POPULAR DEMOCRATIC REPUBLIC OF ALGERIA
HIGHER EDUCATION AND SCIENTIFIC RESEARCH'S MINISTRY
KASDI MERBAH OUARGLA UNIVERSITY

Faculty of new information technologies and communication

Department of computer and information technology

LMD Licence 2019/2020

Dissertation
For the License degree

Presented by: Houari Hocine Abdellatif

Title:

Genetic Algorithms Simulator

Supervised by: Dr. Belkebir Djalila

Academic Year 2019-2020

Thanks and Dedication

For every one who reads this report

For every one who uses the simulator

Table of Contents

ABSTRACT

CHAPTER I : GENETIC ALGORITHM PRINCIPLES

CHAPTER II : GA SIMULATOR DESCRIPTION AND IMPLEMENTATION

ABSTRACT

Abstract

Humans faced many challenges over the centuries, but with science revolution and the raise of knowledge, new types of problems appeared that caught researchers especially in mathematics, which are considered hard problems. Hard mathematical problems are much known that they require tremendous amount of computation because of their complexity and the size of the search space or number of possibilities, and because of that, researchers shifted their focus during the past decades to optimization, where it is not practical to look for the best solution by trying every possibility, but it is more feasible to hold a good set of possible solutions and optimize upon them in a repetitive process. This is where we provide a simulator which uses genetic algorithms in the same manner to relativaly solve those mathematical problems on considerably less amount of time.

Keywords: hard mathematical problems, optimization, simulator, genetic algorithms, search space.

عبر قرون مضت واجهت البشرية الكثير من التحديات، لكن مع التطور العلمي و نمو المعرفة، نوع جديد من المشكلات لفت اهتمام العلماء خاصة في مجال الرياضيات، والتي اعتبرت مشكلات صعبة. عرفت المشكلات الرياضية الصعبة بأنها تتطلب كمية هائلة من العتاد الحسابي نظرا لتعقيدها و حجم فضاء البحث أو عدد الاحتمالات الممكنة للحل، ولهذا، توجهت أنظار الباحثين نحو التحسين، بمعنى ليس عمليا محاولة إيجاد أحسن حل ممكن دفعة واحدة من خلال تجربة كل احتمال ممكن، ولكن من المعقول امتلاك مجموعة ذات جودة حسنة من الحلول الممكنة، ومن ثم ايجاد طريقة لتحسين باستعمال هذه الحلول وتكرير الاجراء مرات عدة. هنا يأتي دور المحاكي الذي يستعمل الخوارزميات الجينية بنفس الطريقة المذكورة سابقا لإيجاد حلول نسبية لهذه المشكلات الرياضية في وقت قصير مقارنة بالطريقة الأسبق.

الكلمات المفتاحية: المشكلات الرياضيات الصعبة، التحسين، المحاكي، الخورزميات الجينية، فضاء البحث.

CHAPTER I:

GENETIC ALGORITHM PRINCIPLES

CHAPTER I

Genetic Algorithm Principles

1. *Introduction*

Optimization problems are concerned with finding the values for one or several variables that meet the best objectives without violating the constraints if it is considered as hard, and allows small violating if it is considered as soft. But before defining the optimization methods needed to solve the given problems, it might be helpful to find a classification for its size and its complexity.

The computational complexity of an optimization problem as well as optimization algorithms is given by $O(\bullet)$ which indicates the asymptotic time needed to solve the problem when it involves n variables and the problem size is determined by the number of these variables and the complexity becomes $O(n^2)$ and is therefore quadratic in n. A real improvement of the complexity can only be achieved when a better algorithm is found. This applies not only for the polynomial problems where the computation increases factorially with the system size. This chapter is dedicated to defining the genetic algorithms that is used by our simulator GeneticPy, also its main principles that we offered to the users.

2. *Heuristic optimization*

2.1 Preliminaries

Optimization is an intelligent selecting of the best alternative among a given set of options or in other way, it can be viewed as a decision making in order to optimize one or several objectives according to the given scenarios. Optimization problems arise in various disciplines such as engineering design, manufacturing system, economics etc. Thus, in the view of the practical utility of optimization problems there is a need for efficient and robust computational algorithms that can numerically solve with computers the mathematical models of medium as well as large size optimization problems arising in different fields. In this section, we will define some preliminaries related with the heuristic optimization [1].

A. *Objective function*

The objective function is in general in the form of an equation that need to be optimized and it is given based on specific constraints and based on the variables whose need to be minimized or maximized using nonlinear programming techniques.

B. *Search space*

The search space is in the form of a range or values, those values represent the variables or the space of all solutions that will be searched during optimization that are called search space.

C. Types of solutions

A solution to an optimization problem is the results of the objective function, it can be a feasible solution that satisfies all constraints or in other means; each point in the search space represents one feasible solution. Also, it can be an optimal solution that represents a feasible solution that provides the best objective function value. And the last one is the near-optimal solution that is also a feasible solution, but provides a superior objective function value, but not necessarily the best.

D. Constraints

Each constraint can be hard that is must be satisfied or it can be soft that is optimized.

E. Heuristics

Heuristics are rules of searching or finding optimal or near-optimal solutions. Examples are FIFO, LIFO, earliest due date first, largest processing time first, shortest distance first, etc. Heuristics can be constructive while a solution is built piece by piece, or it can improvement where a better solution is searched based on primary used solution.

2.2 Characteristics of heuristic optimization methods

The main common characteristics of all heuristic optimization methods is that, they start with an initial solution, and iteratively produce new solutions by some generation rules and evaluate those new solutions, and in the final, we report the best solution found during the search process. The execution of the iterated search procedure is usually stopped or achieved in case of:

- No further improvement over a given number of iterations.
- When the required solution has been found.
- When the allowed CPU time (or other external limit) has been reached.

Heuristic optimization methods may differ substantially in their underlying concepts, the following is given the mains over them [2]:

A. Generation of new solutions

A new solution can be generated by modifying the current solution or by building a new solution based on past experience or results. In doing so, a deterministic rule, a random guess or a combination of both can be employed.

B. Treatment of new solutions

In order to overcome local optima, heuristic optimization methods usually consider not only those new solutions that lead to an immediate improvement, but also some of those that are knowingly inferior to the best solution found so far. To enforce convergence, inferior solutions might either be included only when not being too far from the known optimum.

C. Limitation of the search space

Some methods exclude certain neighborhoods or regions or individuals to avoid cyclic search paths or spending too much computation time on supposedly irrelevant alternatives.

D. Prior knowledge

The prior knowledge is a guideline of making a good solution by storing previous generation or solutions; it can be incorporated in the choice of the initial solutions or in the search process. Though the inclusion of prior knowledge might significantly reduce the search space and increase the convergence speed.

E. Computational complexity

For heuristic optimization methods, the complexity depends on the costs of evaluating each solution, the number of iterations, and the population size.

3. Genetic algorithms principles

Optimization problems can be of different type such as multi objective optimization, multimodal optimization and combinatorial optimization. In this session, we will detail the genetic algorithms which is a combinatorial optimization. First, we will give a short description of tabu search since it is included in our work, and in the next chapter, we will describe how we use it and how we integrated with GA.

Tabu search is a metaheuristic that guides a local search heuristic to escape from local minima and in the same time, to implement an exploration scheme. The simple tabu search algorithm applies an improving local search where at each iteration, the best solution among the list of neighborhoods is selected and remarked as a new current solution. A short-term memory is implemented as a tabu list where solution attributes are stored to avoid short term cycling. A term Aspiration criteria are defined which is allowed to include some unvisited solution of good quality which are excluded from allowed set. Starting from the basic search strategy [3].

3.1 History

Darwin's theory of evolution states that life is related and has descended from a common ancestor. The theory presumes that complex creatures have evolved from more simplistic ancestors naturally over time. In a nutshell, as random genetic mutations occur within an organism's genetic code, the beneficial mutations are preserved because they aid survival -- a process known as "natural selection." These beneficial mutations are passed on to the next generation. Over time, beneficial mutations accumulate and the result is an entirely different organism [4].

Genetic algorithms are adaptive heuristic search algorithm premised on the Darwin's evolutionary ideas of natural selection and genetic. The basic concept of genetic algorithms is designed to simulate processes in natural system necessary for evolution. As such, they represent an intelligent exploitation of a random search within a defined search space to solve a problem. First pioneered by John Holland in the 60's, GAs has been widely studied, experimented and applied in many fields in engineering world. Not only does genetic algorithm provide an alternative method to solving problem, it consistently outperforms other traditional methods in most of the problems. Many of the real-world problems which involve finding optimal parameters might prove difficult for traditional methods but are ideal for genetic algorithms [5].

3.2 Definition

Evolutionary algorithms (EAs) are population-based meta-heuristic optimization algorithms that use biology-inspired mechanisms and survival of the fittest theory in order to refine a set of solution iteratively [6].

Genetic algorithms are subclass of evolutionary algorithms where the individuals of the search space are binary strings or arrays of other elementary types. Genetic algorithms are computer based search techniques patterned after the genetic mechanisms of biological organisms that have adapted and flourished in changing highly competitive environment. The last decade has witnessed many exciting advances in the use of genetic algorithms to solve optimization problems in process control systems. Genetic algorithms are the solution for optimizing hard problems quickly, reliably and accurately. As the complexity of the real-time controller increases, the genetic algorithms applications have grown in more than equal measure, the pseudo code of genetic algorithm is defined in Algorithm 1.1.

One of the most fundamental principals in our world is the search for an optimal state. Optimization is the process of modifying the inputs or characteristics of a device, mathematical process to obtain minimum or maximum of the output. It is a primary tool, needed to tackle the unsolvable or hard problems. In the trial and error optimization, the processes affect the output without knowing about the constraints responsible to produce the output. A mathematical formula describes the objective function for optimization.

One dimensional optimization contains one variable and a problem having more than one variable requires multi-dimensional approach. As the number of dimensions increases, the process of optimization becomes difficult. Dynamic optimization is time dependent, whereas the static optimization is independent of time. The static problem is difficult to solve for finding the best solution, but the added dimension of time increases the challenge of solving dynamic problems. Discrete variable optimization contains only have a finite number of possible values, whereas continuous variables have an infinite number of possible values. Variables often have limits or constraints. Constrained optimization incorporates variable equalities and inequalities into the cost function, whereas unconstrained optimization allows the variable to take any value [7].

