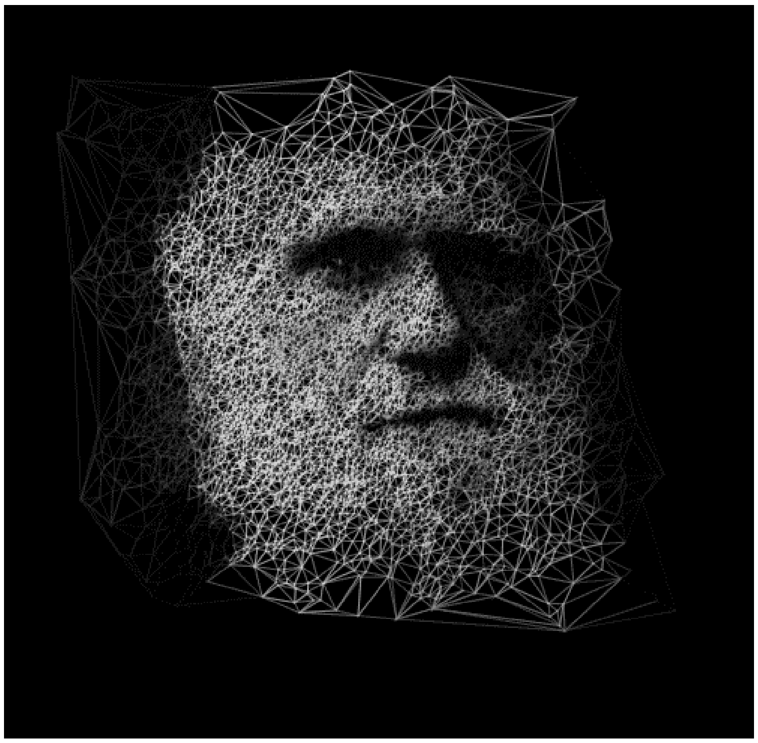
Evolution Image Generation

A process portrayed using Genetic Algorithm.



Rajat Jaswal – 001443168
Srishti Saboo – 001443101

Group 219

2018 Fall

INFO6205 Final Project Report

Table of Content

Problem Description

Human evolution is the lengthy process of change by which people originated from apelike ancestors.

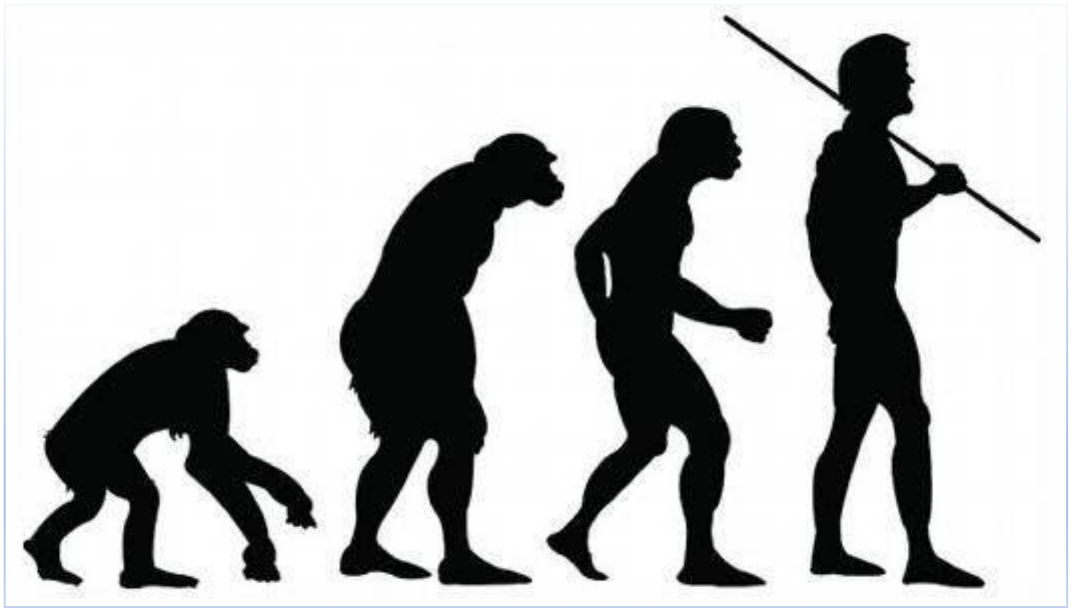
Whenever there was mutation between the Creatures, the gene having the better fitness transfers in the offspring. The process took six million years to transform the creature like Australopithecus, into the creature like a Homo Sapiens.

After these many years, we know the complete transformation of human evolution. We know about the result we are expecting from the problem. So,

the result which we expect is of fitness 100%.

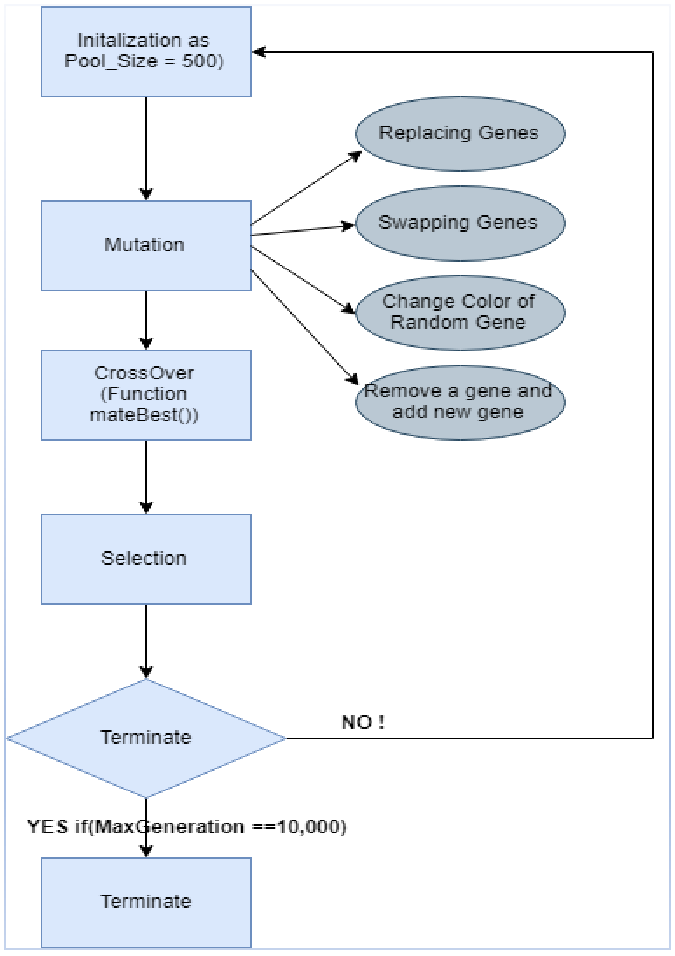
We have the result (the mutation image of fitness 100%), we are starting the process of developing the original image through random set of characters/symbols of random colors and random fonts.

