Finding Solutions to Sudoku Puzzles Using Human Intuitive Heuristics

Nelishia Pillay[1]

School of Mathematics, Statistics and Computer Science, University of KwaZulu-Natal, South Africa.

ABSTRACT

Sudoku is a logical puzzle that has achieved international popularity. Given this, there have been a number of computer solvers developed for this puzzle. Various methods including genetic algorithms, simulated annealing, particle swarm optimization and harmony search have been evaluated for this purpose. The approach described in this paper combines human intuition and optimization to solve Sudoku problems. The main contribution of this paper is a set of heuristic moves, incorporating human expertise, to solve Sudoku puzzles. The paper investigates the use of genetic programming to optimize a space of programs composed of these heuristics moves, with the aim of evolving a program that can produce a solution to the Sudoku problem instance. Each program is a combination of randomly selected moves. The approach was tested on 1800 Sudoku puzzles of differing difficulty. The approach presented was able to solve all 1800 problems, with a majority of these problems being solved in under a second. For a majority of the puzzles evolution was not needed and random combinations of the moves created during the initial population produced solutions. For the more difficult problems at least one generation of evolution was needed to find a solution. Further analysis revealed that solution programs for the more difficult problems could be found by enumerating random combinations of the move operators, however at a cost of higher runtimes. The performance of the approach presented was found to be comparable to other methods used to solve Sudoku problems and in a number of cases produced better results.

CATEGORIES AND SUBJECT DESCRIPTORS

I.2. [**Computing Methodologies**]: Artificial Intelligence.

GENERAL TERMS

Algorithms, Theory

KEYWORDS

Sudoku, genetic programming

1. INTRODUCTION

A fair amount of research has been conducted into the derivation of methods for solving Sudoku problems. Initially these were brute force methods but as this domain developed further optimization techniques such as genetic algorithms, simulated annealing, bee colony optimization and particle swarm optimization, amongst others, have been evaluated as a means of solving Sudoku puzzles. It is evident from this research that Sudoku puzzles are not trivial to solve.

The research presented in this paper firstly contributes to the area investigating techniques for solving Sudoku problems. A set of heuristics, based on human intuition, have been derived for solving Sudoku. These heuristics are essentially moves which are applied to a Sudoku puzzle and incorporate human expertise. The paper also evaluates the use of a genetic programming approach (GPA) to combine these moves into programs for solving Sudoku. Genetic programming (GP) [14] is a variation of genetic algorithms which attempts to solve problems by finding an optimal program, which when implemented will produce a solution to the particular problem. GP is based on Darwin's theory of evolution and iteratively improves an initial population through the processes of evaluation, selection and regeneration to induce a program that produces a solution. Tournament selection is traditionally used to select parents in a GP system. Crossover, mutation and reproduction are commonly used for regeneration. According to Banzhaf et al. [2] programs have been represented using various structures, e.g. parse trees, linear structures, matrices. More recently strings have also been used for this purpose.

The GPA explores a space of programs composed of Sudoku moves so as to identify a sequence of moves that will produce a solution to the puzzle. The GP approach is tested on 1800 puzzles of differing difficulty and where possible the performance of the GPA is compared to other methods used to solve Sudoku puzzles. The study revealed that evolution was not needed and random combinations of the heuristics were found to easily solve Sudoku problems at various difficulty levels. However, for the more difficult problems, including AI Escargot, the genetic programming approach was found to reduce the runtimes needed to generate a solution producing program.

Although this paper focuses on Sudoku puzzles the heuristic moves and approach presented can be applied to a number of puzzles similar to Sudoku including CalcDuko, Killer Sudoku, Futoshiki, Kakuro, Killer Sudoku, Alphadoku, Irregular

[1] Email: pillayn32@ukzn.ac.za

Sudoku, Latin Squares, Sudoku Dragon, Word Sudoku, Jigsaw Sudoku and Samurai Sudoku. The research presented in this paper also contributes to the recent shift in the area of combinatorial optimization to search a space of move operators rather than a solution space to find solutions [27] and provides an example of the derivation of heuristic moves for a particular domain.

A description of Sudoku puzzles and previous methods that have been used to solve these puzzles is presented in Section 2. The heuristic moves and genetic programming approach and the methodology employed to evaluate this approach are described in sections 3 and 4 respectively. The performance of the GPA is discussed in section 5. A summary of the findings of the study and future work are presented in section 6.

2. SUDOKU PUZZLES

This section provides an introduction to Sudoku puzzles and provides an overview of the different methods that have been evaluated for solving Sudoku.

2.1 An overview

Sudoku was developed by Howard Garns and the first Sudoku puzzle become popular in Japan and was named Sudoku meaning single numbers [6]. The grid for an $n \times n$ Sudoku is comprised of $n \times n$ (i.e. n^2) rows, columns and squares. For example a 3 × 3 Sudoku puzzle consists of 3 times 3, i.e. nine rows, columns and blocks. This paper focuses on the 3 × 3 Sudoku puzzle, an example of which is illustrated in Figure 1. The intersection of each row and column forms a cell which contains a number from 1 to 9. For example, the intersection of the first row and column in Figure 1 is a cell containing a 4. The intersection of the first row and second column is an empty cell. Each row and column contains three blocks. In Figure 1 the outlines of these blocks are highlighted. Each block encompasses 3 rows and 3 columns.

