Flexible Code Foundation Development For Further Study In Gender-based GA with Recombination Hotspots

Siu Kei, MUK

Research School of Computer Science, Australian National University
u5721042@anu.edu.au

Abstract. Genetic algorithms have been intensely applied to solve optimization problems that are hard to deal with using traditional differential methods, such as Newton Methods. However, the recent pace of advance of genetic algorithms are slowed. While it is widely acknowledged that the scalability is one of the major hindrance, this does not explain the effectiveness in biological evoluation. As the current abstraction of biology is far from optimal, extra exploration on the possible details in biology that is algorithmically important is desirable. This work serves as a preparation for further studies in the development of genetic algorithms. A flexible code base is developed for further experiments to be implemented and repeated with minimal efforts. This paper provides a high level perspective to the architecture used, together with development details and test experiments validations.

Keywords: Genetic Algorithm, Evolutionary Computing, Implementation, Global Optimization, Machine Learning

1 Introduction

In biology, useful traits and characteristics are passed to the offsprings through DNA mechanisms for them to achieve a higher survivability in the environment. By matching the idea of objective function in an optimization problem to survivability, and solutions to individuals, genetic algorithms provide an abstraction of such biological mechanism into computation systems. The abstraction aims to capture algorithmic aspects of biology that is important in terms of the effectiveness in computing. However, it is not known that which aspects are crtical and which are not. Therefore, the optimality of the current abstraction remains questionable. This is the primary motivation of this exploratory work in genetic algorithms.

1.1 Genesis of Project

In 2014, Antonsen and Fortescue carried out a preliminary study on genetic algorithm with recombination hotspot on a static problem[1] where no advantage was found. The study did not investigate its effect on dynamic problems,

which are more demanding of evolvability and adaptability of the population. This project was supposed to be an extension to the investigation of hotspots in genetic algorithm in a dynamic environment where improved behaviour is anticipated. However, it was discovered when I started the literature review that two weeks before this project commenced, a paper was published in 'The Genetic and Evolutionary Computation Conference' that did exactly the same work and confirmed that recombination hotspots do improve evolvability of modular systems[7]. This led to repeated efforts to contact the authors for more detail, but no response was received. The paper did not provide sufficient information about what problems they were working with, detailed experimental settings nor detailed results of their experiments. The only information that is clear is that they were working with gene regulatory networks, and the problems may be the derived from [2], but the details were not sufficient to fully replicate the experiment in the work. Due to the above situation, a complete restructuring of the project was required. The project is re-defined to be a code foundation development based on the reason that it is not known that which aspects of biology are algorithmically important. The exploration on such area requires a highly reusable and flexible framework that allows different details to be experimented with minimal effort.

This paper is arranged as follows: section two provides a background on the problem and literature survey, section three describes the model architecture, section four includes the development process and challenges encountered, section five provides the descriptions of code validation experiments. The work is concluded and future works are listed in section six. A brief implementation guide is included in the appendix.

2 Background and Literature Survey

2.1 Background

One of the most incredible self-sustaining process of the nature is described by Darwin's theory of natural selection. The theory can be summarized as: Individuals in a world with limited resources compete with each other for survival, the ones with characteristics that fit better to the environmental conditions would have a higher chance to survive and to reproduce. The characteristics would then be passed to their offsprings in the reproduction process. As time goes by, the process would lead to a population with an advanced survival rate then its ancestors. It has become the foundation of evoluationary biology.

Evolutionary computation abstracts this idea in computer-based systems attempting to achieve evolvability. One of the actively researched techniques in evoluationary computation is genetic algorithm. The genetic algorithm was invented by John Holland in the early 1970's. It mimics the natural selection process and applies it to optimization problems by abstracting individuals as candidate solutions and survivability of participants as the objective function

to be optimized (known as the fitness function). The following section gives an introduction to genetic algorithm.

2.1.1 Introduction to Genetic Algorithm

Genetic algorithm aims to solve optimization problems by random research guided by Darwin's theory of natural selection. Without loss of generality, we assume that it is a maximization problem. The objective function, also known as the fitness function in the GA context, serves as a measure of an individual's survivability in the environment. Each candidate solution is encoded into some format, which is called a genotype. As the format of a solution in its genotypic representation may not be compatible with what the fitness function demands, it is decoded into a format, known as the phenotype, that suits the fitness function before performing its survivability evaluation. The following figure shows an illustration of this process. In this example, we want to maximize the number of 1's of a pair of binary strings after performing the decoding on them. The genotype is a pair of binary string of length six, the decoding mapping consists of a table that indicates the dominant gene. The fitness function would be the number of 1's in the phenotype. As we can see in the diagram, the phenotype of the solution encoded by the pair '101011' and '110010' is '110011'. The reason is that 1 is the dominating allele at position $1, 2, 6$, whenever the alleles are not the same at these positions, 1 would be the final value in the phenotype. The case for 0 is the same. Therefore, it has a fitness of 4 as there are four 1's in its phenotype.

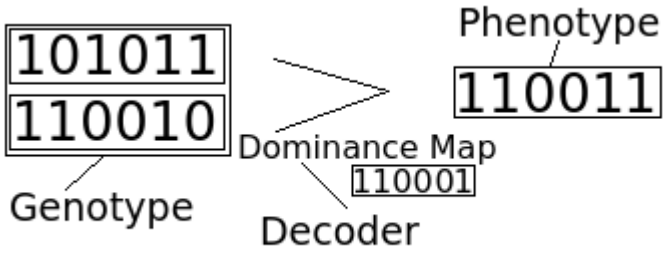

Fig. 1: Illustration of decoding process

In genetic algorithm, every stage, known as a generation, consists of a fixed number of candidate solutions, known as a population. The population will first go through the fitness evaluation mentioned above, then several processes to generate a new generation from the current one. Firstly, several individuals are selected to act as parents to generate offsprings. There are lots of selection schemes to do this, but often the ones that give advantage to individuals with higher fitness are used. Two commonly used schemes are proportional selection and tournament selection. In proportional selection, the probability of

an individual to get selected is proportional to the total fitness achieved in the population. Mathematically,

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j}$$

where i is the index of an individual, f_i is its fitness value. Here the fitness function has to be non-negative. In tournament selection of size $k \in \mathbb{N}$, k individuals are first selected randomly into the selection pool. Then the best individual would be selected by a pre-defined probability p (usually greater than 0.5), the second best individual would be selected by a probability of $p(1-p)$, the third best one would be selected by a probability $p(1-p)^2$ and so forth. If none of the best $k-1$ individuals is selected in the pool, then the worst performing one is chosen.

After selection, the selected parents will go through the reproduction process. This is when new offsprings are generated. The genotypes of the parents will first be copied, then the new copies will go through a gene swapping process (known as recombination/crossover) to exchange the alleles that occurs at gene positions randomly chosen uniformly. If the genotype contains more than one material, then there would be a mating process that materials are paired up. The newly generated offsprings will then go through the mutation process, where the alleles are randomly changed by some probability. The figure below illustrates this. In the example, the genotypes of two parents are first copied. Then, some random genes are swapped before mating. The locations where recombination occurred are indicated by arrows. In the mating process, we can see that the second component of both genotypes 1 and 2 are paired to generate *offspring1*, and the other pair is used to generate *offspring2*. Finally, the offsprings undergo mutation. The mutated genes are indicated in boxes.

