An Ant Algorithm for the Sudoku Problem

Received: 10th December 2014; accepted 3rd February 2015

Krzysztof Schiff

DOI: 10.14313/JAMRIS_2-2015/14

Abstract:
In this paper an ant algorithm for the Sudoku problem is presented. This is the first ant algorithm enabling discovery of an optimal solution to the Sudoku puzzle for 100% of investigated cases. The Sudoku is a one of many combinatorial optimisation problems, as well as an NP-complete problem, hence an ant algorithm which constructs an optimal solution as a meta-heuristic method is important for this problem.

Keywords: *swarm optimization, Sudoku puzzle*

1. Introduction

The Sudoku puzzle is a popular Japanese logical puzzle as well as a combinatorial optimisation problem [4] and an NP-complete problem [15]. Since the Sudoku is a very difficult problem, there are many heuristic methods used to solve it; such heuristic methods, based on human thinking, are described by Pillay [13]. Genetic algorithms for the Sudoku game were discussed by Mantere and Koljonen [7], as well as by Gold [2]. Algorithms based on bee colonies are presented by Pacurib et al. [12] and by Kaur and Goyal [3]. A simulated annealing procedure was shown by Lewis [5]. The ant algorithm was discussed by Mullaney [11]. Particle swarm optimisation algorithms are shown by McGerty [8], Moraglio et al. [9] and Moraglio and Togelius [10]. The Sudoku problem can be transformed into the SAT problem ([6], [14]). The ant algorithm mentioned by Mullaney [11] enables discovery of an optimal solution for only 20% of all investigated problem instances. The ant algorithm presented in this paper works for 100% of all investigated cases.

2. Sudoku Problem

The Sudoku puzzle consists of a 9×9 matrix divided into nine 3×3 sub-grids. Rules for completing the Sudoku game are very simple: each 3×3 sub-grid should contain all 9 digits; each row and every column in a 9×9 matrix should contain all 9 digits.

At the beginning of the game there are already a number of digits given within the 9×9 matrix. An example of an initial matrix is shown in Fig. 1. Empty cells should be filled with digits. Rules for completing the Sudoku puzzle should be observed. In each row and in each column, as well as each 3×3 sub-grid, there should be no repeated digits. In the 9×9 version of Sudoku game there are about 6.671×10^{21} valid grids and generally the problem has been proved by Lawler and Rinnooy [4] to be an NP-complete problem. Sudoku problems can be of different levels of difficulty, from easy to very difficult; some can be solved in a very short time, others not. A very difficult example is shown in Fig. 2; results of tests conducted for a Sudoku problem of such difficulty are discussed in section 5.

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

Fig. 2.1. A very bad example of a Sudoku puzzle

3. Ant Method

Ants search for the best solution to encountered problems. In order to find a such solution, ants communicate among themselves by means of a pheromone *t*. At the beginning of the General Ant Algorithm, which is presented as algorithm 1, a maximal quantity of pheromone is deposited $t(i) = t_{max}$ on all elements $i \in$ M. The set M is the set of elements *i* which can constitute a solution to the given optimisation problem. In the case of the Sudoku problem, set M is the set of all pairs: digit and position. The General Ant Algorithm consists of two main loops: the first is connected with the number of cycles, the second with the number of ants. Within each repetition of the first loop, all repetitions of the second loop have to be performed. The best solution S_b found by all ants in one cycle is compared to the best solution S_{best} found by ants in the previous cycle. In each cycle an evaporation mechanism is also used: some of the pheromone evaporates at the rate *r* from all elements *i* M. In each cycle an additional quantity of pheromone *dt* is deposited on those elements *i* which constitute a solution S_b. When all loops have been done the best solution is obtained. At the beginning of each inner loop, a starting point is prepared for each ant. From this starting point each ant begins to create a solution to the optimisation problem and then while in the loop

each ant selects the next element j with probability p(j) and adds it to the solution set S. The probability p(j) can be expressed by the formula

$$p(j) = \frac{t_j n_j}{\sum_{j=1}^{n}(t_j n_j)} \quad (3.1)$$

where t_j is the quantity of pheromone deposited on element j, ($1 \leq j \leq$ max), max is the maximum number of available elements from which the selection can be made, n_j is a heuristic, that is, the desirability of including element j in the solution set S.

