Code Sequence Generation with Genetic Algorithms, with Correlation Properties Similar to GPS C/A Codes

Hovhannes, Gomtsyan
NPUA
Yerevan, Armenia
e-mail: hovhannes.gomcyan@politechnic.am

Robert, Apikyan
NPUA
Yerevan, Armenia
e-mail: apikyan41@gmail.com

ABSTRACT

C/A codes (course acquisition codes) are pseudo random generated codes with good correlation properties. Those codes are being used in GPS. Each satellite vehicle can generate its unique C/A code sequence and modulate it with output data signal. Each millisecond of satellite data contains 1024 chips (bits) of C/A codes and each 1ms this code sequences are being repeated. Receivers are using locally generated C/A codes in order to filter out the satellite signal from aggregated signals near the receiver's antenna. As the C/A codes are being repeated in each 1ms, in theory it's enough of 1ms satellite signal in order to determine from which satellite it is coming. C/A code's correlation properties are being used for filtering incoming signals. The higher is the autocorrelation properties of the code sequence, the easier to filter it out from summary signal. In other words, the same C/A codes have high correlation values and different C/A codes have low correlation values [1]. The target of this article is to write a program using genetic algorithms [2] that will generate code sequences from 1 and -1 values that will have nearly the same correlation properties as the C/A code, where each individual in algorithm will contain 32 number of code sequences with 1024 length that has low cross correlation and high autocorrelation properties.

Keywords

C/A code, pseudo random codes, GPS, signal correlation, genetic algorithms, Java.

1. INTRODUCTION

The program that will generate *n* number of different codes with *l* length is based on genetic algorithms, where each individual contains an array of *n* * *l* length. The program will be written in Java programming language. For correlation functions and local C/A code generation we will use "GPSToolkit" [9] library.

In Genetic algorithms the individual is an abstract representation of a solution, and it contains the fitness value for that solution, typically fitness values are the numbers between 0 and 1, and where 1 is the best solution and the 0 is the worst. Group of individuals called a population, in other words population is a holder for solutions array. Crossover and mutation could be applied to population. Genetic algorithms could be divided into 6 abstract steps, as shown in Figure 1.

First step is called "Population Initialization", where the individuals are created. Usually they are created based on random behavior during the first initialization process. After initialization the fitness calculation function will be applied to population, for defining the individual's fitness for the required solution. The important step is "Termination condition check". In this step we are searching the individual that will fit our condition. Usually this is not happening for the first time with randomly generated populations. The genetic algorithm will apply crossover and mutation to population's individuals, while termination condition is not giving a positive result, the "Selection" step is a part of "Crossover". For crossover we need two individuals that have high fitness values. There are a number of algorithms for implementing selection and crossover. The "Mutation" is an important step as well. Genetic algorithm that is working with only crossover function will stuck at some point, because it will not have a chance to evolve without mutation function. After applying crossover and mutation to population, the algorithm will calculate the individual's fitness values and will check these values if they fit to the termination condition [2].

In upcoming sections, we will discuss the implementation of genetic algorithms steps that are going to produce 32 number of codes with length 1024 that have low cross correlation and high autocorrelation properties. In other words, these codes will maximally differ from each other.

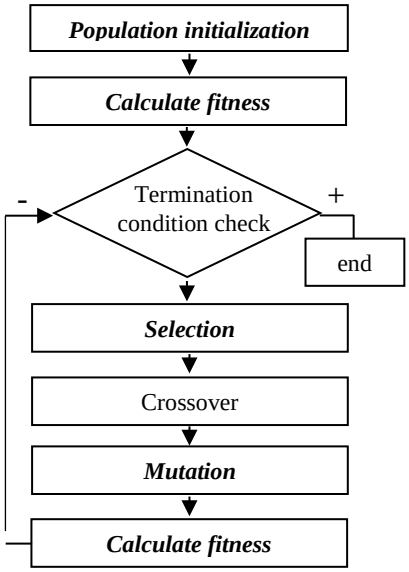

Figure 1. Genetic algorithm's block diagram

1.1. Individual

The first step for implementing the genetic algorithm is to define individual's model. In our case a single individual will contain an array with 32 * 1024 bits and with a fitness value. From 0-1023 bits in this array are codes for the first satellite, from 1024 – 2047 are codes for the second satellite and exc.

The possible values for bit in array are 1 and -1. After the first initialization, each individual will randomly generate and fill array with 1 and -1. You can find the Java model of individual in [4] directory.