Sudoku puzzles are a subset of Latin squares [6, 15, 16]. Solving the puzzle involves placing the digits 1 to 9 so that these digits occur only once in each row, column and block. Depending on the difficulty of the puzzle, the time needed to solve a Sudoku puzzle ranges from O(1) [3] to non-polynomial time. Hence, Sudoku is an NP-complete problem [5, 13, 28]. Various scales have been used to categorize Sudoku problems with respect to difficulty and these appear to vary from one study to the next. The most commonly used scale describes problems as easy, medium, hard and super hard [1, 5]. There has also been an attempt to standardize the scale used according to the grading system used for certain Japanese martial arts such as karate and judo. The categories used are white belt, green belt, brown belt, black belt, second degree black belt and third degree black belt, with white belt being the easiest level and third degree black belt the hardest. In 2006 a Sudoku puzzle named 'AI Escargot" was deemed to be the most difficult Sudoku puzzle [11]. Certain techniques, such as the SAT technique unit resolution with failed literal propagation, were unable to solve this problem [11]. The difficulty of a Sudoku puzzle can be measured by the positioning of the numbers on the grid and not the number of givens on the grid [23]. Pelanek [28] proposes an alternative to assess the difficulty of Sudoku puzzles, namely, the difficulty of steps required to solve the problem and the dependency between the steps.

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

Figure 1. Sudoku puzzle

Based on the literature, there appears to be three main areas of research into Sudoku problems, namely, generating Sudoku puzzles [4], assessing the difficulty of Sudoku puzzles [4, 11, 28] and solving Sudoku puzzles. The research presented in this paper focuses on the latter and an overview of the methods used to solve Sudoku puzzles follows.

2.2 Solving Sudoku

There has been a fair amount of research into solving Sudoku puzzles. The earlier work in this field examined the use of brute force methods for solving Sudoku. While these methods worked well for easy problems they were not very effective for difficult puzzles and had high runtimes.

Evolutionary and genetic algorithms have been the most frequently researched method for solving Sudoku. A lot of this research has focused on investigating the use of effective genetic operators, i.e. crossover and mutation for this domain. While these algorithms appear to perform generally well for this domain they do not scale well with either no success or lower success rates for more difficult Sudoku puzzles. Furthermore, the use of grammatical evolution in a genetic algorithm to produce instructions for solving Sudoku puzzles was not successful at solving this problem.

Hybrid approaches have also proven to be fairly successful in solving Sudoku puzzles. Comparative studies examining the performance of different methods in solving the same set of problems have also been conducted.

Certain methods, such as genetic algorithms, have been found to have high runtimes in solving Sudoku puzzles. The use of parallel processing has been investigated as a means of reducing runtimes.

The following sections give a brief overview of the various methods that have been used for solving Sudoku. The reader is referred to the literature for more detail.

2.2.1 Brute force methods

Kovacs [15] outlines some of the brute-force methods used to solve Sudoku puzzles. The simplest method randomly assigns numbers to the empty cells and checks whether the completed puzzle is a solution. If not, this process is repeated until a solution is found. Clearly, this method can be very time consuming. A similar method is to generate all possible combinations for the empty cells. This can only be done for easy problems. Kovacs also suggests creating a search space for the puzzle and applying searches such as a depth-first search with backtracking.

2.2.2 Evolutionary and genetic algorithms

A fair amount of research has been conducted into the use of evolutionary and genetic algorithms to solve Sudoku puzzles. Evolutionary and genetic algorithms are based on Darwin's theory of evolution and as such evolve an initial population through the processes of evaluation, selection and regeneration to find a solution. Genetic operators such as reproduction, mutation and crossover are generally used for regeneration purposes.

Aid [1] uses an evolutionary algorithm to solve easy, medium and hard Sudoku problems. An individual in the population is composed of nine chromosomes, each representing the nine rows of the Sudoku board. The algorithm was tested on four Sudoku problems with ten runs being performed for each problem. The EA achieved a success rate of a 100% for the easy problem, 80% for the medium and 20% for the hard problem.

Das et al. [5] employ a retrievable genetic algorithm to solve Sudoku puzzles. This GA differs from the standard GA in that the population is reinitialized after a set number of generations in an attempt to escape local optima. The GA was tested on 9 sample problems, 25 easy, 25 medium and 25 hard problems. A 100 runs were performed for each problem. The retrievable GA was also able to find solutions for all 25 easy, 25 medium and 25 hard problems, with lower success rates for the hard problems.

Galvan-Lopez and O'Neill [7] study the locality of genetic operators in an evolutionary algorithm for solving the Sudoku problem. Four operators, namely, one cycle crossover, multi-cycle crossover, partially matched crossover and uniform swap crossover were studied. The evolutionary algorithm is used to solve 6 Sudoku problems. The partially matched crossover and uniform swap crossover were found to be the most successful at finding the global optimum.

Mantere and Koljonen [21] also use a genetic algorithm to solve Sudoku puzzles. The GA was tested on five Sudoku puzzles published in the local newspaper and five generated puzzles. The levels of difficulty of the newspaper puzzles were easy, challenging, difficult and super difficult. Solutions were found for all puzzles with the success rate for the easy puzzles being much higher than that for the other puzzles. In later work Mantere and Koljonen [22] incorporate cultural learning in a GA to solve Sudoku puzzles.

Moraglio et al. [24] propose geometric crossover operators for use with an evolutionary algorithm for solving Sudoku. The algorithm was tested on five problems, three easy, one medium and one hard and was able to find solutions for all problems except the medium problem.

Hamnes and Julstrom [10] introduce an iterated mutation operator for use in an evolutionary algorithm to solve Sudoku puzzles. The approach was applied to 100 Sudoku puzzles.

Sato [36] proposes crossover and mutation operators that preserve building blocks by using local search, for use in a GA to solve Sudoku. The GA with the proposed operators was used to solve two problems for the following levels of difficulty: easy, intermediate and difficult and three difficult Sudoku problems provided in the literature. The approach found solutions for all problems. A 100 runs were performed for each problem. A success rate of 100% was obtained for the easy and intermediate problems, 96% for the difficult problems, 98%, 83% and 58% for the three difficult problems from the literature.

Nicolau and Ryan [26] implement a GA using grammatical evolution (GAuGE) to obtain solutions to the Sudoku problem. In this case a GA is employed to find an optimal set of instructions which, when executed, will solve the puzzle. The approach was not able to solve all the problems it was tested on.

Sato et al. [37, 38] attempt to reduce the runtimes of GAs in solving Sudoku puzzles by taking a multi-core approach to the implementation of the GA. A multiple-population coarse-grained GA is used to solve Sudoku problems.

