

University of Central Lancashire

School of Engineering

EL3300: Machine Intelligence

B.Eng.(Hons.) Robotics Engineering

8th March 2020

Genetic Algorithm (GA) Assignment

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

Page 1 of 42

Table of Contents

1. Introduction	3
2. Genetic Algorithm Code design implementation.....	3-23
3. Testing.....	24-26
4. Results and discussions.....	26-27
5. Conclusion	27
6. References	28
7. Appendix.....	29-42
7.1 Appendix A: List of engine configuration parameters	29
7.2 Appendix B: Selection1 Function code snippet.....	30-32
7.3 Appendix C: Selection2 Function code snippet.....	32-33
7.4 Appendix D: Crossover Functions code snippet.....	34-35
7.5 Appendix E: Mutation Functions Code snippet.....	36-41
7.6 Appendix F: Test and trial calculations.....	42

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

Introduction

“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 offspring of the next generation.”[1] This report deals with designing, implementing and testing a genetic algorithm program to find the optimal engine configurations.

2. Genetic Algorithm Code design implementation

Genetic algorithm comprises of five stages as follows:

1. Initialising population.
2. Evaluating the fitness score of each chromosome string in the population.
3. Selecting the optimal N% chromosome strings based on their fitness score.
4. Implementing Crossover function on the optimal chromosome strings.
5. Implementing Mutation function on the optimal chromosome strings.

This section will review how each of the above five stages are designed and implemented to find optimal engine configurations using genetic algorithm coding in C# programming language using a console application.

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

Step 1: Creating population and evaluating the fitness scores of the initialised population and sending optimal N% to generation loop for crossover and mutation.

This stage involves generating random chromosome strings to indicate whether the parameter is Very Low, Low, Medium, High or Very High. For example, a 'high' value of parameter1 could be indicated by a group of 5 binary elements as "00010". So each group of five elements can have only one 1 and rest zeroes. This is true for all parameters from 1 to 9.

This is coded in C# using list<int> in the following manner:

```
// Adding elements to List
list1.Add(0);
list1.Add(0);
list1.Add(0);
list1.Add(0);
list1.Add(0);
list1[random.Next() % 5] = 1; // changing one 0 to 1 at random
position
```

Figure1: Code snippet for generating the general engine configuration encoding of parameters 1 to 9.

The list1<int> represents creating the group of five elements for parameter1 and list2<int> represents creating the group of five elements for parameter 2 and so on till list9<int> is the last list created in this manner.

Then for parameters 10-14, each parameter can either take 1 or 0 to indicate the presence or absence of a device. The absence of a device is indicated by a zero. Engine configuration for parameters 10 to 14 is stored in list 10. Note that this group of 5 elements would have more than one 1 in them. So, this is programmed in C# as follows:

```

for (int i = 0; i < 5; i++) //loop 5 times
{
    int r = random.Next(0, 2); //generate random number 0 or 1
    list10.Add(r); //add the random number to list10
}

```

Figure2: Code snippet for generating the engine configuration encoding of parameters 10 to 14.

And the last 15th parameter “Fuel Injection Timings” only has three settings : Low, Medium and High. So it encodes a medium setting as “010”, a low setting as “100” and a high setting as “001”. This is programmed in C# as follows:

```

list11.Add(0);
list11.Add(0);
list11.Add(0);
list11[random.Next() % 3] = 1; // changing one 0 to 1 at
random position

```

Figure3: Code snippet for generating the engine configuration encoding of parameter 15.

Thus, in total 11 list<int> are created to store engine configurations of 15 parameters. However, these lists need to be concatenated together to form a chromosome in one list called “Individualslist” as follows:

```

List<int> Individualslist=
list1.Concat(list2).Concat(list3).Concat(list4).Concat(list5).Concat(list6).Concat(list7).Concat(list8).Concat(list9).Concat
(list10).Concat(list11).ToList();

```

Figure4: Code snippet for generating a single chromosome using concatenation of the different parameter encodings in order.

Thus, in preparation for using the optimisation methods each of the 15 parameters have been converted to a binary input represented by the chromosome containing total of 53 elements in the following manner. Refer to Appendix A to view the complete list of engine configuration parameters.

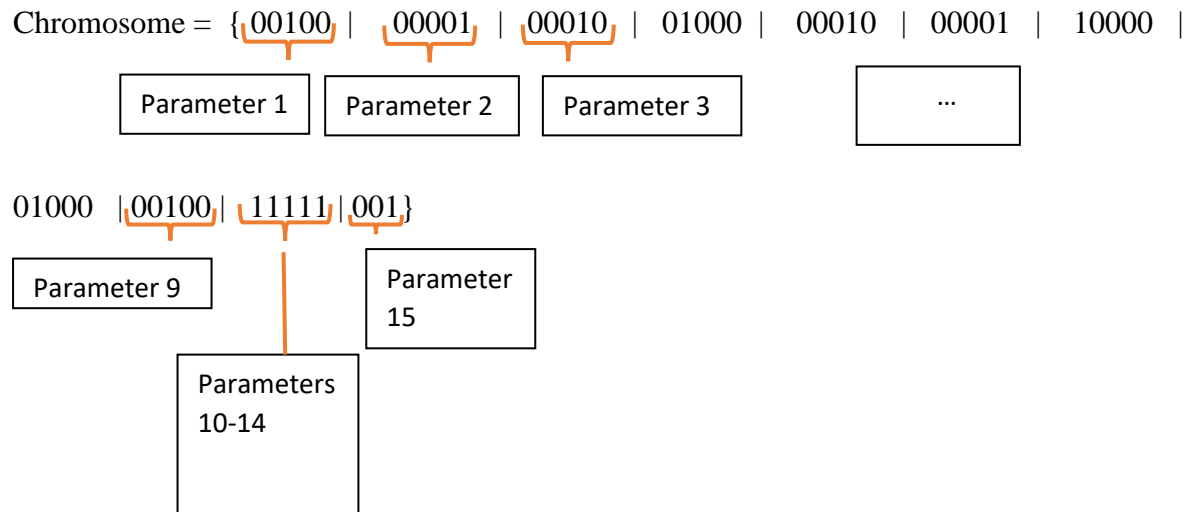


Figure5: Representation of a single chromosome consisting of 53 elements.

In order to test each run of the engine a fitness function is required. The engine has been modelled and a given library (Windows dll) returns a grading value for any run given to it. However, to evaluate the fitness score the chromosome needs to be in string format. Whereas, “Individualslist” is list of integers so “var chromosome” is used to bind each element in “Individualslist” and to store each chromosome string in “chromosomeList” which is a List<string> so that fitness function can be used to evaluate the fitness score of the chromosome string. The population of given size is generated using a “for loop” and stored in “testPopulationlist”. The population size is stored in

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

“int POPULATION_SIZE” variable.

```
for (int i = 0; i < POPULATION_SIZE; i++)
{
    var chromosome = string.Join("", Individualslist);
    // concatenate chromosomes is in List<int> data type and needs to
    be converted to string and this string needs to be stored in "chromosome"
    chromosomeList.Add(chromosome);

    Console.WriteLine(chromosomeList[i]);
}

List<string> testPopulationlist = new List<string>();

testPopulationlist.AddRange(chromosomeList);
```

Figure6: Code snippet for describes the above steps of binding the binary integers in “Individualslist” into a chromosome string and adding each chromosome string into “chromosomeList”.

