Report about Assignment 2 - Genetic Algorithm

1. Lei Xu

2. Peiqi Wang

Table of Content

1. Background

There are four proposed projects, each of which runs for 3 years and their characteristics are also given below for reference.

Project	Return (million)	Capital Required (million)		
		year 1	year 2	year 3
1	0.2	0.5	0.3	0.2
2	0.3	1.0	0.8	0.2
3	0.5	1.5	1.5	0.3
4	0.1	0.1	0.4	0.1

The available budget is 3.1 million for year 1, 2.5 million for year 2 and 0.4 million for year 3.

The goal of the assignment is to develop a program to decide which projects to invest in order to maximize the total return using the **Genetic Algorithm**.

2. Techniques

2.1 What is Genetic Alogithm?

A **genetic algorithm** is a search heuristic that is inspired by Charles Darwin's theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offsprings of the next generation.

2.2 Steps Involved in Genetic Algorithm

The process of natural selection starts with the selection of fittest individuals from a population. They produce offspring which inherit the characteristics of the parents and will be added to the next generation. If parents have better fitness, their offspring will be better than parents and have a better chance at surviving. This process keeps on iterating and at the end, a generation with the fittest individuals will be found.

There are six phases considered in a genetic algorithm:

1. **Initialize Population**
2. **Fitness Function**
3. **Selection**
4. **Crossover**
5. **Mutation**
6. **Stopping criteria**

2.2.1 Initialize Population

The process begins with a set of individuals which is called a **Population**. Each individual is a solution to the problem you want to solve. An individual is characterized by a set of parameters (variables) known as Genes. Genes are joined into a string to form a Chromosome (solution).

In a genetic algorithm, the set of genes of an individual is represented using a string, in terms of an alphabet. Usually, binary values are used (string of 1s and 0s). We say that we encode the genes in a chromosome.

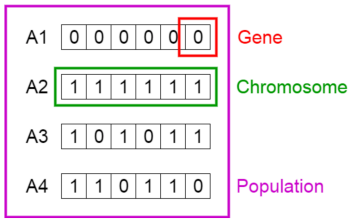


Figure 2.1 Population, Chromosomes and Genes

2.2.2 Fitness Function

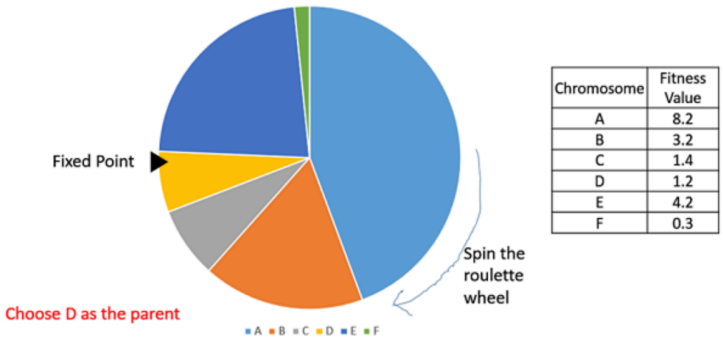
The **Fitness Function** determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its fitness score.

2.2.3 Selection

The idea of selection phase is to select the fittest individuals and let them pass their genes to the next generation. Two pairs of individuals (parents) are selected based on their fitness scores. Individuals with high fitness have more chance to be selected for reproduction.

Here, I use the **Roulette Wheel Selection** method to do the selection.

> **Roulette Wheel Selection** is a common algorithm used to select an item proportional to its probability. This could be imagined similar to a Roulette wheel in a casino. Usually a proportion of the wheel is assigned to each of the possible selections based on their fitness value. This could be achieved by dividing the fitness of a selection by the total fitness of all the selections, thereby normalizing them to 1. Then a random selection is made similar to how the roulette wheel is rotated. A fixed point is chosen on the wheel circumference as shown and the wheel is rotated. The region of the wheel which comes in front of the fixed point is chosen as the parent. For the second parent, the same process is repeated.



