University of Coimbra

Evolutionary Computation

Sudoku
An evolutionary Approach

Author:
Sebastian Rehfeldt
Nº 2016181684
May 23, 2017

Abstract

The Sudoku puzzle became famous over the past years and programmatic approaches became necessary to create, rate and solve new Sudokus. This work proposes an evolutionary algorithm which uses simple swap mutations and cross-overs as well as a restart mechanism to find solutions for given Sudoku puzzles. In contrast to other works, this approach allows the genetic operators to change givens and fixes them later. The results show that the algorithm is able to find solutions for easy problems, but cannot be applied to harder instances.

1 Introduction

Sudoku, or originally known as Number Place, was introduced by an American architect in 1979.[1] It is a combinatorial game where numbers need to be placed in a 9x9 grid with 9 3x3 sub-grids in a way that no duplicated numbers in rows, columns, sub-grids or even diagonals. Starting-off as a puzzle printed in newspapers in England or Japan, Sudoku gained popularity all over the world and even world championships are taking place for this game.

With the increasing popularity of Sudoku, a programmatic way to create, solve and rate Sudokus became necessary. Due to the huge search space for Sudokus and the optimization towards zero conflicts, evolutionary approaches have been implemented to tackle on Sudokus. In this paper, a simple evolutionary algorithm was implemented based on code provided by Ernesto Costa during his Evolutionary Course at the University of Coimbra. The algorithm uses swap mutation and swap cross-over operators. Furthermore, it flushes the population with random immigrants when the algorithm gets stuck in a local optima. Experiments have been conducted to measure the quality of the algorithm and to examine the impact of the difficulty and number of givens of a Sudoku on the algorithm performance. The experiments showed that simple Sudokus can be solved using the evolutionary algorithm.

The remainder of this paper is structured as in the following: Section 2 introduces the Sudoku game in more detail and describes different variants of the game. Additionally, recent approaches to tackle Sudokus are presented in that section. The basic solution implemented in this work, is presented and discussed in section 3. Modifications to the initial algorithm are presented in section 4 and finally evaluated in section 5. The paper is rounded-off by a conclusion in section 6 which summarizes the presented work and discusses the results.

2 Related Work

Given its' popularity as a puzzle and its' programmatic challenge in constraint programming and satisfiability research, Sudoku has been studied by different researches.[1] This section summarizes the Sudoku game first and then presents related approaches to tackle Sudokus programmatically.

2.1 Sudoku

The first Sudoku was published in 1979 and the shipped to Japan were it became popular first. Starting-off from there, Sudoku became a phenomena in the western world around 2005.[1] A Sudoku is given by a 9x9 grid with 9 3x3 sub-grids. The goal of Sudoku is to place the numbers from 1 to 9 inside the cells of the grid so that no duplicates are in columns, rows, sub-grids or

[1] `https://en.wikipedia.org/wiki/Sudoku`

even diagonals. In a beginning, only a few numbers are presented to the puzzler and his task is to fill in the remaining numbers. The simple rules and the combinatorial challenge made this game famous and addictive to people around the world. The difficulty of a Sudoku depends not only on the amount of givens, the numbers visible in the beginning, or their placement. It has been showed that there are 15 to 20 factors which determine the difficulty of a Sudoku.[1] There also exist different variants of Sudoku, but the most popular version is the one presented above where duplicates in diagonals are mostly allowed, which makes the game easier.

2.2 Approaches

The research on Sudokus tries to tackle three different tasks - creating, rating and solving Sudokus. Mantere et al. propose approaches for all of these tasks.[1] They started by creating a Sudoku solver which solves almost every puzzle. The used an integer coded elitist genetic algorithm. A swap cross-over between sub-blocks is applied when no givens would be moved. Furthermore, they implemented a mutation operator which swaps numbers inside a sub-block. Also here the operator is omitted if a given would be moved. They improved their mutation operator by a constraint that a swap is omitted if it would cause a row or column to include one and the same number several times. This limitation increased the computational costs, but enhanced the overall performance. Finally, they figured out that a simple fitness function worked the best and that a restart after 2000 generations without improvement was beneficial. Their solver was later used to create and rate Sudokus. A new Sudoku is created by finding a new 9x9 grid without conflicts and then using the solver to prove that the solution is unique by running it 10 times. The difficulty of a puzzle is given by a function of generations needed for the solver to come up with a solution.