The “testPopulationlist” contains the number of chromosome strings defined by the population size in the following manner:

```
testPopulationlist = {"00010010000001001000000101000001000000010100010101010",
```

The unit test for initialising the population for the purpose of testing is stored in a console application folder in `EL3300_submission` folder at `\EL3300_submission\initialise_population_unit_test\initialise_population` folder path.

Figure7: Code snippet for demonstrating the complete code for initialising the population of the given population size.

```
int POPULATION_SIZE = 1000;
```

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

```

List<string> chromosomeList = new List<string>();

var rand = new Random();
for (int n = 0; n < POPULATION_SIZE; n++) // here 10 is the population size
{
    Random random = new Random(); //create a random object
    List<int> list1 = new List<int>(); //create a list for parameter 1
    List<int> list2 = new List<int>(); //create a list for parameter 2
    List<int> list3 = new List<int>(); //create a list for parameter 3
    List<int> list4 = new List<int>(); //create a list for parameter 4
    List<int> list5 = new List<int>(); //create a list for parameter 5
    List<int> list6 = new List<int>(); //create a list for parameter 6
    List<int> list7 = new List<int>(); //create a list for parameter 7
    List<int> list8 = new List<int>(); //create a list for parameter 8
    List<int> list9 = new List<int>(); //create a list for parameter 9

    List<int> list10 = new List<int>(); //created a list10 correct for 10 - 14 parameters (5 elements)

    List<int> list11 = new List<int>(); //create a list 11 for 15th parameter (without only 1 rest 0)

    // Adding elements to List
    list1.Add(0);
    list1.Add(0);
    list1.Add(0);
    list1.Add(0);
    list1.Add(0);
    list1[random.Next() % 5] = 1; // changing one 0 to 1 at random position

    list2.Add(0);
    list2.Add(0);
    list2.Add(0);
    list2.Add(0);

```



```
list2.Add(0);  
list2[random.Next() % 5] = 1; // changing one 0 to 1 at random position  
  
list3.Add(0);  
list3.Add(0);  
list3.Add(0);  
list3.Add(0);  
list3.Add(0);  
list3[random.Next() % 5] = 1; // changing one 0 to 1 at random position  
  
list4.Add(0);  
list4.Add(0);  
list4.Add(0);  
list4.Add(0);  
list4.Add(0);  
list4[random.Next() % 5] = 1; // changing one 0 to 1 at random position  
  
list5.Add(0);  
list5.Add(0);  
list5.Add(0);  
list5.Add(0);  
list5.Add(0);  
list5[random.Next() % 5] = 1; // changing one 0 to 1 at random position  
  
list6.Add(0);  
list6.Add(0);  
list6.Add(0);  
list6.Add(0);  
list6.Add(0);  
list6[random.Next() % 5] = 1; // changing one 0 to 1 at random position  
  
list7.Add(0);  
list7.Add(0);  
list7.Add(0);
```

```

list7.Add(0);
list7.Add(0);
list7[random.Next() % 5] = 1; // changing one 0 to 1 at random position

list8.Add(0);
list8.Add(0);
list8.Add(0);
list8.Add(0);
list8.Add(0);
list8[random.Next() % 5] = 1; // changing one 0 to 1 at random position

list9.Add(0);
list9.Add(0);
list9.Add(0);
list9.Add(0);
list9.Add(0);
list9[random.Next() % 5] = 1; // changing one 0 to 1 at random position

for (int i = 0; i < 5; i++) //loop 5 times
{
    int r = random.Next(0, 2); //generate random number 0 or 1
    list10.Add(r); //add the random number to list10
}

list11.Add(0);
list11.Add(0);
list11.Add(0);
list11[random.Next() % 3] = 1; // changing one 0 to 1 at random position

List<int> Individualslist =
list1.Concat(list2).Concat(list3).Concat(list4).Concat(list5).Concat(list6).Concat(list7).Concat(list8).Concat(list9).Concat(list10)
).Concat(list11).ToList();

var chromosome = string.Join("", Individualslist); // concatenated chromosomeList is in List<int> data type and
needs to be converted to string and this string needs to be stored in "chromosome"

```

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

```

        chromosomeList.Add(chromosome);
    } // "for loop" of initialising Population

    // printing initialised Population
    for (int i = 0; i < POPULATION_SIZE; i++)
    {
        Console.WriteLine(chromosomeList[i]);
    }

    List<string> testPopulationlist = new List<string>();
    testPopulationlist.AddRange(chromosomeList);

```

Step 2: Evaluating the fitness score of the initialised population sorting the optimal N% of the population.

List<int> “fitnesseslist” is created for the purpose of storing the fitness scores of chromosome strings stored in “testPopulationlist” using the “FitFunc generationtest = new FitFunc();” fitness function as shown in the code snippet below:

Figure8: Code snippet for evaluating the fitness score of the initialised population.

```

List<int> fitnesseslist = new List<int>(testPopulationlist.Count); //fitnesseslist
will contain same number of elements present in testPopulationlist.

    // calculate fitness for test population.

    for (int i = 0; i < 1; ++i) // this loop evaluates the fitness grade of
the engine configurations chromosome string
    {
        FitFunc generationtest = new FitFunc();
        foreach (string chromosomeString in testPopulationlist)
        {
            int gradeValue = generationtest.evalFunc(chromosomeString);

```

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

```

        fitnesseslist.Add(gradeValue);
    }
}

```

The evaluated fitness score corresponding to each chromosome string in “testPopulationlist” is sent as parameter to **Selection1** function which is used to sort the fitness values in descending order and so that only optimal N% of the fit chromosome strings can be stored in “topScorerList” and the original “testPopulationlist” can be modified to now contain only the optimal N% of the fit chromosome strings deleting all the rest of the strings.

Figure9: Code snippet for the sorting the fitnesslist and corresponding chromosome strings in testPopulationlist using **Selection1** function.

```

List<string> topScorerList = Selection1(testPopulationlist, fitnesseslist);

    for (int i = 0; i < fitnesseslist.Count; i++)
    {
        if (testPopulationlist[i] == topScorerList[0])
        {
            Console.WriteLine(topScorerList[0] + " scored the highest
fitness grade of " + fitnesseslist[i]);
            break;
        }
    }

    testPopulationlist = topScorerList; //now contain only the top 10%
student names
//Clear the contents of the
gradeslist completely and keep only

```

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

```
fitnesseslist.Clear();
```

The unit test for Selection1 function for the purpose of testing is stored in a different console application folder in EL3300_submission folder under \EL3300_submission\Selection_unit_test folder path. The Selection1 function code snippet is in the appendix B.

