

Spring 2004  
ECE734 Project  
Implementation of Generic Systolic  
Array for Genetic Algorithm

Liang-Kai Wang  
Advisor: Professor Hu

## **Abstract**

Genetic Algorithms are search mechanisms which employ the principles of natural selection and mutation to develop solutions to a various search and optimization problems that are commonly seen in several fields including artificial intelligence and Computer Aided Design. However, due to a big set of data, developing a local optimized solution usually takes a large amount of time.

This project focuses on the implementation of the genetic algorithm back to the biological evolution. The genetic algorithm from the biological evolution exhibits a big amount of serialization which cannot be applied to the systolic array. However, due to the implicit parallelism characteristics of genetic algorithm, we then can transform the equations to single assignment equation, localized variable broadcasting, and systolic array for high-speed parallel computing.

In this project, Java is used to implement the whole algorithm. The goal is to have a complete user-defined fitness program that can efficiently and quickly compute the result. However, in this project, fitness functions are predefined and a final statistic will be reported over different functions.

## **1. Introduction**

Genetic algorithms (GA) are popular in the field of developing solutions to a wide range of search and optimization problem. They perform actions iteratively until an acceptable result is found. GAs were first demonstrated by Holland [1]. In their design, they handle a group of candidate solutions called chromosomes, each of which are constructed from a series of units called genes. The fitness value of each solution is then computed through user-defined fitness functions which are qualitative measure of how good a chromosome or the whole environment is as a solution to the problem.

Although in some cases, selection is random, fitness value in each chromosome is usually used to bias the selection. After selection is made, crossover is performed by combining sections of two selected parents. The simplest method to perform crossover is to mimic the action of human beings. When an egg is fertilized, the fertilized egg will receive half of chromosomes from its dad and the other half from its mom. In practice,

more elaborate crossover mechanisms are often used. Mutation performs on individual genes by randomly selecting or conditionally setting a bar value and flip the gene value if the fitness value is less than the bar value.

## 2. Algorithm

### 2.1 Flow of Genetic Algorithm

Figure 1 shows the flow of genetic algorithm. First of all, we need to do an initialization step to construct the whole environment. This includes defining the following items:

1. Setup of the seed of the random number generator.
2. The number of person in the environment (E.g. 100 in the design)
3. The number of chromosome in a person (E.g. 23 pairs. Each pairs has chromosome X and chromosome Y)
4. The number of gene in a chromosome (E.g.10 in the design)
5. Construction of the bad gene/chromosome list

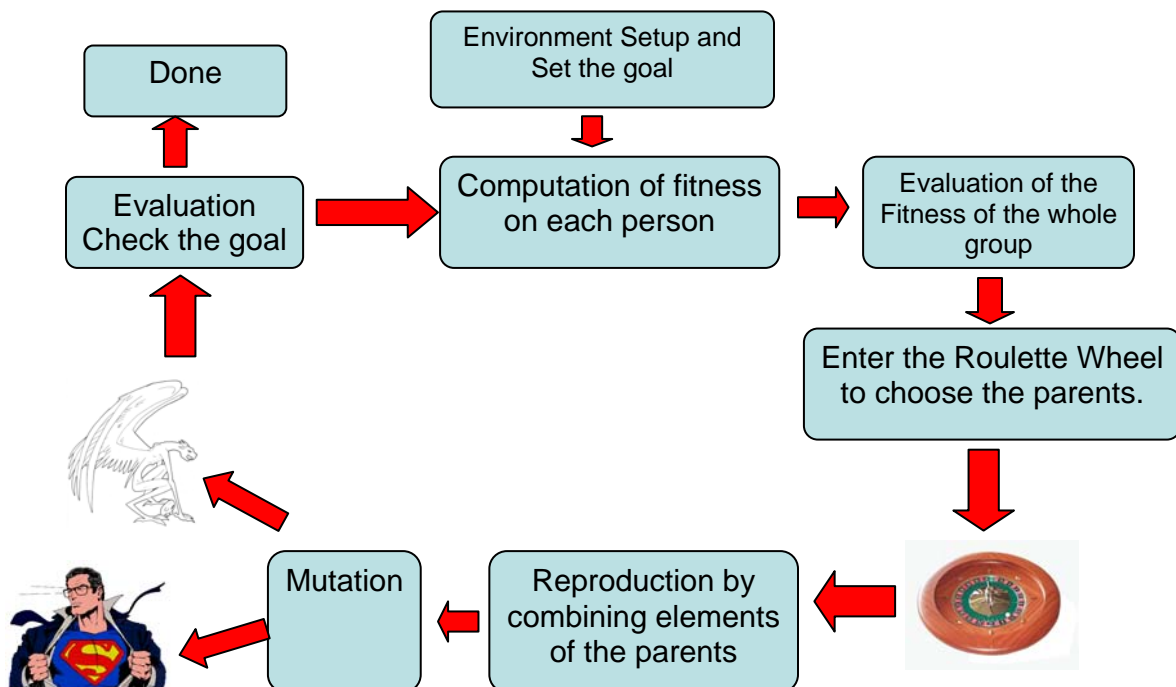


Figure 1 Flow of Genetic Algorithm

All the above items are self-explained except the last item. In my definition, we need to have a bad gene/chromosome list so that the fitness value of a person can be

easily computed. Conventionally, the bad gene/chromosome list can be made by referring to genetic diseases and a specific weight can be assigned to each bad element. For example, in terms of cancers which potentially excited by bad genes, we have Melanoma, Prostate cancer, colon cancer, breast and ovarian cancers, each of which are from different positions among the sea of chromosomes. Researcher may want to weight more for Melanoma cancer because it decreases more chance for a person to have descendent, which follows the basic concept of *Natural Selection*.

In my definition, I equally weight the effect from all the bad genes in the bad gene list since the weighted sum is just a variation of normal sum and does not make a significant difference on the structure of the systolic array.

