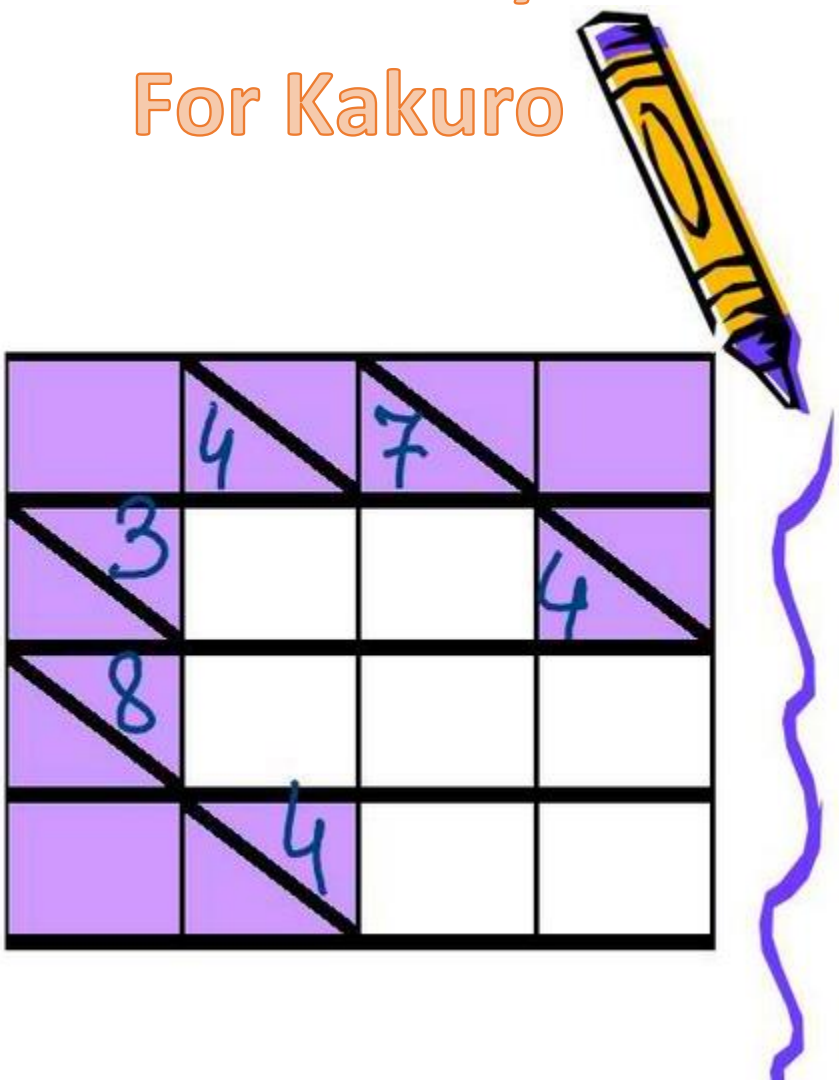
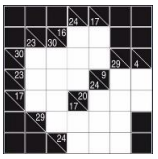
Evolutionary AI For Kakuro





Contents

Kakuro

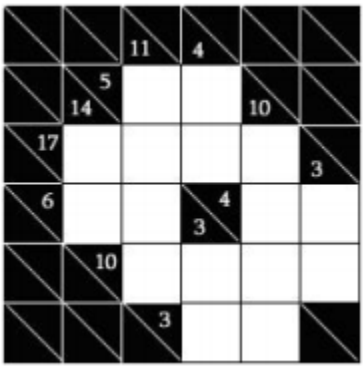
General

Kakuro is a kind of logic puzzle that is often referred to as a mathematical transliteration of the crossword. Kakuro puzzles are regular features in many math-and-logic puzzle publications across the world. The canonical Kakuro puzzle is played in a grid of filled and barred cells, "black" and "white" respectively. Puzzle sizes vary widely. Apart from the top row and leftmost column which are entirely black, the grid is divided into "entries" — lines of white cells — by the black cells. The black cells contain a diagonal slash from upper-left to lower-right and a number in one or both halves, such that each horizontal entry has a number in the black half-cell to its immediate left and each vertical entry has a number in the black half-cell immediately above it. These numbers, borrowing crossword terminology, are commonly called "clues".

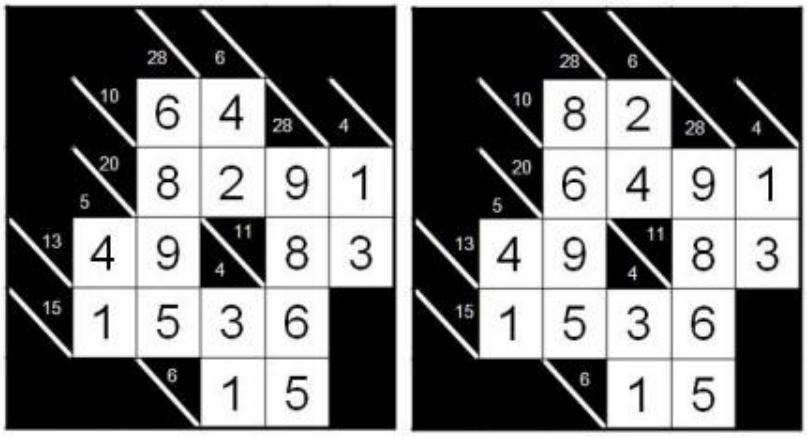
The objective of the puzzle is to insert a digit from 1 to 9 inclusive into each white cell such that the sum of the numbers in each entry matches the clue associated with it and that no digit is duplicated in any entry. It is that lack of duplication that makes creating Kakuro puzzles with unique solutions possible. Like Sudoku, solving a Kakuro puzzle involves investigating combinations and permutations. There is an unwritten rule for making Kakuro puzzles that each clue must have at least two numbers that add up to it, since including only one number is mathematically trivial when solving Kakuro puzzles.

Description of the problem

Kakuro puzzles consist of an n × m grid containing black and white cells. All white cells are initially empty and are organized into overlapping continuous runs that are exclusively either horizontal or vertical. A run-total, usually given in black “clue” cell, is associated with each and every run, and the puzzle is solved by entering values (typically from the range [1-9]) into the white cells such that each run sums to the specified total and no digit is duplicated in any run. Assuming only numbers in the range 1, . . . , 9 are used, a run can be between one and nine cells in length with a corresponding run-total in the range 1, . . . , 45, although the majority of published puzzles contain runs that are at least two cells in length. The figure below shows a sample Kakuro puzzle consisting of a 5 × 5 grid:



Most published puzzles are well-formed, meaning that only one unique solution exists. Such puzzles are also called promise-problems (the promise being a unique solution). An example of a puzzle that possesses multiple solutions is shown in the figure below:



Kakuro is very popular with *jigsaw* puzzles and often publishes in Israeli magazines and on the last pages of weekly newspapers. At present, very little has been published specifically on Kakuro and its related puzzles, however, their solution has been shown to be NP-Complete, through demonstrating the relationship between the Hamiltonian Path Problem, 3SAT (the restriction of the Boolean satisfiability problem) and Kakuro.

In this project, we tried to solve the problem of the Kakuro game using the genetic algorithm and genetic programming and find out how much they will help us in cutting this problem.

Terminology

The size of the playing field (grid) - the number of rows and the number of columns.

White cells - cells of the playing field, which must be filled in with digits from 1 to 9.

Black cells - cells of the playing field which cannot be changed.

Clue cells - cells of the playing field that contain two numbers, the first number is the sum of the elements of the group horizontally, the second number is the sum of the elements of the group vertically.

Horizontal group - a continuous sequence of white cells horizontally.

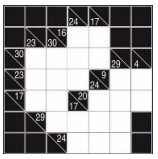
Vertical group - a continuous sequence of white cells vertically.

Group sum - the sum of all elements of one group.

Duplicate values - elements that occur in the same group more than once.

GA light version algorithm with default parameters.

GA pro version - algorithm with optimal parameters obtained during the experiment.



Literature Review

In this part, we will present a number of different methods to solve a Kakuro board.

