CS 451 - Computational Intelligence

Assignment 1 - Evolutionary Algorithm

Instructor - Dr. Saleha Raza
Authors - Abdullah Siddique (ms03586) and Kuldeep Dileep (kl04008)

Contents

1 A Note on Acronyms:

Note that the following acronyms have been used throughout this report as well as the code files:

- BT = Binary Truncation
- FPS = Fitness Proportional Scheme
- RBS = Rank Based Scheme
- BFS = Best Fitness so Far
- ASF = Average Fitness so Far
- PS = Parent Selection
- SS = Survivor Selection
- Also note that wherever we have referred to schemes in the fashion of X&Y or X and Y, the former X refers to the Parent Selection scheme while the latter Y refers to the Survivor Selection scheme.

2 Travelling Salesman Problem (TSP)

In this problem we are given a dataset containing 194 cities of Qatar and we have to find the shortest possible route that visits each city exactly once and returns to the origin city. To achieve this we have implemented selection schemes such as Fitness Proportional Selection (FPS), Rank Based Selection (RBS), Binary Tournament (BT), Truncation and Random selection. Following are the results generated by our algorithm for different combinations of selection schemes for Parent Selection (PS) and Survivor Selection (SS).

2.1 Results:

All the following results are plotted for 500 generations, 10 iterations, 30 number of population and 10 number of off-springs with a mutation rate of 0.5. These values were chosen since it was computationally fast enough that it allowed us to plot a lot of the different scheme combinations with ease so that we could note the difference in trends as the schemes changed.

