

Solving Sudoku with the GAuGE System

Miguel Nicolau and Conor Ryan

Biocomputing and Developmental Systems Group
Department Of Computer Science And Information Systems
University of Limerick, Ireland
`{Miguel.Nicolau,Conor.Ryan}@ul.ie`

Abstract. This paper presents an evolutionary approach to solving Sudoku puzzles. Sudoku is an interesting problem because it is a challenging logical puzzle that has previously only been solved by computers using various brute force methods, but it is also an abstract form of a timetabling problem, and is scalably difficult. A different take on the problem, motivated by the desire to be able to generalise it, is presented. The GAuGE system was applied to the problem, and the results obtained show that its mapping process is well suited for this class of problems.

1 Introduction

The GAuGE system (Genetic Algorithms using Grammatical Evolution) [14] is a genetic algorithm that uses a position independent representation of fixed-length genotype strings which, through a mapping process, generate fixed-length phenotype strings, which are neither under- nor over-specified. By position independent it is meant that each phenotypic variable is encoded into the genotype string along with an associated phenotypic position; that leads to a simultaneous evolution of both the structure and the contents of the genotype strings.

This simultaneous evolution has the potential to learn linear relationships (or dependencies) between variables; as the structure of genotype strings can adapt, more important (or salient) variables can be grouped together, boosting the exchange capability of the crossover operator. Due to its specific mapping process, GAuGE also allows phenotypic variables to exchange places, thus searching the space of permutations of such variables in the phenotype strings. Although not always a desirable effect, certain problem domains possess characteristics that make this permutation search a welcome feature [14, 9].

The Sudoku puzzle seems to require that characteristic. The game is composed of a $n \times n$ board, and the objective is to fill it with numbers, following a set of simple constraints (see Section 3). Although there exist many algorithms that solve these puzzles in a matter of seconds, in this work a different problem is solved: a sequence of instructions is evolved and applied to the board, and a fitness reward is given back. The objective is therefore to provide a sequence of instructions that solves the puzzle, and that is human-readable; to that effect, a number of logical instructions are available to the algorithm, which mimic the way a human solves this kind of puzzle. Moreover, Sudoku is analogous to many

scheduling and timetabling problems, and a system that can not only provide a timetable, but also the logical steps used in deriving it can be a powerful tool.

The results obtained show that GAuGE is able to solve this problem, if a sufficient function set is available. Also, a fair degree of temporal saliency clearly helps the system to restructure individuals at the genotypic level, leading to the discovery of better individuals as evolution progresses.

This paper is structured as follows: Section 2 presents the GAuGE system, and Section 3 presents the Sudoku game. Section 4 presents the practical issues of the experiments performed, and Section 5 analyses the results obtained.

2 GAuGE

The GAuGE system shares many of the biologically inspired features of Grammatical Evolution (GE) [13], the main ones being a genotype-to-phenotype mapping process, a functional dependency between genes, and the use of degeneracy.

In GE, a population of binary strings is evolved. When an evaluation is required, these are first converted into integer strings, and the integers are then used to choose productions from a given grammar, creating a phenotype string.

In GAuGE, a similar process is employed. When a binary string is to be evaluated, it is first converted into an integer string; these integers are then interpreted as a sequence of *(position, value)* pairs, to create a phenotype string.

In GE, there is a functional dependency between each gene and all the genes that precede it. This is because the grammar production chosen by a given gene affects the context of the following genes; as a result, the set of productions available for each gene is dependent on the context created by previous choices.

A similar effect is observed in GAuGE, regarding each position specification at the genotypic level. The phenotypic position corresponding to that specification affects the context of the following specifications, as the set of available positions in the phenotype string changes; as a result, each specification is dependent on the context created by previous specifications.

Finally, the use of degenerate code plays an important role in GE: by using the *mod* operator to map an integer to a choice of productions from a grammar rule, neutral mutations can take place, creating a many-to-one mapping between the search and solution spaces, and introducing variety at the genotypic level.

In GAuGE, this feature is also present, as a direct result of the mapping process employed. It has also been shown that the explicit introduction of degeneracy can reduce structural bias at the genotypic level [10].

2.1 Background

Many systems have been developed using similar techniques to the ones employed in GAuGE. Bagley [1] used fixed-length strings of *(position, value)* specifications, and an inversion operator to move those specifications around in the chromosome strings; both Frantz [3] and Holland [7] extended some of that work, and similar operators were later designed, with the same purpose [12].

