Development and Analysis of Genetic Algorithms: Sudoku Case Study

Anthony Milton and Cesar Ortega-Sanchez
Department of Electrical and Computer Engineering
Curtin University, Australia
anthony.milton@ieee.org; c.ortega@curtin.edu.au

***Abstract*—This paper discusses the design and subsequent analysis of software implementing a configurable genetic algorithm. The genetic algorithm is primarily targeted towards the solving of Sudoku puzzles. Sudoku is regarded as an ideal test-bed for algorithm development due to the fact that it is a constrained optimisation problem that belongs to the NP-complete class of computational problems. The aim of this paper is to outline the various features currently implemented in the software, and to present preliminary results of an analysis of various aspects of the underlying genetic algorithm.**

I. Introduction

Evolutionary computation techniques have long been recognised as suitable for application to problems that can be formulated as function optimisation problems [1], [2]. Sudoku is one such problem, and hence is attractive to evolutionary computation approaches. Genetic algorithms are but just one evolutionary computation technique that may be applied to the problem of solving Sudoku puzzles.

The primary objective of this paper is to outline the features of a genetic algorithm aimed at solving Sudoku puzzles. This genetic algorithm is implemented in software that may be configured by the user at runtime to meet some desired algorithm configuration. It is the configurable nature of the software that makes it attractive for use in analysing the underlying algorithm. A preliminary analysis of the algorithm is the secondary objective of this work.

II. Background

The origins of the ideas behind the Sudoku puzzle can be traced back to at least 1783 and Euler's idea of Latin squares: square $n \times n$ arrays containing the symbols $1, \ldots, n$ in such a way that each symbol occurs exactly once in each row and each column of an array [3]. In 1956 W.U Behrens introduced a variation on Euler's Latin squares, which he coined "gerechte", which matches what is now recognised as Sudoku [4]. He proposed partitioning a $n \times n$ Latin square array into n regions, with each region containing n cells with the aim being to place the symbols $1, \ldots, n$ into the cells of the array such that each symbol occurs once in each row, column and region.

The notion of creating a puzzle from these ideas can be attributed to Harold Garns, who developed the puzzle concept in the 1970s, with the first puzzle published under the name "Number Place" in 1979 [4], [5], [6]. In 1984 the puzzle concept was introduced into Japan, where it became known as Sudoku, with "su" roughly translating to number and "doku" to single [6]. It was not until 2004 that the puzzle was reintroduced to the western world, and by 2005 it had become a worldwide phenomenon [5].

The appeal of Sudoku lies in the few and simple rules or constraints required to be satisfied for the solution of a puzzle to be found. The constraints that must be met for a solution to be found are as follows:

1) Each of the numbers provided in the unsolved puzzle must remain in their original positions,
2) Each row of the puzzle grid must contain each integer from 1 to 9 exactly once,
3) Each column of the puzzle gird must contain each integer from 1 to 9 exactly once,
4) Each sub-block of the puzzle grid must contain each integer from 1 to 9 exactly once,

As solving a Sudoku puzzle is an exercise in constraint satisfaction, Sudoku is formally considered a constraint optimisation problem, as such, it has been widely studied in constraint programming and satisfiability research [7], [8], [9].

A. Related Work

The application of genetic algorithms to general problem solving is well researched and the suitability of genetic algorithms for solving optimisation problems is widely recognised. Given that this is the case, the application of genetic algorithms to solve Sudoku puzzles is not original, with the first scientific paper detailing such an approach appearing in 2005 [10]. This work successfully showed that genetic algorithms can be used to solve Sudoku puzzles, albeit not very effectively. Although the algorithm used to obtain results and draw conclusions is described in detail in [10], different algorithm configurations consisting of varied algorithm parameters were not used to either explore various configurations, or seek out an optimum configuration.

Since that work was published, a number of researchers have applied genetic algorithms and other evolutionary computation techniques to Sudoku. Mantere and Koljonen followed up their initial work with another study applying genetic algorithms to Sudoku, introducing a number of more advanced algorithm features such as an ageing process and

population regeneration [11]. Studies comparing a number of evolutionary algorithms against one another have also been performed [3], [12].