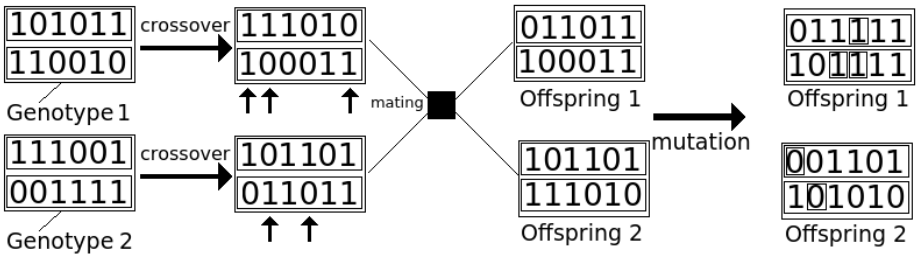

Fig. 2: Illstration of the reproduction process

Finally, when sufficient amount of offsprings are generated for the next generation, they form a population. This new population will then be evaluated for the fitess values, and the processes described above will be repeated until the stopping crterion is achieved.

2.1.2 Corresponding Mechanisms in Biology

2.1.2.1 Dominance

In biology, a single gene can have multiple different alleles (or gene values), which determine the characteristics that an individual would possess. Due to the fact that most of the plants and animals have paired chromosomes, known as diploids, the alleles at the same gene location (known as locus) on each chromosome can be different. According to the behaviour and interaction between alletes, the contribution of each allele may be fully acknowledge, interfered, or completely masked. Three dominance behaviours have been discovered in biology.

Complete Dominance

In complete dominance, the effect of one allele completely masks the effect of the other. A classic example is from Gregor Mendel's crossover experiment on peas. There are two types of peas, one of them has round-shaped seed, and the other type has wrinkled ones. Mendel noticed that when he tried to breed them separately according to their types, he will always get offsprings of the same type. However, when he tried to breed the peas by mixing the types, both some of the offsprings give rounded seeds, and some give wrinkled ones, in the ratio of 3:1. This phenomenon is explained by the alleles complete dominance. Let the allele that gives rounded seeds be 'R', and that gives wrinkled seeds be 'r'. The original combinations before crossing over are 'RR', and 'rr'. After crossing over, possible combinations are 'RR', 'Rr', 'rR' and 'rr'. From the observation above, one can conclude that the allele 'R' is dominant to 'r', or 'r' is recessive to 'R', as the effect of 'R' completely masks the effect of 'r'.

Incomplete Dominance

Incomplete dominance occurs when the phenotype is distinct from, or often intermediate to those with pure alleles. The snapdragon flower is a good example for this behaviour. The flower colour is either homozygous (having the same allele in both strands at a locus) for red or white. However, when the red homozygous flower is paired with the white one, the offspring becomes pink. None of the effect of white or red allele is fully expressed. Instead, the contributions are mixed to give a pink colour. When the pink offsprings are self-pollinated, the of flower colours in the next generation would be 1:2:1 (for red : pink : white), which can be explained by the possible permutations 'rr','rw','wr' and 'ww'.

Co-Dominance

Co-Dominance occurs when both traits from different alleles are visble in the expression. A well-known example is the human blood type. The alleles for type A and B at the ABO locus are co-dominant. This leads to the existence of blood type AB. Another example is the flower *Camellia*. The homozygous flower colour is either red or white. However, when they are paired, the offsprings contain white and red spots on the petals, rather than pure red, pure white, or pink colour.

2.1.2.2 Meiosis

In the reproduction process of human, the gametes are generated from both a female and a male for pairing up to generate an offspring. The process that the gametes are produced is meiosis. In meiosis, the chromosome pairs of an individual would first copy itself into two identical pairs. Then, each pair of them undergoes a recombination process where the genetic materials are exchanged between the paternal and maternal chromosome in the pair. After the recombination process, one would have four modified chromosomes in the cell. The cell is then split two times to form four gametes so that each contains one modified chromosome.

Recombination Hotspots

In the recombination process mentioned above, the alleles from each chromosome in the pair are swapped. The frequency that a swap occurs at a particular locus is called the rate of recombination (or recombination rate). The distribution of recombination rates in species is not uniform. The regions with a significantly higher rate are called recombination hotspots, and those with particularly low rate are called coldspots. The distribution of recombination hotspots are found to be different among species in various studies. In a recent research by Adam Auton et al.[4], it is shown that although the broad-scale (at the level of entire chromosomes) recombination rates were found to be very similar in humans and chimpanzees, at fine scales no shared recombination hotspots were found between the species, although it is shown that humans and chimpanzees share 99.4% identity at nonsynonymous (functionally important) sites and 98.4% at synonymous (functionally less important) sites in the work of Derek E. Wildman et al.[3]. Furthermore, the distribution of recombination rates between male and female is shown to be different[5] that about 15% of hotspots in one sex are specific to that sex[6].

2.1.3 Difference of GA and Biology

First of all, from the about one can easily see that there are at least two differences between traditional GA and biology: dominance mapping and recombination process. Traditional GA uses the abstraction of the complete dominance in biology where the characteristic from exacly one allele gets expressed for a heterozygous individual, while in biology this is not always the case. Another difference lies in the recombination process that traditional GA uses a uniform recombination rate everywhere in the chromosome, while in biology it is far from uniform, and the distribution depends heavily on the species, sex and the environment.

Recently, Ari Larson et al.[7] reported in their work that recombination hotspots may have a potential significance on genetic algorithms. In the experiment, modular networks are evolved with different level of guidances/restrictions on recombination as follows: E_1 uses mutation without recombination, E_2 allows random crossover, E_3 limits crossover to points known to emerge modularity for

fit networks, E_4 evolves recombination rates guided by linkage learning, and finally in E_5 crossover follows the phenotypic structure of the network using the Q metric for modularity. The evolvability of the networks under guided recombination styles (E_3 - E_5) is shown to be outperforming the unguided ones.

Together with the works mentioned above ([6], [5], [4]), it is suspected that the recombination hotspots may be algorithmically important. This may serve as a good starting point for exploratory researches, although the reason why traditional GA is way less effective than the system in biology is not explicitly known.

As recombination hotspot is not an independent mechanism in the nature that exists by its own, several other supporting elements might also turn out to be important in the exploration. Some examples are problem modularity, dynamicity, diploid chromosomes, interaction with dominance mechanisms, et cetera. It is shown that the diploid chromosome is capable to preserve diversity of the population in evolution[8], and the dominance schemes are crucial for adaptability in dynamic environment[9][10][11].

2.2 Literature Survey

The following subsections provide a brief review of general genetic algorithm architecture developed.

2.2.1 Initialization

The initialization of population is usually done in a random fashsion. The main advantage is that an initial population with individuals distributed uniformly through out the solution space can be expected. This normally provides an adequate level of diversity for evolution when the population size is sufficiently large. On the other hand, several guided approaches, such as opposition-based[12] and hill-climbing[13] initialization, are developed to enhance convergence speed.