The next block in the flow is the computation of personal fitness function and environmental fitness function. As usual, fitness functions can be defined in various ways. For example, in [3], the authors define the personal fitness function as the accumulation of the fitness value of each chromosome in a person. However, I believe the fitness value should include the effect of bad genes; therefore, I define the fitness function as the conditional sum of the genes among the sea of chromosome. For more details, please refer to the Section 2.2.

Selection also has various implementations; however, most of them roots from the same principle, the higher fitness value a person has, the better chance this person have to get selected to produce a descendent. In order to best utilize the concept of systolic array, I set a bar value randomly during each round of the selection. Whenever the selector finds a good candidate, it first checks if we already have the candidate in our selection list, if not, then it includes this new comer to the selection list. Then it regenerates the random number again and scans through the group of population to search the next candidate.

In crossover stage, selected persons are asked to exchange their chromosomes. In [2-4], the authors change the genes in a chromosome, which is not the normal way we know in the true crossover. In the implementation, I exchange two candidates' X and Y chromosomes so that their descendents' chromosomes are the mixture of candidate A and candidate B, which is closer to the true mechanism. In addition, I make an assumption that the parents die immediately after generating their two children. Although this may not be realistic, it fully utilizes the systolic array and reduces the complexity. Moreover, I

believe some more modification can be made to facilitate those complex designs in the future.

Mutation is the last step in the loop of the genetic algorithm. In order to mimic the real world, a mutation list is created. Based on the mutation list, a randomly generated bar value, and the fitness value in each person, we can determine if the specific gene should reverse. In other word, if the fitness value of a person is less than the bar value, the gene specified by the mutation list should be flipped.

The number of loop of the genetic algorithm is determined by researchers. For example, in the project, I set 50, 100, 200, and 400 to see the variation of the environmental fitness value.

## 2.2 Fitness Functions

As was mentioned above, we compute the conditional sum of the genes among the sea of chromosome. Besides, in order to limit the final fitness value of a person in the range of -1 to 1, every gene in a person contributes the same amount of fitness value, which

is  $\frac{1}{\text{Total Number of Gene} \times \text{Total Number of Chromosome}}$ . The pseudo code of the

personal fitness function is shown below:

```
personalFit (int sizeOfPerson, int productOfChrGene) {
    step =  $\frac{1}{\text{Total Number of Gene} \times \text{Total Number of Chromosome}}$ 
    for (int i=0; i<sizeOfPerson, i++) {
        for (int j=0; j<productOfChrGene; j++) {
            if (checkBadGene(person[i], j%numberOfChromosome,
                j/numberOfGene) == 1)
                person[i].fit = person[i].fit -step;
            else
                person[i].fit = person[i].fit +step;
        }
    }
}
```

```

int checkBadGene(Person person, int geneNumber, int chromosomeNumber) {
    while ( !endOfBadlist)
        if ((geneNumber==geneAtBadList) &&
            (chromosomeNumber == chromosomeAtBadList)) {
            if (person  $\rightarrow$ chromosomeNumber  $\rightarrow$ geneNumber ==0)
                return 1;
        }
    }
    return 0;
}

```

It is easily to see that it reveals some potential parallelism in the code. In Section 3.1, we would demonstrate the possible implementation in the systolic array.

### 2.3 Selection

As we mentioned in Section 2.1, natural selection can be done by initializing a random number as a bar, passing this bar value down, selecting a candidate based on the bar value, checking for repetition if there is a match and then attaching this candidate to the list. Although using this method is quite straightforward, it may bring some problems if most of the fitness value in the group is similar, in which case the selection sequence may be the same and limited to the scope of search order. In order to increase the uncertainty, two other ways are proposed. The first method relies on subtraction of the bar value along the way down the group. Whenever the bar value is less than zero, the person gets selected. By using this method, the structure is less modified and still can apply the systolic array because only one extra subtraction is added in each process element.

On the other hand, random selection can be used and the selection will not depend on the fitness value in each person. Although this method increases uncertainty and is potentially more similar to the real world, it is not easy (or not quite efficient) to implement this in systolic array. Therefore, I only implement this for reference. The pseudo code of the personal fitness function is shown below:

```

Selection(int sizeOfPerson) {
    for (int j=0; j<selection, j++) {
        rand = new Random();
    }
}

```

```

    for (int i=0; i<sizeOfPerson; i++) {
        rand = rand - person[i].fit;
        if (rand<0) {
            if (checkRepitition(i) == 0);
                selection[j] = i;
            }
        }
    }
}

```

```

checkRepitition(int i) {
    int r=0;
    while (endOfSelectionArray) {
        if (selection[r]==i)
            return 1;
        r++;
    }
    return 0;
}

```

More details about how to translate this into systolic array can be seen in Section 3.2.

## 2.4 Crossover

To my best knowledge, crossover is the only part that may not be able to do systolic array since it is not easy for a person to release half of chromosomes on the fly and at the same time wait for the other half to generate a new descendent. For completeness, I list the pseudo code of the crossover for reference.

```

Crossover(int numberOfSelection) {
    for (int i=0; i<numberOfSelection/2; i = i+2) {
        for (int j=0; j<numberOfChr; j++) {
            temp = person[i].chromosome_X[j];
            person[i].chromosome_X[j] = person[i+1].chromosome_Y[j];
            person[i+1].chromosome_Y[j] = temp;
        }
    }
}