Jilg and Carter compared genetic algorithms with geometric particle swarm optimisation, bee colony optimisation, artificial immune system, simulated annealing and quantum annealing using Sudoku as the problem of interest [12]. They demonstrated that genetic algorithms performed worse than all other techniques analysed, but admitted that their implementation was not at all optimised and could have yielded drastically different results had they done so. Similar conclusions were drawn by Almog in comparing cultural genetic algorithms with a hybrid between simulated annealing and genetic algorithm and quantum simulated annealing techniques [3].

Although evolutionary algorithms are recognised as being well suited to constraint optimisation problems of a combinatorial nature, such as Sudoku, there are far more appropriate methods for simply arriving at a solution for Sudoku puzzles of small dimensionality. Almog [3] notes that there are many classical Sudoku-solving algorithms that outperform evolutionary approaches. This notion is supported by Moraglio *et al.* [13] who makes the argument that 'evolutionary algorithms are not the best technique to solve Sudoku because they do not exploit systematically the problem constraints to narrow down the search.'

In spite of the apparent unsuitability of using evolutionary algorithms to solve Sudoku puzzles, the use of Sudoku puzzles as the problem of interest for algorithm development holds a great deal of merit [11], [13]. This merit can be attributed to the fact that Sudoku is a non-trivial, computationally complex problem that has been shown to belong to the NP-complete class of problems [14]. It is true, however, that for small puzzle dimensionality, such as the standard 9×9 puzzle, Sudoku puzzles are trivial to solve, but as puzzle dimension increases evolutionary algorithms are likely to be of great value in exploring the solution of Sudoku problems [3].

III. Software

An examination of studies comparing the use of genetic algorithms to solve Sudoku puzzles to other solving techniques, evolutionary and otherwise, reveals that genetic algorithms are not the most effective method for solving such puzzles. In spite of this, there is still value in application of genetic algorithms to solve Sudoku puzzles. As the Sudoku puzzle is a relatively simple problem in terms of the number and complexity of rules or constraints, it is an ideal testbed for algorithm development [11]. Moreover, because Sudoku is a NP-complete constraint optimisation problem, it is a non-trivial problem to explore.

The motivation behind developing the software was to create a tool that could allow for the easy analysis of various genetic algorithm properties. There are two main features of the software, written in C, that permit such easy analysis to occur: the scriptable nature of the software and the log files generated for each solving attempt. The scriptable nature of the software is supported through a large number of options and associated arguments that may be supplied at runtime to specify algorithm configuration for the solving attempt. The options largely relate to the underlying genetic algorithm features described in the following discussion on features of the implemented algorithm. The log files exhaustively log the evolution of the population as the number of iterations increase, allowing for extensive analysis of the various features of the genetic algorithm.

A. Puzzle Representation

The conditions that must be met for a solution to a Sudoku puzzle to be found provide some indication as to appropriate encoding or representation schemes. As constraints 2, 3, and 4 place restrictions on the sets of numbers forming rows, columns and sub-blocks, by developing a representation scheme around rows, columns or sub-blocks, one of the constraints may be fundamentally captured in the scheme. Additionally, by developing an encoding scheme around the division of the Sudoku puzzle grid into rows, columns or sub-blocks, the corresponding constraint can be easily enforced, thus requiring fewer constraints to be optimised against during solving attempts. One consequence of fewer constraints is the potential for a simpler fitness evaluation method.

Two unique representation schemes were identified, one involving mapping the rows of a puzzle grid into a vector and the other involving a similar mapping with the sub-blocks. A mapping using columns is no different from one using rows, as the resultant mapping is essentially structurally invariant under a transpose operation. This is not true when comparing the row representation or encoding with the one based around sub-blocks, hence only row and sub-block encodings were considered for implementation.

B. Crossover Operations

Although the constraints to be satisfied for a solution to a Sudoku puzzle to be found are of benefit in representation scheme and fitness evaluation design, they are restrictive to the design of crossover operations. In both encoding schemes, the puzzle array is mapped to a vector, with implicit internal boundaries existing within this vector separating each row or sub-block. Crossover operations must treat rows or sub-blocks as atomic units, as any attempt to perform crossover internal to rows or sub-blocks would likely lead to a violation of the constraint inherently satisfied by the encoding scheme. For example, if a crossover point within a row is selected when using a row encoding, after the crossover operation it is highly likely that the row will no longer satisfy the constraint that each number from $1, \ldots n$ must appear exactly once.