1.2. Fitness Calculation

As mentioned earlier each individual contains 32*1024 bits (1 and –1), where the first 1024 bits are intended for the first satellite, the next chunk of 1024 bits (from 1024 - 2047) are intended for the second satellite and exc. While calculating the function of fitness we take the first satellite's bits (0-1023) and correlate them with other satellite's bits iterating through each 1024 bits as shown in Figure 2. As we can see fitness function implementation has two loops [6]. First loop's index is i, its range is from 0 to 32*1024, and it's incrementing with 1024 on each loop. The similar loop is designed for second iteration with j. Here we take satellite[i] and calculate its correlation [8] value with other satellites with index j. As you can see *fitnessValue = fitnessValue + (1 – correlationValue)*. The key point here is *1 – correlationValue* , the reason for this is that we search codes that are mostly differ from each other. The *correlationValue* shows how similar are two codes, so by subtracting it from 1 we will receive the difference value. In the code of fitness function [6], you can see that after each iteration *fitnessValue* is divided by codes count in individual, this is for receiving an average value between -1 and 1.

- If individual has fitness that is closer to -1, it means that its codes are mostly familiar to each other,
- If individual has fitness value closer to 1, it means that its codes mostly differ from each other.

We will iterate through individuals and pick the one that has higher values, while checking for termination condition in the next section.

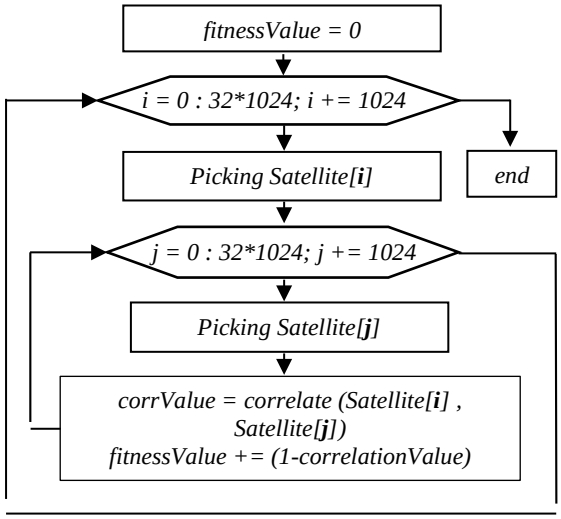

Figure 2. Fitness calculation function's block diagram

1.3. Termination Condition

In this step of algorithm, we will check as name explains the termination, in other words when to stop our genetic algorithm. We can implement this function in different ways, the simplest implementation could be, to check the higher fittest value in population, whenever it is higher from some nominal value, we will terminate the algorithm. But with this approach it is easy to stick in local optimum solution. Let's say that the fittest value is 0.85 in population and the nominal fitness value we have chosen is 0.8. The termination condition will work, since the fittest value is higher than required nominal value, but in upcoming evolution, population could evolve and give us higher fitness values than 0.85. To overcome this problem, we will not be using nominal fitness value, instead we will add two variables ***currentEffectiveGenerationFitness*** and ***currentEffectiveGenerationCounter***.

- ***currentEffectiveGenerationFitness****:* This variable will remember the highest fitness value in population in each termination condition check.
- ***currentEffectiveGenerationCounter****:* Initially some high value (like 1000 or 2000) will be assigned to this variable. While every generation check, if the fitness value is not changed this variable's value will be decremented and checked for zero equality, if its value is equal to zero, then algorithm will be terminated, in other hand if the fitness value is changed the variable's value will be assigned to its initial value. For example, if its initial value is equal to 1000, then genetic algorithm will terminate whenever the fitness value is not changed in past 1000 generations [6].

1.4. Selection and Crossover

This is one of the important points in genetic algorithms. In our case, we need to make a crossover of two individuals where each individual contains 32 * 1024 codes. For making a crossover, at first we need to pick two parents from population [5]. Important point here is to pick parents that have higher fitness values. For that we are going to sort the existing individuals in population by higher fitness value in descending order. By iterating through sorted individuals we well pick the first parent as shown in Figure 3. The second parent will be selected with roulette selection algorithm [6]. In short, with roulette selection algorithm individuals with higher fitness value have more chances to be picked, then individuals with lower fitness s value.

Crossover rate parameter will be used in order to add randomness in applying crossover to individual. The random number will be generated between [0,1), if it is smaller than crossover rate value, crossover will be applied to individual, otherwise it will be moved to new population without crossover. After picking two parents we make a crossover and create a new individual. While crossover parent that has higher fitness value will provide more bits than other parent.

Also for crossover we will use elitism count. If elitism count is two, then the first two individuals will be inserted to a new population without crossover. This approach allows to always keep individuals with higher fitness values.