```

```
}
```

## 2.5 Mutation

Mutation plays an important role in natural selection. If there is no mutation, some new features cannot occur although some of them are not good. The operation of mutation is not hard, and can be extended to systolic array as well. As was described in Section 2.1, we maintain a mutation list and one or several randomly generated bar values. Although we can subtract the bar value as describe in Section 2.3, it makes more sense to use it as is, which means we forward the random value and do comparison to the current fitness value rather than subtracting the value. Once the bar value is greater than the fitness value of a person, all the genes listed in the mutation list should be reversed. The pseudo code of the personal fitness function is shown below:

```
mutation (double bar; int sizeOfPerson; int geneNo, int ChrNo) {  
    for (int i=0; i < sizeOfPerson; i++) {  
        if (person[i].fit < bar) {  
            if (person[i].ChrNo.geneNo == 0)  
                person[i].ChrNo.geneNo = 1;  
            else  
                person[i].ChrNo.geneNo = 0;  
        }  
    }  
}
```

```
goMutation( ) {  
    while (!endOfBadList) {  
        randomDouble = new Random();  
        mutation(randomDouble, sizeOfPerson, badGene, badChromosome);  
        check next pair in the bad list;  
    }  
}
```

### 3. Systolic Array Implementation

The goal of this project is to implement the above algorithms into systolic array so that we can fully utilize the concept in parallel computing to speed up the evaluation of the natural selection. In the following subsection, we will demonstrate each algorithm except the crossover to show how we can implement them by using systolic arrays.

#### 3.1 Environment Setup

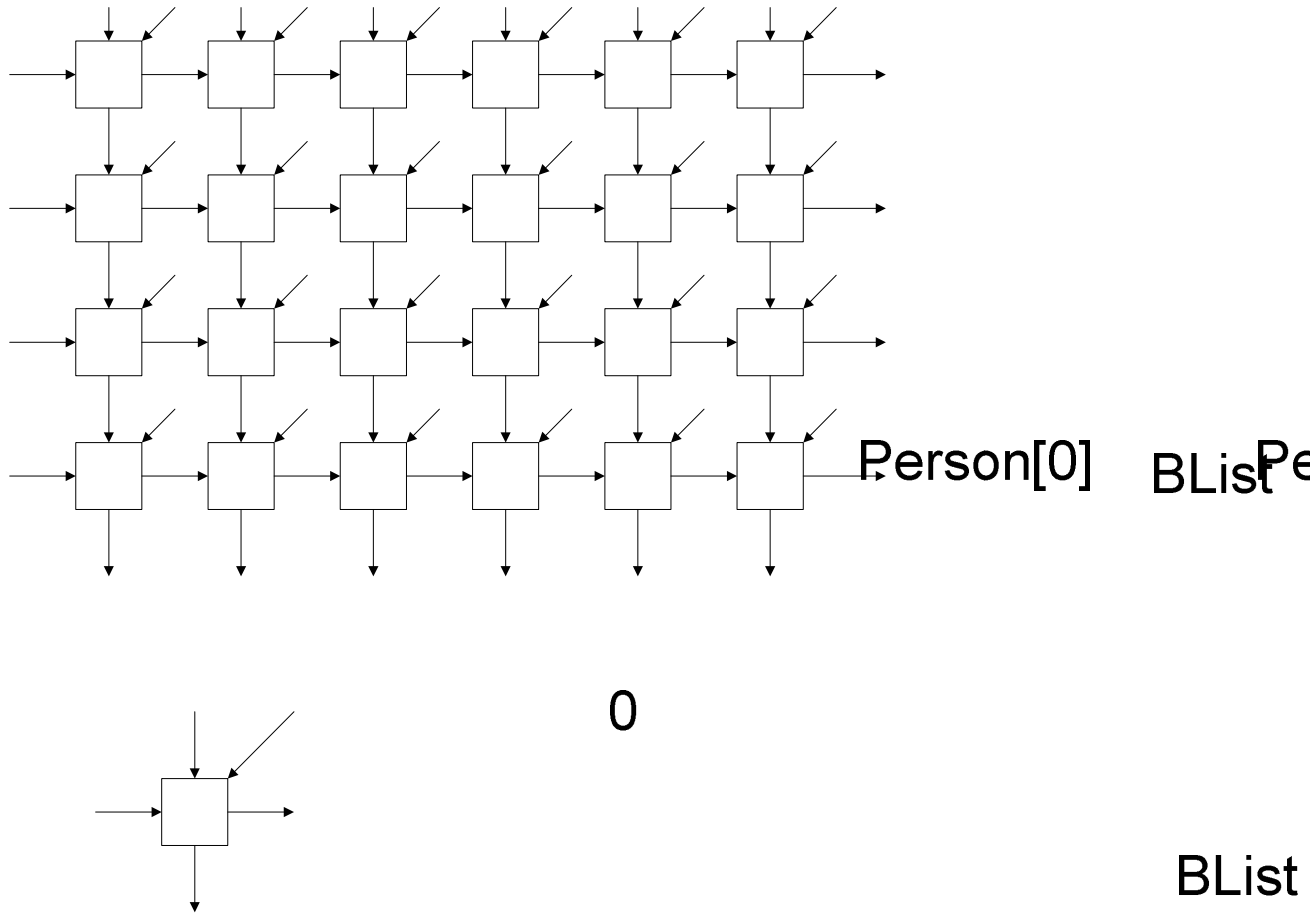
The systolic array implementation starts from the global initialization. In the global initialization, bad genes are attached to the bad gene list and the number of bad genes could be decided by the user. This is followed by the initialization stage which randomly decides the condition of each gene in a person, or even reset all the genes to bad condition. After the process, all the persons in the environment is then dispatched to the next stage to compute the personal fitness value and the overall fitness value.

#### 3.1 Fitness Functions

Fitness functions are basically made of conditional summation. Consequently, it reveals a possibility to optimize the code to make all the variables locally and all operations evenly distributed among processing units.