- Brute Force - A given Kakuro puzzle could be solved exhaustively. That is, all possible values are tried in all cells, fully enumerating the search space in order to locate the solution. This approach is adequate for smaller grids or when a smaller range of numbers is to be used but very time consuming and inefficient for most puzzle grids where very large numbers of puzzle states would have to be checked

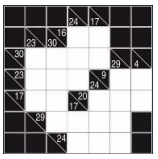
- Harmony Search - a metaheuristic approach to solving Kakuro puzzles. This method is widely known to be a music-inspired algorithm based on the process of Jazz improvisation of musicians. Founded on the concepts of randomly generating musical pitches from musician's memory, this algorithm uses a memory structure called Harmony Memory, which is used for generating random solutions in the beginning, and later is used to construct random combinations of these candidate solutions and evaluating these newly generated candidate solutions. This relatively new optimization technique has obtained its popularity due to fast convergence and effective avoidance of local optima.

- Binary integer programming – in this method, ten binary decision variables, $A_{i,j,k}$, are associated with every cell, where row i and column j specify the cell position, and k specifies an available value for assignment to the cell (with zero indicating a black square). Puzzle constraints and trivial constraints (such as there only being one value per cell and only values in the range can be added to white cells) are expressed explicitly. The solution is indicated by the collection of binary decision variables that are set to 1, showing which value k should be assigned to the cell at row i and column j. This approach works well for small puzzles, However, the large number of binary decision variables for larger grids may make this an inefficient general approach.

- Backtracking - A backtracking algorithm, employing a depth-first approach to examining the search space, can be made appropriate for the solution of Kakuro puzzles if suitable heuristics to guide the backtracker, and effective pruning conditions can be determined to reduce search space size. This approach begins with an empty grid and attempts assignments of values to each white cell in turn. It follows a depth-first enumeration of the search space, favouring the assignment of low numerical values, but tests within the algorithm ensure that some fruitless paths through the search space are avoided. This approach is ideal for small puzzles.

- Wisdom of Artificial Crowds - Genetic Algorithms can be augmented with post-processing by a Wisdom of Artificial Crowds (WoAC) to constrain the solution space after a number of generations. Using the WoAC method, compared to using GA alone, can reduce the time to a successful solution by a factor of 50% for easy and medium-difficulty puzzles. WoAC method that is used to constrain the solution space based on aggregate agreement between members of the population dataset.

- Retrievable genetic algorithm - differs from the standard GA in that the population is reinitialized after a set number of generations in an attempt to escape local optima.



Genetic Algorithm

At this stage of our project, we used a genetic algorithm to solve the Kakuro game problem presented above. A genetic algorithm is a heuristic-based algorithm for finding a solution to a specific problem. We searched for a solution to the game by randomly selecting, combining, and varying the desired parameters using methods reminiscent of biological evolution. For this, we encoded the problem in such a way that its solution could be represented as a set of parameters - genes. Below are the steps of the genetic algorithm that we have implemented.

Genetic Algorithm Procedure

1. <u>From Phenotype to Genotype</u>

To start with we need to define the genotype.

- In our task, the *phenotype* is represented as the playing field. We have implemented 4 playing fields (boards) of various difficulty levels:

❖

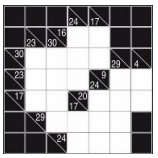
```
_________You choose easy_____

|XXXXX|| 0/17|| 0/16|
|17/ 0||_____||_____|
|16/ 0||_____||_____|

_________Number of spaces 4___
```

❖

```
_________You choose medium____

|XXXXX|| 0/ 3|| 0/ 9||XXXXX|
| 4/ 0||_____||_____|| 0/ 3|
| 8/ 0||_____||_____||_____|
|XXXXX|| 3/ 0||_____||_____|

_________Number of spaces 7___
```

❖

```
_________You choose hard_________

|XXXXX|| 0/ 7|| 0/23||XXXXX||XXXXX|
|10/ 0||_____||_____|| 0/24|| 0/ 7|
|16/ 0||_____||_____||_____||_____|
|22/ 0||_____||_____||_____||_____|
|XXXXX||XXXXX||13/ 0||_____||_____|

_________Number of spaces 12_________
```



❖

```
_________You choose expert_________

|XXXXX|| 0/13|| 0/23||XXXXX|| 0/43|| 0/ 5||XXXXX||XXXXX|
|17/ 0||_____||_____|| 4/ 0||_____||_____||XXXXX||XXXXX|
| 8/ 0||_____||_____||12/12||_____||_____||XXXXX||XXXXX|
|27/ 0||_____||_____||_____||_____||XXXXX||XXXXX||XXXXX|
|XXXXX||XXXXX|| 8/ 0||_____||_____|| 0/ 7||XXXXX||XXXXX|
|XXXXX||XXXXX||XXXXX|| 7/ 0||_____||_____|| 0/24|| 0/10|
|XXXXX||XXXXX||XXXXX||28/16||_____||_____||_____||_____|
|XXXXX||XXXXX||10/ 0||_____||_____|| 9/ 0||_____||_____|
|XXXXX||XXXXX||11/ 0||_____||_____||12/ 0||_____||_____|

_________Number of spaces 28_________
```

where:

- "XXX" - a black cell,
- "___" - a white cell,
- 0/0 - a clue cell.

- The *genotype* is encoded as a 1-D array, the size of which corresponds to the number of empty cells in the playing field. The array contains values from 1 to 9, which fill empty cells in the field. For this, the function of the line-by-line filling *filling_desk())* was implemented.

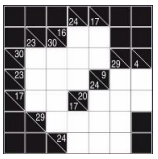
2. Initialization

Initialization of the initial *population* - at the first step, a set of candidates in the amount of 500 is generated for the solution, which gives new offspring as a result of the selection, mutation, and crossover operators.

3. Fitness Evaluation

The next step is the evaluation of candidates.

- To evaluate the fitness parameter, the fitness function estimates how close the resulting field is to the board's solution, taking into account the number of duplicate elements in a group horizontally and vertically and how much the sum of the group's elements differs from the clue. Zero value of the Fitness function means the right decision for a given game board (optimal value).
- In the event that duplicate elements are found in the same group, then fitness-value is decreased by adding negative value to its fitness. We consider all groups (vertically and horizontally) and calculate the number of duplicate values, which are the sum of the number of each value in the group, and minus 1. That is, if an element occurs in the group 1 time, then the amount of duplication is 0. And, for example, if the element met 5 times, the number of duplications is 4. Thus, we check all the values of the group and summarize the number of duplications of each value. As a result, we get the number of duplications in one group, and then sum these values of all groups.
- In the fitness function, we take into account all the discrepancies in all groups between the sum of the elements in the group and the corresponding clue. To do this,



we consider each group individually and summarize all the elements of the group, then subtract the resulting value from the clues, we take the difference modulo. Next, we summarize the values of all groups.

- We presented a fitness function as the sum of the number of duplicate elements in each group and the difference of each group between the sum and the clue:

$$F = F_1 + F_2 + F_3 + F_4,$$

where:

- F_1 - the number of duplicate values in horizontal groups,
- F_2 - the number of duplicate values in vertical groups,
- F_3 – the difference between the sum of elements and the clue in horizontal groups,
- F_4 - the difference between the sum of elements and the clue in vertical groups.

Thus, only fitness 0 means that we put the numbers right, and therefore we solved the Kakuro puzzle.

4. Parent selection

The selection operator selects representatives for the use of genetic operators. For selection, we used *Tournament selection*. Selected candidates are then passed on to the next generation. In a 7-way tournament selection, we select 7 individuals and run a tournament among them. Only the fittest candidate amongst those selected candidates is chosen and is passed on to the next generation. In this way many such tournaments take place and we have our final selection of candidates who move on to the next generation.

5. Crossover

We used *one-point crossover*, in this crossover, a random crossover point is selected and the tails of its two parents are swapped to get new offsprings. Resulting in two new representatives, with the genes of the first and second parent.

6. Mutation

The next step is the mutation operator. The function (*mut_k_point()*) was implemented, which takes the number of genes *k* and performs their change to a random digit. As the initial data of our experiment, we took *k = 1*, which corresponds to a one-point mutation, in which one gene of a random representative is mutated.

