Data Science and Advanced Analytics

COMPUTATIONAL INTELLIGENCE FOR OPTIMIZATION

2021/2022 PROJECT

SUDOKU SOLVER

Group 7:

Yuri Marques Ferreira - m20180427

Francisco Pita Olival Trindade de Ornelas - m20210660

Leonor de Almeida Candeias - m20210990

Miguel Lopes – m20211001

https://github.com/MiguelOva/Sudoku-solver-GA

Index

Introduction

This project makes use of the Genetic Algorithm (GA) knowledge to solve an optimization problem. Our group choose the Sudoku Solver problem to that purpose using the Charles Genetic Algorithm Library developed throughout the semester.

Sudoku is a number-placement puzzle that has achieved remarkable popularity in the past few years. The goal of this puzzle is to fill a 9x9 grid with the digits 1 to 9, subject to the following constraints: each row, each column, and each of the nine 3x3 sub-grids must contain a permutation of the digits from 1 to 9.

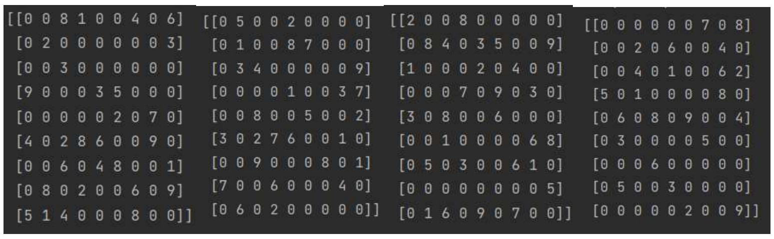
Genetic algorithms (GAs) are a class of evolutionary algorithms based on the fundamental concepts of evolution in biology. The classic GA starts with a random population of initial solutions. These are ranked by a fitness function that determines the goodness of a solution. Better solutions are selected and recombined. The GA runs until a solution is found, or the maximum number of generations is reached. The best (minimum) and worst (maximum) cost function results are output for each generation.

Implementation

Genetic algorithms encode a population of solutions, rank them with a fitness function, and apply computational analogues of selection, recombination, and mutation to evolve better solutions.

Initialization of the puzzle

Firstly, four initial problems represented in lists with numbers from 1 to 9 were created simulating the predefined filled positions on a 9x9 grid of the Sudoku puzzle. It starts with a very easy approach, followed by an easy one, then a moderate version, and finally a hard one. This becomes interesting to test the several solutions within the puzzle according to their complexity/difficulty level. All positions that need to be filled as solutions to the problem are identified as zeros:



```
[[0 0 8 1 0 0 4 0 6]   [[0 5 0 0 2 0 0 0 0]   [[2 0 0 8 0 0 0 0 0]   [[0 0 0 0 0 0 7 0 8]
 [0 2 0 0 0 0 0 0 3]    [0 1 0 0 8 7 0 0 0]    [0 8 4 0 3 5 0 0 9]    [0 0 2 0 6 0 0 4 0]
 [0 0 3 0 0 0 0 0 0]    [0 3 4 0 0 0 0 0 9]    [1 0 0 0 2 0 4 0 0]    [0 0 4 0 1 0 0 6 2]
 [9 0 0 0 3 5 0 0 0]    [0 0 0 0 1 0 0 3 7]    [0 0 0 7 0 9 0 3 0]    [5 0 1 0 0 0 0 8 0]
 [0 0 0 0 0 2 0 7 0]    [0 0 8 0 0 5 0 0 2]    [3 0 8 0 0 6 0 0 0]    [0 6 0 8 0 9 0 0 4]
 [4 0 2 8 6 0 0 9 0]    [3 0 2 7 6 0 0 1 0]    [0 0 1 0 0 0 0 6 8]    [0 3 0 0 0 0 5 0 0]
 [0 0 6 0 4 8 0 0 1]    [0 0 9 0 0 0 8 0 1]    [0 5 0 3 0 0 6 1 0]    [0 0 0 6 0 0 0 0 0]
 [0 8 0 2 0 0 6 0 9]    [7 0 0 6 0 0 0 4 0]    [0 0 0 0 0 0 0 0 5]    [0 5 0 0 3 0 0 0 0]
 [5 1 4 0 0 0 8 0 0]]   [0 6 0 2 0 0 0 0 0]]   [0 1 6 0 9 0 7 0 0]]   [0 0 0 0 0 2 0 0 9]]
```