In ealier works, Moraglio et al.[2] solved the Sudoku using an evolutionary algorithm using geometric cross-over. The others claimed a better performance than traditional hill-climbers, but had problems in finding solutions for medium and hard puzzles. Furthermore, they stated that evolutionary algorithms are inefficient as they do not exploit the problem constraints of Sudokus systematically to narrow down the search.[2] In another work, Nicolau and Ryan[3] proposed a different approach. Anyway, their GAuGe (Genetic Algorithms using Grammatical Evolution) was proven to be inefficient to find solutions to Sudokus by Mantere et al.[1]

A solution that preserves building blocks was presented in 2010 by Sato.[4] Sato claimed that the complexity of finding solutions is not only grounded on the huge search space, but also on the big number of local optima. In his work, he proposed a special cross-over operator that averts the destruction of building blocks over creating diversity in the search process. In this approach, a 2D Chromosome was used to represent a puzzle and initial solutions were found by adding random numbers in open fields in a way that every sub-block is a valid permutation. For mutation, a swap operator was used inside sub-blocks and a local search heuristic was added to it to further improve its' performance. Sato also provides example puzzles and experimental results, which are used to benchmark the algorithm of this work. The examples can be found in figure 1.

The evaluation was based on how often solutions were found and when they were found in average. The results were than the ones of Mantere et al. in reference [1].

Especially, the papers of Mantere et al.[1] and Sato[4] gave good ideas for this work. In that way, the 2D representation and the generation of an initial population were copied. Also the ideas for a swap mutation and cross-over were taken for the implementation of a simple evolutionary algorithm. Furthermore, the restart was studied in this work.

No. 1

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

(a) Easy level Sudoku (Givens: 38)

No. 11

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

(b) Easy level Sudoku (Givens: 34)

No. 27

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

(c) Medium level Sudoku (Givens: 30)

No. 29

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

(d) Medium level Sudoku (Givens: 29)

No. 77

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

(e) Difficult level Sudoku (Givens: 28)

No. 106

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

(f) Difficult level Sudoku (Givens: 24)

Fig. 7. The puzzles used to investigate the effectiveness of the genetic operations proposed in this paper.

SD1

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

(a) GA generated Super difficult Sudoku (Givens: 24)

SD2

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

(b)AI Escarcot - Claim to be the most difficult Sudoku (Givens: 23)

SD3

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

(c) Super difficult Sudoku from www.sudoku.com (Givens: 22)

Figure 1: The puzzles used for comparison given by Sato[4]

3 A first solution

During this work, different configurations and features for evolutionary algorithms have been tested to find solutions for Sudoku instances. This section introduces the first version which only uses the mutation operator. The first subsection introduces the problem representation and helper constructs which were used during the evolution. Afterwards, the mutation operator and the fitness function are presented. In a last part, the initial solution is analyzed in an experiment.

3.1 Representations

The puzzles are represented by a 2-dimensional array. The interpretation of a row in this representation can be either a row of the Sudoku or a sub-block of the Sudoku. The first interpretation has the advantage, that it is fast to compute the number of conflicts and that the genotype is the same as the phenotype, but the disadvantage of a slow mutation as the sub-blocks first need to be build. Also a block swap cross-over would be slow for this representation. In contrast, the sub-block interpretation would allow for a fast mutation and block swap cross-over, but the fitness evaluation would be much slower.