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Figure 2.2 Population, Chromosomes and Genes

It is clear that a fitter individual has a greater pie on the wheel and therefore a greater chance of landing in front of the fixed point when the wheel is rotated. Therefore, the probability of choosing an individual depends directly on its fitness.

Implementation wise, we use the following steps:

1. Calculate S = the sum of a finesses.
2. Generate a random number between 0 and S.
3. Starting from the top of the population, keep adding the finesses to the partial sum P, till P<S.
4. The individual for which P exceeds S is the chosen individual.

2.2.4 Crossover

Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a crossover point is chosen at random from within the genes.

For example, consider the crossover point to be 3 as shown below.

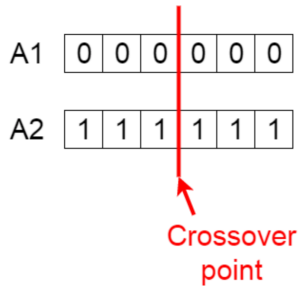


Figure 2.3 Crossover Point

Offspring are created by exchanging the genes of parents among themselves until the crossover point is reached.

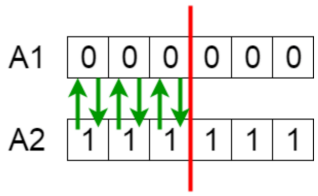


Figure 2.4 Exchanging genes among parents

The new offspring are added to the population.

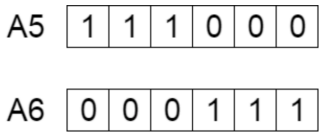


Figure 2.5 New Offspring

2.2.5 Mutation

In certain new offspring formed, some of their genes can be subjected to a mutation with a low random probability. This implies that some of the bits in the bit string can be flipped.

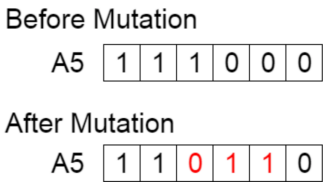


Figure 2.6 Mutation

Mutation occurs to maintain diversity within the population and prevent premature convergence.

2.2.6 Stopping criteria

The stopping criteria, also known as termination condition, is of importance in determining when the genetic algorithm run will end. It has been observed that initially, the genetic algorithm progresses very fast with better solutions coming in every few iterations, but this tends to saturate in the later stages where the improvements are very small. We usually want a stopping criteria such that our solution is close to the optimal, at the end of the run.

Usually, we keep one of the following termination conditions:

1. When there has been no improvement in the population for X iterations.
2. When we reach an absolute number of generations.
3. When the objective function value has reached a certain pre-defined value.

In the project, I will take the first one as the stopping criteria.

3. Implementation

According to the genetic algorithm described above, the psuedocode is given below.

```
START
Generate the initial population
Compute fitness
REPEAT
    Selection
    Crossover
    Mutation
    Compute fitness
UNTIL population has converged
STOP
```

3.1 Define a Chromosome Class

According to the description about the genetic algorithm, the population consists of chromosomes. So, I, firstly, define a chromosome class (**Chromosome.java**) to store chromosome attributes and functions.

Java

```java
public class Chromosome {
    private int[] genes = new int[Constants.CHROMOSOME_LENGTH];
    private String chromosome;
    public Chromosome() {
        for(int i = 0; i < Constants.CHROMOSOME_LENGTH; i++) {
            this.genes[i] = Math.random() >= 0.5 ? 1 : 0;
        }
        StringBuffer sb = new StringBuffer();
        for(int i = 0; i < Constants.CHROMOSOME_LENGTH; i++) {
            sb.append(this.genes[i]);
        }
        this.chromosome = sb.toString();
    }
    public Chromosome(String chromosome) {
        char[] genesChar = chromosome.toCharArray();
        for(int i = 0; i < genesChar.length; i++) {
            if(Character.isDigit(genesChar[i])) {
                this.genes[i] = Integer.parseInt(String.valueOf(genesChar[i]));
            }
        }
        this.chromosome = chromosome;
    }

    public double calculateFittness() { ... }
    public void selfMutation(int position) { ... }

    public int[] getGenes() {
        return this.genes;
    }
    public void setGenes(int geneValue, int position) {
        this.genes[position] = geneValue;
    }
    public String getChromosome() {
        return this.chromosome;
    }
}
```

3.2 Initialize Population

To start the genetic algorithm, initial population (**GeneticAlgorithm.java**) is necessary as the seed. In addition, it will be triggered to initialize population when the population shrinks to one chromosome after several iterations.