The messy genetic algorithms [4, 5] are also based on the idea of a separate encoding of the position and the value of each phenotypic variable. They dealt with the problem of over-specification with a system of “first come, first served” basis, whereas under-specification was dealt with the use of an evolved template.

More recently, Harik [6] applied the principles of functional dependency to the Linkage Learning Genetic Algorithm, in which a chromosome is expressed as a circular list of genes, with the functionality of a gene being dependent on a chosen interpretation point, and the genes between that point and itself.

2.2 GAuGE Mapping

A formal description of the mapping process can be found elsewhere [10]; in this work, a practical approach is presented. As an example, consider a simple problem composed of four phenotypic variables ($\ell = 4$), ranging between the values 0 and 7. The length of each individual depends on a chosen position field size (pfs) and a value field size (vfs). As there are four variables, a value of $pfs = 2$ has been chosen, as that is the minimum number of bits required to encode four positions; for the value fields, a value of $vfs = 4$ has been chosen, to introduce degeneracy (the minimum value required for vfs is 3). The required length for each binary string is therefore $L = (pfs + vfs) \times \ell = (2 + 4) \times 4 = 24$.

For example, take the following individual as an example genotype string:

001001101101110100010010

By using the pfs and vfs parameters, an integer string is created:

(0, 9), (2, 13), (3, 4), (1, 2)

These values are then interpreted as a sequence of (*position*, *value*) pairs: for each one, the *position* is mapped to the number of positions available in the phenotype string, and the *value* is mapped to the range of the phenotypic variables (8). For the first pair, the *position* becomes $(0 \bmod \ell) = (0 \bmod 4) = 0$, as at this stage no positions have been specified yet; the *value* becomes $(9 \bmod 8) = 1$. The phenotype string can thus begin to be constructed, by placing *value* 1 into *position* 0 (that is, the first **available** position in the phenotype string):

1,?,?,?

The second pair is decoded in the same way: the *position* becomes $(2 \bmod 3) = 1$ (as there are now only 3 positions available in the phenotype string), and the *value* becomes $(13 \bmod 8 = 5)$. The *value* 5 can then be placed into *position* 1 (the second currently available position in the phenotype string):

1,?,5,?

The third pair is processed in the same way: *position* = $(3 \bmod 2) = 1$, and *value* = $(4 \bmod 8) = 4$, so *value* 4 is placed in the second available position:

1,?,5,4

Finally, after processing the last pair, the phenotype string becomes:

1,2,5,4

		columns								
		1	2	3	4	5	6	7	8	9
rows	1	block	block	block						
	2	1			2					
	3									
	4	block	block	block						
	5									
	6	4			5				6	
	7	block	block	block						
	8									
	9	7			8				9	

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

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

Fig. 1. Example Sudoku boards. The referencing system used is shown on the left, an example board in the middle, and its solution in the right.

3 Sudoku

The *Sudoku* game is a logic-based placement puzzle. It originated in the United States in 1979, under the name *Number Place*; in 1984, it was slightly changed in Japan, and quickly gained popularity. In November 2004, the British newspaper “The Times” first published a Sudoku puzzle; since then, many newspapers followed suit, and its popularity in the western world has increased immensely.

Traditionally, the puzzle consists of a 9×9 grid, made up of 3×3 *blocks*, for a total of 81 *cells* (Fig. 1, left). The objective of the puzzle is to place the numbers 1 through 9 in each cell, such that the following rule set holds:

1. Each row must contain the numbers 1 through 9 only once;
2. Each column must contain the numbers 1 through 9 only once;
3. Each block must contain the numbers 1 through 9 only once.

Each puzzle comes with a set of numbers already placed (called *givens*). It is considered *well-formed* if it has only one solution, and it can be solved using logic (that is, no guessing is required to place any of the numbers). Fig. 1 shows an example of a well-formed Sudoku puzzle, along with its (only) solution.

Although numbers have traditionally been used, any set of symbols can be used in Sudoku (such as letters, shapes or colours). Also, although the 9×9 size grid is the most common, other variants exist, such as 16×16 and 25×25 .

There are clear parallels between Sudoku and scheduling problems, such as timetabling. Each row can be viewed as a time slot, each column a room and each number a course. All courses must be scheduled in each room exactly once per day, and no class can be scheduled in two different rooms at the same time.