The optimized code is shown below:

```
public void personalFit(int sizeOfPerson, int productOfChrGene){
    Person [][] newPersonArray = new
    Person [sizeOfPerson][productOfChrGene+1];
    for (int i=0; i<sizeOfPerson; i++) {
        newPersonArray[i][0]=environment.retPerson(i);
        newPersonArray[i][0].setFit(0.0);
        for (int j=1; j<=productOfChrGene; j++) {
            newPersonArray[i][j]=newPersonArray[i][j-1];
            if (checkBadGene(newPersonArray[i][j-1], j%23, j/23)) {
                newPersonArray[i][j].
                setFit(newPersonArray[i][j-1].retFit()-
                    (double)1/productOfChrGene);
            }
            else {
                newPersonArray[i][j].
                setFit(newPersonArray[i][j-1].retFit()+
                    (double)1/productOfChrGene);
            }
        }
        if (1-newPersonArray[i][productOfChrGene].retFit(<0.004)
            newPersonArray[i][productOfChrGene].setFit(1.0);
        environment.setPerson(i,newPersonArray[i][productOfChrGene]);
    }
}
```



**Figure 2 Systolic Array Implementation of Computation of the Personal Fitness Function**

In each round, we generate a new Person. After initialization, we start each gene in the person by checking the bad list. If there is a match in the bad list and the gene is

bad, then the fitness value is decremented by  $\frac{1}{\text{product of \#Chromosome and \#Gene}}$ ;

otherwise, it is incremented by  $\frac{1}{\text{product of \#Chromosome and \#Gene}}$ .

This implementation in the systolic array can be best explained by the dependence graph as shown in Figure 2. From the figure, each processing unit can check a chromosome or a gene and see if there is any bad gene in a person which is identified in the bad list. If

there is, the temporary fitness value is subtracted by

$\frac{1}{\text{product of \#Chromosome and \#Gene}}$  and this value is then passed to the next

processing unit. At the end of the column, the personal fitness value is then computed. In addition, at the end of a row, we can have a statistic on how many persons have problems on a particular gene.

### 3.2 Selection

Selection is one of the most important parts in this project. As I mentioned in Section 2.2, there are two selection methods I introduced in this project. First of all, we can use a random number generator to choose the pairs who should be matched during the crossover stage. The Java code for this is shown below:

```
public void randomSelection(int sizeOfPerson) {
    ListIterator iterator = selectionList.listIterator(0);
    Random seed = new Random();
    Integer select = new Integer("0");
    int temp;
    for (int i=0; i<sizeOfPerson; i++) {
        do{
            temp = seed.nextInt(sizeOfPerson);
        }while(checkRepetition(temp));
        select = new Integer(temp);
        selectionList.add(select);
    }
}
```

Although this strategy looks reasonable, it is almost impossible to use this in terms of systolic array because of its non-regularity. Therefore, I use an alternative method to randomly select candidates. The Java code is shown below:

```
public int selection (int sizeOfPerson) {
    ListIterator iterator = selectionList.listIterator(0);
    Double [][]rand = new Double[sizeOfPerson][sizeOfPerson+1];
    final Double ZERO = new Double("0");
    Random singleRandom;
    Integer select = new Integer("0");
    Integer temp;
    int matchCount=0, repetitionCount=0;
    for (int i = 0; i<sizeOfPerson; i++) {
        iterator = selectionList.listIterator(0);
        singleRandom = new Random();
        rand [i][0] = new Double(singleRandom.nextDouble());
        for (int j=1; j<=sizeOfPerson; j++) {
            rand[i][j] = new Double(rand[i][j-1].doubleValue() -
```

```

        environment.retPerson(j-1).retFit());
    if (rand[i][j].compareTo(ZERO)<0) {
        if (!checkRepetition(j-1)) {
            select = new Integer(j-1);
            selectionList.add(select);
            rand[i][j] = new Double((double)sizeofPerson);
            matchCount++;
        }
        else {
            repetitionCount++;
        }
    }
}
}
}
if (matchCount != sizeofPerson) {
    System.out.println("Match Count is " + matchCount);
}
return matchCount;
}

public boolean checkRepetition(int j) {
    ListIterator iterator = selectionList.listIterator(0);
    Integer temp;
    while (iterator.hasNext()) {
        temp = (Integer)iterator.next();
        if (temp.intValue()==j)
            return true;
    }
    return false;
}
}

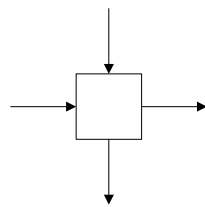
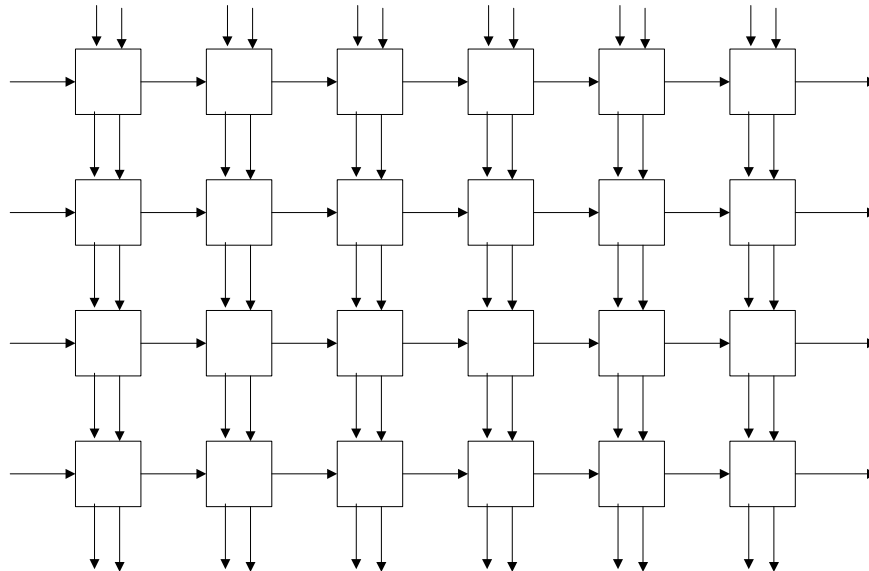
```

In each round, a random number is generated. This random number is subtracted by the fitness value of each person and a person is selected if the result of the subtraction is less than zero.

