PROJECT REPORT

A Genetic Algorithm implementation for Sudoku Puzzle solving

CURRICULAR UNIT:	**COMPUTATIONAL INTELLIGENCE FOR OPTIMIZATION**
GROUP MEMBERS:	**DANIEL CORREIA (M20200665),**
	JOANA RAFAEL (M20200588),
	RICARDO SANTOS (M20200620)

DATE: 28.05.2021

THIS DOCUMENT IS ACCOMPANIED BY THE FOLLOWING CODE:

1. SUDOKU_HARDCODED.PY
2. CHARLES_LIBRARY
    a. CHARLES_SUDOKU.PY
    b. RUN_CHARLES.PY
    c. GET_SUDOKU.PY
    d. FITNESS_FUNCTIONS.PY
    e. SELECTION.PY
    f. CROSSOVER.PY
    g. MUTATION.PY
3. CHARLES_ANALYSIS.IPYNB

I. Introduction

Sudoku is a logic-based placement puzzle very familiar to all of us. Every row, column and 3x3 grid needs to be filled with a number from 1 to 9, while containing only 1 instance of each of the numbers per row, column and grid. Thus, only one solution exists for each puzzle, which is initiated with a few numbers. Solving easy sudokus is quite straightforward: a simple Python program with the hardcoded rules for solving sudokus does the job. However, for medium to hard sudokus there's a need to make guesses and continue evolving the sudoku until errors are found. At this point, the hardcoded algorithm should backtrack and make a new guess. Evidently, for hard sudokus, this becomes computationally burdensome quickly if a lot of guesses are being made in chain.

Genetic algorithms are a good alternative to surpass the computational burden of hardcoded algorithms. Thus, in this work, we develop a genetic algorithm with the intent of solving sudoku puzzles. We aimed at developing an algorithm that optimizes the search problem that finding correct solutions to sudoku puzzles poses. Here, we implement multiple genetic operators, fitness functions (both maximization and minimization) and population selection methods and design a strategy to benchmark the different configurations and deploy a final algorithm.

II. Methods

Vector Encoding

Every individual was encoded as a vector of 9 rows (lists) of a variable number of integers each (9 – amount of numbers initially given by the sudoku grid at the given row).

Initialization

We designed out Genetic algorithm to work under a few hard-coded constraints:

- We initialize the sudoku grid by row, allowing only one number through 1 to 9 to be placed in each row;
- To accommodate for this constraint, we used genetic operators that do not change the cardinality of the rows; besides the exception of the single point crossover (further discussed in the Results section), all genetic operators operate at the row level;

The reason for these hard constraints is obvious. In parallel with this genetic algorithm, we have also developed a hard-coded model to solve sudokus, using brute-force. Our hard-coded algorithm is not capable of solving some medium difficulty sudokus, nor most hard ones. However, the hard-coded algorithm is extremely efficient (timewise and computationally) in solving easy to medium-easy sudoku grids. Thus, we thought that hard-coding the initialization of our sudoku grids in the genetic algorithm would diminish the horizon of possible permutations to be made inside the whole grid, making it more efficient in solving any difficulty-level sudoku.

Fitness Functions

We designed 4 different fitness functions to work along our genetic algorithm. Please note:

1. Minimize Errors (min_errors): assesses an individual's fitness by counting the number of errors identified in each row, column and 3x3 grids;
2. Minimize Factorial (min_factorial): computes product of elements in rows, columns and 3x3 grids. The fitness of individuals is computed as the sum of the difference in absolute value between the expected result (9 factorial) and the real value;
3. Minimize Sum (min_45sum): calculates the fitness of an individual by summing the absolute value of the sum of all elements of all rows, columns and 3x3 grids subtracted by 45 each (the total sum value a row, column of 3x3 can contain);
4. Maximize Cardinality (max): assesses an individual's fitness by counting the number of unique numbers (cardinality) existing in each row, column and 3x3 grid .

Please note that all the minimization functions should, ideally, converge to 0, while the maximization function should converge to 243 (case when all rows, columns and 3x3 grids have a cardinality of 9).

Selection & Genetic Operators

We have adapted the Charles library, developed in class, to our own problem. All genetic operators were adapted to work at the row level, as well as to contain a few more implementations that will be discussed at opportune time. For the sake of brevity and clarity, we will not provide details about the functioning of any operator, except to the ones that were not developed in class, but implemented by us from scratch, in which cases a quick description will be provided.