The original image is displayed below.



Genetic Algorithm in Action!

Genetic Evolution (GE) is a method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. It is a population based, stochastic function minimizer. It involves the following steps-
We took initial population of random 1800 (char_ count) characters as 500 (pool_size). And the objective function to be maximized was similarity of images. When the value of similarity of parent image and produced image is greater than 0.93, or when the generation count (MAX_GENERATIONS) crosses 10,000 the algorithm terminates. The basic flow of the algorithm is:


Initalization as Pool_Size = 500)
Mutation
Replacing Genes
Swapping Genes
Change Color of Random Gene
Remove a gene and add new gene
CrossOver (Function mateBest())
Selection
Terminate
NO !
YES if(MaxGeneration ==10,000)
Terminate

The Implementation Facet!

1) Individual:

An individual is the image consisting of thousands of random characters with different shapes and colors.
An individual organism consists of a genetic constitution that is a single character image or a ‘Genotype’ which has a constant encoding type. We represent that as below.

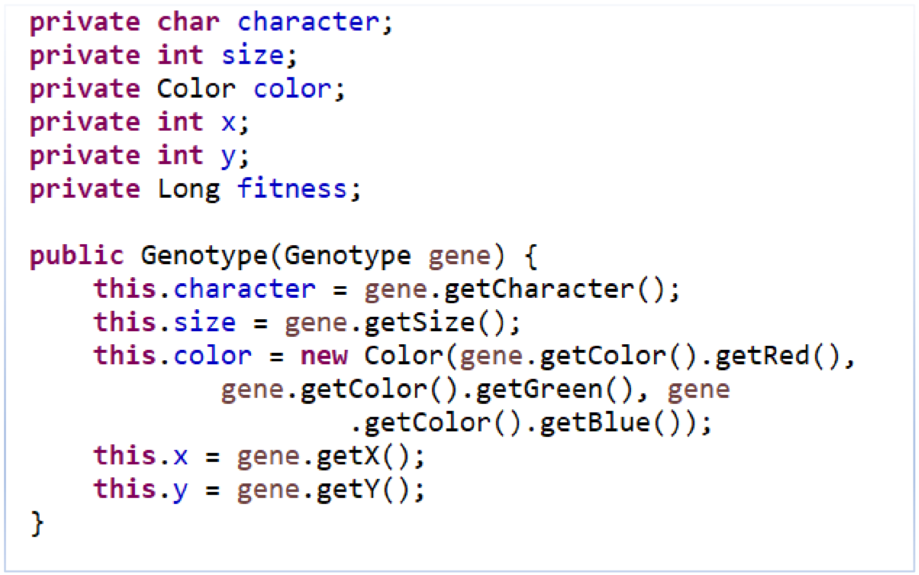
Genotype1

Z	40	🟥	300	50

Genotype2

$	32	🟦	154	1000

The Genotype is:

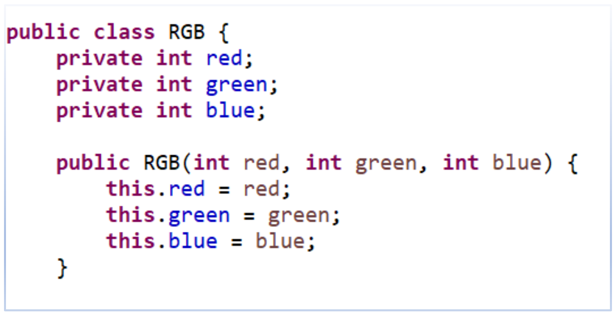


```java
private char character;
private int size;
private Color color;
private int x;
private int y;
private Long fitness;

public Genotype(Genotype gene) {
    this.character = gene.getCharacter();
    this.size = gene.getSize();
    this.color = new Color(gene.getColor().getRed(),
            gene.getColor().getGreen(), gene
                    .getColor().getBlue());
    this.x = gene.getX();
    this.y = gene.getY();
}
```

2) The Helper Classes:

a) RGB.java:

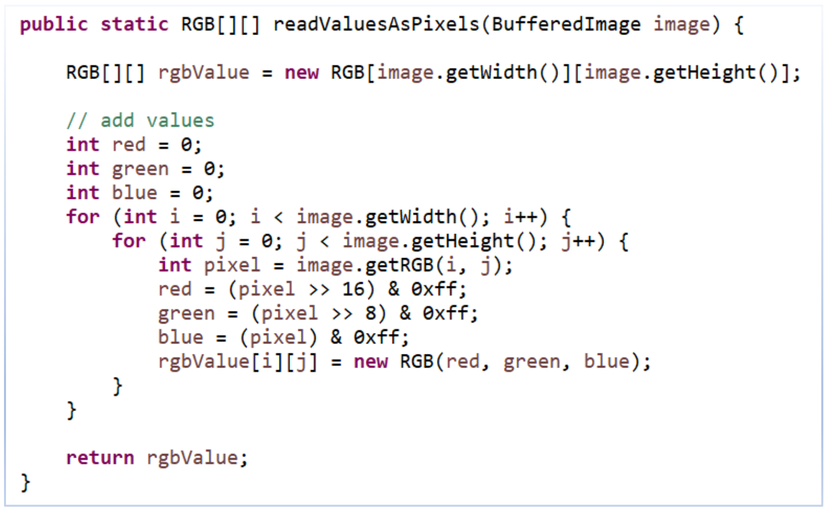
It represents the color code in RGB format. It is used to calculate the fitness in pixels.



```java
public class RGB {
    private int red;
    private int green;
    private int blue;

    public RGB(int red, int green, int blue) {
        this.red = red;
        this.green = green;
        this.blue = blue;
    }
```

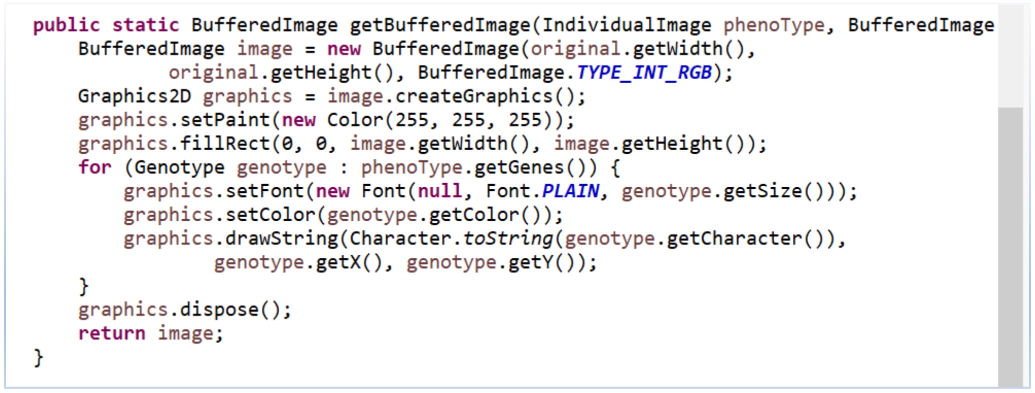
b) ProcessImage.java:

Read the image in the form of RGB matrix which is used to perform calculations based on pixels for further fitness.



```java
public static RGB[][] readValuesAsPixels(BufferedImage image) {

    RGB[][] rgbValue = new RGB[image.getWidth()][image.getHeight()];

    // add values
    int red = 0;
    int green = 0;
    int blue = 0;
    for (int i = 0; i < image.getWidth(); i++) {
        for (int j = 0; j < image.getHeight(); j++) {
            int pixel = image.getRGB(i, j);
            red = (pixel >> 16) & 0xff;
            green = (pixel >> 8) & 0xff;
            blue = (pixel) & 0xff;
            rgbValue[i][j] = new RGB(red, green, blue);
        }
    }

    return rgbValue;
}
```

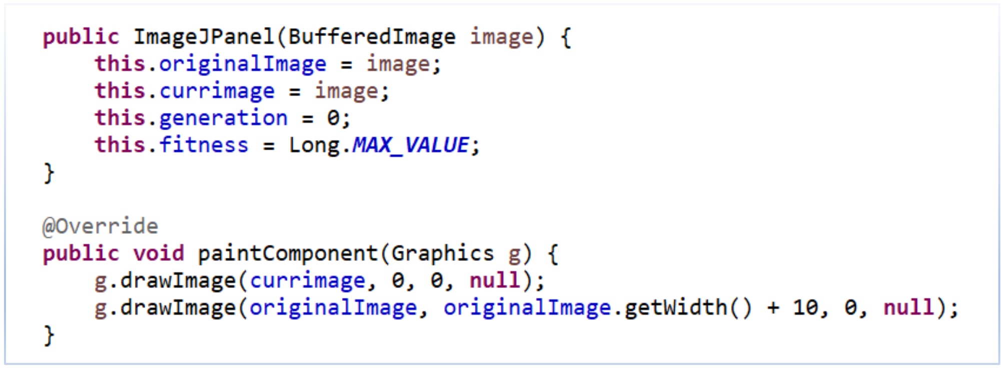
The below method in ProcessImage.java is converting the list of genotypes or an Individual into a BufferedImage which helps us to further visualize the algorithm better.



```
public static BufferedImage getBufferedImage(IndividualImage phenoType, BufferedImage
    BufferedImage image = new BufferedImage(original.getWidth(),
            original.getHeight(), BufferedImage.TYPE_INT_RGB);
    Graphics2D graphics = image.createGraphics();
    graphics.setPaint(new Color(255, 255, 255));
    graphics.fillRect(0, 0, image.getWidth(), image.getHeight());
    for (Genotype genotype : phenoType.getGenes()) {
        graphics.setFont(new Font(null, Font.PLAIN, genotype.getSize()));
        graphics.setColor(genotype.getColor());
        graphics.drawString(Character.toString(genotype.getCharacter()),
                genotype.getX(), genotype.getY());
    }
    graphics.dispose();
    return image;
}
```

c) ImageJPanel.java:

ImageJPanel is the interface to visualize the fittest individual after every generation.



```
public ImageJPanel(BufferedImage image) {
    this.originalImage = image;
    this.currimage = image;
    this.generation = 0;
    this.fitness = Long.MAX_VALUE;
}

@Override
public void paintComponent(Graphics g) {
    g.drawImage(currimage, 0, 0, null);
    g.drawImage(originalImage, originalImage.getWidth() + 10, 0, null);
}
```

3) **Main Classes**:

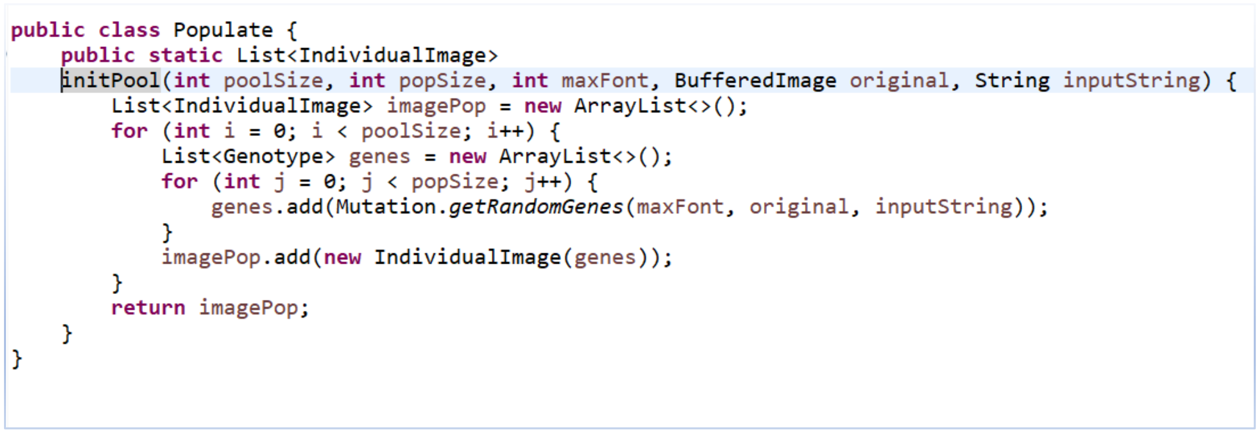
a) Main.java:

- The program starts from the main.java class where firstly we read an image and then converted the image into pixels.
- We initialized the population of 500 individuals of 1800 random characters, colors, size, and position each.
- We distributed the fitness evaluator according to the thread, each thread is computing fitness of different set of individuals each.
- During computing the fitness of individuals, we directly compute the fitness of each genotype.
- After finding about the fitness, we get the best fit individual and converted it into an image and display it through the JPanel.