7. Final

The final stage of each iteration is the formation of a new population. Stop occurs on the *hundredth generation*.

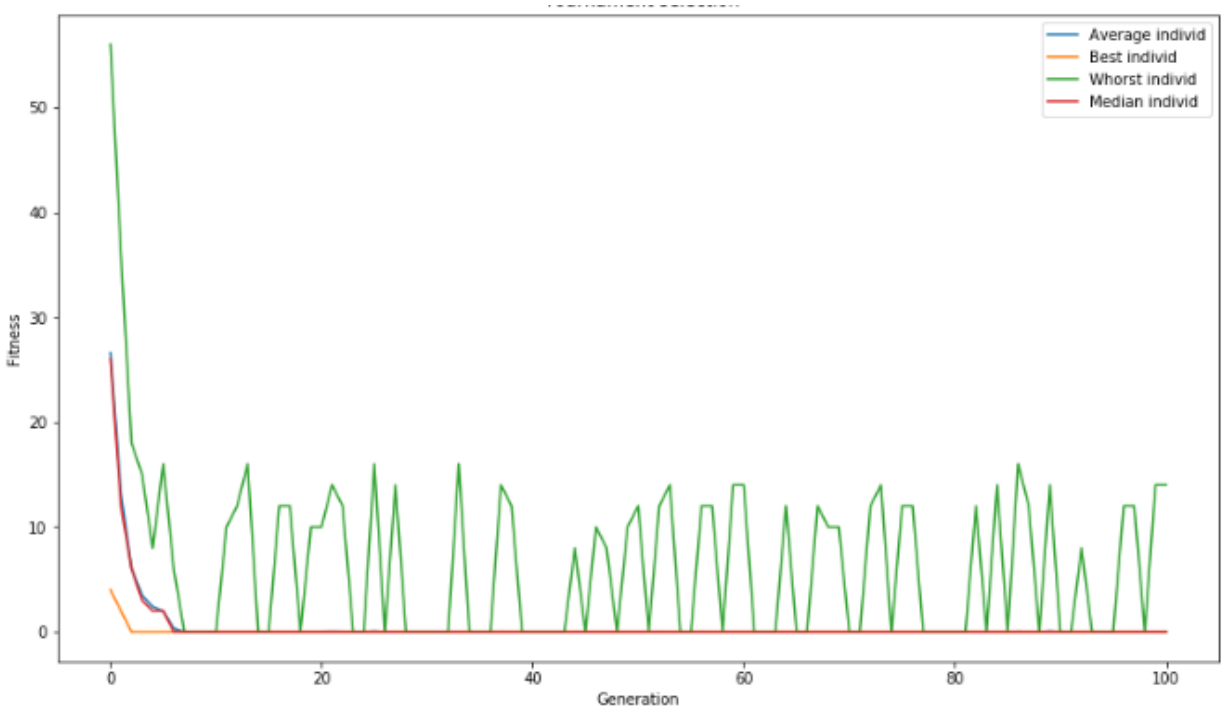
Experiments, Evolutionary Process and Results (GA)

As the initial data of our experiment, we selected the standard parameters for selection, crossover, and mutation, which are presented in the table:

Representation	array
Recombination	single point crossover
Recombination probability	70%
Mutation	one-point mutation
Mutation probability	0.1%
Parent selection	Tournament (7 individuals)
Population size	500 individuals
Initialization	random
Termination condition	100 generations

1. Easy level

The attempt to solve the playing field - Easy level:




```
The best solution:
|XXXXX|| 0/17|| 0/16|
|17/ 0||   8 ||   9 |
|16/ 0||   9 ||   7 |
Fitness: 0
Array: [8, 9, 9, 7]
Algorithms time 5.9301886558532715
```

- We made three runs. In all three runs, the correct solutions were found - which can be seen on the graph as convergence to a fitness value of 0.
- In all those runs, the best individual finds a solution almost immediately, from the first-third generation. The entire population converges to the optimal solution from about the tenth generation.
- As we see after the fifth generation, the entire population becomes approximately the same (the same fitness value). Individuals come to one certain value of fitness function, which does not change. In the future, a mutation of only one gene with a 0.1% mutation probability cannot lead to strong changes in the population as a

whole, only in point individuals does fitness deteriorate. The crossover, in this case, does not affect since the entire population is the same.

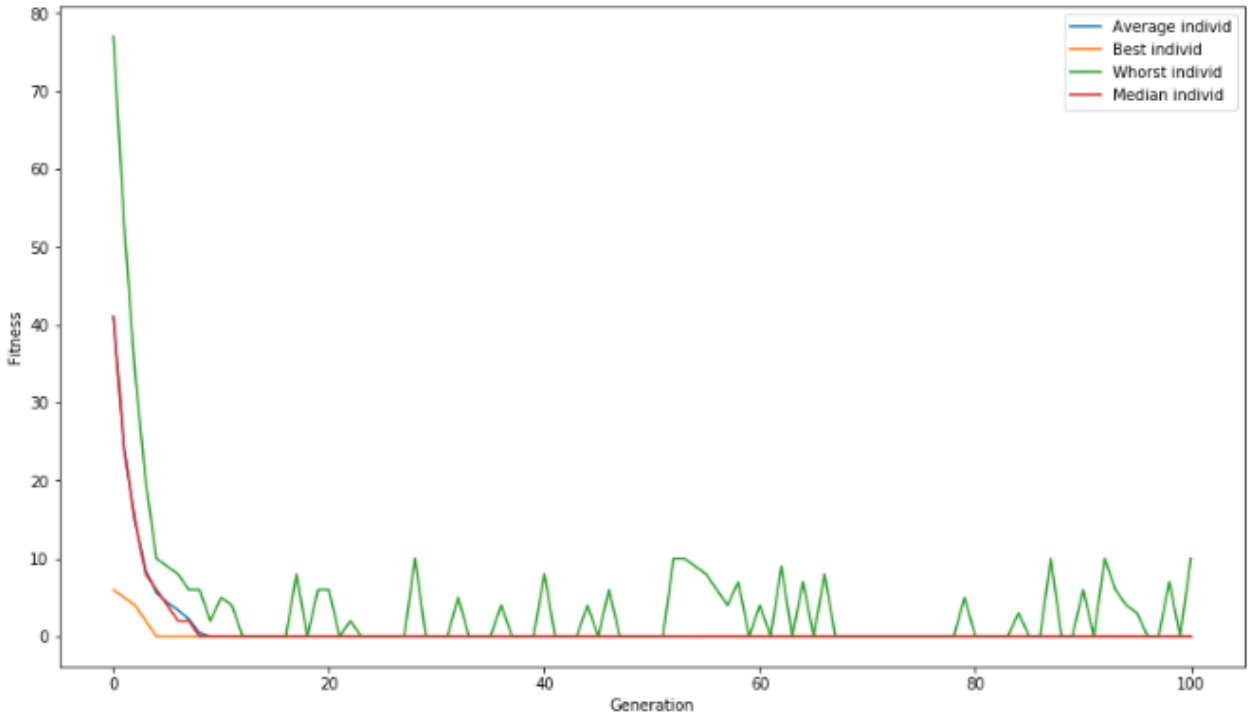
- The correct solution for the Easy level Kakuro game, which was found using the implemented genetic algorithm:

```
|XXXXX|| 0/17|| 0/16|
|17/ 0||   8 ||   9 |
|16/ 0||   9 ||   7 |
```

- Run time – 5.93s.

2. Medium level

The attempt to solve the playing field - Medium level:




```
The best solution:
|XXXXX|| 0/ 3|| 0/ 9||XXXXX|
| 4/ 0||   1 ||   3 || 0/ 3|
| 8/ 0||   2 ||   5 ||   1 |
|XXXXX|| 3/ 0||   1 ||   2 |
Fitness: 0
Array: [1, 3, 2, 5, 1, 1, 2]
Algorithms time 5.563068628311157
```

- We made three runs. We got the same results as in the previous case.
- In two of three runs, the correct solutions were found - which can be seen on the graph as convergence to a fitness value of 0.
- The best individual finds a solution almost immediately, from the first-third generation.
- As we see after the seventh generation, the entire population becomes approximately the same (the same fitness value). Individuals come to one certain value of fitness function, which does not change.