3.1 Solving Sudoku with computers

The problem of solving Sudoku puzzles on $n^2 \times n^2$ boards of $n \times n$ blocks is known to be NP-complete [20]; this gives an indication of why solving Sudoku puzzles can be difficult. However, due to the finite size of the puzzle, it can be solved by a deterministic finite automaton that knows the entire game tree [19].

Fig. 2. Applying slice and dice to place number 9 in block 2. Starting with the left board, slice finds a 9 in the third row, so 9 cannot be placed anywhere else in that row (relevant cells are marked with an X in the second board). Dice finds a 9 in the fifth column, so 9 cannot be placed anywhere else in that column (third board). This results in only one cell being available to place 9 in block 2 (last board).

A different approach is to base each number-placing action purely on logic. This technique is limited to well-formed puzzles, but has the advantage of producing a list of logical actions, which can be easily reproduced by a human. Most Sudoku software packages tend to use a mix of logic and brute force computation.

3.2 Logical Operations

There are many logical operations that can be used when solving Sudoku puzzles; below are some of the most common (and simpler) techniques:

Last Remaining. This is a simple logic operation, that can be applied to any kind of region (row, column or block). It simply checks if that region has already eight numbers placed, in which case it places the remaining one.

Slice and Dice. This is a combination of two operations, *slice* and *dice* [17], and can be applied when trying to place number n in block b . Slice looks for n in each row passing through b ; if it contains n , then the three cells intersecting with b cannot contain n . Dice works with columns instead. If, after applying slice and dice, only one cell is available, then it must contain n . Fig. 2 shows an example.

Column Fill. This technique tries to place number n in column c . It looks for n in all rows and blocks passing through c ; if a row or block contain n , then the cell(s) corresponding to the intersection of that region and column c cannot contain n . If after checking all rows and blocks there is only one cell available in column c , then that cell must contain n . Fig. 3 shows an example.

Row Fill. This technique tries to place the number n in row r , and works in the same way as Column Fill, but going through all the columns instead.

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

Fig. 3. Applying Column Fill to place number 9 in column 6. By searching through all rows in the left board for the number 9, a set of cells can be marked as being unsuitable to receive that number (second board); then looking through blocks 2, 5 and 8 (the blocks that intersect column 6) another unsuitable cell is discovered (third board); finally, only one cell is available in column 6, so it must contain 9 (last board).

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

1

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

2

Fig. 4. Applying Raising Numbers to place number 4 in block 4. By analysing the left board, there are four empty cells in block 4, and the numbers 2, 4, 6 and 8 are missing. Through Slice and Dice operations, the set of possible numbers for each of the empty cells can be deduced: the leftmost empty cell can receive the numbers {2, 6}, the center top cell can receive {4, 6, 8}, the rightmost one {2, 4, 6}, and the bottom center {4}. Since this last cell can only receive the number 4, it is placed there (right board).

Raising Numbers. This technique tries to place the number n in block b , by checking each empty cell in b to see if n is the only number that can be placed in that cell; Fig. 4 shows an example.

Sometimes different operations can be used to place the same number in the same cell; other times only a specific one will do. Note also that often an operation can only be applied if a previous one has placed a specific number. In other words, there are many *logical sequences* when using these logical operations.

3.3 Blind Sudoku

In this work, a variation of the original Sudoku puzzle is solved, which is termed *Blind Sudoku*. All the same rules and constraints of the original puzzle still apply; the difference lies in the way in which it is solved. A sequence of logic instructions

is applied to a puzzle, and a measure of goodness is returned to the algorithm; in other words, the puzzle is never available to the algorithm, neither is a measure of goodness for each individual logic instruction. Once a sequence of instructions has been evaluated, the puzzle is reset to its original composition.

This way of solving Sudoku puzzles is not dissimilar to the Santa Fe Ant Trail problem [8], in which a sequence of instructions is given to an ant in a toroidal grid world, and the number of food objects caught by the ant is returned as the fitness measure.

