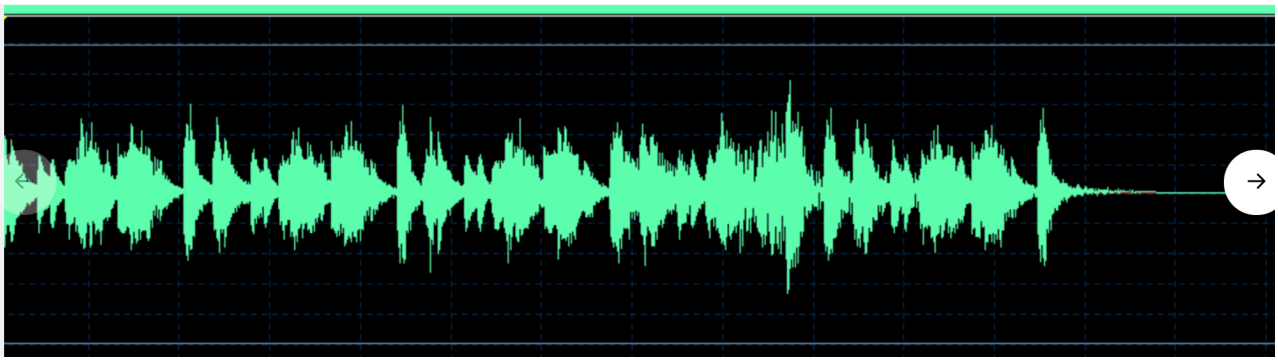
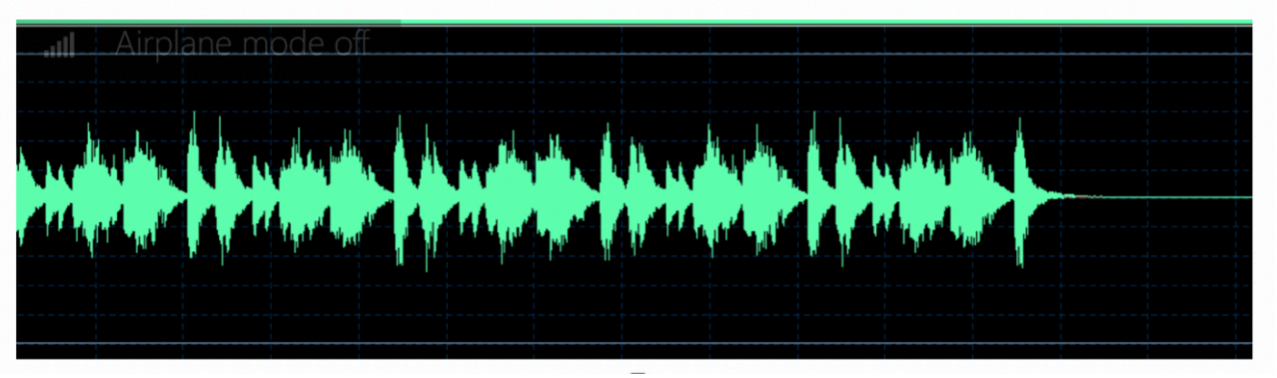
Generation of Music Composition
TEAM INFO6205_513

Vishnu Bhargavi Sabbisetty – 001497937
Aman Gupta – 001448126

OriginalSounds/Audio_original.mid



OriginalSounds/Audio_final.mid



Introduction

Generating music has always been considered a work of art with great respect and value. The music, which is present seems to be complicated, but it has always been divided into a simple form. The smallest unit is known as a rhythm. So, we can make an analogy that sentences are generated from alphabets; similarly, music is also a mix and match of units of rhythm where it has its standards to be followed. It needs a lot of effort and knowledge to make a small part of the music. And the music which is composed is again measured against a few protocols and rules to tell whether it is academically acceptable and artful.

From the other side of the coin even though music is generated by following all the rules and criteria, it is very much challenging to regenerate using the computer as it does not understand all these rules and regulations. It only understands 1,0. So, we have taken this as a challenge to regenerate the music similar to the final goal which is given as input for reference from random data. And we have used Genetic Algorithms to process the data and compare with goal continuously till the music similar to the target is generated.

What and Why Genetic Algorithm...?

These algorithms are used when we want to generate desired solution which gets created after processing for n times. The intermediate solutions always come together to generate new solution which is better than previous solutions.

