SUDOKU SOLVER

Pyuzzle

Computational Intelligence for Optimization

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

Svitlana Vasylyeva | m20200617
Tomás de Sá | m20200630
Valentina Rusinova | m20200591

NOVA IMS | Masters in Data Science and Advanced Analytics

1 Introduction

In this project, we studied and implemented an artificial intelligence technique (Genetic Algorithms) for Sudoku solutions generation. Genetic Algorithm (GA) is a search-based optimization technique method that simulates the Darwin's theory of evolution that occurs in nature (Genetics and Nature Selection). It is frequently used to find optimal or near-optimal solutions to difficult problems which otherwise would take a lifetime to solve. There were implemented different GA operators (selection, mutation and crossover), with 3 variations for each one to solve Sudoku puzzle, but let's first check how Sudoku works.

2 The rules of Sudoku

General Sudoku puzzles consist of a 9x9 matrix of square cells, some of which already contain a number from 1 to 9. We will work with 9x9 matrices only. The arrangement of given numbers when the puzzle is presented is called the starting point. There are multiple difficulty degrees that can vary based on the given numbers quantity and their positions. An example of Sudoku puzzle in its starting point is shown and its final solution in figure 2.1.

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

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

Fig. 2.1. *Sudoku puzzle starting point and solution*

Each Sudoku puzzle has one and only one possible solution.

A Sudoku puzzle can be divided by rows, columns and 3x3 blocks (shown in the figures above by the different colour squares) and it is completed by filling in all of the empty cells with numbers from 1 to 9 without repeating a single one in each row, column and block 3x3.

3 Genetic Algorithm operators

3.1 Selection operators

During each successive generation, a portion of the existing population was selected to breed a new generation. Individual solutions were selected through a fitness-based process, where fitter solutions (measured by a fitness function) are more likely to be selected. There were used three types of selection to find a constantly improving population:

- Fitness Proportion Selection,
- Tournament Selection,
- Rank Selection.

3.2 Fitness Proportion Selection

Each individual has a proportional probability to become a parent based on their fitness. Classical Roulette Wheel Selection resembles a spin of a roulette wheel in casino where a proportion of the wheel is assigned to each of the possible selections based on individual's fitness value. Figure 3.1 shows an example of selection of an individual.

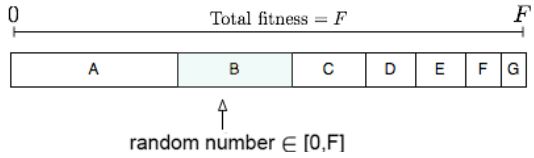


Fig. 3.1. *Selection of an individual (maximization)*

Our code implementation of FPS based on "Charles library" works for both minimization and maximization where for maximization each 'sector of a wheel' proportional to the fitness and for minimization (our Sudoku task) is inversely proportional to the fitness of individual because we are solving minimization problem (code implementation was done for minimization and maximization problems where it was applicable)

3.3 Tournament

N individuals were randomly selected from the population to create a sample where they compete to find the best individual to become a parent. Then a process was repeated to find next parent. As we work with minimization, it was always selected the individual with the smallest fitness from the sample.

3.4 Rank

All the individuals from the population were selected and ranked according to their fitness value, as in the example in table 3.1. The selection of the parents depends on the rank of each individual and not the fitness value. Again, as our goal was to solve minimization problem, individuals with smaller fitness were higher ranked and selected with higher probability than the lower-ranked ones.

3.5 Crossover operators

Crossover operators were implemented for the 81-symbol string. The crossover point could only occur at the subgroup (group of every 9 digits in our 81-symbol string) border, the crossover point cannot be chosen within a subgroup. It should also be noted that the fixed numbers (given numbers) cannot change their positions. We used one of the select operators to choose two parents to perform the crossover. It was three different crossover operator types implemented: one-point crossover, two-point crossover and uniform crossover.

3.6 One-point crossover

A random point was selected to perform the crossover between the two parents, obtaining two new children that result from the tail swap of their parents.

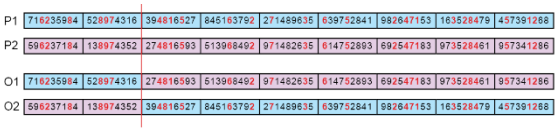


Fig. 3.2. *One-points Crossover*

3.7 Two-point crossover

Two crossing points are selected randomly (on the borders of subgroups) in the parents 81-symbol strings between sub-groups. The parent segments between the two points are exchanged, generating two new children.