Selection1 Function takes in two parameters namely, `IEnumerable<string> testPopulationlist` and `IEnumerable<int> fitnesseslist`

And sorts the fitnesseslist in descending order and stored in “sortedfitnessGradesList” using the following code line:

```
//Fitness Grades attained by each chromosome string are sorted  
List<int> sortedfitnessGradesList = fitnesseslist.OrderByDescending(x => x).ToList();
```

Then, the optimal N% fitness values from “sortedfitnessGradesList” are searched by using the ‘for loop’ along with the corresponding chromosome strings who attained those top N% fitness values. These optimal N% chromosome strings are stored in “topChromosomesList” from “testPopulationlist”. Then we check if we have reached the optimal N% using following code line and if we have then break out of this loop:

```
//checking if we reached optimal x%  
if (i >= Percent)  
    break;
```

The 2nd ‘for loop’ in the **Selection1** Function is for printing the optimal N% of chromosomes in the Console along with their fitness values for the purpose of testing.

Selection2 Function also implements the same operation inside the generation loop but for sorting “Generationlist” and “fitnesseslist1”.

Step3: Generation loop

The optimal N% chromosomes in a population undergo a series of steps to produce the next generation. Below are some terms used to describe these steps:

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

- **Fitness** is a number describing the probability for a chromosome to survive and reproduce.
- **Crossover** is the process of the fittest 2 parent chromosomes to exchange pieces of their data with one another to produce offsprings
- **Mutation** is the process of flipping the bit from 1 to 0 or vice versa randomly at specific positions in this case since first 9 engine parameters and 15th parameter can only have one 1 in groups of 5 and 3 elements respectively.
- **Reproduce** is when a chromosome copies itself into the next generation.

The steps of the genetic algorithm “generation loop” are as follows:

1. Produce an initial generation of chromosome strings with a given population size using a random number generator as already discussed above.
2. Determine the fitness of all of the chromosome strings and apply **Selection1** function `static List<string> Selection1(IEnumerable<string> testPopulationlist, IEnumerable<int> fitnesseslist)` to sort the optimal N% of the chromosome strings to be passed into the generation loop for crossover and mutation.
3. Crossover the parent chromosome pairs from the randomly selected ‘n’ parent strings in “OptimumPopulationlist” and store the crossover offspring strings generated using crossover function in AppendixD in “crossoverResultlist”.
4. Produce random mutations through the next generation population.
5. Evaluate the fitness scores of all the chromosomes in the “Generationlist” and determine which chromosomes are allowed to reproduce by selecting the optimal N% using **Selection2** function `static List<string> Selection2(IEnumerable<string> Generationlist, IEnumerable<int> fitnesseslist1)` (refer to Appendix C) which is identical to Selection1 function in Appendix B with the difference of different lists of chromosome strings and their corresponding fitness values lists sent as parameters to the functions. Then, loop back to step 3.
6. Keep iterating through the generation loop till the stopping criteria of the generation loop is achieved. The stopping criteria in this case is to stop if the number of iterations given in the “generations” variable is completed. That is if the “generation” variable is given 10 then the loop should stop after 10 iterations.

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

Substep1 : Crossover

The unit test for 2 functions for implementing crossover operation namely `private static string[] crossover(string[] list, int[] IndexArray)` and `private static string[] combine(string str1, string str2, int index)` for the purpose of testing crossover operation is stored in a different console application folder in `EL3300_submission` folder at `EL3300_submission\Crossover_unit_test\Binary_crossover\binary crossover` folder path. The code snippet of two crossover operation functions is in the appendix D.

Single Crossover is implemented by taking 10 crossover index points into consideration. The 10 index points are selected in the such a way that when crossover happens the groups of 5 elements and 3 elements for engine parameters 1 to 9 and parameter 15 respectively remain undisturbed that is, each group of 5 or 3 elements continues to have only one 1 in them. All the crossover index points are stored in an array as shown below.

```
int[] array = { 5, 10, 15, 20, 25, 30, 35, 40, 45, 50 }; // array of crossover index points
```

The calculations for selecting ‘n’ random parent chromosomes from “Generationlist” to be stored in “OptimumPopulationlist” is described in detail in the next section (refer to section 3. Testing). And then crossover function is applied to compute the calculated number of offsprings.

The crossover function takes in two arrays as parameters. One parameter is for chromosome strings stored in array of string called “list” and other parameter for array of crossover index points. The crossover functions calls another function called “combine” function to break two strings at a crossover index from the `IndexArray[]` and recombine those two strings together. And crossover function tries all combinations of the randomly selected parents inside “OptimumPopulationlist” and returns a `result[]` of strings to the main program. This array of strings is converted to list of strings

“`List<string> crossoverResultlist`” and so that crossover offsprings can be added to “Generationlist” eventually through “runPopulationlist”.

For a single crossover at Index 5 for a string of total characters 13 is as follows:

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

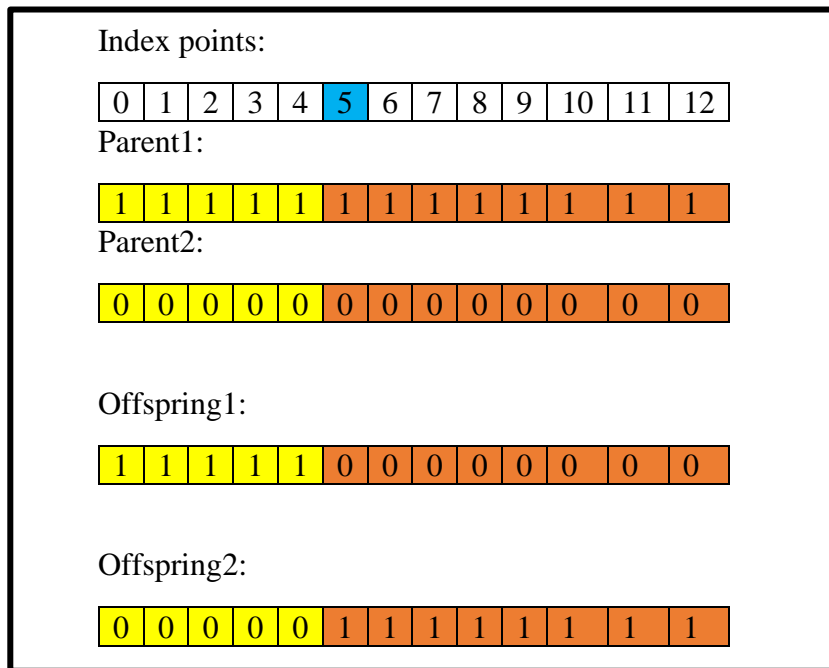


Figure10: Single Crossover operation at Index point 5 for total of 13 elements in the parent strings.