- Until we reach our terminating condition i.e. fitness approximately 0.93, many generations are produced with fitness better than the previous one.

b) Populate.java:

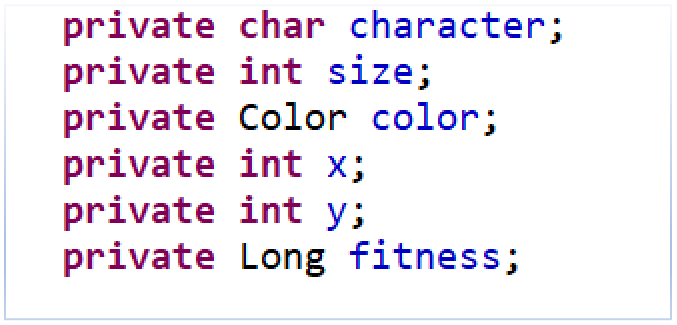
It generates a population of random 1800 characters as 500 individuals. The image returned by population function is of better fitness.



```java
public class Populate {
    public static List<IndividualImage>
    initPool(int poolSize, int popSize, int maxFont, BufferedImage original, String inputString) {
        List<IndividualImage> imagePop = new ArrayList<>();
        for (int i = 0; i < poolSize; i++) {
            List<Genotype> genes = new ArrayList<>();
            for (int j = 0; j < popSize; j++) {
                genes.add(Mutation.getRandomGenes(maxFont, original, inputString));
            }
            imagePop.add(new IndividualImage(genes));
        }
        return imagePop;
    }
}
```

c) Genotype.java:

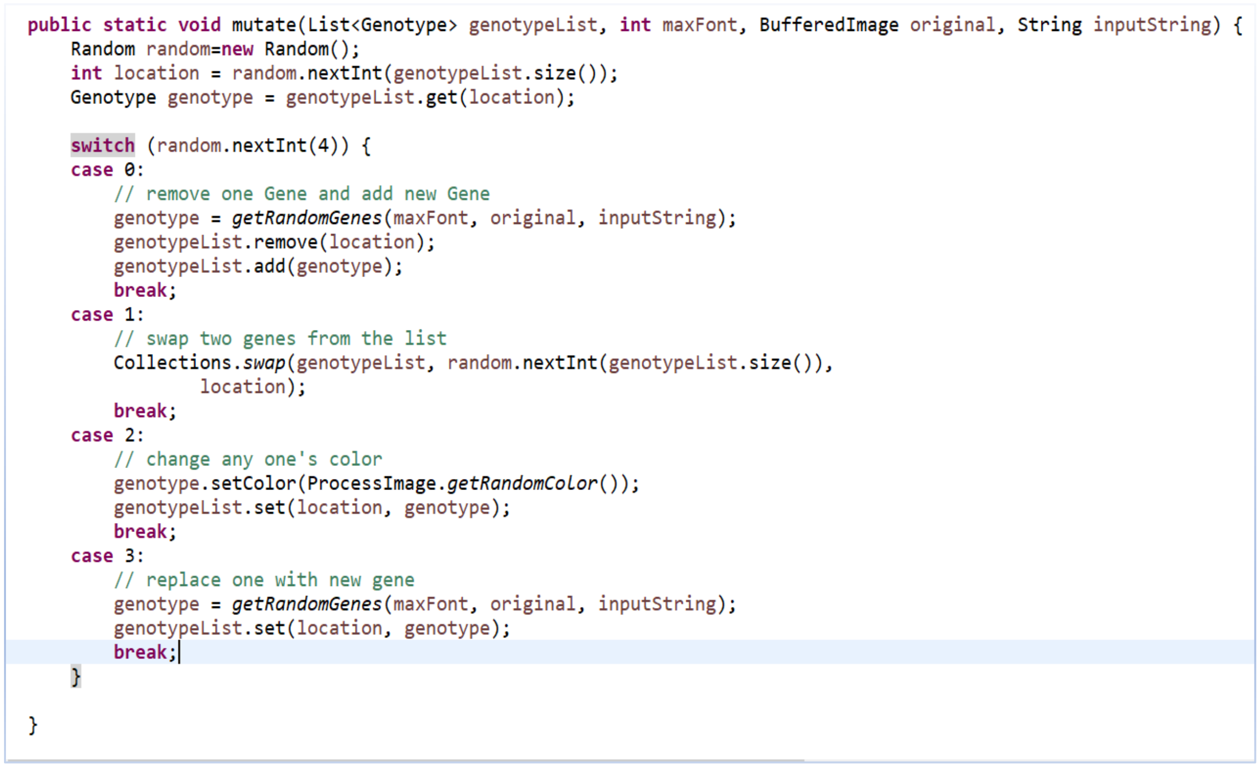
The genes with different characters, size, color, x-coordinate and y-coordinate are created in this class. The gene level fitness of the respective gene is also stored.



```java
private char character;
private int size;
private Color color;
private int x;
private int y;
private Long fitness;
```

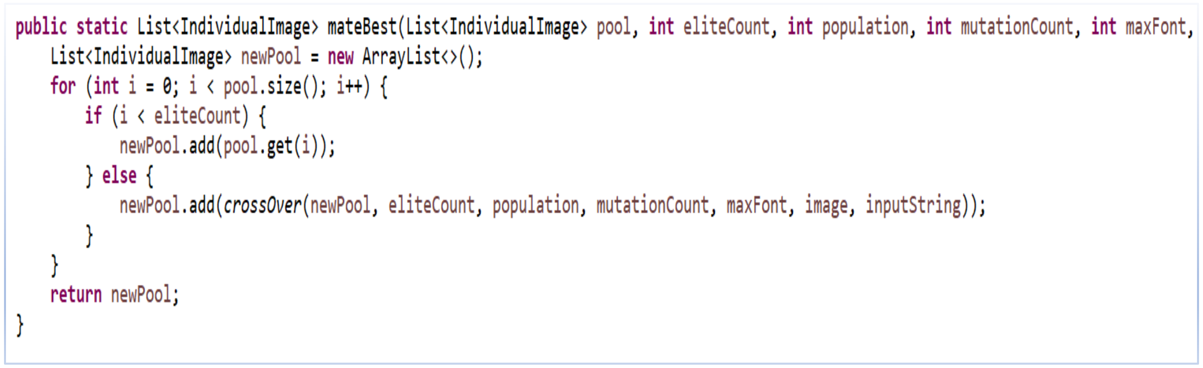
d) Mutation.java:

In mutation.java we are having list of gene which we are changing by performing few cases. These cases are like replacing a gene by another gene, swapping two genes from the list or changing any traits of the gene. By performing these cases are goal is to get a gene with the best fitness which provides the best image.