ALGORITHM 1.1. PSEUDO CODE FOR GENETIC ALGORITHM

```
Generate P random chromosome;
REPEAT
    Determine fitness of all chromosomes i=1..P;
    Determine probabilities Pi based on relative fitness;
    For number of reproduction;
        Randomly select two parents based on Pi;
        Generate two children by crossover operation on parents;
    END
    Insert offspring into the population;
    Apply mutation to all/some individuals;
UNTIL halting criterion is met;
```

3.3 Working principle of genetic algorithms

The workability of genetic algorithms is based on the Darwinian's theory of survival of the fittest. Genetic algorithms may contain a chromosome, a gene, a set of the population, fitness function, breeding, mutation and selection. Genetic algorithms begin with a set of solutions represented by chromosomes, called a population. Solutions from one population are taken and used to form a new population, which is motivated by the possibility that the new population will be better than the old one. Further, solutions are selected according to their fitness to form new solutions, that is, offsprings. The above process is repeated until some condition is satisfied. Algorithmically.

The genetic algorithms' performance is largely influenced by crossover and mutation operators. The processing done on those 2 steps can lead to offsprings similar looking to their parents or solutions that are far different. The block diagram representation of genetic algorithms is shown in Figure 1.1.

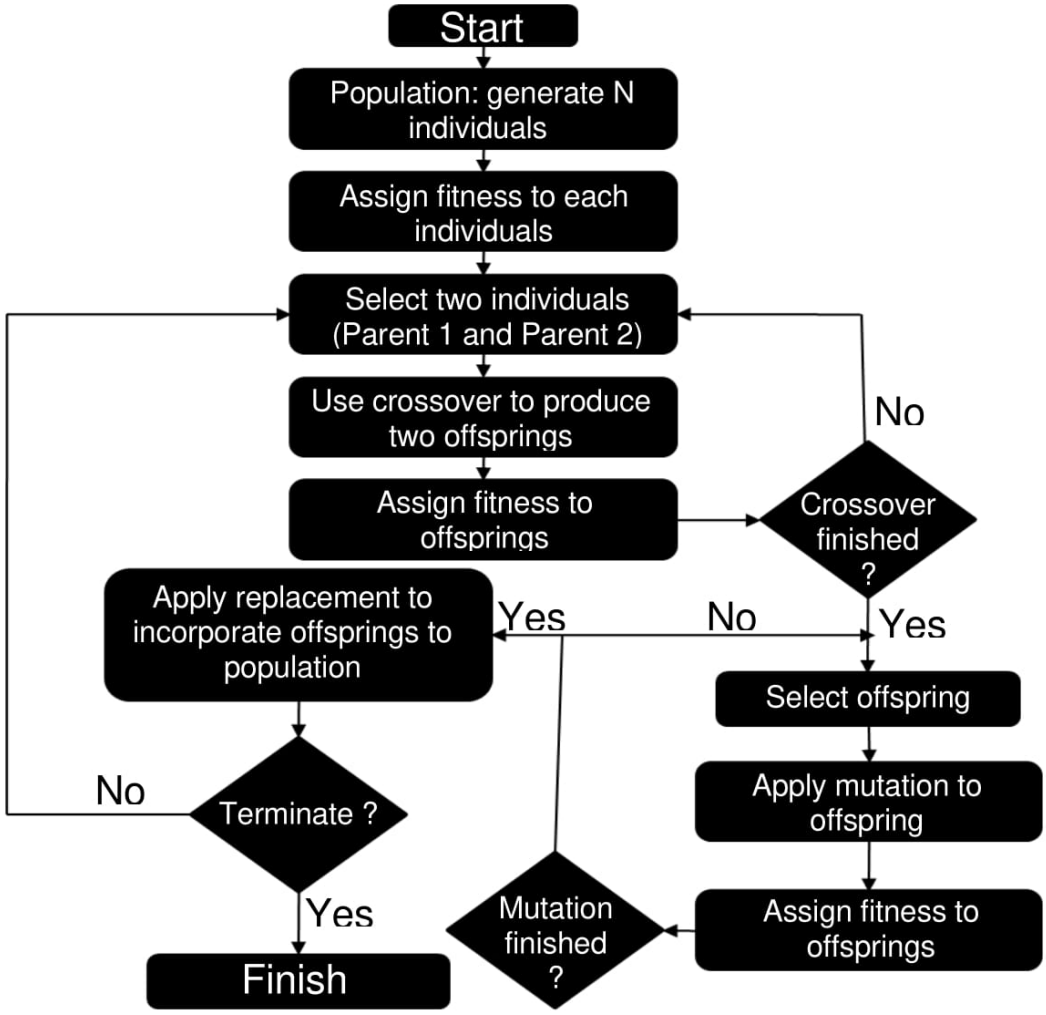


Figure 1.1. Block diagram representation of genetic algorithms.

3.3.1. Encoding technique in genetic algorithms

Encoding techniques in genetic algorithms are problem specific, which transforms the problem solution into chromosomes. Various encoding techniques used in genetic algorithms are binary encoding, permutation encoding, value encoding and tree encoding.

A. Binary encoding

It is the most common form of encoding in which the data value is converted into binary strings. Binary encoding gives many possible chromosomes with a small number of alleles. A chromosome is represented in binary encoding as shown in Figure 1.2.

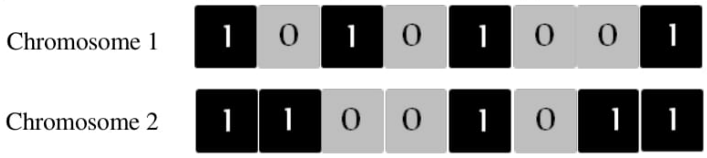


Figure 1.2. Binary encoding.

B. Permutation encoding

Permutation encoding is best suited for ordering or queuing problems. Travelling salesman is a challenging problem in optimization, where permutation encoding is used. In permutation encoding, every chromosome is a string of numbers in a sequence as shown in Figure 1.3.

Figure 1.3. Permutation encoding.

C. Value encoding

Value encoding can be form number, real number of characters to some complicated objects. Value encoding is a technique in which every chromosome is a string of some values and is used where some more complicated values are required. It can be expressed as shown in Figure 1.4.

Figure 1.4. Value encoding.

D. Tree encoding

It is best suited technique for evolving expressions or programs, such as genetic programming. In tree encoding, every chromosome is a tree of some objects, functions or commands in programming languages.

Locator/identifier separation protocol (LISP) programming language is used for this purpose.LISP programs can be represented in a tree structure for crossover and mutation. In the tree encoding, the chromosomes are represented as shown in Figure 1.5.

There are no specific directions for using the type of encoding scheme in the specified problem rather; it depends upon the applicability and the requirements of the problem.

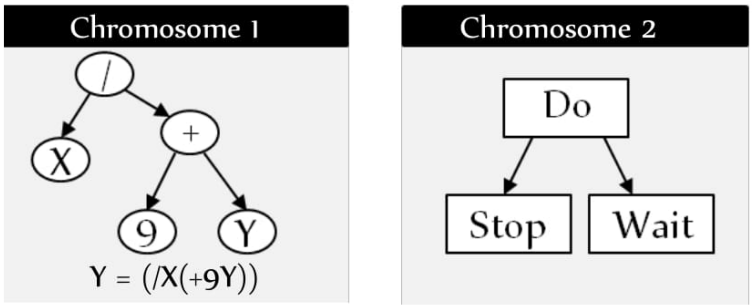


Figure 1.5. Tree Encoding.

3.3.2. Selection techniques in genetic algorithms (GAS)

Selection is an important function in genetic algorithms, based on an evaluation criterion that returns a measurement of worth for any chromosome in the context of the problem. It is the stage of genetic algorithm in which individual genomes are chosen from the string of chromosomes. The commonly used techniques for selection of chromosomes are Roulette wheel, rank selection and steady state selection.

A. Roulette wheel selection

In this method the parents are selected according to their fitness. Better chromosomes, are having more chances to be selected as parents. It is the most common method for implementing fitness proportionate selection. Each individual is assigned a slice of the circular roulette wheel, and the size of the slice is proportional to the individual fitness of chromosomes, that is, bigger the value, larger the size of slice is. The functioning of the roulette wheel algorithm is described below:

Step 1 [Sum] Find the sum of all chromosomes' fitness in the population.

Step 2 [Select] Generate random number from the sum interval.

Step 3 [Loop] Go through the entire population and sum the fitness. When this sum is more than a fitness criteria value, stop and return this chromosome.

Figure 1.6 shows the roulette wheel for six individuals having different fitness values. The Sixth individual has a higher fitness than any other, it is expected that the Roulette wheel selection will choose the sixth individual more than any other individual.

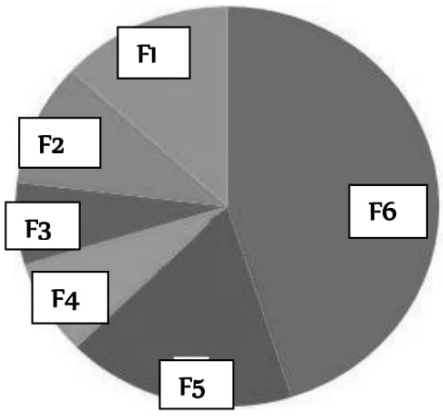


Figure 1.6. Roulette wheel method.

B. Stochastic universal sampling selection (SUS)

It is very similar to roulette wheel selection, but instead of generating a one random number in step 2, it generates two numbers and the two selected chromosomes are coupled for crossover.

C. Rank selection method (Random selection)

The application of the roulette wheel selection is not satisfactory in genetic algorithms, when the fitness value of chromosomes differs very much. It is a slower convergence technique, which ranks the population by certain criteria and then every chromosome receives fitness value determined by this ranking. This method prevents quick convergence and the individuals in a population are ranked according to the fitness and the expected value of each individual depends on its rank rather than its absolute fitness.

The rank selection method is shown in Figure 1.7. For example, if the best chromosome fitness is 80 percent, its circumference occupies 80 percent of the roulette wheel and then other chromosomes will have minimum chances to be selected. On the other hand, the rank selection first ranks the population according to their fitness and then every chromosome receives ranking. The worst will have fitness 1, the second worst will have a fitness of 2, and the best one will have a fitness value n, where n is the number of chromosomes in the population.

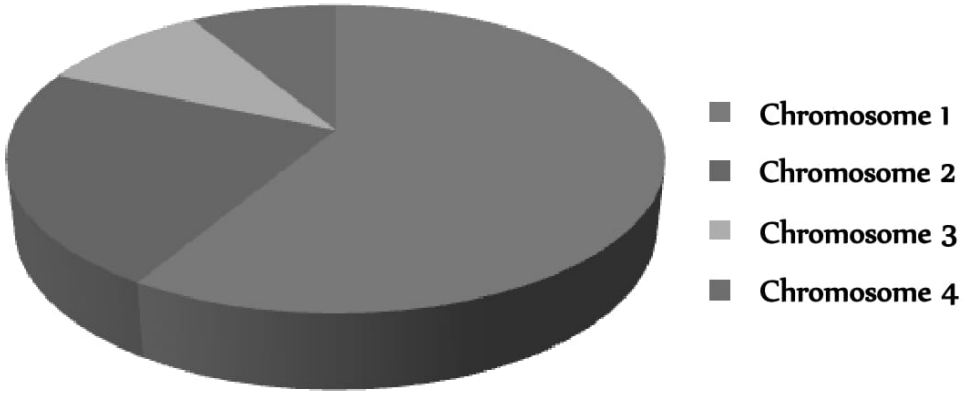


Figure 1.7. Rank selection methods.