Substep2 : Mutation

The unit test for Mutation operation functions namely `public List<string> Mutation(List<string> Generationlist)` and `public static void RemoveIdenticalString(List<string> resultList, List<string> Generationlist)` for the purpose of testing mutation operation is stored in a different console application folder in `EL3300_submission` folder at `EL3300_submission\mutation_unit_testing` folder path. The code snippet for of two mutation functions is in the appendix E.

Mutation is done by replacing the string at specific index positions maintain the validity of the chromosome string for parameter configuration as follows:

Figure11: Code snippet for basic Mutation operation

```
int pos1 = 5; // specific index position
public const string switch1 = "00100"; // flip bit in the original string at index position specified to this switch1 string
    // Replacing replacingString1 = "00100" in different positions
string s1 = str.Substring(0, pos1) + switch1 + str.Substring(pos1 + 5);
// Adding that replaced string to the resultlist
resultlist.Add(s1);
```

`str.Substring(pos1 + 5);` ensures that the groups of 5 elements and 3 elements for engine parameters 1 to 9 and parameter 15 respectively remain undisturbed that is, each group of 5 or 3 elements continues to have only one 1 in them.

The following diagram shows mutation operation for 13 characters:

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

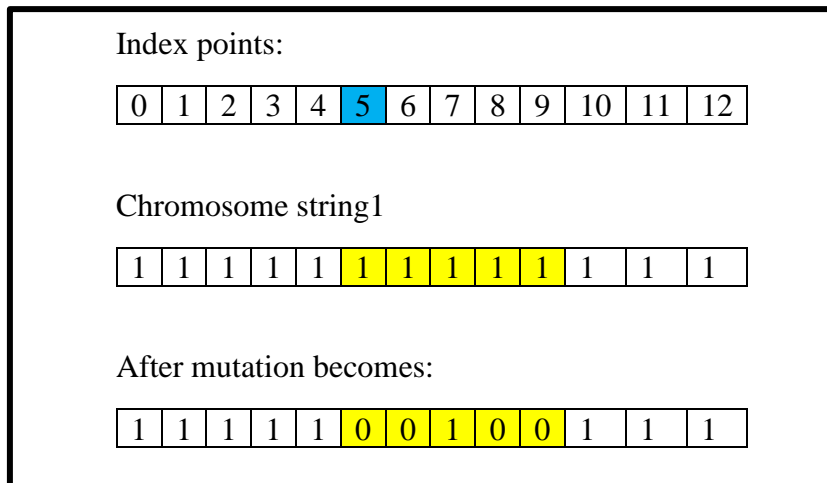


Figure12: Mutation function

However, since there is the possibility that the chromosome string may remain unmutated without any bits being flipped if the original chromosome string had the same switch1 string “00100” at index position 5. Therefore, to deal with removal of unmutated strings, the `RemoveIdenticalString` function is implemented. `RemoveIdenticalString` function takes in 2 parameters namely `List<string> resultList` and `List<string> Generationlist` where the chromosome strings which are fed into `Mutation` function are compared to the strings returned by `Mutation` function so that that if there is an identical string found in both the lists then it can be concluded that the chromosome string present in the `List<string> resultList` is not mutated and can be removed. `RemoveIdenticalString` function returns the list of mutated chromosome strings only to the main program discarding all the rest which were not mutated.

Thus, the mutation operation is implemented by `Mutation` function and `RemoveIdenticalString` function together as shown in the code snippet in Appendix E.

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

After the crossover and mutation operations are implemented the, the lists returned by these operations are added to “Generationlist” and fitness is evaluated on all the strings and then sorted by sending the “Generationlist” and “fitnesseslist1” to `Selection2` function as shown in the generation loop code snippet below.

Figure13: Code snippet for Generation loop

```
List<string> runPopulationlist = new List<string>();
List<string> OptimumPopulationlist = new List<string>();
List<string> MutatePopulationlist = new List<string>();

List<string> Generationlist = new List<string>();

Generationlist.AddRange(testPopulationlist);

int generation = 10;

// continuously generate populations until number of iterations is met.
for (int iter = 1; iter < generation; ++iter)
{

    if (iter == generation) break; // condition for breaking out of generation loop.

    while (Generationlist.Count <= POPULATION_SIZE)
    {

        // crossover

        // Instantiating random number generator
        var random1 = new Random();

        //OptimumPopulationlist which will store randomly selected parent strings as per calculations for
crossover
        for (int c = 0; c < 10; c++) // random 10 chromosome strings (n = 10)
```

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

```

    {
        //generating a random index
        int index1 = random1.Next(Generationlist.Count);
        // adding the chromosome string present at randomly generated index in runPopulationlist to
OptimumPopulationlist
        OptimumPopulationlist.Add(Generationlist[index1]);
    }

    int[] array = { 5, 10, 15, 20, 25, 30, 35, 40, 45, 50 }; // array of crossover index points
    string[] list = OptimumPopulationlist.ToArray();

    string[] result = crossover(list, array); // send to crossover function

    List<string> crossoverResultlist = new List<string>(result);

    runPopulationlist.AddRange(crossoverResultlist); //crossoverResultlist has 900 chromosomees for
10 randomly selected chromosome strings from OptimumPopulationlist (from top 5% of test Populationlist of 1000
population size)

    // Mutation

    Program p = new Program();
    List<string> resultlist = p.Mutation(Generationlist); // strings in resultlist from 1 randomly
selected string // resultlist has 50 strings
    //function to remove chromosomes strings in resultlist which are identical to
OptimumPopulationlist
    RemoveIdenticalString(resultlist, Generationlist); //string is reference type, so resultlist is
updated on ReturnIdenticalString

```

```

//MutatePopulationlist which will store randomly selected 50 mutated chromosome strings for 1000
population
50 for mutation
    for (int m = 0; m < 50; m++) // random 1 string from runPopulationlist so 1 x 50 combinations is
    {
        //generating a random index
        int index1 = random1.Next(resultlist.Count);
        // adding the chromosome string present at randomly generated index in runPopulationlist to
        MutatePopulationlist
        MutatePopulationlist.Add(resultlist[index1]);
        // 50 strings generated from one randomly selectedstring
    }

    resultlist.Clear();

//*****

    Generationlist.AddRange(MutatePopulationlist);

    Generationlist.AddRange(runPopulationlist);

}

List<int> fitnesseslist1 = new List<int>(Generationlist.Count);

// calculate fitness for test population.
for (int i = 0; i < 1; ++i) // this loop evaluates the fitness grade of the engine configurations
chromosome string
{
    FitFunc generationtest1 = new FitFunc();
    foreach (string chromosomeString in Generationlist)
    {
        int gradeValue1 = generationtest1.evalFunc(chromosomeString);
    }
}