In terms of systolic arrays, Figure 3 shows the potential implementation. In each processing unit, it first does subtraction and compares the result to zero. If it is less than zero, it sends out a message to a central unit to check if the candidate has been selected. This checking is done by the code segment in the function *checkRepetition()*. If a candidate is selected, the random value is set to maximum value so that no other person after the candidate may have chance to get selected.

Although the above algorithm is perfect, we still have a problem since whenever there is a match, we need to check if there is any repetition. One way to avoid it is to reset the personal fit value to “0” whenever this person is selected. Because the fitness value is

set to zero, it is impossible to have negative result after the subtraction, which avoids the checking for repetition.



Person[0]

Figure 3 Systolic Array Implementation of Selection Mechanism

### 3.3 Mutation

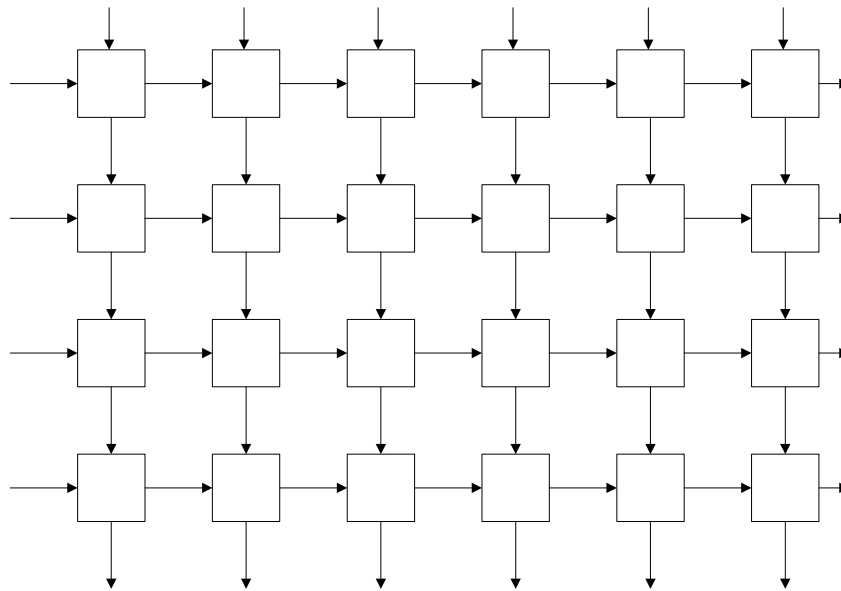
Mutation is a crucial process in the genetic algorithm. The major reason is that most of the time, every person has the same gene and it is almost impossible to have a variation even we have different match functions. Most of the mutation degrades the overall fitness value; however, still part of the mutation can improve the fitness and bring in more variation to the environment.

Person[1]

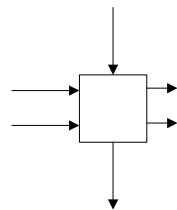
The occurrences of mutation can be categorized into two classes. First, mutation can happen randomly on a specific gene of specific person. Although it is a reasonable assumption and can be easily implemented by using random variables, it brings too much uncertainty.

Person[2]

S RND



RND 1  
and List 1



Person[0]

Figure 4 Systolic Array Implementation of Mutation Process

Instead of random selection, having a random bar value seems more reasonable and more practical. Figure 4 shows the implementation for the mutation process. The code segment is shown below:

```

public void mutation(double bar, int sizeOfPerson, int geneNo, int
ChrNo){
    Person currentPerson = new Person();
    Chr    currentChrX = new Chr();
    Chr    currentChrY = new Chr();
    int [] geneNoLocal = new int[sizeOfPerson+1];
    int [] ChrNoLocal  = new int[sizeOfPerson+1];
    double [] barLocal = new double[sizeOfPerson+1];

    geneNoLocal[0] = geneNo;
    ChrNoLocal[0]  = ChrNo;
    barLocal[0]    = bar;
}

```

Person[1]

14  
Person[2]

```

//System.out.println("Entering the Mutation stage");
for (int i=0; i<sizeOfPerson; i++) {
    currentPerson = environment.retPerson(i);
    if (barLocal[i] < currentPerson.retFit()) {
        currentChrX = currentPerson.retChrX(ChrNoLocal[i]);
        currentChrY = currentPerson.retChrY(ChrNoLocal[i]);
        if (currentChrX.retGene(geneNoLocal[i])==1)
            currentChrX.setGene(geneNoLocal[i], 0);
        else
            currentChrX.setGene(geneNoLocal[i], 1);

        if (currentChrY.retGene(geneNoLocal[i])==1)
            currentChrY.setGene(geneNoLocal[i], 0);
        else
            currentChrY.setGene(geneNoLocal[i], 1);
    }
    currentPerson.setChrX(ChrNoLocal[i], currentChrX);
    currentPerson.setChrY(ChrNoLocal[i], currentChrY);
    geneNoLocal[i+1]=geneNoLocal[i];
    ChrNoLocal [i+1]=ChrNoLocal[i];
    barLocal [i+1]=barLocal[i];
    environment.setPerson(i, currentPerson);
}
//System.out.println("Leaving the Mutation stage");
}