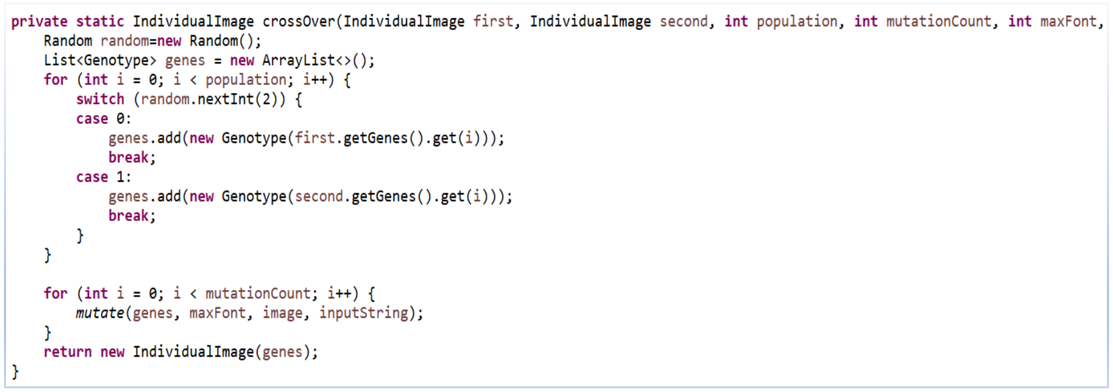
```java
public static void mutate(List<Genotype> genotypeList, int maxFont, BufferedImage original, String inputString) {
    Random random=new Random();
    int location = random.nextInt(genotypeList.size());
    Genotype genotype = genotypeList.get(location);

    switch (random.nextInt(4)) {
    case 0:
        // remove one Gene and add new Gene
        genotype = getRandomGenes(maxFont, original, inputString);
        genotypeList.remove(location);
        genotypeList.add(genotype);
        break;
    case 1:
        // swap two genes from the list
        Collections.swap(genotypeList, random.nextInt(genotypeList.size()),
                location);
        break;
    case 2:
        // change any one's color
        genotype.setColor(ProcessImage.getRandomColor());
        genotypeList.set(location, genotype);
        break;
    case 3:
        // replace one with new gene
        genotype = getRandomGenes(maxFont, original, inputString);
        genotypeList.set(location, genotype);
        break;
    }

}
```

We are performing Elitism for doing crossover, so that always the fittest generation breed in order to give us the best result. We have taken a pool of 10 elite individuals and then breeding through the elites.



```java
public static List<IndividualImage> mateBest(List<IndividualImage> pool, int eliteCount, int population, int mutationCount, int maxFont,
    List<IndividualImage> newPool = new ArrayList<>();
    for (int i = 0; i < pool.size(); i++) {
        if (i < eliteCount) {
            newPool.add(pool.get(i));
        } else {
            newPool.add(crossOver(newPool, eliteCount, population, mutationCount, maxFont, image, inputString));
        }
    }
    return newPool;
}
```

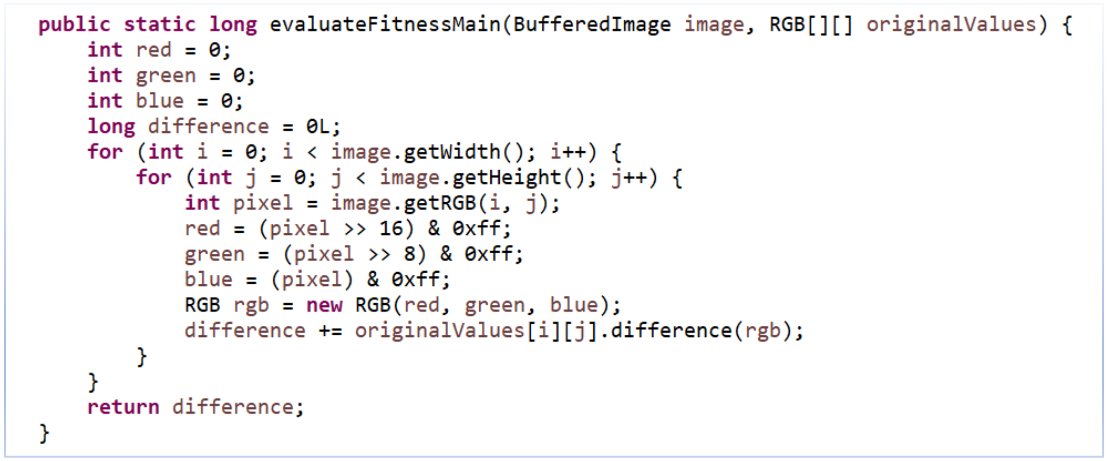
In the Crossover function defined in Mutation.java we are generating one child by randomly taking the traits of any of the two-elite parent.



```java
private static IndividualImage crossOver(IndividualImage first, IndividualImage second, int population, int mutationCount, int maxFont,
    Random random=new Random();
    List<Genotype> genes = new ArrayList<>();
    for (int i = 0; i < population; i++) {
        switch (random.nextInt(2)) {
        case 0:
            genes.add(new Genotype(first.getGenes().get(i)));
            break;
        case 1:
            genes.add(new Genotype(second.getGenes().get(i)));
            break;
        }
    }

    for (int i = 0; i < mutationCount; i++) {
        mutate(genes, maxFont, image, inputString);
    }
    return new IndividualImage(genes);
}
```

e) <u>FitnessFunction.java</u>:

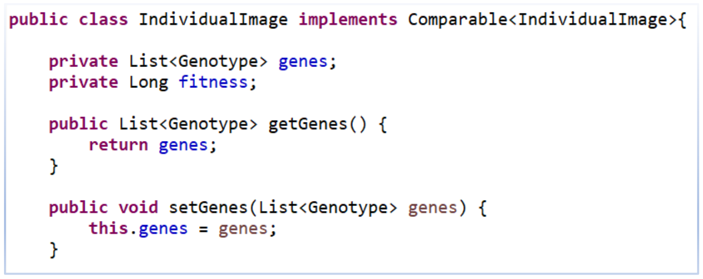
In this file we find out the difference of the RGB values of the original image and the image we just generated through randomGene selection. This fitness tells us how far the image is left for it to be breaded more.



```java
public static long evaluateFitnessMain(BufferedImage image, RGB[][] originalValues) {
    int red = 0;
    int green = 0;
    int blue = 0;
    long difference = 0L;
    for (int i = 0; i < image.getWidth(); i++) {
        for (int j = 0; j < image.getHeight(); j++) {
            int pixel = image.getRGB(i, j);
            red = (pixel >> 16) & 0xff;
            green = (pixel >> 8) & 0xff;
            blue = (pixel) & 0xff;
            RGB rgb = new RGB(red, green, blue);
            difference += originalValues[i][j].difference(rgb);
        }
    }
    return difference;
}
```

f) IndividualImage.java:

An IndividualImage file comprises of a collection of genes and a fitness of its own. This is further used to create a visible interface and is used to mutate and breed further.