Selection Methods: for selection methods, we have implemented Tournament selection, Rank selection, Roulette Wheel and Stochastic Universal Sampling (SUS). SUS is a variation of the roulette wheel, designed to reduce the problem of no bias and minimal spread that's introduced when using the Fitness Proportionate Selection (aka Roulette Wheel), by taking advantage of a single random value to sample all of the individuals at once, and chosen at evenly spaced intervals, giving the weaker individuals the opportunity to be selected with higher chances than in the Roulette Wheel.

Crossover Operators: For crossover operators, we have implemented single point crossover, cycle crossover operating at the row level and PMX crossover, operating at the row-level as well.

Mutation Operators: For mutation operators, we have implemented swap mutation, inversion mutation and scramble mutation. All mutation operators were applied at the row-level and to a single row, chosen at random. In the scramble mutation, the chosen subset of genes is scrambled or shuffled randomly. For the inversion and scramble mutations, the length of mutated strings of the chosen row was selected at random.

Benchmarking & Statistical Analysis

For the benchmarking we have opted to start from a default setting, and benchmark each selection method, genetic operator and corresponding probabilities, as well as ideal population size and use, or not, of elitism, in a "tournament-type-of-manner", with the configuration showing the highest average best fitness (ABF) being chosen for the following "tournament". Every comparison between configurations was statistically evaluated. When ties happened, we evaluated the time each configuration took to run, and chose the lowest time-consuming configuration. Although we were not interested in the computational burden of our algorithm (our focus was to get the best scores possible), we used the ties as an opportunity to reduce the running time of our algorithm during the benchmarking process.

All of our benchmarking was performed using with the fitness function "Maximize Cardinality", as it showed to take the least time to run among all implemented fitness functions (Figure 1). We used the following "default" configuration:

- Selection: Roulette Wheel
- Crossover: Cycle crossover, with probability 0.8
- Mutation: Swap mutation, with probability 0.2
- No Elitism
- Pop size = 100
- Generations = 100

Every benchmarked configuration was performed for 100 runs. All produced data was tested for normality by using the Shapiro-Wilk test. A Rank Sum Test was found appropriate to benchmark all our analysis. The later was performed at the last generation of the different configurations. Statistical significance is represented in the corresponding figures when not obvious by eye, or when found relevant for taking conclusions. Please refer to the corresponding figure legend for more information about the statistical analysis performed in each situation.

For benchmarking selection methods, crossover and mutation operators, we compared the ABF of different configurations against the number of generations (as it should be the same). When benchmarking configurations with different population sizes, we guaranteed that every configuration performs the same number of fitness evaluations per run and plotted the ABF of each configuration against computational effort. Finally, a second metric, other than ABF, the success rate (SR), was used and presented when appropriate. SR measures the proportion of correctly solved sudokus among all performed runs (i.e. proportion of times our algorithm converged to a global maxima/minima).

III. Results

We started by choosing a fitness function to perform our benchmarking. Although we were not interested in the computational burden of our algorithm (our focus was to get the best scores possible), we did an initial analysis to check which fitness function took the least time to run (Figure 1), to optimize the benchmarking process as much as possible.

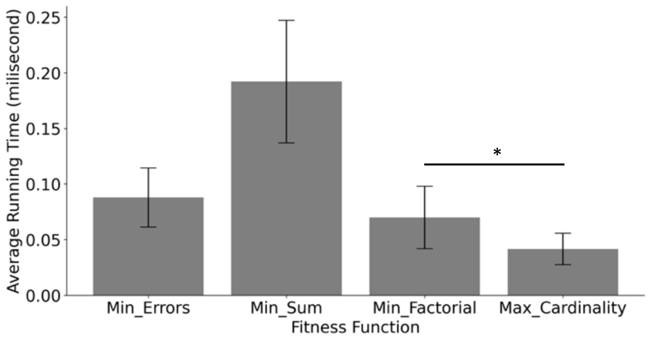


Figure 1: Running times of each implemented fitness function. Error bars represent standard deviation. The average Running Times between all fitness functions tested positive for all combinations with a Wilcoxon Rank Sum Test. * $p < 0.05$, Mann-Whitney U Test.