There are two crossover operations currently implemented in the software: `crossover1()` and `crossover2()`. Conceptually, `crossover1()` involves randomly selecting an arbitrary number of rows or sub-blocks of the first parent and copying them in-situ into the child. The remaining rows or sub-blocks of the child are filled with the corresponding rows or sub-blocks of the second parent. `crossover2()` differs from `crossover1()` in that a random number of consecutive rows or sub-blocks are copied from the first parent, with the remaining rows or sub-blocks provided by the second parent.

C. Mutation Operations

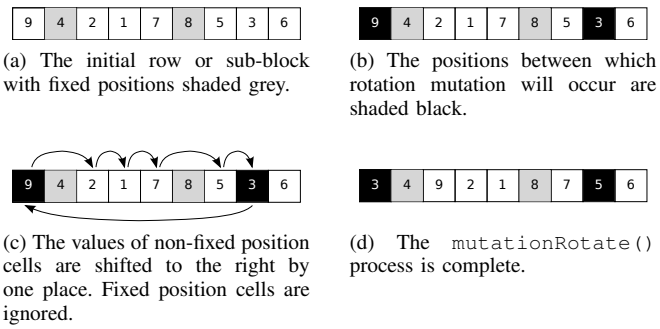
Much like the crossover operations, the mutation operations are limited by the inherent constraints of Sudoku. However, unlike crossover operations which treat each rows or sub-blocks as atomic units, mutation operations are only permitted to occur within single rows or sub-blocks. Mutation methods that involve randomly changing the value of a cell within an individual may not be performed as they would result in a violation of the requirement that each row, for row encoding, or sub-block, for sub-block encoding, contains each number from $1, \ldots n$ exactly once.

Because of the restrictions imposed by the constraints of Sudoku, mutation is limited to the swapping of two or more values within a row or sub-block while ensuring that the values of the fixed positions of the supplied puzzle are not altered. There are four mutation operations currently implemented: `mutationSwap()`, `mutation3Swap()`, `mutation5Seq()` and `mutationRotate()`.

`mutationSwap()` is the most basic of the mutation functions, and simply swaps the value of two, non-fixed cells within a particular row or sub-block. `mutation3Swap()` is very similar to `mutationSwap()`, with the only difference being the swapping of values of three randomly selected, non-fixed cells within a particular row or sub-block. `mutation5Seq()` is capable of performing up to 5 separate swap mutations on a given row or sub-block. Unlike all of the other mutation functions that contain some form of leniency, if a fixed location is selected, this mutation function terminates.

The final mutation function, `mutationRotate()`, performs mutation on a row or sub-block through a limited rotation of the values of cells within the row or sub-block. The process followed by `mutationRotate()` is best demonstrated in Figures 1a-1d.

These mutation functions are utilised by mutation strategies. The mutation strategies use combinations of mutation functions to perform mutation on each individual. The first strategy, `mutation0` performs evaluation for mutation on each individual as a whole and utilises the mutation functions `mutationSwap()`, `mutation3Swap()` and `mutationRotate()`. The second strategy, `mutation1` performs evaluation for mutation on every row or sub-block of an individual and utilises the same three mutation functions as `mutation1`. The third mutation strategy, `mutation2` also performs evaluation for mutation on every row or sub-block of an individual, but utilises the `mutation5Seq()` function to perform mutation.



(a) The initial row or sub-block with fixed positions shaded grey.

(b) The positions between which rotation mutation will occur are shaded black.

(c) The values of non-fixed position cells are shifted to the right by one place. Fixed position cells are ignored.

(d) The `mutationRotate()` process is complete.

Fig. 1. An illustrative example showing the steps to perform mutation in `mutationRotate()`.

D. Fitness Evaluation

It has been noted that the design of an appropriate fitness evaluation scheme is crucial to the overall success of the genetic algorithm in solving the problem of interest [1]. Although design of an appropriate fitness evaluation function is traditionally a difficult process, the constraints that must be met by a solution to a Sudoku puzzle provide an excellent foundation for the design of such a function. Because each number from $1, \ldots n$ appears once and only once in each row, column and sub-block of the solution, if a number is found more than once in any row, column or sub-block, the fitness of the individual should be penalised to represent this deviation from what is required.