2.2.3 Harmony search

Geem [9] evaluates harmony search as a means of solving Sudoku. This algorithm emulates different behaviours of musicians including random play, memory-based play and pitch-adjusted play. The harmony algorithm was able to solve an easy Sudoku puzzle in 9 seconds. The algorithm was unable to solve the hard problem which it was also applied to.

2.2.4 Boolean satisfiability methods

Methods that have been successfully used to solve the Boolean satisfiability problem have also been applied to this domain. Henz and Truong [11] test the following SAT propagation techniques to solve Sudoku problems, namely, failed literal propagation, binary failed literal propagation, hyper-binary resolution and a variation of grid analysis. These methods were tested on the AI Escargot problem and 20 other problems described as the most difficult Sudoku problems. All the methods except failed literal propagation were able to find solutions for all problems. The grid analysis variation proved to be the most efficient with the shortest runtimes.

2.2.5 Bee colony optimization

Bee colony optimization has also been applied to solving Sudoku puzzles [12]. Each Sudoku puzzle is represented as a two-dimensional array. The algorithm used solves the puzzle by emulating the process used by bees when foraging for food. Pacurib et al. [28] have also successfully applied a bee colony optimization algorithm to solve Sudoku puzzles.

2.2.6 Simulated annealing

Lewis [16] employs a process using simulated annealing to solve Sudoku puzzles published in daily UK newspapers. The process begins by randomly assigning numbers to empty cells in such a manner that each square contains just one occurrence of each number. This initial potential solution is then improved using simulated annealing. The approach found solutions for all puzzles it was applied to.

2.2.7 Particle swarm optimization

Moraglio et al. [23] evaluate geometric particle swarm optimization as a means of solving Sudoku puzzles. This approach was able to find a solution to the problem it was tested on but the success rate over fifty runs was not as high as other methods applied to the same problems.

2.2.8 Hybrid approaches

Khan et al. [13] use a combination of message passing and Sinkhorn balancing to solve Sudoku puzzles. This hybrid approach was found to scale well. After a certain number of iterations the message passing process is terminated and a solution check which attempts to guess the missing number is performed. The hybrid was tested on 2356 9x9 puzzles and 44 16x16 puzzles. The approach solved all problems with lower success rates for problems with more unknowns.

Machado and Chaimowicz [19] combine a constraint satisfaction algorithm and simulated annealing to solve Sudoku puzzles. Three approaches are tested, namely, the implementation of two constraint satisfaction algorithms (arc consistency and path consistency), an extension of the first approach using simulated annealing to further improve a potential solution that could not be improved any further using the constraint satisfaction algorithms, the third method again extends the second approach by using a process to fill empty cells on the board prior to the application of simulated annealing. These methods were applied to generated puzzles. All puzzles were solved in less than five seconds by the latter two approaches.

Mullaney [25] employs an ant system to solve Sudoku problems. Each ant uses a tabu search to explore the puzzle space. The approach was tested on a set of 95 hard Sudoku

problems. Ant colonization optimization together with shortest tabu list produced the best results finding solutions for all problems and required 120 minutes to solve each problem.

Deng and Li [7] take a hybrid approach combining genetic algorithms and particle swarm optimization to solve Sudoku puzzles.

2.2.9 Comparative studies

Perez and Marwala [31] compare the performance of a cultural genetic algorithm, repulsive particle swarm optimization, quantum simulated annealing, and a hybrid approach combining a genetic algorithm and simulated annealing, in solving Sudoku. Repulsive particle swarm optimization differs from the standard approach by escaping from local optima by causing particles to repel each other. Quantum simulated annealing incorporates the use of quantum tunneling into simulated annealing. The hybrid approach firstly creates an initial potential solution using the GA. When the GA can no longer improve the potential solution, simulated annealing is used to obtain improvements. All the methods except repulsive particle swarm optimization were able to solve the Sudoku puzzle with the hybrid approach requiring the least amount of time to solve the problem.

In a survey paper on Sudoku Serpen and Greenm [39] present an overview of various methods that have been used to solve Sudoku including a steepest ascent and greedy hill-climbing hybrid, a chromatic polynomial formulation of the Sudoku problem, geometric particle swarm optimization, artificial immune systems, neural networks, simulated annealing, quantum simulated annealing, bee colonization, a constraint propagation formulation of Sudoku, constraint propagation and a depth-first search with backtracking hybrid and linear programming. The constraint propagation and depth-first hybrid was able to solve 95 hard Sudoku problems requiring 0.125 seconds to solve a puzzle. The linear programming approach described by Serpen solved puzzles in 16 seconds. Simulated annealing was found to perform better than quantum simulated annealing solving 14 of the 40 problems it was tested on while quantum simulated annealing could only solve 6 of these problems. Geometric particle swarm optimization and bee colonization obtained higher success rates of 47.5% and 55% respectively.

3. HEURISTIC MOVES AND THE GPA

This section describes the genetic programming approach implemented to solve Sudoku problems. The first section presents components each program will be composed of, namely, Sudoku moves. This is followed by a description of the algorithm used to search the program space.

3.1 Sudoku moves

This section describes nine moves for solving Sudoku. These moves are based on human tactics used to solve Sudoku. Please note that the character in brackets is used to represent the move.

3.1.1 Row move (r)

This move applies a row operator to each row of the Sudoku grid. The pseudo code for the row operator is listed in Figure 2. The row operator attempts to place the missing numbers in the row. For example, consider the first row in the grid in Figure 1. The row operator will attempt to place the missing numbers, namely, 2, 3, 6, 7 and 8. The numbers 2, 6 and 7 have more than one empty cell as an option and therefore are not placed. The numbers 3 and 8 can be placed. The number 3 cannot be placed in the first two available cells as the block these cells occur in contains a 3. Similarly, a 3 cannot be placed in the next two available cells as the columns that these cells occur in already have a 3. Thus, the 3 is placed in the last available cell. The number 8 cannot be placed in the first two available cells and the last two available cells in this row as the blocks containing these cells already have the number 8. The 8 is placed in the third empty cell in this row. The row operator is applied to the nine rows of the grid sequentially. The effect of applying this heuristic to the grid in Figure 1 is illustrated in Figure 3.