We use genetic algorithms to solve problems which have np-complete solutions which take complex time. So, by using genetic algorithm it helps to produce a better solution compared to existing solution.

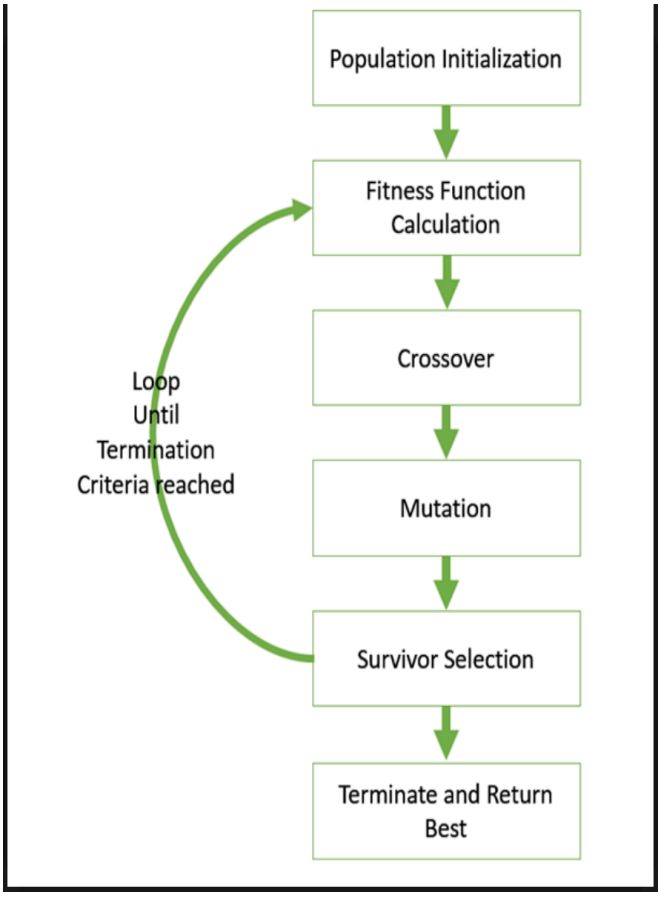
Keywords in GA:

- Individual: It is the final solution of the problem
- Gene: Smallest unit of Individual. Combination of genes form an individual. If at least one gene different between two individuals then they are said to be unique.
- I1 != I2 only when there is atleast one gene say g1 different in two Individuals
- Phenotype: Set of genes coming together to represent a characteristic for an individual.
- Genotype: The smallest part of Individual which cannot be future divided is called gene and the traits of gene represent genotype.

	CD	Cd	cD	cd
CD	CCDD (seal)	CCDd (seal)	CcDD (seal)	CcDd (seal)
Cd	CCdD (seal)	CCdd (blue)	CcdD (seal)	Ccdd (blue)
cD	cCDD (seal)	cCDd (seal)	ccDD (chocolate)	ccDd (chocolate)
cd	cCdD (seal)	cCdd (blue)	ccdD (chocolate)	ccdd (lilac)

Here cd, cD, Cd, CD are the genes which as a combination form CCDD, CCDb….ccdd which result blue, choclate, seal which are known as phenotypes.

Steps in Genetic Algorithm:



For any problem the GA is classified into 6 steps. The steps remain same but the fitness function changes based on the problem.

1. Population Initialization:
- First define gene.
- Then design individual and generate individuals of fixed size which is called as population generation.
2. Fitness Function
- This function helps you to decide how much similar is a particular individual to the goal individual
- This is keeps on changing based on problem
3. Selection Function
- Helps to decide best individuals from the pool of a particular generation and helps to generate the next generation
- We need to be careful during this process. If we discard many individuals then we narrow down the type of individuals. If we keep many then there will be less new generation. So, we must be careful during the selection process.
4. Crossover
- This function helps you to generate new genes by taking to best individuals.

5. Mutation

- This is the process where we randomly consider a gene in the individual and change

Note: Cross-over and Mutation help to create randomness and generate new generation. This process continues till the new generation having best from old generation and new individuals form the pool size of old generation.

6. The new generation again go through the step 2 to 5 till there is individual exact as the goal

There are three terminating conditions for any genetic algorithms:

- When the best individual similar to original is generated
- When maximum number of generations is finished
- When there is no change with the generated generations for some time.