Objective Function. As the number of cells in a 9×9 Sudoku board is always 81, that is used as the size of a sequence of instructions; this allows that sequence to be applied to any puzzle, regardless of the number of givens. The fitness of a sequence of instructions is simply the sum of the fitness of all its instructions; the fitness of each single instruction is

$$f_i = \begin{cases} k \times (82 - i) & \text{if successful} \\ \text{coverage} - 9 & \text{if unsuccessful} \end{cases}$$

where k is a constant, and *coverage* is a measure of how many cells were ruled out when unsuccessfully trying to place a number in a region (the **X** marks)¹. If the puzzle is completed before using up all the instructions, all remaining instructions are considered neutral, and their fitness is $f_i = 0$.

The fitness function heavily rewards successful instructions, and punishes unsuccessful ones. A linear decreasing reward is also applied to each successful instruction: the earlier it is executed, the bigger the reward. This temporal saliency is regulated by the k parameter; the higher its value, the higher the reward (in these experiments, a value of $k = 81$ was used).

This problem has interesting characteristics, and can be compared to a class of scheduling problems. There is a clear temporal dependency between each phenotypic variable, as certain instructions can only be successful if a set of numbers has been placed before their execution. The negative score applied to unsuccessful instructions can be seen as an effort factor.

The GAuGE system seems adequate to solving this kind of problem. It allows for successful instructions, which have already been discovered, to change their phenotypic location (by mutating their position specification at the genotypic level), allowing them to be moved to the start of the phenotype string, thus possibly maximising their contribution to the fitness of the set of instructions.

4 Experiments

4.1 Practical Issues

In these experiments, GAuGE was used to evolve sequences of 81 instructions, from the set {SliceDice, RowFill, ColFill, RaisingNumbers}. If an instruction is

¹ Note that the Raising Numbers technique does not mark any cells as unsuitable, and thus if it is unsuccessful its fitness is always -9 ; this is judged to be fair, as it is a slightly more expensive technique than all others.

successful, the LastRemaining instruction is tried on the corresponding region (as it is a fast instruction), and if successful the remaining number is placed.

Also, the logic instructions are mapped onto the original board. For example, if the algorithm tries to execute the instruction *SliceDice(2,8)* (place number 8 in block 2 using SliceDice) on the board from Fig. 1, that instruction is translated to *SliceDice(2,9)*, as 8 already exists in block 2. This is only applied to the original board: if a subsequent instruction is also *SliceDice(2,8)*, then it is considered unsuccessful, as number 9 has already been successfully placed in block 2.

The test set for these experiments consisted of puzzles taken from Carol Vorderman’s How to do Sudoku [17] (pp. 178–187). These were taken from the “Difficult” section, and the first ten puzzles were picked (#111 to #120).

4.2 GAuGE Encoding

As there are 81 instructions in each sequence, the *pfs* parameter (size of position fields) needs to be at least 7 (as $2^7 = 128$). In these experiments, degeneracy is used to soften the biases of the *mod* operator [10], so the value chosen is *pfs* = 8.

The *vfs* parameter is more complex. Each variable encodes three choices:

1. which instruction to use (out of 4 instructions);
2. which region to apply it to (out of 9 regions, be it blocks, rows or columns);
3. which number to attempt to place (out of 9 numbers).

To encode an instruction, 2 bits are sufficient. To encode a region and a number, the minimum number of bits is 4 ($2^4 = 16$); as with the *pfs* parameter, degeneracy is used, and 5 bits are used for each of these encodings, so *vfs* = 2 + 5 + 5 = 12. This means that the length of each GAuGE string is:

$$L = (pfs + vfs) \times \ell = (8 + 12) \times 81 = 1620 \text{ bits}$$

4.3 Parameters

Table 1 shows the parameters used in these experiments. The mutation probabilities are in a *per-bit* basis; the following rough formulae were used to set those probabilities, to limit mutation events on each field to 1 per individual:

$$P_{\text{mut}}(\text{pos}) = \frac{1}{(8 \times \ell)} = \frac{1}{648} \approx 0.0015 \quad P_{\text{mut}}(\text{val}) = \frac{1}{(12 \times \ell)} = \frac{1}{972} \approx 0.001$$

As a replacement strategy the Minimal Generation Gap model (MGG) [15] was used; previous tests, both published [11] and not, suggest that it is appropriate for GAuGE, as it maintains diversity at the early stages of evolution, and also keeps the population from stagnation at the later stages. It works as follows:

1. Two random parents are selected from the population;
2. The crossover operator is applied;
3. The two generated offspring are mutated and evaluated;
4. The best of the four individuals (both parents and offspring) is selected to replace the first parent in the population;
5. A roulette wheel is used to select another individual from all four to replace the second parent in the population.