```

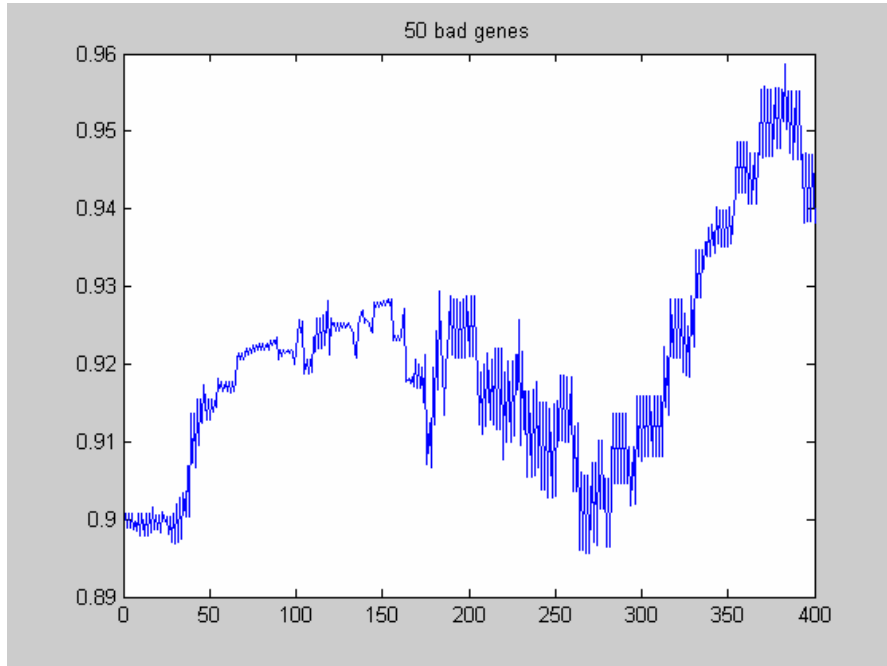
First, a bar value associated with each mutation element is generated. In each round, this random value is passed through the column and mutate the specific gene if the fitness function is less than the bar value. On the other hand, all persons need to go through the whole mutation list and finally they achieve the end and finish the process. A counter can be attached on each element on the mutation list so that researchers can do more research on it.

## 4. Experimentation Result

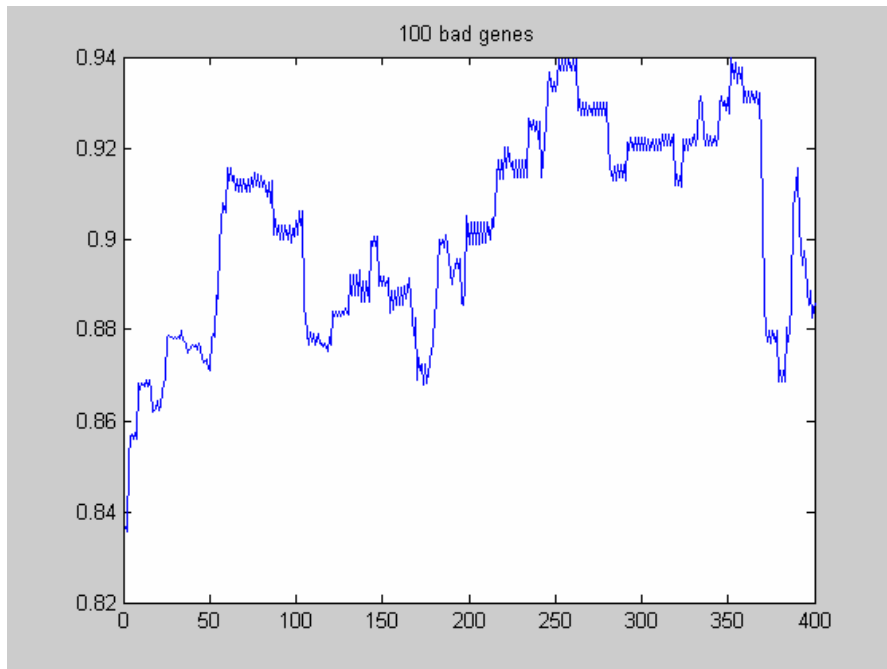
In this section, experimental results are present. The setup stage configures the environment so that there are 100 bad genes listed on the Bad Gene List. Besides, we setup the genes in each person by using a random generator. During each round, we perform the selection, crossover, and mutation and finally compute the personal fitness function and overall fitness function.

Figure 5-8 shows the overall fitness function during a period of 400 rounds. The basic environment setup includes 50, 100, 200, or 400 bad genes in the bad gene list, and random initial gene value. The selection criteria here is based on the personal fitness

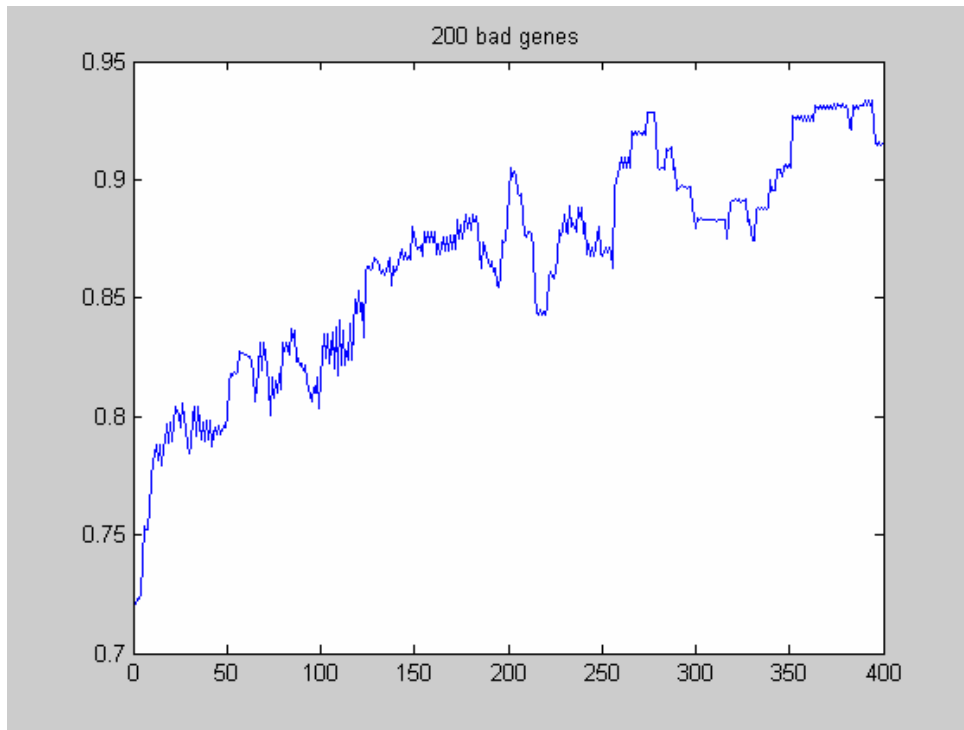
value; that is, a bar value is generated and is subtracted by the personal fitness value until it reaches zero, on which case, the candidate is selected and the person's number is fetched to the selection list. The mutation list is updated every round and one to ten mutation genes are added randomly.