3.1.2 Column move (*c*)

Like the row move the column move applies the column operator to each column sequentially. The column operator also attempts to place the missing numbers in the particular column. An application of the column heuristic to the Sudoku grid in Figure 1 is depicted in Figure 4.

```
Procedure row_op(row)
Begin
 for  nums →1 to 9
  begin
   fin → false
   for cells → 1 to available spaces in row AND NOT fin
    begin
     if (the column containing the next available cell does not contain
       nums AND the block containing the next available cell does
       not contain nums)
      if (a cell has not been found yet)
       cell → next available cell
      else
       fin → true
    endfor

   //If fin is not true only one cell has been found for nums

    if (!fin)
     Update the grid to store nums in cell
  endfor
End
```

Figure 2. Pseudo code for the row operator

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

Figure 3. Application of the row operator

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

Figure 4. Application of the column move

3.1.3 Block move (*b*)

This move performs a similar function to that of the row and column heuristics and applies the block operator to each block sequentially. The operator is applied from left to right. Figure 5 illustrates an application of the block operator to Figure 1.

3.1.4 3-row move (w)

This move applies a 3-row operator to each row that contains 3 empty cells. The operator is applied sequentially. If there are three spaces in a row and two of the missing numbers cannot be placed in a particular cell because the column or block containing the cell already has these numbers, the third number has to be placed in this cell. For example, consider the seventh row of Figure 5. The numbers missing in this row are 1, 2 and 8. The numbers 1 and 2 cannot be placed in the third empty cell in the row. Thus, 8 should be placed in this cell.

3.1.5 3-column move (*l*)

The 3-column operator is applied to each column sequentially. The 3-column operator performs the same function as the 3-row operator for columns. For example, consider the fifth column in Figure 5. The numbers 1, 4 and 9 must be added to this column. The numbers 4 and 9 cannot be added to the third empty cell in this column thus 1 must be placed in this column.

3.1.6 3-block move (*k*)

This move performs the same function as the 3-row and 3-column moves on blocks. The 3-block operator is applied to the blocks sequentially from left to right. For example, there are 3 available spaces in the sixth block in Figure 5. The numbers missing from this block are 6, 7, and 8. Consider the third cell. The numbers 7 and 8 cannot be placed in this cell as both these numbers already occur in the column the cell lies in. Hence, 6 must be placed in this cell.

3.1.7 Try-row move (*3*)

The try-row move operator applies the try-row operator to each row sequentially. The try-row operator works through the missing numbers in the row until it finds a number for which there are two empty cell options the number can be placed in. A cell option would be a cell which does not lie in a column or block that already contains the missing number. Note that one of the two cells will be the correct position for the number but at this stage it is not known which one is the right cell. Thus one of the cells is randomly selected and the number is allocated to it. Hence, this move can contribute to generating a solution or may prevent a solution from being generated if the incorrect cell

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

Figure 5. Application of the block move

```
Procedure try_row_op(row)
Begin
  fin → false
  for  nums → NOT fin AND 1 to 9
    begin
      if (there are two free cells in which nums can be placed)
       begin
         cell → randomly chose between the two cell options
         fin → true
         Update the grid to store nums in cell
       endif
    endfor
End
```

Figure 6. Pseudo-code for the try-row operator

is chosen. The pseudo code for this operator is illustrated in Figure 6.

For example, consider the fifth row in Figure 5. The number 2 can be placed in the fourth or fifth cells in this row. The try-row operator will randomly choose one of the cells to place the number 2 in.

***c**lkb*

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

*cl**k**b*

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

*cl**k**b*

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

*clk**b***

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

Figure 7. Application of *clkb* in solving the Sudoku puzzle

3.2 GP approach

The GPA employs a generational control model, which iteratively refines an initial population through the processes of evaluation, selection and regeneration, to search the program space. These processes are described below.

3.2.1 Initial population generation

Each element of the population is a program consisting of one or more characters representing a Sudoku move. For example, *2wllc* is an element of the population. In this case the Sudoku puzzle is solved by firstly applying the try-block move, followed by the 3-row move, the 3-column move twice and finally the column move. The length of each program is chosen to be between 1 and 50. This value was chosen given the results of trial runs conducted. The initial population has a variation of 100%, i.e. duplicates are not permitted. The fitness of each program is defined in terms of its effectiveness in solving the Sudoku puzzle. This is discussed in the following section.

3.2.2 Evaluation and selection

Each program is evaluated by using it to solve a Sudoku puzzle. For example, suppose that an individual of the population is *clkb* and the Sudoku puzzle to be solved is that depicted in Figure 1. Figure 7 illustrates how *clkb* is applied to solving this puzzle.

The fitness of the program is the number of missing numbers in the Sudoku grid after the program has been applied to it. In this example the fitness of *clkb* is 6. A fitness of 0 indicates that application of the moves in the string have solved the puzzle. Thus, the GPA aims to minimize the fitness. Note that there is no need to check for duplicates when calculating the fitness as the moves are defined to ensure that duplication of numbers cannot occur.

Tournament selection is used to choose parents for regeneration. This method creates a subset of individuals which are randomly selected from the population. The size of the subset is problem dependant. The fittest individual, in this case the individual with the lowest fitness, of the subset is the winner of the tournament and is a parent of the next generation. Selection is with replacement, and thus an individual may be chosen to be a parent more than once.

3.2.3 Genetic operators

The mutation and crossover operators are used to create the offspring of each generation. As mutation and crossover can be reduced to reproduction, the reproduction operator is not explicitly applied.