Fig. 1 – Very easy, easy, moderate, and hard levels of Sudoku problems to solve respectively.

Following this, a class *Individual* was created with three different parameters *cues, i_representation* and *fill_values). Cues* is a string in list shape, containing 81 numbers corresponding to each of the entries in the Sudoku 9x9 matrix problem. The *__init__* method creates the attribute *self.cues* which corresponds to the matrix form of this list as exemplified above in Fig.1. It also creates the attributes *given_index* and *to_fill_index* which corresponds to the indexes of the entries in the matrix which were already filled and those which need to be filled respectively. If the *i_representation* argument is "None" there are two possible outcomes: either the *fill_values* argument is also "None" or it is not. In the case of *fill_values* also being "None", an attribute called *fill_values* - of type list with the length of the number of entries which need to be filled in the matrix - is created and filled with random integers from 1 to 9.

The final complete matrix of the puzzle is the result of a property called *representation*, which we defined after the *__init__* method. In the property definition, we create a variable *rep* which is a deep copy of the *self.cues* attribute (exemplified in Fig.1). Then, for each i in self.to_fill_index, rep[index] is = self.fill_values[self.to_fill_index.index(i)] and rep is then returned as the representation. We found this to be a good solution, because it allows the matrix representation to be automatically updated every time the fill_values list is changed which is very useful for the application of genetic operators. We chose to fill the matrix based on this fill_values list because it would be easier to apply the genetic operators like crossover and mutation (and making sure it is only on the spots which are meant to be filled), while also having the benefit of the representation matrix which allows us to use 2-d indexes which we found more intuitive for this problem than the original 1-d list of numbers.

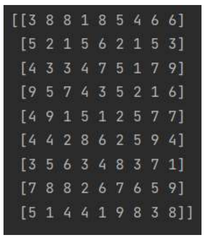
Although we didn´t plan on this particular scenario to happen in this project, in the case the i_representation argument is none (the final matrix is passed as an argument) an attribute I_representation is created and set as i_representation.

After having created the Individual class (remaining methods and dunder methods can be checked in the code), we then moved on to the Population class, which has as parameters (size, optim and **kwargs). Put simply, size is the number of individuals to be created, optim specifies whether this is a minimization or maximization problem and **kwargs is to allow for the creation of individuals with their specific parameters.

We then adapted the original evolve method in the Charles library to our specific problem, with one of the main changes being that crossover receives as parameters the parents fill_values list and the mutation operates on the offspring lists. The new population then receives as inputs individuals with the different final offspring fill_values until it is complete (len(new_pop) < self.size(original number of individuals). Both the crossover and mutation are probabilistic events which only happen with a certain probability co_p and mu_p respectively and elitism is a user dependent functionality which when set to True saves the best individual for the next one. The complete list of parameters in evolve is then ((self, gens, select, crossover, mutate, co_p, mu_p, elitism, **kwargs) with gens indicating the number of generations, select the selection algorithm (how to choose which individuals are selected as parents) , crossover the crossover algorithm, mutate the mutation algorithm, co_p and mu_p as crossover and mutation probabilities, elitism the application or not of the elitism functionality and **kwargs to allow for the creation of individuals with their specific parameters.

The function ***make_matrix*** developed to analyse each problem when initialized as shown in the previous Fig.1 verifies if each cell from the puzzle is populated with zero or if it is with a number other than zero. Either way, it returns the possible coordinates from the puzzle (built as a matrix) to check in which cells a solution can be populated (all cells filled with zeros).