As the row approach was more used in previous works and seems to be more intuitive it is used inside this work. Besides the 2-dimensional representation of a genotype, a helper dict was constructed when reading an instance. The dict stores the indices for givens in the corresponding sub-blocks. This becomes necessary when allowing genetic operators which can move givens, e.g. during cross-over. In contrast to previous works as in Mantere[1], genetic operators can move givens during the evolution. But as they need to be in their original position, a function was written to swap them to their original place.

3.2 Algorithm Structure

The algorithm is an integer coded genetic algorithm which uses tournament selection and elitism. The algorithm is designed in a way that some constraints of Sudoku puzzles are kept in each generation. In that way, each block forms a valid permutation after each generation where givens are in their original positions.

The initial population is created by placing a random 1-9 permutation in each sub-block and swapping the givens to the original positions. During the evolution, a mutation operator swaps 2 non-given numbers inside each sub-block based on a fixed probability. This operation ensures that our two constraints are not hurt and that only conflicts in rows, columns and diagonals can occur. As the previous work of Sato[4] did not count the diagonal conflicts, the fitness function of this work only penalizes duplicates in rows and columns. The fitness value for a genotype is therefore the amount of duplicated values in all rows and columns where lower values are superior and 0 is the global optimum.

3.3 Evaluation

To evaluate the quality of the initial solution, to identify some problems and to find a good mutation probability, a first experiment was conducted. Therefore, the algorithm was tested on three different instances of different difficulties. The number of runs was set to 10 and the number of generations to 10,000. This was due to long runtimes and does not allow a statistically valid conclusion of the results, but still gives some evidence on how well it performs and what probability

seems to be good for the mutation operator. The population is 150, the elite percentage is 0.1 and the tournament size is 3. The evaluation is based on the best and average number of conflicts, as well as the best and average generation where the solution was found and how many solutions were found in total. Figures 3 and 2 show the results for the first experiment. Note that the average generations are missing for harder Sudokus as no solutions were found. Also they are missing for the easy Sudokus as many runs did not find a solution which would interfere the value.

	0.02		0.05		0.1		0.2		0.3	
File	Best	Average	Best	Average	Best	Average	Best	Average	Best	Average
easy 38	0	2	0	1,4	0	1,4	0	**0,7**	0	1
medium 30	3	5,8	2	6,9	2	6,5	2	**4,6**	2	5,4
difficult 28	4	5,9	**2**	**4,4**	**2**	4,9	4	5,7	4	5,8

Figure 2: Final conflicts after 10k generations

	0.02		0.05		0.1		0.2		0.3	
File	Best	Solutions	Best	Solutions	Best	Solutions	Best	Solutions	Best	Solutions
easy 38	289	4	**109**	3	190	5	222	**8**	1103	3
medium 30	-	0	-	0	-	0	-	0	-	0
difficult 28	-	0	-	0	-	0	-	0	-	0

Figure 3: Generations for finding the solution, limited to 10k generations and average over 10 runs

The results show that the algorithm was able to find solutions for the easy Sudoku, but failed for harder instances. The most solutions were found by the algorithm with a mutation probability of 0.2. For harder instances, the best results still had two conflicts and in average 4-6 conflicts. It can be seen that especially the probabilities, 0.05, 0.1 and 0.2 performed well. It is assumed that a low mutation rate, converges slower to the optimal solution, but that higher probabilities tend to overshoot the global optimum as too many numbers are changed. As there are 9 sub-blocks, a mutation probability of around 0.1, swaps one block in average, this probability was selected for further experiments. A plot which shows the evolution of the best solution over runs and generations can be seen in figure 4.