This selection can be made only from set A, i.e. from those elements i which are available and which can constitute, at this moment of algorithm use, a solution to the optimisation problem. When any element is added to the solution set S, not all elements from set A still satisfy constraints; thus, from the previous set A a new set A is created by including in this set A only those elements from the previous set A which satisfy constraints. In the case of the Sudoku problem, when an element i (representing a pair: digit and position) is included in set S, then, because a digit has been used, it cannot be used in any other cell in that column or row of the 9×9 grid nor in any other cell of the 3×3 sub-grid. Set A should be now updated so that all digit-position pairs which can no longer constitute a proper solution to the problem are removed from set A.

Algorithm 1. The General Ant Algorithm

```
for all i ∈ M: t(i) = t_max
for all cycles
  for all ants
    make a starting point
    while (a solution S is not completed) do
      check which elements are available to be selected, add them to set A
      select the next element from the set A with probability p(j)
      add a selected element to S
    save in the S_b the best solution which has been found by all ants in a
cycle
  if S_b is better than S_best then save S_b as S_best : S_best = S_b
  for all i:  t(i) = t(i) + r* t
  dt = f (S_b)
  if  i ∈ S_b then t[i]= t(i) + dt
return S_best
```

4. Ant algorithm for the Sudoku Problem

All ants search for the optimal solution to the Sudoku problem; they communicate among themselves by means of a pheromone, which is stored in a 3-dimension table *t[][][]*. The pheromone has been placed on a digit k in each cell of the 9×9 grid, which is represented by a table *b[i][j]* (line 1), so 3-dimensional table *t[i][j][k]* is used in order to store the amount of pheromone placed on each digit-position pair. If no digit has been entered into the 9×9 grid the set M will consist of all digit-position pairs; that is, it consists of all cells from 3-dimensional table *t[][][]*. At the beginning, if some digits have been entered into the 9×9 grid *b[][]*, then set M does not consists of all cellules from 3-dimensional table *t[][][]*. Whenever any digit-position pair is selected, some cellules from 3-dimensional table *t[][][]* are no longer available for selection according to the rules. Set A now should be updated (lines 11–15). After the first selection of a digit-position pair, set A is obtained from set M; after the following selection is made, a new set A is obtained from the previous set A. When a digit-position pair is selected, the position (*i,j*) is filled with a digit k: b[i][j]=k (lines 22 or 29 or 39). Since each of the nine digits can be entered in only one cell in each row and in each column, two matrices are used: one for rows and one for columns, in order to prevent entry of the same digit twice in the same row *digit_row[][]* or in the same column *digit_column[][]*. These two tables, *digit_row[][]* and *digit_column[][]*, are used to indicated these cells of 3-dimensional table *t[][][]*, which are included in set A. By using these two 2-dimensional tables an update of the set A is made after each selection of a digit-position pair (lines 13,14). At the beginning of each session of ant work there are some digits placed in the 9×9 grid, which is represented by matrix *a[i][j]*. The matrix *b[i][j]* is a work matrix, which each ant fills with digits during algorithm use (line 6). For each of the digits k in each 3×3 sub-grid, the number of positions in which this digit can be entered is calculated and this number is stored in *places[i][j][k]* (lines 16–19). For each vacant in the 9×9 grid the number of digits which can be entered there is calculated and stored in *digits[i][j]* (line 25–27). Next, all digits which can be entered in only one position in grid *a[i][j]* are entered into these positions (lines 21–23); also, positions which can be filled with only one digit are thus filled (lines 28–30). Of course, the same digit should not be entered twice in the same row, column or 3×3 sub-grid (line 20). Afterwards, if there are no digits which can be entered only in one position and there is no position which can be filled with only one digit, there are digits which can be entered in more than one position and there are positions which can be filled with more than one digit. In such a situation it is necessary to make a selection of a pair: a digit and a position (lines 31–40). In order to make a such selection, the heuristic pattern is proposed (line 31)