One aspect of fitness evaluation is based on this idea and involves examining each row, column and sub-block for such violations. For each number missing from a row, column or sub-block a penalty of -2 is applied to the individual's fitness. The other components of fitness evaluation are based on the observations that, for each row, column and sub-block, the sum of all values should equal:

$$\sum_{x=1}^{n} x \tag{1}$$

and the product of all values should equal:

$$\prod_{x=1}^{n} x \tag{2}$$

If a row, column or sub-block within an individual does not meet the sum requirements, it's fitness is decremented. Similarly, if the product requirement is not met, the fitness of the individual is also decremented.

TABLE I
INPUT OPTIONS AND ASSOCIATED ARGUMENT TYPE AND PERMITTED VALUES.

Name	Option	Arg Values
Sudoku Puzzle File	–in, -i	Path to puzzle
Population Size	–pop, -p	+int
Maximum # Iterations	–gens, -g	+int
Encoding Scheme	–encode, -r	0, 1
Mutation Rate	–mutrate, -m	0-100
Crossover Function	–crossfunc, -x	1, 2
Mutation Strategy	–mutstrat, -s	0, 1, 2
Elitism	–elite, -e	0-Population Size
Common Random Numbers	–crn	+int†
No Mutation of Top Individual	–nomuttop	1-Elitism†
Cataclysm Generation	–catagen, -c	0, +int
Ageing Decrement # Iterations	–agedec, -a	1-Max # Iterations
# Mutations Per Individual	–mutind, -u	0, +int
Unrepeated Cataclysm	–nocatarep	N/A
Log File	–out, -o	Path for log file

† Optional arguments, option can be used without argument.

Every generation, each individual in the population is evaluated using these three criteria to give a fitness value. The optimal fitness value (representing a solution) is 0; every individual that deviates from the solution in some way has a negative fitness value.

E. Selection Process

The selection mechanism of a genetic algorithm is simply a process for selecting parent individuals from the population for the purpose of reproduction. Although the selection mechanism was intended to be one of the key user-configurable features, only a single selection method has currently been implemented. This method is based on the popular roulette wheel selection method, which favours the selection of fit individuals in the population without being overly aggressive in this favouritism.

F. Options & Arguments

The user is able to configure algorithm parameters and operational behaviour by providing options and associated arguments at runtime. Allowing the user a great deal of control over algorithm configuration enhances ease of use and ensures that the software is scriptable for batch experimentation.

The current incarnation of software has 15 options, of which only two are required each time the software is executed. Table I outlines all program options and the arguments associated with these options, while the following describes some of the less obvious options.

1) Elitism: argument specifies the number of elite individuals that exist in the population, with the number of elite individuals remaining constant throughout the solving attempt.

2) Common Random Numbers: allows common random numbers to be used for the solving attempt. The user may provide a specific seed value (optional) to be supplied to the random number generator to allow for greater reproducibility of results.

3) Cataclysm Generation: allows the user to specify when or with what frequency a cataclysmic regeneration of the population should occur. The argument value determines either at which generational iteration of the solving attempt a one-off cataclysmic regeneration of the population will occur or the number of iterations between which cataclysmic events periodically occur. The function of this option is dependent on the whether the Unrepeated Cataclysm option is supplied or not. If a value of 0 is provided as an argument, then cataclysmic events are disabled.

4) Ageing Decrement Number of Iterations: allows the user to specify whether elite individuals are susceptible to ageing, and, if so, how aggressive the ageing process is. The value of the argument stipulates the number of generations any particular individual may exist in the elite group of individuals before a penalty of a decrement in fitness is applied.

IV. RESULTS

The Sudoku puzzles for the experimental component of this work were all standard, 9×9 puzzles of a variety of difficulty levels. Two sets of puzzles were used, one for comparative purposes with other studies, and another for the exploration of algorithm configurations. All puzzles used were sourced from [15].