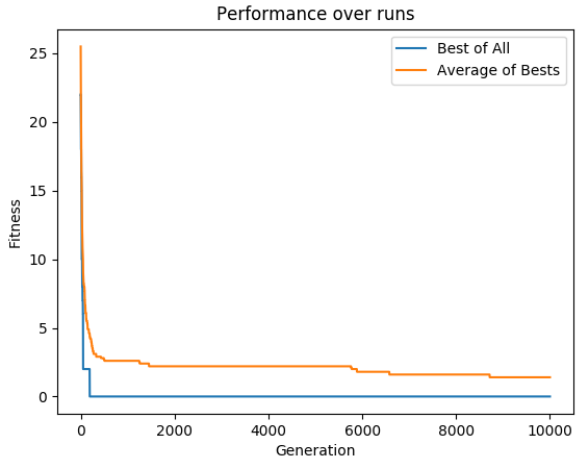

Figure 4: Performance over runs and generations for the easy 38 Sudoku

The plot shows that the fitness value decreases very fast in the beginning and that one run even found the optimal solution. Nevertheless, the average fitness stays around 3 until generation

6000 where it decreases again. From the plot it can be assumed, that the algorithm gets stuck in local optimum and barely is able to improve from there on. This also can be caused by a lack of diversity. The plots for other configurations and files are not listed in the report, but show the same problem.

4 Improved Algorithm

The analysis of the initial solution has shown that the algorithm gets stuck at local optimum which it cannot really escape. This section introduces to new ideas to create more diversity and to escape local optima.

4.1 Cross-Over operators

One common genetic operator to increase the diversity of the population is the cross-over operator. In the context of this work, their are two cross-overs possible: row swap or block swap. The first operator would select two parents and swap two rows of the first parent with two rows of the second parent to create new individuals. This operator has the advantage that it is fast to swap the rows as each row of the genotype represents a row of the Sudoku. In contrast to that, the block swap operator would swap sub-blocks of the genotype to increase the diversity, but needs to first identify the sub-blocks. Both operators have the disadvantage that they can move givens to new positions which is against the constraint of having them fixed. Different works, e.g. Mantere[1] did not allow these operations. This works wants to analyze whether it is beneficial to still swap rows or blocks and afterwards fix the givens. In the block swap approach, the givens are easily fixed by identifying the givens inside a sub-block using the helper dictionary and swapping them to the correct positions. In the row swap approach, this alsp needs to be done, but there is an additional problem as sub-blocks there could also contain duplicated numbers which is not the case in the block swap approach. To fix the duplicates, more computing power is needed to identify the missing and duplicated values and to change them in order to create a valid 1-9 permutation in the sub-blocks. Another way instead of fixing the givens, would be to adjust the fitness function. Due to time constraints, this approach is left open to future work.

4.2 Evaluation of cross-over operators

Experiments for both approaches have been conducted to figure out which configuration works best. Therefore, 10 runs with 1000 generations were tested. The parameters were the same as in the following experiment and use the mutation probability of 0.1 which showed good results in the former experiment. The decrease of generations was done in order to speed up the testing as many tests were run and it would have taken too long to run proper experiments. Tables 5 and 6 show the experimental results. As in the basic approach, the algorithm was unable to find solution for harder instances with one exception.

Due to the probabilistic nature of these algorithms and the low number of runs, it is hard to draw good conclusions from these tables. Both approaches were able to find solutions for the easy Sudoku where the block swap performed slightly better. The final conflicts are quiet close on the easy instance, but with a plus for the block operator on the difficult instance and a plus for the row operator on the medium instance. The generations table shows that one configuration using the block swap even found a solution for the medium instance. Regarding the close results and

	0.05		0.1		0.3		0.5		0.8	
File	Best	Average	Best	Average	Best	Average	Best	Average	Best	Average
easy 38 row	0	3,3	0	2,8	0	3,7	2	3,5	2	5,9
easy 38 block	2	4	0	2,9	0	**2,3**	0	4	0	5,1
medium 30 row	2	**6**	2	6,7	4	7,9	4	9	6	11,2
medium 30 block	**0**	6,8	2	6,8	2	8,6	7	9,7	9	11,4
difficult 28 row	4	7,2	6	7,8	3	6,2	5	8,6	6	9,8
difficult 28 block	4	5,9	**2**	**5,5**	4	6,7	6	8,2	5	8,3