It is worth noting that all the chromosomes in the initial population meet the constraint requirement.

Java

```java
public ArrayList<Chromosome> initPopulation(int populationSize) {
        ArrayList<Chromosome> population = new ArrayList<Chromosome>();
        for(int i = 0; i < populationSize; i++) {
            Chromosome chromosome;
            do {
                chromosome = new Chromosome();
            } while(!isValid(chromosome));
            population.add(chromosome);
        }
        System.out.println("\nInitial Population: ");
        displayPhaseInfo(population);
        return population;
    }
```

3.3 Compute Fitness Score

Thirdly, the fitness score (**Chromosome.java**) is considered as the criteria of the selection phase to divide the population. The goal of this separation is that, later, the successful chromosomes will have more “chance” to get picked to form the next generation.

For precise computation of double type, I use BigDecimal calss here to achieve it. And the fitness score will be converted to double type after computation

Java

```java
public double calculateFittness() {
        BigDecimal fittnessScoreBD = BigDecimal.valueOf(0.0);
        for(int i = 0; i < Constants.FITTNESS_FACTORS.length; i++) {
            fittnessScoreBD = fittnessScoreBD.add(BigDecimal.valueOf(this.genes[i]).multi
        }
        return fittnessScoreBD.doubleValue();
    }
```

3.4 Define Selector Method

At the fourth phase (**GeneticAlgorithm.java**), high-quality chromosomes will be selected according to the Roulette Wheel Selection algorithm.

```java
public ArrayList<Chromosome> selector(ArrayList<Chromosome> parentPopulation, int childPo
        ArrayList<Chromosome> childPopulation = new ArrayList<Chromosome>(childPopulation
        double totalFitness = 0.0;
        double[] fitness = new double[parentPopulation.size()];
        for(Chromosome chromosome : parentPopulation) {
            totalFitness += chromosome.calculateFittness();
        }
        for(int index = 0; index < parentPopulation.size(); index++) {
            fitness[index] = parentPopulation.get(index).calculateFittness() / totalFitne
        }
        for(int i = 1; i < fitness.length; i++) {
            fitness[i] = fitness[i - 1] + fitness[i];
        }
        for(int i = 0; i < childPopulationSize; i++) {
            Random random = new Random();
            double probability = random.nextDouble();
            int choose;
            for(choose = 1; choose < fitness.length; choose++) {
                if(probability < fitness[choose]) {
                    break;
                }
            }
           childPopulation.add(parentPopulation.get(choose));
        }
        System.out.println("Selection: ");
        displayPhaseInfo(childPopulation);
        return childPopulation;
    }
```

3.5 Define Crossover Method

The crossover phase (**GeneticAlgorithm.java**) is analogous to reproduction and biological crossover. In this more than one parent is selected and one or more off-springs are produced using the genetic material of the parents.

Here, I employ the **One Point Crossover** method to produce off-springs.

> In this one-point crossover, a random crossover point is selected and the tails of its two parents are swapped to get new off-springs.

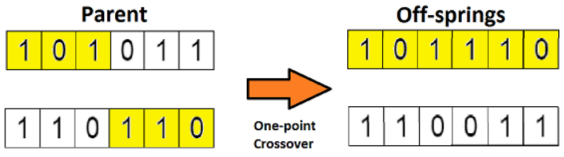


Figure 3.1 One Point Crossover

It is worth mentioning that all the off-springs must be checked whether they meet the constraint requirement before added in the population.