The mutation operator firstly randomly selects a mutation point in a copy of the parent. The move at this point is replaced with a move that is randomly selected from the moves. For example suppose that *kwllc* is a copy of the chosen parent and 3 has been randomly selected as the mutation point. The move *l* will be replaced by a randomly chosen move. A possible offspring is *kwklc*, i.e. the 3-block move (*k*) was randomly selected to replace the 3-column move (*l*). Note that *l* could be chosen as a replacement move in which case the mutation operator will basically perform reproduction.

The crossover operator is applied to two parents. Crossover points are randomly selected in copies of both the parents. Both programs are "crossed over" at the crossover points. The fitter of the two offspring is returned as the result of the operation. This operator varies from the standard crossover operator used in GP as it returns the fitter of two offspring. It was evident from trial runs conducted that it was more effective to return the fitter offspring instead of both offspring.

Suppose that *2wllc* and *clkb* are copies of the selected parents and *2* is randomly selected as a crossover point in the first parent and *3* in the second. The corresponding offspring are *2kb* and *clwllc*. Both these offspring are applied to solving the Sudoku problem and the fitter of the two is returned as the offspring. If the offspring have the same fitness the first offspring is returned. Suppose that *2wllc* and *clwllc* have been chosen as parents and *3* is the crossover point in the first parent and *4* in the second parent. Then in this case crossover is reduced to reproduction.

4. EXPERIMENTAL SETUP

The GPA was tested on the following 1800 problems of varying difficulty rated according to the Japanese martial art grading taxonomy with white belt representing easy problems and 3rd degree black belt representing the most difficult problems:

- 300 white belt problems (Rios 2005)
- 300 green belt problems (Rios 2005)
- 300 brown belt problems (Rios 2005)
- 300 2nd degree black belt problems (Longo 2005)
- 300 3rd degree black belt problems (Longo 2007)

These problems were obtained from the Sudoku puzzle books compiled by Michael Rios and Frank Lango.

Most of the methods proposed by previous studies on Sudoku were evaluated on inaccessible sources obtained from newspapers and books. The following problems cited in previous work were used to compare the GPA to other methods applied to the same domain:

- The most difficult problem, namely, AI Escargot presented in (Henz and Truong 2009).
- The 20 most difficult problems tested by Henz and Truong (2009).
- The 9 problems used by Sato (2010).
- The easy and hard problem evaluated in (Geem 2007).
- The problem presented by Perez (2008).

Values of the genetic parameters were determined empirically by performing trial runs. These values are listed in Table 1. Using fairly small values like 20 for the population size resulted in the GPA not being able to find solutions for the more difficult problems. While slightly larger population sizes resulted in programs producing solutions being evolved, the number of generations and runtimes needed for the harder problems were higher than when using a value of 500. Larger population sizes did not result in any improvements in performance.

Table 1. Genetic parameter values

Parameter	Value
Population size	500
Number of generations	50
Tournament size	4
Mutation rate	0.5
Crossover rate	0.5

Table 2. Simulation results

Problem Set	Success Rate	Min. No. of Generations	Max. No. of Generations
White belt	100%	0	1
Green belt	100%	0	1
Brown belt	100%	0	2
Black belt	100%	0	7
2nd degree black belt	100%	0	9
3rd degree black belt	100%	0	8

The genetic programming approach was implemented in Java using JDK 1.6.0 and all simulations were run on a Windows XP machine with a 1995 MHz Intel processor.

5. RESULTS AND DISCUSSION

Due to the stochastic nature of genetic programming 30 runs, each with a different random number generator seed, was performed for each problem. The results of the simulations are listed in Table 2.

The success rate refers to the percentage of the 30 runs solving all 300 problems (300 x 30) in the set. The table also lists the minimum and maximum number of generations needed to find a solution over the 30 runs for all 300 problems in the group. For Sudoku puzzles of all levels of difficulty the minimum number of generations needed to find a solution is 0. This means that no evolution was needed and a program producing a solution was found in the initial population. The maximum number of generations needed to produce a solution varies, depending on the difficulty of the problem. The maximum and minimum runtimes are listed in Table 3.

Table 4 and Table 5 show the distribution of the number of generations needed to find a solution for the 300x30 runs conducted for each group of problems of differing difficulty. From these tables it is evident that as the difficulty of the puzzle increases there is more of a distribution over the number of generations needed to find a solution.

Table 3. GPA Runtimes

Problem Set	Minimum Runtime	Maximum Runtime
White belt	< 1 ms.	407 ms.
Green belt	< 1 ms.	422 ms.
Brown belt	< 1 ms.	578 ms.
Black belt	< 1 ms.	625 ms.
2nd degree black belt	< 1 ms.	1281 ms.
3rd degree black belt	< 1 ms.	1219 ms.

Table 4. Number of generations distribution over 30 runs

Generation	White	Green	Brown	Black
0	8992	8985	8923	8881
1	8	15	63	96
2	0	0	14	11
3	0	0	0	5
4	0	0	0	3
5	0	0	0	0
6	0	0	0	1

Table 5. Number of generations distribution over 30 runs continued

Generation	Black 2nd Degree	Black 3rd Degree
0	7644	7780
1	1053	968
2	188	170
3	60	47
4	33	19
5	13	9
6	4	5

Table 6. Examples of programs evolved

Level of Difficulty	Programs
White	cckbr431,2kr34brr14223w1, kbrcw1kbwcr33ww3r, 3rr4b2wbb2334, 1rbkc2kwr
Green	r324krw1c4k4bkb11k1, 33rcw3cww14, b41bcc3rrrwk, cwcrkrrbcrb, k1crccr
Brown	b1bwcrcc2cr44k23kccw4, 3w22bccr1crbr42, wwc1k131ckbbk, kccccww4r1rbw, 3b12rkw1rrkww
Black	4k4442b1c4w1w3cwwrb4, br144wbcrk14c, 1bk1ckbbkcbk1c13w1b3 wbkrkkwc2rw4rw2kkwb1, 2rk441r3r3r1wwk311k
Black 2	31b2brk43wbrkc1b3kc3c, 413rbb31kr3bwbbck, kwr42bc1rwcbb132r, rc11b1cwc3223rwk2r4kb4b43, 1wkc3b2brwbb
Black 3	3bbc33w3kc22kbcrb13c1c42k, k1r3rbw3b41c1w1cbkkr, rcr11b2423rr13324kc114, 41r224k2crbrwbrc, bw4r24cb1b121crrbc

Programs producing a solution for each of the runs for a problem were found to be different. Similarly, programs that produced solutions for the different problems also differed. This can be expected as more than one combination of moves can lead to a solution. Furthermore, due to the stochastic nature of the approach and some of the operators, the approach appears to adapt to random noise and hence different optimal programs are produced for different runs. Thus the programs are disposable, i.e. a program is generated for a particular run.