D. Tournament selection

In K-way tournament selection it is different, picked chromosomes are not necessarily the selected ones, the following steps describes how it works:

Step 1 [Pick] Pick **k** number of chromosomes from the population at random.

Step 2 [Select] Select the best 2 chromosomes to be parent 1 and parent 2.

Step 3 [Loop] Go through the same previous steps until it is not possible to couple anymore.

It became popular among GA users and researchers because by it is nature it is also compatible with negative fitness values unlike the previously stated methods.

E. Steady-state selection

This method replaces few individuals in each generation, and is not a particular method for selecting the parents.

Only a small number of newly created offspring put in place of least fit individual. The main idea of steady-state selection is that bigger part of chromosome should retain to successive population.

3.3.3. Genetic algorithms operators

Genetic algorithms can be applied to any process control application for optimization of different parameters. Genetic algorithms use various operators, viz., crossover and mutation for the proper selection of optimized value. Selection of proper crossover and mutation techniques depends upon the encoding method and as per the requirement of the problem.

3.3.3.1. Crossover

It is the process in which genes are selected from the parent chromosomes and new offspring is created. Crossover can be performed with binary encoding, permutation encoding, value encoding and tree encoding.

A. Binary encoding crossover

In the binary encoding, the chromosomes may crossover at a single point, two points, uniformly or arithmetically. In single point crossover, a single crossover point is chosen and the data before this point are exactly copied from first parent and the data after this point are exactly copied from the second parent to create new offspring. Two parents in this method give two new offspring. The two point crossover is illustrated in Figure 1.8.

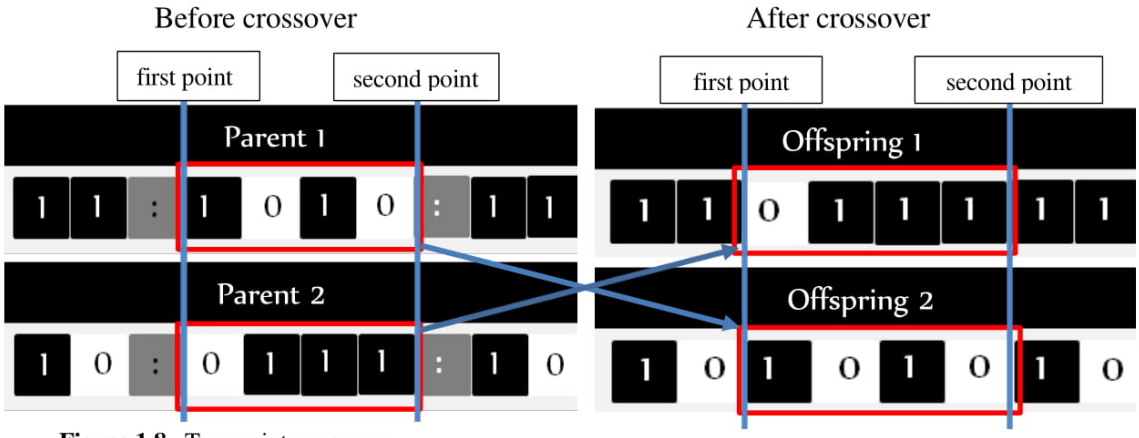


Figure 1.8. Two point crossover.

B. Uniform crossover

In uniform crossover, data of the first parent chromosome and second parent chromosome are randomly copied, which is illustrated in Figure 1.9.

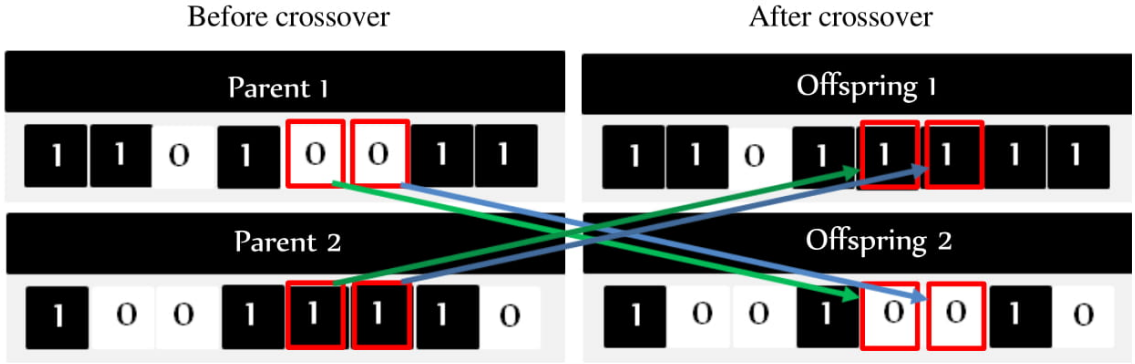


Figure 1.9. Uniform crossover.

C. Arithmetic crossover

In arithmetic crossover, crossover of chromosomes is performed by AND and OR operators to create new offsprings as illustrated in Figure 1.10.

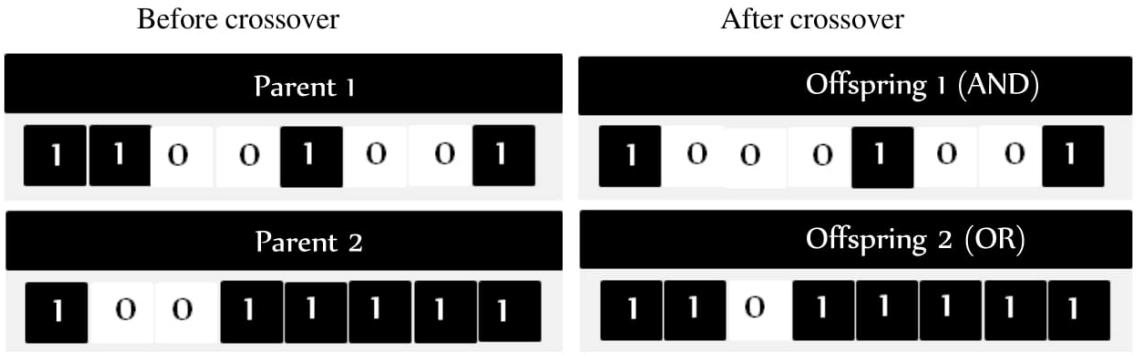


Figure 1.10. Arithmetic crossover.

D. Permutation encoding crossover

In permutation encoding crossover, one crossover point is selected. The permutation is copied from first parent chromosome up to the point of crossover and the other parent chromosome is exactly copied to ensure that no number is left to be put in the offspring. Further, if the number is not yet in the offspring, it is added to the offspring chromosome. Travelling salesman problems and task ordering problems can be easily solved by permutation encoding. Figure 1.11 illustrates the single point crossover with permutation encoding.

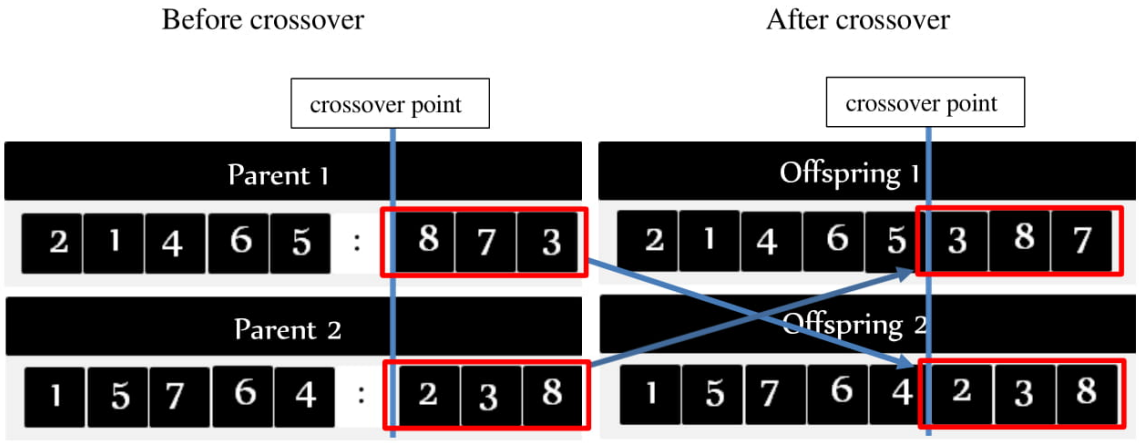


Figure 1.11. Permutation encoding crossover.

E. Value encoding crossover

It can be performed at a single point, two point, uniform and arithmetic representation as in binary encoding technique. Figure 1.12 illustrates the single point crossover with value encoding.

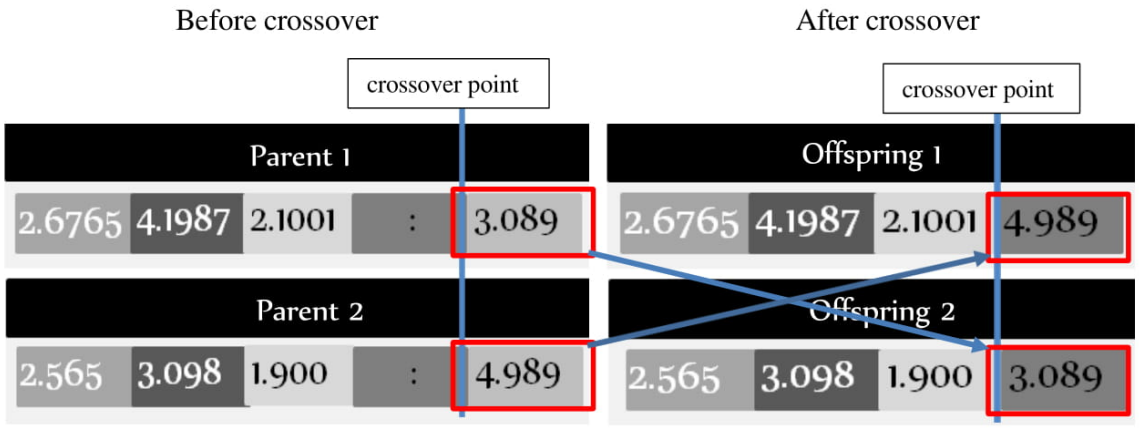


Figure 1.12. Value encoding crossover.