Java

```java
public ArrayList<Chromosome> crossover(ArrayList<Chromosome> parentPopulation, double cros

        ArrayList<Chromosome> childPopulation = new ArrayList<Chromosome>();
        ArrayList<Chromosome> individualsSelected = new ArrayList<Chromosome>();
        childPopulation.addAll(parentPopulation);
        Random random = new Random();
        for(int i = 0; i < parentPopulation.size() / 2; i++) {
            if(crossRate > random.nextDouble()) {
                int m = 0;
                int n = 0;
                do {
                    m = random.nextInt(parentPopulation.size());
                    n = random.nextInt(parentPopulation.size());
                } while (m == n);
                int position = random.nextInt(Constants.CHROMOSOME_LENGTH-1);
                Chromosome parent1 = parentPopulation.get(m);
                Chromosome parent2 = parentPopulation.get(n);
                individualsSelected.add(parent1);
                individualsSelected.add(parent2);
                String child1;
                String child2;
                child1 = parent1.getChromosome().substring(0, position+1) + parent2.getChr
                child2 = parent2.getChromosome().substring(0, position+1) + parent1.getChr
                Chromosome childChromosome1 = new Chromosome(child1);
                Chromosome childChromosome2 = new Chromosome(child2);
                if(isValid(childChromosome1)) {
                    childPopulation.add(childChromosome1);
                }
                if(isValid(childChromosome2)) {
                    childPopulation.add(childChromosome2);
                }
            }

        }
        System.out.println("\nCrossover: ");
        System.out.println("Individuals selected for crossover: " + chromosomesInPopulati
        displayPhaseInfo(childPopulation);
        return childPopulation;
    }
```

3.6 Define Mutation Method

Mutation (**GeneticAlgorithm.java**) may be defined as a small random tweak in the chromosome, to get a new solution. It is used to maintain and introduce diversity in the genetic population and is usually applied with a low probability.

Here, I apply the **Bit Flip Mutation** method to prevent the algorithm to be blocked in a local minimum.

> In this bit flip mutation, One or more random bits is/are selected and flipped. This is used for binary encoded genetic algorithm.

Figure 3.2 Bit Flip Mutation

It is also worth mentioning that:

1. Before mutation, the best chromosome needs to be selected and stored for later merge.
2. After mutation, the validity check operation is also necessary.

Java

```java
public ArrayList<Chromosome> mutation(ArrayList<Chromosome> parentPopulation, double muta
        String mutationProbability;
        int mutationCount = 0;
        ArrayList<Chromosome> individualsSelected = new ArrayList<Chromosome>();
        Random random = new Random();
        ArrayList<Chromosome> bestChromosomes = selectBestChromosomes(parentPopulation);
        do {
            for(Chromosome chromosome : parentPopulation) {
                if(mutationRate > random.nextDouble()) {
                    int position = random.nextInt(Constants.CHROMOSOME_LENGTH);
                    chromosome.selfMutation(position);
                    individualsSelected.add(chromosome);
                    mutationCount++;
                }
            }
        } while(mutationCount <= 0);
        DecimalFormat df = new DecimalFormat("0.00");
        mutationProbability = df.format((double)mutationCount / parentPopulation.size());
        System.out.println("\nMutation: ");
        System.out.println("Mutation Probability: " + mutationProbability);
        System.out.println("Individuals selected for mutation: " + chromosomesInPopulatio
        System.out.println("After mutation, the population size: " + parentPopulation.size
        deleteInvalidChromosome(parentPopulation);
        parentPopulation.addAll(bestChromosomes);
        System.out.println("After deleting invalid chromosomes and adding the best chromos
        displayPhaseInfo(parentPopulation);
        return parentPopulation;
    }
```

3.7 Setting stopping criteria

The algorithm terminates if the population has converged (does not produce offspring which are significantly different from the previous generation). Then it is said that the genetic algorithm has provided a set of solutions to our problem.

Here, when there has been no improvement in the population for X iterations, the genetic algorithm will stop running.