Table 6 lists examples of the programs evolved for each group of problems. As the difficulty of the puzzle increases the size of the program appears to increase, i.e. more moves are needed to solve the problem.

The performance of the GPA was compared to that of the following methods used to solve Sudoku problems:

- The harmony search algorithm implemented by Geem (2007). This method was applied to an easy and hard problem.
- The SAT methods used by Henz and Truong (2009) to solve AI Escargot and 20 difficult Sudoku puzzles.
- The cultural genetic algorithm, quantum simulated annealing and the genetic algorithm and simulated annealing hybrid tested by Perez and Marwala (2008).
- A genetic algorithm applied by Sato (2010) to solve 9 Sudoku puzzles.

These methods are described in section 2. The GPA was applied to same set of problems that each of these methods was tested on. Thirty runs were performed for each problem.

The harmony algorithm implemented by Geem (2007) took a minimum of 9 seconds to solve the easy problem. The GPA was able to solve this problem in less than a millisecond. For all 30 runs zero generations were needed to generate a program that produced a solution. The harmony search was unable to

solve the hard Sudoku problem. The GPA was able to solve this problem in a minimum time of 15 milliseconds. As with the easy problem no evolution was needed to solve the puzzle and a program producing a solution was found in the initial population for all 30 runs.

Henz and Truong (2009) evaluate various SAT methods, namely, failed literal propagation, binary failed literal propagation, hyper-binary resolution and a variation of grid analysis. The minimum runtime needed to produce a solution for AI Escargot was 76 seconds. Runtimes for the 20 difficult problems are not specified. All of these methods with an exception of failed literal propagation were able to find solutions to AI Escargot and the 20 difficult problems. The GPA was also able to find solutions for all of these problems. The success rate, minimum and maximum number of generations needed to find a solution, and minimum and maximum runtimes (seconds (s) or milliseconds (ms)) over the 30 runs are listed in Table 7. Depending on the difficulty of the problem, the minimum and maximum number of generations needed to evolve a program producing a solution varies between 0 and 2 and 2 and 20 respectively. From Table 7 it appears that the GPA found problems 7, 15, 17 and 20 just as challenging to solve as AI Escargot. Hence, these problems can be categorized as being of similar difficulty given the performance of the GPA.

Perez and Marwala (2008) compare the performance of quantum simulated annealing, a cultural genetic algorithm and a hybrid combining a genetic algorithm and simulated annealing. All three approaches were successful at solving the same Sudoku problem. The minimum time required by quantum simulated annealing was 65 seconds, the cultural genetic algorithm 28 seconds and the hybrid 1.447 seconds. The GPA was tested on the same problem. On all runs a program producing a solution was found in the initial population, i.e. evolution was not needed to derive a program that produces a solution. The minimum runtime taken by the hyper-heuristic to solve the problem was less than a millisecond.

Sato [36] implemented a genetic algorithm with building block preserving genetic operators, namely, mutation and crossover incorporating local search, to solve the Sudoku problem. The genetic algorithm was tested on two easy, two intermediate, two difficult Sudoku puzzles and three difficult problems from the literature. The GA was able to solve all 6 problems with the following success rates: 100% for the easy and intermediate problems and the first difficult problem, 96% for the second difficult problem and 98%, 83% and 58% for the three difficult problems from the literature. The runtimes of the GA were not specified. The GPA was able to solve all 9 problems. The success rate, minimum and maximum number of generations and minimum and maximum runtime for each problem is listed in Table 8. Given these results, we note that the GPA performed better than other approaches in solving Sudoku puzzles.

From the results presented in this section it is evident that for a majority of the problems evolution was not needed and programs producing solutions were found in the initial population. For the more difficult problems, namely AI Escargot and a number of the problems presented by Henz and Truong (2009) in Table 7, at least one generation was needed to evolve a solution producing program. In order to ascertain

Table 7. GPA performance on the problems presented in Henz and Truong (2009)-Caption Index: A- Success rate, B-Minimum number of generations, C- Maximum number of generations, D - Minimum runtime, E - Maximum runtime

Problem	A	B	C	D	E
AI Escargot	100%	1	15	1 s	4 s
Problem 1	100%	0	11	31 ms	5 s
Problem 2	100%	1	11	281 ms	4 s
Problem 3	100%	0	9	156 ms	5 s
Problem 4	100%	0	8	16 ms	4 s
Problem 5	100%	0	7	141 ms	3 s
Problem 6	100%	0	9	15 ms	3 s
Problem 7	100%	2	20	62 ms	8 s
Problem 8	100%	0	12	219 ms	4 s
Problem 9	100%	1	12	406 ms	4 s
Problem 10	100%	0	14	47 ms	5 s
Problem 11	100%	0	4	< 1ms	1 s
Problem 12	100%	1	13	750 ms	4 s
Problem 13	100%	0	12	31 ms	4 s
Problem 14	100%	1	9	375 ms	3 s
Problem 15	100%	1	19	1.5 s	4 s
Problem 16	100%	0	5	< 1 ms	2 s
Problem 17	100%	1	18	687 ms	8 s
Problem 18	100%	0	2	< 1ms	1 s
Problem 19	100%	0	5	15 ms	2 s
Problem 20	100%	2	17	594 ms	5 s