2.2.2 Fitness Function

The fitness function is the objective function to be optimized by genetic algorithm. Normally, given an objective function, one can use that directly as the fitness function. However, a great amount of functions exhibits certain kinds of problems, such as precision (function value being too large or too small), execution time (especially for model parameter search problem, where cross-validation or other performance evaluation methods are needed), function behaviour (differentiability, negative values, etc), and so forth. In [14], several scaling schemes are presented attempting to convert the function into one with better precision and behaviour. A neural network approach is proposed in [15] where the actual fitness function is replaced with a neural network trained with past evalution results to achieve efficient evaluation.

2.2.3 Reproduction

The reproduction process is often divided into two parts: offspring generation and mutation. The offspring generation is normally done by gene value swapping at each randomly selected gene location (known as recombination/crossover), while mutation is performed by randomly modifying gene values at random gene positions. With diploid or multiploid chromosomes, the offspring genertaion is divided into recombination and mating. Recombination is responsible for gene material exchange, while mating combines materials from different parents to form a new chromosome. **Fig.2** provided an illustration of this process. In [16], recombination occurs before mating, while in [17] it was done in the opposite order.

2.2.4 Selection

The selection process is mainly responsible for choosing parents to participate in the reproduction process according to some selection scheme. One of the most commonly used scheme is tournament selection, in which k individuals are randomly chosen to compete with each other with a winning probability according to their fitness values. k is known as the tournament size. Other common examples are propotional scheme, non-linear ranking scheme, etc.

2.2.5 Other Operations

2.2.5.1 Expression Map

The expression map performs the genotype-to-phenotype mapping. This operation is mainly used in GA with diploid or multiploid chromosomes. A simple exmaple would be a diploid having two binary strings, some of the values are not equal at the same gene location, illustrated as fig[] below. In this case, the expression map is required to decide the outcoming gene value in the phenotype. The expression map can be changed according to the progress of GA. In [9] and [10], dominance change schemes are proposed and shown to be crucial to the adaptability of population to dynamic environments.

2.2.5.2 Elitism Selection

The elitism selection is one of the most common operation. It chooses a number of best performing individuals in the population. The selected individuals are often allowed to survive the next generation, rather than to participate in the reproduction process.

2.2.6 General Genetic Algorithm Architecture

The general main flow of genetic algorithm is as follows:

```
Input: Hyperparameters, such as population size,
       crossover rate, etc.

Output: Best solution in the evolution

1. Initialize population
```

```
2.  Evaluate population
3.  While (stopping criteria is not reached):
4.  Do
5.      Generate next population
6.      Perform mutation on next population
7.      Population <- next population
8.      Evaluate population
9.  End
10. Return fittest individual
```

This base procedure is used in most of the published works on genetic algorithm. The only difference between those is the additional details involved. In [9], [10], [17], [11], the process consists of a change in dominance based on individuals' performance and the environment. In [16], the fitness function gets updated according to number of constraint violations by individuals, and elitism is applied after reproduction.

The above architecture provides a general framework of how genetic algorithm works on the highest level. This becomes an critical piece of information in the development of the flexible platform described in the following sections.

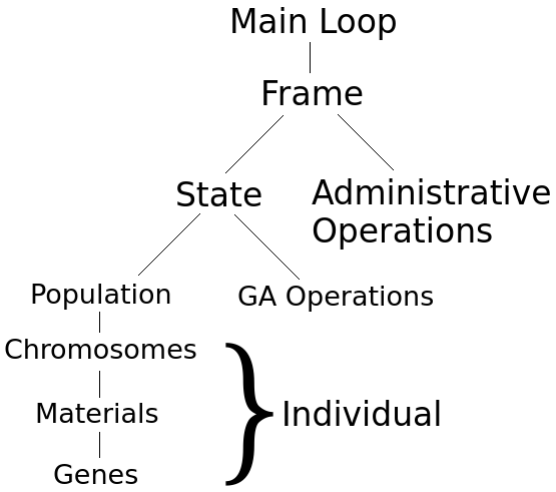

Fig. 3: Overall Architecture

3 Model Architecture

The implementation is divided into two modules: main module and extension module. The main module provides a foundation for a general genetic algorithm. The extension modules provide additional functionalities for GA with

specific features. This work includes the main module and an extension module for gender-based GA with recombination hotspots. A detailed description of the architecture and components is provided in this section.

3.1 Main Module

3.1.1 Overview

The overall architecture of the main module is illustrated in **Fig.1**. The highest level is the main loop defined in the client code, which uses the *Frame* object to trigger the evolution for each generation. A *Frame* object contains a *State* object that manages the state of progress, and a set of operatrors that perform administrative actions, such as updating the statistics. A *State* object consists of a *Population* object and a set of genetic operators. It is responsible for triggering the reproduction process. A *Population* object comprises a collection of individuals in the current generation, and several pools for individuals for the next generation. When a new generation of individuals is generated, they would be temporarily stored in the pools until the new generation is ready to replace the old one. An *Individual* contains a *Chromosome* object, and a fitness value for sorting purpose. More details are given in the following subsections.

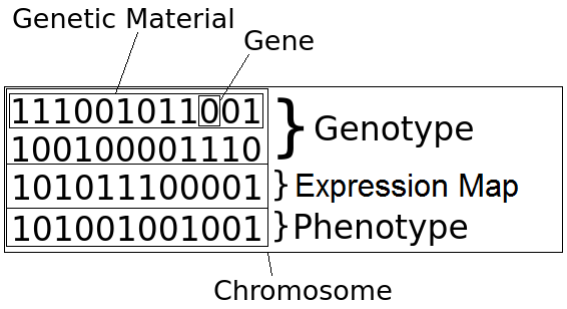

Fig. 4: Hierarchy of Chromosome Composition

3.1.2 Individuals

An *Individual* consists of a *Chromosome* and a fitness value. The structure of a *Chromosome* object is illustrated in **Fig.2**. Each component is described below.

3.1.2.1 Gene

The *Gene* class is at the bottom of the hierarchy. A *Gene* object is a container of a value. The value can be anything defined by the user that is used in encoding the solution, such as a number, a character, or even an object. The value can be modified in run-time to allow efficient gene-level operation, such as mutation and recombination.

3.1.2.2 Material

The *Material* interface is in the middle of the hierarchy. Its implementation serves as a collection of *Gene* objects. It is abstracted as an interface to achieve maximal flexibility as different material structures are allowed in the further studies.

3.1.2.3 Chromosome

The *Chromosome* class represents the candidate solutions in the genetic algorithm. A *Chromosome* object consists of a collection of *Material* objects as its encoding for the corresponding candidate solution (known as genotype), a *Material* object as the decoded version of the solution (known as phenotype), and a *Expression Map* object as the decoder.

At the time when the evaluation of an *Individual* object is requested, a *Fitness Function* object would be provided to perform the task. This allows an efficient and flexible exchange of fitness function in run-time.