To specify, in the genetic algorithm, I keep a counter which keeps track of the generations for which there has been no improvement in the population. Initially, we set this counter to zero. Each time we don't generate off-springs which are better than the individuals in the population, we increment the counter. However, if the fitness any of the off-springs is better, then we reset the counter to zero. The algorithm terminates when the counter reaches a predetermined value.

Java

```java
int countOfSameResult = 0;
while(countOfSameResult != Constants.MAX_COUNT_OF_SAME_RESULT) {
    ...
    if(Constants.optimal_total_return == optimalReturnOfIteration) {
        countOfSameResult++;
    } else if(Constants.optimal_total_return < optimalReturnOfIteration) {
        Constants.optimal_total_return = optimalReturnOfIteration;
        countOfSameResult = 0;
    } else {
        countOfSameResult = 0;
    }
    ...
}
```

4. Results & Analysis

As requested in the project:

1. the **crossover rate** should be between **0.8 and 0.95**.
2. the **mutation rate** needs to be set between **0.001 and 0.1**.
3. the **fitness function** is **maximize** $f(x_1, x_2, x_3, x_4) = 0.2x_1 + 0.3x_2 + 0.5x_3 + 0.1x_4$

4.1 Scenario 01 – Accuracy & Performability

Case 01 – with small initial population

Purpose:

Check whether the program can return the correct result when running with small initial population.

Requirements:

1. **Basics**::
    - Chromosome size: **4**
    - Initial population size: **10 <–**
    - Maximum iteration count (result with no improvement): **50**
    - Crossover rate: **0.9**
    - Mutation rate: **0.1**
2. **Constraints**:
    - $0.5x_1 + 1.0x_2 + 1.5x_3 + 0.1x_4 \leq 3.1$
    - $0.3x_1 + 0.8x_2 + 1.5x_3 + 0.4x_4 \leq 2.5$

- $0.2x_1 + 0.2x_2 + 0.3x_3 + 0.1x_4 \leq 0.4$

Hypothesis & Analysis:

1. **Optimal solution**: [0011]
    - $x_1 = 0$
    - $x_2 = 0$
    - $x_3 = 1$
    - $x_4 = 1$
2. **Optimal total return**: 0.6

Results:

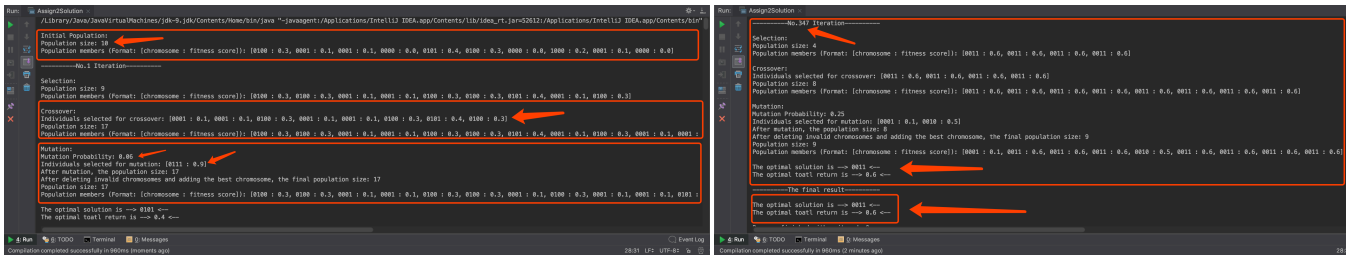


Figure 4.1 Scenario 01 - Case 01

Case 02 – with large initial population

Purpose:

Check whether the program can return the correct result when running with large initial population.