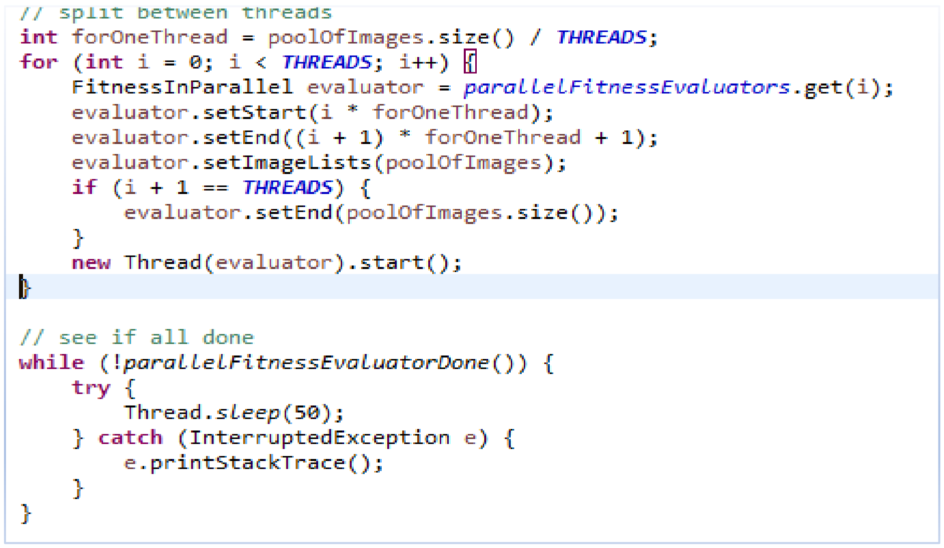
```java
public class IndividualImage implements Comparable<IndividualImage>{

    private List<Genotype> genes;
    private Long fitness;

    public List<Genotype> getGenes() {
        return genes;
    }

    public void setGenes(List<Genotype> genes) {
        this.genes = genes;
    }
```

Multithreading, the Cure!

Any huge task in this universe needs to be broken down into smaller tasks so that it takes less time to perform that huge task. We came across such similar problem while calculating fitness for hundreds and thousands of entities.

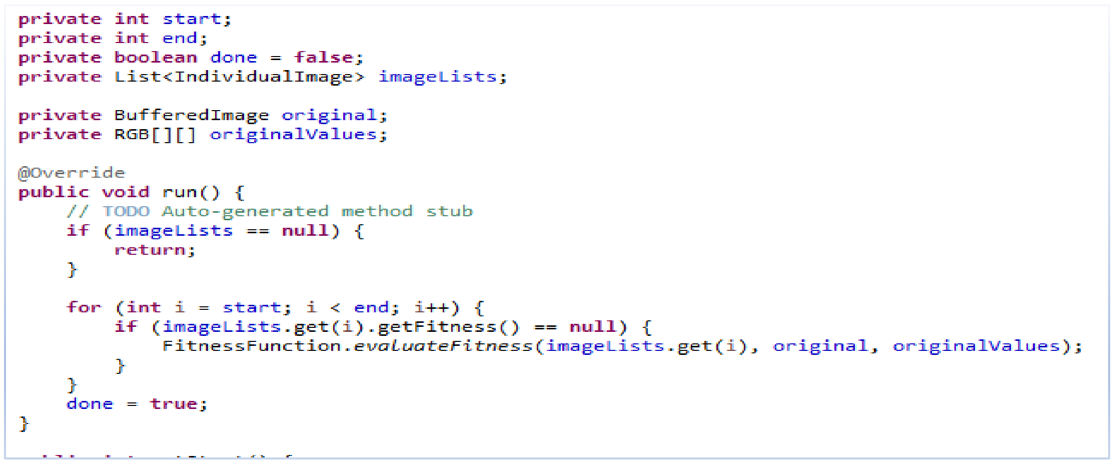
We solved this problem by the concept of multitasking where we initialized some threads and gave a thread each responsibility of solving a problem of multiple entities. This massively impacted on the run time and helped us reducing the overall time of reaching the optimal image.

fitnessofMultipleImages()



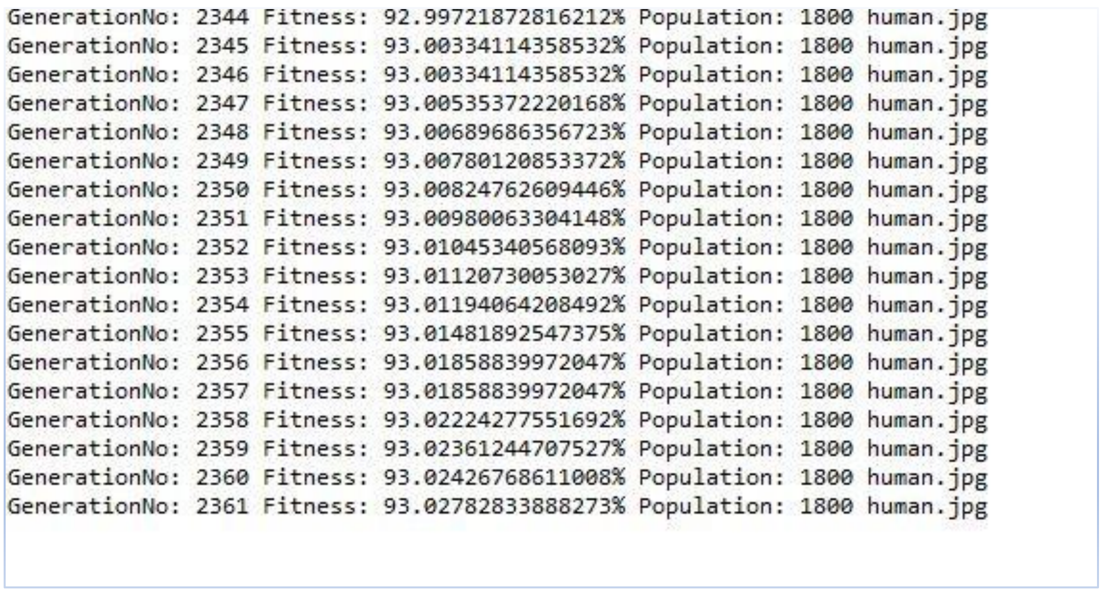
```
// split between threads
int forOneThread = poolOfImages.size() / THREADS;
for (int i = 0; i < THREADS; i++) {
    FitnessInParallel evaluator = parallelFitnessEvaluators.get(i);
    evaluator.setStart(i * forOneThread);
    evaluator.setEnd((i + 1) * forOneThread + 1);
    evaluator.setImageLists(poolOfImages);
    if (i + 1 == THREADS) {
        evaluator.setEnd(poolOfImages.size());
    }
    new Thread(evaluator).start();
}

// see if all done
while (!parallelFitnessEvaluatorDone()) {
    try {
        Thread.sleep(50);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

FitnessInParallel.java



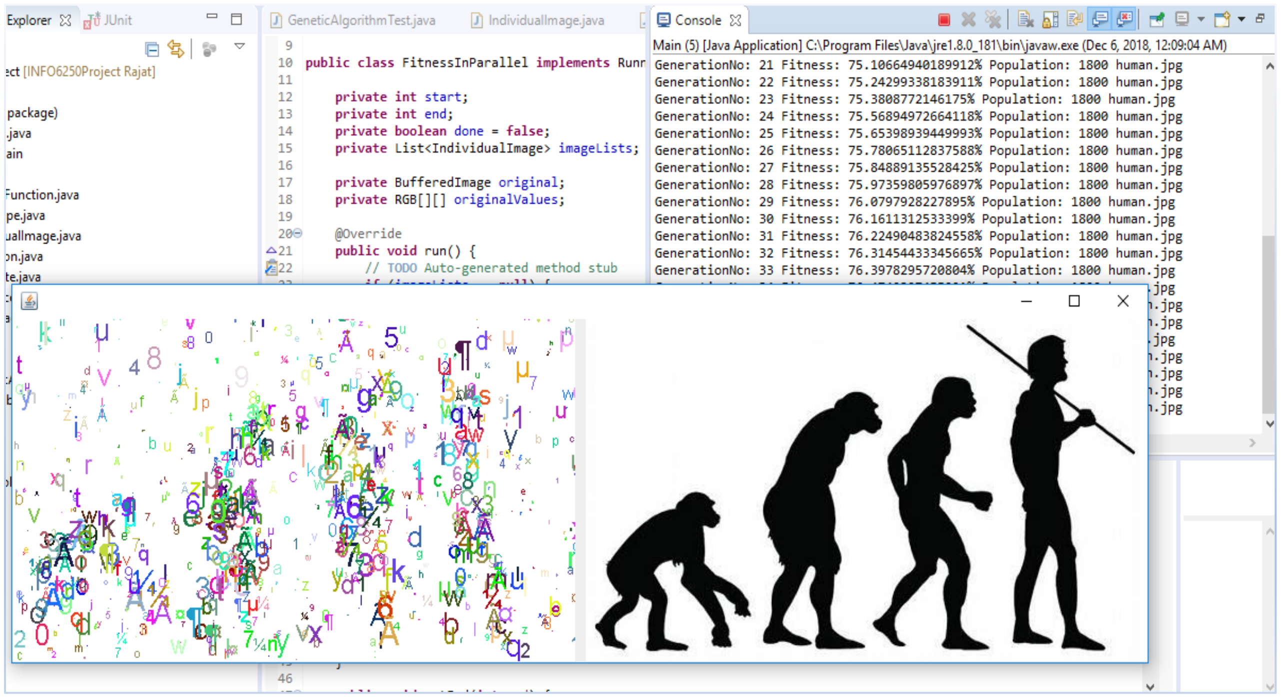
```
private int start;
private int end;
private boolean done = false;
private List<IndividualImage> imageLists;