2.1.1 When PS = FPS and SS = FPS:

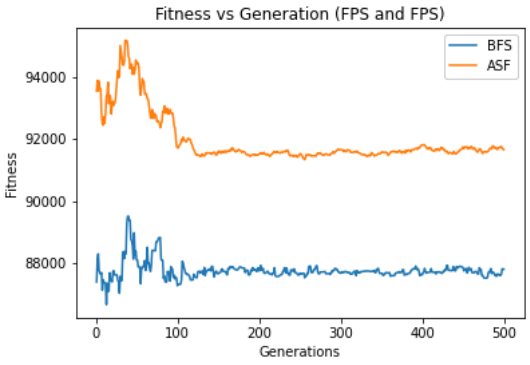

Figure 2.1: Fitness vs Generation plot

2.1.2 When PS = RBS and SS = RBS:

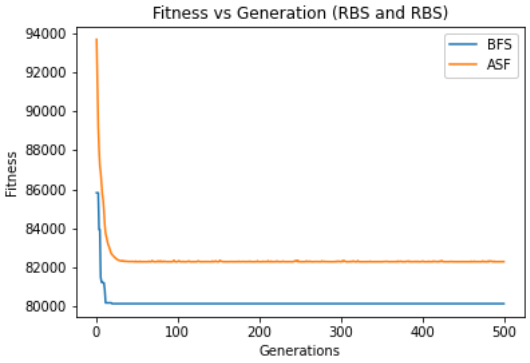

Figure 2.2: Fitness vs Generation plot

2.1.3 When PS = BT and SS = BT:

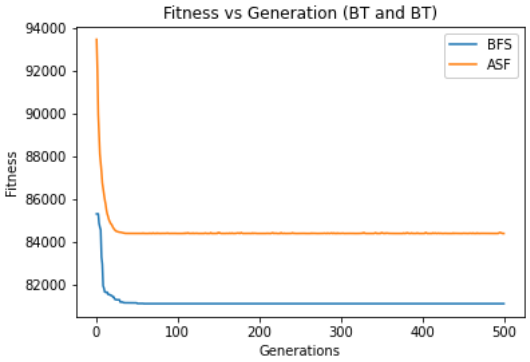

Figure 2.3: Fitness vs Generation plot

2.1.4 When PS = Truncation and SS = Truncation:

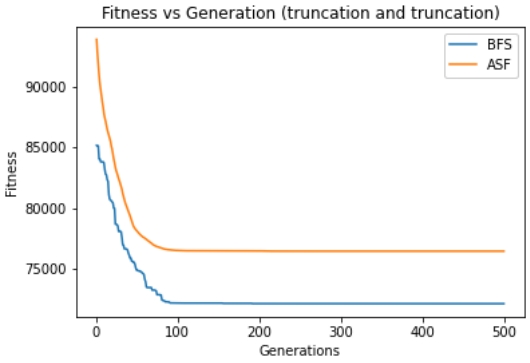

Figure 2.4: Fitness vs Generation plot

2.1.5 When PS = Random and SS = Random:

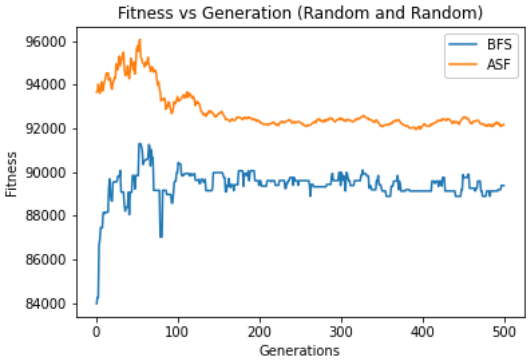

Figure 2.5: Fitness vs Generation plot

2.1.6 When PS = FPS and SS = RBS:

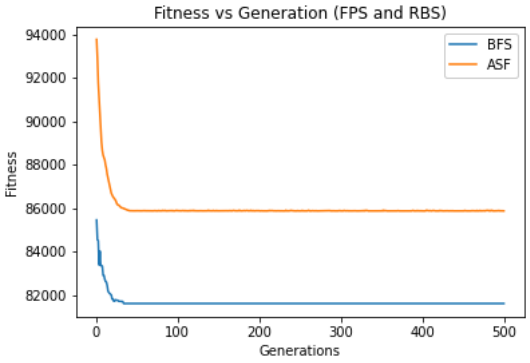

Figure 2.6: Fitness vs Generation plot

2.1.7 When PS = RBS and SS = FPS:

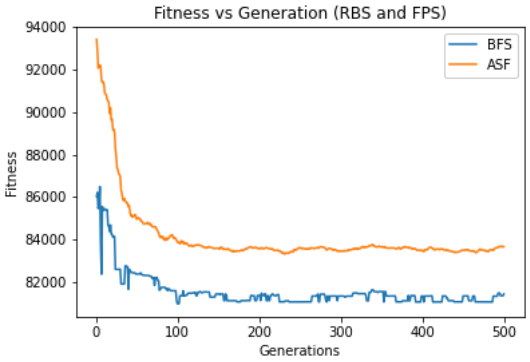

Figure 2.7: Fitness vs Generation plot

2.1.8 When PS = RBS and SS = BT:

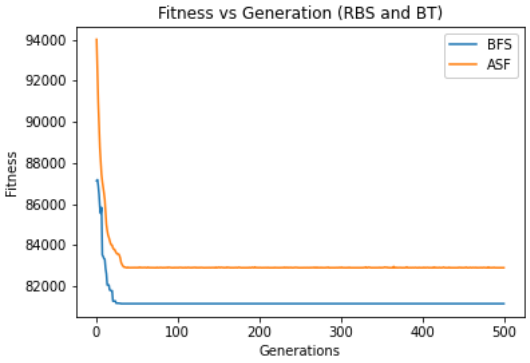

Figure 2.8: Fitness vs Generation plot

2.1.9 When PS = BT and SS = RBS:

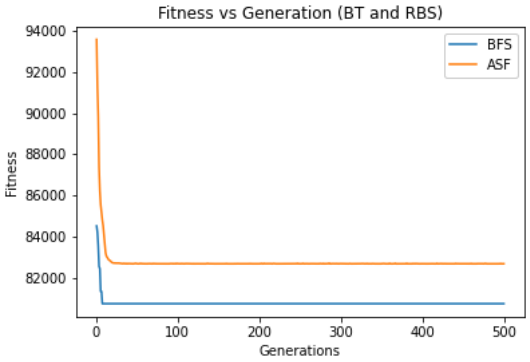

Figure 2.9: Fitness vs Generation plot

2.1.10 When PS = BT and SS = Truncation:

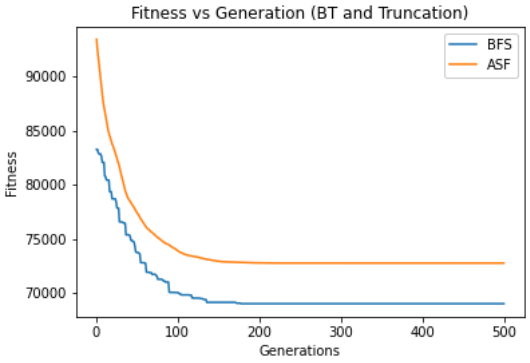

Figure 2.10: Fitness vs Generation plot

2.1.11 When PS = Truncation and SS = BT:

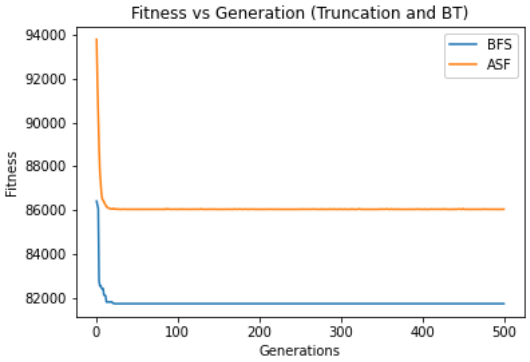

Figure 2.11: Fitness vs Generation plot

2.1.12 When PS = Truncation and SS = Random:

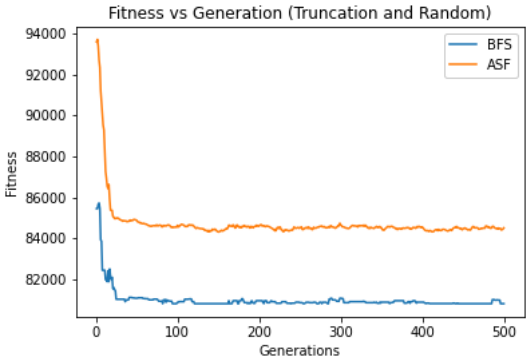

Figure 2.12: Fitness vs Generation plot

2.1.13 When PS = Random and SS = Truncation:

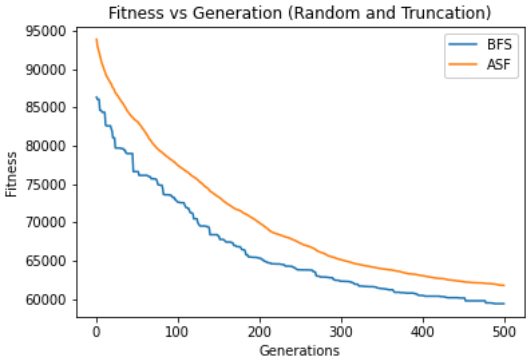

Figure 2.13: Fitness vs Generation plot

2.1.14 When PS = FPS and SS = Random:

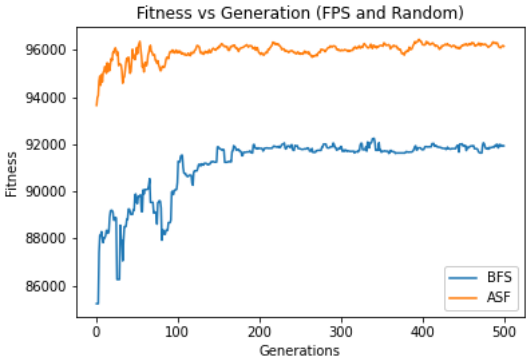

Figure 2.14: Fitness vs Generation plot