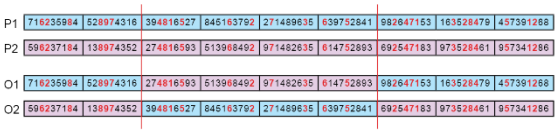


Fig. 3.3. *Two-points Crossover*

3.8 Uniform crossover

Each segment of the parent is selected with the same probability (we created a 'binary mask' to choose segments from different parents as it shown in Figure 3.4). Thus, we are creating two new children who inherit different segments from parents.

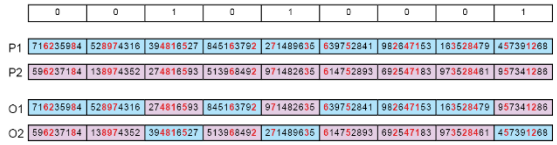


Fig. 3.4. *Uniform crossover*

3.9 Mutation operators

Due to the existence of fixed (given) numbers, which position cannot be changed, mutation operators have been implemented for each subgroup, thus avoiding duplication of values within each subgroup. Two subgroups were randomly selected, where one of the following mutations was implemented: exchange mutation, inversion mutation or shuffling mutation.

3.10 Swap mutations

In this mutation, two values that haven't fixed positions are selected at random and their positions are swapped.

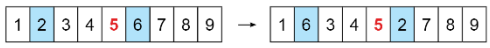


Fig. 3.5. *Swap mutation (where "5"is fixed)*

3.11 Inversion mutations

For an easier implementation of inversion mutation, due to the existence of numbers that cannot be mutated, we created a list that appends only the values that don't have fixed positions. In this list, we select two positions randomly, all the values between the selected positions are inverted. After obtaining a mutated list, the values are returned to the subgroup, to the empty spaces.

Fig. 3.6. *Inversion mutation*

3.12 Scramble mutations

In the same way, as for the inversion mutation, we create a list with the values that can be mutated. Two positions are randomly selected, the values between those positions are randomly shuffled. Then, the values from the modified list are re-appended to the empty spaces in the subgroup.

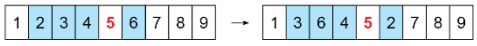


Fig. 3.7. *Scramble mutation*

4 Fitness function

The fitness function requires that each row, column and block must contain numbers from 1 to 9 without repetitions. The function calculates a sum of missing digits in each row x_i set, column x_j set and block x_k set as sum of $|A - x_i|$, $|A-x_j|$, $|A-x_k|$, where A is a set of numbers from 1 to 9:{1,2,3,4,5,6,7,8,9}. In the

optimal situation when all digits appear in the row, column and block sets, our fitness function value becomes zero. Otherwise, a penalty value $|A-x_i|$ added to the fitness function. To solve the sudoku we minimize the fitness function, and when the fitness function value is zero, the sudoku is solved.

5 GA implementation

The individual is an array with 81 integers in range (1-9), interpreted as a vector of 9 rows. Some integers are given and these values cannot be changed. The empty spaces were filled with values randomly, and a row with 9 integers cannot have duplicated values in range (1-9). An initial population of 1000 individuals were created randomly, the fitness was calculated by the fitness function, if some of individual got a fitness 0, the quiz was solved, if not, two parents were picked, with one of the selection operators (FPS, tournament or rank), to create two new children by crossover and mutation. New individuals were created until perform the population size. The individual with the best fitness (lower, as it is a minimization problem) was considered an elite individual, and we save this individual. This elite individual is re-added to the population and we discard the worst individual. This process runs until found the solution, or the stop criterion was achieved, all defined generations were executed. When the solution is found the solved quiz is returned.
To avoid premature convergence, we add 1 to the elite fitness if the fitness of the best new created individual is the same, it makes to select other individuals and vary a population.

6 Results

We start by testing an easy sudoku puzzle that has 38 givens numbers. We run each combination of selection, crossover and mutation (27 combinations) 25 times with a population size of 1000 and with 250 generations. The mutation rate was 0.3 and the crossover rate was 0.7.
Each selection was running 225 times. As we can see from 675 runs, the sudoku puzzle was solved 228 times, and in average the quiz was solved by generation 68. The Rank and FPS operators are able to solve the quiz more times. The combination with the Scramble mutation failed to solve the quiz many times.

	Fps	Tournament	Rank	Total
two_swap	22	13	23	58
single_swap	17	9	25	51
uniform_swap	17	10	23	50
single_inversion	8	3	13	24
uniform_inversion	5	4	9	18
two_inversion	5	0	8	13
two_scramble	2	1	3	6
single_scramble	1	0	4	5
uniform_scramble	1	0	2	3
Total	78	40	110	228

The crossover operators, each of them was executed 225 times. These operators were able to solve the quizz almost the same number of times. And the combination with scramble mutation was the worst.