F. Tree encoding crossover

In this type of crossover, one point of crossover is selected in both parent tree chromosomes, which are divided at a point. The parts of the tree below crossover point are exactly exchanged to produce new offspring, which is illustrated in Figure 1.13.

The choice of the type of the crossover is strictly depends upon the problem.

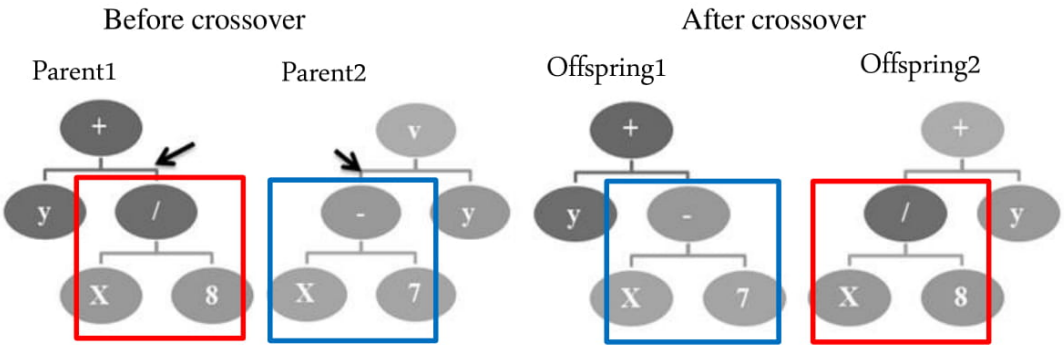


Figure 1.13. Tree encoding crossover.

3.3.3.2. Mutation

Premature convergence is a critical problem in most optimization techniques, consisting of populations, which occurs when highly fit parent chromosomes in the population breed many similar offspring in early evolution time. Crossover operation of genetic algorithms cannot generate quite different offspring from their parents because the acquired information is used to crossover the chromosomes. An alternate operator, mutation, can search new areas in contrast to the crossover. Crossover is referred as exploitation operator, whereas the mutation is exploration one. Like crossover, mutation can also be performed for all types of encoding techniques.

A. Binary encoding mutation (Bit Flip)

In binary encoding mutation, the bits selected for creating new offspring are inverted, which is illustrated in Figure 1.14.

In binary encoding mutation, if the bit 1 is converted into bit 0, it decreases the numerical value of the chromosome, and is known as down mutation. Similarly, if the bit 0 is converted into bit 1, the numerical value of the chromosome increases and is referred as up mutation.

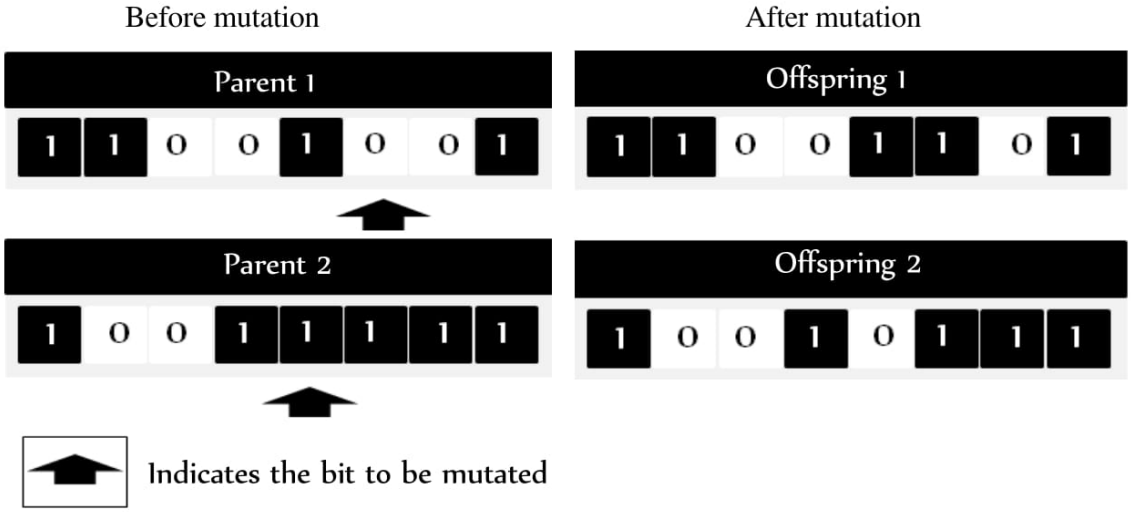


Figure 1.14. Binary encoding mutation.

B. Permutation encoding mutation

In permutation encoding mutation, the order of the two numbers given in a sequence are exchanged as it is illustrated in Figure 3.15.

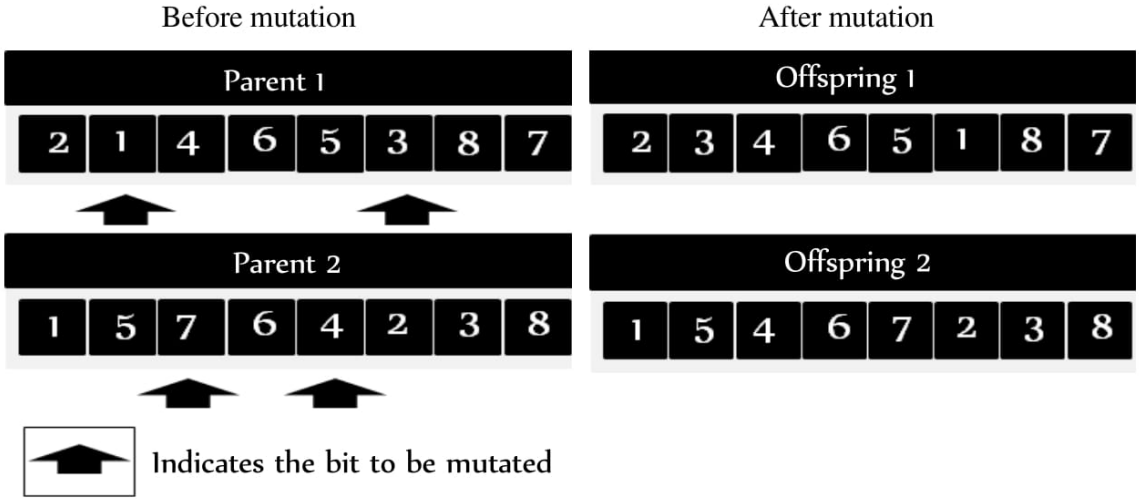


Figure 1.15. Permutation encoding mutation.

C. Value encoding mutation

In value encoding mutation, a smaller numerical value is either added or subtracted from the selected values of chromosomes to create new offspring, which is illustrated in Figure 1.16.

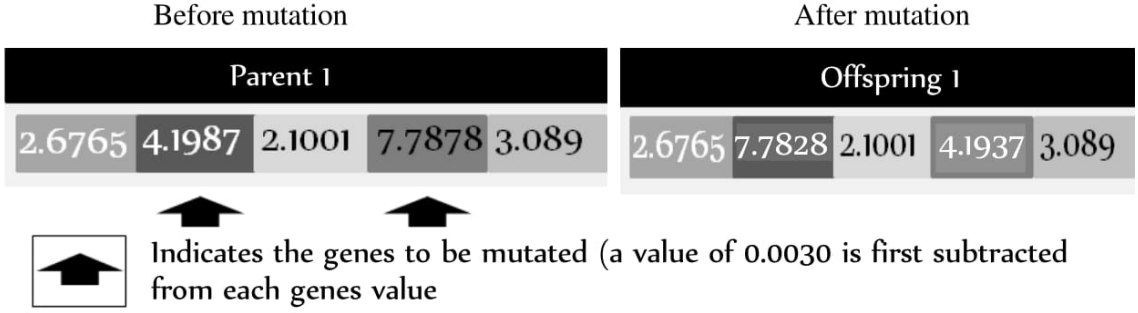


Figure 1.16. Value encoding mutation.

D. Tree encoding mutation

Tree encoding mutation, mutates the certain selected nodes of the tree to create new offspring, which is illustrated in Figure 1.17.

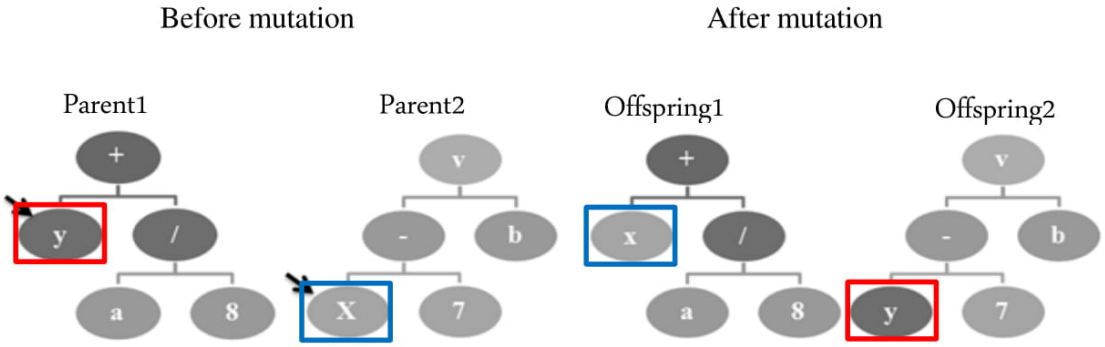


Figure 1.17. Tree encoding mutation.

3.4 Genetic algorithms issues

Genetic algorithms can be applied in complex non-linear process controllers for the optimization of parameters. Some issues are important to be considered for proper implementation of genetic algorithms to a plant to be optimized. Deciding on population size is an important issue while applying genetic algorithms. It is recommended by the researchers, that the population size should be of about 20 to 30 chromosomes. A very big population size consumes more time for finding optimum solution, which may deteriorate the performance of genetic algorithms [8].

Genetic algorithms may suffer from the problem of premature convergence due to improper selection of crossover rates. The higher crossover rate of about 85 percent to 95 percent is recommended to minimize premature convergence problems.

The low mutation rate of about 0.5 percent to 1 percent is generally recommended to obtain optimized results from genetic algorithm. Mutation is an artificial and forced method of changing the numerical value of the chromosome. Mutation should be avoided as far as possible because it is totally random in nature. Small mutation rates prevent genetic algorithms from falling into local maxima or minima.

Deciding on selection method for selecting good chromosomes is another important issue while applying genetic algorithms for process control applications. Rank selection method and roulette wheel selection methods have shown good results over other methods of selection.