Table 8. GPA performance on the problems presented in Sato (2010) -Caption Index: A- Success rate, B-Minimum number of generations, C- Maximum number of generations, D - Minimum runtime, E - Maximum runtime

Problem	A	B	C	D	E
Easy (No. 1)	100%	0	0	< 1 ms	16 ms
Easy (No. 11)	100%	0	0	< 1 ms	16 ms
Intermediate (No. 27)	100%	0	0	< 1ms	78 ms
Intermediate (No. 29)	100%	0	0	N< 1ms	31 ms
Difficult (No. 77)	100%	0	1	< 1ms	594 ms
Difficult (No. 106)	100%	0	2	< 1 ms	531 ms
S1	100%	0	3	31 ms	1 s
S2	100%	0	13	93 ms	4 s
S3	100%	0	1	< 1 ms	1 s

Table 9. Runtimes for random generation of solution program

Problem	Min. Runtime (secs)	Max. Runtime (secs)
AI Escargot	3	12
Problem 2	2	60
Problem 7	3	60
Problem 9	1	60
Problem 12	1	60
Problem 14	1	22
Problem 15	1	60
Problem 17	1	60
Problem 20	1	15

whether genetic programming is necessary to evolve programs producing solutions for these problems, randomly created programs were generated for each of these problems iteratively until a program producing a solution to the problem was induced. Using this process solution producing programs were found for all the difficult problems. However, the runtimes needed to generate these programs were higher than that of the genetic programming approach. The minimum and maximum runtimes needed for each of these problems over ten runs, each using a different random number generator seed, is listed in Table 9. From the results presented it appears that more time was needed to solve AI Escargot compared to some of the other problems such as problems 2, 7, 9, 12, 15 and 17. However, given the stochastic nature of the process and that only ten runs were performed, concrete conclusions cannot be drawn from this.

6. CONCLUSION

Attempts at automating the process of solving Sudoku has led to research into the effectiveness of artificial intelligence and optimization techniques for this purpose. This paper firstly presents a set of heuristic moves for solving Sudoku and evaluates a genetic programming approach as a means of inducing programs composed of these moves to solve Sudoku problems. The GPA searches a space of programs composed of Sudoku moves. The moves are based on human strategies for solving Sudoku puzzles. The genetic programming approach was successfully used to solve 1800 Sudoku puzzles of different levels of difficulty. The study revealed that evolution was only needed to reduce the runtimes associated with producing solutions for more difficult problems, namely AI Escargot and eight of the difficult Sudoku problems made available by Henz and Truong (2009). For a majority of the problems solution programs were found during initial population generation, i.e. they were randomly generated and evolution was not required.

It was found that as the difficulty of the puzzles increase there is a wider distribution of the number of generations needed to evolve a program to produce a solution over the 30 runs performed for each problem. The size of the programs evolved also appeared to increase with problem difficulty.

More than one optimal program was found for each problem. This can be expected as a number of different combinations of the low-level heuristics can result in a Sudoku puzzle being solved. An empirical comparison of the performance of the genetic programming approach and other methods used to solve Sudoku was also conducted.

The performance of the GPA was found to be comparable to and in some cases an improvement on other techniques applied to the same puzzles. Given the success with Sudoku, future work will examine the effectiveness of the genetic programming approach in solving other logical puzzles such as Killer Sudoku and Kakuro. Future work will also investigate the effectiveness of the different heuristic moves in creating solutions to Sudoku problems.

7. ACKNOWLEDGEMENTS

The author would like to thank the reviewers for their constructive comments and suggestions.

REFERENCES

[1] Aid, T. 2011. Evolutionary Sudoku Solver, http://www.scienceandatheism.com/wp-content/uploads/2011/05/EvolutionarySudokuSolver-ThaddeusAid.pdf.

[2] Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D. 1998. *Genetic Programming – An Introduction – On the Automatic Evolution of Computer Programs and its Applications*, Morgan Kaufmann Publishers, Inc.

[3] Brouwer, A.E. 2006. Sudoku Puzzles and How to Solve them. *Nieuw Archief voor Wiskunde* (Vijfde Serie), 7 (2006), 258–263.

[4] Chang, C, Zhou, F., Sun, Y. 2008. *hSolve: A Difficulty Metric and Puzzle Generator for Sudoku*, http://web.mit.edu/yisun/www/papers/sudoku.pdf. Accessed 16 January 2012.

[5] Das, K.N., Bhatia, S., Puri, S., Deep, K. 2012. A *Retrievable GA for Solving Sudoku Puzzles*. Technical Report, http://www.cse.psu.edu/~sub194/papers/sudokuTechReport.pdf.

[6] Delahaye, J.P. 2006. The Science Behind Sudoku. *Scientific American*, 81-87.

[7] Deng, X., Li, Y. 2011. A Novel Hybrid Genetic Algorithm for Solving Sudoku Puzzles. *Optimization Letters*, October 2011, 1-17, doi:10.1007/s11590-011-0413-0.

[8] Galvin-Lopez, E., O'Neill, M. 2009. On the Effects of Locality in a Permutation Problem: The Sudoku Puzzle. In proceedings of the 2009 *IEEE Symposium on Computational Intelligence and Games*, 80-87, IEEE Press.

[9] Geem, Z.W. 2007. Harmony Search Algorithm for Solving Sudoku. In *Proceedings of KES 2007*, 371-378.

[10] Hamnes, D.O., Julstrom , B.A. 2008. Iterated Mutation in an Evolutionary Algorithm for Sudoku. In *Proceedings of Artificial Intelligence and Applications*, 595.

[11] Henz, M., Truong, H.M. 2009. SUDOKUSAT – A Tool for Analyzing Difficult Sudoku Puzzles. *Tools and Applications for Artificial Intelligence, Studies in Computational Intelligence*, Vol. 166, 25-35.

[12] Kaur, A., Goyal, S. 2011. A Survey on the Applications of Bee Colony Optimization Techniques. *International Journal on Computer Science and Engineering (IJCSE)*, Vol. 3, No. 8, August 2011, 3037-3046.