	Single	Two points	Uniform	Total
rank_swap	25	23	23	62
fps_swap	17	22	17	56
tournament_swap	9	13	10	32
rank_inversion	13	8	9	22
fps_inversion	8	5	5	18
tournament_inversion	3	0	4	7
rank_scramble	4	3	2	5
fps_scramble	1	2	1	4
tournament_scramble	0	1	0	1
Total	80	77	71	228

Likewise, each mutation operators has been running 225 times. The swap mutation was able to solve more than others operators. To run the next quiz, we discarded the scramble and inversion mutations, like the swap mutation showed better performance in solving the puzzle.

	Swap	Inversion	Scramble	Total
rank_single	25	13	4	42
rank_two	23	8	3	34
rank_uniform	23	9	2	34
fps_two	22	5	2	29
fps_single	17	8	1	26
fps_uniform	17	5	1	23
tournament_single	9	3	0	20
tournament_two	13	0	1	14
tournament_uniform	10	4	0	14
Total	159	55	14	228

Next, we run a medium sudoku puzzle with 33 givens numbers, with the same parameters as for the easy quiz.

In this scenario, we have 9 possible combinations, each combination was run 25 times. In total we have 225 runs, each operator was running 75 times, and we can see that without the scramble mutation the percentage in solved quiz increases, as it solves 154 times, and in average the quiz was solved by generation 83. The tournament selection operator solve the quiz less times.

	Fps	Tournament	Rank	Total
single_swap	22	11	22	55
uniform_swap	18	13	23	54
two_swap	17	7	21	45
Total	57	31	66	154

Again, crossover operators have the same performance in the ability to solve the quiz.

	Single	Two points	Uniform	Total
rank_swap	22	21	23	62
fps_swap	22	17	18	57
tournament_swap	11	7	13	31
Total	55	45	54	154

We can see that the combinations with tourmanent operator are the worst, so, we will discard them to solve the next quiz.

	Swap
rank_uniform	23
fps_single	22
rank_single	22
rank_two	21
fps_uniform	18
fps_two	17
tournament_uniform	13
tournament_single	11
tournament_two	7
Total	154

In order to make the last decision, we run a hard sudoku puzzle with 30 givens numbers with the same parameters as for previous ones. Here, we have 6 different combinations, and each of them was running 25 times, performing all 150 running times. The hard quiz only was solved 35 times, each operator was running 50 times and in average with 112 generations, so, for better performance it will be better run for more generations. The FPS selection operator perform better than Rank, and all of them did not have good result with two points crossovers.

	Fps	Rank	Total
single_swap	7	7	14
uniform_swap	10	3	13
two_swap	3	5	8
Total	20	15	35

The two-points crossover had the worst results and the single and uniform were able to solve with same capability.

	Single	Two points	Uniform	Total
fps_swap	7	3	10	20
rank_swap	7	5	3	13
Total	14	8	13	35

We can see that the best combinations, was FPS with uniform, FPS with single and Rank with single. We will select these three combinations to make comparations.

	Swap
fps_uniform	10
fps_single	7
rank_single	7
rank_two	5
fps_two	3
rank_uniform	3
Total	35

7 Difficulty levels comparison

In order to compare the results, we run all combination 100 time, with a population size of 1000 and with 350 generations, increased by previous ones. The mutation rate was 0.3 and the crossover rate was 0.7.

We can see that the easy sudoku was solved better with the combination of FPS selection operator, single point crossover with swap mutation. This combination solve the sudoku faster and with less generations.

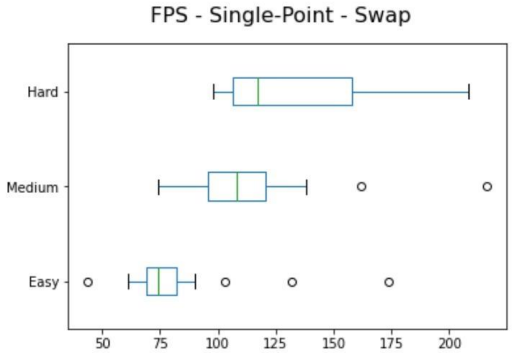
As well, the medium sudoku level was solved better with the same combination, but faster, on average with 35 seconds, and 62 generations (average). Compared with the easy sudoku, the medium sudoku it took more time to be solved with the combination FPS_uniform_swap.

