parts of them are exchanged then the resulting chromosomes undergo mutation whereby parts of them are changed. It is important to point out that the selection of the two chromosomes is proportionate to their fitness. The fitness value of the chromosome reflects the quality of the underlying solution. Crossover and mutation occur with a certain probability. Figure 2 shows the pseudocode of genetic algorithms.

```
GA:
Initialize  Population P
Until stopping_criterion met:
    G = null
   while  |G|≠ |P|
       1. (C1, C2) = Select (P)
       2. (Ch1, Ch2) = Crossover (C1, C2)
       3.  Ch1= Mutate (Ch1)
       4.  Ch2 = Mutate (Ch2)
       5.  Add Ch1 and Ch2 to G
    P = G
```

Figure 2. Pseudo-code for GA.

3.1 Instantiated Genetic Algorithm

The pseudo-code of the instantiated GA is shown in Figure 3.

```
Sudoku-GA:
Population P = Initialize Random Population
Generation G = P
Generation newG= null
While Penalty(Best)! =0 and iteration <
max_iteration:
    While |newG| < |G| :
        Parent1 = K-tournament (G)
        Parent2 = K-tournament (G-{Parent1})
        Child = Crossover (Parent1, Parent2)
        Mutate(child,µ)
        Mutate(Parent1, µ')
        Mutate(Parent2, µ')
        bestParent = Best (Parent1, Parent2)
        newG.add (bestParent)
        newG.add (child)
    newG.add (BEST_OF_G)
    G = newG
    Purge (G)
```

Figure 3. Our Instantiated GA.

3.1.1 *Encoding*

In our instantiation of the algorithm, a chromosome is a filled 9x9 grid. Figure 4 shows two different chromosomes generated randomly from the Sudoku instance shown in Figure 1. The black numbers are the fixed cells, and the red ones are the randomly placed numbers.

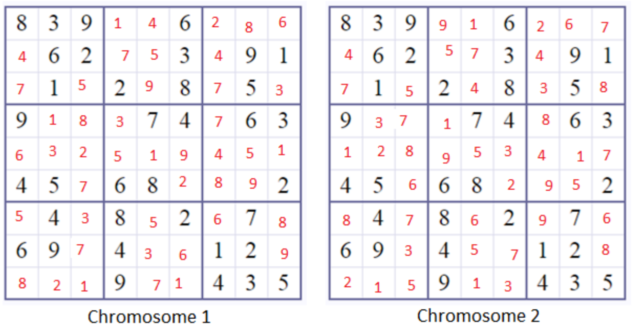


Figure 4. Two randomly generated chromosomes from the instance in Figure 1.

3.1.2 *Initial Population*

The GA starts by generating *n* random Sudoku grids from the initial instance which will be the input of GA. Generating a Sudoku grid consists of placing a random number between 1 and 9 in every empty cell. We do not allow duplicates in the same sub-grid. However, at this stage, it is possible for duplicates to occur in rows and/or columns.

3.1.3 *Fitness Function*

For each Sudoku grid, we define a penalty which is the count of duplicate numbers in the same row, column or sub-gird (block). Since, as initially stated, our implementation ensures that no duplicates are found in the same block when initializing the population, the penalty defines the number of duplicates in rows and columns (Equation 1).

$$Pe = \sum_{i=1}^{9}\sum_{j=1}^{8}\sum_{L=j+1}^{9}(x_{ij} == x_{iL}) + \sum_{i=1}^{8}\sum_{j=1}^{9}\sum_{L=i+1}^{9}(x_{ij} == x_{Lj}) \quad \text{Equation 1}$$

Where *i* and *j* represent the indices of the row and the column respectively.

Since, the objective is to minimize the penalty of the Sudoku grid, the fitness function is the inverse of the penalty (Equation 2).

$$\text{Fitness} = 1 / \text{Pe} \quad \text{Equation 2}$$

3.1.4 *Selection*

We use the K-tournament method as our selection technique whereby *k* chromosomes are picked randomly from the population and the fitter one is kept. The other *k*-1 chromosomes are thrown back into the population. This is repeated to select the second chromosome.