The following approach was to replace all zeros within the desired matrix/puzzle problem with random numbers comprehended between 1 and 9 using the *representation* method from the *Individual* class (Fig. 2) - our Initial Population. This allows us to analyse and identify all numbers that are repeated by row, by column, and within the respective block of 9 digits.



```
[[3 8 8 1 8 5 4 6 6]
 [5 2 1 5 6 2 1 5 3]
 [4 3 3 4 7 5 1 7 9]
 [9 5 7 4 3 5 2 1 6]
 [4 9 1 5 1 2 5 7 7]
 [4 4 2 8 6 2 5 9 4]
 [3 5 6 3 4 8 3 7 1]
 [7 8 8 2 6 7 6 5 9]
 [5 1 4 4 1 9 8 3 8]]
```

Fig. 2 – Randomly filled matrix/puzzle.

The Fitness Function

To develop our fitness function, several approaches were considered in order to check all repeated numbers by row, column, and block of numbers (sub-grid of 3x3). Fewer repeated numbers in these previous three situations means a lower fitness value (closer to zero).

In our first tries, we defined the ***entry fitness*** function which for each entry in the sudoku matrix, calculated how many times that number appeared on its row, its column, and its block, and then average those values. The result is the entry fitness for a single entry in the sudoku matrix. As such, the ***get_fitness*** function would then average this fitness for all the entries in a single individual (i.e., sudoku matrix) and that would be the individual´s fitness. Not only was this approach more time-consuming, it also didn´t provide satisfactory results.

Another approach taken was to use the functions we had previously defined ***repeated_by_row***, ***repeated_by_col***, and ***repeated_by_block*** that for each individual return two objects: *(*We will exemplify with the repeated_by_row, but the logic is the same for columns and blocks)*

- An array with the values on each row as well as their counts, so if a row was [1,1,1,2,2,2,2,2,2] the array would be ([[1,3], [2,6]]);
- The individual´s row fitness which starts at 0 and increases by (x-1) for each count. This way, when the count is 1, the count doesn´t change and the fitness doesn´t get worse, considering we are defining this problem as a minimization one. In our example, if the entire matrix had only that row, the individual´s row fitness would be 7 ((3-1) + (6-1)).

We initially tried using only the row fitness of the individual as the fitness function. Even though the algorithm was quick to solve the repeated entries in the rows, there were still quite some repeated entries in the blocks and columns. Our next option was then to set the fitness function as the sum of the row fitness, block fitness and column fitness. This approach provided considerably better, more balanced results although in most cases, after a considerable number of generations, it got close to the solution but never actually found it. The next attempt was to define the fitness function as the product of the row, block and column finesses but this came with a problem. If any one of the 3 got to 0, the final fitness function would turn to 0 and so to solve this problem we defined the fitness as function as (row_fitness+1) * (block_fitness+1) * (col_fitness+1).

Selection

Parent Selection is the process of selecting parents' which mate and recombine to create off-springs for the next generation. Parent selection is very crucial to the convergence rate of the GA as good parents drive individuals to a better and fitter solutions.

Tournament – tournament. Tournament Selection is a Selection Strategy used for selecting the fittest candidates from the current generation in a Genetic Algorithm. With this technique, to select one individual, we do not have to evaluate the fitness of all the individuals in the population, but only a part of them.

Tournament selection involves running several "tournaments" among a few *k* individuals chosen at random from the population. Every time that an individual must be selected, a number k of individuals are chosen randomly, with uniform distribution, from the current population. The winner of each tournament, the individual presented with best fitness is selected for crossover.

Ranking – ranking. Ranking Selection is an algorithm that first organizes the individuals in the population based on their fitness, ranking them from worst to best and then selecting one in the group based on that.