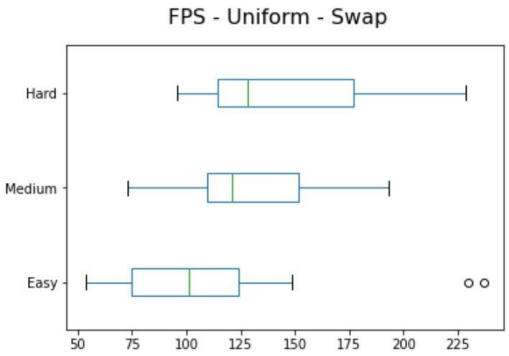
In the difficult sudoku level, the quiz has been solved a few times, getting more solutions with the combination FPS_uniform_swap, but the faster combination that solve the quiz is Rank_single_swap.

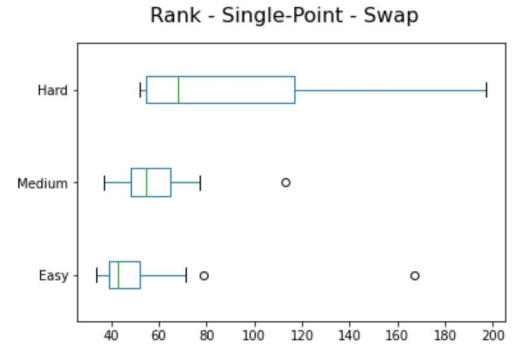
		Easy			Medium			Difficult		
Combination	**Times**	Solved	Gens	Seconds	Solved	Gens	Seconds	Solved	Gens	Seconds
fps_ single_ swap	100	79	118	102	74	119	116	23	171	166
fps_ uniform_ swap	100	85	100	89	87	134	135	29	162	165
rank_ single_ swap	100	92	63	63	93	62	35	26	97	55

Table 7.1. *Dificulty levels comparison*

However, the existence of outliers, as we ca see below, especially in the easy sudoku, causes that average of needed generations to solve the quiz is affected. From the boxplots, we can conclude that it is only needed approximately 40 generations to solve the quiz, with the rank_single_swap combination.

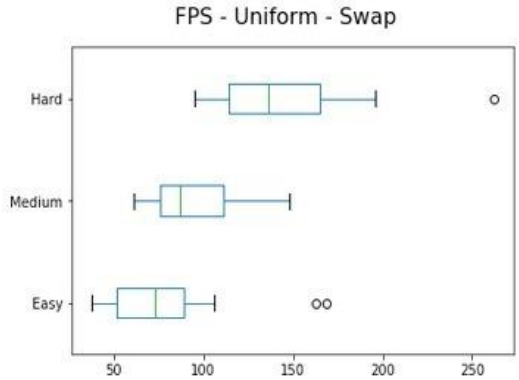


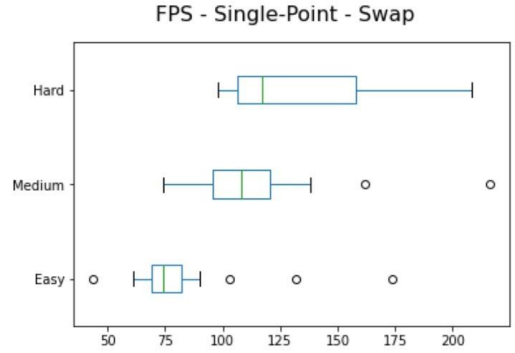


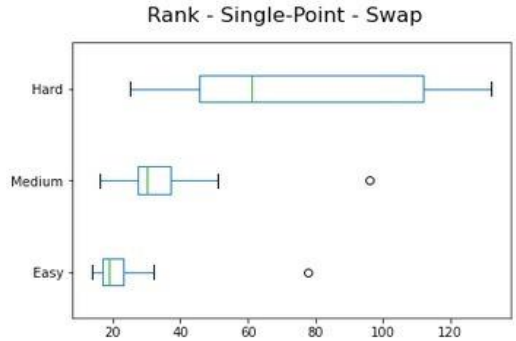


The solving time result is affected by the outliers as could be seen from a boxplot.

We can see that we need approximately 20 seconds to solve the easy sudoku, not 63 seconds as table 7.1 shows for the rank_single_swap combination.







We can conclude that for easy and medium sudoku we have better performance with a Rank selection, single point crossover, and swap mutation. On the other hand the difficult quiz has better performance with a combination FPS selection, uniform point crossover, and swap mutation, but the Rank selection operator works faster than the FPS selection operator.