CS 451: Computational Intelligence

Assignment 01

Syed Hasan Faaz Abidi (sa06195), Syed Hammad Ali (sa04324), Hasan Naseem (hn05102)

13 February 2022

1. TSP Problem

As required by the assignment, we had to optimize the Traveling Sales Person problem on the Qatar dataset containing 194 cities using Evolutionary Algorithm.

For this particular problem, the chromosome representation was as follows,

1.1. Chromosome Representation

In the TSP dataset, there were 194 nodes representing each city with their position in euclidean distances. Keeping the constraints in consideration of a valid route that the route visits each city exactly once and returns to the origin city, the representation we chose was a simple **1D list**. Each element of the list represents a unique integer for each city.

For example if we have 3 cities, a possible candidate solution with population 3 looks like:

Population = [[1,2,3],[3,2,1],[2,3,1]]

1.2. Fitness Function

The fitness function for this problem was straightforward. The fitness score in TSP for a valid route/solution was the total distance between each city in that route. So, to calculate the total score, we first calculated the distance between each city using the given euclidean distances in the data. Summing up all those distances would give us a fitness score of a particular route.

TSP was different in a way that to optimize it, we need to minimize the fitness score. However, our selection functions were implemented in a way that they were biased towards the fittest chromosome. To counter this issue, we made our fitness values negative, so that selection would work the same but TSP would also get optimized.

Pseudocode for fitness function:

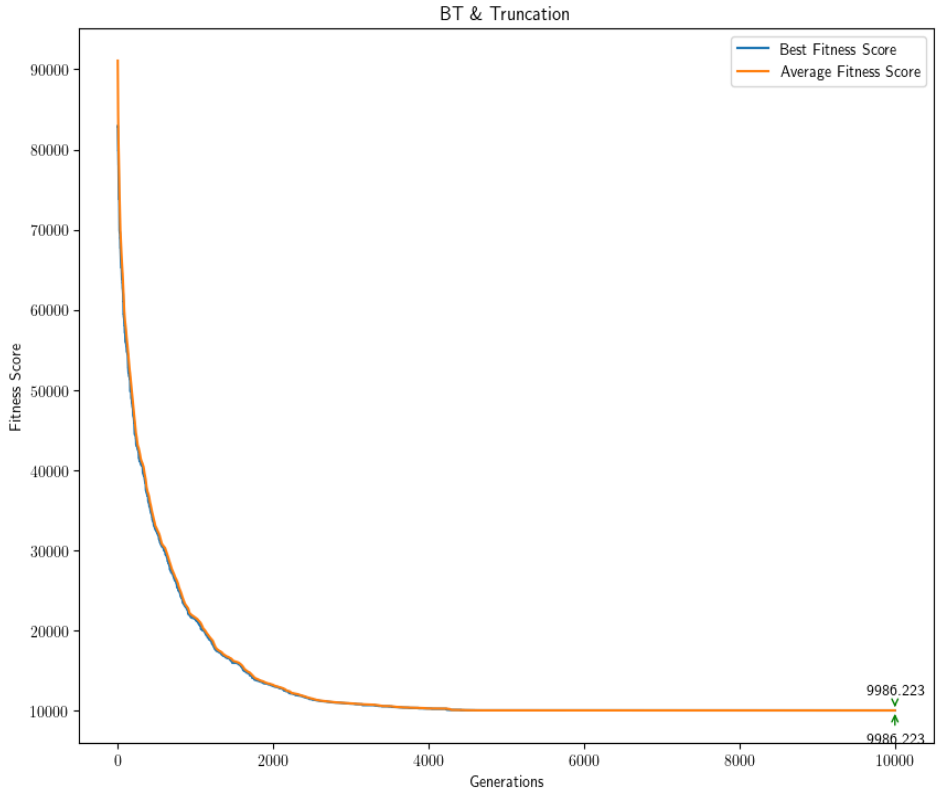
```
def __fitness_function(self, chromosome: list)-> int:
        total_distance = 0
        for i in range(len(chromosome)-1):
                total_distance += self.__distance(chromosome[i],
                chromosome[i+1])
        total_distance += self.__distance(chromosome[0], chromosome[-1])
        return total_distance*-1

def __distance(self, city1: int, city2: int)-> int:
        return ((self.data[city1][0] - self.data[city2][0])**2 + (self.data[city1][1] -
        self.data[city2][1])**2)**0.5
```

1.3. EA Evaluation

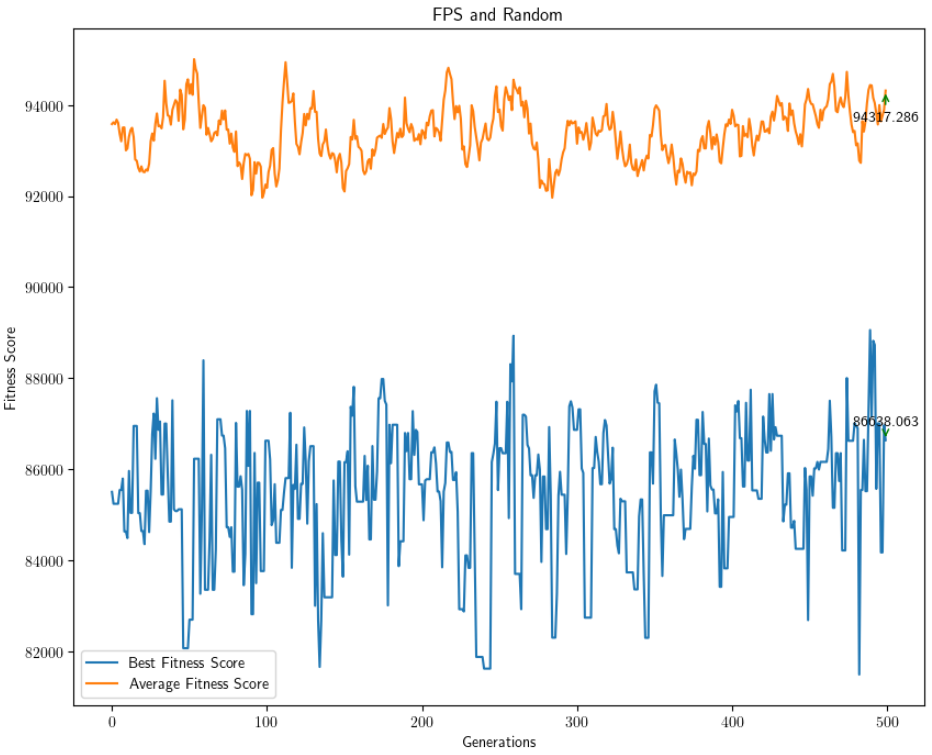
1.3.1 Binary Tournament & Truncation Plot