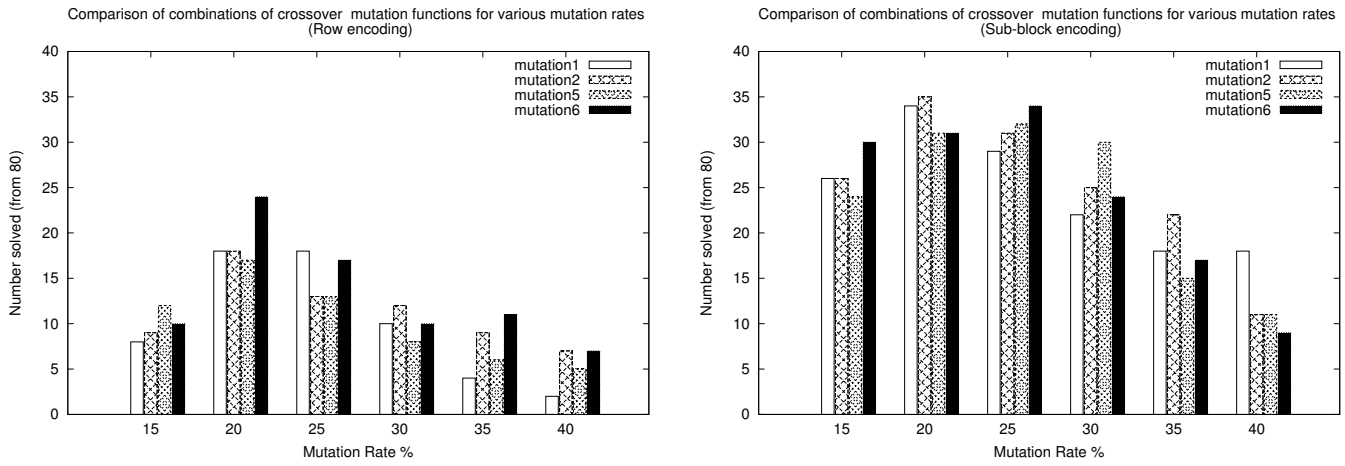
A. Initial Comparison

For the purpose of comparison with other studies, as this work was most influenced by previous research performed by Mantere and Koljonen and detailed in [10], [11], initial testing of the software involved an attempt to replicate the configuration and results presented in [10]. Although a great deal of information regarding algorithm configuration was available, many features such as cataclysmic events and an ageing process were not mentioned, hence were disabled. Additionally, it was not possible to source the same puzzles for experimentation. However, because information regarding difficulty rating, number of givens and nature of symmetry of each puzzle was provided, best effort was made to obtain similar puzzles.

To configure the algorithm to match the description provided in [10], the options and arguments presented in Table II were used. The results from [10] as well as those obtained for this work are presented alongside one another in Table III. For the most part, the results are comparable, with the few discrepancies likely to be a result of puzzle differences rather than algorithm configuration. This initial comparison serves to demonstrate that the implemented algorithm functions as expected and can be used to reproduce results already published in the literature.

B. Analysis of Algorithm Features

An objective of the analysis component of this work was to explore a variety of configurations to examine the effect of algorithm parameter variation. Although a number of algorithm features were analysed, only a selection of the



(a) Row encoding scheme.

(b) Sub-block encoding scheme.

Fig. 2. Results obtained from various combinations of crossover functions and mutation strategies for both the row and sub-block encoding schemes.

TABLE II
PARAMETERS USED TO CONFIGURE ALGORITHM TO COMPARE WITH [10].

Name	Arg Value	Option & Arg
Population Size	100	-p 100
Maximum # Iterations	1×10^5	-g 100000
Encoding Scheme	Sub-block	-r 0
Mutation Rate	12%	-m 12
Crossover Function	1	-x 1
Mutation Strategy	1	-s 1
Elitism Percentage	40%	-e 40
Common Rate Numbers	Off	N/A
No Mutation of Top Individual	True	–nomuttop
Cataclysm Generation	0	-c 0
Ageing Decrement # Generations	0	N/A
# Mutations Per Individual	1	N/A
Repeated Cataclysm	False	–nocatarep
# Attempts/Puzzle Difficulty	100	N/A

TABLE III
FOR EACH LEVEL OF DIFFICULTY THE TOP SET OF RESULTS WERE OBTAINED BY MANTERE AND KOLJONEN [10], WITH THE BOTTOM SET OF RESULTS BELONGING TO THIS WORK. THE NUMBER SOLVED IS FROM 100.