Problem Statement

To generate music similar to the original music. In our music is composed of bols where each bol is having some set of properties. So, the combination of bols form music. So, different combinations form different kinds of music. We tried to generate back music which is similar to original input by considering some parameters.

Initial Population:

```
poolOfSounds = Populate.initPool(POOL_SIZE, originalSound.size(), MAX_KEY, MAX_VELOCITY, MAX_TICK);
```

This helps to generate pool of individuals with random genes.

```
public class Populate {
    public static ArrayList<Individual> initPool(int poolSize, int popSize, int maxKey, int maxVelocity, long maxTick){
        ArrayList<Individual> soundPopulation = new ArrayList<>();
        for(int i = 0; i < poolSize; i++){
            ArrayList<Genotype> genes = new ArrayList<>();
            for(int j = 0; j < popSize; j++){
                genes.add(Mutation.getRandomGenes(maxKey, maxVelocity));
            }
            soundPopulation.add(new Individual(genes));
        }
        return soundPopulation;
    }
}
```

Generate Individual out of input music:

```
originalSound = midi.parseMidi( filename: "audio.mid");
```

```
tick: 0 Note on,  key=45 velocity: 70
tick: 1 Note off,  key=45 velocity: 64
tick: 1 Note on,  key=47 velocity: 76
tick: 2 Note off,  key=47 velocity: 64
tick: 2 Note on,  key=49 velocity: 74
tick: 3 Note off,  key=49 velocity: 64
tick: 3 Note on,  key=52 velocity: 67
tick: 4 Note off,  key=52 velocity: 64
tick: 4 Note on,  key=55 velocity: 61
tick: 4 Note on,  key=62 velocity: 44
```

This is how genes in the individual will be.

Genotype and it's Fitness:

```
private boolean note;
private int key;
private int velocity;
private Long gene_fitness;
```

Our gene is having following properties. Where each property has its own range of values:

Note: on and off

Key: 0 to 127

Velocity: 0 to 127

As, each gene is having following properties we are computing the fitness of each gene by comparing its characters with corresponding gene from goal. So, we have following fitness function to compare the characteristic

```
public static long computeFitnessGene(Genotype parent,Genotype child){
    long fitness = 0, diff_key=0,diff_vel=0;
    if(parent.isNote()==child.isNote()){
        fitness++;
    }
    diff_key=Math.abs(parent.getKey()-child.getKey());
    if(diff_key==0){
        fitness++;
    }
    diff_vel=Math.abs(parent.getVelocity()-child.getVelocity());
    if(diff_vel==0){
        fitness++;
    }
    if(fitness==3){
        fitness++;
    }
    if(diff_vel==0 && diff_key==0){
        fitness++;
    }
    if(diff_vel==0 && parent.isNote()==child.isNote()){
        fitness++;
    }
    if(diff_key==0 && parent.isNote()==child.isNote()){
        fitness++;
    }

    return fitness;

}
```

So, we have considered 7 conditions to calculate fitness of a particular gene.

- When only velocity/note/key is matched then fitness will be only 1.
- When a combination of two properties match either velocity and key/ key and note / velocity and note then the resultant fitness is 3.
- When all 3 attributes are same then the fitness will be 7 in which it is the best gene in the individual.

Fitness of Individual:

So, after calculating the fitness of all genes present in individual they are added to form the fitness of the individual.

```
public class Fitness {

    public static void computeFitnessIndividual(Individual goal, Individual random){
        long total=0;
        for(int i=0;i<goal.getIndividual().size();i++){
            Genotype parent = goal.getGene(i);
            Genotype child = random.getGene(i);
            long fitness = computeFitnessGene(parent,child);
            child.setGene_fitness(fitness);
            total+=fitness;
        }
        random.setFitness(total);
    }
```

So, each individual is having its own fitness.