3.1.5 *Crossover*

Blocks are indexed from 1 to 9 starting from left to right. Our crossover operator consists of a block-by-block method where at each index *z*, the block at this index in the child is retrieved from the block (*z*) from one of the parents randomly. Figure 5 shows the child obtained from crossover between Chromosome 1 and Chromosome 2 (Figure 4). The blue blocks are inherited from Chromosome 1 and the grey ones are from Chromosome 2.

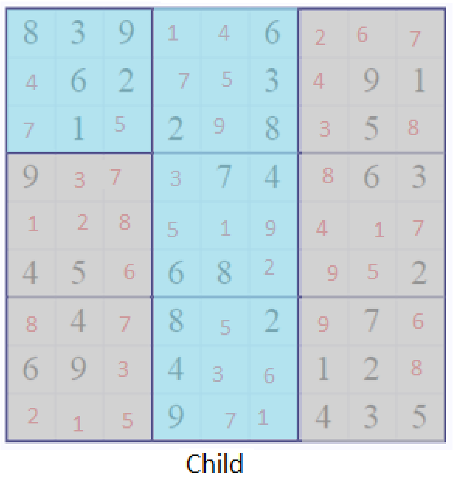


Figure 5. Child resulting from crossover between Chromosome 1 and Chromosome 2 found in Figure 2.

3.1.6 *Mutation*

The child resulting from crossover undergoes mutation with probability *μ*. Mutation consists of swapping two pairs of cells in the same block. The mutated chromosome is added to the next generation. The two parents are also mutated with probability *μ'* and the best one is copied to the next generation allowing for interaction between old and new chromosomes.

3.1.7 *Elitism*

We apply elitism whereby the best chromosome in terms of fitness is always transferred as is to the next generation.

3.1.8 *The Purge Operator*

A major problem with GA is getting stuck in local optima. In order to avoid this, we present a new idea called "Purge" where we crossover all the population with the best chromosome which will be mutated at the end. All the resulted individuals will be copied to the next generation. This step is performed each time the objective function does not change for ρ iterations.

3.1.9 *Stopping Criteria*

We define our stopping criteria to be a maximum number of iterations or when we find a solution with 0 penalty.

4. EXPERIMENTATION AND RESULTS

To test our developed technique, we used www.websudoku.com to generate 3 different puzzles (two examples of easy level and one medium). We ran our GA on an Intel(R) Xeon(R) E7-8870 v2 @ 2.30GHz CPU with 30720 KB cache and 629GB RAM running Red Hat CentOS Linux, Version 7. We repeated our GA 50 times on each puzzle as done in [1] for the sake of comparison. We tested the algorithm with different parameters. We show in Table 1 the ones that gave the best results.

Table 1. Parameters used in our GA instantiation.

Variable	Value	Description
n	700	Size of population
K	7	K- Tournament
μ	0.05	Child mutation probability
μ'	0.5	Parent mutation probability

ρ	200	Number of iteration until purge
Max Iterations	3000	Maximum number of iterations

We computed the average number of generations needed in order to solve each puzzle. In Table 2, we show the obtained results and we compare them to [1].

Table 2. Results of our GA in terms of average number of generations needed to find a feasible solution compared with [1]

Level	Sudoku-GA	GA of [1]
Easy 1	**127.59**	254.16
Easy 2	**220**	312.46
Medium 1	**483.44**	Cannot be solved

To test the performance of our algorithm we did an additional comparison with the work in [6] as shown in Table 3. Here, also, and for the sake of comparison, we run our GA 25 times.