2.1.15 When PS = Random and SS = FPS:

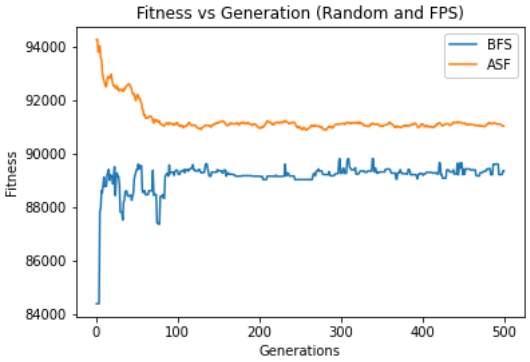

Figure 2.15: Fitness vs Generation plot

2.1.16 When PS = FPS and SS = Truncation:

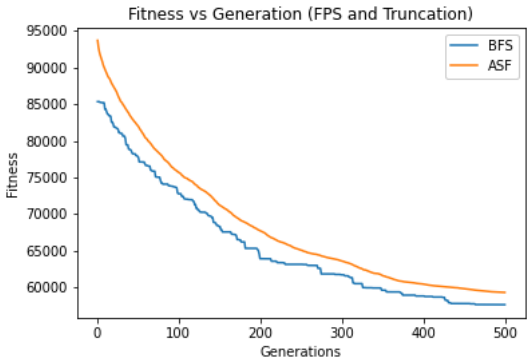

Figure 2.16: Fitness vs Generation plot

2.1.17 When PS = Truncation and SS = FPS:

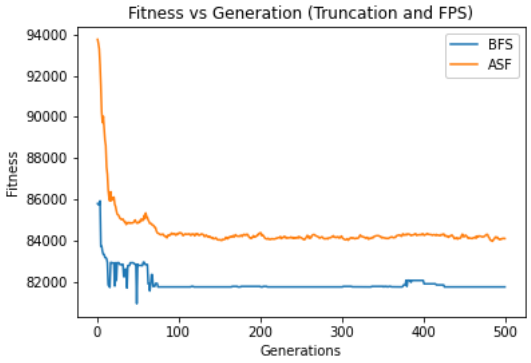

Figure 2.17: Fitness vs Generation plot

2.2 Analysis:

2.2.1 Chromosome Representation:

The problem was representing by creating a population of individuals with a chromosome the length of 194 elements. Each element would represent one city. The code ensured that no gene i.e. city would be repeated in the chromosome. The order of cities in chromosome defines the order in which each city will be visited, with the first city also being visited after the last city to complete a loop.

For initialization, a sorted list was generated of numbers from 1 to 194, and then shuffled in order to achieve a randomly distributed initial population.

In each generation, we strove to improve the solutions represented by our population by manipulating the chromosomes of individuals. Apart from parent and survivor selection on the basis of the fitness values, two point crossover was implemented for knowledge sharing. Two point crossover means that a random chunk of cities was lifted from parent 1 and inserted into child 1. Then all the cities that were not already in child 1, were lifted from parent 2 and inserted in the remaining places in child 2, in the same order that they appeared in parent 2. The same process was repeated with the order of parents reversed, to obtain child 2. This process is more apparent in the two images below. The red lines in Fig. 2.18 show where the parents were partitioned from.