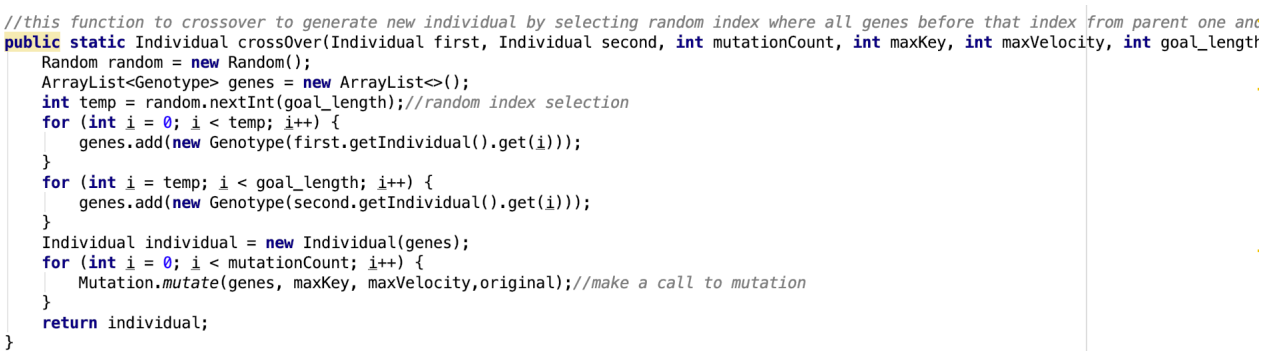
Selection Operation:

```
public static ArrayList<Individual> mateBest(ArrayList<Individual> pool, int best, int goal_length, int mutationCount, int maxKey, i
    ArrayList<Individual> newPool = new ArrayList<>();
    for (int i = 0; i < pool.size(); i++) {
        if (i < best) {
            newPool.add(pool.get(i));//Selection process
        } else {
            newPool.add(crossOver(newPool, best, maxKey, maxVelocity, mutationCount,goal_length,original));//cross-over for new gene
        }
    }
    return newPool;
```

So, the best individuals based on count are inserted into new generation.

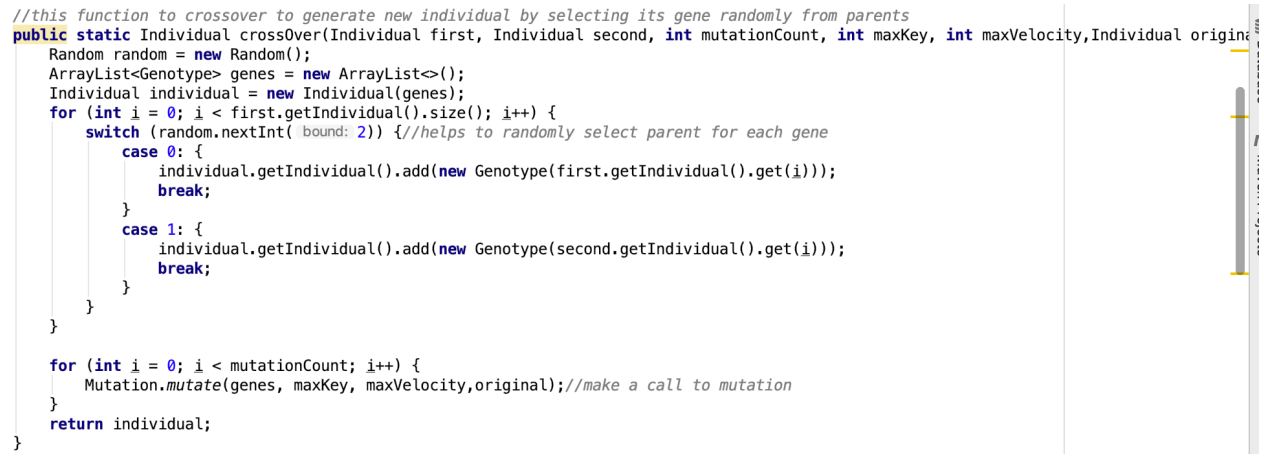
Cross-over: Best two individuals are selected and used to generate new child.

1. Based on random index. All genes before that particular index are taken from parent 1 and all genes after that index are taken from parent 2.



```java
//this function to crossover to generate new individual by selecting random index where all genes before that index from parent one and
public static Individual crossOver(Individual first, Individual second, int mutationCount, int maxKey, int maxVelocity, int goal_length
    Random random = new Random();
    ArrayList<Genotype> genes = new ArrayList<>();
    int temp = random.nextInt(goal_length);//random index selection
    for (int i = 0; i < temp; i++) {
        genes.add(new Genotype(first.getIndividual().get(i)));
    }
    for (int i = temp; i < goal_length; i++) {
        genes.add(new Genotype(second.getIndividual().get(i)));
    }
    Individual individual = new Individual(genes);
    for (int i = 0; i < mutationCount; i++) {
        Mutation.mutate(genes, maxKey, maxVelocity,original);//make a call to mutation
    }
    return individual;
}
```