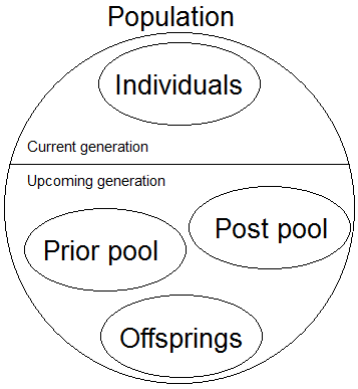

Fig. 5: Composition of Population

3.1.3 Population

The *Population* class serves as a container of individuals. A *Population* object consists of a collection of individuals in the current generation, and several pools that store the individuals for the upcoming generation. The basic pools are: prior pool, offspring pool, and post pool.

3.1.3.1 Prior Pool

The prior pool holds the individuals generated/selected to survive to the next generation before reproduction occurs. Its necessity comes from the situation

where one may not have enough rooms for the individuals generated/selected separately after reproduction.

3.1.3.2 Offspring Pool

The offspring pool stores the individuals generated from the reproduction process.

3.1.3.3 Post Pool

The post pool is responsible for holding the individuals generated/selected to survive to the next generation after reproduction occurs. Its necessity becomes clear when a reproduction rate is used resulting in the population size not being fully achieved after reproduction.

The pools are separated so that the pool-level operation (say mutation) would not be able to affect each other. An example would be the use of elitism scheme, where it is more desirable for the chosen best individuals from the current generation to stay unchanged, while the newly reproduced offsprings go through the mutation process for diversity enrichment.

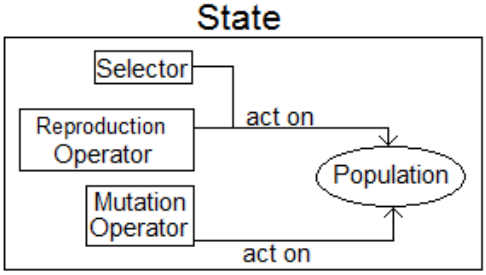

Fig. 6: Composition of State

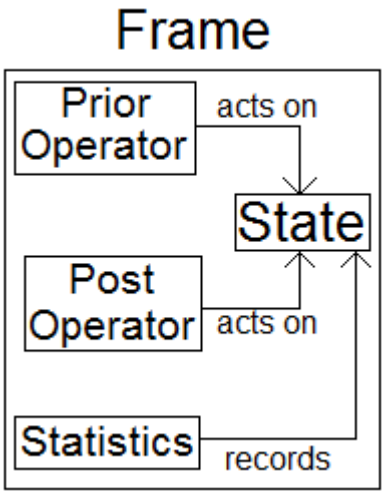

Fig. 7: Composition of Frame

3.1.4 State

The *State* class manages the state of the progress. A *State* object consists of a *Population* object and the operators required for reproduction, such as reproduction, mutation and selection. It is responsible for triggering the reproduction

and mutation process. The state would be provided to the *Statistics* for recording purpose.

3.1.5 Frame

The *Frame* class provides administrative supports for the progress. A *Frame* object contains a *State* object and administrative operators, namely *Prior Operator*, *Post Operator*, *Dynamic Handler* and Statistics. This class defines the high level procedure of how the population evolve by one generation. This includes the use of prior operation, post operation, handling of environment change and recording the current state to *Statistics*.

3.1.6 Operations

Every operation in the module is abstracted as an interface. This allows a great degree of flexibility to be achieved by modularization of operations. The operators are provided only when needed, as an object of the corresponding interface type.

3.1.6.1 Fitness Function

The *FitnessFunction* interface represents the objective function to be optimized. Here, it is assumed that the problem is always a maximization problem. Other than evaluating a solution by phenotype, this interface also has an *update* method that signals the fitness function to update its environmental parameter in a user-defined way.

3.1.6.2 Expression Map

This interface provides an abstraction to the mapping from genotype to phenotype for each chromosome. The *map* method performs the mapping by accepting a list of *Material* objects and returns a single *Material* as phenotype.

3.1.6.3 Dynamic Handler

This operation is responsible for handling environment changes. After modifications of fitness function/environment, the fitness function value of each individual may not be the same as those before the change has occurred. Therefore, handling is necessary after the fitness function update, such as the re-evaluation of the population in the current generation.

3.1.6.4 Initializer

The *Initializer* is an abstraction of population initialization. It is not necessarily implemented for this framework to work. However, it is recommended to provide an implementation as one may need to repeat the experiment with different initial conditions on the population.

3.1.6.5 Prior Operator

The *PriorOperator* represents the operations on the current population before reproduction. One of the possible usage is elitism. The importance of the operation being executed before reproduction process is that it ensures there are sufficient vacancies for the survivors to be included in the next generation. This operation is optional.

3.1.6.6 Selection Operator

The selection operator is responsible for performing selecting parents to participate in offspring generation. This operator is split into two parts, namely *SelectionScheme* and *Selector*.

3.1.6.6.1 Selection Scheme

This interface provides an abstraction for a selection scheme, given a list of fitness function values sorted in descending order. The implementation is expected to return the index of the selected individual.

3.1.6.6.2 Selector

The implementations of this interface are expected to choose an appropriate number of parents for reproduction. The selection scheme returns the index of exactly one chosen entity, while the selector returns a list of individuals to participate in a single reproduction process.

3.1.6.7 Reproduction Operator (Reproducer)

The *Reproducer* is an interface for reproduction operation. The implementation of this interface is expected to perform reproduction. Note that recombination and mating are to be done internally in the *Reproducer* implementation, they are not separated at this level as their order are not fixed. The *Reproducer* returns the offspring generated by the given parents.

3.1.6.8 Mutation Operator (Mutator)

The *Mutator* is an interface for mutation operation. A mutator modifies the *Gene* values randomly subject to a probability. The implementions are expected to loop through every single gene in a population.

3.1.6.9 Post Operator

The *PostOperator* represents the operations on the current population after reproduction. In constrast to *PriorOperator*, the *PostOperator* is required in the algorithm to maintain a constant population size. A simple example would be a filling operator that fills up the vacancies by individuals in the current generation chosen by some selection scheme.

3.2 Extension Module for Geneder-based GA with Recombination Hotspots

3.2.1 Overview

The module is built on the basis of the main module. The *Coupleable* interface is defined to force chromosomes to have a gender together with a hotspot. The *Hotspot* class defines hotspots that determine the recombination rate of each gene loction to be encodable by some user-defined scheme. A *GenderPopulation* class is an extension of *Popluation* defined to maintain the gender proportion in the population. The *State* and *Frame* classes are extended to provide handling for specific features to gender-based GA with recombination hotspots. The *Selector* is now required to choose exactly one individual from each gender independently to participate in reproduction. The *CoupleReproducer* forces the parents to be coupleable, and an abstract class *GenderReproducer* is provided as an implementation foundation for further uses. An abstraction of a new operator *HotspotMutator* is defined for hotspot mutation process.

3.2.2 Coupleable

This interface requires a chromosome to contain a gender and a *Hotspot* object. The gender flag is to be used in selection and reproduction process. The *Hotspot* object supports guided recombination.