Requirements:

1. **Basics**::
    - Chromosome size: **4**
    - Initial population size: **100 <–**
    - Maximum iteration count (result with no improvement): **50**
    - Crossover rate: **0.9**
    - Mutation rate: **0.1**
2. **Constraints**:
    - $0.5x_1 + 1.0x_2 + 1.5x_3 + 0.1x_4 \leq 3.1$
    - $0.3x_1 + 0.8x_2 + 1.5x_3 + 0.4x_4 \leq 2.5$
    - $0.2x_1 + 0.2x_2 + 0.3x_3 + 0.1x_4 \leq 0.4$

Hypothesis & Analysis:

1. **Optimal solution**: [0011]
    - $x_1 = 0$
    - $x_2 = 0$
    - $x_3 = 1$
    - $x_4 = 1$
2. **Optimal total return**: 0.6

Results:

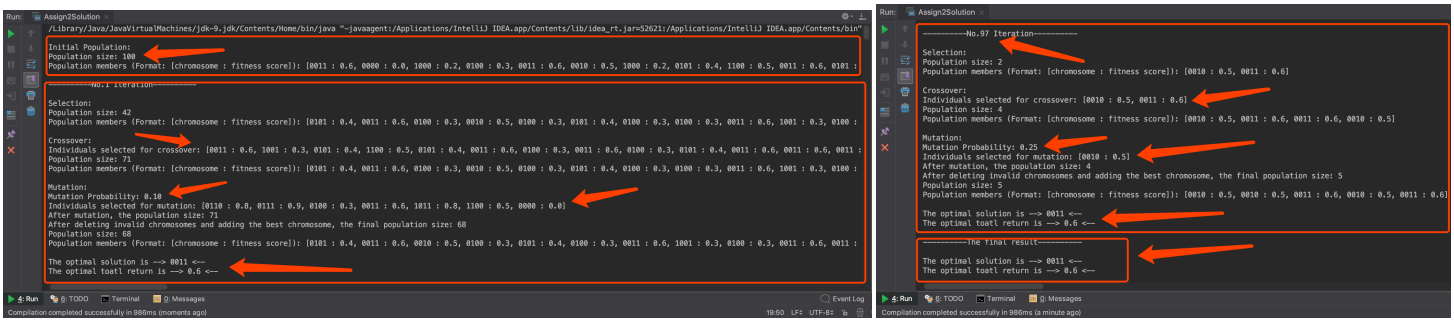


Figure 4.2 Scenario 01 - Case 02

Case 03 – with large maximum iteration count

Purpose:

Check whether the program can return the correct result when running with large maximum iteration count.

Requirements:

1. **Basics**::
    - Chromosome size: **4**
    - Initial population size: **100**
    - Maximum iteration count (result with no improvement): **500 <–**
    - Crossover rate: **0.9**
    - Mutation rate: **0.1**
2. **Constraints**:
    - $0.5x_1 + 1.0x_2 + 1.5x_3 + 0.1x_4 \leq 3.1$
    - $0.3x_1 + 0.8x_2 + 1.5x_3 + 0.4x_4 \leq 2.5$
    - $0.2x_1 + 0.2x_2 + 0.3x_3 + 0.1x_4 \leq 0.4$

Hypothesis & Analysis:

1. **Optimal solution**: [0011]
    - $x_1 = 0$
    - $x_2 = 0$
    - $x_3 = 1$
    - $x_4 = 1$
2. **Optimal total return**: 0.6

Results:

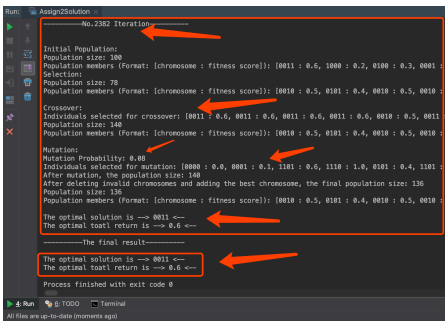


Figure 4.3 Scenario 01 - Case 03

5. Conclusion

According to the discussion above, the genetic algorithm is a method for solving both constrained and unconstrained optimization problems that is based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions. At each step, the genetic algorithm selects individuals at random from the current population to be parents and uses them to produce the children for the next generation. Over successive generations, the population “evolves” toward an optimal solution.

Because of the attribute of the problem in the project, I decided to apply the genetic algorithm to solve the maximization of the investment. Via the operations of selection, crossover, and mutation, the genetic algorithm will converge over successive generations towards the global optium to obtain the optimal total return.