Figure 5: Final conflicts after 1000 generations

	0.05		0.1		0.3		0.5		0.8	
File	Best	Solutions	Best	Solutions	Best	Solutions	Best	Solutions	Best	Solutions
easy 38 row	410	**1**	218	3	196	2	-	0	-	0
easy 38 block	-	0	212	**4**	375	**3**	391	**1**	787	**1**
medium 30 row	-	0	-	0	-	0	-	0	-	0
medium 30 block	739	**1**	-	0	-	0	-	0	-	0
difficult 28 row	-	0	-	0	-	0	-	0	-	0
difficult 28 block	-	0	-	0	-	0	-	0	-	0

Figure 6: Generations for finding the solution, limited to 1000 generations and average over 10 runs

the longer runtime for the row operator, the block operator is considered to be better. Also it has the advantage that no duplicates can appear inside sub-blocks, which would lead to a destruction of the block when fixing the duplicates. But in comparison to the mutation-only approach it is a bit worse regarding the final conflicts and the amount of solutions found. Nevertheless, this approach has its' justification and will be used in a later experiment. For the configuration than, a probability of 0.1 will be used as it found the most solutions after 1000 generations.

After deciding on the cross-over approach and probabilty, one more run was tested with 10,000 generations. This approach found 5 solutions for the easy instance, but no solutions for the harder puzzles. The final amount of conflicts were quiet similar to the once shown in before. The performance over runs and generations is shown in figure 8 below.

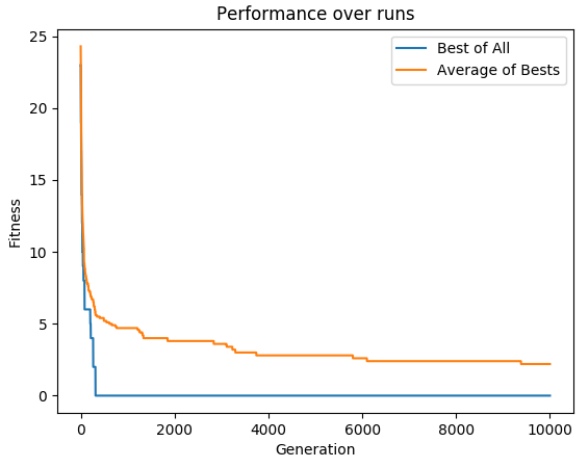

Figure 7: Performance over runs and generations for the easy 38 Sudoku

The plot shows that the average performance did not change between generation 4000 to 6000 and 6000 to 9700 approximately. This means that the algorithm still gets stuck in local optima and needs a lot of "luck" to further improve.

4.3 Adding random immigrants

One idea, which was also applied by Mantere et al.[1] is to re-new the population by replacing old individuals with random new ones. In this work, a certain amount of the worst individuals of the current population are replaced by new random individuals after a certain amount of generations. This approach still keeps the best individuals but allows it to escape local optima by creating more diversity in the population´when the algorithm is stuck. There are two parameters to be determined. First, an experiment on the percentage of new individuals was conducted and later it was tested when the random immigrants should replace old individuals.

Both experiments use the mutation-only approach with a probability of 0.5. 10 runs over 10,000 generations are then used to determine good parameters. In the first test, the amount of generations without improvement, the time when immigrants are added, is set to 500 and percentages of 0.1, 0.3, 0.5, 0.7 and 0.9 are used for the immigrants where higher percentages add more immigrants.

The experimental results show that the best percentage for the immigrants is 0.5 and that this approach finds 6 instead of 5 solutions (without immigrants). Also the average numbers of conflicts are lower than without t (1.1 instead of 1.4 for the easy Sudoku, 4.8 instead of 6.5 for the medium, and slightly worse with 5.4 instead of 4.9 for the hard Sudoku). This parameter has a crucial impact on the t. If the parameter is too low, the t has almost no influence, but if it is too big, the population is too diverse and needs to start from the beginning on.