3.2.3 Hotspot

The *Hotspot* class represents the recombination rate vector that determines the likelihood of material swapping to occur in recombination. It comprises an encoding and a recombination rate vector. The encoding vector allows the rate to be expressed in another form. A discretized scheme in which the swapping probability is determined by $\exp(-sn-d)$ where $s, d \in \mathbb{R}_+$ with variable $n \in \mathbb{N}$ is a possible encoding. The encoding is transformed into actual probabilities when it is used in recombination.

3.2.4 Gender Population

The *GenderPopulation* has an additional proportion field that determines the proportion of male to female individuals to be maintained in the evoluation. This provides a way for the diversity to be preserved, and most importantly prevents extinction of population due to total dominance of a particular gender over the other from happening.

3.2.5 Gender State and Frame

The *GenderState* and *SimpleGenderFrame* classes requires chromosomes to be coupleable. A *GenderState* object has an additional functionality that triggers the hotspot mutation process, while *SimpleGenderFrame* includes this action into the evoluation step of a generation.

3.2.6 Operations

3.2.6.1 CoupleReproducer and Gender Reproducer

This interface is dedicated to the reproduction process carried out by coupleable chromosomes. The *GenderReproducer* provides an overall implementation. The specific detail of how an offspring is generated is left to be defined by the user in the *recombine* method.

3.2.6.2 Hotspot Mutator

The *HotspotMutator* interface is an abstraction of mutation for *Hotspot* objects. The implementation is responsible for modifying the encoding values of a *HotSpot* object in a probabilistic fashion.

3.2.6.3 Couple Selector

The *CoupleSelector* class provides a foundation implementation for general selectors for gender-based GA. It consists of one collection of individual for each gender to perform selection separately.

4 Development

4.1 Choice of Language

The code is developed in Java 7. This choice is due to the fact that in genetic algorithms, the interaction between different components, such as chromosomes and genes, plays an important role in the progress, which is best to be implemented in an object-oriented language.

4.2 Development Approach

The architecture is developed using a bottom-up approach. In this approach, the base components are first defined, and then the whole structure is built on the elements. As the structures on the top are the extensions of the bottom elements, minimal restrictions are placed on the potential add-ons, which ensures flexibility in terms of further development of the library.

During the development, three test experiments are implemented to test the validity of the library. The first test experiment tests whether the basic components interact correctly with each other in a static environment. The second test experiment checks whether the engine is sufficiently flexible for handling dynamic optimization problems. The last test experiment tests if the main module is extendable by running the GA with several additional features, such as gender and hotspots in chromosomes. The descriptions are included in the *Code Validation*. The test experiments served as guide to the desired structure of the library.

4.3 Main Challenges

The first challenge is that one must take care of backward compatibility while adding new components on top of the original ones. The compatibility problem often occurs when a feature discovered important later in the development is decided to be a compulsory element of the whole model, which affects everything that was already fixed in place.

The second challenge is that the abstractions have to be designed very carefully. The reason is that this library serves as a preparation implementation for further studies in genetic algorithms, inappropriate restrictions that prevent any possible details from being explored in the future shall be forbidden.

4.4 Development Process

The first prototype was finished in four days. The development was guided by a test experiment of a simple bit-matching problem. This initial prototype is capable of handling static optimization problem with haploid (single material) encoding. The library at this stage is largely limited, and the expression mapping was not defined yet. The first test experiment code consists of 6 self-defined

classes, which is almost the same as a standalone experiment implementation without the assistance of this library.

The second prototype was completed in the next two weeks time. In this prototype, the expression mapping and the environmental change handler are added to the library. The modification was guided by the experiments replicated from [9] and [10]. The *Chromosome* is modified to include an experssion mapping that maps from the genotype to phenotype. This allows every chromosome, even haploid, to separate its genotype from phenotype so that more encoding schemes, such as a combination of several genes for a single trait, are possible. Furthermore, the handler for dynamic changes in the environment is inspired by the dominance change performed in the experiment, where the same population is re-evaluated and the genotype is changed according to its performance related to the previous environment.

The third prototype was completed in the following one month time. The extension was guided by a simple one-max problem on gender-based GA with recombination hotspots. In this prototype, the additional features are implemented in an extension module. An interface *Coupleable* is defined to force a chromosome to have a gender and hotspot. Every collections and several operations need special handling for the gender difference. The selection operator is a good example that it is more desirable to perform the selection separately for male and female, rather than in a mixed pool. The hotspot implementation replicates the structure of chromosome that it has an encoding part and an actual probability representation determined by the encoding. The original idea behind was to allow discrete probability levels, but it turns out this allows more flexible schemes to be used in the hotspots. The hotspot mutation operation is also mirrored from the mutation process in GA.

The final prototype was completed in the following three weeks time. In this prototype, implementations are provided for some of the frequently used basic components or operations, such as binary genes, tournament selection scheme, reproduction operation using uniform random recombination, et cetera. This version is solely developed for the convenience of potential client codes, as some of the settings are usually used, such as binary genes.

5 Code Validations

Testing implementations are done to see if the library works as it is intended to be. Three experiments are implemented to verify the functionality of different part of the library.

5.1 Test 1

This test aims to verify that the basic interactions between components in the main module work as intended. The components included in this test are as follows:

Gene, *Material*, *Chromosome* (Haploid), *Expression Map*, *Initializer*, *Population FitnessFunction*, *Mutator*, *Reproducer*, *SelectionScheme*, *Selector*, *State*, *PriorOperator*, *PostOperator*, *Statistics*, *Frame*.

In this test, a simple bit-matching problem is employed. In this problem, a 32-bit target bit-string is defined by the user. Then, the genetic algorithm is used to evolve solutions (encoded as 32-bit bit-string) towards the target string. The fitness function is the number of bits that matches with the target. The propotional selection scheme is used. The expression map is the identity map, so the genotype and the phenotype are the same. Elistism is applied (as prior operator) to allow the best several individuals to survive to the next generation. A simple filling operator is used to make sure the population size stays constant.

Result

The code runs as expected. The fitness of the best individual normally increases to 30 in about 50 generations, and the time for it to reach the target exactly varies from 100 to 1600 generations. The following is a snapshot of the output.

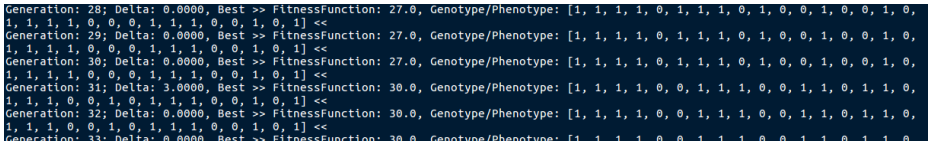

Fig. 8: Snapshot of output of test 1

5.2 Test 2

This test tries to replicate the experiments conducted in [9] and [10]. For the details of the experiments, readers are referred to [9] and [10]. Most of the components tested in the above are included in this test, with the following extra components:

Chromosome (Diploid), *DynamicHandler*, *FitnessFunction* (update method), *Frame* (change handling).

In this test, the genes used in [9] and [10] are implemented. As the environment needs to be changed regularly, the handler is required to handle the change.