```

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

```

        fitnesseslist1.Add(gradeValue1);
    }

}

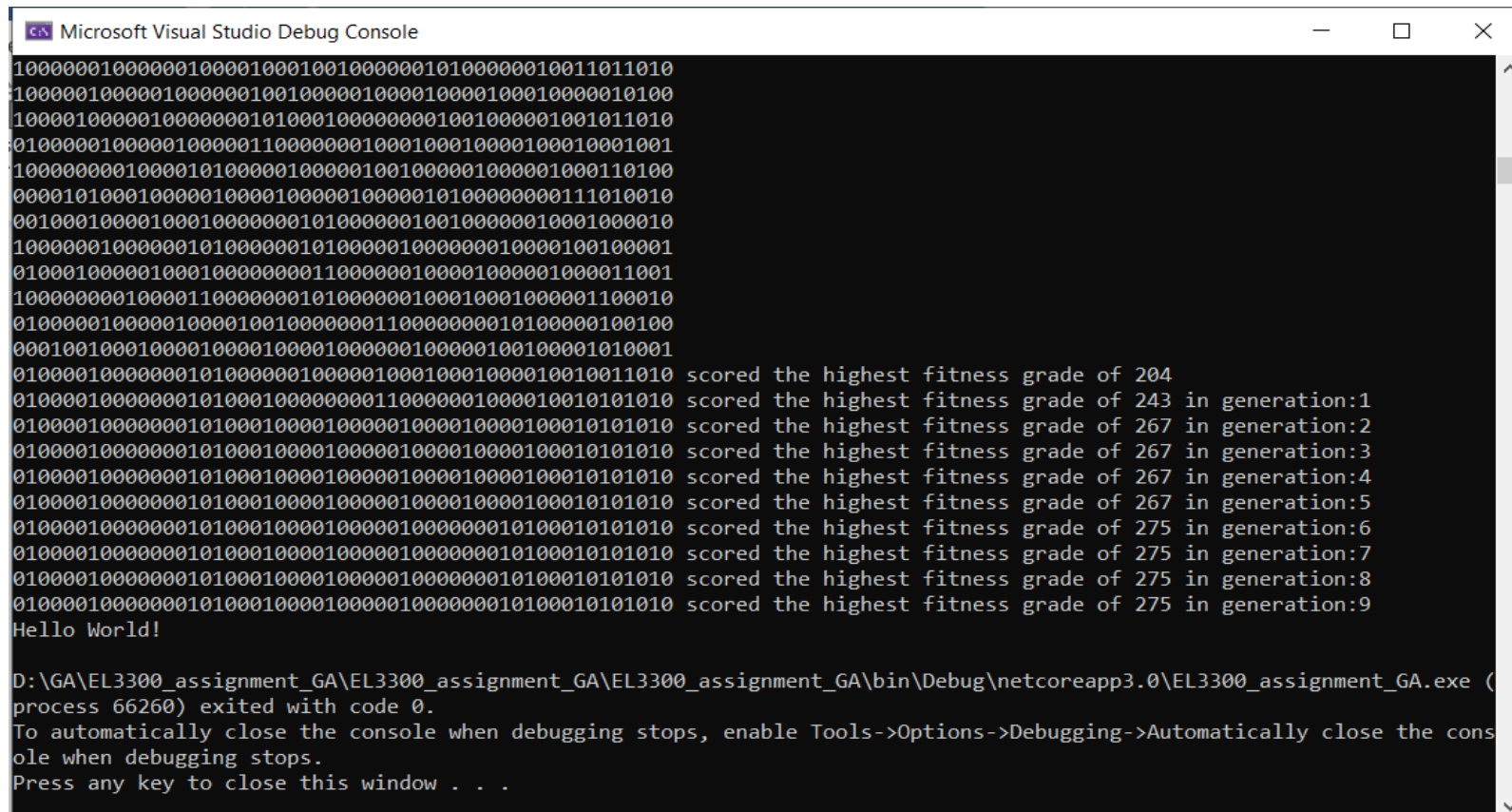
List<string> topScorerList1 = Selection2(Generationlist, fitnesseslist1);
for (int i = 0; i < fitnesseslist1.Count; i++)
{
    if (Generationlist[i] == topScorerList1[0])
    {
        Console.WriteLine(topScorerList1[0] + " scored the highest fitness grade of " +
fitnesseslist1[i] + " in generation:" + iter);
        break;
    }
}

Generationlist = topScorerList1; //now contain only the top 10% student names
                                //Clear the contents of the gradeslist completely and keep only

fitnesseslist1.Clear(); // clear this list after each generation loop.
runPopulationlist.Clear();
MutatePopulationlist.Clear();
}

```

Figure14: Screenshot of output for generation loop for population size of 1000.



```
Microsoft Visual Studio Debug Console

10000001000000100001000100000010100000010011011010
10000010000010000001001000001000010000100010000010100
10000100000100000001010001000000001001000001001011010
01000001000001000001100000001000100010000100010001001
10000000010000101000001000001001000001000001000110100
0000101000100000100001000001000001010000000111010010
00100010000100010000000101000000100100000010001000010
10000001000000101000000101000001000000010000100100001
01000100000100010000000011000000100001000001000011001
10000000010000110000000101000000100010001000001100010
010000100000100001001000000011000000010100000100100
00010010001000010000100001000000100000100100001010001
01000010000000101000000100000100010001000010010011010
0100001000000010100010000000110000001000010010101010 scored the highest fitness grade of 204
01000010000000010100010000000110000001000010010101010 scored the highest fitness grade of 243 in generation:1
010000100000000101000100001000001000010000100010101010 scored the highest fitness grade of 267 in generation:2
010000100000000101000100001000001000010000100010101010 scored the highest fitness grade of 267 in generation:3
010000100000000101000100001000001000010000100010101010 scored the highest fitness grade of 267 in generation:4
010000100000000101000100001000001000010000100010101010 scored the highest fitness grade of 267 in generation:5
010000100000000101000100001000001000000010100010101010 scored the highest fitness grade of 275 in generation:6
010000100000000101000100001000001000000010100010101010 scored the highest fitness grade of 275 in generation:7
010000100000000101000100001000001000000010100010101010 scored the highest fitness grade of 275 in generation:8
010000100000000101000100001000001000000010100010101010 scored the highest fitness grade of 275 in generation:9
Hello World!

D:\GA\EL3300_assignment_GA\EL3300_assignment_GA\EL3300_assignment_GA\bin\Debug\netcoreapp3.0\EL3300_assignment_GA.exe (
process 66260) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the cons
ole when debugging stops.
Press any key to close this window . . .
```

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

3. Testing

The different population sizes were considered using the crossover combination formula. But population size of 1000 served the purpose.

Since

$$N = K * n * (n - 1)$$

Where, N = Number of crossover offspring combinations

K = Crossover Index points in this case it is 10 since we have 10 index points which we apply single crossover.

n = Number of parent chromosome strings randomly selected from “Generationlist” and stored in “OptimumPopulationlist”.