Also the second parameter, the number of generations has a big impact on the quality. If it is selected too low, the flush with new individuals starts too early and prevents from converging. Otherwise, if it is too big, it might slow down the process and prevents from converging. Mantere et al. tested different numbers of generations without improvement and in their case 2000 achieved the best results. As their approach is a bit different from the one of this work, experiments with 100, 250, 500, 1000 and 2000 generations have been conducted. As before, 10 runs with 10,000 have been tested with a fixed immigrant percentage of 0.5. The results showed that 1000 generations are slightly better than 500 for harder instances and that a solution for the medium hard Sudoku was found. When using the 2000 generations as suggested by Mantere et al., the results got worse, meaning more conflicts in the final solution. The version with 250 generations was able to find more solutions than the ones with 500 or 1000 generations. Anyway, the configuration with 1000 generations seems to be superior when operating on harder instances. Also the restart happens more frequently when using more generations in a final evaluation. The performance of the configuration with an immigrant percentage of 0.5 and 1000 generations wihtout improvement is visualized in figure 8.

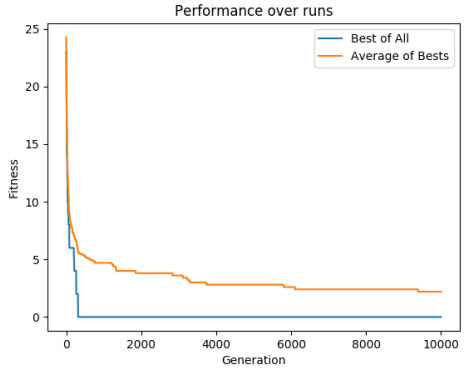

Figure 8: Performance over runs and generations for the easy 38 Sudoku

The plot shows that a solution was found quickly and that the average of bests decreases over the time. If you look carefully, you can see that it decreased after 6000 generations. This is around 2000 generations after the last improvement and shows that the restart, which took place there, helped to escape a local minimum and move towards better solutions.

To help to compare the best solutions so far, different configurations are summarized in the following tables.

	0.1 mutation only		0.2 mutation only		0.1 muta ; 0.1 block cross		0.1 muta ; 0.5/1000 immigrants	
File	Best	Average	Best	Average	Best	Average	Best	Average
easy 38	**0**	1,4	0	**0,7**	**0**	2,2	**0**	2,2
medium 30	2	6,5	2	**4,6**	4	6	**0**	**4,6**
difficult 28	**2**	**4,9**	4	5,7	**4**	6,3	4	**4,9**

Figure 9: Final conflicts of the different configurations

	0.1 mutation only		0.2 mutation only		0.1 muta ; 0.1 block cross		0.1 muta ; 0.5/1000 immigrants	
File	Best	Solutions	Best	Solutions	Best	Solutions	Best	Solutions
easy 38	**190**	5	222	**8**	313	5	4251	2
medium 30	-	0	-	0	-	0	359	**1**
difficult 28	-	0	-	0	-	0	-	0

Figure 10: Generations for finding the solution of the different configurations

Due to the probabilistic nature of the algorithms, it is hard to draw definite conclusions. Nevertheless, you can see some differences. First, the mutation-only approach seems to work better than the approach which also uses cross-over. This might be explained by the fact that the cross-over further creates diversity and might lead to worse individuals during the evolution as it is more likely to destroy good blocks rather than create new ones. Another conclusion is that the immigrant approach works better for harder instances and probably also works better when increasing the total amount of generations. Also it seems that the run with immigrants was "unlucky" as it only found two solutions for the first instance. In former experiments with a lower threshold for adding random immigrants, it found 6 solutions and it can be assumed that these should also be found by an approach which uses more generations. (The solutions for the lower threshold were found early in the experiment)

As the results could be different when running the same experiments again, a final experiment was conducted to figure out the best performance.