Puzzle	Givens	Solved	Min	Max	Average	Median	Stdev
1 star	33	100	184	23993	2466.6	917	3500.98
		96	84	24176	2120.9	251	4051.05
2 star	30	69	733	56484	11226.8	7034	11834.68
		50	221	90016	20839.4	6599	25455.49
3 star	28	46	678	94792	22346.4	14827	24846.46
		26	414	87209	44062.4	51992	28004.36
4 star	28	26	381	68253	22611.3	22297	22429.12
		21	1493	91926	32713.0	25306	29396.27
5 star	30	23	756	68991	23288.0	17365	22732.25
		43	411	95240	26348.8	14785	27391.67
Easy	36	100	101	6035	768.6	417	942.23
		93	56	78726	2072.4	181	8761.48
Challenging	25	30	1771	89070	25333.3	17755	23058.94
		13	3708	96294	41068.4	40463	24868.06
Difficult	23	4	18999	46814	20534.3	26162	12506.72
		14	3541	99268	49453.1	42038	29777.74
Super Difficult	22	6	3022	47352	14392.0	6722	17053.33
		6	898	96735	33580.3	15873	39293.24

results are presented here. Perhaps the most interesting results relate to the mutation rate, encoding scheme, population size and percentage of elite individuals in the population. In all that follows, results were obtained by using 8 puzzles of varied difficulty, with each puzzle attempted 10 times for each individual parameter configuration.

To observe the effect of mutation rate on the solving rate of the algorithm, a number of brief tests using a variety of algorithm configurations were performed. It was found that a mutation rate somewhere between 15% and 40% lead to the highest solving rate. Subsequent experimentation involved varying the mutation rate between these two bounds, for a small selection of crossover function and mutation strategy combinations, and for both encoding schemes, with an aim to jointly explore crossover function and mutation strategy combinations with encoding schemes.

The results of this experimentation are presented in Figure 2. Figure 2 provides clear visual evidence that the sub-block encoding exhibits a significantly increased solving rate as compared to the row encoding scheme. On average, the sub-block encoding scheme exhibits a solving rate that is a factor of approximately 2.58 better than the row encoding scheme for the combinations of crossover and mutation functions used to obtain results.

Given that the population size and percentage of elite individuals in the population are closely related, to explore the effect that each has on the solving performance of the algorithm, experimentation was performed by joint variation of these parameters. Population sizes of 10, 20, 50, 75 and 100 were used, with the percentage of elite individuals varied in 10% increments between 0% and 100% for each of these population sizes.

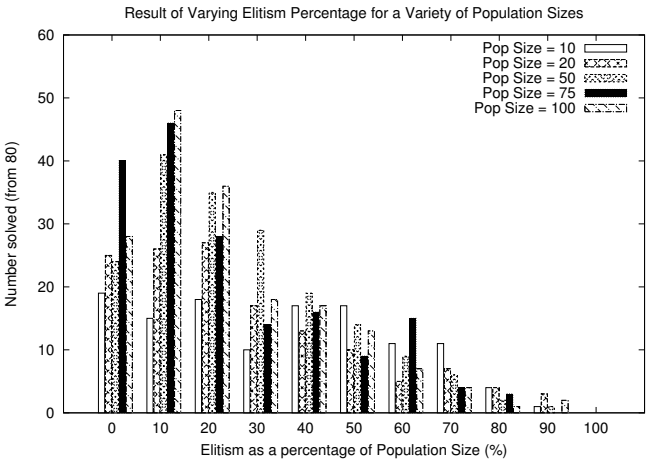


Fig. 3. Results obtained from a concurrent exploration of percentage of elitism with population size.

The results displayed in Figure 3 indicate that the percentage of elite individuals in the population can have substantial effect on performance. The optimal percentage of elite individuals in the population appears to be around 10%. Figure 3 also provides some evidence that population size does not have significant effect on the rate of solving Sudoku puzzles. However, given that there was only one order of magnitude between the minimum population size and the maximum population size, further experimentation is required to draw strong conclusions on the effect of population size on solving rate.