Result

The code executes for both of the experiments. However, after the first change in environment, the population lost its ability to converge to the new optimum solution. This might due to the lost of diversity when the population converges to the optimal solution in the first environment. The following is a snapshot of the outputs.

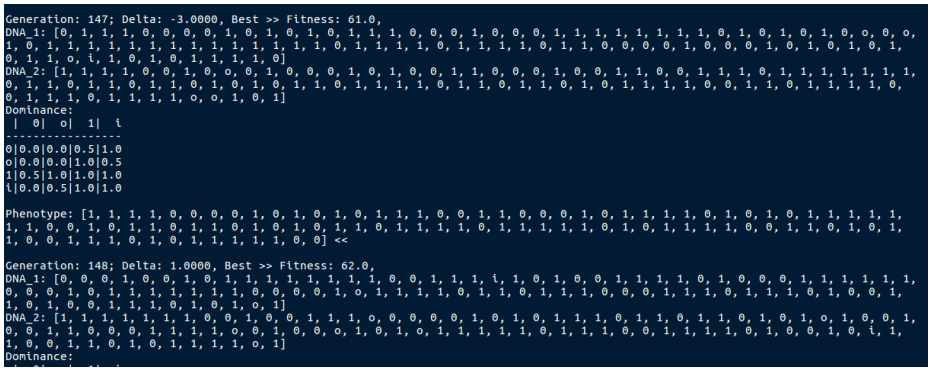

Fig. 9: Snapshot of output of experiment in [9]

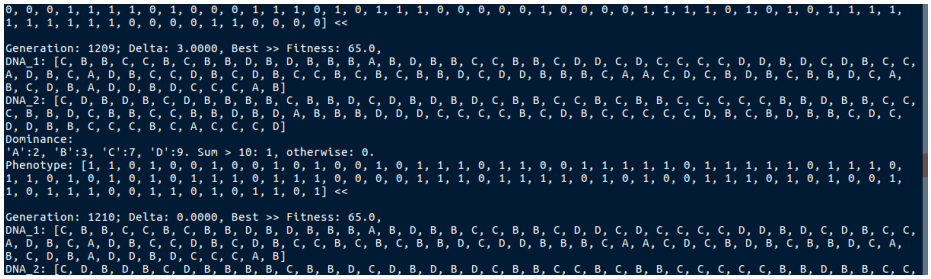

Fig. 10: Snapshot of output of experiment in [10]

5.3 Test 3

This test attempts to verify the validity of the "gender-based GA with recombination hotspots" extension module. The one-max problem is employ, where the binary encoded solutions try to achieve as much number of 1's as possible. Every components implemented in this module are included in this test.

As most of the classes used in this test are defined in the module (see Appendix), only the fitness function and initializer are implemented specifically for this test. The test uses a completely random map that chooses any gene in the corresponding position randomly. This mapping is completely useless in practice, but it provides an easy mapping for this test.

Result

The code executes with no errors and crashes. The unexpected outcome is that the solutions actually converge to the optimal one (a bit-string with all 1s). The following is a snapshot of the output.

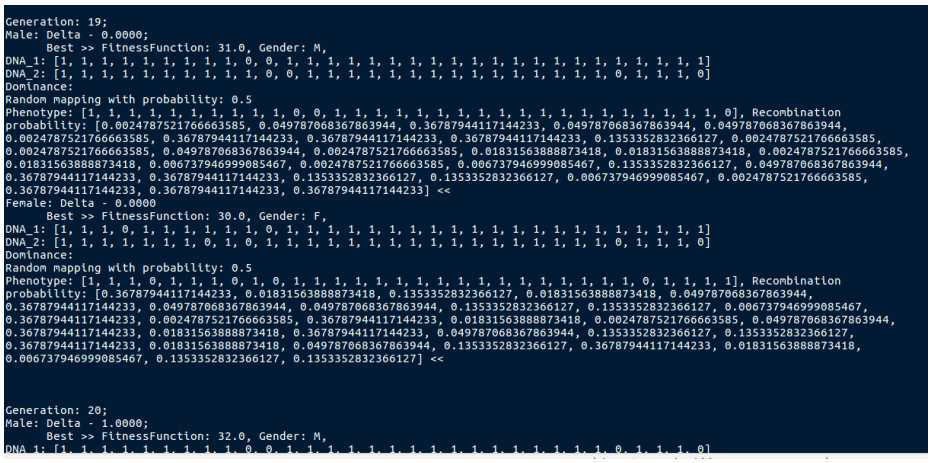

Fig. 11: Snapshot of output of one-max problem

6 Conclusion and Future Work

6.1 Conclusion

Genetic algorithms are developed to solve optimization problems by mimicking the natural selection process in biology. Although it is heavily utilized in industries, the current abstraction is still far from optimal. One of the most widely acknowledged problem is the scalability which significantly limits the potential

of GA. This indicates that there are some algorithmically important details in biology not being involved in the current model, which are not explicitly known at this stage. For this reason, further explorations on potentially critical algorithmic aspects from biology are desirable.

This project was supposed to be an extension to [1] before the existance of the work [7] that was published two weeks before this work had commenced. Due to the insufficiency of information provided in [7], repeated efforts to contact the author for more details were made, but no response was given. This project is then restructured into flexible code foundation development for further research in this area that minimizes efforts in experiment constructions.

The general process of how genetic algorithms work is briefly described. The candidate solutions of the optimization problems are encoded as chromosomes. The encoding is used in reproduction process that generates offsprings. The fitness function acts as a measure of suitability of an individual (chromosome) in a given environment. Individuals are selected according to some schemes that usually favours the one with greater fitness (in maximization problem) to participate in the reproduction process so that the traits or characteristics that fit the environment would be passed to the next generation, which eventually lead to convergence to the global optimum.

Two corresponding biological mechanisms, namely dominance and meiotic recombination, are presented as a comparison to the current model of genetic algorithms. We can see that the current model chooses the recombination locations uniformly randomly, but in biology the recombination is guided by evolved distributions of recombination rate at different locations. Another difference that is readily seen here is that the current chromosome decoding process included solely the complete dominance, the other two possible behaviours are ignored. It is suspect that those details are potentially algorithmically important, and in Larson et al.[7] it is shown that recombination hotspots do have a significance in evolvability of modular system. In light of this finding, we know that this direction is worth exploring.

The model is divided into the main module and an extension module. The main module is designed in a hierarchical architecture. At the top is the main loop used in the client code, which uses a *Frame* object to perform the evoluation for one generation in each iteration. The *Frame* consists of the current *State* and several operations that provide administrative supports. The *State* contains the current population and genetic operations for generating the next generation. The *Population* is a collection of individuals, that is candidate solutions to be optimized.

In this work, the extension module presented in this work provides modified classes specialized to gender-based genetic algorithms with recombination hotspots. The *Hotspot* and its corresponding mutation operator are defined in a similar way to how a *Chromosome* object is structured. The reproduction operator handles the gender-based reproduction with recombination hotspots, and several other classes, such as *State*, *Frame*, *Population* and so forth, are modified to handle the individuals from each gender separately.