Genetic algorithms play an important role in process control applications for the optimization of parameters. Many researchers have contributed in this field. A broad review of genetic algorithm applications gives the directions to use this technique for the optimization of the process controllers.

According to [8], even GAs present many advantages, GA have also some weakness, limitations, and difficulties includes:

1. The problem of identifying the fitness function and definition of representation of the problem;

2. The problem of choosing the various parameters like the size of the population, mutation rate, crossover rate, the selection method and its strength;

3. Some time, premature convergence occurs and have trouble finding the exact global optimum;

4. Not effective for smooth unimodal functions, no effective terminator, and cannot use gradients;

5. Cannot easily incorporate problem specific information and Configuration is not straightforward;

6. Not good at identifying local optima, it needs to be coupled with a local search technique.

Even if GAs met these limitation, one can mitigate some by profiting its advantages like its possibility of parallelism and easy to discover global optimum by parallel search from multiple points in the space, hybridization with other different methods etc; or by identifying the problems that are solved easily by GAs (e.g. GA solve comfortably combinatorial optimization and transportation problems than smooth unimodal functions, for the later one can hybrid GA with other methods like Gradients etc.

4. Conclusion

As Optimization grows to be a concerning topic to many researcher and achieving it can put a system ahead of the others, it can be viewed as an effective solution only when the problem is hard enough and there is no straight forward algorithm to solve it, in which GA will shine depending on the given scenario. In this chapter, we have defined the main used optimization algorithm in literature, and especially based on what the simulator promises to deliver and more than that. In the followed chapter, we will detail how GA works inside our simulator with a proposed example to demonstrate its power. In that, our objective is going to form a number of students into groups of fairly similar performance, depending on the number of students the problem might turn to be impossible to solve without optimization, which is going to be constrained by two hard constraints and one soft.

CHAPTER II:

GA SIMULATOR DESCRIPTION AND IMPLEMENTATION

CHAPTER II

GA Simulator Description and Implementation

1. Introduction

Genetic Algorithms (GAs) has been around for decades, yet many researchers and students still facing problems of the unavailability of the required tools, or finding solutions that are related to very specific problems and may or may not be applicable to their own problems. The GA simulator presented in our work, is a general case of software that can be configured to process any kind of optimization problem that is solvable using genetic algorithms. GA simulator provides the basic tools and parameters to advanced ones, and also it offers the ability to implement certain parts separately for later integration. In addition, the processing updates are displayed on charts for the users.

In this chapter, we will show the major changes and versions of developing the simulator over its life cycle, the requirements needed for its lunch, some useful web links needed to download the simulator. Also, we will put the light on the main features that can be used to reduce the time needed to get the required solutions for a specific problem. Finally, we will conclude with a demonstrative example.

2. GA simulator Versions and Requirements

2.1. Major versions

In this section, we state only the versions with major changes, the complete log is available on GitHub release page [1]:

- 11 Sep 2019: initiated the project with initial code, MIT Licence, and set up essential packages, marked as version 0.0.1-alpha.
- 17 Sep 2019 (version 0.1.0):
  - Established a communication between python process and electronJS process.
  - Implemented a population and individual (Chromosome) classes on python.
  - Implemented a working Evolve class with crossover and mutation.
  - Refactored code and removed unnecessary packages.
- 11 Oct 2019 (version 0.2.0):
  - Migrating to high-charts instead of Charts.js for chart plotting.
  - Created Stand-alone python executable instead of communicating with python file.
  - Added step-forward button (runs for 1 generation then pauses).

[1] https://github.com/dnory0/genetic-pv/releases/ (visited 13 Sep 2020).

- Adjusting simulator to be able to run and be installed on Linux and Windows.
- Added tooltip when cursor hover over the chart.
- Added parameter support: population size, genes number…etc.
- Separated parts of the UI as different processes (Views).
- Bug fixes, documentation improvements and UI style changes.

- 28 Oct 2019 (version 0.3.0):
  - Zoom functionality over the entire interface.
  - Added loading animation when starting the simulator.
  - Added delay rate parameter.
  - Saving parameters changes when closing the simulator and loading them on start time.
  - Big fixes.
- 09 Nov 2019 (version 0.4.0):
  - Added a button for settings.
  - Ability to enter parameters now by either text-field or scrollbar.
- 22 Dec 2019 (version 0.4.3):
  - Heavy fixes.
  - Separating functionalities as files and importing them.
  - Added Zoom keyboard shortcut.
  - Added developer tools keyboard shortcut for each view.
- 08 Feb 2020 (version 0.4.4):
  - Live rendering button is added (toggles whether the chart should draw on every update or after pausing/stopping the simulation, critical for low hardware machines).
  - Code/Documentation optimizations.
- 15 April 2020 (version 0.4.7):
  - Updated packages.
  - Using heat-map type of chart for showing fittest genes.
  - Renamed settings button to GA Control Centre and it is now working and shows a window with more advanced parameters to configure.
  - Added pin button to some parameters in GA Control Centre to pin in main window for easy access.
  - Disabled some parameters when simulation is in process.
  - Added feature to full-screen charts, save them as picture on jpeg/svg format.
  - Added the ability to disable some parameters if not used (ex: delay rate).
  - Adding zoom in and zoom out on charts with panning through when zoomed.
  - More polished tooltip, bug fixes and speed improvements.
- 26 Jul 2020 (version 0.5.1 and the last version on 0.x.x):
  - Python interpreters of version 3.7.7 are shipped with executables on windows.
  - Improved/Added icons, UI style and bug fixes on borders between views and more.
  - Packaging the MIT licence file with executables (installers and Portable versions).
  - Added json-view package to show loaded genes.
  - Added a red dot on top of GA Control Centre button and flicker browse buttons inside the Control Centre to indicate user did not load genes data or fitness function (or both).
  - Added a parameter that limits the numbers of 1s and 0s inside a gene.
  - Added selection, crossover, and mutation types' parameters.
- 17 Sep 2020 (version 1.0.4, latest to date 17 Sep 2020):

- Users now can download fitness-function.py template through get template button and implement their own fitness function can add specialized crossover/mutation function.
- Added ability to download the results of simulation through a button.
- Added Compatibility with negative values fitness function.
- Updated packages to latest bug fixes versions and added prettier package.
- Updated python interpreters to version 3.8.5.
- Fixed Icon on Linux installers and other bug fixes.

2.2. Requirements

2.2.1. System Support

System support depends on the Graphical Interface Framework (ElectronJS) and the platforms used for developing the simulator; here are the specifications from the ElectronJS website [10]:

- Windows 7 and later (32/64 bits).
- For Linux: Ubuntu 12.04, Fedora 21, Debian 8 or newer versions (32/64 bits).
- MacOS 10.10 Yosemite and later (only 64 bits).

2.2.2. Computer Specifications

There is no information concerning hardware requirements for the simulator, however, the machine with the minimum specifications that we have tested has the following configurations:

- CPU: i5 M460 2.53 GHz
- RAM: 4 GB
- System: Windows 7 64 bits.

2.2.3. Additional Installations

- For Linux: python 3.8 or later is required.
- For windows: nothing, everything needed is pre-packaged with the simulator.

2.3. Development

The development is carried on the following configurations:

- CPU: i5 4200U 1.6 – 2.3 GHz.
- RAM: 6 GB
- System: Ubuntu 20.04 / Windows 10 (same machine with both systems and both 64 bits)
- Editor/IDE: Visual Studio Code and PyCharm.
- Programming Languages: Typescript (Javascript), HTML, SCSS (CSS), and Python.
- Packages used for development purpose:
  - Electron v8.5.1: the graphical interface framework.
  - Electron-builder v22.8.0: used to create Installers and Portables (runs without install).
  - Prettier v2.1.1: code formatter to improve readability.
  - Sass v1.26.10: compiles SCSS files to CSS.
  - Typescript v3.9.7: compiles typescript files to Javascript.
- Packages that are pre-packaged with the simulator on deployment:
  - Highcharts v8.2.0: the framework responsible for drawing graphs.
  - Json-view v1.0.0: used to present json files for user after loading genes data.
  - For Windows: portable python interpreters v3.8.5 are embedded (both 32 and 64 bits).

Note: The packages versions are applicable on v1.0.4 of the simulator [2].

3. *GA simulator principles: Case of Study*

3.1. Structure

GA simulator is structured in a manner that allow it to be flexible and responsive, in instance, a chart processing does not block the whole UI, and using multiple processes, threads allows tasks splitting, while IPC (inter-process communication) assures information is up-to-date over all the simulator parts. In addition, using electronJS framework that uses html and css for interface design allows having modern and more polished look, which enhances the user experience. Briefly, the following scheme summarizes the structural components: (start with the main process)