private BufferedImage original;
private RGB[][] originalValues;

@Override
public void run() {
    // TODO Auto-generated method stub
    if (imageLists == null) {
        return;
    }

    for (int i = start; i < end; i++) {
        if (imageLists.get(i).getFitness() == null) {
            FitnessFunction.evaluateFitness(imageLists.get(i), original, originalValues);
        }
    }
    done = true;
}
```

The Result



```
GenerationNo: 2344 Fitness: 92.99721872816212% Population: 1800 human.jpg
GenerationNo: 2345 Fitness: 93.00334114358532% Population: 1800 human.jpg
GenerationNo: 2346 Fitness: 93.00334114358532% Population: 1800 human.jpg
GenerationNo: 2347 Fitness: 93.00535372220168% Population: 1800 human.jpg
GenerationNo: 2348 Fitness: 93.00689686356723% Population: 1800 human.jpg
GenerationNo: 2349 Fitness: 93.00780120853372% Population: 1800 human.jpg
GenerationNo: 2350 Fitness: 93.00824762609446% Population: 1800 human.jpg
GenerationNo: 2351 Fitness: 93.00980063304148% Population: 1800 human.jpg
GenerationNo: 2352 Fitness: 93.01045340568093% Population: 1800 human.jpg
GenerationNo: 2353 Fitness: 93.01120730053027% Population: 1800 human.jpg
GenerationNo: 2354 Fitness: 93.01194064208492% Population: 1800 human.jpg
GenerationNo: 2355 Fitness: 93.01481892547375% Population: 1800 human.jpg
GenerationNo: 2356 Fitness: 93.01858839972047% Population: 1800 human.jpg
GenerationNo: 2357 Fitness: 93.01858839972047% Population: 1800 human.jpg
GenerationNo: 2358 Fitness: 93.02224277551692% Population: 1800 human.jpg
GenerationNo: 2359 Fitness: 93.02361244707527% Population: 1800 human.jpg
GenerationNo: 2360 Fitness: 93.02426768611008% Population: 1800 human.jpg
GenerationNo: 2361 Fitness: 93.02782833888273% Population: 1800 human.jpg
```

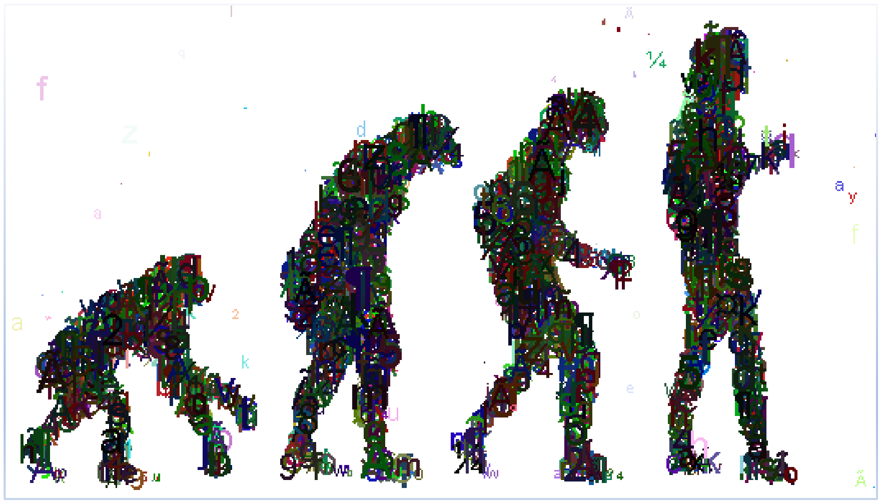
<u>Output in Process</u>



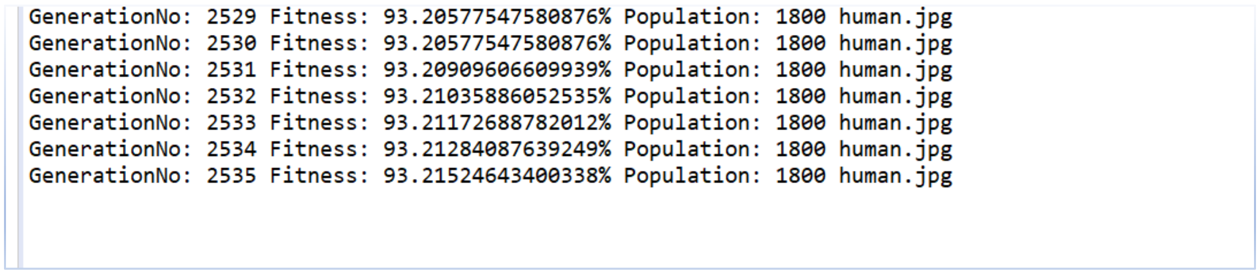
```
public class FitnessInParallel implements Runr

    private int start;
    private int end;
    private boolean done = false;
    private List<IndividualImage> imageLists;

    private BufferedImage original;
    private RGB[][] originalValues;

