# Genetic Algorithm Scheduling Problem

The project is written in the Python programming language and consist of the follow files:

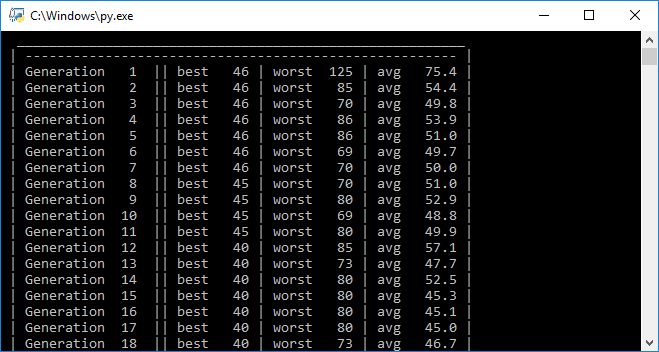
* **parameters.py** defines variables for the algorithm and data for the task, such as population size, mutation rate, crossover rate, task durations and others,
* **genome.py** has the definition of the Genome class, which is a genotype representation of a individual in terms of the algorithm (phenotype being a schedule proposal),
* **operators.py** defines genetic operators (selection, crossover, mutation) as functions,
* **algorithm.py** defines a function that creates an initial population and then iterate through the defined number of generations, applying the appropriate genetic operators; for each generation, outputs the best, worst and average genome stats to the console,
* **main.py** the entry point of the project which decorates the output of algorithm.py.

Genetic algorithm is a heuristic which is interesting in that it allows us to see a solution being born. Generally, he steps are the following: populate the initial generation with random genomes, then iterate over and over making new generations based on applying the genetic operators (**selection** of parents from a population, **crossover** of those parents, and finally **mutation** to enhance diversity).

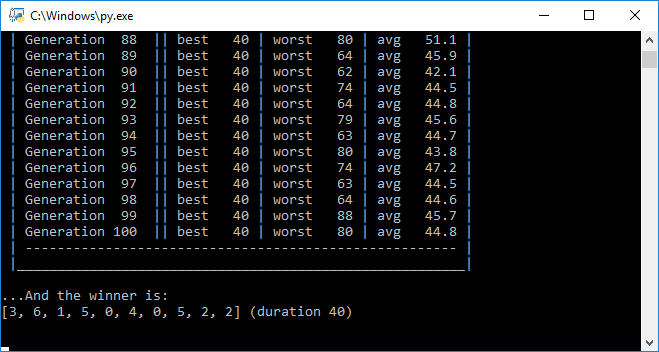
All the operators depend on certain sets of parameters, some of which can be set explicitly in the **parameters.py** source file; those are the following:

* task\_durations — a Python list (enclosed within a pair of brackets, comma-separated) of integer numbers representing time (in agreed units — does not matter within the algorithm as long as we use them consistently) required for a corresponding task to be completed; for example, a list  
  [20, 35, 12] means that 0th task needs 20 time units, 1st — 35, 2nd — 12,
* task\_count — number of tasks; calculated automatically from the length of task\_durations,
* resource\_count — number of available resources; the resources will be referred to by indices from 0 to N-1 (N being the value of this parameter) in the algorithm, *high values can take longer time for the program to complete calculations*,
* population\_size — number of individuals (genomes) in the population; must be even because of the parity of the offspring count (two parents make two offsprings), *high values can take longer time for the program to complete calculations*,
* generation\_limit — a stop condition expressed in the number of iteration (i.e. number of generations) of the algorithm before a final solution is considered to be found, *high values can take longer time for the program to complete calculations*,
* mutation\_rate — probability of a single gene mutation (from 0.0 to 1.0),
* crossover\_rate — probability of two parents’ genome to be “split-and-combined” before passing genetic info to the offsprings (from 0.0 to 1.0),
* elitism\_on — whether elitism is enabled or not.

To start the program, run “python main.py” (Python 3.x required).