V. Conclusion

This paper outlined the features of a genetic algorithm software tool currently configured to tackle Sudoku puzzles. This tool was developed to allow for an exploration of genetic algorithms in solving constraint optimisation problems such as Sudoku. The software was then verified, by comparing obtained results with results presented in the literature. Finally, various algorithm configurations were explored to allow for an analysis of features of the genetic algorithm. Analysis of the algorithm through experimentation with a number of algorithm configurations provided some insight into which features affect the solving rate in what way, and what algorithm configurations are likely to exhibit higher solving rates than others.

A. Future Work

The software presented in this work is still undergoing sporadic modification and revision and there are a number of features yet to be implemented but planned for inclusion. In terms of the fundamental processes comprising genetic algorithms, only a single selection procedure has been implemented. Introduction of a variety of selection procedures is planned to give the user even greater control over the configuration of the algorithm, allowing for a much deeper analysis to be performed. Similarly the user currently has no choice in the fitness evaluation method as only a single procedure has been implemented. By introducing a greater variety of fitness evaluation procedures, the user will have even finer grained control over algorithm configuration. Future feature development will initially focus on these aspects of the software.

Although the software is fully operational, it still leaves much to be desired in terms of the time required for execution. The performance of the software could be vastly improved by exploiting some of the many opportunities for parallelism. One of the most obvious opportunities for parallelisation is the evaluation of each individuals fitness. As the fitness of each individual is independent of the fitness of all other individuals, parallel fitness evaluation would lead to drastically reduced execution time. Furthermore, given the use of the sequential quicksort algorithm used to sort individuals by fitness, a parallel sorting algorithm could also be used to reduce execution time. Future work on improving performance will involve modifying the software to utilise a graphics processing unit via either CUDA or OpenCL.

References

[1] S. Sivanandam and S. Deepa, *Introduction to Genetic Algorithms*. New York: Springer, 2008.

[2] A. E. Eiben and J. Smith, *Introduction to Evolutionary Computing*. Springer, 2003.

[3] J. Almog, "Evolutionary computing methodologies for constrained parameter, combinatorial optimization: Solving the Sudoku puzzle," in *AFRICON, 2009. AFRICON '09.*, 2009, pp. 1–6.

[4] G. Santos-Garcia and M. Palomino, "Sudoku, gerechte designs, resolutions, affine space, spreads, reguli, and hamming codes," *American Mathematical Monthly*, vol. 115, pp. 383–404, 2008.

[5] J. P. Delahaye, "The science behind Sudoku," *Scientific American*, pp. 80–87, 2006.

[6] G. Santos-Garcia and M. Palomino, "Solving Sudoku puzzles with rewriting rules," *Electronic Notes in Theoretical Computer Science*, vol. 179, pp. 79–93, 2007.

[7] H. Simonis, "Sudoku as a constraint problem," in *Proceedings of 4th International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, 2005, pp. 13–27.

[8] I. Lynce and J. Ouaknine, "Sudoku as a SAT problem," in *9th International Symposium on Artificial Intelligence and Mathematics AIMATH'06*, 2006.

[9] K. Moon and J. Gunther, "Multiple constraint satisfaction by belief propagation," in *2006 IEEE Mountain Workshop on Adaptive and Learning Systems*, 2006.

[10] T. Mantere and J. Koljonen, "Solving and rating Sudoku puzzles with genetic algorithms," in *12th Finnish Artificial Intelligence Conference STeP*, Helsinki University of Technology, Espoo, Finland, 2006, pp. 86–92.

[11] ——, "Solving, rating and generating sudoku puzzles with GA," in *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, 2007, pp. 1382–1389.

[12] J. Jilg and J. Carter, "Sudoku evolution," in *Games Innovations Conference, 2009. ICE-GIC 2009. International IEEE Consumer Electronics Society's*, 2009, pp. 173–185.

[13] A. Moraglio, J. Togelius, and S. Lucas, "Product geometric crossover for the Sudoku puzzle," in *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, 2006, pp. 470–476.

[14] T. Yato and T. Seta, "Complexity and completeness of finding another solution and its application to puzzles," *IEICE Transations of Fundamentals of Electronics, Communication and Computer Sciences*, vol. E86-A, no. 5, pp. 1052–1060, 2003.

[15] V. Hanssen. (2010) Sudoku puzzles. [Online]. Available: http://www.menneske.no/sudoku/eng/