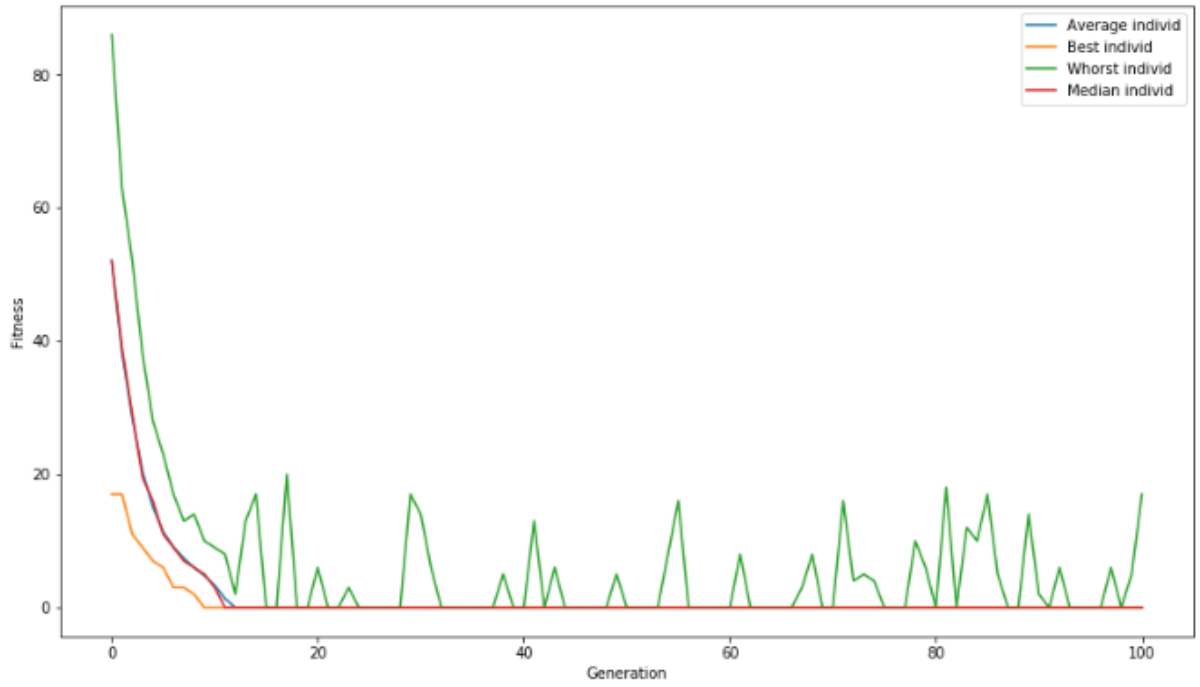
- The correct solution for the Medium level Kakuro game, which was found using the implemented genetic algorithm:

```
|XXXXX|| 0/ 3|| 0/ 9||XXXXX|
| 4/ 0||   1 ||   3 || 0/ 3|
| 8/ 0||   2 ||   5 ||   1 |
|XXXXX|| 3/ 0||   1 ||   2 |
```

- Run time – 5.56s.

3. Hard level

The attempt to solve the playing field - Hard level:




```
The best solution:
|XXXXX|| 0/ 7|| 0/23||XXXXX||XXXXX|
|10/ 0||   2 ||   8 || 0/24|| 0/ 7|
|16/ 0||   1 ||   6 ||   7 ||   2 |
|22/ 0||   4 ||   9 ||   8 ||   1 |
|XXXXX||XXXXX||13/ 0||   9 ||   4 |
Fitness: 0
Array: [2, 8, 1, 6, 7, 2, 4, 9, 8, 1, 9, 4]
Algorithms time 8.014682292938232
```

- We did three runs. In one of three runs, the correct solutions were found - which can be seen on the graph as convergence to a fitness value of 0.
- The best individual finds a solution since the eighth generation.
- As we see after the tenth generation, the entire population becomes approximately the same (the same fitness value). Individuals come to one certain value of fitness function, which does not change.

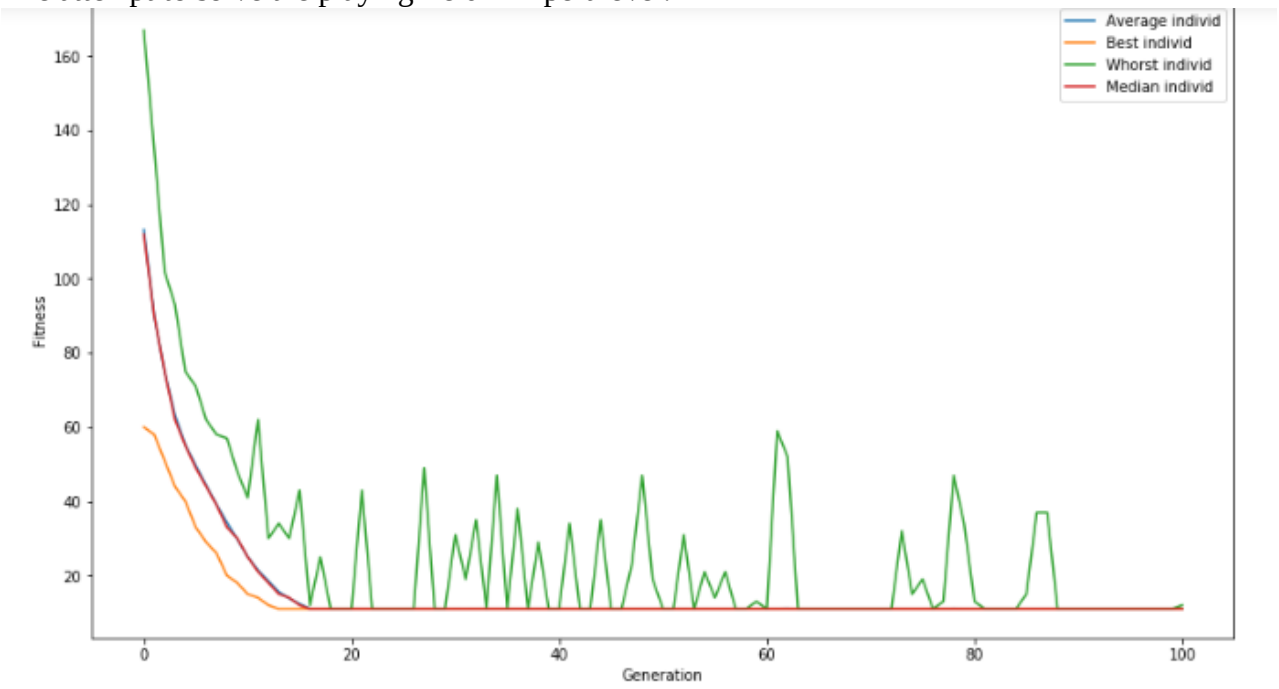
- The correct solution for the Hard level Kakuro game, which was found using the implemented genetic algorithm:

```
|XXXXX|| 0/ 7|| 0/23||XXXXX||XXXXX|
|10/ 0||   2 ||   8 || 0/24|| 0/ 7|
|16/ 0||   1 ||   6 ||   7 ||   2 |
|22/ 0||   4 ||   9 ||   8 ||   1 |
|XXXXX||XXXXX||13/ 0||   9 ||   4 |
```

- Run time– 8.01s.