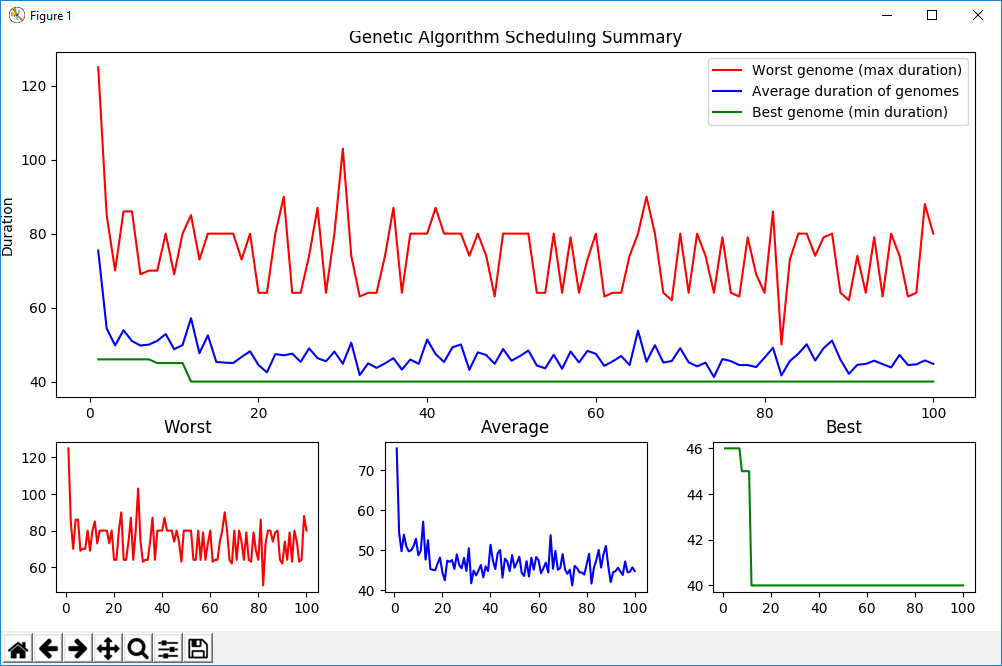
The beginning of the iteration output.



The end of the per-generation output, and the final result.

The information gathered at each generation is presented in the format of a log so that the user can monitor how good the algorithm is doing its job, what are the dynamics of change of the best and worst genomes, and what is the mean duration of current generation’s genomes.

Shortly after the CLI output is done printing, plots occur with visualization of the data presented in this plain text format (see below). The plots are done using the matplotlib Python library.



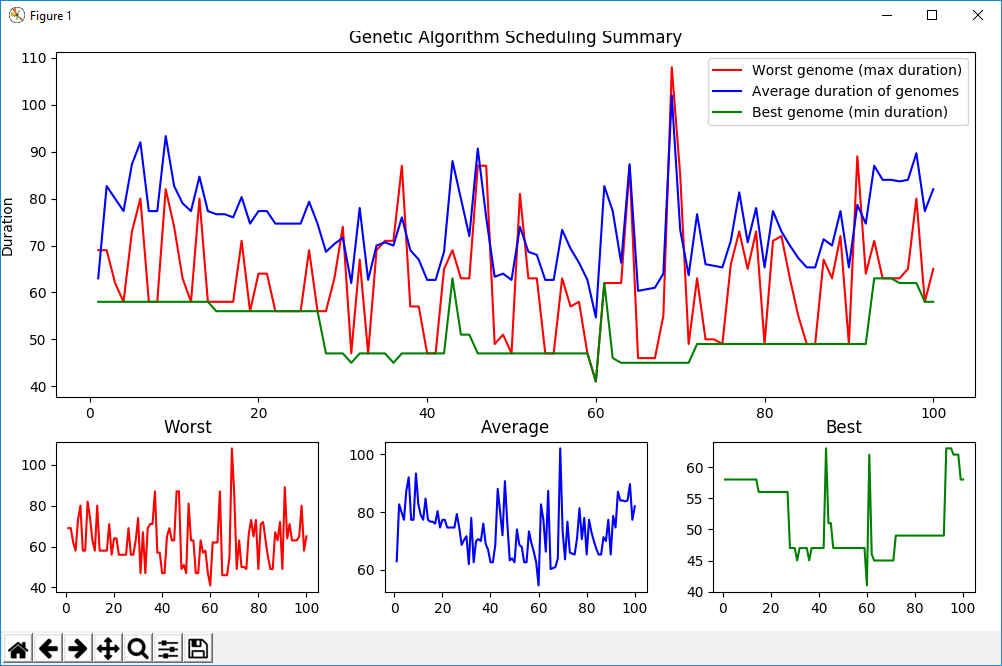
In most of the cases (with big enough population size, low enough mutation rate etc. — we’ll discuss those later), it can be seen that during the first few generations, worst genome’s duration is decreased drastically. This is a natural thing, because out initial population is completely random, unlike all the following ones which are results of the GA working.