The above formula is derived as follows:

For, permutation of n parent chromosome strings taken two at a time is = $n!/(n-2)! \dots [2]$

$$\begin{aligned} &= \frac{n * (n - 1) * (n - 2)!}{(n - 2)!} \\ &= n * (n - 1) \end{aligned}$$

900 is selected after considering the different combinations of two consecutive number product alternatives available as shown in

Appendix F. And, $1000 - 900 = 100$. Here, 900 is the number of crossover offspring combinations i.e., N in the above formula.

This 100 is divided by 2 so 50 from mutation and rest 50 from initial population. So optimal 5% of the population would be considered.

And then we need 50 mutation strings only to be added to the “Generationlist” so that “Generationlist” always has a constant population size of 1000. So, we need to find how many random parent chromosomes ‘n’ we need to select so that 900 combinations are generated.

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

Substituting the values we have determined so far for crossover combinations formula above,

$$900 = 10 * n * (n - 1)$$

We need to find n using quadratic formula as follows:

$$90 = n * (n - 1)$$

Thus, solving for n we get, n = 10 or n = -9 and n cannot be a negative number. Therefore we have n = 10.

So 10 random chromosome strings are selected from “Generationlist” and stored in “OptimumPopulationlist”.

Table1: Summary of the testing (refer to Appendix F to see test and trial calculations for different population sizes)

Sr. No.	Chromosome string	Fitness Grade	POPULATION_SIZE	iterations	Crossover rate	Mutation rate
1.	0100001000000001010001000010000010000000010100010101010	275	1000	10	90% (900)	5% (50)
2.	0100001000000001010001000000100010000000010100010101010	270	1000	5	90% (900)	5% (50)
3.	01000010000000010100010000100000100001000010001010101010	267	1000	4	90% (900)	5% (50)
4.	1000001000000001010001000000100010000000010100010101010	275	1000	10	72% (720)	14% (140)
5.	01000010000000010100010000001000100000100010001010101010	268	1000	7	72% (720)	14% (140)

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

6.	1000001000000001010000100000100010000000010100010101010	255	1000	4	72% (720)	14% (140)
7.	010000100000000101000100001000001000001000100010101010	273	200	20	60% (120)	20% (40)

4. Results and discussions

From the above testing it is clear that the highest fitness value of 275 can be achieved by the following binary encoding for engine configuration as shown in testing no. 1 and 4 :

01000 01000 00001 01000 10000 10000 01000 00001 01000 10101 010 → 275

The table below summarises the decoded the engine configuration states of the chromosome string for all the 15 parameters to achieve maximum optimal fitness value of 275.

Table2: Summary of the Decoding the engine configuration of the chromosome of all the 15 parameters to achieve maximum optimal fitness value of 275.

Sr. No.	Parameter	Binary encoding for the optimal engine configuration	engine configuration
1	External Temperature (temperature in the engine housing).	01000	Low
2	Internal Temperature (temperature inside the engine).	01000	Low
3	Cylinder Pressure.	00001	Very High

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

4	Value Opening Pressure.	01000	Low
5	Load Torque.	10000	Very Low
6	NOx Emissions (Nitrous Oxide Emissions)	10000	Very Low
7	CO Emissions (Carbon Monoxide Emissions)	01000	Low
8	HC Emissions (Hydrocarbon Emissions)	00001	Very High
9	PM Emissions (Particle Matter Emissions)	01000	Low
10	UCLan developed Electronic Timing Device.	1	Device present
11	UCLan developed Fuel Flow Device	0	Device absent
12	UCLan developed Emissions Limiter.	1	Device present
13	UCLan developed Battery Management System.	0	Device absent
14	UCLan developed Air Flow Management Device.	1	Device present
15	Fuel Injection Timings.	010	Medium

Conclusion

The project has successfully determined the engine configuration of all the 15 parameters to obtain the optimal fitness value of 275 by designing and implementing a simple genetic algorithm from scratch using C# language in a Console application using the given .dll library.

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

References

1. Mallawaarachchi V. Introduction to Genetic Algorithms — Including Example Code .Available from: <https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3> [Accessed 23rd February 2020].
2. Arora N. Permutation and Combination. Available from: <https://www.geeksforgeeks.org/permutation-and-combination/>[Accessed 3rd March 2020].
3. Jacobson L., Kanber B. Genetic Algorithms in Java Basics – Apress Source Code. Available from: https://github.com/apress/genetic-algorithms-in-java-basics_[Accessed 21st February 2020].

Appendix

Appendix A

List of engine configuration parameters

- 1) External Temperature (temperature in the engine housing).
- 2) Internal Temperature (temperature inside the engine).
- 3) Cylinder Pressure.
- 4) Valve Opening Pressure.
- 5) Load Torque.
- 6) NOx Emissions (Nitrous Oxide Emissions)
- 7) CO Emissions (Carbon Monoxide Emissions)
- 8) HC Emissions (Hydrocarbon Emissions)
- 9) PM Emissions (Particle Matter Emissions)
- 10) UCLan developed Electronic Timing Device.
- 11) UCLan developed Fuel Flow Device.
- 12) UCLan developed Emissions Limiter.
- 13) UCLan developed Battery Management System.
- 14) UCLan developed Air Flow Management Device.
- 15) Fuel Injection Timings.

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

Appendix B

Selection1 Function code snippet

```
/**
    // Sorting top 5% chromosomees according to fitness grade function

    /**
    static List<string> Selection1(IEnumerable<string> testPopulationlist, IEnumerable<int> fitnesseslist)
    {
        int totalChromosomes = testPopulationlist.Count();

        //POPULATION_SIZE = 1000 here totalchromosomes
        int Percent = (5 * totalChromosomes) / 100; // top 5%

        //Fitness Grades attained by each chromosome string are sorted
        List<int> sortedfitnessGradesList = fitnesseslist.OrderByDescending(x => x).ToList();

        List<string> topChromosomesList = new List<string>();

        //Finding the top 10% fitness grades in the list and corresponding chromosome strings who attained those top 10%
        fitness grades.
        for (int i = 0; i < Percent;)
        {
            //getting top element
            int grade = sortedfitnessGradesList[i];

            for (int j = 0; j < totalChromosomes; j++)
```

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

```

    {
        //searching element in unsorted list
        if (fitnesseslist.ElementAt(j) == grade)
        {
            //add chromosome string based on index of fitness grade
            topChromosomesList.Add(testPopulationlist.ElementAt(j));
            i++;
        }
        //checking if we reached optimal x%
        if (i >= Percent)
            break;
    }
}

//print the name of the student attaining the highest grade in the console along with the highest grade obtained
for (int i = 0; i < Percent; i++)
{
    //getting top element
    string chromosomegenes = topChromosomesList[i];
    for (int j = 0; j < totalChromosomes; j++)
    {
        //searching element in unsorted list
        if (testPopulationlist.ElementAt(j) == chromosomegenes)
        {
            //Console.WriteLine("Chromosome : " + chromosomegenes + "\t\tFitness Score:" + fitnesseslist.ElementAt(j));
            i++;
        }
        //checking if we reached optimal x%
        if (i >= Percent)
            break;
    }
}

//returning list of optimal x chromosome strings in a list.
return topChromosomesList;