Best Fitness Score: **9986.223**
Average Fitness Score: **9986.223**



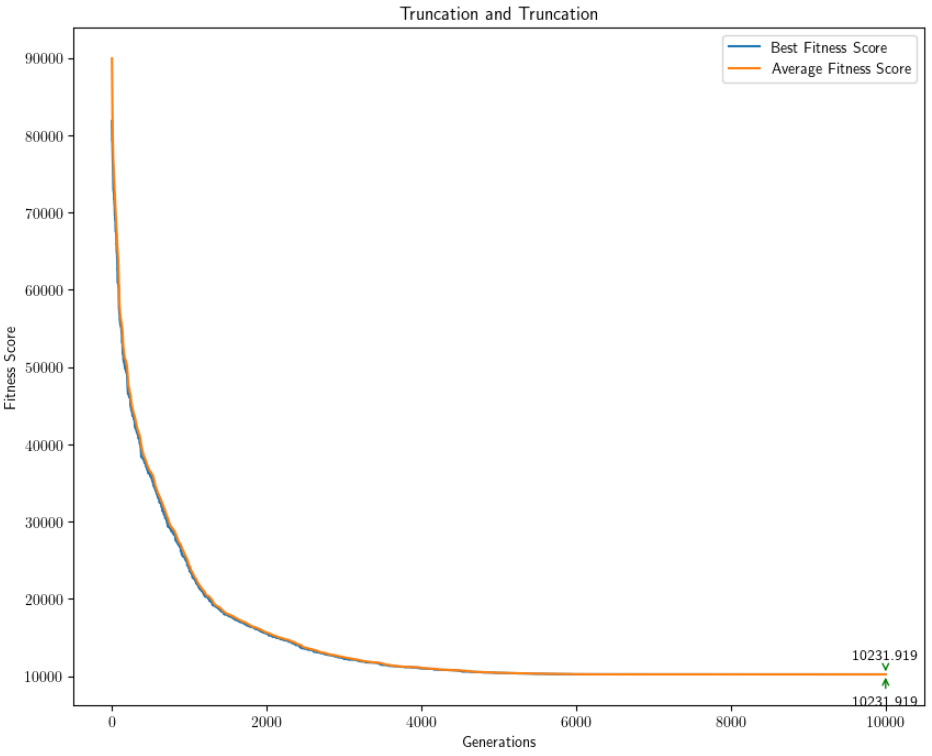
1.3.2 FPS & Random Plot

Best Fitness Score: **86638.063**
Average Fitness Score: **94317.286**



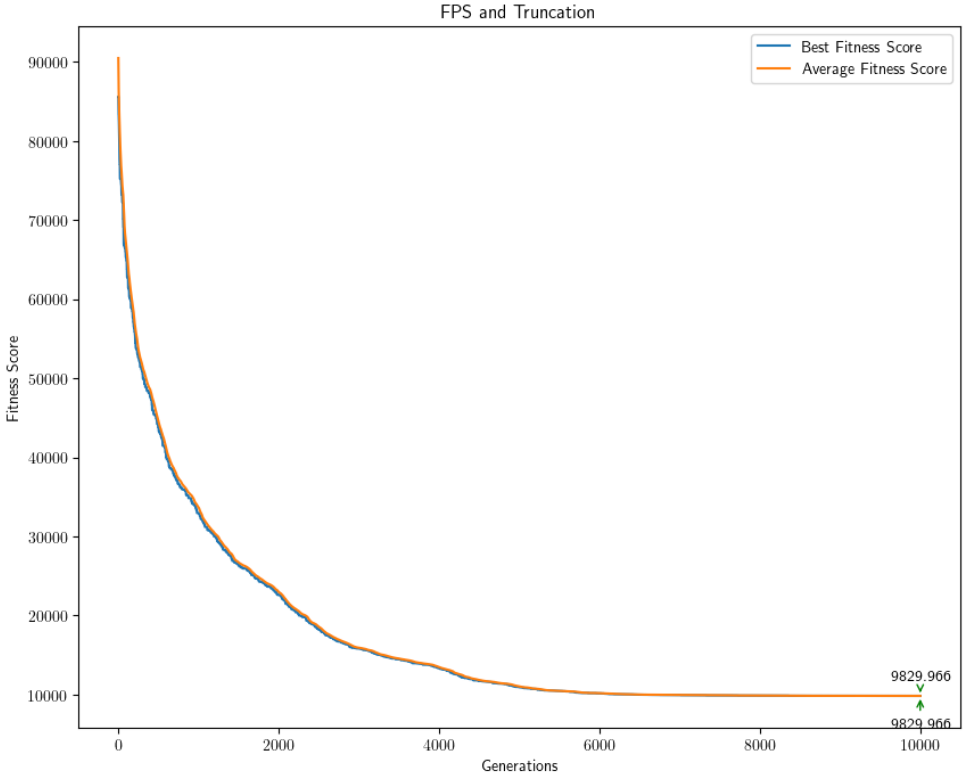
1.3.3 Truncation & Truncation Plot

Best Fitness Score: **10231.919**
Average Fitness Score: **10231.919**



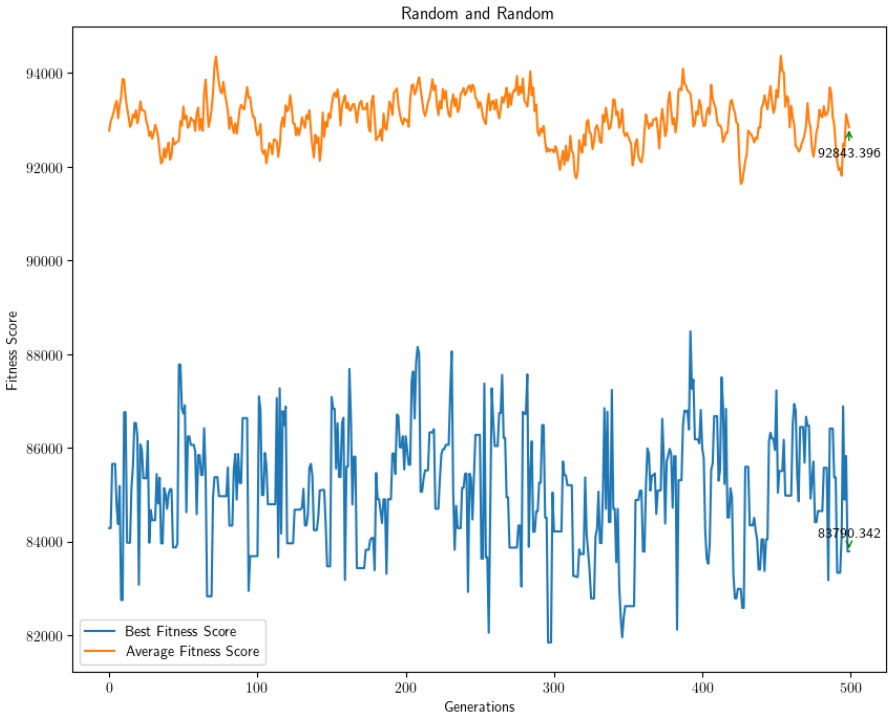
1.3.4 FPS & Truncation Plot

Best Fitness Score: **9829.966**
Average Fitness Score: **9829.966**



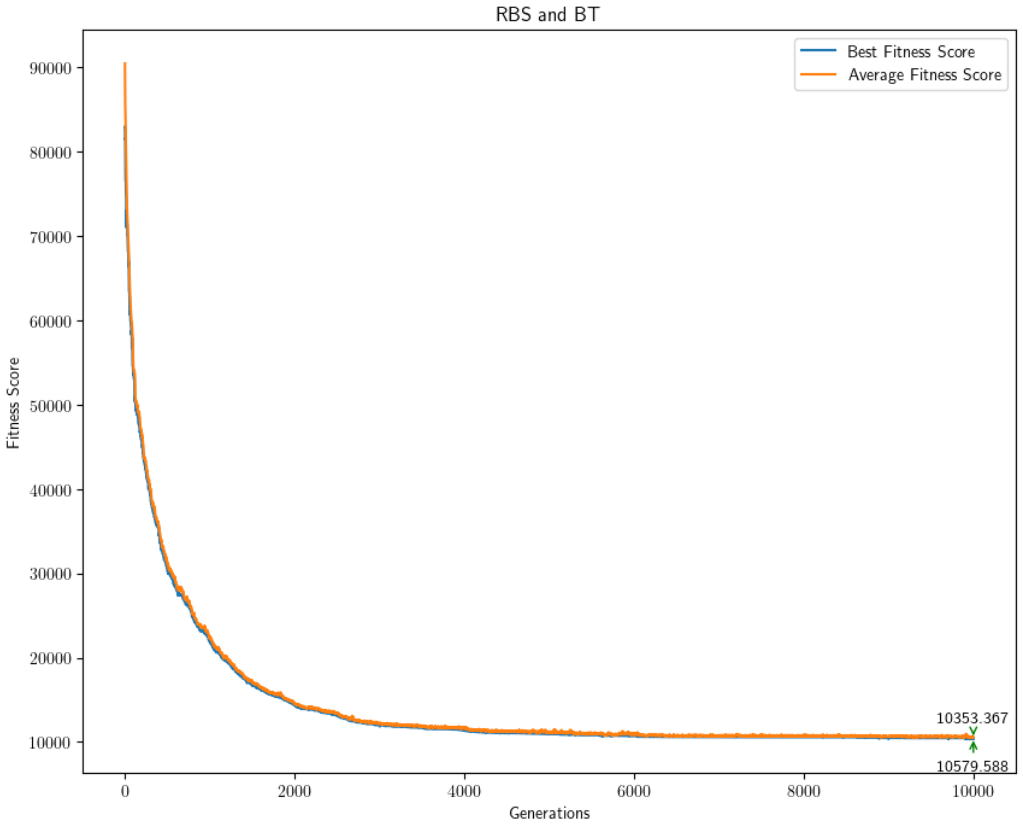
1.3.4 Random &Random Plot

Best Fitness Score: **83790.342**
Average Fitness Score: **92843.396**



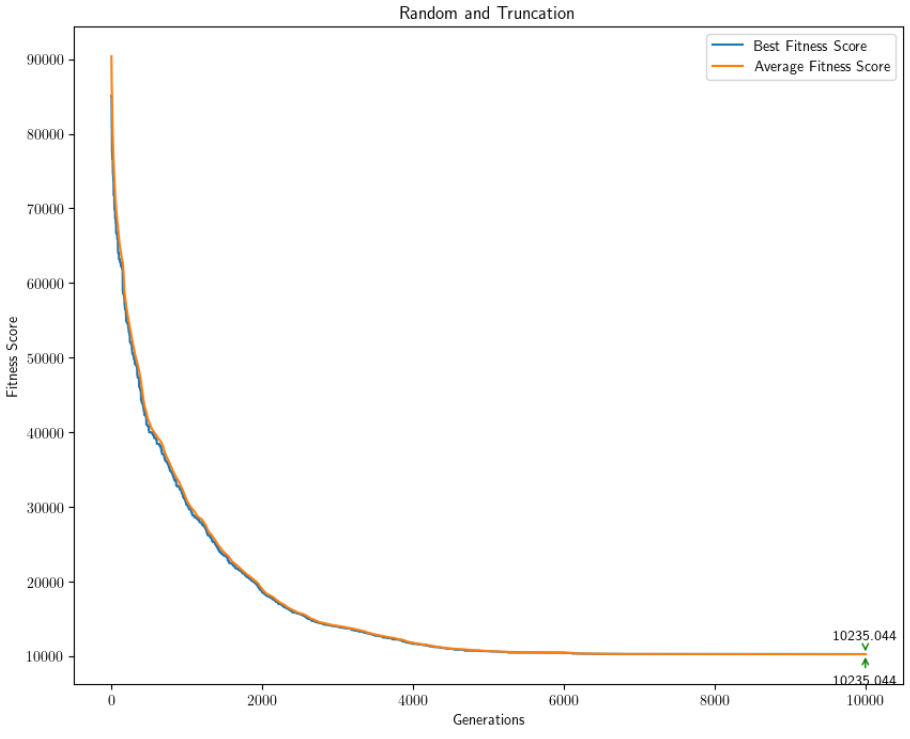
1.3.5 RBS & BT Plot

Best Fitness Score: **10353.367**
Average Fitness Score: **10579.588**



1.3.6 Random and Truncation Plot

Best Fitness Score: **10235.044**
Average Fitness Score: **10235.044**



1.4 Analysis

The scheme that worked best for this problem was FPS and Truncation which resulted in the best score of **9829**. The scheme consists of BT and Truncation, on the other hand, worked really well as well and was really close. I got a **9989** fitness score by using BT and Truncation. Other schemes didn't come below 10,000 and most of them were really close to 10,000 like RBS and BT and Truncation and Truncation. The scheme which did worst was random and random. This scheme was completely random and the results were around the same as the initial population which was expected. According to my finding right, crossover and mutation function plays a huge role in optimizing. Initially, I had a different 1 point crossover and for mutation, I was just swapping random at random index. But I changed these and I implemented a two-point crossover and used a different mutation function. The results after that were really good. In addition to it, I think, the mutation is really important. I got my best result when my mutation rate was 0.7. Mutation helps to keep optimizing and it reduces the chances of reaching a plateau early.

