GENETIC ALGORITHMS PROJECT

In this project, you will solve Sudoku puzzles by using a genetic algorithm. Sudoku is a puzzle of 81 squares where each row and each column of the puzzle contains the digits 1 to 9, as does every smaller 3 x 3 grid. Some squares are already filled, as seen here:

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

Therefore, the task is to find suitable numbers for the blank squares. More examples of Sudoku puzzles can be found here:
https://sudoku.com/

You may implement a genetic algorithm yourself. Alternatively, you may use the implementation of the genetic algorithms from the JGAP library:
http://jgap.sourceforge.net/

If you choose to use JGAP, please download the jgap_3.6.3_full.zip file from the virtual campus, which contains the version of the JGAP library that you must use to carry out the optimization. You may start by having a look at the knapsack example within the jgap_3.6.3_full.zip file. You must add the JGAP library (*jgap.jar*) to the Eclipse project in order to use it in your code.

No matter whether you use JGAP or your own implementation of genetic algorithms, you have to design your own fitness function to solve Sudoku puzzles.

Also, you must use the Sudoku generator and solver from the QQWing library:
https://qqwing.com/

Please download the qqwing-1.3.4.jar file from the virtual campus, which contains the version of the QQWing library that you must use to generate a Sudoku puzzle with its solution, and print Sudoku matrices.

You must develop a Java console application that does the following:

1. Generate a 9 x 9 matrix that is a valid Sudoku puzzle. This must be done with the help of the QQWing library, version 1.3.4, which you should

download from the virtual campus. Then the puzzle and its solution must be printed out.

2. Run a genetic algorithm that tries to solve the generated Sudoku puzzle. Each time that a new generation is obtained, the application must print the mean fitness of the individuals, and the fitness of the best individual. The genetic algorithm must be based on the JGAP library, version 3.6.3.
3. Once the genetic algorithm has finished, a message must be printed indicating whether the solution to the puzzle was found. After that, the best individual of the last generation must be printed out.

A plain text file named output.txt must be generated which contains the full output of the program.

Optional task:

You can show the progression of the genetic algorithm graphically, by showing how the mean fitness and the fitness of the best individual progress as new generations are obtained. You must also analyze how the performance of the genetic algorithm changes as the genetic algorithm parameters are changed.

Tentative schedule:

Laboratory session 1: Develop the code to generate, manage, and print out a Sudoku puzzle. Also, design the fitness function for the Sudoku puzzle.

Laboratory session 2: Develop the genetic algorithm to solve the Sudoku puzzle.

Laboratory session 3: present the final result to the class with the help of a slide show and do a practical demonstration of the software (maximum 10 minutes overall). All the materials (slide show, source code and output.txt file) must be submitted to the associated virtual campus task.

HINTS:

The chromosome must be a class that implements the IChromosome interface of the JGAP library. It is advisable that the chromosome is composed by an integer gene for each empty cell of the Sudoku puzzle. Let us consider the previous example:

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

Then each empty cell of the puzzle must be identified as a distinct integer value to be determined:

1	9	A1	A2	2	A3	5	A4	8
B1	6	7	B2	B3	B4	B5	4	B6
C1	C2	4	6	8	3	C3	9	C4
3	D1	D2	7	D3	D4	2	D5	9
E1	E2	E3	1	E4	E5	6	E6	5
F1	F2	F3	5	9	8	F4	F5	4
4	G1	5	8	G2	G3	9	G4	6
2	H1	6	H2	4	H3	H4	5	1
9	I1	1	I2	I3	6	I4	7	I5

Therefore, the structure of the chromosome for the example puzzle is as follows (38 integer genes):

There are 27 constraints to be fulfilled: 9 row constraints, 9 column constraints, and 9 block constraint. It is advisable that you ensure that all the individuals satisfy at least one of these three kinds of constraints. Therefore, there are three ways to go:

1. Enforce the row constraints.
2. Enforce the column constraints.
3. Enforce the block constraints.

From now on, we will see how to proceed with the first approach (enforce the row constraints).

In order to generate the initial population (say about 20 individuals), first you must generate a first individual which satisfies all 9 row constraints. To do so, for each row, you must populate the genes associated to the row with the missing digits of that row. In our example, for the first row, genes A1, A2, A3 and A4 should be assigned to digits 3, 4, 6 and 7, which are the missing digits for that row. After the first individual is generated, you must generate the initial population by randomly permutating the digits of each row. In our example, for the first row, you must do a random permutation of the values of the genes A1, A2, A3 and A4. This way, all the individuals of the initial population fulfill the 9 row constraints.

The crossover operator must be implemented as a subclass of the CrossoverOperator class of the JGAP library. It must be ensured that if the two parent individuals fulfill all the row constraints, then any offspring individual will also fulfill all the row constraints. This can be done by restricting the crossover points in the chromosome to the boundaries between adjacent rows. For example, if we have these two parents:

Then the only allowed crossover points must be: A4-B1, B6-C1, C4-D1, D5-E1, E6-G1, G1-H1, and H4-I1. If the C4-D1 crossover point is chosen, then the two possible offspring are:

The mutation operator must be a subclass of the MutationOperator class of the JGAP library. It must be ensured that for any individual that fulfills all the row constraints, the resulting mutated individual also fulfills all the row constraints. This can be accomplished by defining the mutation as the swap of two values of genes which belong to the same row. For example, if we have this individual:

Then we may mutate by swapping the values of B2 and B5, for example, because they both belong to the second row. But we cannot swap the values of D4 and E3 because they do not belong to the same row; D4 belongs to the fourth row, while E3 belongs to the fifth row.

The fitness function may be defined as the sum of the counts of unique values contained in each column and block. Rows are not considered since each row contains exactly 9 unique values, thanks to the design of the initial population, the crossover and the mutation operators. This means that the fitness function is the sum of 18 counts (9 column counts and 9 block counts), so that each count is a natural number between 1 and 9. Therefore, the fitness function always returns an integer between 18*1=18 and 18*9=162. An individual is a solution to the puzzle if and only if its fitness is 162.