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. To be included

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

1 Introduction

To be included

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). GA has been applied successfully in industries to solve multi-dimensional optimization problems, such as drug creation, stock trading, waste material minimization, et cetera. Despite its huge success in industries and the simplicity of the principle behind, the development of genetic algorithm has recently encountered resistance. While the reason is not explicitly known, it is widely accepted that the inscalability of genetic algorithm should be one of the contributing factor to its slowed pace of advance. It is clear that biological evoluation is way more effective than the current computational evoluationary systems, this suggests that our simulated system may have missed some important details in the nature that can account for the difference in effectiveness. For this reason, further investigations on the potential internal mechanisms in genetic algorithms are necessary.

One of the differences between genetic algorithm and evoluationary biology lies in the recombination process. In traditional genetic algorithm, the recombination rate of a gene is independent of its location. However, this is far from the case in evolutionary biology, where the recombination rate is shown to be highly dependent upon the locus, which leads a huge difference in behaviour of individuals. A well-known example in biology is the genetic similarity between humans and chimpanzees. It is shown in the work of Derek E. Wildman et al.[1] that humans and chimpanzees share 99.4% identity at nonsynonymous (functionally important) sites and 98.4% at synonymous (functionally less important) sites. However, in a recent research by Adam Auton et al.[2], 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. Furthermore, the distribution of recombination rates between male and female is shown to be different[3] that about 15% of hotspots in one sex are specific to that sex[4]. Based on the discovery above, it is suspected that recombination hotspots may be algorithmically important, and their sex-specificity suggests that the sex of an individual may have a significant contribution in the evolution process.

The potential significance of recombination hotspots on genetic algorithms is comfirmed in a recent work by Ari Larson et al.[5]. 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.

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, etc. It is shown that the diploid chromosome is capable to preserve diversity of the population in evolution[6], and the dominance schemes are crucial for adaptability in dynamic environment[7][8][9].

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[10] and hill-climbing[11] 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 [12], several scaling schemes are presented attempting to convert the function into one with better precision and behaviour. A neural network approach is proposed in [13] 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[][] provided an illustration of this process. In [14], recombination occurs before mating, while in [15] 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 Dominance Map

The dominance 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 dominance map is required to decide the outcoming gene value in the phenotype. The dominance map can be changed according to the progress of GA. In [7] and [8], 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

Initialize population
Evaluate population
While (stopping criteria is not reached):
Do
    Generate next population
    Perform mutation on next population
    Population <- next population
    Evaluate population
End
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 [7], [8], [15], [9], the process consists of a change in dominance based on individuals' performance and the environment. In [14], 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.

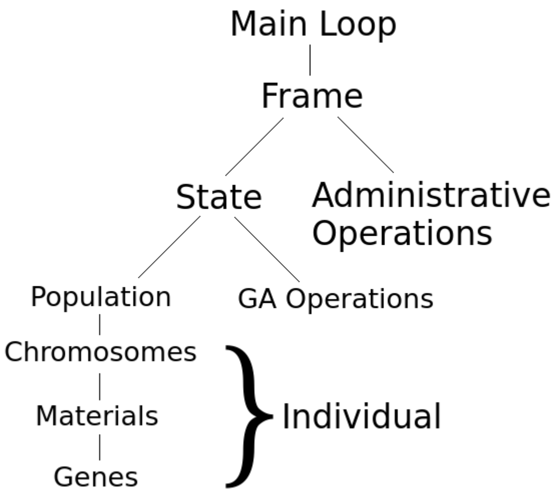

Fig. 1: Overall Architecture

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.

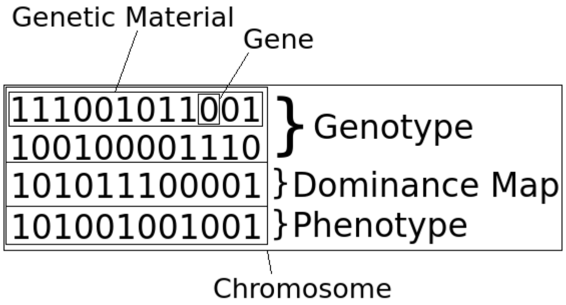

Fig. 2: Hierarchy of Chromosome Composition

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 *Dominance 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.

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

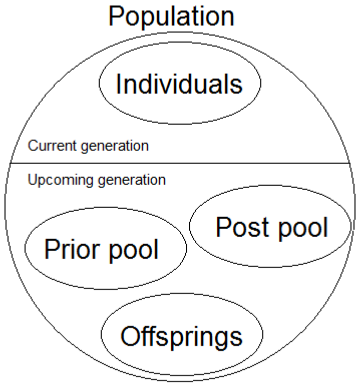

Fig. 3: Composition of Population

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.

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

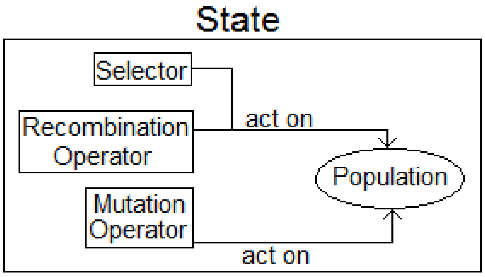

Fig. 4: Composition of State

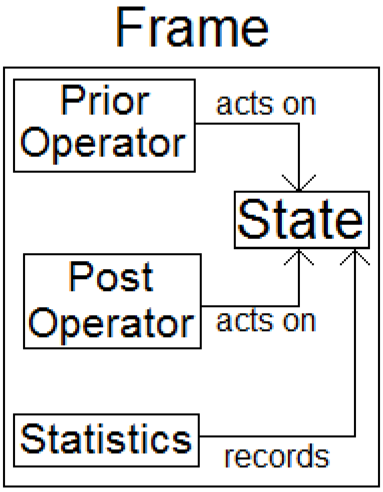

Fig. 5: Composition of Frame

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 Dominance Map

This interface provides an abstraction to the dominance map 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 Implementation Example

To be included

5 Conclusion and Future Work

To be included

References

1. 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.
2. 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.
3. 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).
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.*
13. 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.

14. 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.
15. 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.