2. Graph Coloring Problem

The GC Problem required us to optimize the number of colors on a graph given that no two adjacent vertices have the same color. For our problem, the graph for which we had to perform the optimization for had 100 nodes with 2487 edges.

2.1. Chromosome representation

The chromosome for this problem is a 1D list where each gene is an RGB tuple such that R = G = B (for ease of generation). The index + 1 of that item in the list (chromosome) corresponds to the color of that gene.
So for example;
In the chromosome = [(110, 110, 110) , (100, 100, 100) , (67, 67, 67)], the node labeled '1' has the color (110, 110, 110).

2.2. Fitness Function

Since we want to attain the minimum number of colors for our graph then considering the chromosome representation, the fitness simply returns the number of unique colors in the

chromosome; essentially the number of unique genes. Why? Because the one with less unique colors will have a higher fitness. Similar to the case of TSP, the fitness function returns the negative value.

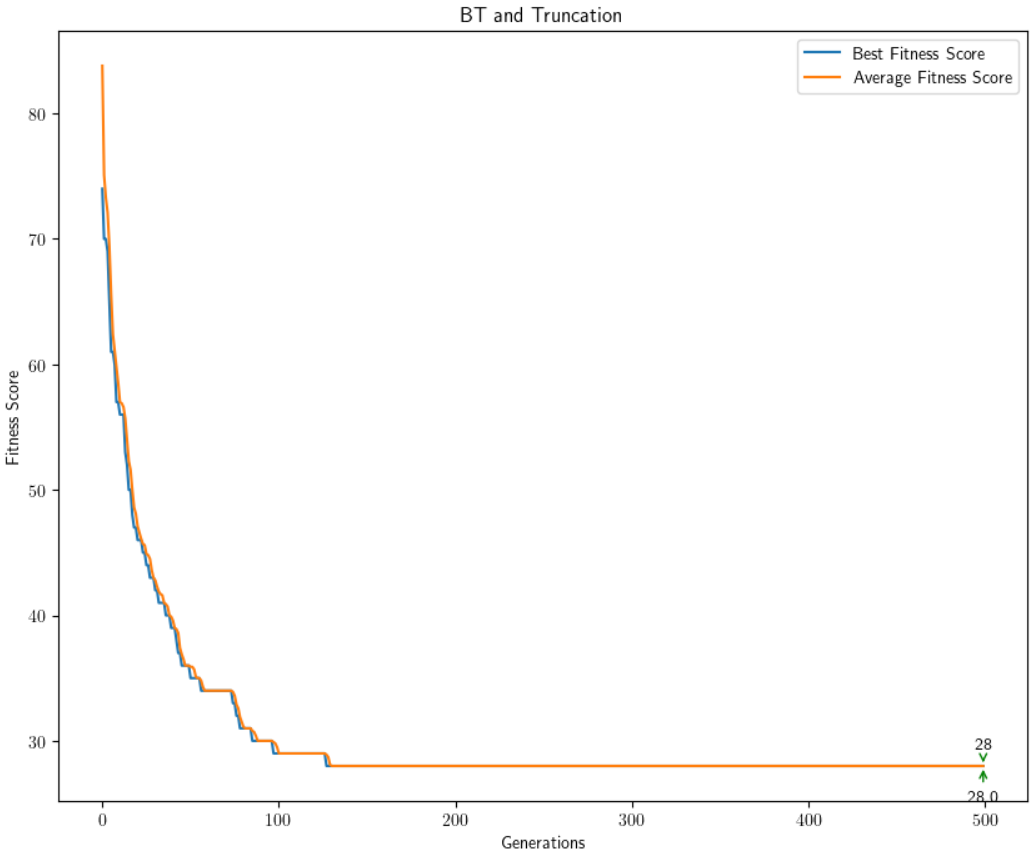
Pseudo code Fitness function

```
def fitness(chromosome):
    return length (unique elements of chromosome)
```