5 Final experiments

This section evaluates four different configurations of the algorithm to determine the best configuration. Therefore, a more reliable experiment was conducted. After determining the final parameters, the algorithm was tested on five problem instances of different difficulties and numbers of givens.

5.1 Final parameter tests

Final tests were run on the "AI Escarcot" which is claimed to be the most difficult Sudoku. 20 runs have been tested with 100,000 generations for each configuration which consist of mutation-

only approaches with mutation probabilities of 0.1 and 0.2 as well with a mix of cross-over and mutation with a cross-over probability of 0.1 and the mutation probabilities as mentioned before. All algorithms use the immigrant approach with a percentage of 0.5 and 1000 generations as this showed improvements on the quality of the algorithm. The different mutation and cross-over probabilities are used because the results were not really clear from the former experiments.

It can be assumed that algorithms which solve hard Sudokus, could also solve easier Sudokus and that algorithms with less conflicts in the end are superior to ones with high numbers of conflicts. The results of the experiment can be seen below in figure 11.

	0.1 muta ; 0 cross		**0.2 muta ; 0 cross**		0.1 muta ; 0.1 cross		0.2 muta ; 0.1 cross	
File	Best	Average	Best	Average	Best	Average	Best	Average
Ai Escarcot	**2**	5,2	**2**	**4,95**	4	5,6	4	5,9

Figure 11: Final conflicts of the different configurations

From the figure, it can be seen that no solution have been found, but that the mutation-only approach was better than the approach using cross-over and mutation. This might be explained by the fact that the cross-over furthermore changes the individuals. This might be good in the beginning, but hinders the algorithm to find the optimal solution when it is already close to it. A possible solution would be to implement a different and more domain specific genetic operators, which are more likely to improve the population by taking into consideration the conflicts in rows and columns. Also dynamic probabilities might help in the end. Another good idea comes from Sato who implement a more local search to the mutation operator by creating more than just one new individual. This increases the probability of finding a better individual when the current population is already of good quality. By running 100 trials with 100,000 generations, Sato's approach found 83 solution to the AI Escarcot which was clearly better than the approach of Mantere et al. presented in [1] which found 5 solutions in the same experiment. It might be interesting to know if the presented approach in this work is superior than the one of Mantere et al. Therefore, more trials would need to be run which is unfortunately not possible due to time constraints.

To evaluate the algorithm quality on easier instances, it was run on five other Sudokus with the best configuration highlighted in figure 11.

5.2 Final experiments

Due to the long runtimes and the big number of experiments performed on one personal computer, the number of runs was again limited to 20 and the number of generations to 100,000. This barely allows a statistical analysis, but is a compromise of time and significance of results. Each experiment with 100,000 generations took around one day. Even when running multiple experiments in parallel, it took around three days to finish the experiments of this section. The limited time is also the reason for selecting only some of the Sudokus presented in figure 1 for a final test. This test uses the easy Sudokus with 34 and 38 givens to test whether more givens make it easier for the algorithm to solve a puzzle. Furthermore, the medium 29 and difficult 24 have been tested. These files should give some evidence on how the difficulty impacts the algorithm capability of finding results. The results are given in figure 12. The results of Sato, which only consist of the number of solutions found and the average generation when the solution was found, are included in these figures.

	Own results		Sato muta		Sato muta+cross		Sato muta+cross+LS	
File	Solutions	Average	Solutions	Average	Solutions	Average	Solutions	Average
easy 38	10	56459	100	223	100	105	100	62
easy 34	5	83518	96	6627	100	247	100	137
medium 29	1	-	66	42141	100	6609	100	3193
difficult 24	0	-	9	94314	74	56428	96	26825

Figure 12: Generations for finding the solution for the different files

The results show that the approach presented in this work was able to find half of all solutions of the easy 38 puzzle, but that it took much longer to come up with the solutions. For the second easy instance with less givens only 25% of the puzzles were solved in contrast to Sato where almost every approach found the solution. The mutation-only approach of Sato also struggled with harder instances but still found many more solutions, ¿60% more on the medium instance. The results of Sato also showed that the cross-over improved the quality which is in contrast to the results of this work. This can be explained by a different cross-over approach implemented by Sato.