```

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

```
} // end of sorting function
```

Appendix C

Selection2 Function code snippet

```
/**
    // Sorting top 5% chromosomees according to fitness grade function inside generation loop
*/
/**
static List<string> Selection2(IEnumerable<string> Generationlist, IEnumerable<int> fitnesseslist1)
{
    int totalChromosomes = Generationlist.Count();

    //POPULATION_SIZE = 1000 here totalchromosomes
    int Percent = (5 * totalChromosomes) / 100; // top 5%

    //Fitness Grades attained by each chromosome string are sorted
    List<int> sortedfitnessGradesList = fitnesseslist1.OrderByDescending(x => x).ToList();

    List<string> topChromosomesList = new List<string>();

    //Finding the top 10% fitness grades in the list and corresponding chromosome strings who attained those top 10%
    fitness grades.
    for (int i = 0; i < Percent;)
    {
        //getting top element
    }
}
```

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020


```

        int grade = sortedfitnessGradesList[i];

        for (int j = 0; j < totalChromosomes; j++)
        {
            //searching element in unsorted list
            if (fitnesseslist1.ElementAt(j) == grade)
            {
                //add chromosome string based on index of fitness grade
                topChromosomesList.Add(Generationlist.ElementAt(j));
                i++;
            }
            //checking if we reached optimal x%
            if (i >= Percent)
                break;
        }
    }

    //print the name of the student attaining the highest grade in the console along with the highest grade obtained
    for (int i = 0; i < Percent;)
    {
        //getting top element
        string chromosomegenes = topChromosomesList[i];
        for (int j = 0; j < totalChromosomes; j++)
        {
            //searching element in unsorted list
            if (Generationlist.ElementAt(j) == chromosomegenes)
            {
                //Console.WriteLine("Chromosome : " + chromosomegenes + "\t\tFitness Score:" +
                fitnesseslist1.ElementAt(j));
                i++;
            }
            //checking if we reached optimal x%
            if (i >= Percent)
                break;
        }
    }
}

```

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

```

    }
    //returning list of optimal x chromosome strings in a list.
    return topChromosomesList;

} // end of sorting function

```

Appendix D

Crossover Functions Code snippet

```

//*****
// Crossover function
//*****

private const int NUM_CHAR = 53;

// this function returns the combination of two strings
private static string[] combine(string str1, string str2, int index)
{
    string[] result = { "", "" }; // initializing the array of strings to be returned
    string first, second; // these strings are used for breaking and recombining the two strings

    // creating the first combination
    first = str1.Substring(0, index); // take the first part of the first string
    second = str2.Substring(index, str2.Length - index); // take the rest from the second one
    result[0] = first + second; // concatenate both parts

    // creating the second combination
    first = str2.Substring(0, index); // take the first part of the second string

```

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

```

        second = str1.Substring(index, str1.Length - index); // take the rest from the first string
        result[1] = first + second; // concatenate both parts
        return result; // return the resulting array of strings
    }

    private static string[] crossover(string[] list, int[] IndexArray)
    {
        string[] temp = new string[0]; // temporary variable for storing the result from 'combine' method
        string[] resultArray = new string[0]; // initializing the array of strings to be returned
        for (int k = 0; k < IndexArray.Length; k++) // for each index in integer array
        {
            for (int i = 0; i < list.Length; i++) // try all the combinations of strings
            {
                for (int j = i; j < list.Length; j++)
                {
                    if (i != j) // to avoid combining a string with itself
                    {
                        temp = combine(list[i], list[j], IndexArray[k]); // temporary array that stores combinations
                        Array.Resize(ref resultArray, resultArray.Length + temp.Length); // changing the size of the array for
adding
                        // the array below keeps collecting the combinations, 2 combinations at a time
arrays
                        Array.Copy(temp, 0, resultArray, resultArray.Length - temp.Length, temp.Length); // merging the two
                    }
                }
            }
        }
        // collection is complete
        return resultArray; // returning the results
    }
}

```

Appendix E

Mutation Functions Code snippet

```
/**
 * Mutation function
 */

public const string switch1 = "00100";
public const string switch2 = "10000";
public const string switch3 = "01000";
public const string switch4 = "00010";
public const string switch5 = "00001";

public const string switch6 = "001";
public const string switch7 = "010";
public const string switch8 = "100";

public const string switch9 = "00000";
public const string switch10 = "11111";

public List<string> Mutation(List<string> Generationlist)
{
    // Declaring a list of result strings
    List<string> resultlist = new List<string>();
    int index = 0;

    // Looping through the list
    foreach (string str in Generationlist)
```

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020

```

{
    // Getting the position from which characters have to be replaced
    int pos1 = 5;
    int pos2 = 10;
    int pos3 = 15;
    int pos4 = 20;
    int pos5 = 25;
    int pos6 = 30;
    int pos7 = 35;
    int pos8 = 40;

    int pos9 = 45;

    int last3pos = 50;

    /*
    *   Replacing the 5 characters and storing in a new string
    *   */

    // Replacing replacingString1 = "00100" in different positions
    string s1 = str.Substring(0, pos1) + switch1 + str.Substring(pos1 + 5);
    string s2 = str.Substring(0, pos2) + switch1 + str.Substring(pos2 + 5);
    string s3 = str.Substring(0, pos3) + switch1 + str.Substring(pos3 + 5);
    string s4 = str.Substring(0, pos4) + switch1 + str.Substring(pos4 + 5);
    string s5 = str.Substring(0, pos5) + switch1 + str.Substring(pos5 + 5);
    string s6 = str.Substring(0, pos6) + switch1 + str.Substring(pos6 + 5);
    string s7 = str.Substring(0, pos7) + switch1 + str.Substring(pos7 + 5);
    string s8 = str.Substring(0, pos8) + switch1 + str.Substring(pos8 + 5);

    // Engine Configuration parameters from 10 to 14 at index position 45 to 49 in chromosome string are mutated as follows using pos9
    string s9 = str.Substring(0, pos9) + switch1 + str.Substring(pos9 + 5);

    // Replacing replacingString2 = "10000" in different positions