The development is done in Java 7, due to the fact that its object-orientedness matches with the needs in genetic algorithms. Test experiments have been implemented as a guide and validation to the development. Roughly four prototypes are produced in the development. The first one defines the basic functionalities, the second prototype takes care of the genotype decoding and dynamic environment handling problem, the third one includes the extension module for gender-based GA with recombination hotspots, and the final version provides implementations to frequently used components.

The code validation is done iteratively through the development process, three test experiments are implemented. The first test experiment is a bit-matching problem that validates the basic functionalities of the library. The second test experiment is replicated from [9] and [10] for the valiation of dynamic environment handling and genotype decoding process. The last experiment is a simple one-max problem that provides a proof of concept of the extension module.

6.2 Future Works

First of all, there is a great room for modification of the main module. Secondly, for the convenience of potential experiments to be performed on this platform, the development of extra extension modules are desirable. As this project serves as a preparation for further exploration on genetic algorithms, the exploration of possible effective biological features to be included in the GA model is highly encouraged.

References

1. Antonsen, M., Fortescue, J. 2014. Approximating a solution for a moving target with recombination hotspots in genetic algorithms
2. Espinosa-Soto, C., Wagner., A. 2010 Specialization can drive the evolution of modularity. *PLoS Comput Biol*, 6(3):e1000719.
3. Wildman, D. E., Uddin, M., Liu, G., Grossman, L. I., Goodman, M. 2003. Implications of natural selection in shaping 99.4% nonsynonymous DNA identity between humans and chimpanzees: Enlarging genus Homo, *PNAS*, 100(12), pp. 71817188.
4. Auton, A., Fledel-Alon, A., Pfeifer, S., Venn, O., Sgurel, L., Street, T., Leffler, E. M., Bowden, R., Aneas, I., Broxholme, J., Humburg, P., Iqbal, Z., Lunter, G., Maller, J., Hernandez, R. D., Melton, C., Venkat, A., Nobrega, M. A., Bontrop, R., Myers, S., Donnelly, P., Przeworski, M., McVean, G. 2012. A Fine-Scale Chimpanzee Genetic Map from Population Sequencing, *Science* 336(6078), pp. 193-198.
5. Chowdhury, R., Bois, P. R. J., Feingold, E., Sherman, S. L., Cheung, V. G. 2009. Genetic Analysis of Variation in Human Meiotic Recombination, *PLoS Genetics*, 5(9).
6. Myers, S., Bottolo, L., Freeman, C., McVean, G., Donnelly, P. 2005. A Fine-Scale Map of Recombination Rates and Hotspots Across the Human Genome, *Science*, 310(5746), pp. 321-324.

7. Larson, A., Bernatskiy, A., Cappelle, C., Livingston, K., Livingston, N., Long, J., Schwarz, J., Smith, M., Bongard, J. C. 2016. Recombination Hotspots Promote the Evolvability of Modular Systems, *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, ACM, New York, USA, pp. 115-116.

8. Sima Etaner-Uyar, A. and Emre Harmanci, A. 2002. Preserving Diversity through Diploidy and Meiosis for Improved Genetic Algorithm Performance in Dynamic Environments, *Advances in Information Systems: Second International Conference, ADVIS 2002 Izmir, Turkey, October 2325, 2002 Proceedings*, 2457, Springer Berlin Heidelberg, pp. 314-323.

9. Ng, K. P. and Wong K. C. 1995. A new diploid scheme and dominance change mechanism for non-stationary function optimization. *Proc. Int. Conf. Genetic Algorithms*, Pittsburgh, PA, pp. 159-166.

10. Lewis, J., Hart, E., and Ritchie, G. 1998. A comparison of dominance mechanisms and simple mutation on non-stationary problems. *Proc. Parallel Problem Solving From Nature*, Amsterdam, The Netherlands, pp. 139-148.

11. Yang, S. 2006. Dominance Learning in Diploid Genetic Algorithms for Dynamic Optimization Problems, *Proceedings of the 2006 on Genetic and Evolutionary Computation Conference Companion*, ACM, New York, USA, pp. 1435-1436.

12. Rahnamayan, S., Tizhoosh, H. R., Salama, M. M. A. 2006. A novel population initialization method for accelerating evolutionary algorithms, *Computers and Mathematics with Applications*, Elsevier, 53(10), pp. 1605-1614.

13. Kumar, R., Narula, S., Kumar, R. 2013. A Population Initialization Method by Memetic Algorithm, *International Journal of Advanced Research in Computer Science and Software Engineering*, 3(4), pp. 519-523.

14. Lima, J. A., Gracias, N., Pereira, H., Rosa, A. 1996. Fitness Function Design for Genetic Algorithms in Cost Evaluation Based Problems, *Proceedings of IEEE International Conference on Evolutionary Computation, 1996.*

15. Ward, K., McCarthy, T. J., 2006. Fitness Evaluation for Structural Optimization Genetic Algorithms Using Neural Networks, *International Conference on Engineering Computational Technology*, Civil Comp Press Ltd, Stirling, UK, pp. 1-11.

16. Wu, Y. G., Ho, C. Y., Wang, D. Y. 2000. A Diploid Genetic Approach to Short-Term Scheduling of Hydro-Thermal System, *IEEE Transactions on Power Systems*, 15(4), pp. 1268-1274.

17. Uyar, A. . and Harmanci, A. E. 2005. A new population based adaptive domination change mechanism for diploid genetic algorithms in dynamic environments, *Soft Computing*, 9(11), pp. 803-814.