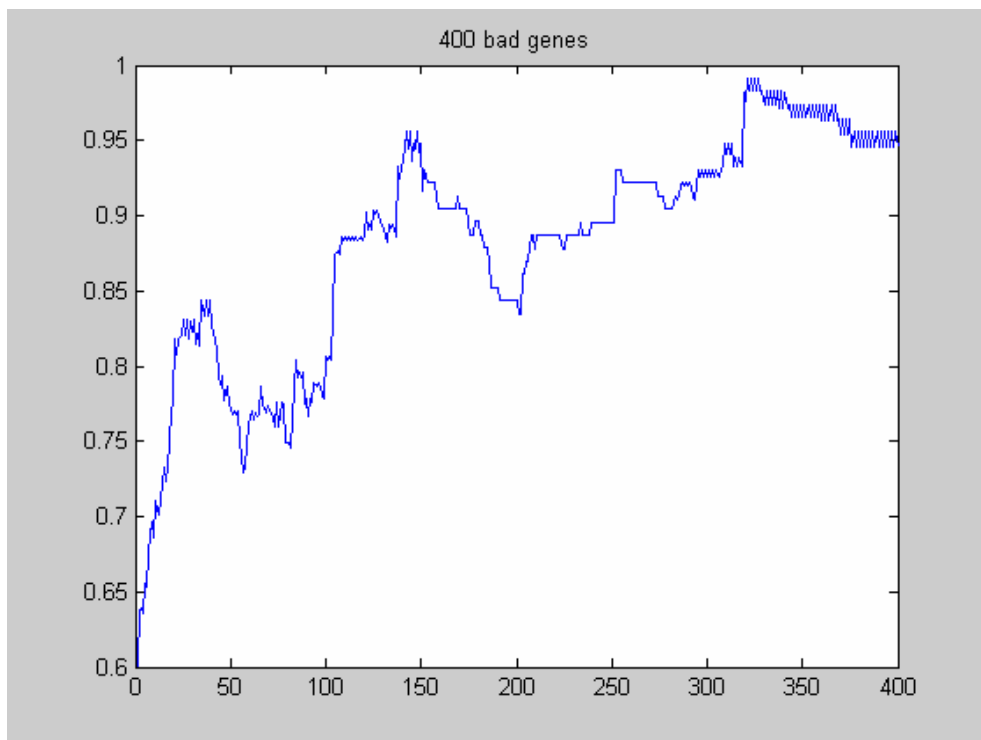
**Figure 5 Simulation Results for 50 Bad Genes in the Bad Gene List**



**Figure 6 Simulation Results for 100 Bad Genes in the Bad Gene List**



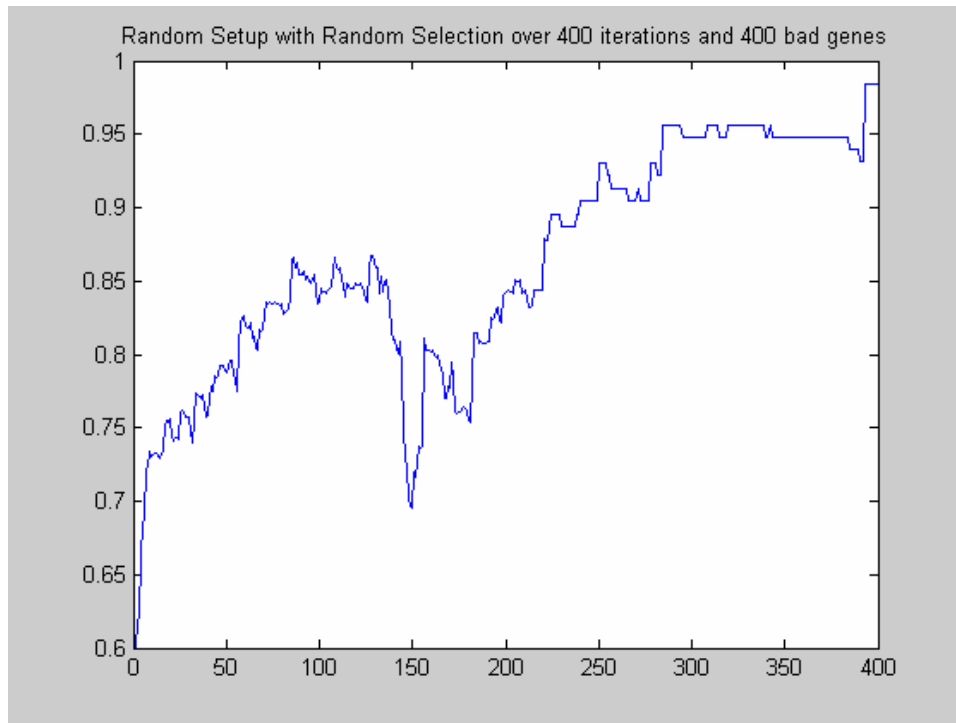
**Figure 7 Simulation Results for 150 Bad Genes in the Bad Gene List**