As Sato did not include the final conflicts, the results for this experiment are given in figure 13 to get an impression on how close the algorithm came to the solutions.

File	Best	Average
easy 38	0	1,25
easy 34	0	4,3
medium 29	0	3,8
difficult 24	2	5,15

Figure 13: Final conflicts for the different files

The results show that the algorithm for the easy 38 instance came very close to the optimal solution, even in cases where the optimal solution was not found. For the easy 38, the algorithms which did not found a solution were more far away from the optimal solution by in average 4.3 conflicts. On the other files, the algorithm ended with averages of 3.8 and 5.15 which shows that the algorithm was probably close to more solutions for the medium instance. In comparison to the work of Sato, the presented results of this work are clearly worse and more future work would be needed. Also it is quiet surprising that the mutation-only approach of Sato worked surprisingly good in comparison to the own implementation which should be similar.

6 Conclusion

This work was done as a project in a Evolutionary Computation course at the University of Coimbra. It aimed for finding an evolutionary solution for Sudoku puzzles. In this paper, a mutation-only approach was implemented and discussed. Furthermore, variants including random immigrants and two different cross-over operators have been implemented and bench-marked against the first implementation. Due to the many experiments for variants and parameters selection, the experiments consist of lower numbers of runs and numbers of generations which would not have been done in a more sophisticated project with more time and computing resources. The experiments showed that the cross-over did not improve the solution and jsut increased the runtime. In contrast to that, the restart seemed to help with harder instances and to escape the many

local optima. The code, all experimental results and some instructions for running the experiments can be found on Github.[2]

Another (obvious) conclusion is that Sudoku with harder difficulty ratings and less givens are harder to solve. This can be seen in figure 12 where the easy 38 puzzle performed much better than the easy 34. Also there could be seen that puzzles with a medium and hard rating could barely or not be solved by the presented algorithm. The experiments also showed that it takes a very long time to come up with a solution. This can be seen on the high number of generations needed for finding a solution and the pure runtime of up to days for many runs.

In comparison to other works, the presented approach is worse and more work would be needed to improve the quality of the algorithm. Even when Sato found almost every time solutions to the puzzles, it seems that evolutionary algorithms are not perfectly suited to the task of solving Sudokus.

6.1 Future Work

As already mentioned, the cross-over operators used in this approach are not really helpful for finding solutions and more investigations are needed there to find a better operator. Also it would be interesting if it is beneficial to let the operators move givens and to penalize them in a fitness function. This would allow for faster executions, but probably lead to more invalid solutions.

Future work could also be done regarding the mutation operator. Here domain specific knowledge, as the conflicts in rows and columns, could be used to come up with better solutions after mutating.

Finally, the initialization of the population could be changed in a way that less conflicts exist in the first population and that more numbers are already in their right position.

Due to the fact of solving a two-person project alone, these ideas will stay open and it is concluded that EA's can solve Sudoku puzzles even if they are probably not the best choice for it.

References

[1] Timo Mantere and Janne Koljonen. Solving, rating and generating sudoku puzzles with ga. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 1382–1389. IEEE, 2007.

[2] Alberto Moraglio, Julian Togelius, and Simon Lucas. Product geometric crossover for the sudoku puzzle. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 470–476. IEEE, 2006.

[3] Miguel Nicolau and Conor Ryan. Genetic operators and sequencing in the gauge system. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 1561–1568. IEEE, 2006.

[4] Yuji Sato and Hazuki Inoue. Solving sudoku with genetic operations that preserve building blocks. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 23–29. IEEE, 2010.

[2] `https://github.com/SebastianRehfeldt/evolutionaryComputation`