Some correlation between worst and average durations can also be noticed. The unstable amplitude of those two indicates a fair diversity of the researched genomes and prevents the algorithm to be stuck at a local optimum.

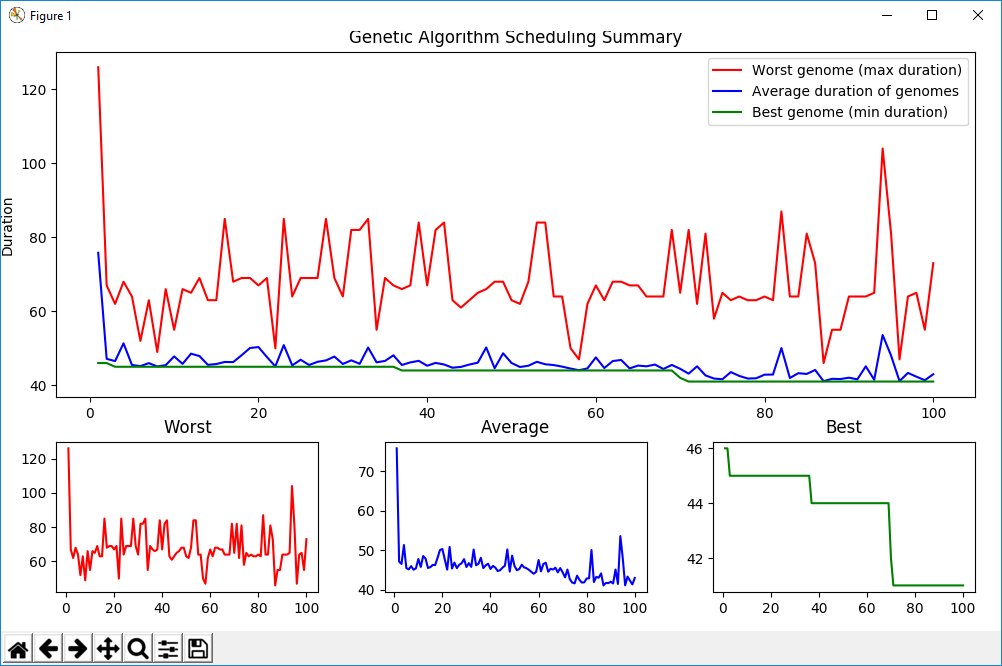
For input data not very complicated, the best duration can be found rather quickly in the beginning. With elitism on, the two best duration genomes persist across generations, but even with elitism off, best genomes can still make into the next generation with a high likelihood.