$$n[i][j][k] = (10 - places\,[i][j][k])(10 - digits\,[i][j]). \quad (4.1)$$

and the probability *p[i][j][k]* of selecting a digit k together with a cell *b[i][j]* has to be calculated (lines 32–34). When all ants have finished their work, the best solution from their work, which is stored under the variable *maxselected*, is used in order to put an additional quantity of pheromone *dt* = *maxselected*/81 on each connection between a digit and a cell in the

```
for all t[i][j][k]=1000
 gmaxselected=0
 for all cycles
        maxselected=0
        for all ants
                for all b[i][j]=a[i][j]
                selected=0
                can_select=1
                while(can_select)
                {
                for all a[i][j]≠0
                        digit=a[i][j]
                        digit_row[i][digit]=1
                        digit_column[j][digit]=1
                        selected=selected+1
                for all digits m
                        for all positions in the 9×9 grid
                        into how many positions in the 3×3 sub-grid can this digit can be entered
                                positions[i][j][m]=number of positions
                        when you can enter any digit into the 9×9 grid then can_select=0
                        when you can enter a digit k in only one position then
                                this digit has to be entered in this position: b[i][j]=k
                                selected=selected+1
                                this is repeated for all digits m
                for all positions in the 9×9 grid
                        how many digits can be entered in one position
                        digits[i][j]=number of digits
                        when only one digit can be entered in one position
                    then enter this digit in this position: b[i][j]=k
                    selected=selected+1
                for all w[i][j][k]=t[i][j][k]* (10-places[i][j][k]))(10-digits[i][j])
                sumw=0;
                for all sumw = sumw + w[i][j][k]
                for all p[i][j][k] = w[i][j][k]/sumw
                p = rand()
                sump=0;
                for all sump=sump+p[i][j][k]
                        if (sump>p)
                                place a digit k in a position b[i][j]: b[i][j]=k
                                selected=selected+1
                if (selected==81) can_select=0;
                }//end_while(can_select)
                if (selected>maxselected)
                        maxselected=selected
                        for all mb[i][j]=b[i][j]
        dt = maxselect/81
        for all t[i][j][k]= r * t[i][j][k]
        for all mb[i][j]!=0
             k=mb[i][j] and t[i][j][k]=t[i][j][k]+dt
        if (maxselect>gmaxselect)
                gmaxselect=select
                for all gmb[i][j]=mb[i][j]
```

Figure 4.1. A pseudo-code of the elaborated ant algorithm

3-dimensional matrix *t[][][]* (lines 43–49). This additional quantity of pheromone *dt* is deposited in each cycle (line 49); in each cycle an evaporation mechanism *r* is used (line 47). In some cases the solution to the problem is not obtained by the ants; in such cases the maximum number of digits entered into the 9×9 grid is remembered under the variable *selected*. Another variable *can_select* indicates that the next selection of a digit-position pair is possible and can be made. The ant algorithm succeeds when a solution has been found or after all cycles have been performed, with the best obtained solution stored in *gmb[][]* (lines 50–52). The pseudo-code of the ant algorithm is presented as algorithm 2.

5. Results of Experiments

The elaborated ant algorithm was tested. This ant algorithm enabled discovery of a solution to the Sudoku problem at the greatest level of difficulty only for an evaporation rate ranging from 0.995 to 0.999; outside this range, the algorithm does not arrive at the optimal solution to the Sudoku puzzle. Tests were conducted for a number of ants equal to 700. The number of cycles needed to obtain an optimal solution to the Sudoku puzzle depends on the evaporation rate. Results as average values from 20 measurements are shown in Table 5.1 and in Fig. 5.1.

Other tests were conducted for cases in which the number of ants varied and for a constant evaporation

rate, which was equal to 0.998. The number of cycles needed to obtain an optimal solution to the Sudoku problem decreased when the number of ants rose and was rather stable when the number of ants was greater than 700. Results, as average values from 20 measurements, are shown in Table 5.2 and in Fig. 5.2.

Table 5.1. Number of cycles in dependency of evaporation rate

evaporation rate	0.999	0.998	0.997	0.996	0.995
number of cycles	241.7	188.8	215.8	219.5	203.8

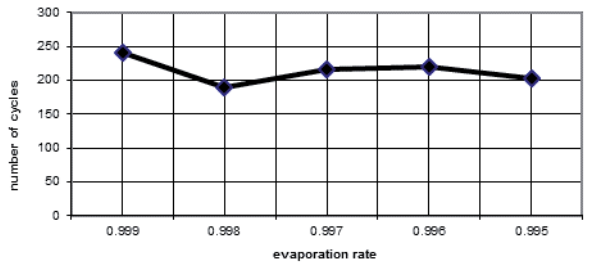

Figure 5.1 Number of cycles in dependency of evaporation rate

Table 5.2. Number of cycles in dependency of ant numbers

number of ants	900	800	700	600	500
number of cycles	201.4	174.3	188.8	248.1	358.9