2. Randomly new child is generated where is gene of individual is selected from parents randomly



```java
//this function to crossover to generate new individual by selecting its gene randomly from parents
public static Individual crossOver(Individual first, Individual second, int mutationCount, int maxKey, int maxVelocity,Individual origina
    Random random = new Random();
    ArrayList<Genotype> genes = new ArrayList<>();
    Individual individual = new Individual(genes);
    for (int i = 0; i < first.getIndividual().size(); i++) {
        switch (random.nextInt( bound: 2)) {//helps to randomly select parent for each gene
            case 0: {
                individual.getIndividual().add(new Genotype(first.getIndividual().get(i)));
                break;
            }
            case 1: {
                individual.getIndividual().add(new Genotype(second.getIndividual().get(i)));
                break;
            }
        }
    }

    for (int i = 0; i < mutationCount; i++) {
        Mutation.mutate(genes, maxKey, maxVelocity,original);//make a call to mutation
    }
    return individual;
}
```

Then a call is made to mutation.

Mutation:

Here a random cell of the new child is changed. And mutation is done based on count of mutation defined in program.

In our program mutation function is also randomly selected. We have 6 functionalities. This just helps to create some randomness.

```
switch (random.nextInt( bound: 6)) {//helps to randomly select mutation function
    case 0:
        // remove one Gene and add new Gene
        gene = original.getGene(position);
        geneList.remove(position);
        geneList.add(gene);
        break;
    case 1:
        // swap two genes from the list
        Collections.swap(geneList, random.nextInt(geneList.size()), position);
        break;
    case 2:
        // change any one's velocity
        gene.setVelocity(random.nextInt(maxVelocity)+1);
        geneList.set(position, gene);
        break;
    case 3:
        // change any one's key
        gene.setKey(random.nextInt(maxKey)+1);
        geneList.set(position, gene);
        break;
    case 4:
        // change any one's note
        gene.setNote(!gene.isNote());
        geneList.set(position, gene);
        break;
    case 5:
        // replace one with new gene
        gene = Populate.getRandomGenes(maxKey,maxVelocity);
        geneList.set(position, gene);
        break;
}
```

Maximum Fitness:

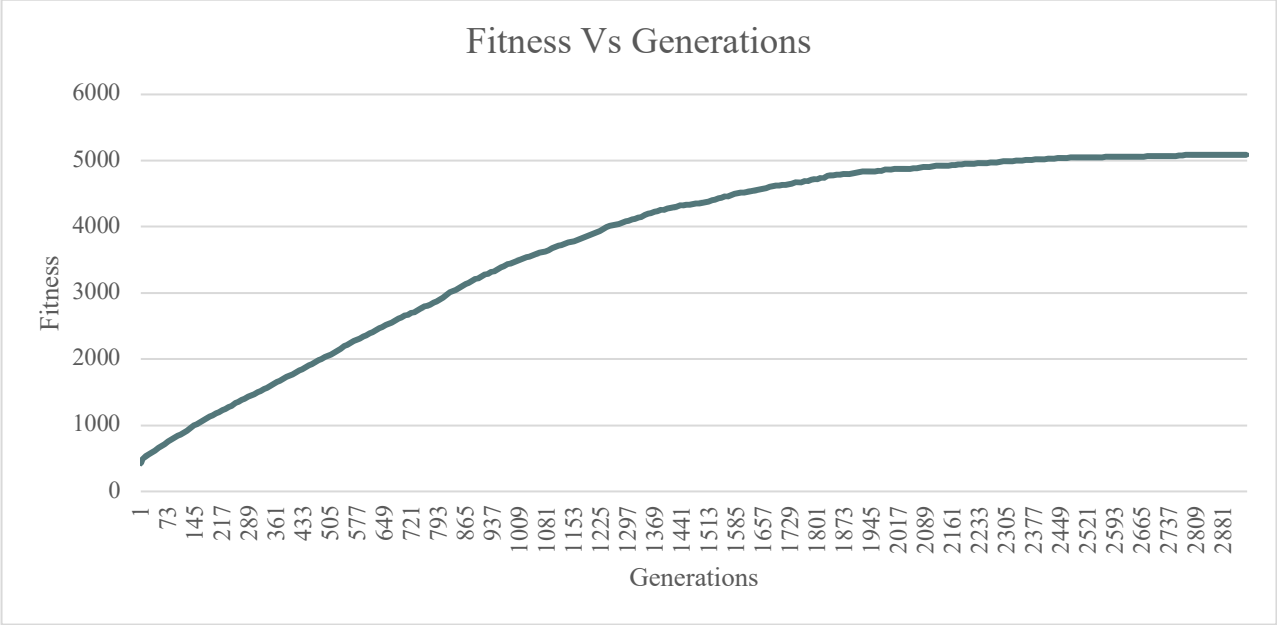
Each gene is having maximum fitness of 7 based on fitness function.