Observations and repeated execution of the algorithm made me able to do the following conclusions:

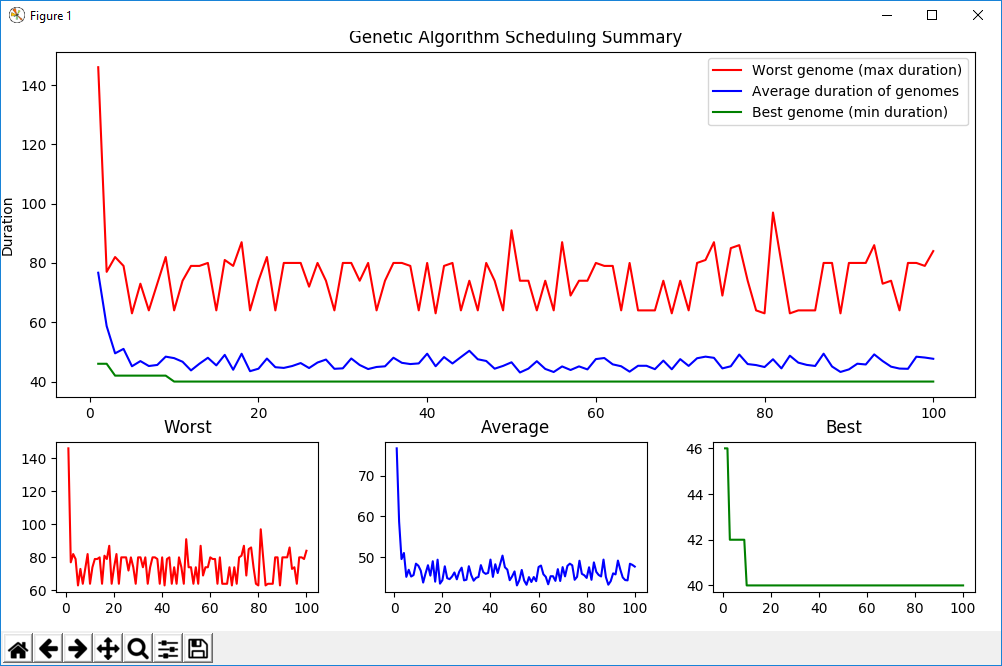
1. The population size is extremely crucial. Below is the plot of the GA in work with population size set to 3. The dynamics of the green line clearly implies the algorithm is malfunctioning. The reason is not enough initial diversity (i.e. not enough genomes to be randomly created in the first generation).



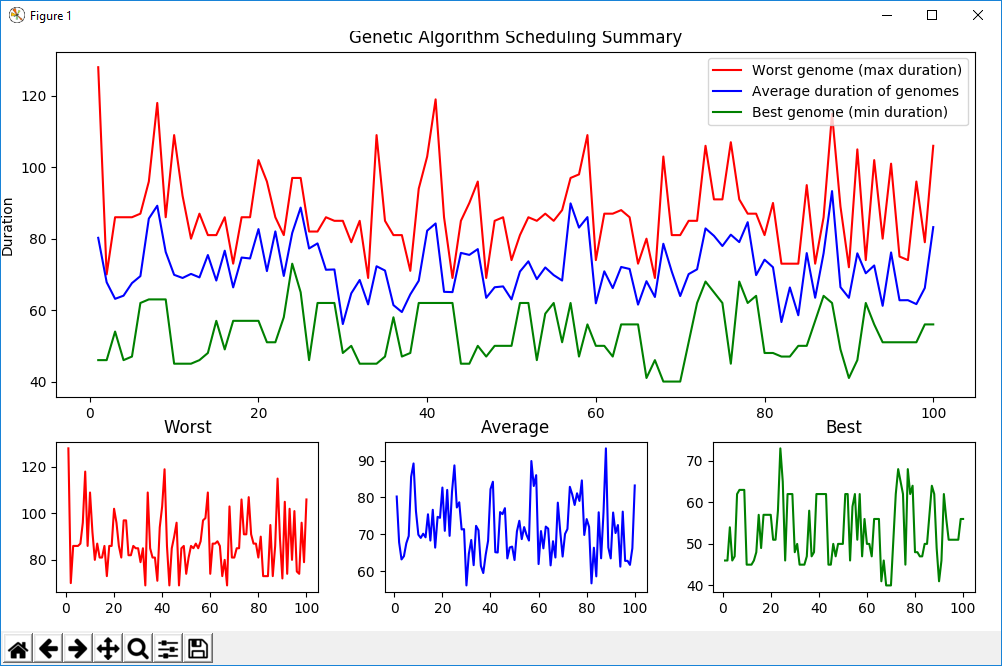
1. Mutation rate also takes huge take in diversifying the population. Below is the plot of GA with parameter of mutation rate set to 0.0, i.e. no mutation at all. It takes much more generations to come up with a new best genome.