Parent 1	A C	B F H	D E G	43
Parent 2	F H	A D C	B E G	49

Figure 2.18: Parents being partitioned for Crossover

Offspring 1	D C B F H E G A
Offspring 2	F H A D C E G B

Figure 2.19: off-springs obtained as result of aforementioned Crossover

For mutation, we simply randomly exchanged two randomly selected cities with a probability of 0.5.

2.2.2 Fitness Functions:

Since this is a version of the traveling salesman problem, we chose to calculate the combined distance between the cities as the metric to represent fitness. The Euclidean distance between every consecutive pair of cities in a chromosome, as well as the last and the first cities, was calculated and summed up. This value represented the total distance for a specific individual solution.

However, note that we had an implementation in place that was working towards maximizing the fitness value from our work on the Knapsack problem. Therefore, in the interest of maintaining a generic implementation, we simply took the reciprocal of the total distance to obtain the fitness value for one individual. This works since minimizing a value is the same as maximizing its reciprocal.

2.2.3 Varying Schemes and Parameters:

We have attached graphs detailing many of the different combinations of parent and survivor selection schemes possible.

Apart from manipulating the survivor schemes, other parameters we varied were the mutation rate, the number of generations and iterations, and the population size and number of off-springs.

We noticed that the results were generally best for a mutation value for 0.5. The fitness values wouldn't improve as much for mutation values both above and below 0.5.