After ranking the individuals, each one will have a probability of being selected that is calculated by the position in the ranking, divided by the total sum of the ranking indexes(in case of linear function.)

Roulette – roulette_wheel. This selection method starts by getting the fitness of the population and then dividing eaching one by the total sum of the individual fitnesses, this is the propability of each individual beign selected.

The algorithm then chooses one ramdoly, the bigger the fitness, the bigger the probability of beign selected, in maximization problems.

Mutation

In evolution, mutation helps with natural selection, things mutate, and successful mutations survive, eventually breeding new, stronger, fitter species.

In simple terms, mutation may be defined as a small random tweak in the chromosome, to get a new solution. It is used to maintain and introduce diversity in the genetic population and is usually applied with a low probability – mu_p (mutation rate). If the probability is very high, the GA gets reduced to a random search.

Mutation is the part of the GA which is related to the “exploration” of the search space. Mutation Operator is an innovation operator, and it needs only one parent to work on. It does so by selecting a few genes from our selected chromosome and apply the desired algorithm.

Swap – swap_mutation. In swap mutation, simply put we select two positions on the chromosome at random and interchange the values. This is common in permutation-based encodings.

Inversion – inversion_mutation. In this, from the entire chromosome, a subset of genes is chosen, and instead of scrambling or shuffling randomly, we merely invert the entire string in the subset.

Sudoku_Mutation - This was a mutation technique we tried to implement which was specific to the sudoku problem. For the mutation point, the possible values (I.e. points not in its column, row or block) are computed and the mutation point is substituted by one of those possible values.

Crossover

The crossover operator is analogous to reproduction and biological crossover. In this more than one parent is selected and one or more off-springs are produced using the genetic material of the parents. Crossover is usually applied in a GA with a high probability – co_p (crossover rate).

Single Point – single_point_co .It works by executing a one-point crossover on the input individuals. The two individuals' points are modified in place. The resulting individuals will respectively have the length of the other. One arbitrary combination point is selected for both parents' chromosomes and after these combinations of points are swapped with each other, giving birth to two new off springs.

Dual Point – **single_point_co** .It works by executing a two-point crossover on the input individuals. The two individuals' points are modified in place. The resulting individuals will respectively have the length of the other. Remarkably similar to one-point crossover but instead of one, two arbitrary conditions points are selected.

Arithmetic – arithmetic_co. It works by taking a percentage of each parent gene and adds them to produce new solutions. The percentage of each parent gene present in the offspring is determined by a parameter alpha. This operator linearly combines the two parent chromosomes, and by the combination of these chromosomes, two off springs are produced.

Partially Mapped -- pmx_co. It works by executing a partially matched crossover (pmx) on the input individuals. The two individuals are modified in place. This crossover method builds an offspring by choosing a subsequence of elements from one parent and preserving the order and position of as many elements as possible from the other parent. A subsequence of elements if selected by choosing two random cut points, which serve as boundaries for the swapping operations.

Results

For a Population of 2000 individuals, 100 generations and parameters **co_p** = 0.6 and **mu_p** = 0.14 of the *evolve* function applied to the Population, several combinations of selection, crossover and mutations were developed to analyse our final fitness value as well as its solution accomplished with this algorithm. The fitness value returned in each of the following scenarios (Table 1) closer to zero is the one correspondent to the best result/solution of this sudoku problem:

Problem	Select	Crossover	Mutate	Fitness	Problem Solution
Very Easy				2.666(6)	Fig A
Easy				3.666(6)	Fig B

Moderate	tournament			2.333(3)	Fig C
Hard				5.666(6)	Fig D
Very Easy	ranking_selection	single_point_co	swap_mutation	1.333(3)	
Easy				2.3333(3)	
Moderate				2.0	
Hard				4.333(3)	

Table 1 – Possible combinations of selection, crossover and mutation operators when applied to different complexity Sudoku problems. This fitness value is the best value among all the fitness values encountered for each try with these specific parameters. Other combinations with the different crossovers and mutations presented were tried(Table 2) but these had the best performance