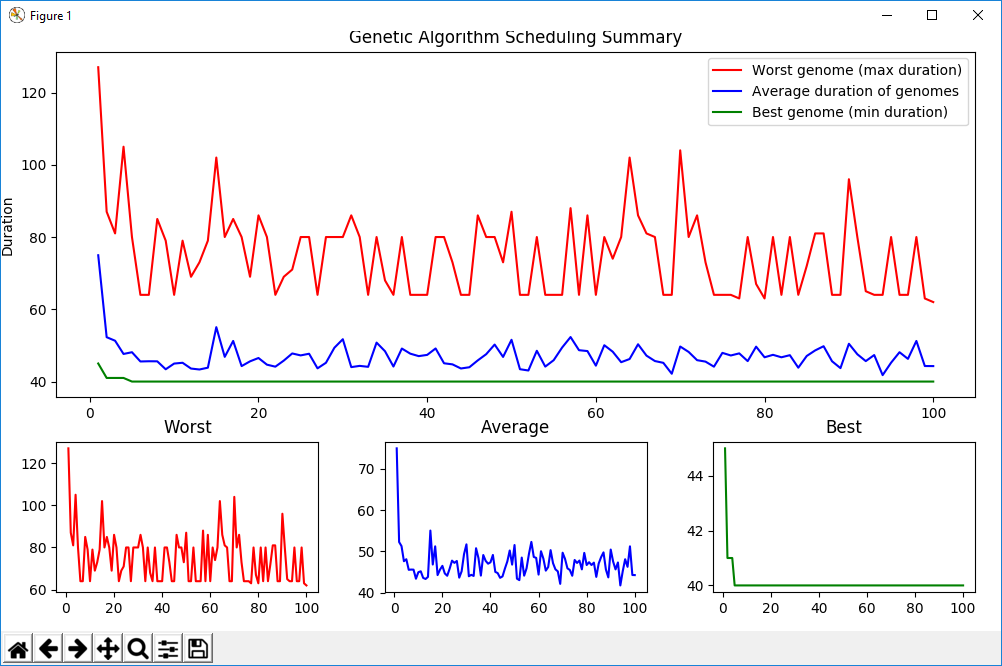


1. Elitism is a good thing to have, because it keeps track of the current two best genomes and moves them to the next generation, so that they are not “forgotten”. It’s good in the sense that if we ever come up with a better solution, it’ll immediately become a new best; but if we lost the one that was once the best, we might never achieve that good a duration again. Below is the plot with elitism on.