4. Expert level

The attempt to solve the playing field - Expert level:




```
The best solution:
|XXXXX|| 0/13|| 0/23||XXXXX|| 0/43|| 0/ 5||XXXXX||XXXXX|
|17/ 0||   8 ||   9 || 4/ 0||   3 ||   1 ||XXXXX||XXXXX|
| 8/ 0||   1 ||   7 ||12/12||   8 ||   4 ||XXXXX||XXXXX|
|27/ 0||   4 ||   7 ||   5 ||   9 ||XXXXX||XXXXX||XXXXX|
|XXXXX||XXXXX|| 8/ 0||   7 ||   1 || 0/ 7||XXXXX||XXXXX|
|XXXXX||XXXXX||XXXXX|| 7/ 0||   7 ||   2 || 0/24|| 0/10|
|XXXXX||XXXXX||XXXXX||28/16||   9 ||   5 ||   9 ||   5 |
|XXXXX||XXXXX||10/ 0||   9 ||   1 || 9/ 0||   7 ||   1 |
|XXXXX||XXXXX||11/ 0||   7 ||   5 ||12/ 0||   8 ||   4 |
Fitness: 11
Array: [8, 9, 3, 1, 1, 7, 8, 4, 4, 7, 5, 9, 7, 1, 7, 2, 9, 5, 9, 5, 9, 1, 7, 1, 7, 5, 8, 4]
Algorithms time 15.994901895523071
```

- We launched more than three runs, but could not find a solution.

We assume that in order to solve the Kakuro problem using a genetic algorithm, it is necessary that the correct solution is found in the first twenty generations. If no correct solution has been found within the first twenty generations, we assume that a new run is needed. Also, special attention should be paid to the parameters of the genetic algorithm.

To search for the most suitable parameters, we created various combinations of parameters:

Tourn size	3	5	7	
Mutation point	1	2	3	
Crossover	One-Point	Two-Point		

Crossover Probability	0.3	0.7	0.9	
Mutation Probability	0.001	0.01	0.1	0.5
Population	100	800	2000	5000

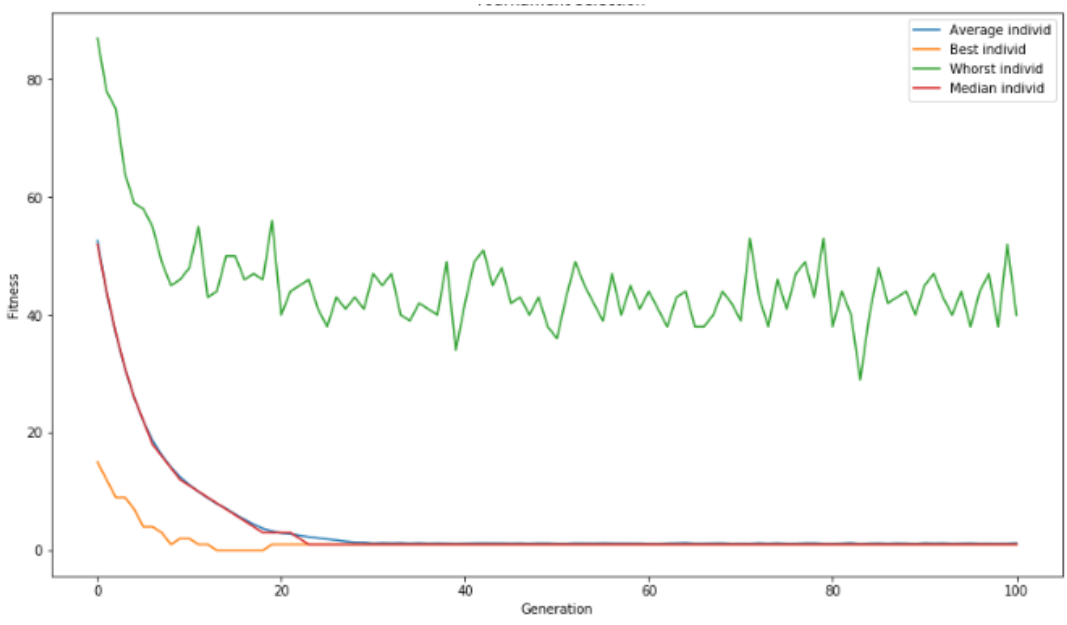
In the next step, we launched an algorithm for the playing field of the expert level with each combination of parameters three times. As a result, we chose a combination of parameters that allows us to win (Fitness = 0) in the Kakuro game at the expert level the most times.
We decided to dwell on the following parameters of the genetic algorithm:

Recombination	two point crossover
Recombination probability	90%
Mutation	three-point mutation
Mutation probability	1%
Parent selection	Tournament (3 individuals)
Population size	5000 individuals

Let's see how these parameters helped us solve the problem of the Kakuro game.

5. Hard level (second attempt)

The second attempt to solve the playing field - Hard level (second attempt):



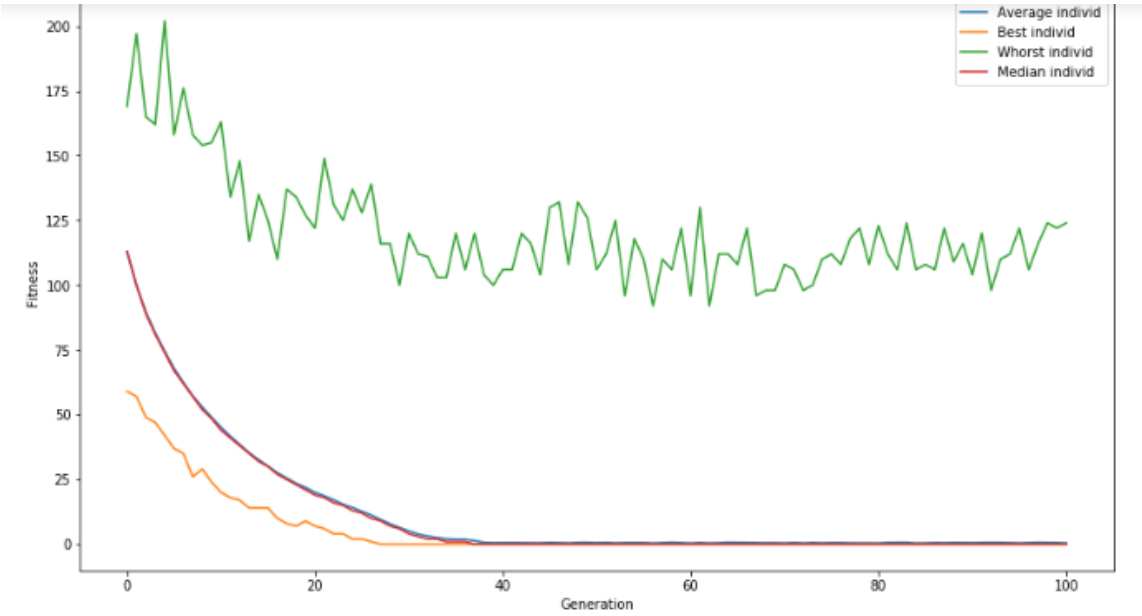

```
The best solution:
|XXXXX|| 0/ 7|| 0/23||XXXXX||XXXXX|
|10/ 0||   2 ||   8 || 0/24|| 0/ 7|
|16/ 0||   1 ||   6 ||   7 ||   2 |
|22/ 0||   4 ||   9 ||   8 ||   1 |
|XXXXX||XXXXX||13/ 0||   9 ||   4 |
Fitness: 0
Array: [2, 8, 1, 6, 7, 2, 4, 9, 8, 1, 9, 4]
Algorithms time 109.9023756980896
```

- We had three runs. In all three starts, we managed to find the right solution for the game.

Next, we tried to solve the expert level Kakuro game.