2.3 EA Evaluation

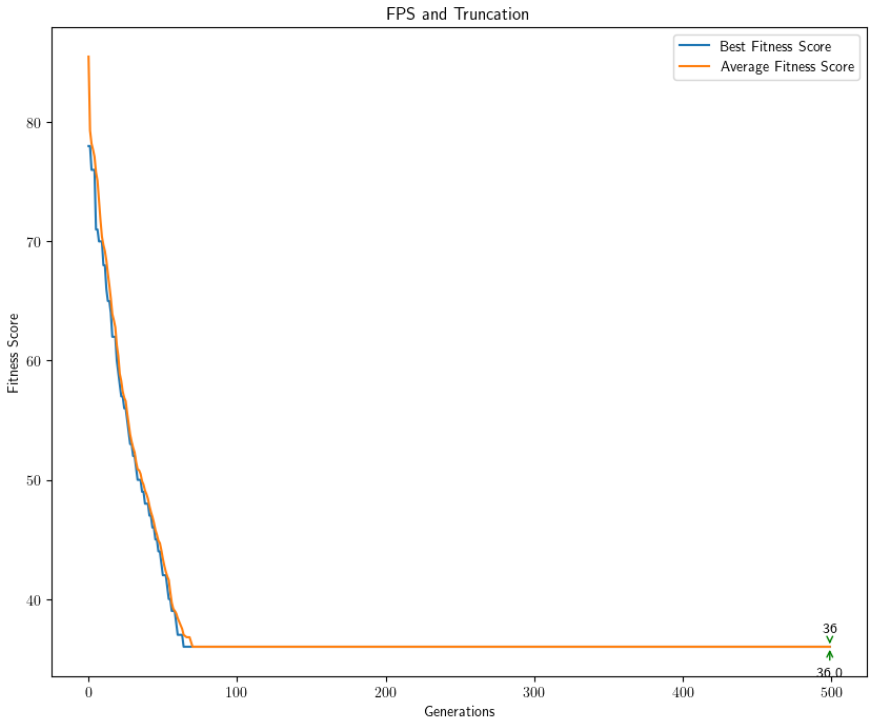
2.3.1 BT & Truncation Plot

Best Fitness Score: 28
Average Fitness Score: 28



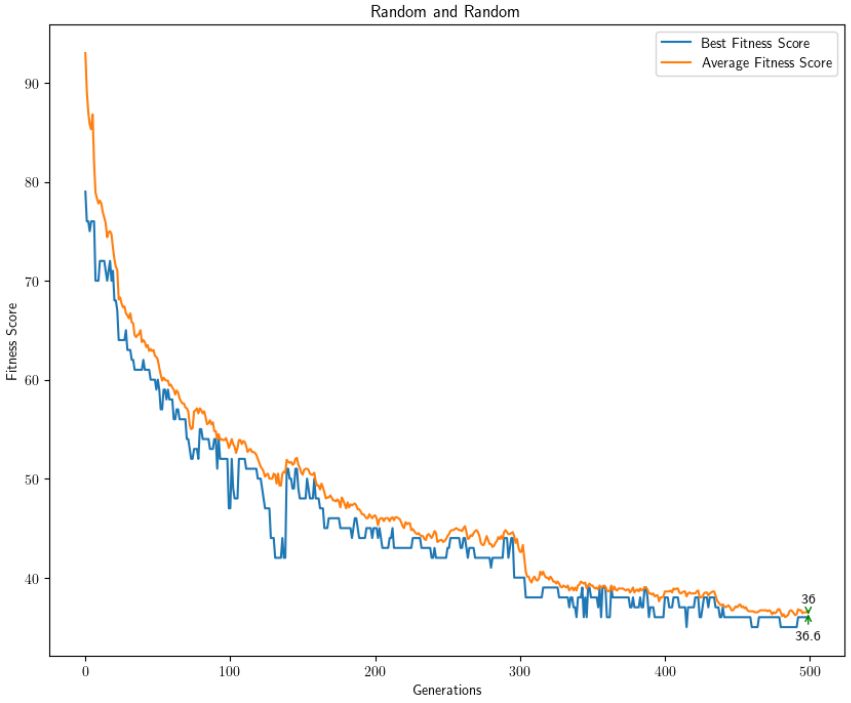
2.3.2 FPS & Truncation Plot

Best Fitness Score: 36
Average Fitness Score: 36



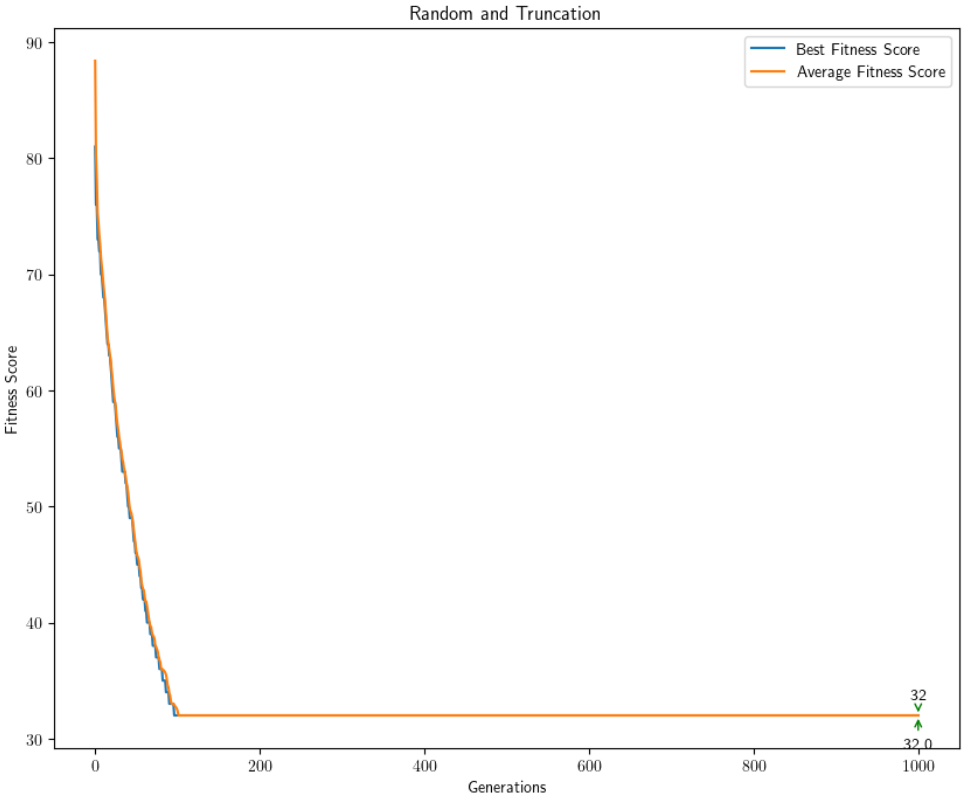
2.3.3 Random & Random Plot

Best Fitness Score: 36
Average Fitness Score: 36



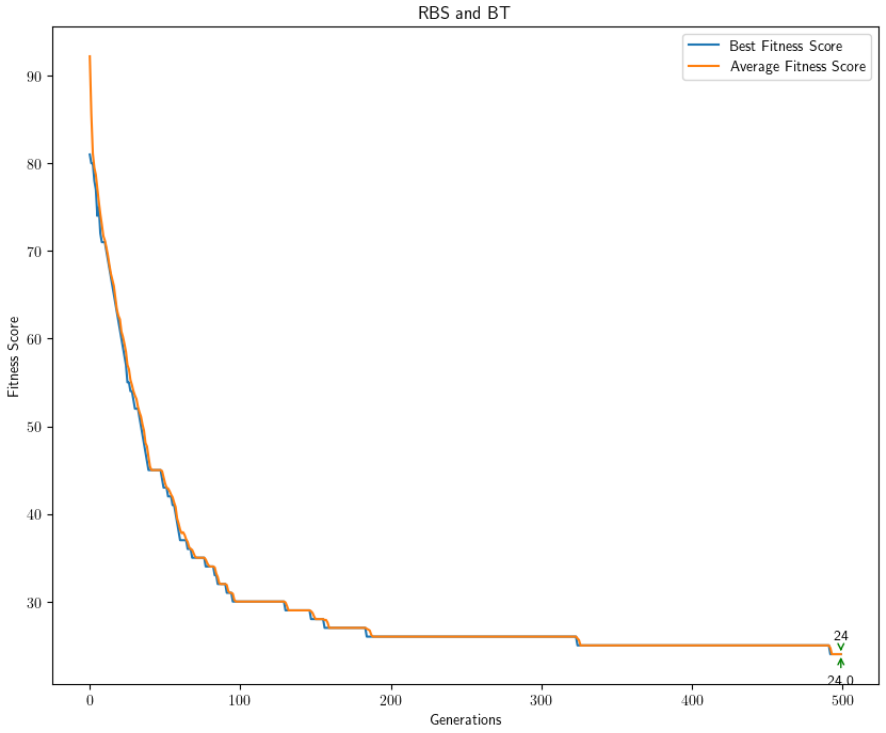
2.3.4 Random & Truncation Plot

Best Fitness Score: 32
Average Fitness Score: 32



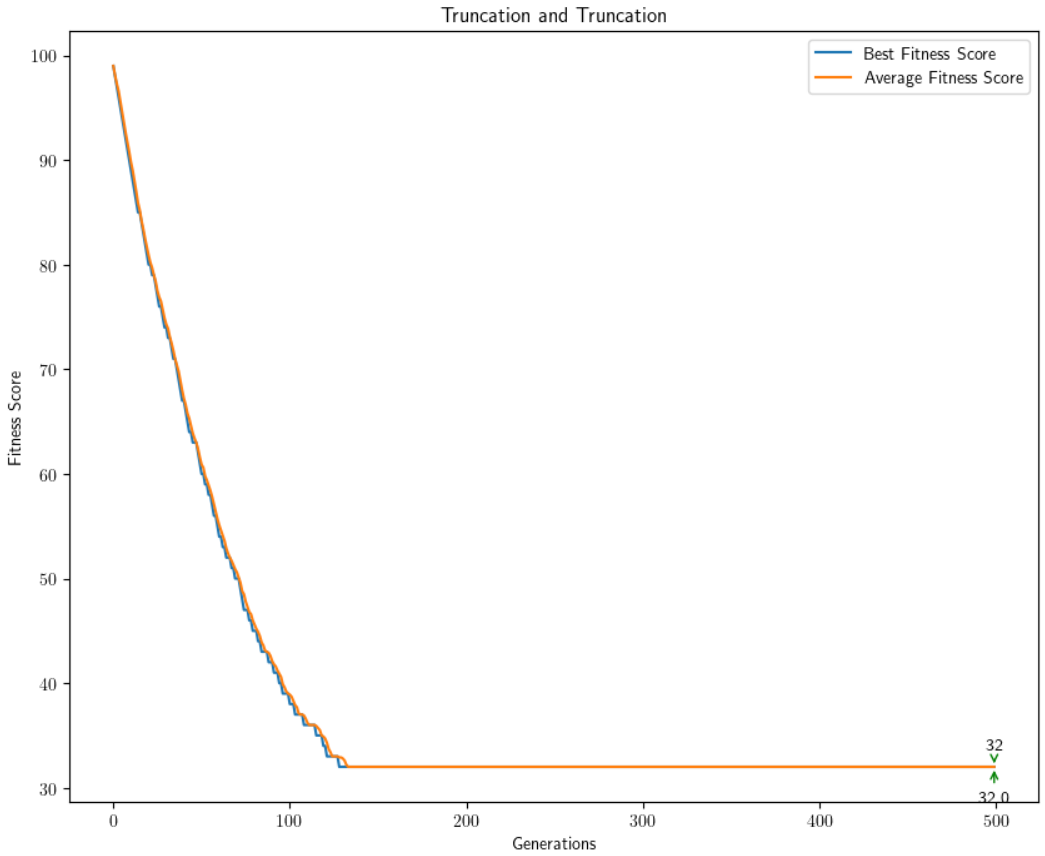
2.3.5 RBS & BT Plot

Best Fitness Score: 24
Average Fitness Score: 24



2.3.7 Truncation & Truncation Plot

Best Fitness Score: 32
Average Fitness Score: 32



2.4 Analysis