    @Override
    public void run() {
        // TODO Auto-generated method stub
```

```
Main (5) [Java Application] C:\Program Files\Java\jre1.8.0_181\bin\javaw.exe (Dec 6, 2018, 12:09:04 AM)
GenerationNo: 21 Fitness: 75.10664940189912% Population: 1800 human.jpg
GenerationNo: 22 Fitness: 75.24299338183911% Population: 1800 human.jpg
GenerationNo: 23 Fitness: 75.3808772146175% Population: 1800 human.jpg
GenerationNo: 24 Fitness: 75.56894972664118% Population: 1800 human.jpg
GenerationNo: 25 Fitness: 75.65398939449993% Population: 1800 human.jpg
GenerationNo: 26 Fitness: 75.78065112837588% Population: 1800 human.jpg
GenerationNo: 27 Fitness: 75.84889135528425% Population: 1800 human.jpg
GenerationNo: 28 Fitness: 75.97359805976897% Population: 1800 human.jpg
GenerationNo: 29 Fitness: 76.0797928227895% Population: 1800 human.jpg
GenerationNo: 30 Fitness: 76.1611312533399% Population: 1800 human.jpg
GenerationNo: 31 Fitness: 76.22490483824558% Population: 1800 human.jpg
GenerationNo: 32 Fitness: 76.31454433345665% Population: 1800 human.jpg
GenerationNo: 33 Fitness: 76.3978295720804% Population: 1800 human.jpg
```

Final Output

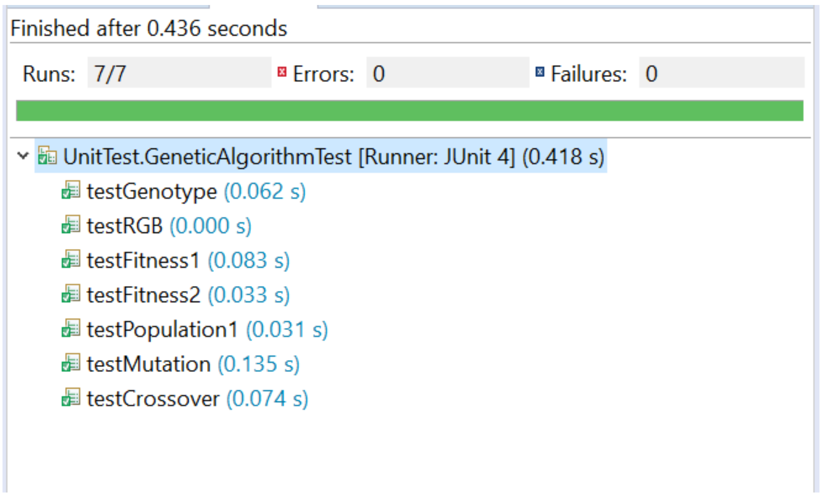


The output generated above is when similarity of parent image and produced image is greater than 0.93215. This image was formed after 2535 Generations.



```
GenerationNo: 2529 Fitness: 93.20577547580876% Population: 1800 human.jpg
GenerationNo: 2530 Fitness: 93.20577547580876% Population: 1800 human.jpg
GenerationNo: 2531 Fitness: 93.20909606609939% Population: 1800 human.jpg
GenerationNo: 2532 Fitness: 93.21035886052535% Population: 1800 human.jpg
GenerationNo: 2533 Fitness: 93.21172688782012% Population: 1800 human.jpg
GenerationNo: 2534 Fitness: 93.21284087639249% Population: 1800 human.jpg
GenerationNo: 2535 Fitness: 93.21524643400338% Population: 1800 human.jpg
```

TEST CASES

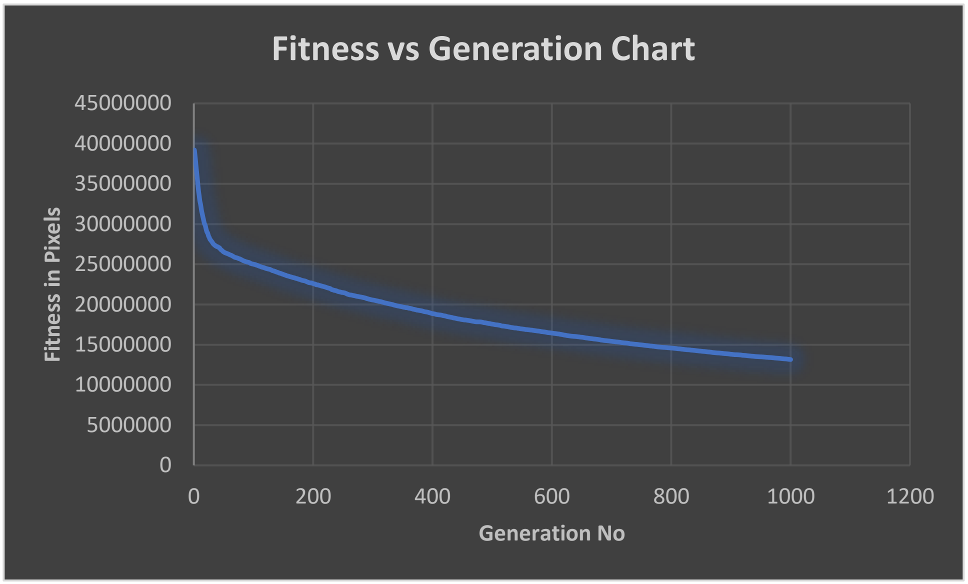


Finished after 0.436 seconds

Runs: 7/7 Errors: 0 Failures: 0

- UnitTest.GeneticAlgorithmTest [Runner: JUnit 4] (0.418 s)
  - testGenotype (0.062 s)
  - testRGB (0.000 s)
  - testFitness1 (0.083 s)
  - testFitness2 (0.033 s)
  - testPopulation1 (0.031 s)
  - testMutation (0.135 s)
  - testCrossover (0.074 s)

GRAPH



References

- https://en.wikipedia.org/wiki/List_of_genetic_algorithm_applications

- IEEE Digital Explore, Genetic Algorithms: A tool for image segmentation by Alaa Sheta ; Malik S. Braik ; Sultan Aljahdali.

- A Review of Genetic Algorithm application for Image Segmentation by Raj Kumar Mohanta.

- Image processing optimization by genetic algorithm with a new coding scheme by D. Snyers.