Table 3. The results (number of iterations) of our GA compared to the work in [6]

Level	Sudoku-GA	GA/ACO of [6]
Easy (No. 1)	**46.56**	275
Easy (No. 11)	**88.4**	682
Medium (No. 27)	**188**	9103
Medium (No. 29)	**357.15**	11783
Difficult (No. 77)	**702.67**	62727
Difficult (No. 106)	**1797**	174401

5. DISCUSSION

As shown in Table 2 and Table 3, our GA combined with the "Purge technique" is able to outperform the GA presented in [1] on both puzzles "Easy 1" and "Easy 2". In addition to that, the GA in [1] was not able to solve the medium puzzle that we successfully solved. Our GA implementation was also able to outperform the hybrid combination of GA and ACO presented in [6]. Results in Table 3 show the significant improvement that our GA gives over the heuristic presented in [6]. In fact, the improvement becomes more significant as the difficulty level of the game increases and reaches close to 100 times more on the most difficult level (Difficult No. 106).

6. CONCLUSION

In this paper, we present a new GA combined with a "purge technique" to solve the Sudoku game. The new purge technique is introduced and used to avoid local optima. We compare based on the number of iterations required to solve the same puzzle. Results show a significant improvement over two known techniques.

7. REFERENCES

[1] Douglas, R. 2017. Solving Sudoku Puzzles with Genetic Algorithm. Game Behaviour 1, 1.

[2] Mantere, T. and Koljone, J. 2006. Solving and rating sudoku puzzles with genetic algorithms. In New Developments in Artificial Intelligence and the Semantic Web, Proceedings of the 12th Finnish Artificial Intelligence Conference STeP, 86-92.

[3] Kazemi, S. M. and Fatemi, B. 2014. A retrievable genetic algorithm for efficient solving of Sudoku puzzles. International Journal of Computer, Information Science and Engineering 8, 5.

[4] Sato, Y. and Inoue, H. 2010. Solving Sudoku with genetic operations that preserve building blocks. In Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games. 23-29.

[5] Kamal, S., Chawla, S. and Goel, N. 2015. Detection of Sudoku puzzle using image processing and solving by Backtracking, Simulated Annealing and Genetic Algorithms: A comparative analysis. In 2015 Third International Conference on Image Information Processing (ICIIP). 179-184.

[6] Mantere, T. 2013. Improved ant colony genetic algorithm hybrid for Sudoku solving. In 2013 Third World Congress on Information and Communication Technologies (WICT). IEEE, 274-279.

[7] Mantere, T. and Koljonen, J. 2009. Ant Colony Optimization and a Hybrid Genetic Algorithm for Sudoku Solving. In International Conference on Soft Computing. Mendell, Brno Czech Republic, 41-48.

[8] Geem, Z. W. 2007. Harmony search algorithm for solving sudoku. In International Conference on Knowledge-Based and Intelligent Information and Engineering Systems. Springer, Berlin, 371-378.

[9] Mandal, S. N. and Sadhu, S. 2011. An Efficient Approach to Solve Sudoku Problem by Harmony Search Algorithm. International Journal of Engineering Sciences. 312-323.

[10] Monk, J., Hanselman, K., King, R. and Flagg, R. 2012. Solving Sudoku using Particle Swarm Optimization on CUDA. in In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA).

[11] Pacurib, J. A., Seno, G. M. M. and Yusiong, J. P. T. 2009. Solving sudoku puzzles using improved artificial bee colony algorithm. In 2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC). IEEE, 885-888.

[12] Soto, R., Crawford, B., Galleguillos, C., Monfroy, E. and Paredes, F. 2013. A hybrid ac3-tabu search algorithm for solving sudoku puzzles. Expert Systems with Applications 40, 15, 5817-5821.

[13] Soto, R., Crawford, B., Galleguillos, C., Paredes, F. and Norero, E. 2015. A hybrid alldifferent-tabu search algorithm for solving sudoku puzzles. Computational intelligence and neuroscience 40.

[14] Job, D. and Varghese, P. 2016. Recursive Backtracking for Solving 9* 9 Sudoku Puzzle. Bonfring International Journal of Data Mining 6, 1, 7.

[15] Holland, J. H. 1975. Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. MIT press.