Population of 10 was initialized. The crossover picks genes from parents and checks if the graph is correctly colored i.e. no adjacent nodes have the same color. Mutation picks a random gene and makes it equal to some already existing color. The minimum number of colors that can be used to color the graph in this way was found to be 31 in our population.

3. Knapsack Problem

3.1 Chromosome Representation

A chromosome in the knapsack problem is just the collection of items, the sum of whose weights remains below the set weight limit.

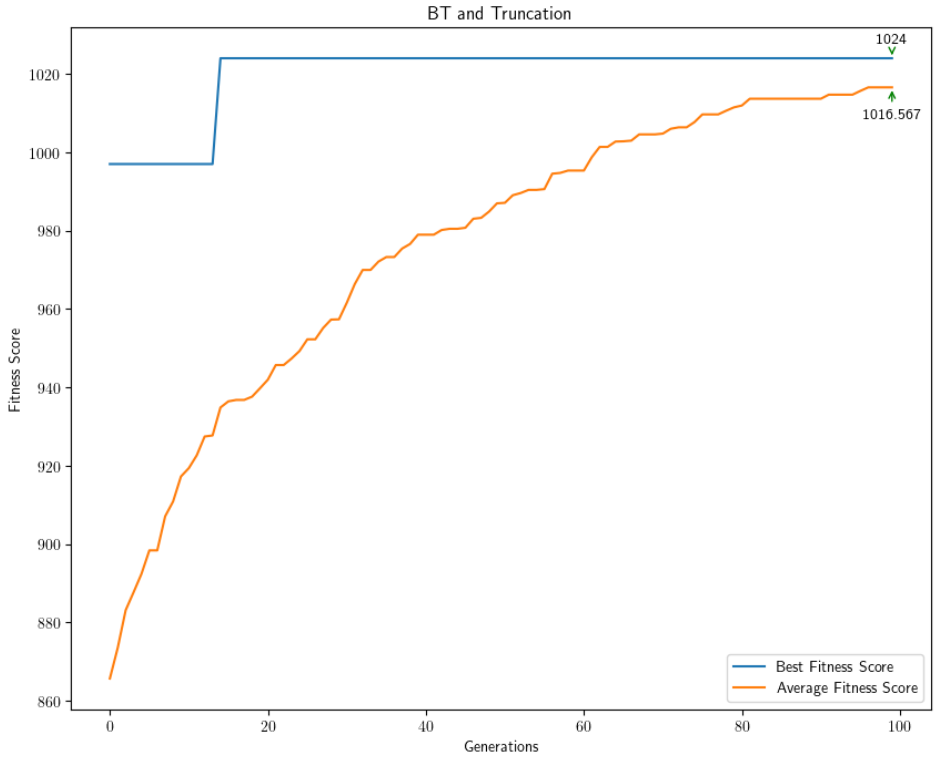
3.2 Fitness Function

The fitness of a chromosome in the knapsack problem was just the sum of all the values of the items in that chromosome.