Since the Maximization fitness function took the least to run, we went ahead and started our benchmarking process with the selecting methods we have implemented. Fitness Proportionate and Stochastic selection showed a clear diminished performance (measured by the Average Best Fitness - ABF) relatively to the remaining two (Figure 2A). Rank selection and tournament selection did not show significantly different results in terms of performance. However, tournament selection did show to run faster relatively to rank selection (Figure 2B). Thus, we proceeded with tournament selection as the winner population selection method.

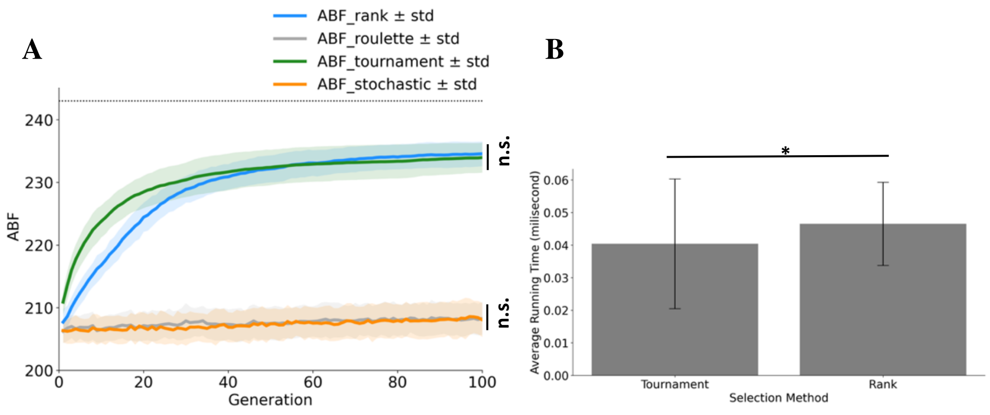
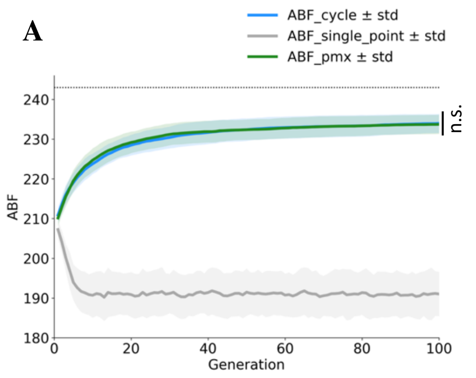


Figure 2: Benchmarking of population selection methods. **(A)** ABF for Rank, Roulette, Tournament and Stochastic selection throughout generations. All data tested negative for normality. Non-obvious statistical comparisons are indicated in the figure. **(B)** Running times of Tournament and Rank selection methods. * $p < 0.05$, Mann-Whitney U Test; n.s., not significant.

We next moved on to the benchmarking of crossover genetic operators (Figure 3). The extremely low ABF of the single point crossover was expected as the single crossover is the only implemented operator allowing the swap of elements between rows of an individual, thus introducing repeated elements into each row, which destroys our initialization of one number of each cardinality per row. Hence, the bad performance of the algorithm with this crossover operator came at no surprise. Cycle and PMX crossover operators were the winners of this run, with cycle crossover being more time efficient.

Moving on to mutation operators, we found the swap mutation to provide the highest ABF score (Figure 4A). We hypothesize that the reason for this is that both the inversion and scramble mutation operators change drastically the number sequence of the rows. Indeed, one should remember that we are performing mutation in vectors of 9 (maximum) numbers only; thus, a mutation that changes the entire order of the row, might be an overkill to produce a better fit individual in an algorithm with such hard-coded constraints. Conversely, a point swap mutation in such a small vector has good chance of producing the desired digit swap. We also tested different mutation probabilities and found that very high probabilities of mutation provide better ABF (this result will be discussed further on) (Figure 4B).



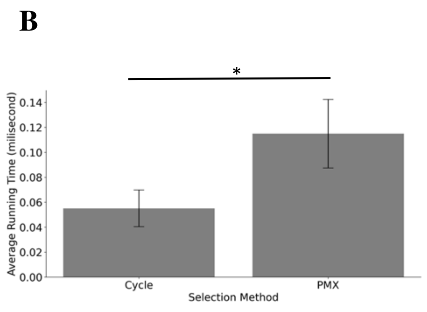


Figure 3: Benchmarking of crossover operators. **(A)** ABF for Cycle, Single point and PMX crossover operators, throughout generations. All data tested negative for normality. Non-obvious statistical comparisons are indicated in the figure. **(B)** Running times of Cycle and PMX crossover methods. * p < 0.05, Mann-Whitney U Test; n.s., not significant.