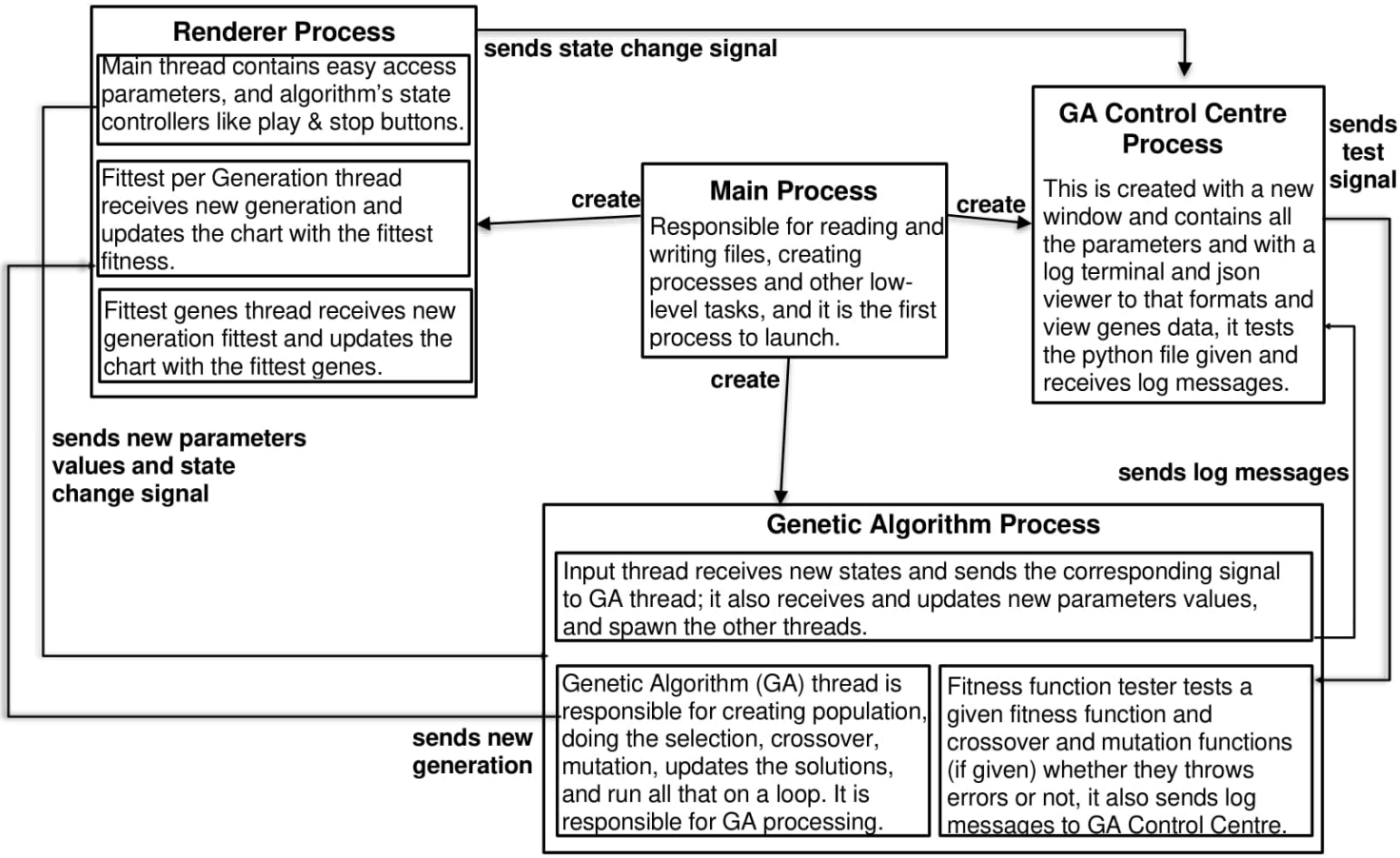


Figure 2.1. Structure of the simulator.

3.2. Interface and Features

One of the things that define the progress of a software is the interface and the features on it, how much they help, so here we present how the simulator looks like, and how to use it:

[2] https://github.com/dnory0/genetic-py/releases/tag/v1.0.4 (published & visited 17 Sep 2020).

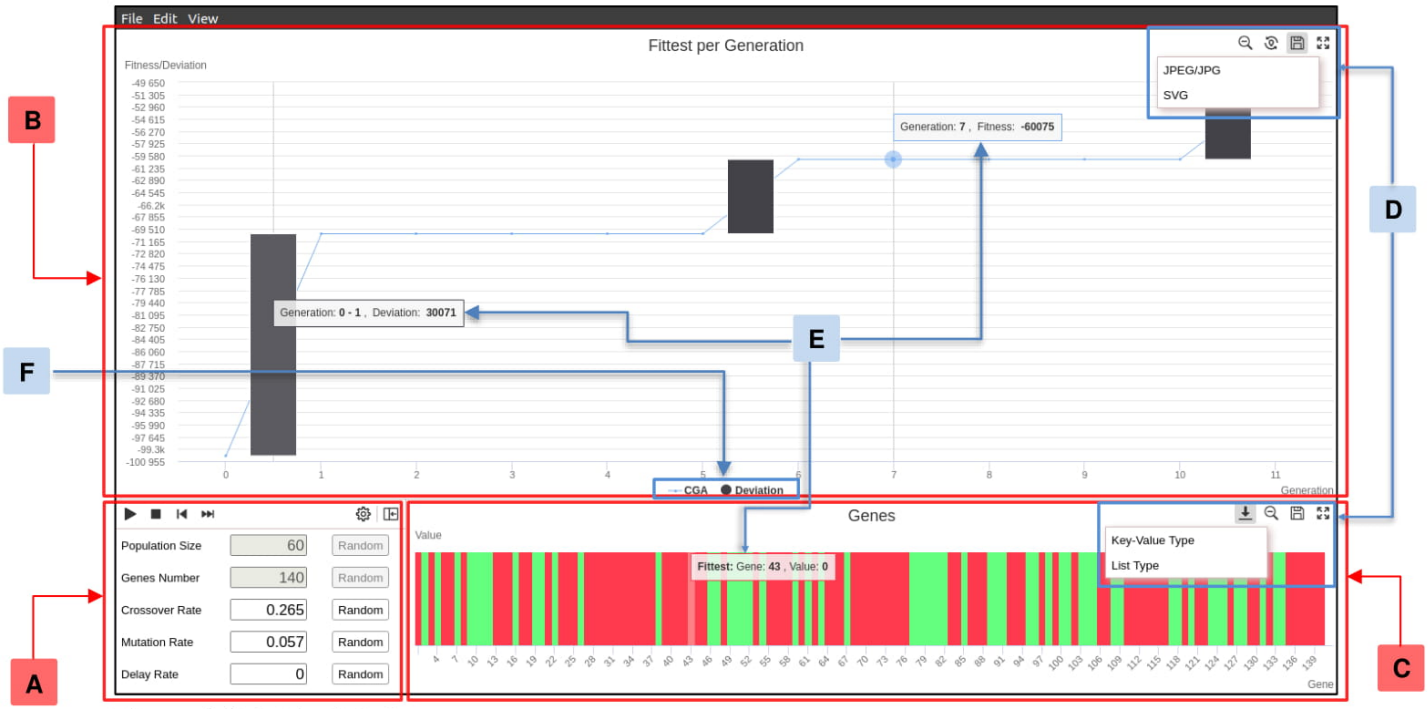


Figure 2.2: Main Window.

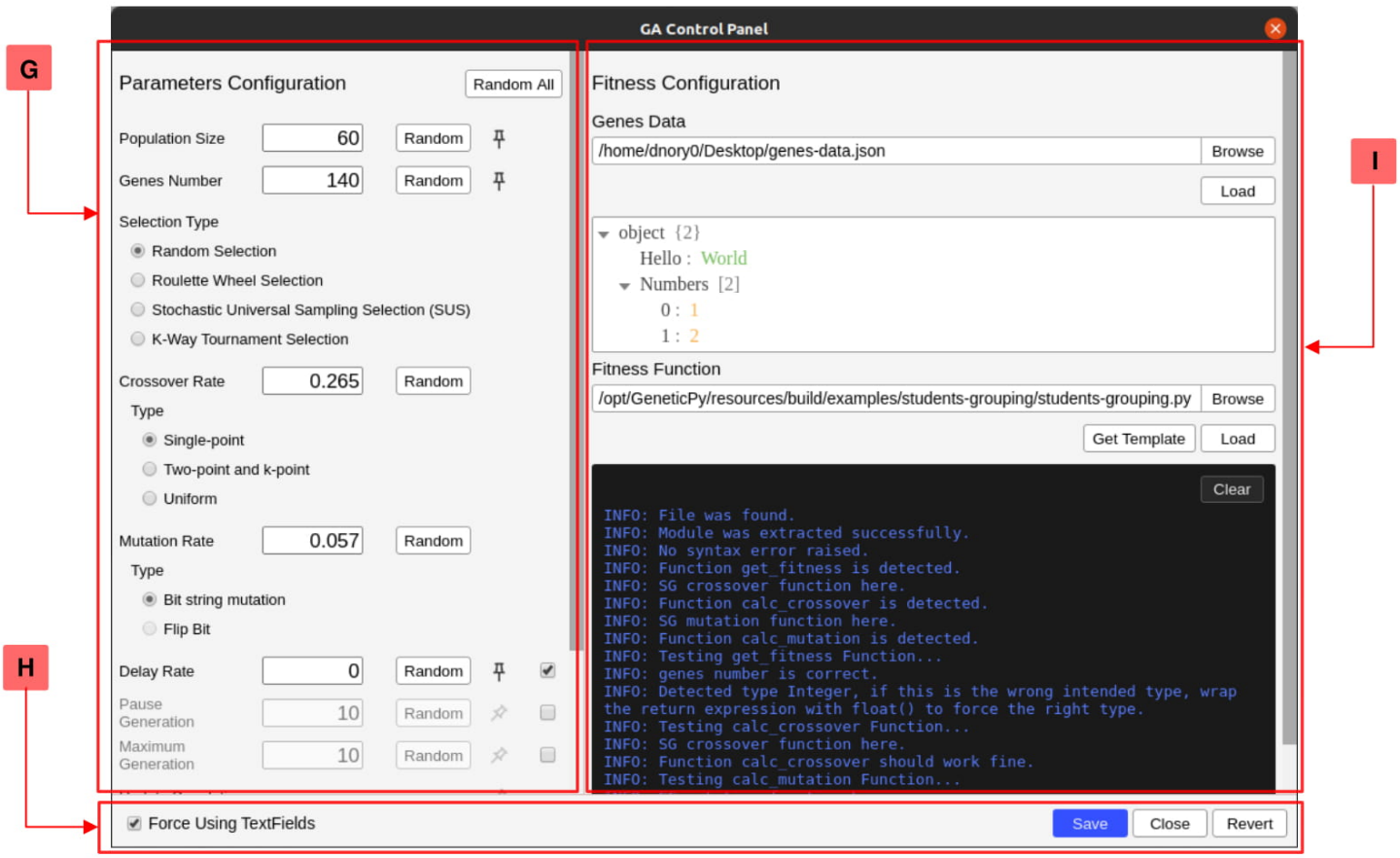


Figure 2.3. GA Control Center.

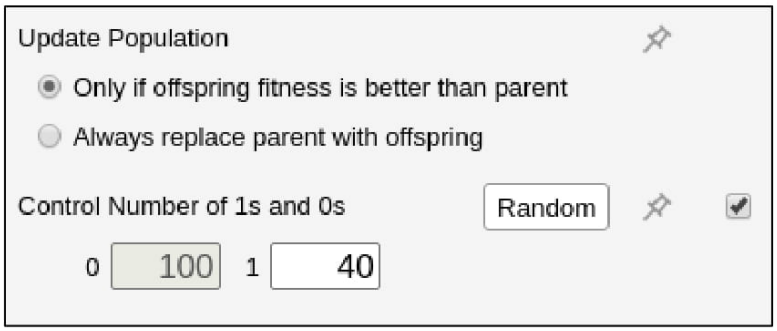


Figure 2.4. Rest of parameters.

A. ***Control Part:*** Located on the renderer process main thread, it has the buttons controlling the optimization process: play, pause, stop, relaunch algorithm, and step forward. In addition to that GA Control Center button and hide button, which is responsible for hiding the part the whole control part and showing it again, this is useful to allow part 3 to spread horizontally. There is also an easy access parameters part, this usually has basic and frequently adjusted parameters with random button for each, notice that population size and genes number are disabled when the genetic algorithm is running, they are going to be enabled after pressing the stop button.