We also noticed that if we tried to increase population and/or off-springs, there seemed to be an improvement in the fitness value at first, but after certain population and offspring values, the fitness values weren't improving by a significant amount anymore. We couldn't check up to really high magnitudes of population/off-springs since it got computationally expensive and started to take up precious time.

Similarly, increasing the number of generations also gave us increasing returns at first, but beyond a certain value the rate of improvement of fitness was small enough and so computationally expensive that we chose not to pursue it.

A lot of the scheme combinations gave us terrible results initially. Lots of them went on to converge at very high values of fitness for example: FPS&FPS, RBS&RBS, BT&BT, Truncation&Truncation, FPS&RBS, RBS&BT. Note that here the first value signifies parent selection and the second signifies survivor selection. Initially, the only promising trends were shown by FPS&Truncation and Random&Truncation. Note that these graphs were obtained at the relatively smaller magnitudes of population = 30 and offsprings = 10. However, much to our disappointment, even increasing the generations did not get us results below 50k with either of these two combos.

We then started to vary the population and tested again for the different combination of schemes. All of the many, many, many results we obtained while searching for the optimum solution have not been attached in the interest of saving time and avoiding redundant complexity.

The Best Solution:

We obtained the best value for the fitness for a combination of Binary Tournament for Parent Selection and Truncation for Survivor Selection, with a population of 200 and 150 off-springs, with 5k generations. The value obtained was about 18,500. The minimum achievable value for this problem is 9352 according to the website. The result graph for this combo with the reciprocal of the fitness on the y-axis can be seen in Fig. 2.20. We were unable to plot the graph with the corrected magnitudes on the y-axis due to lack of time, it was taking too long to render.

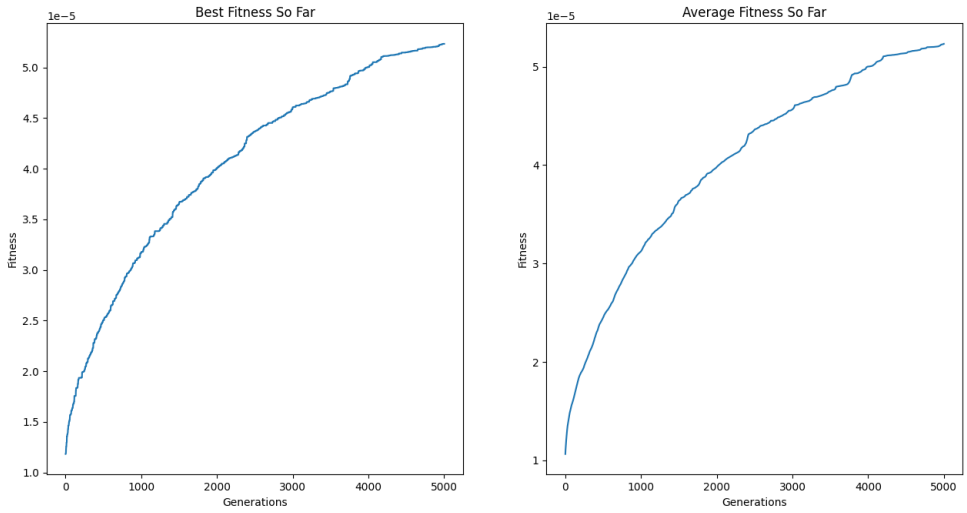

Figure 2.20: Reciprocal of Fitness vs Generation plot

Some promising results were obtained which did not achieve convergence within the specified number of generations, and may have even improved beyond the aforementioned results if allowed to run for longer than the specified number of generations. These are mentioned below.

A promising result was obtained for combination of Binary Tournament for Parent Selection and Truncation for Survivor Selection, with a population of 800 and 1000 off-springs, with 2k generations. The value obtained was about 21,000. This was not pursued further because 2k generations for this combo took much much longer than the 5k generations for the population = 200, off-springs = 150 combination. The result graph for this combo with the reciprocal of the fitness on the y-axis can be seen in Fig. 6.3.