**Figure 8 Simulation Results for 200 Bad Genes in the Bad Gene List**

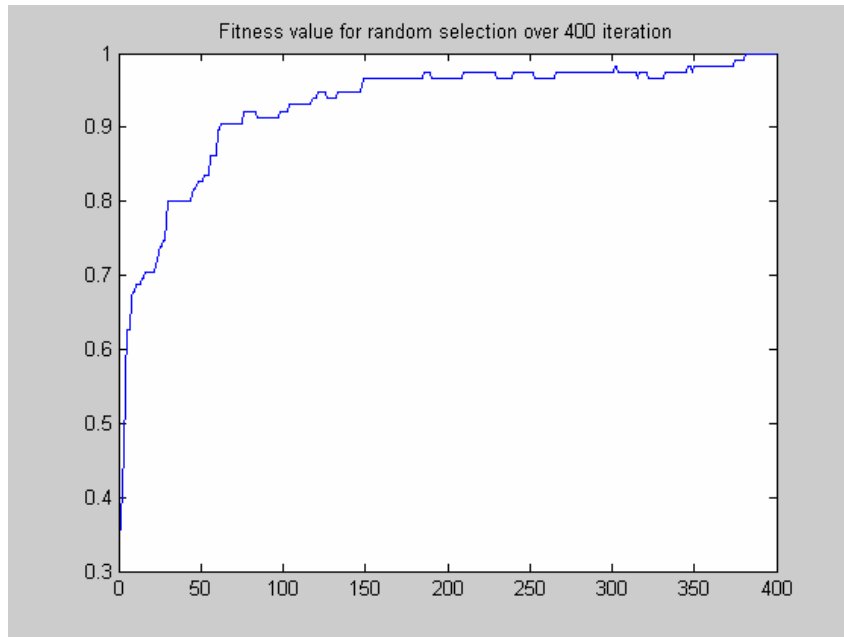
From these four figures, we can easily figure out that number of bad gene in the bad gene list only affect the fitness value at the first few hundred iterations. After around 200 iterations, fitness value is then converged to 0.9, which is roughly the same in four cases. In reality, it is also the case since the overall fitness value should converge to a same value if the number of mutation is small and the definition of bad gene is defined as the bad genes on both chromosome-X and chromosome-Y. In addition, since the personal fitness value increases, the possibility to have mutation decreases, which tends to level off the fitness value.

Figure 9 shows the environment fitness value over 400 iterations. The environment is set up with 400 bad genes (almost all genes' are bad if it is zero). The selection is random. From the figure, it can shows the overall fitness value is slowly increasing although there is a steep fall around 150<sup>th</sup> round.



**Figure 9 Simulation Results for Random Selection and 400 Bad Genes in the Bad Gene List**

Compared to Figure 8, we can conclude that having a random selection scheme almost has the same effect as our subtraction scheme; however, our scheme has the benefit of systolic array implementation, which is not allowed in the random selection scheme.



**Figure 10 Random Selection Starting with All Bad Genes with 400 Bad Genes in the Bad Gene List**

Figure 10 shows another interesting result if the genes of each person are all bad. From the figure, you can easily figure out that the curve has a steep increase at the very first iteration since the possibility to mutate is very high; however, it becomes level off; like what it is in the other case.

## 5. Conclusion

In this project, I have purposed a systolic-based genetic algorithm used in biological evolution process. First, I demonstrate the general algorithms which are frequently used in the biological evolution, as well as how to do the environmental setup in Section 2. In Section 3, I modified the original algorithms so that those new algorithms can fit into the systolic array criteria. In addition, several options are discussed to both have reasonable algorithms and meet the systolic array fashion. Section 4 shows the simulation result which show that the number of bad genes only affects the initial fitness value and the overall fitness value is usually converged to a specific value since the opportunity of mutation is reduced if all personal fitness values are large. Moreover, the result in Figure 8 and Figure 9 also demonstrates that our subtract-random selection can fit the parallel computing environment as well as reasonably select candidates.

It is easily seen that genetic algorithm can be fully implemented in a parallel computing environment and would potentially speed up the simulation especially when

the size of the data set is increasing. In addition, although we show the systolic array implementation in different figures, it does not mean those actions cannot be done at the same node. Actually, by using Field Programmable Gate Array (FPGA), we can easily reconfigure each processing unit at the end of each action, and then parse the result back to the systolic array to achieve the output of the next action. Consequently, a Computer-Aided-Design (CAD) tool which can convert a Java code to Verilog code depending on the number of persons, chromosome, and genes is expected in the near future.

Furthermore, fine-grained processing may not be always good if the computation within a node is trivial; therefore, a coarse-grained processing unit which can handles several data at a time is also expected to achieve the tradeoff among speed, cost and area. I personally believe by using either distributed processors or FPGA based multi-processing unit, researches can be conducted more efficiently in the field of biology.

## Reference:

- [1] **Adaptation in natural and artificial systems**, Holland, J., University of Michigan Press, Ann Arbor, 1975
- [2] **ECE 556 lecture note**, available at <http://www.cae.wisc.edu/~ece556/>
- [3] **Synthesis of a systolic array genetic algorithm**, *Megson, G.M.; Bland, I.M.*; Parallel Processing Symposium, 1998. 1998 IPSP/SPDP. Proceedings of the First Merged International...and Symposium on Parallel and Distributed Processing 1998 , 30 March-3 April 1998, Page(s): 316 -320
- [4] **The systolic array genetic algorithm, an example of systolic arrays as a reconfigurable design methodology**  
*Bland, I.M.; Megson, G.M.*;  
FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on , 15-17 April 1998  
Page(s): 260 -261
- [5] **Efficient operator pipelining in a bit serial genetic algorithm engine**  
*Bland, I.M.; Megson, G.M.*;  
Electronics Letters , Volume: 33 Issue: 12 , 5 June 1997  
Page(s): 1026 -1028
- [6] **JAVA API**, available at: <http://java.sun.com/j2se/1.3/docs/api/>