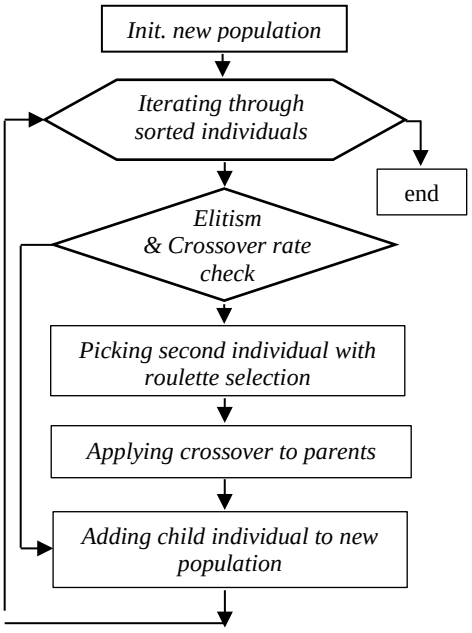

Figure 3. Crossover function's block diagram

1.5. Mutation.

Mutation is applied to each individual in population. This step allows to randomly change the individuals in population. Important parameter here is the mutation rate, which is usually a small number. Bits in individual will be mutated depending on the mutation rate value. While mutation, we will generate a random number between [0,1) if it is smaller than mutation rate value, mutation will be applied to individual's bit, if value is higher than mutation rate it will still be unchanged as shown in Figure 4.

As in case of crossover, elitism count will be used here as well, in order to keep the individuals with higher fitness values without mutation [6].

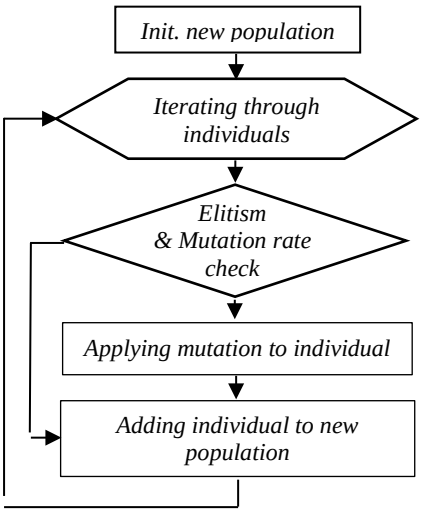

Figure 4. Mutation function's block diagram

2. CODE GENERATION PROGRAM

Based on the discussions in the last sections we can now generate C/A code sequences based on genetic algorithms. For testing first we need to run "DiffCodes.java"class [9]. Here we can specify mutation rate, crossover rate, elitism count, population size and effective generation count. After specifying these parameters we can run the program. The program will automatically stop when the required fitness value will be achieved and 32 code sequences for each satellite will be written on local disc in .txt format, so we can compare generated code sequences with real generated C/A codes. For generating C/A codes based on LFSRs we are going to use "GPSGenerator" library [8]. After generating both kinds of code sequences we can see that they have nearly the same high autocorrelation and low cross correlation values.

3. CONCLUSION

As a result, we will have code generation program that is based on genetic algorithms which will generate 32 codes for each GPS satellites with nearly same parameters as originally generated C/A codes with LFSR algorithms.

The program is available in GitHub [3] repository. We can also compare the generated GPS C/A codes [8] with codes that are generated with our program with Correlation library, which is also available on GitHub [7].

REFERENCES

[1] James Bao-Yen Tsui, "Fundaments of Global Positioning System Receivers: A Software Approach", A Wiley Interscience publication, year 2000.

[2] Lee Jacobson, Burak Kanber, "Genetic Algorithms in Java Basics", Apress publishing house, year 2015.

[3] Project reference in GitHub repository - https://github.com/RobertApikyan/CodeGenWithGeneticAlgorithm

[4] Individual reference in GitHub repository - https://github.com/RobertApikyan/CodeGenWithGeneticAlgorithm/blob/master/src/diffCodes/CodeIndividual.java

[5] Population reference in GitHub repository - https://github.com/RobertApikyan/CodeGenWithGeneticAlgorithm/blob/master/src/diffCodes/CodesPopulation.java

[6] Termination condition, fitness, crossover, mutation, functions implementation reference - https://github.com/RobertApikyan/CodeGenWithGeneticAlgorithm/blob/master/src/diffCodes/GeneticAlgorithm.java

[7] GpsToolkit program's reference - https://github.com/RobertApikyan/GPSToolkit

[8] GpsGenerator program's reference - https://github.com/RobertApikyan/GpsGenerator

[9] CodeDiff.java class reference - https://github.com/RobertApikyan/CodeGenWithGeneticAlgorithm/blob/master/src/diffCodes/DiffCodes.java