We also obtained a promising result for the combination of Fitness Proportionate Scheme for Parent Selection and Truncation for Survivor Selection, with a population of 200 and 150 off-springs, with 2k generations. The value obtained was about 26,000. This is shown in Fig. 2.21

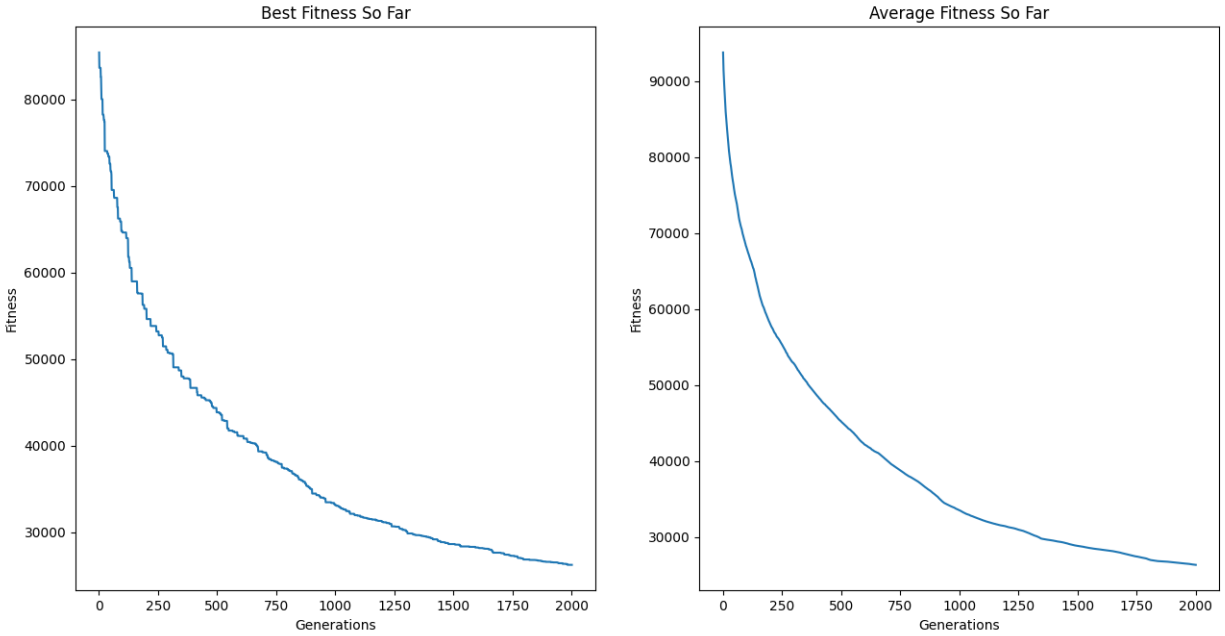

Figure 2.21: Fitness vs Generation plot

3 Knapsack Problem

In this problem, we were given a dataset containing weight and value of 20 items and it is required to give a list of items with optimal value within knapsack capacity which was 878. To achieve this we have implemented selection schemes such as Fitness Proportional Selection (FPS), Rank Based Selection (RBS), Binary Tournament (BT), Truncation and Random selection. Following are the results generated by our algorithm for different combinations of selection schemes for Parent Selection (PS) and Survivor Selection (SS).

3.1 Results

All the following results are plotted for 500 generations, 10 iterations, 30 number of population and 10 number of off-springs with a mutation rate of 0.5. These values were chosen since it was computationally fast enough that it allowed us to plot a lot of the different scheme combinations with ease so that we could note the difference in trends as the schemes changed. Later we increased the resolution of few combinations which seemed to converge faster by reducing the number of generations to 100.