So, maximum fitness is 7*length of input music.

We are restricting to 0.99 of max fitness for the final output.

Which is like a break point.

Graph between Fitness Vs Generation



Observations: Fitness keeps on increasing as the generations pass. The maximum fitness is achieves at 2881 generation where maximum fitness is around 5095 for input length 735

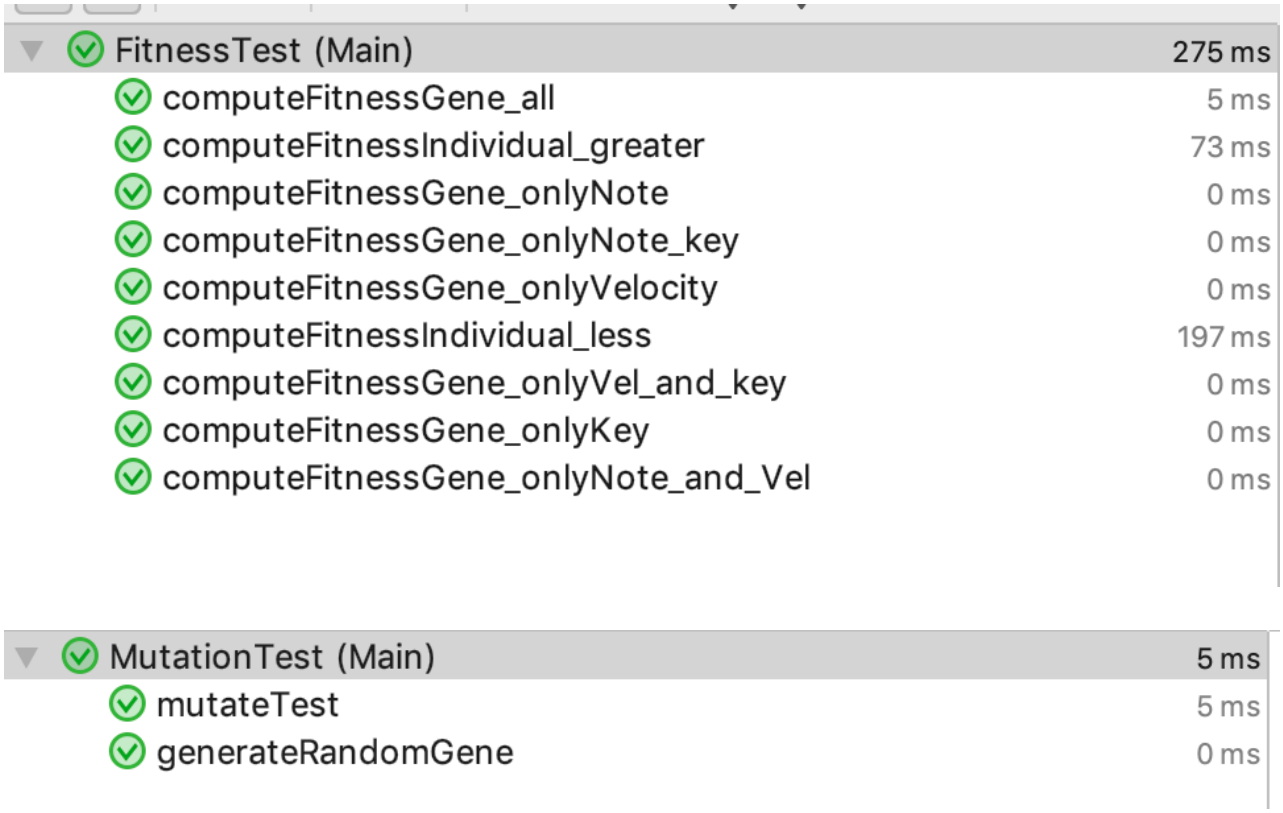
Fitness achieved = 5095 and actual maximum fitness is 7*735= 5145