3.4 EA Evaluation

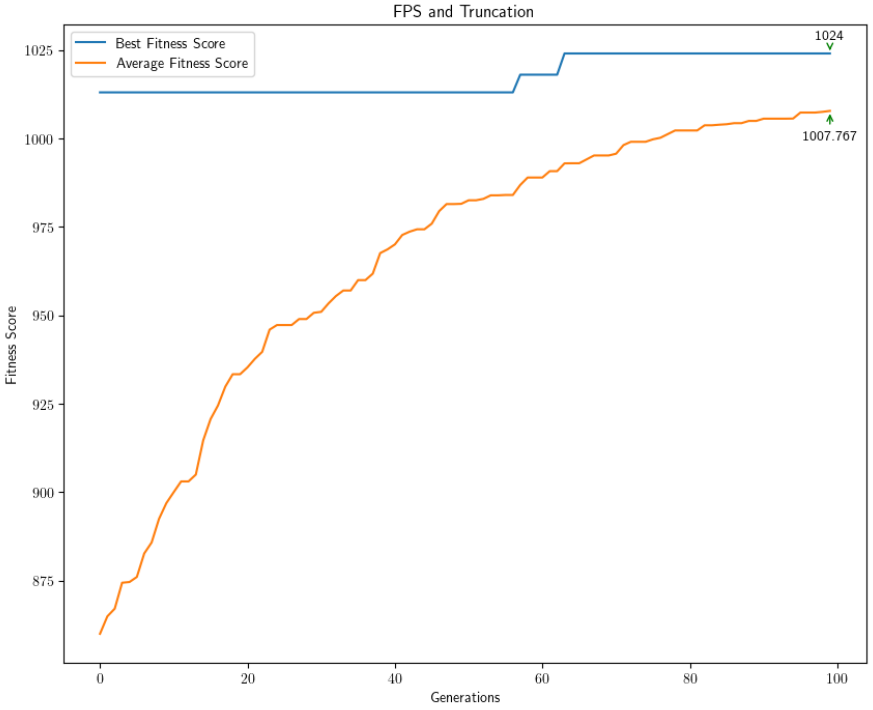
3.4.1 BT & Truncataion Plot

Best Fitness Score: **1024**
Average Fitness Score: **1016.567**



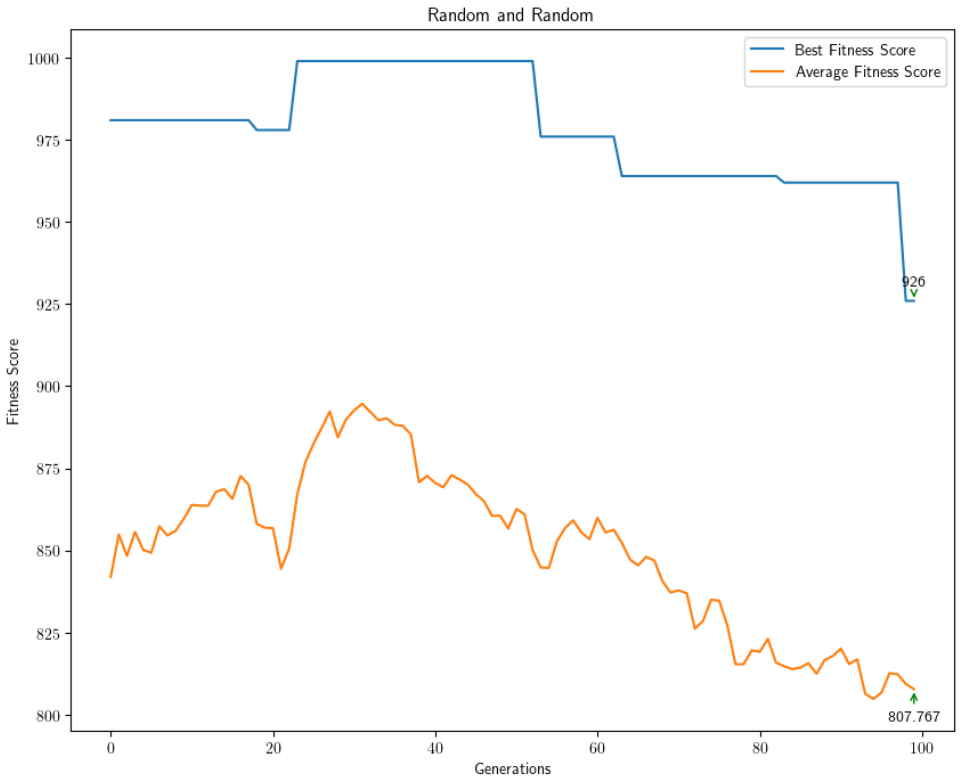
3.4.2 FPS & Truncation Plot

Best Fitness Score: **1024**
Average Fitness Score: **1007.767**



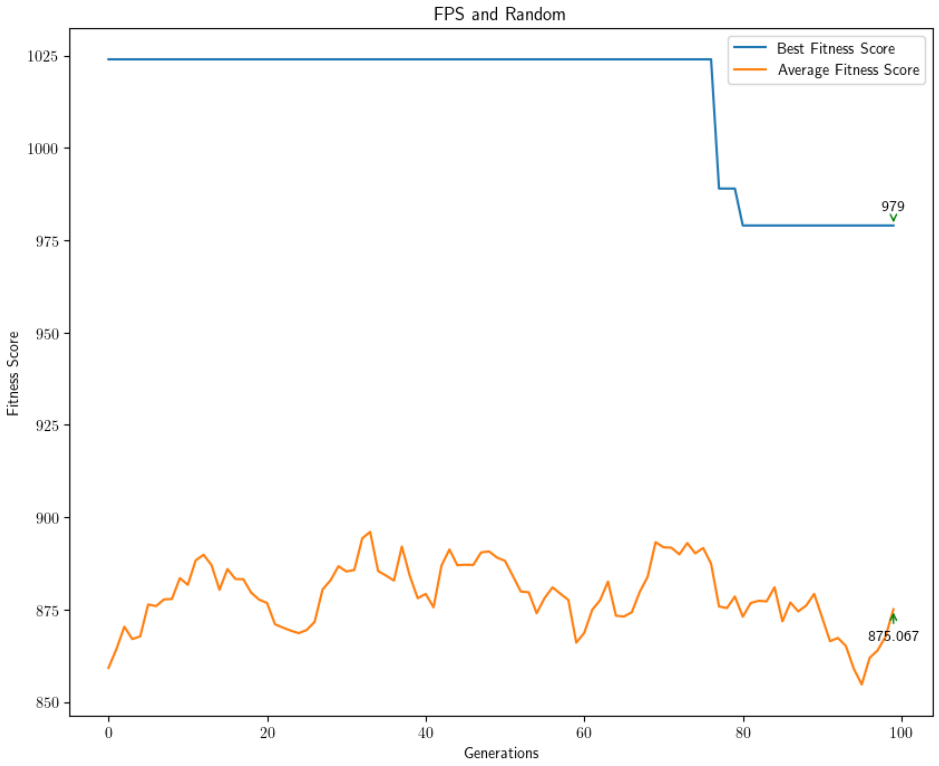
3.4.3 Random & Random Plot

Best Fitness Score: **926**
Average Fitness Score: **807.767**



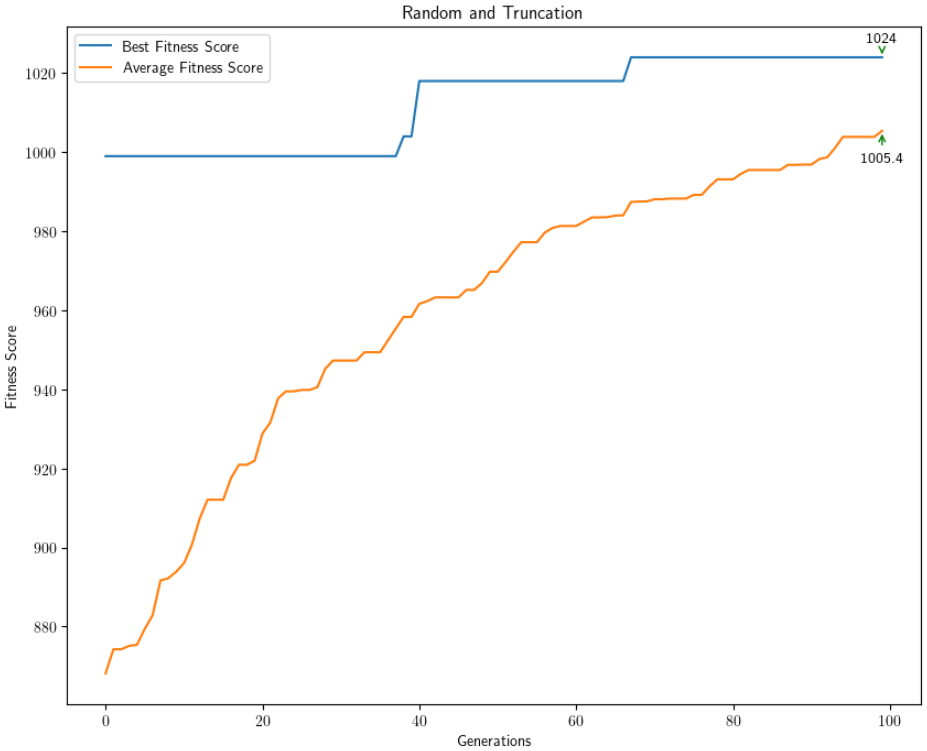
3.4.4 FPS & Random Plot

Best Fitness Score: **979**
Average Fitness Score: **875.067**