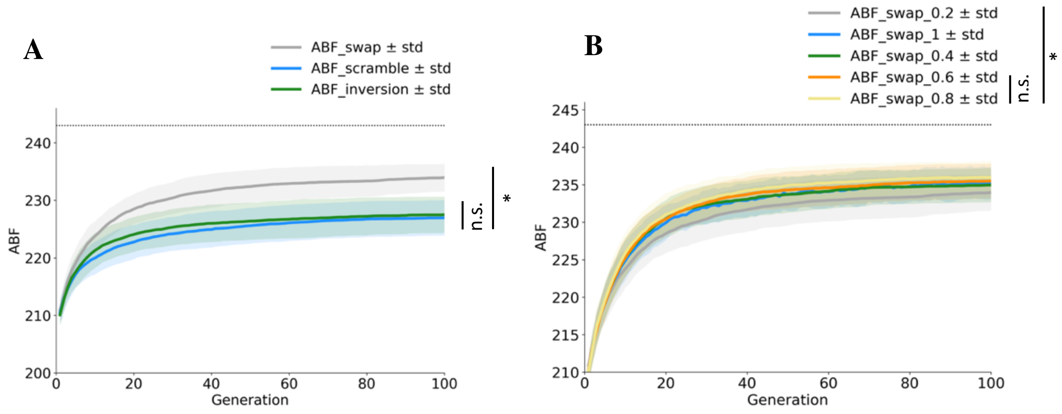


Figure 4: Benchmarking of mutation operators – Swap, Scramble and Inversion **(A)** - and respective mutation probability **(B)**, throughout generations. All data tested negative for normality. Non-obvious statistical comparisons are indicated in the figure. * p < 0.05, Mann-Whitney U Test; n.s., not significant.

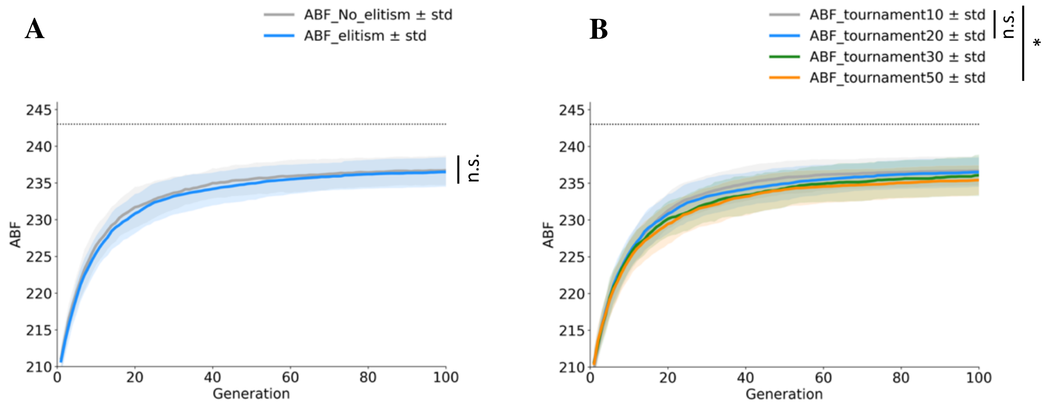
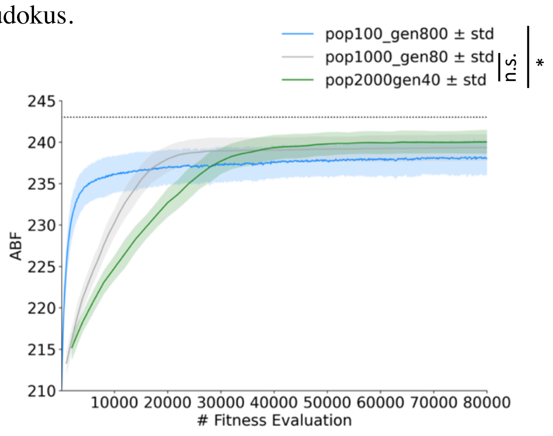


Figure 5: Benchmarking of elitism vs. non-elitism **(A)** and different sizes of the tournament selection method **(B)**, throughout generations. All data tested negative for normality. Non-obvious statistical comparisons are indicated in the figure. * p < 0.05, Mann-Whitney U Test; n.s., not significant.