Table 1. Experimental setup

Replacement strategy:	MGG
Crossover operator:	1-point
Problem length (ℓ):	81
Population size (N):	100
Max. number of generations:	800
Position field size (pfs):	8 bits
Value field size (vfs):	12 bits
Crossover probability:	0.5
Position field mutation probability:	0.0015
Value field mutation probability:	0.001

Table 2. Results obtained for all puzzles. For each one, the average numbers placed (over the total missing numbers) are shown, along with the average generation at which those placements were achieved, and the number of successful runs (out of 30)

Puzzle	#111	#112	#113	#114	#115	#116	#117	#118	#119	#120
Avg. placements	53/53	51/51	53/53	53/53	51/51	21/53	13/53	54/54	51/51	51/51
Avg. generation	238	181	166	210	123	55	14	301	188	135
Successful runs	30	30	30	30	30	0	0	30	30	30

5 Analysis

5.1 Results

Table 2 shows the results obtained in all puzzles; all runs for each puzzle (apart from puzzles #116 and #117) were successful. The average number of generations required to complete a puzzle can be seen as a rough measure of its difficulty: notice how for puzzle #118, which required 54 numbers to be placed, a higher number of generations were required for all runs to be successful.

Puzzles #116 and #117 were never solved. Close analysis of these puzzles showed that the function set available to the system was not sufficient to solve them: a brute force search with the available logical functions was performed, and a result could not be reached. Also, the maximum amount of numbers placed for each puzzle with the brute force search method was 21 and 13 respectively, which were the results obtained with GAuGE.

5.2 Sample run and solution

Figure 5 shows results from a sample run (puzzle #111, first run). It illustrates a behaviour observed in all runs: as evolution progresses, the phenotypic strings are rearranged, with successful instructions being placed at the start, due to fitness pressure, and through GAuGE’s flexible genotypic representation. As these instructions are forced towards the left side, more instructions are discovered in the right side, which are then moved to the left as well.

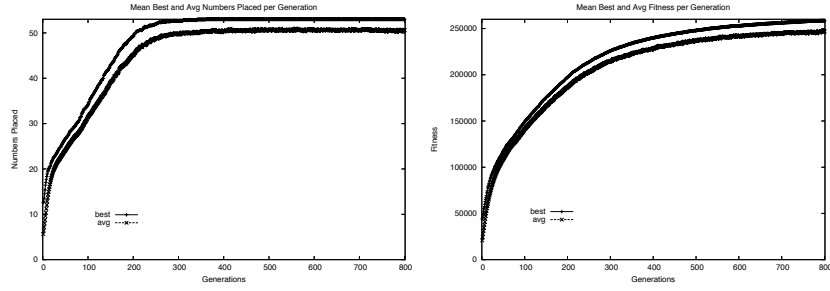


Fig. 6. Results for puzzle #111, averaged across 30 runs. The graph on the left shows the mean maximum and average number of successful instructions, and the graph on the right the mean best and average fitness scores. Notice how around generation 350 all numbers have already been placed on all runs, but yet the fitness score continues to raise, due to the nature of the fitness score used in the experiments.

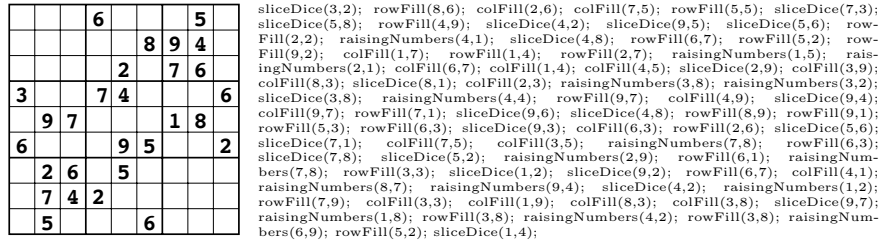


Fig. 7. Puzzle #111 and a sample solution found for it.

The characteristics of the problem make it easily translated to real-world problems, such as timetabling, where variables exhibit a degree of temporal saliency, and potential dependency. It has the added bonus that the solution is not known in advance (and not required for the evaluation of solutions), so each evolved solution is potentially unique.

The GAuGE system, when supplied with suitable logical instructions, is able to solve the problem, and does so by producing a list of the logical steps taken. This could be invaluable when tackling real world scheduling problems, as the list of steps provides an audit trail, so the solutions produced are *provably correct*.