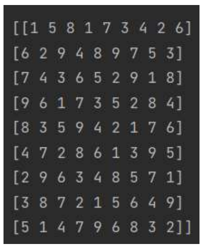
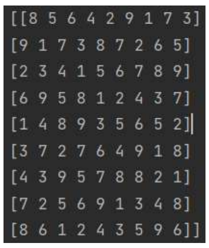
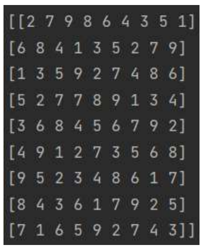
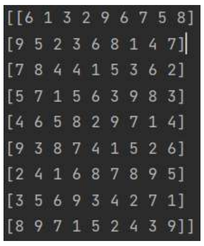

```
[[1 5 8 1 7 3 4 2 6]
 [6 2 9 4 8 9 7 5 3]
 [7 4 3 6 5 2 9 1 8]
 [9 6 1 7 3 5 2 8 4]
 [8 3 5 9 4 2 1 7 6]
 [4 7 2 8 6 1 3 9 5]
 [2 9 6 3 4 8 5 7 1]
 [3 8 7 2 1 5 6 4 9]
 [5 1 4 7 9 6 8 3 2]]
```

```
[[8 5 6 4 2 9 1 7 3]
 [9 1 7 3 8 7 2 6 5]
 [2 3 4 1 5 6 7 8 9]
 [6 9 5 8 1 2 4 3 7]
 [1 4 8 9 3 5 6 5 2]
 [3 7 2 7 6 4 9 1 8]
 [4 3 9 5 7 8 8 2 1]
 [7 2 5 6 9 1 3 4 8]
 [8 6 1 2 4 3 5 9 6]]
```

```
[[2 7 9 8 6 4 3 5 1]
 [6 8 4 1 3 5 2 7 9]
 [1 3 5 9 2 7 4 8 6]
 [5 2 7 7 8 9 1 3 4]
 [3 6 8 4 5 6 7 9 2]
 [4 9 1 2 7 3 5 6 8]
 [9 5 2 3 4 8 6 1 7]
 [8 4 3 6 1 7 9 2 5]
 [7 1 6 5 9 2 7 4 3]]
```

```
[[6 1 3 2 9 6 7 5 8]
 [9 5 2 3 6 8 1 4 7]
 [7 8 4 4 1 5 3 6 2]
 [5 7 1 5 6 3 9 8 3]
 [4 6 5 8 2 9 7 1 4]
 [9 3 8 7 4 1 5 2 6]
 [2 4 1 6 8 7 8 9 5]
 [3 5 6 9 3 4 2 7 1]
 [8 9 7 1 5 2 4 3 9]]
```

Fig A **Fig B** **Fig C** **Fig D**

Problem	Select	Crossover	Mutate	Fitness
Very easy	tournament	pmx_co	Swap_mutation	6.0
		Single_point_co	Inversion_mutation	5.333(3)
		pmx_co	Inversion_mutation	12.333(3)
		Artithmetic_co	Sudoku_mutation	7.0

Table 2: Other results given these combinations with very easy problem as reference

Results and Conclusion

The previous solutions found are from the Very Easy and Easy Sudoku problems respectively. Moderate and Hard Sudoku problems were also tried to be solved with this algorithm but unfortunately no solutions were found successfully. With this in mind, our focus was than only in the only problems with available solutions. Using tournament in selection together with single_point_co in crossover and swap_mutation in mutation, they all return the best fitness values in both Very Easy and Easy problems (2,(6) and 3,(6) respectively).

It is also possible to verify that a greater complexity level didn´t always equate to worse results. Also worth noting that different selection methods have different needs of population and convergence speeds.

References

http://micsymposium.org/mics_2009_proceedings/mics2009_submission_66.pdf