B. ***Fittest per Generation Part:*** This is a separated view, holding a chart that plots two series, the first one for fitness value of each generation's fittest chromosome, and a deviation serie, plotted as pillars whenever there is a difference between two consecutive generations in fitness value.

C. ***Fittest Genes Part:*** Genes chart refreshes on every generation holding the genes of the newest generation fittest, while the green color presents genes of value 1, and red for genes of value 0.

D. ***Action buttons:*** holding full screen button, to full screen the corresponding chart, and a save button, which holds two options (as shown on the fittest per generation part) and saves an image of the chart, and a zoom out button. There also buttons which are exclusive to a certain chart:


Download the result data, which is the genes values of the presented chromosome on the Fittest Genes Part.


Live rendering, when enabled, the chart refreshes on every generation, but in low-end hardware computers, it is helpful to disable it to save CPU computation and the chart is going to update the new values only after pressing pause or stop button.

E. ***Tooltip:*** Tooltip shows information concerning the point or the bar that is selected.

- The chart keys.
- The user can hide or show an option of series on the chart by clicking on its name on the key.

F. ***Parameters:*** This part on the GA Control Centre holds all the parameters you can configure with the simulator, you can randomize them individually or by random all button (randomizes only enabled parameters), some of them can be pinned to the easy access part on main window or unpinned, and some other can be disabled and enabled if they are optional to use. Therefore, here is what some parameters can do:

- *Population Size:* controls the number of possible solutions on every generation.

- *Genes Number:* controls the number of genes present on every solution at any given generation.
- *Delay Rate*: time to pause between each generation in seconds, default to 0.
- *Pause Generation*: in case users are not going to be facing the simulator all the day, they can specify a generation to pause the algorithm automatically.
- *Maximum Generation:* same as Pause Generation but stops the algorithm instead of pausing it.
- *Update Population*: this adjust how the simulator behaves when updating the population with new offsprings, they can always replace parents or only if an offspring fitness value is better than its parent fitness value.
- *Control Number of 1s and 0s*: as the name states, allows you to specify the number of genes containing value 1 on every solution and as a result the number of genes containing the value 0 as well. It is not possible to control the number of 1s and 0s with bit flip mutation type so it is disabled when this parameter is enabled.
- *Note*: both Update Population and Control Number of 1s and 0s parameters are shown on figure 4, because figure 3 does not show the rest of the parameters.

G. This contains the buttons to revert any change made to GA Control Center parameters, to close the panel, or to save the new changes, it also contains the checkbox to force using textFileds or scrollbars on some parameters that allow, and here is how some of them looks like with a scrollbar:

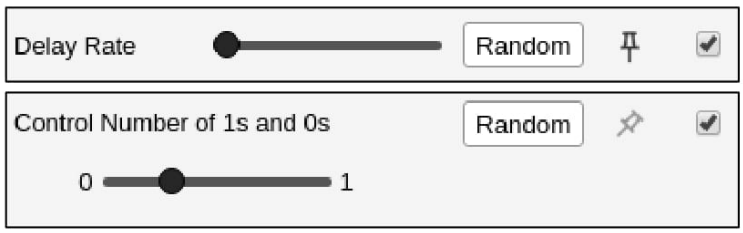


Figure 2.5. Parameters with scrollbar input.

H. ***Additional parameters:*** Genes Data is imported as JSON format file and supplied as an argument to the fitness function, this is in case fitness function depends on external data; the content is also formatted and shown below the path input, if the file contain errors, the content will be unable to show.

Fitness Function is a python file that contains the fitness function, and optionally can contain user's specific crossover and/or mutation function. Below it there are log messages as shown on figure 3, this logs the important steps to show whenever there is something wrong (on red color) or a warning (in yellow), in case user finds the log part became very long there is the clear button at top-right corner to clean the log messages. In addition to that, there is a get template button next to load button, this downloads a copy of fitness function file with empty body and built-in functions and documentation to help users to implement their own fitness function and import it later.

Note that the browse button by default points to directories with ready examples that can be used directly just by importing them, and this is where we showcase one of the examples we prepared.

3.3. Case of study

Previously we talked about how the simulator looks like and how it works, but in this part we are going to show an example where the simulator works on a group formation problem mentioned on this

paper [9]. The basic idea is that we have a number of students that vary in terms of their grades and how much they are productive, we want to form them in groups so that each group's performance is expected to be approximately similar, and we achieve that by systematically distributing the students rather than randomly assigning them to groups.

In this example we have 28 students, we want to form them into 4 groups so that each group could contain 7 students, we labelled each student from 0 to 27 and assigned each of them a value to how good of students they are and their quality. Those values are going to be supplied to the simulator as genes data because we need them to calculate the fitness values of the solutions, and each chromosome is going to represent a mixture of students distributed all over the groups and will contain all the 28 students, so for example: (again students are labelled from 0 to 27)

Chromosome =

[5, 16, 4, 1, 2, 21, 25, 8, 22, 9, 20, 15, 12, 14, 11, 3, 7, 13, 17, 19, 24, 0, 26, 10, 6, 23, 27, 18]

Group 1 | Group 2 | Group 3 | Group 4

In the simulator, we are going to represent students labels as binary representation, so each student is going to be represented on the base 2, the same chromosome above is going to be like:

Chromosome =

[0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, Group 1

0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, Group 2

0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, Group 3

0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0] Group 4

1st student — 28th student

Notice that the chromosome length has changed so that every student is represented now by 5 bits, so the length of the chromosome is going to be 28 * 5 = 140 bits.

We used 5 bits because log2 (28) = 4.8 ≈ 5, and so we have 31 representations. But we only need 28 (0 – 27), and because the generation number 0 is randomized solutions, the representations (28 – 31) which do not point to any student are going to appear, so in this case we have to implement the fitness function to drop the solutions fitness value or treat them in the mutation function.

3.3.1. Fitness Function

The fitness function is composed of two hard constraints and one soft constraint:

- The hard constraints are when a block is repeated twice meaning a student is mentioned twice, or when we have a block that translates to representations from 28 – 31. We add the value–10000 to the fitness value of the corresponding chromosome every time one of these constraints is met, this way we tell the GA that this solution is useless to us and it is highly recommended to replace it or treat it.
- The soft constraint optimizes so that all the groups' performance have to converge to a median value.

The group fitness is calculated as a sum of quality values of the members of that group:

$$groupfitness_i = \sum_{j=0}^{7} studentquality_{ij} \qquad where\ i = the\ i^{th}\ group \tag{1}$$

The soft fitness is the sum of the absolute values of the differences of each group fitness to the median value of all the group fitness values, multiplied by the minus sign because we want to lower this difference between groups so the optimal value is 0, thus their values and performance is identical:

$$softfitness = -(\sum_{i=0}^{4} |groupfitness_i - median(groupfitness)|) \quad (2)$$

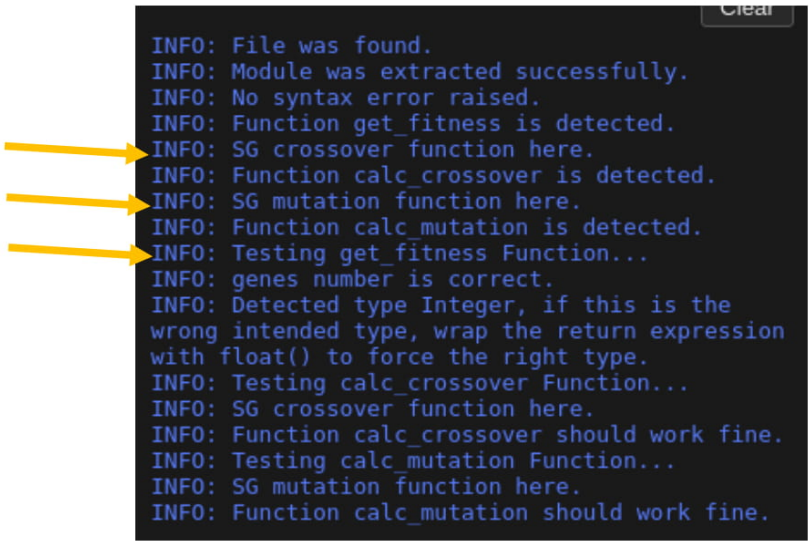
The whole fitness value is:

$$fitness = hardfitness + softfitness \quad (3)$$

Note that in the end, the fitness value is actually wrapped by a function that removes the fraction part and only keeps the integer part; it is only removed so the chart looks clean with no fractions.

3.3.2. Setting up parameter inside the simulator

- *Population size:* after testing couple of times, we found that if the population size is less than 60 the algorithm is going to struggle breaking out of local optimums, we are going to use 70 to assure we are on the safe zone while saving up some speed.
- *Genes number:* this is going to be 140 as mentioned above.
- *Selection type:* this is user's preference; we are going with roulette wheel selection.
- *Crossover rate:* a value bigger than 0.1 is recommended here, we took 0.265.
- *Crossover type:* we provided a special crossover function inside the imported python file. It works just like Single-point crossover but makes sure that the chosen point is a multiple of 5 (number of bits), ex: taking value 0.265, then, 0.265 * 140 ≈ 37 but crossover at this point is cutting the 8th student representation, this is discouraged, the solution is 37 / 5 ≈ 7, then 7 * 5 = 35, then 35 is a multiple of 5.
- *Mutation rate:* a value bigger than 0.05 is recommended here, we took 0.115.
- *Mutation type:* we also provided a special mutation function inside the imported python file. It does not mutate one bit at a time but a whole block of 5 bits with another block of the same size. In addition, if there is a block that is repeated in one chromosome that is to say a student repeated twice or a block that parses to (28 – 31), this function randomizes the block again.
- *Update population:* set to only if offspring fitness is better than parent option.
- Other parameters on the left side of the GA Control Center are disabled, but we still have genes data and fitness function importing:
- *Genes data*: as mentioned before, when processing fitness value of a solution or a chromosome we need the quality value assigned to each student; this is where we import it.
- *Fitness function*: we import the python file containing the fitness function, along with both the special crossover and mutation functions, here are the log messages:



As shown by the arrows, all the 3 functions are detected, and there are no error or warning messages which means the testing went without any issues. Here is how an error looks like:

Moreover, here is how a warning looks like:

3.3.3. Running the genetic algorithm

Now all that remains is saving the parameters values and pressing the play button for the algorithm to start processing. The processing graph is zoomed and separated into 2 important parts as shown in figure 2.6 and 2.7. The first part where the GA is trying to generate solutions that satisfy the hard constraints, and this is achieved at the 95th generation, where the fitness value appears to be near 0 (actually it is -196), which took approximately 9 seconds to achieve, and any solution before that is considered unusable.