Future work will consider increasing the number of operators made available to the system, to make it possible to solve more difficult puzzles, although this will increase the search space. We will also examine real world scheduling problems with this system, which, particularly in the case of timetabling problems, can be viewed as special cases of Sudoku.

References

1. Bagley, J. D.: The Behaviour of Adaptive Systems which Employ Genetic and Correlation Algorithms. PhD Thesis, University of Michigan. (1967)
2. Felgenhauer, B., and Jarvis, F.: Enumerating Possible Sudoku Grids. Technical Report. (2005) <http://www.shef.ac.uk/~pmlafj/sudoku/>
3. Frantz, D. R.: Non-linearities in Genetic Adaptive Search. PhD Thesis, University of Michigan. (1972)
4. Goldberg, D. E., Korb, B., and Deb, K.: Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, Vol. **3**, number 5. (1989) pp. 493–530
5. Goldberg, D. E., Deb, K., Kargupta, H., and Harik, G.: Rapid, Accurate Optimization of Difficult Problems Using Fast Messy Genetic Algorithms. In Forrest, S. (Ed.): *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers. (1993) pp. 56–64
6. Harik, G.: Learning Gene Linkage to Efficiently Solve Problems of Bounded Difficulty Using Genetic Algorithms. Doctoral Dissertation, University of Illinois (1997)
7. Holland, J. H.: *Adaptation in Natural and Artificial Systems* (Second Edition). University of Michigan Press. (1992)
8. Koza, J. R.: *Genetic Programming: On the Programming of Computers by Means of Natural Evolution*. MIT Press. (1992)
9. Nicolau, M. and Ryan, C.: How Functional Dependency Adapts to Salience Hierarchy in the GAuGE System. In: Ryan et al, (eds.): *Proceedings of EuroGP-2003. Lecture Notes in Computer Science*, Vol. 2610. Springer-Verlag. (2003) pp. 153–163
10. Nicolau, M., Auger, A., and Ryan, C.: Functional Dependency and Degeneracy: Detailed Analysis of the GAuGE System. In: Liardet et al, (eds.): *Proceedings of Évolution Artificielle 2003. Lecture Notes in Computer Science*, Vol. 2936. Springer-Verlag. (2003) pp. 15–26
11. Ohnishi, K. Sastry, K., Chen, Y.-P., and Goldberg, D.: Inducing Sequentiality Using Grammatical Genetic Codes. In: Deb et al, (eds.): *Genetic and Evolutionary Computation - GECCO 2004. Lecture Notes in Computer Science*, Vol. 3102. Springer. (2004) pp. 1426–1437
12. Oliver, I. M., Smith, D. J., and Holland, J. R. C.: A Study of Permutation Crossover Operators on the Travelling Salesman Problem. In Grefenstette, J. J. (Ed.): *Proceedings of the Second International Conference on Genetic Algorithms*. Lawrence Erlbaum Associates. (1987) pp. 224–230
13. O'Neill, M. and Ryan, C.: *Grammatical Evolution - Evolving programs in an arbitrary language*. Kluwer Academic Publishers. (2003)
14. Ryan, C., Nicolau, M., and O'Neill, M.: Genetic Algorithms using Grammatical Evolution. In: Foster et al, (eds.): *Proceedings of EuroGP-2002. Lecture Notes in Computer Science*, Vol. 2278. Springer-Verlag. (2002) pp. 278–287
15. Satoh, H., Yamamura, M., and Kobayashi, S.: Minimal Generation Gap Model for GAs Considering Both Exploration and Exploitation. In: *Proceedings of the 4th International Conference on Fuzzy Systems, Neural Networks and Soft Computing* (vol. 2). World Scientific, Singapore. (1996) pp. 494–497
16. Sudoku Bulletin Board. <http://www.setbb.com/phpbb/?mforum=sudoku>
17. Vorderman, C.: *Carol Vorderman's How To Do Sudoku*. Ebury Press. (2005)
18. Websudoku. <http://websudoku.com>
19. Wikipedia: Sudoku. <http://en.wikipedia.org/wiki/Sudoku>
20. Yato, T., and Seta, T.: Complexity and Completeness of Finding Another Solution and its Application to Puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. **86**, No. 5. (2003) pp. 1052–1060