[13] Khan, S. Jabbari, S., Jabbari, S., Ghanbarinejad, M. 2011. *Solving Sudoku Using Probabilistic Graphical Models*. http://ebookbrowse.com/solving-sudoku-using-probabilistic-graphical-models-pdf-d139791661. Accessed 16 January 2012.

[14] Koza, J. R. 1992. *Genetic Programming I: On the Programming of Computers by Means of Natural Selection*, MIT Press.

[15] Kovacs, T. 2008. *Artificial Intelligence Through Search: Solving Sudoku Puzzles*, 12 November 2008, Technical Report, Department of Computer Science, University of Bristol, http://www.cs.bris.ac.uk/Publications/Papers/2000948.pdf.

[16] Lewis, R. 2007. Metaheuristics can Solve Sudoku Puzzles. *Journal of Heuristics*, Vol. 13, Issue 4, 387-401.

[17] Longo, F. 2005. *2nd Degree Black Belt Sudoku*. Sterling Publishing Co., New York.

[18] Longo, F. 2009. *3rd Degree Black Belt Sudoku*. Sterling Publishing Co., New York.

[19] Machado, M.C., Chaimowicz, L. 2011. Combining Metaheuristics and CSP Algorithms to Solve Sudoku. In

Proceedings of the Brazilian Symposium on Computer Games and Digital Entertainment (SBGames '11), http://www.sbgames.org/sbgames2011/proceedings/sbgames/papers/comp/full/14-92261_2.pdf. Accessed 16 January 2012.

[20] Mantere, T., Koljonen, J. 2006. Solving and Rating Sudoku Puzzles with Genetic Algorithms. In *New Developments in Artificial Intelligence and Semantic Web, Proceedings of the 12th Finnish Artificial Intelligence Conference STeP 2006*, 86-92.

[21] Mantere, T., Koljonen, J. 2007. Solving, Rating and Generating Sudoku Puzzles with GA. In *Proceedings of the 2007 IEEE Congress on Evolutionary Computation*, September 2007, 1382-1389.

[22] Mantere, T., Koljonen, J. 2008 Solving and Analyzing Sudokus with Cultural Learning. In *Proceedings of the 2008 IEEE Congress on Evolutionary Computation*, 1-6 June 2008, Hong Kong, September 2008, 4053-4060.

[23] Moraglio, A., Togelius, J. 2007. Geometric Particle Swarm Optimization for the Sudoku Puzzle. In *Proceedings of GECCO '07*, July 7-11, 118-125, London, England, ACM Press.

[24] Moraglio, A., Togelius, J., Lucas, S. 2006. Product Geometric Crossover for the Sudoku Puzzle. In *Proceedings of the IEEE Congress on Evolutionary Computing (CEC 2006)*, 476-483.

[25] Mullaney, D. 2006. *Using Ant Systems to Solve Sudoku Problems*. http://ncra.ucd.ie/comp40580/ crc2006/mullaney.pdf. Accessed 16 January 2012.

[26] Nicolau, M., Ryan, C. 2006. Solving Sudoku with the GAuGE System. In Collet, P. et al. (eds) *EuroGP 2006*, LNCS 3905, 213-224, Springer-Verlag Berlin Heidelberg.

[27] Ochoa, G., Hyde, M., Curtois, T., Vazquez-Rodriguez, J. A., Walker, J., Gendreau, M., Kendall, G. , McCollum, B., Parkes, A. J., Petrovic, S., Burke, E. K. 2012. HyFlex: A Benchmark Framework for Cross-Domain Heuristic Search. In *Proceedings of the European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP 2012)*. 7245: 136-147.

[28] Pacurib, J.A., Seno, G.M.M., Yusiong, J.P.T. 2009. Solving Sudoku Puzzles Using Improved Artificial Bee Colony Algorithm. In *Proceedings of the Fourth International Conference on Innovative Computing, Information and Control (ICICIC 2009)*, 7-9 December 2009, 885-888.

[29] Pelanek, R. 2011. Difficulty Rating of Sudoku Puzzles by a Computational Model. In *Proceedings of the 2011 FLAIRS conference*, http://www.fi.muni.cz/~xpelanek/publications/flairs-sudoku.pdf.

[30] Pelanek, R. 2011. *Human Problem Solving: Sudoku Case Study*. FIMU-RS-2011-01, FI MU Report Series, January 2011, Faculty of Informatics, Masaryk University Brno.

[31] Perez, M., Marwala, R. 2008. *Stochastic Optimization Approaches for Solving Sudoku*. Technical Report, Cornell University Library, http://arxiv.org/ftp/arxiv/papers/0805/0805.0697.pdf.

[32] Rios, M. 2005. *White Belt Sudoku*. Sterling Publishing Co., New York.

[33] Rios, M. 2005. *Green Belt Sudoku*. Sterling Publishing Co., New York.

[34] Rios, M. 2005. *Brown Belt Sudoku*. Sterling Publishing Co., New York.

[35] Rios, M. 2005. *Black Belt Sudoku*. Sterling Publishing Co., New York.

[36] Sato, Y. 2010. Solving Sudoku with Genetic Operations that Preserve Building Blocks. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games (CIG '10)*, 23-29.

[37] Sato, Y., Hasegawa, N., Sato, M. 2011. GPU Acceleration for Sudoku Solution with Genetic Operations. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, June 2011, 296-303.

[38] Sato,Y., Hasegawa, N., Sato, M. 2011. Acceleration of Genetic Algorithms for Sudoku Solution on Many-Core Processors. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, 407-414, ACM Press.

[39] Serpen, G., Greenm, R.C. 2009. *Survey of the Applications of Artificial Intelligence Techniques to the Sudoku Puzzle, Fundamentals of Intelligence Systems*, Technical Report, December 10, 2009, http://www. parallelcoding.com/wp-content/uploads/2010/01/Full-Paper.pdf.