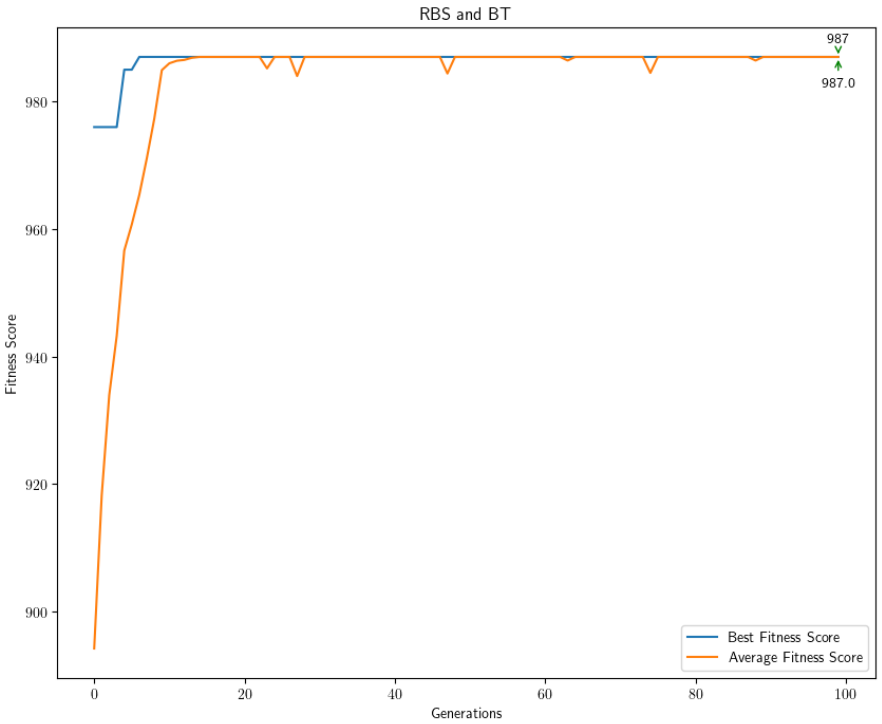
3.4.5 Random & Truncation Plot

Best Fitness Score: **1024**
Average Fitness Score: **1005.4**



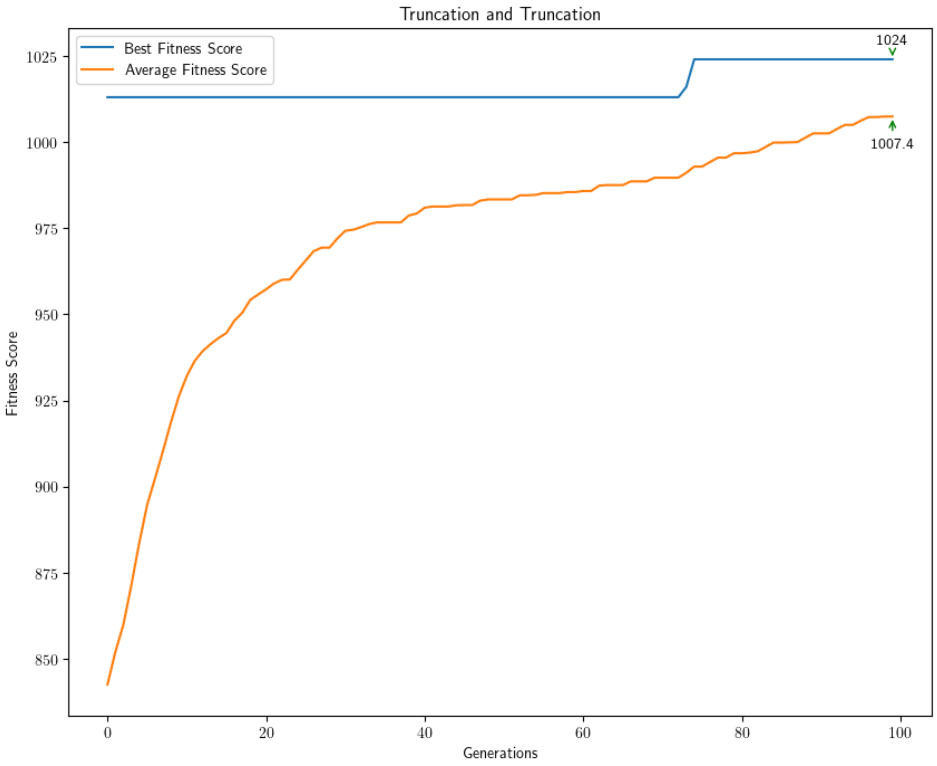
3.4.6 RBS & BT Plot

Best Fitness Score: **987**
Average Fitness Score: **987.0**



3.4.7 Truncation & Truncation

Best Fitness Score: **1024**
Average Fitness Score: **1007.4**



3.5 Analysis

We saw that the average score remained to be 1019 at the end of 50 generations. The best score; however, reached 1024 on around the ninth generation.