7 Appendix

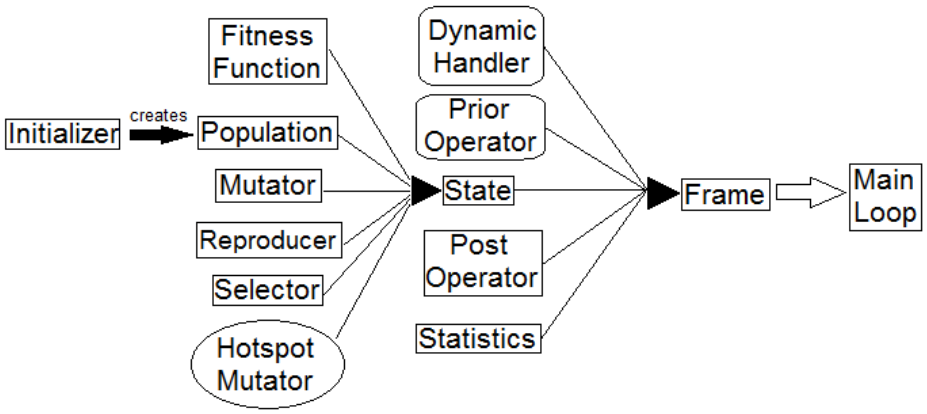

Fig. 12: General flow of main method

7.1 Brief Implementation Guide

Before conducting the experiment, one has to define the required operations in the process. Some simple classes are provided in the library (see the section below). One may define your own classes if the pre-defined ones do not meet the experiment requirement. Every self-defined components has to implement the corresponding interface in the library to be able to fit in the constructors of *State* and/or *Frame*. A general guide for the main method is illustrated in **Fig.6**. After defining all necessary classes, the instantiation goes from left to right. Note that the initial population has to be save manually by calling "statistics.record(state)", and at the end if a log file is required one can call "statistics.save(filename)" to save the content. The following code from the first validation experiment provides an example for the main method.

```
public class Exp1Main {

    private static final int target = 0xf71b72e5;
    private static final int size = 200;
    private static final int maxGen = 2000;
    private static final int numElites = 20;
    private static final double mutationRate = 0.05;
    private static final double crossoverRate = .8;
    private static final double epsilon = .5;
    private static final double maxFit = 32;
    private static final String outfile = "Exp1.out";

    public static void main(String[] args) {
        // Fitness Function
        FitnessFunction fitnessFunction = new Exp1FitnessFunction(target);
        // It is not necessary to write an initializer, but doing so is
        // convenient to repeat the experiment using different parameter.
        Initializer<SimpleHaploid> initializer = new BinarySimpleHaploidInitializer(size, 32);
        // Population
        Population<SimpleHaploid> population = initializer.initialize();
```

```
        // Mutator for chromosomes
        Mutator mutator = new Exp1Mutator(mutationRate);
        // Selector for reproduction
        Selector<SimpleHaploid> selector = new SimpleProportionalSelector<>();
        // PriorOperator is optional.
        PriorOperator<SimpleHaploid> priorOperator = new Exp1PriorOperator(numElites, selector);
        // PostOperator is required to fill up the vacancy.
        PostOperator<SimpleHaploid> postOperator = new SimpleFillingOperatorForNormalizable<>(
                                                   new ProportionalScheme());
        // Statistics for keeping track the performance in generations
        Statistics<SimpleHaploid> statistics = new Exp1Statistics();
        // Reproducer for reproduction
        Reproducer<SimpleHaploid> reproducer = new Exp1Reproducer();

        State<SimpleHaploid> state = new SimpleState<>(population,
                                                   fitnessFunction, mutator,
                                                   reproducer, selector, 2, crossoverRate);
        state.record(statistics);
        Frame<SimpleHaploid> frame = new SimpleFrame<>(state,postOperator,statistics);
        frame.setPriorOperator(priorOperator);
        statistics.print(0);
        for (int i = 1; i <= maxGen; i++) {
            frame.evolve();
            statistics.print(i);
            if (statistics.getOptimum(i) > maxFit - epsilon)
                break;
        }
        statistics.save(outfile);
    }
}
```

7.2 Simple Implementation Provided in the Library

7.2.1 Main Module

7.2.1.1 Components

7.2.1.1.1 Gene

BinaryGene: A *Gene* that stores only 0 or 1.

7.2.1.1.2 GeneFactory

BinaryGeneFactory: Generates a random *BinaryGene*.

7.2.1.1.3 Material

SimpleMaterial: Uses an fixed length array to store genes.

7.2.1.1.4 Chromosome

SimpleHaploid: Contains a single *SimpleMaterial* as genotype, uses the identity map as dominance map.

SimpleDiploid: Contains exactly two *SimpleMaterial* as genotype, the phenotype is also a *SimpleMaterial*.

7.2.1.2 Collections

Individual: Contains a chromosome and a fitness value, implements *Comparable* for sorting purpose according to fitness value.

Population: Contains one *List* for individuals in the current generation, and three *List*s for the upcoming generation.

PopulationMode: Specifies which pool a newly added individual should be placed in the *Population*.

SimpleElitesStatistics: Records only the best individual from each generation.

7.2.1.3 Operations

7.2.1.3.1 Expression Maps

ProjectionMap: Chooses one of the materials in the genotype to be the phenotype.

SimpleDiploidRandomMap: Chooses a random gene at each location from the genotype to be included in the phenotype.

7.2.1.3.2 Initializers

BinarySimpleHaploidInitializer: Initializes a random population of *SimpleHaploid* objects that use *BinaryGene*.

7.2.1.3.3 Mutators

BinaryGeneMutator: Flips a *BinaryGene* value under a user-defined probability.

7.2.1.3.4 Prior Operators

SimpleElitismOperator: Chooses the best individuals to survive to the next generation.

7.2.1.3.5 Post Operators

SimpleFillingOperator: Chooses individuals to survive to the next generation by some given selection scheme.

SimpleFillingOperatorForNormalizable: Chooses individuals to survive to the next generation by proportional selection scheme. This requires the fitness values to be non-negative.

7.2.1.3.6 Reproducers

SimpleHaploidReproducer, **SimpleDiploidReproducer**: Generate offspring by uniform crossover on *SimpleHaploid* and *SimpleDiploid* respectively.

SimpleHaploidNPReproducer, **SimpleDiploidNPReproducer**: Generate offspring by N-point crossover on *SimpleHaploid* and *SimpleDiploid* respectively.

7.2.1.3.7 Selection Operators

7.2.1.3.7.1 Selection Schemes

ProportionalScheme: Selects individual using the proportional selection scheme. The fitness function must be non-negative.

TournamentScheme: Selects individual using the tournament selection scheme.

7.2.1.3.7.2 Selectors

SimplePropotionalSelector: Normalizes the fitness values, then performs parents selection using *ProportionalScheme*.

SimpleTournamentSelector: Performs parents selection using *TournamentScheme*.

7.2.1.4 Frame

SimpleState: Implements a reproduction-rate-based reproduction subroutine. Mutator runs through all the individuals in the offspring pool.

SimpleFrame: Implements a basic evoluation step: (handle dynamic change if any), prior operation → reproduction + mutation → post operation → proceed to next generation → evaluate fitness value → record state to statistics.

7.2.2 Gender-based GA with Recombination Hotspots

7.2.2.1 Components

7.2.2.1.1 Chromosomes

SimpleGenderDiploid: A coupleable version of *SimpleDiploid*.

7.2.2.1.2 Hotspots

DiscreteExpHotspot: Uses the exponential probability distribution with integer encoding.

SimpleHotspot: The encoding is the recombination rate itself.

7.2.2.2 Collections

GenderPopulation: Forces chromosomes to be coupleable, and maintains a user-defined gender proportion in every generation.

SimpleGenderElitesStatistics: It is the same as *SimpleElitesStatistics* but having a dedicated record for each gender.

7.2.2.3 Frame

SimpleGenderState: It is the same as *SimpleState*, except that it forces everything to be compatible with gender-based GA with recombination hotspots.

SimpleGenderFrame: Forces the state to be compatible with gender-based GA with recombination hotspots, and a new "mutateHotspots" method call is inserted after mutation and before advancing to the next generation.

7.2.2.4 Operations

7.2.2.4.1 HotspotMutators

SimpleDiscreteExpHotspotMutator: Mutates the *SimpleDiscreteExpHotspot* randomly by increasing or decreasing the encoding by 1.

7.2.2.4.2 Prior Operators

SimpleGenderElitismOperator: Chooses the same amount of the best individuals from each gender to survive to the next generation.

7.2.2.4.3 Reproducers

SimpleGenderDiploidReproducer: Reproduces offspring by guided swapping of materials and combination of random choice of gametes from each of the parents.

7.2.2.4.4 Selectors

SimpleTournamentCoupleSelector: Selects a pair of male and female individuals as parents separately by tournament selection.