Although no statistical difference was observed between both ON and OFF elitism configurations (Figure 5A), we have turned on elitism, thus ensuring that the best individuals are kept from generation to generation. We also tested the size of the tournament selection method on the performance of the algorithm (Figure 5B), and found that smaller sizes are preferable, likely because they allow the introduction of variance into the population with a higher probability. We thus moved on with a tournament size of 10. We kept the tournament size low even when increasing the population size (Figure 6). We found that the algorithm seemed to have a higher success rate when the tournament size was small for very large populations (comparison not shown). This could be due to the fact that, at later stages of the run, all the individuals are very similar; if the tournament size is small, the chance of selecting individuals who suffered mutations that can be beneficial is increased, even if they don't show as higher finesses as the current best individual.

Although we observed higher ABFs when increasing the population size (Figure 6A), this also highly increased the running time of the algorithm, making this a very inefficient and not ideal configuration to reach convergence. A population size of 2000 showed a higher success (13%) rate than a population of 1000 individuals (8%), so we moved on with the highest tested population size. Finally, we ran this configuration in every of the initially designed fitness functions and compared success rates (Figure 6B). We found that with the maximization fitness function the success rate of the algorithm was clearly higher than for all other functions. This success rate lowered to levels of 5% when the algorithm was fed with hard sudokus.

A



B

	Min. Errors	Min. Sum	Min. Factorial	Max. Cardinality
Success Rate	0.05	0.04	0	0.13

C

	Easy	Medium	Hard
Success Rate	0.13	0.12	0.05

Figure 6: (A) Benchmarking of different population sizes, against number of fitness evaluations. All data tested negative for normality. Non-obvious statistical comparisons are indicated in the figure. * $p < 0.05$, Mann-Whitney U Test; n.s., not significant. **(B)** Success rate of final algorithm using the 4 different implemented fitness functions. **(C)** Success rate of final algorithm at solving different difficulty-levels of sudoku puzzles (max fitness function).

IV. Discussion

In this work, we developed a genetic algorithm to solve sudokus. We imposed a few hard constraints at the initialization, in order to increase efficiency and maximize chances of success. We implemented multiple genetic operators, fitness functions (both maximization and minimization) and population selection methods. We designed a strategy to benchmark the different configurations and deployed a final algorithm.

Our deployed GA shows to not be able to converge to a global optimum often, getting stuck in a local optimum, which is demonstrated by the low achieved success rates. We hypothesize that this is happening mainly due to lack of diversity within the population. In fact, once increased the population size and maintained the tournament size at a low value, we saw an increase in the success rate of the model, likely due to random selection of individuals with low fitness, which have the potential to mate and introduce beneficial changes in the population. Thus, implementing strategies to increase the diversity in the population, while avoiding very similar individuals to mate and keep being in the population, will be our principal focus of attention to try improving the current develop GA. Moreover, we found our GA to perform best with very high mutation rates, to levels in pair with the crossover probability, which could seem counterintuitive. However, in final stages of the search of the global optima for the solution of the sudoku, it may be important to keep mutation rates high, in order to promote generation of diversity in an already very similar population.

Finally, our benchmark strategy allowed to draw a couple conclusion about the (in)efficiency of multiple genetic operators and population selection methods; for example, we observed that selection methods similar to Fitness Proportionate Selection allow no bias and minimal spread, thus leading to very small variation quickly, which can be detrimental to the convergence of the GA.

V. Conclusions & Prospects

Increasing the population size, and running the algorithm for long generations, increases the chance for a beneficial mutation to eventually happen - we got both higher ABF as well as Success Rate. However, this is computationally burdensome and time-consuming. As for next steps to continue the optimization of our algorithm, hopefully allowing us to reach the global optima more often and more efficiently, we will try the following implementations:

a) Implementing a strategy of penalization that penalizes individuals that are alike, thus increasing the diversity in the population, hopefully allowing to reach the global optima;
b) Implementing a strategy to promote mating between individuals that are variant and restrict mating between similar individuals;
c) Adapting the mutation rate according to the generation, making it increase as generations go by, providing more variation at final generations, hence increasing the chance of reaching the global optima.