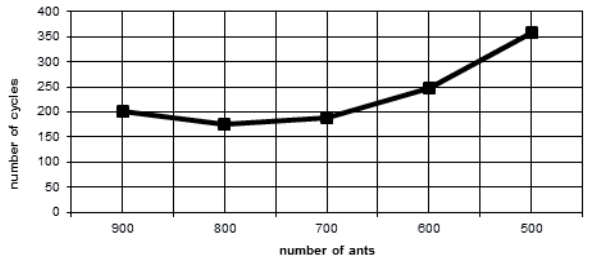

Figure 5.2. Number of cycles in dependency of ant numbers

6. Conclusion

Problem cases were taken from www.websudoku.com. Most people can solve a Sudoku puzzle in about 30 minutes. The ant algorithm presented in this paper finds an optimal solution for many problem instances in milliseconds, but for some very difficult cases it took about 20–25 minutes on a computer with an Intel Celeron CPU 1.7GHz and 256 MB RAM, though this is still faster than people can do. This new elaborated ant algorithm enabled discovery of an optimal solution to all investigated cases, not only to some as in [11]. The elaborated algorithm was not compared to the Mullaney algorithm, since Mullaney provided neither computer code nor pseudo-code for the algorithms in his paper.

AUTHOR

Krzysztof Schiff – Department of Automatic Control and Information Technology, Faculty of Electrical and Computer Engineering, Cracow University of Technology, ul. Warszawska 24, 31-155 Kraków, Poland.
E-mail: kschiff@pk.edu.pl.

REFERENCES

[1] Boryczko U., Juszczuk P., "Solving The Sudoku With Differential evolution", *Zeszyty Naukowe Politechniki Białostockiej. Informatyka*, no. 9, 2012, 5–16.
[2] Gold M., *Using Genetic Algorithms to come up with Sudoku Puzzles*, 2005.
[3] Kaur A., Goyal S.,Survey on the Applications of Bee Colony Optimization Techniques. *International Journal on Computer Science and Engineering (IJCSE)*, 3, 8, 2011, 3037-3046.
[4] Lawler E. and Rinnooy K.A. (1985) The Traveling Salesman Problem: A guided Tour of Combinatorial Optimization.
[5] Lewis R. (2009) Metaheuristics can solve Sudoku puzzles. *Journal of Heuristics*,13, 387-401.
[6] Lynce I. and Ouaknine J. (2006) Sudoku as a SAT Problem, *Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics*.
[7] Mantere T. and Koljonen J. (2007) Solving, Rating and Generating Sudoku Puzzles with GA *IEEE Congress on Evolutionary Computation*, 1382-1389.
[8] McGerty S. (2009) Solving Sudoku Puzzles with Particle Swarm Optimization – *Final Report*, Macquarie University.
[9] Moraglio A. et al. (2007) Geometrical Particle Swarm Optimization – Research Article, *Journal of Artificial Evolution and Applications*, 2008.
[10] Moraglio A. and Togelius J. (2009) Geometrical differential evolution, GECCO 2009, *Genetic and Evolutionary Computation Conference*, 1705-1712.
[11] Mullaney D., Using Ant Systems to solve Sudoku Problems, University College Dublin, 2009.
[12] Pacurib, J. A. et al. (2009) Solving Sudoku Puzzles using Improved Artificial Bee Colony Algorithm. In: *Proc. 4th Int. Conf. Innovative Computing*, *Information and Control*, 885-888.
[13] Pillay N. (2012) Finding Solutions to Sudoku Puzzles Using Human Intuitive Heuristics, *Research Article — SACJ*, 49, 25-34.
[14] Weber T. (2005) A SAT-based Sudoku solver, 12th International Conference on Logic for Programming, *Artificial Intelligence and Reasoning*, LPAR 2005, 11-15.
[15] Yato T. (2003) Complexity and Completeness of Finding Another Solution and Application to Puzzles, *IEICE – Transactions on Fundamentals of Electronic Communications and Computer Science*, 5, 1052-1060.