```

```
string s10 = str.Substring(0, pos1) + switch2 + str.Substring(pos1 + 5);
string s11 = str.Substring(0, pos2) + switch2 + str.Substring(pos2 + 5);
string s12 = str.Substring(0, pos3) + switch2 + str.Substring(pos3 + 5);
string s13 = str.Substring(0, pos4) + switch2 + str.Substring(pos4 + 5);
string s14 = str.Substring(0, pos5) + switch2 + str.Substring(pos5 + 5);
string s15 = str.Substring(0, pos6) + switch2 + str.Substring(pos6 + 5);
string s16 = str.Substring(0, pos7) + switch2 + str.Substring(pos7 + 5);
string s17 = str.Substring(0, pos8) + switch2 + str.Substring(pos8 + 5);
string s18 = str.Substring(0, pos9) + switch2 + str.Substring(pos9 + 5);

// Replacing replacingString3 = "01000" in different positions
string s19 = str.Substring(0, pos1) + switch3 + str.Substring(pos1 + 5);
string s20 = str.Substring(0, pos2) + switch3 + str.Substring(pos2 + 5);
string s21 = str.Substring(0, pos3) + switch3 + str.Substring(pos3 + 5);
string s22 = str.Substring(0, pos4) + switch3 + str.Substring(pos4 + 5);
string s23 = str.Substring(0, pos5) + switch3 + str.Substring(pos5 + 5);
string s24 = str.Substring(0, pos6) + switch3 + str.Substring(pos6 + 5);
string s25 = str.Substring(0, pos7) + switch3 + str.Substring(pos7 + 5);
string s26 = str.Substring(0, pos8) + switch3 + str.Substring(pos8 + 5);
string s27 = str.Substring(0, pos9) + switch3 + str.Substring(pos9 + 5);

// Replacing replacingString4 = "00010" in different positions
string s28 = str.Substring(0, pos1) + switch4 + str.Substring(pos1 + 5);
string s29 = str.Substring(0, pos2) + switch4 + str.Substring(pos2 + 5);
string s30 = str.Substring(0, pos3) + switch4 + str.Substring(pos3 + 5);
string s31 = str.Substring(0, pos4) + switch4 + str.Substring(pos4 + 5);
string s32 = str.Substring(0, pos5) + switch4 + str.Substring(pos5 + 5);
string s33 = str.Substring(0, pos6) + switch4 + str.Substring(pos6 + 5);
string s34 = str.Substring(0, pos7) + switch4 + str.Substring(pos7 + 5);
string s35 = str.Substring(0, pos8) + switch4 + str.Substring(pos8 + 5);
string s36 = str.Substring(0, pos9) + switch4 + str.Substring(pos9 + 5);

// Replacing replacingString5 = "00001" in different positions
```

```

string s37 = str.Substring(0, pos1) + switch5 + str.Substring(pos1 + 5);
string s38 = str.Substring(0, pos2) + switch5 + str.Substring(pos2 + 5);
string s39 = str.Substring(0, pos3) + switch5 + str.Substring(pos3 + 5);
string s40 = str.Substring(0, pos4) + switch5 + str.Substring(pos4 + 5);
string s41 = str.Substring(0, pos5) + switch5 + str.Substring(pos5 + 5);
string s42 = str.Substring(0, pos6) + switch5 + str.Substring(pos6 + 5);
string s43 = str.Substring(0, pos7) + switch5 + str.Substring(pos7 + 5);
string s44 = str.Substring(0, pos8) + switch5 + str.Substring(pos8 + 5);
string s45 = str.Substring(0, pos9) + switch5 + str.Substring(pos9 + 5);

// from index 50 Fuel Injection Timings parameter is encoded which only uses three binary number sequences
string s46 = str.Substring(0, last3pos) + switch6 + str.Substring(last3pos + 3); // to flip the last three
chromosome characters to '001'
string s47 = str.Substring(0, last3pos) + switch7 + str.Substring(last3pos + 3); // to flip the last three
chromosome characters to '010'
string s48 = str.Substring(0, last3pos) + switch8 + str.Substring(last3pos + 3); // to flip the last three
chromosome characters to '100'

// for 45 to 49 index
string s49 = str.Substring(0, pos9) + switch9 + str.Substring(pos9 + 5);
string s50 = str.Substring(0, pos9) + switch10 + str.Substring(pos9 + 5);

// Adding that replaced string to the resultlist
resultlist.Add(s1);
resultlist.Add(s2);
resultlist.Add(s3);
resultlist.Add(s4);
resultlist.Add(s5);
resultlist.Add(s6);
resultlist.Add(s7);
resultlist.Add(s8);
resultlist.Add(s9);

```

```
resultlist.Add(s10);  
resultlist.Add(s11);  
resultlist.Add(s12);  
resultlist.Add(s13);  
resultlist.Add(s14);  
resultlist.Add(s15);  
resultlist.Add(s16);  
resultlist.Add(s17);  
resultlist.Add(s18);  
resultlist.Add(s19);  
resultlist.Add(s20);  
resultlist.Add(s21);  
resultlist.Add(s22);  
resultlist.Add(s23);  
resultlist.Add(s24);  
resultlist.Add(s25);  
resultlist.Add(s26);  
resultlist.Add(s27);  
resultlist.Add(s28);  
resultlist.Add(s29);  
resultlist.Add(s30);  
resultlist.Add(s31);  
resultlist.Add(s32);  
resultlist.Add(s33);  
resultlist.Add(s34);  
resultlist.Add(s35);  
resultlist.Add(s36);  
resultlist.Add(s37);  
resultlist.Add(s38);  
resultlist.Add(s39);  
resultlist.Add(s40);  
resultlist.Add(s41);  
resultlist.Add(s42);  
resultlist.Add(s43);  
resultlist.Add(s44);
```

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020


```

        resultlist.Add(s45);
        resultlist.Add(s46);
        resultlist.Add(s47);
        resultlist.Add(s48);
        resultlist.Add(s49);
        resultlist.Add(s50);

        index++;
    }

    return resultlist;
}
//*****
// RemoveIdenticalString function

//*****

public static void RemoveIdenticalString(List<string> resultList, List<string> Generationlist)
{
    foreach (string tStr in Generationlist)
        for (int i = 0; i < resultList.Count; i++)
        {
            if (resultList[i] == tStr)
                resultList.Remove(tStr);
        }
    //Console.WriteLine("inside RemoveIdenticalString function");
}

```

Appendix F:

Test and trial calculations

Sr. No.	N (crossover offsprings combinations)	K (crossover index points)	n (number of randomly selected parent chromosomes)	(n-1)	Selected or not ?	Population size	Mutation strings	Optimal x% population
1	900 (i.e., 90% of population size)	10	10	9	yes	1000	50 (i.e., 5%)	50 (i.e., 5%)
2	560 (i.e., 93.33% of population size)	10	8	7	No	600	20 (i.e., 3.33%)	20 (i.e., 3.33%)
3	720 (i.e., 72% of population size)	10	9	8	yes	1000	140 (i.e., 14%)	140 (i.e., 14%)
4	120 (i.e., 60% of population size)	10	5	4	yes	200	40 (i.e., 20%)	40 (i.e., 20%)

Mrunal Prakash Gavali (G20753327)

Date: 8-3-2020