→5095~5145(Achieved~Maximum)

```
tick: 43 Note off,  key=82 velocity: 64
Other message: class javax.sound.midi.Track$ImmutableEndOfTrack javax.sound.midi.Track$ImmutableEndOfTrack@706a04ae

735
Generation: 1 has best individual with Fitness: 434
Generation: 2 has best individual with Fitness: 449
Generation: 3 has best individual with Fitness: 464
Generation: 4 has best individual with Fitness: 472
Generation: 5 has best individual with Fitness: 477
Generation: 6 has best individual with Fitness: 484
Generation: 7 has best individual with Fitness: 488
```

```
Generation: 2937 has best individual with Fitness: 5091
Generation: 2938 has best individual with Fitness: 5091
Generation: 2939 has best individual with Fitness: 5091
Generation: 2940 has best individual with Fitness: 5091
Generation: 2941 has best individual with Fitness: 5091
Generation: 2942 has best individual with Fitness: 5091
Generation: 2943 has best individual with Fitness: 5091
Generation: 2944 has best individual with Fitness: 5091
Generation: 2945 has best individual with Fitness: 5091
Generation: 2946 has best individual with Fitness: 5091
Generation: 2947 has best individual with Fitness: 5091
Generation: 2948 has best individual with Fitness: 5091
Generation: 2949 has best individual with Fitness: 5091
Generation: 2950 has best individual with Fitness: 5091
Generation: 2951 has best individual with Fitness: 5091
Generation: 2952 has best individual with Fitness: 5095
with parallel processing 103373ms
```

Testcases for fitness of individual, gene, mutation and crossover



Implementation of Parallel Processing:

Here we are taking 4 threads. And processing them parallel. Time taken is 103373ms

```
Generation: 2952 has best individual with Fitness: 5095
with parallel processing 103373ms
```

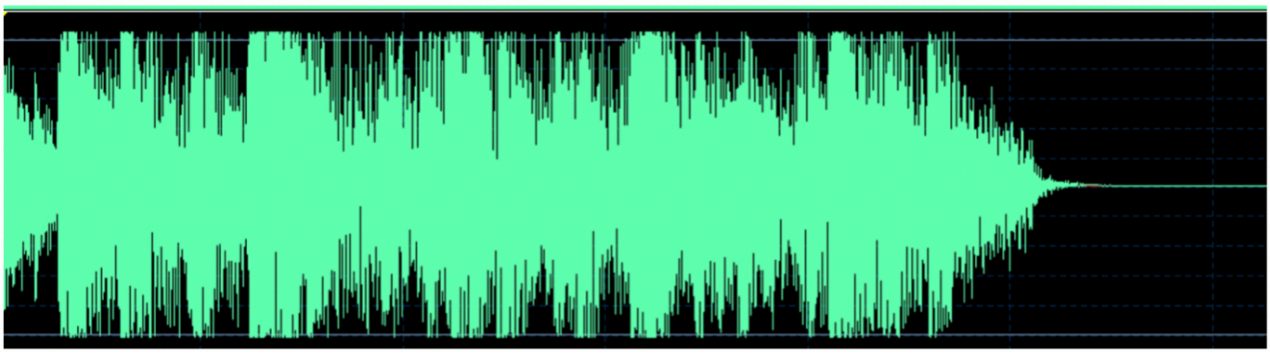
```
private static void fitnessOfMultipleMidis(Individual original,List<Individual> poolOfSounds ) {
    // split between threads
    int forOneThread = poolOfSounds.size() /THREADS;
    for (int i = 0; i < THREADS; i++) {
        FitnessInParallel evaluator = parallelFitnessEvaluators.get(i);
        evaluator.setStart(i * forOneThread);
        evaluator.setEnd((i + 1) * forOneThread + 1);
        evaluator.setPoolOfSounds(poolOfSounds);
        evaluator.setOriginal(original);
        if (i + 1 == THREADS) {
            evaluator.setEnd(poolOfSounds.size());
        }
        new Thread(evaluator).start();
    }

    //see if all done
    while (!parallelFitnessEvaluatorDone()) {
        try {
            Thread.sleep( millis: 0);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

}
//check status of each evaluator if done or not
private static boolean parallelFitnessEvaluatorDone() {
    for (FitnessInParallel evaluator : parallelFitnessEvaluators) {
        if (!evaluator.isDone()) {
            return false;
        }
    }
    resetparallelFitnessEvaluator();
    return true;
}

//if done set evaluator to done state
private static void resetparallelFitnessEvaluator() {
    for (FitnessInParallel evaluator : parallelFitnessEvaluators) {
        evaluator.setDone(false);
    }
}
```

```
//help to start the processing for threads
private static void initThreads() {
    parallelFitnessEvaluators = new ArrayList<>();
    for (int i = 0; i < THREADS; i++) {
        FitnessInParallel evaluator = new FitnessInParallel();
        evaluator.setPoolOfSounds(poolOfSounds);
        evaluator.setOriginal(new Individual(originalSound));
        parallelFitnessEvaluators.add(evaluator);
    }
}
```