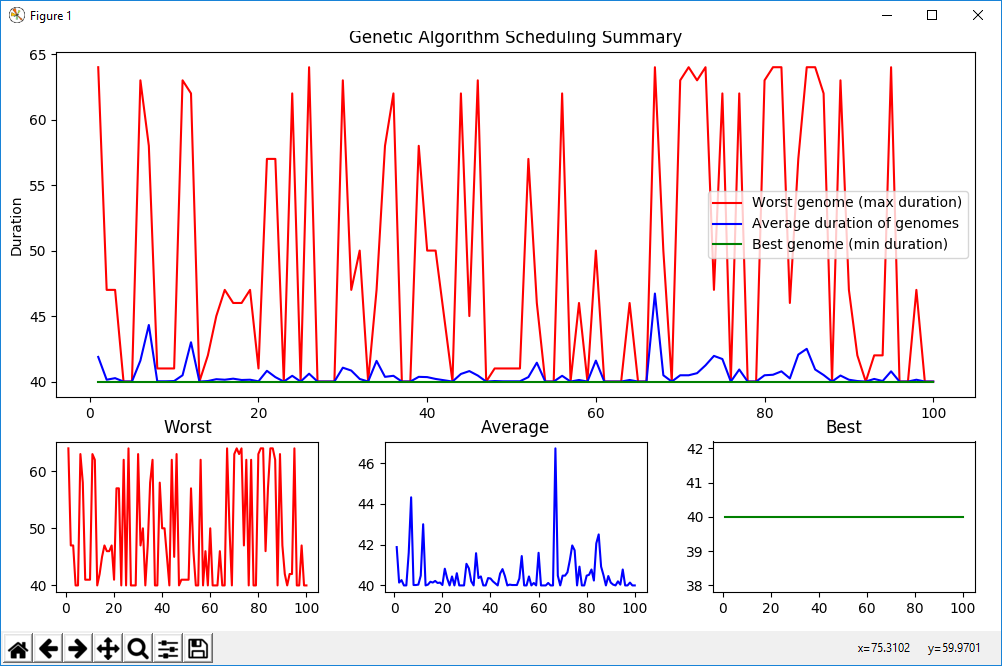


1. Crossover should happen as often as optimally possible, because the insufficient crossover rate might result in dull semi-random outcomes. Compare crossover rate 0.1 and 0.8 below.

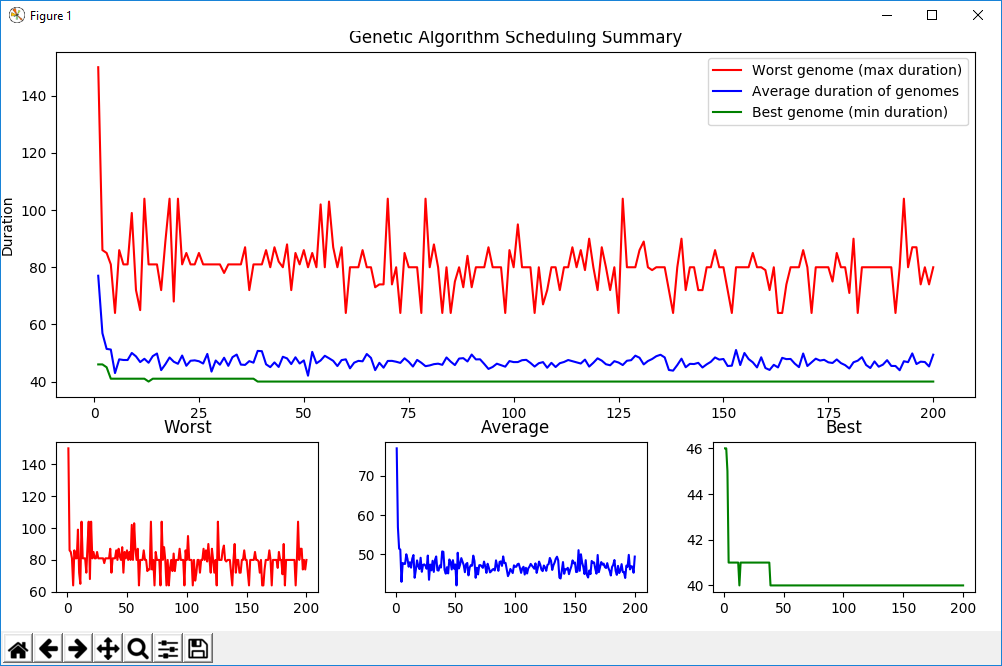




1. Huge resource count (e.g. 100 for only 10 tasks) make the algorithm close to useless if only time is concerned. The best solution is most likely to be found in the first generations, and the worst/average stats would only vary due to the algorithm trying to shuffle and mix the schedules not that optimal. Notice the “Best” plot dynamics (or rather its absence).



1. Sometimes an optimal solution can be yet to come, so we need to take care of generation limit big enough to let the algorithm find and stick to the best solution for sure. Below you can see a “gap” in the best genome’s plot which indicates that sometimes best duration may go both ways.



1. Qwertyuiop
2. Qwertyu