6. Expert level (second attempt)

The second attempt to solve the playing field – Expert level (second attempt):




```
The best solution:
|XXXXX|| 0/13|| 0/23||XXXXX|| 0/43|| 0/ 5||XXXXX||XXXXX|
|17/ 0||   8 ||   9 || 4/ 0||   3 ||   1 ||XXXXX||XXXXX|
| 8/ 0||   2 ||   6 ||12/12||   8 ||   4 ||XXXXX||XXXXX|
|27/ 0||   3 ||   8 ||   9 ||   7 ||XXXXX||XXXXX||XXXXX|
|XXXXX||XXXXX|| 8/ 0||   3 ||   5 || 0/ 7||XXXXX||XXXXX|
|XXXXX||XXXXX||XXXXX|| 7/ 0||   6 ||   1 || 0/24|| 0/10|
|XXXXX||XXXXX||XXXXX||28/16||   9 ||   6 ||   8 ||   5 |
|XXXXX||XXXXX||10/ 0||   9 ||   1 || 9/ 0||   7 ||   2 |
|XXXXX||XXXXX||11/ 0||   7 ||   4 ||12/ 0||   9 ||   3 |
Fitness: 0
Array: [8, 9, 3, 1, 2, 6, 8, 4, 3, 8, 9, 7, 3, 5, 6, 1, 9, 6, 8, 5, 9, 1, 7, 2, 7, 4, 9, 3]
Algorithms time 242.42257595062256
```

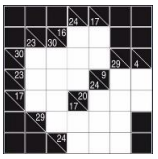
- We had three runs. In all three runs, a solution to the game was found.
- We assume that as the population grows, so does the complexity of the tasks that we can successfully solve.

Genetic Algorithms Running Time

Separately, we examined the running time of the implemented genetic algorithms. At the easy level of the Kakuro game, we compared the algorithm with the initial parameters (*GA light version*) and the algorithm with optimal parameters (*GA pro version*) obtained during the experiments:

```
Time GA light version=5 sec.
Time GA pro version=59 sec.
Difference between=54 sec.
GA light version faster than GA pro version in 12
```

For small Kakuro playing fields, the GA light version is optimal. On the contrary, the GA pro version is recommended to find a solution in large playing fields. It is worth considering that the GA pro version works 12 times slower.



Genetic Programming

At this stage of our project, we used genetic programming to solve the problem of the Kakuro game presented above.
Genetic programming is focused on solving problems of automatic synthesis of programs based on training data by inductive inference. Chromosomes or structures that are automatically generated by genetic operators are computer programs of various sizes and complexity.
Programs consist of functions, variables, and constants and have a tree form. The following are the steps for genetic programming.

Genetic Programming Procedure

1. Initialization

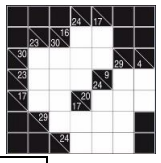
At this stage, we randomly generated a population consisting of μ tree programs, with the root node of the tree being a function whose arguments are randomly selected from the *function set* or *terminal set*. The trailing vertices of a tree are variables or constants. We have set at this stage the maximum height of the tree - 2.

Function set:

Function	Number of Input Terminals	Description
+	2	Addition Operator
-	2	Subtraction Operator
*	2	Multiplication Operator
%	2	Safe Division Operator
if_then_more_else	4	return x_3 if $x_1>x_2$, else x_4
vv	3	return x_3 if $x_1<x_2$, else 1
bb	6	if $x_1>x_2$ and if $x_3>x_4$ return x_5 elif $x_1<x_2$ and $x_3<x_4$ return x_6 else 1
nn	3	if $x_1>x_2$ return $x_1+x_2-x_3$ else return $x_1+x_2+x_3$
kk	3	return x_2 if $x_1>0$ else x_3
hh	5	return x_4 if $x_1+x_2<x_3$ else x_5

Terminal set:

Terminal Value	Description
AllCountGroupHor	number of all horizontal groups
AllCountGroupVer	number of all vertical groups



SumGroupHor	clue for the current horizontal group
SumGroupVer	clue for the current vertical group
CurrentlySumGroupHor	actual horizontal group sum at the moment
CurrentlySumGroupVer	actual vertical group sum at the moment
BalanceStepsHor	number of remaining white cells in the horizontal group
BalanceStepsVer	number of remaining white cells in the vertical group
DubbleHor	number of duplicate elements in a horizontal group
DubbleVer	number of duplicate elements in a vertical group
BalanceStepsNextGroupHorHor	number of white cells in the horizontal in the next horizontal group
BalanceStepsNextGroupHorVer	number of white cells in the vertical in the next horizontal group
BalanceStepsNextGroupVerHor	number of white cells in the horizontal in the next vertical group
BalanceStepsNextGroupVerVer	number of white cells in the vertical in the next vertical group
SumNextGroupHorHor	horizontal clue in the next horizontal group
SumNextGroupHorVer	vertical clue in the next horizontal group
SumNextGroupVerHor	horizontal clue in the next vertical group
SumNextGroupVerVer	vertical clue in the next vertical group
CurrentlySumNextGroupHorHor	actual sum in the horizontal in the next horizontal group at the moment
CurrentlySumNextGroupHorVer	actual sum in the vertical in the next horizontal group at the moment
CurrentlySumNextGroupVerHor	actual sum in the horizontal in the next vertical group at the moment
CurrentlySumNextGroupVerVer	actual sum in the vertical in the next vertical group at the moment
1,2,3,4,5,6,7,8,9	Constants used to adjust weight

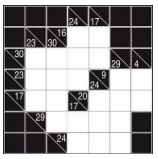
2. Fitness Evaluation

- The next step is to evaluate the candidates. To evaluate the fitness parameter, we created a function (*evaluate()*). The function takes a tree, compiles it, and using the function *filling_desk()*, the playing field is filled. After that, the playing field is subject to fitness evaluation.
- We presented the fitness function as the sum of the number of duplicate elements in each group and the difference of each group between the sum and the clue:

$$F = 10*(F1 + F2) + F3 + F4,$$

where:

- F_1 - the number of duplicate values in horizontal groups,



- F_2 - the number of duplicate values in vertical groups,
- F_3 – the difference between the sum of elements and the clue in horizontal groups,
- F_4 - the difference between the sum of elements and the clue in vertical groups.

- We added a small penalty in order to minimize the number of solutions with duplicate values.
- Thus, only fitness = 0 means that we set the numbers correctly, that is, we solved the Kakuro puzzle.

3. Parent selection

The selection operator selects representatives for the use of genetic operators. For selection, we use *Tournament selection*. Selected candidates are then passed on to the next generation. In a 7-way tournament selection, we select 7 individuals and run a tournament among them. Only the fittest candidate amongst those selected candidates is chosen and is passed on to the next generation. In this way many such tournaments take place and we have our final selection of candidates who move on to the next generation.

4. Crossover

We use *one-point crossover*, in this crossover, a random crossover point in each individual is selected and and exchanged each subtree with the point as root between each individual. Resulting in two new representatives, with the genes of the first and second parent.

5. Mutation

The next step is the mutation operator (*mutUniform*). A point in the individual is randomly selected, then the subtree at that point as a root is replaced by the generated expression. Generation of expression also occurs as individuals in the first generation and with the same restrictions on the height of the tree.

6. Final

The GP procedure is iterative. The final stage of each iteration is the formation of a new population. Stop occurs on the *fiftieth generation*.

Experiments, Evolutionary Process and Results (GP)

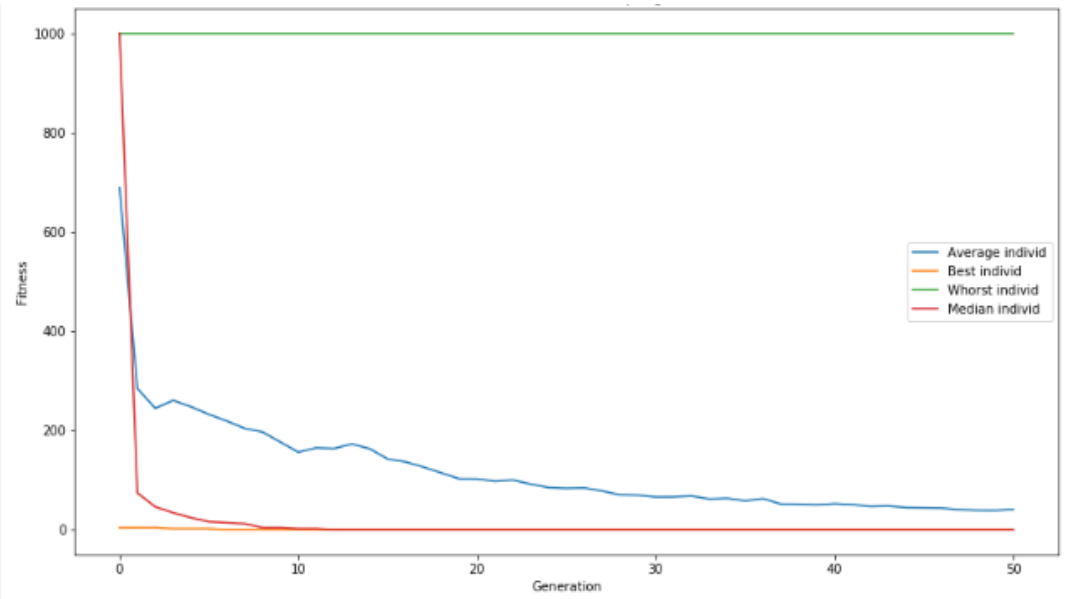
As the initial data of our experiment, we selected the standard parameters for selection, crossover and mutation, which are presented in the table:

Representation	Tree structures
Recombination	Single point crossover (exchange of subtrees)
Recombination probability	70%
Mutation	mutUniform (random subtree replaced by the expression)
Mutation probability	10%
Parent selection	Tournament (7 individuals)
Population size	5000 individuals
Initialization	random
Termination condition	50 generations

We used an infinite loop to run implemented genetic programming. The condition for the exit of their cycle is to find the optimal solution. We have chosen a breakpoint equal to 50 generations to save time.