Outputs:

For 100 generation

OriginalSounds/generatedAudio100.mid

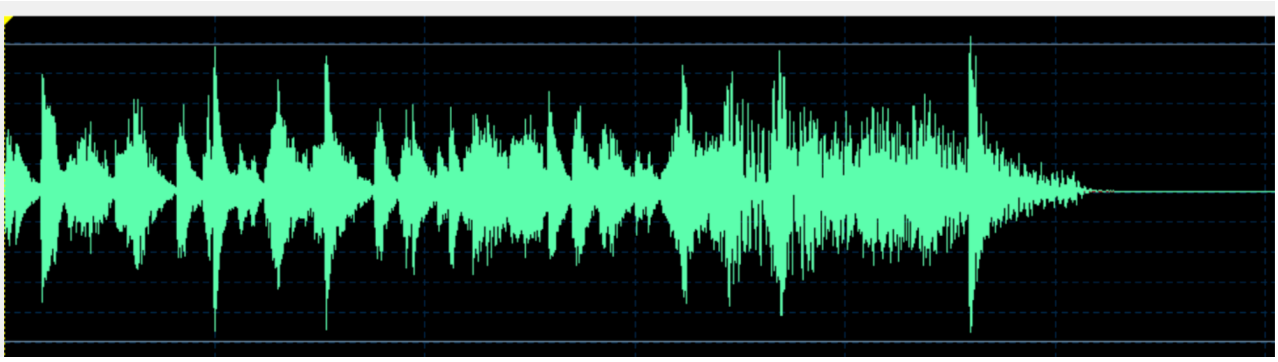


For Original sound

OriginalSounds/Audio_original.mid

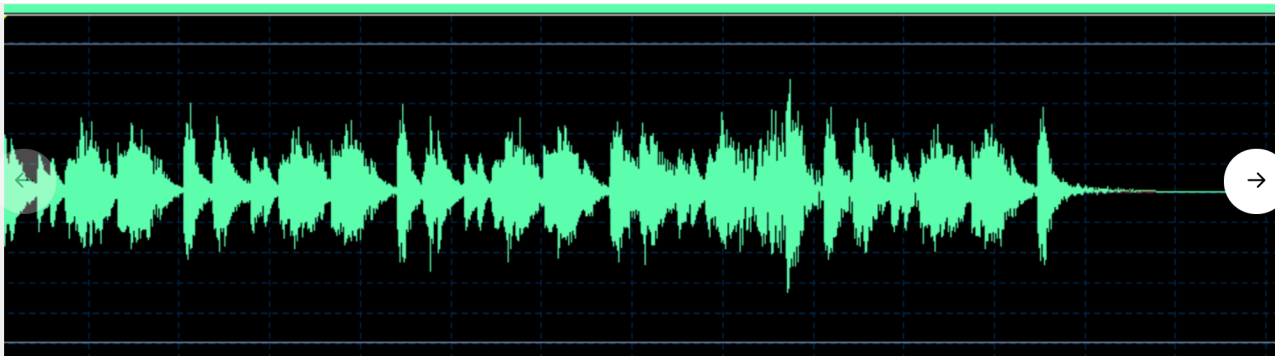
For 2000 Generation

OriginalSounds/generatedAudio2000.mid



For Final generation

OriginalSounds/Audio_final.mid



Bibilography:

https://link.springer.com/chapter/10.1007/978-3-319-76351-4_34