The second part is the GA trying to optimize the soft constraint, and tries to bring the groups' performance closer. This is an interesting part because the more you keep the algorithm running the more the fitness value converges to 0. We only kept the simulator running for 5 minutes, and that resulted to fitness value equals to -14 at the 3305th generation, the figure 2.7 shows the soft constraint optimization progress.

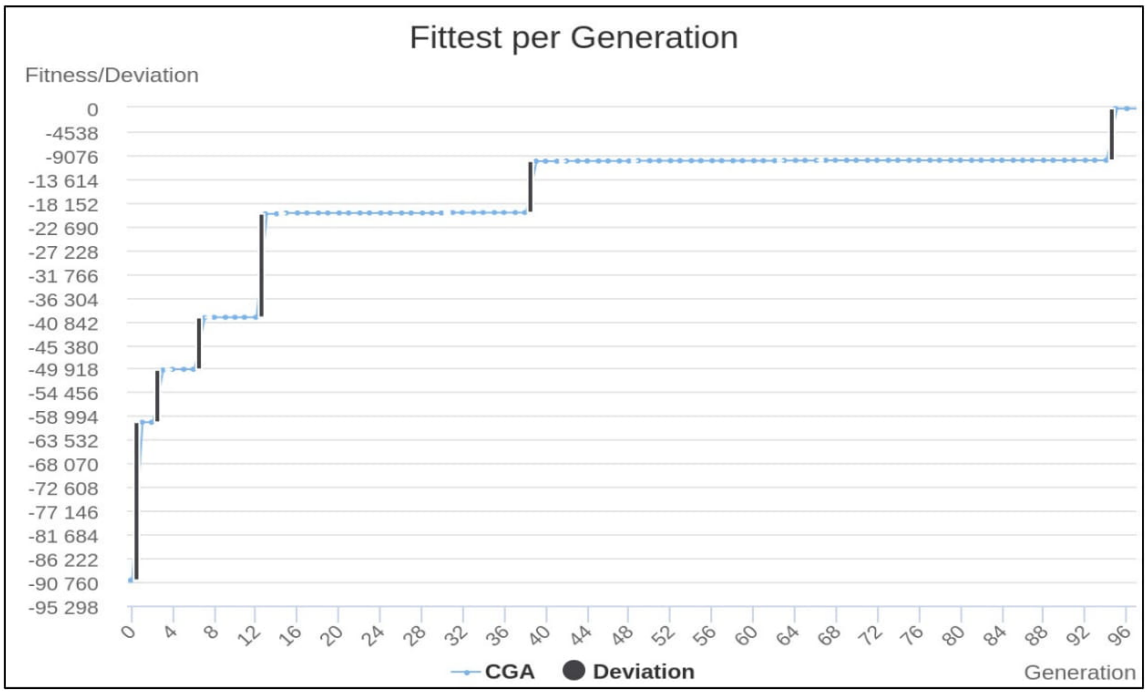


Figure 2.6. GA satisfying the hard constraints.

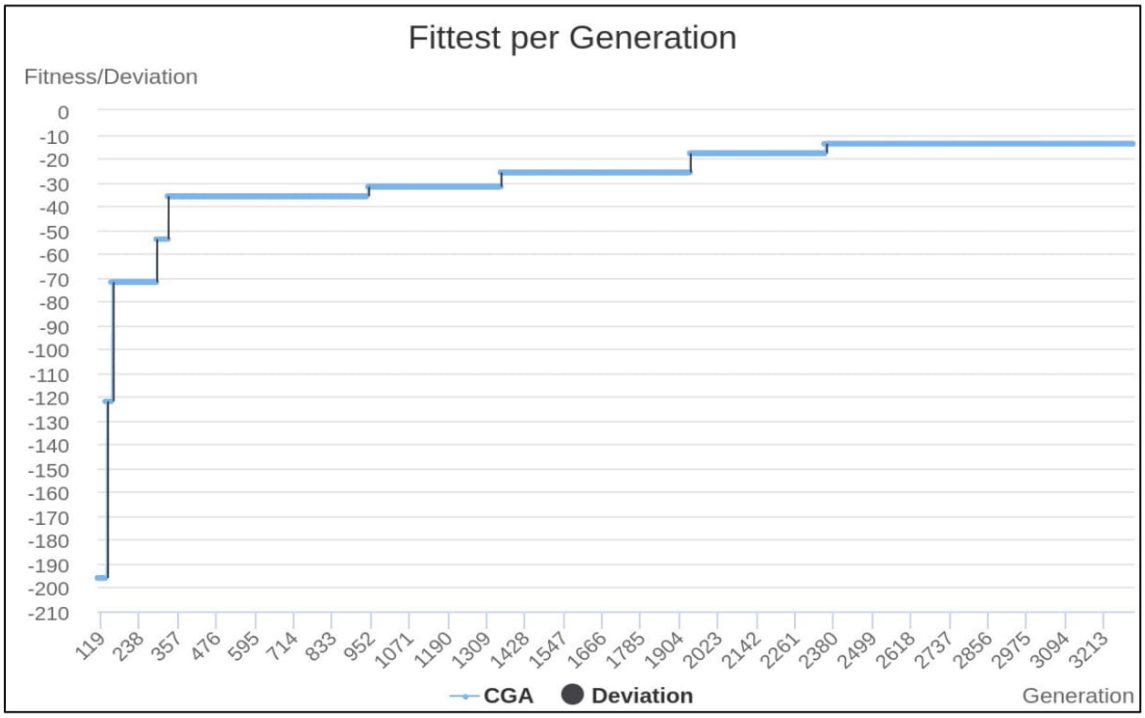


Figure 2.7. GA optimizing the soft constraint.

After stopping the algorithm, we download the fittest genes as a list type, which obviously are going to be a sequence of 140 bits:

Fittest Chromosome =
[1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0,

Group 1

Group 2

1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1,
0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, **Group 3**
1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1] **Group 4**

At this point, we created a python code that translate each block of 5 bits to an integer value to get the 28 students and the 4 groups and this is the result:

```
dnory0@dnory0-computer:~/Documents$ python3 deserializer.py
paste the solution: [1,0,0,1,1,0,1,0,0,1,0,0,0,0,0,0,0,1,0,0,1,0,0,0,1,0,0,0,0,0,0,1,0,0,1,0,1,
0,0,0,0,1,1,1,0,1,1,0,0,0,0,0,0,1,0,1,0,1,0,1,0,1,1,0,0,0,0,1,1,0,1,1,1,1,1,1,0,1,1,1,1,0,0,0,1,1
,0,1,0,0,0,1,1,0,0,0,1,0,1,0,1,1,1,0,1,0,0,1,0,1,0,0,0,1,0,1,0,1,1,1,0,1,1,1,0,1,1,0,1,1,0,1,0,1,
1,1,0,0,1]
paste the genes data: {"0": 53,"1": 50,"2": 86,"3": 89,"4": 99,"5": 73,"6": 54,"7": 84,"8": 59,"9
": 99,"10": 71,"11": 95,"12": 69,"13": 85,"14": 51,"15": 88,"16": 78,"17": 66,"18": 71,"19": 52,"
20": 78,"21": 86,"22": 81,"23": 80,"24": 61,"25": 50,"26": 99,"27": 97,"28": -10000,"29": -10000,
"30": -10000,"31": -10000}

group 1 has the following members: [19, 9, 0, 2, 8, 16, 4], and the group fitness is: 526
group 2 has the following members: [20, 7, 12, 1, 10, 22, 3], and the group fitness is: 522
group 3 has the following members: [15, 27, 24, 26, 6, 5, 14], and the group fitness is: 523
group 4 has the following members: [18, 17, 11, 23, 13, 21, 25], and the group fitness is: 533
```

Figure 2.8. Decoding the results.

First, the code asks for the genes of the solution as an input, and then asks for genes data as shown on the blue box, because it is going to calculate every group's fitness value. The result appears in the red box which is each group's number followed with its members (7 students), at the group's fitness value, the values are as expected fairly close to each other (around 525).

Note that this decoder file is beyond the simulator's job, because there is no way to know the meaning of the fittest genes for every problem, the user is responsible for decoding his genes values by his own, depending on what every gene represents to his problem. However, this decoder might come pre-packaged with this example's fitness function file in the future versions of the simulator so that other users can inspect its code and hopefully benefit from it.

4. Conclusion

Working with genetic algorithms can turn to be very productive, as they proved to be very efficient for solving complex problems. Having set of tools that is inside this simulator can save up time and effort, as long as the user focuses on representing the problem.

Having all the basic features and functions, with small steps of setting your parameters and importing the fitness function that calculates your genes fitness value, the simulator sets up all the rest needed to start processing any problem. User is also allowed to use external code if the problem given is a little special and that means more extensibility.

Finally, the future work is dedicated to include more parameters, as this can help to bring more options graphically instead, reduce the programming part with special crossover and mutation functions becomes less needed. Our aim is also to provide more examples, and adding support to non-binary

chromosomes representations, as some problems might be processed better if they are represented otherwise.

References

[1]. Christian Blum, Jakob Puchinger, Gunther R Raidl, and Andrea Roli. Hybrid metaheuristics in combinatorial optimization: A survey. Applied soft computing, 11(6):4135–4151, 2011.

[2]. JB Atkinson. A greedy randomised search heuristic for time-constrained vehicle scheduling and the incorporation of a learning strategy. Journal of the Operational Research Society, 49(7):700–708, 1998.

[3]. Manuel Laguna, J Wesley Barnes, and Fred W Glover. Tabu search methods for a single machine scheduling problem. Journal of Intelligent Manufacturing, 2(2):63–73, 1991.

[4]. Alex S Fraser. Simulation of genetic systems[1] by automatic digital computers I Introduction. Australian Journal of Biological Sciences, 10(4):484–491,1957.

[5]. H.J. Bremermann. The Evolution of Intelligence: The Nervous System as a Model of Its Environment. University of Washington, Department of Mathematics, 1958.

[6]. David E Goldberg et al. The Design of Innovation: Lessons from and for Competent Genetic Algorithms by David E. Goldberg, volume 7. Springer Science & Business Media, 2002.

[7]. Melanie Mitchell.An introduction to genetic algorithms. MIT press, 1998,

[8]. Jingcao Hu and Radu Marculescu. Energy-and performance-aware mapping for regular NOC architectures. IEEE Transactions on computer-aided designof integrated circuits and systems, 24(4):551–562, 2005.

[9]. Zhamri Che Ani, Azman Yasin, Mohd Zabidin Husin, and Zauridah AbdulHamid. A method for group formation using genetic algorithm. International Journal on Computer Science and Engineering, 2(9):3060–3064, 2010.

Webography

[10]. https://www.electronjs.org/docs/tutorial/support#supported-platforms