1. Easy level

The attempt to solve the playing field - Easy level:



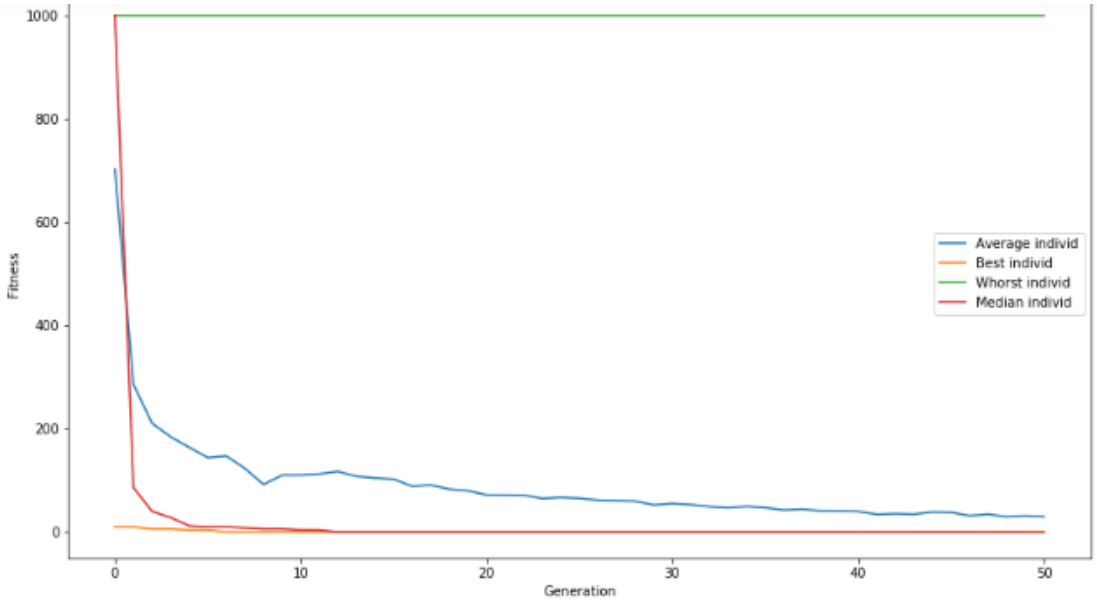

```
|XXXXX|| 0/17|| 0/16|
|17/ 0||   8 ||   9 |
|16/ 0||   9 ||   7 |

A solution was found in an attemp: 1

Fitness: 0

Optimal strategy:
 nn(sub(8, BalanceStepsNextGroupHorVer), kk(if_then_more_else(if_then_more_else(2, SumNextGroupVerHor, BalanceStepsHor, Balance
StepsNextGroupVerHor), protectedDiv(SumNextGroupHorVer, AllCountGroupHor), if_then_more_else(BalanceStepsHor, 7, CurrentlySumNe
xtGroupHorVer, SumNextGroupVerVer), kk(BalanceStepsNextGroupVerVer, BalanceStepsNextGroupHorHor, SumGroupHor)), 2, SumNextGroup
VerVer), bb(DubbleVer, BalanceStepsHor, CurrentlySumGroupHor, 8, CurrentlySumNextGroupVerVer, DubbleVer))

Algorithms time 216.29183673858643
```

- We found a solution to the Kakuro puzzle from the first attempt.
- The optimal solution was found in the initial generations:

```
nn(sub(8, BalanceStepsNextGroupHorVer), kk(if_then_more_else(if_
then_more_else(2, SumNextGroupVerHor, BalanceStepsHor, BalanceSt
epsNextGroupVerHor), protectedDiv(SumNextGroupHorVer, AllCountGr
oupHor), if_then_more_else(BalanceStepsHor, 7, CurrentlySumNextG
roupHorVer, SumNextGroupVerVer), kk(BalanceStepsNextGroupVerVer,
BalanceStepsNextGroupHorHor, SumGroupHor)), 2, SumNextGroupVerVe
r), bb(DubbleVer, BalanceStepsHor, CurrentlySumGroupHor, 8, Curr
entlySumNextGroupVerVer, DubbleVer))
```

2. Medium level

The attempt to solve the playing field - Medium level:



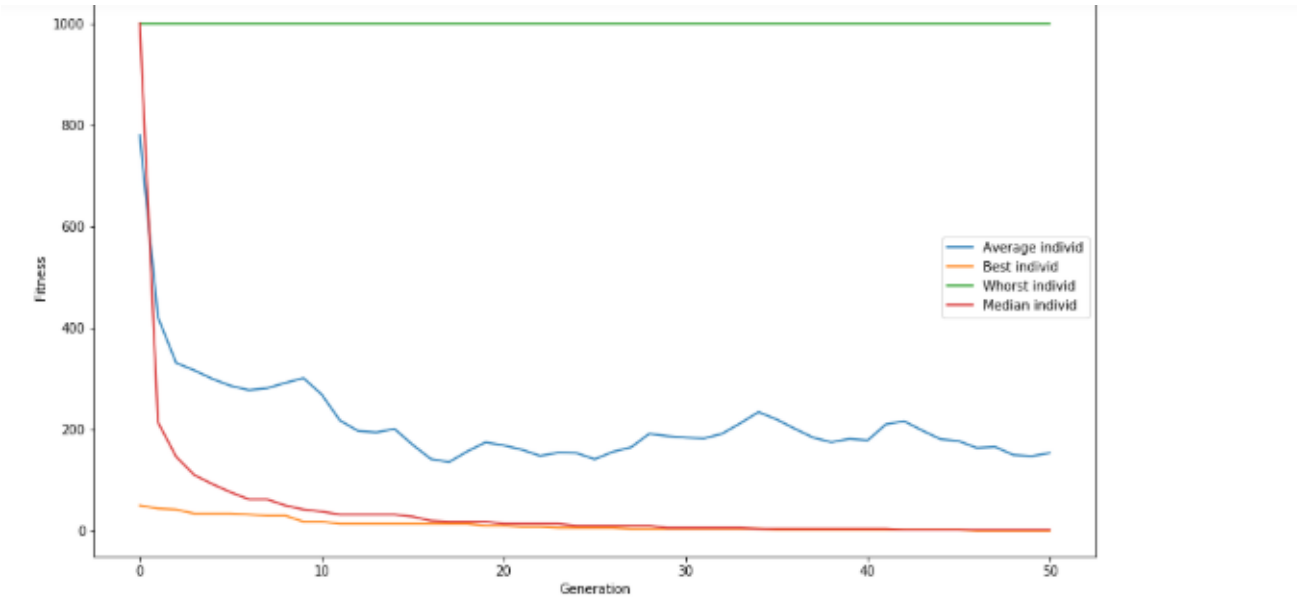

```
|XXXXX|| 0/ 3|| 0/ 9||XXXXX|
| 4/ 0||   1 ||   3 || 0/ 3|
| 8/ 0||   2 ||   5 ||   1 |
|XXXXX|| 3/ 0||   1 ||   2 |

A solution was found in an attemp: 1

Fitness: 0

Optimal strategy:
 protectedDiv(nn(bb(vv(BalanceStepsNextGroupHorHor, BalanceStepsNextGroupVerVer, 4), vv(SumNextGroupHorVer, SumGroupHor, Curren
tlySumNextGroupHorVer), mul(BalanceStepsHor, BalanceStepsNextGroupVerVer), kk(BalanceStepsVer, 3, 4), bb(SumNextGroupVerVer, Cu
rrentlySumNextGroupHorHor, CurrentlySumNextGroupVerHor, 1, SumGroupHor, DubbleHor), hh(BalanceStepsNextGroupVerVer, 1, Currentl
ySumNextGroupHorHor, 3, SumGroupVer)), hh(DubbleHor, SumGroupVer, 5, CurrentlySumGroupVer, 2), CurrentlySumNextGroupHorHor), pr
otectedDiv(2, BalanceStepsNextGroupHorVer))

Algorithms time 297.94906735420227
```

- We found a solution to the Kakuro puzzle from the first attempt.
- The optimal solution was found in the initial generations:

protectedDiv(nn(bb(vv(BalanceStepsNextGroupHorHor, BalanceStepsNextGroupVerVer, 4), v
v(SumNextGroupHorVer, SumGroupHor, CurrentlySumNextGroupHorVer), mul(BalanceSteps
Hor, BalanceStepsNextGroupVerVer), kk(BalanceStepsVer, 3, 4), bb(SumNextGroupVerVer, C
urrentlySumNextGroupHorHor, CurrentlySumNextGroupVerHor, 1, SumGroupHor, DubbleHo
r), hh(BalanceStepsNextGroupVerVer, 1, CurrentlySumNextGroupHorHor, 3, SumGroupVer)),
hh(DubbleHor, SumGroupVer, 5, CurrentlySumGroupVer, 2), CurrentlySumNextGroupHorHor
), protectedDiv(2, BalanceStepsNextGroupHorVer))

3. Hard level

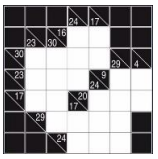
The attempt to solve the playing field - Hard level:




```
|XXXXX|| 0/ 7|| 0/23||XXXXX||XXXXX|
|10/ 0||   2 ||   8 || 0/24|| 0/ 7|
|16/ 0||   1 ||   6 ||   7 ||   2 |
|22/ 0||   4 ||   9 ||   8 ||   1 |
|XXXXX||XXXXX||13/ 0||   9 ||   4 |

A solution was found in an attemp: 107

Fitness: 0

Optimal strategy:
 nn(protectedDiv(vv(9, SumGroupVer, hh(CurrentlySumGroupVer, BalanceStepsNextGroupHorHor, mul(BalanceStepsNextGroupHorVer, Bala
nceStepsNextGroupHorHor), protectedDiv(BalanceStepsNextGroupHorHor, 9), add(8, 7))), nn(if_then_more_else(if_then_more_else(Bal
anceStepsVer, 2, protectedDiv(protectedDiv(CurrentlySumNextGroupHorHor, BalanceStepsNextGroupHorHor), CurrentlySumGroupVer),
3), SumNextGroupVerHor, BalanceStepsNextGroupHorHor, 8), CurrentlySumGroupVer, DubbleHor)), if_then_more_else(BalanceStepsVer,
1, protectedDiv(protectedDiv(CurrentlySumNextGroupHorHor, BalanceStepsNextGroupHorHor), CurrentlySumGroupVer), 3), vv(9, SumGro
upVer, 6))

Algorithms time 353.2214467525482
```

- We found a solution using the 107th attempt.
- The hard level of the Kakuro puzzle required a significantly larger number of attempts to find a solution by comparison with simpler levels of complexity, and, accordingly, the ru nning time of the algorithm was longer.
- The optimal solution was found in the initial generations:

```
nn(protectedDiv(vv(9, SumGroupVer, hh(CurrentlySumGroupVer, Bala
nceStepsNextGroupHorHor, mul(BalanceStepsNextGroupHorVer, Balanc
eStepsNextGroupHorHor), protectedDiv(BalanceStepsNextGroupHorHor
, 9), add(8, 7))), nn(if_then_more_else(if_then_more_else(Balanc
eStepsVer, 2, protectedDiv(protectedDiv(CurrentlySumNextGroupHor
Hor, BalanceStepsNextGroupHorHor), CurrentlySumGroupVer), 3), Su
mNextGroupVerHor, BalanceStepsNextGroupHorHor, 8), CurrentlySumG
roupVer, DubbleHor)), if_then_more_else(BalanceStepsVer, 1, prot
ectedDiv(protectedDiv(CurrentlySumNextGroupHorHor, BalanceStepsN
extGroupHorHor), CurrentlySumGroupVer), 3), vv(9, SumGroupVer, 6
))
```



Software Overview

GA & GP - Main methods:

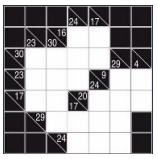
1) **Start() –** Levels menu. The user can choose the level of the board at the start of the game- easy, medium, hard or expert.
2) **ga()/gp_al() –** The whole process is happening from this function. The initialization of all GA parameters (individual, fitness function, population, number of generations, mutation and selection methods), and the calculation itself. This function returns the runtime of the process, statistics (average, median, worst and best) and the "behavior" of the individuals over generations.
3) **Graph() -** Plotting the statistics over generations.
4) **Evaluation(individual) –** Calculates the fitness function (in a way that was mentioned before).
5) **check_sum_hor(matrix)/check_sum_vert(matrix) –** These functions check the difference in sum between the required to the current sum over rows and columns in the board (it happenes inside the fitness function).
6) **check_double_hor(matrix)/check_double_vert(matrix) –** These functions check if there are duplicate values in rows and columns.

GA – Unique methods:

1) **Mut_k_point(individual, k) –** Mutation function that changes a random digit in k individuals.
2) **Filling_desc(array) –** this function fills the board with the genome values (numbers from 1 to 9).

GP – Unique methods:

1) **Filling_desc(gp_func) –** This function fills the board with values from the tree.
2) **ps() –** This function generates the tree with all the nodes and terminals.



Conclusions

- In the case of GA, we saw that the success of solving the Kakuro problem directly depends on the size of the population, namely, with the growth of the population, the decisive ability of the algorithm increases.
- On the other hand, as the population increases, the running time of the algorithm increases in direct proportion.
- When solving the Kakuro problem, the complexity of the playing field should be determined in advance to maintain a balance between the population size and the algorithm time.
- When using GA to solve Kakuro problems, it is necessary that the correct solution be initialized in the initial generations (up to the 20th generation).
- This conclusion is due to the fact that minimizing the Fitness function, in case of incorrect initial initialization, takes us farther and farther from the optimal solution with each subsequent generation.
- In the case of GP, we saw that finding a solution takes a lot of time and depends on the size of the playing field.
- The functions and terminals that we have chosen do not provide us with an optimal strategy when it comes to really large fields.
- GA and GP are great ways to automatically solve the Kakuro puzzle.
- In both GA and GP, a noticeable decrease in speed occurred with an increase in grid size. Such a decrease in speed with an increase in grid size limits the usefulness of Kakuro in real applications that require display on very large grids.

Future Work

- In future work, we propose to see how our implemented GA pro version (with optimal parameters) manifests itself on more complex grids.
- As a continuation of this work, it is possible to improve the GP by adding new terminals and functions and also think about reducing the speed of work on large grids for use in real applications.

Bibliography

1. https://en.wikipedia.org/wiki/Kakuro
2. An Investigation into the Solution to, and Evaluation of, Kakuro Puzzles, Davies, Ryan P, 2009
3. Automation of the Solution of Kakuro Puzzles, R. P. Davies, P. A. Roach, S. Perkins
4. Solving NP-hard number matrix games with Wisdom of Artificial Crowds, J. Schreiver ; C. Shrum ; A. Lauf ; R. Yampolskiy, 2015