3.1.1 When PS = FPS and SS = FPS:

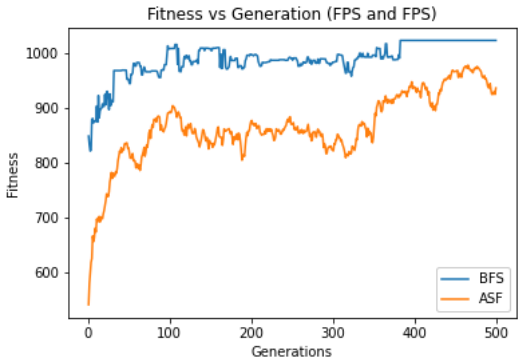

Figure 3.1: Fitness vs Generation plot

3.1.2 When PS = RBS and SS = RBS:

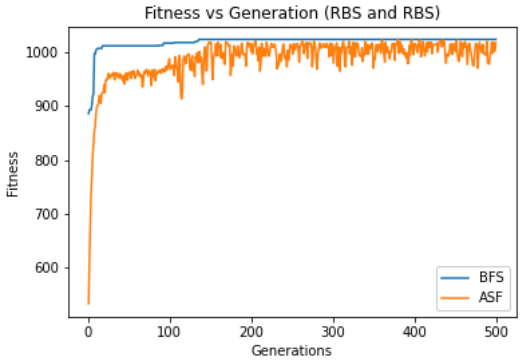

Figure 3.2: Fitness vs Generation plot

3.1.3 When PS = BT and SS = BT:

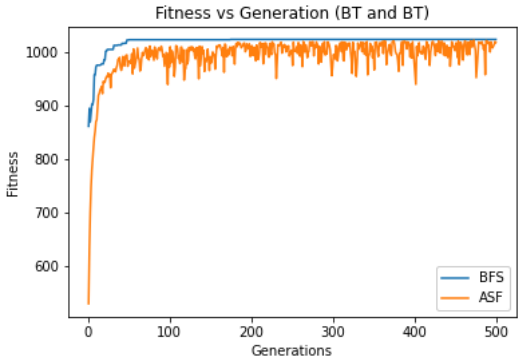

Figure 3.3: Fitness vs Generation plot

3.1.4 When PS = Truncation and SS = Truncation:

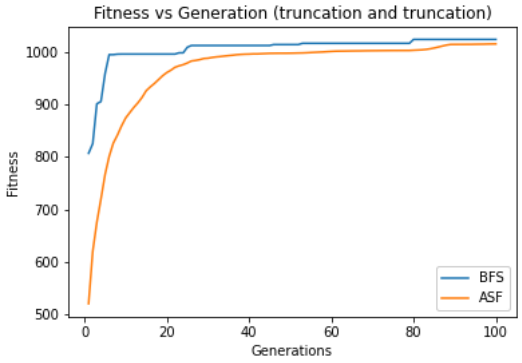

Figure 3.4: Fitness vs Generation plot

3.1.5 When PS = Random and SS = Random:

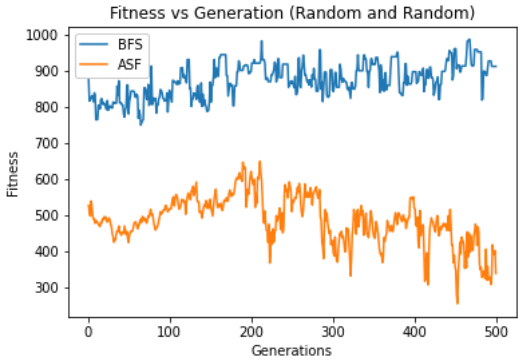

Figure 3.5: Fitness vs Generation plot

3.1.6 When PS = FPS and SS = RBS:

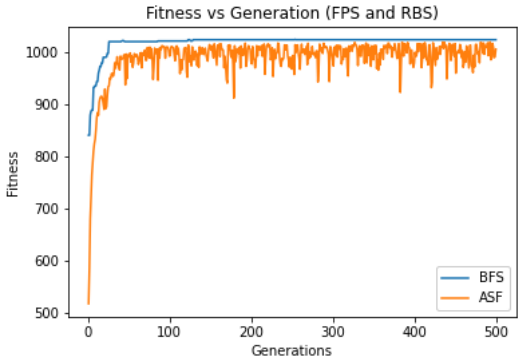

Figure 3.6: Fitness vs Generation plot

3.1.7 When PS = RBS and SS = FPS:

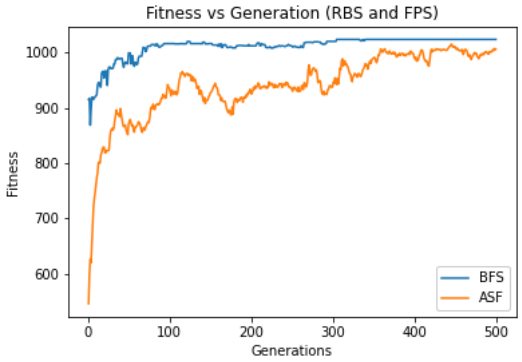

Figure 3.7: Fitness vs Generation plot

3.1.8 When PS = RBS and SS = BT:

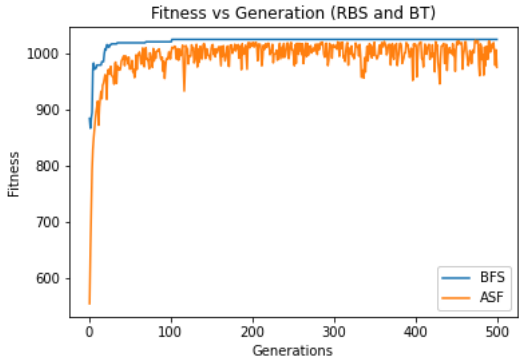

Figure 3.8: Fitness vs Generation plot

3.1.9 When PS = BT and SS = RBS:

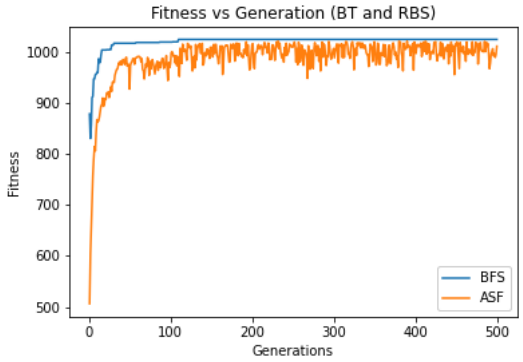

Figure 3.9: Fitness vs Generation plot

3.1.10 When PS = BT and SS = Truncation:

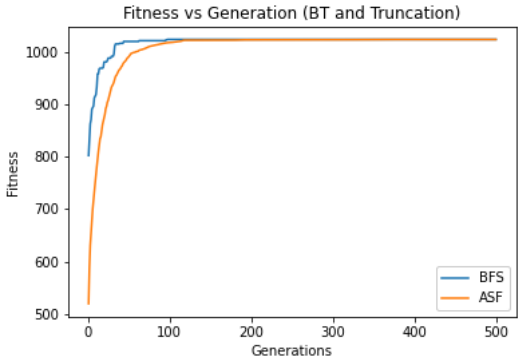

Figure 3.10: Fitness vs Generation plot

3.1.11 When PS = Truncation and SS = BT:

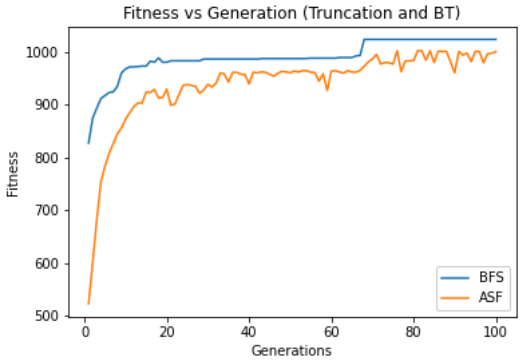

Figure 3.11: Fitness vs Generation plot

3.1.12 When PS = Truncation and SS = Random:

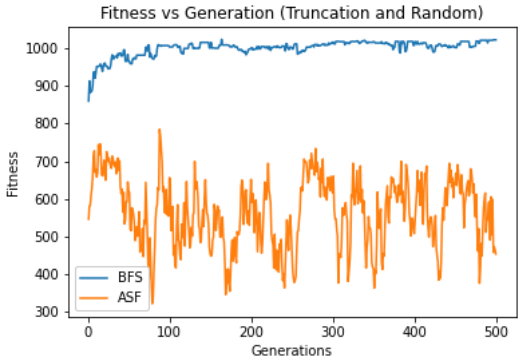

Figure 3.12: Fitness vs Generation plot

3.1.13 When PS = Random and SS = Truncation:

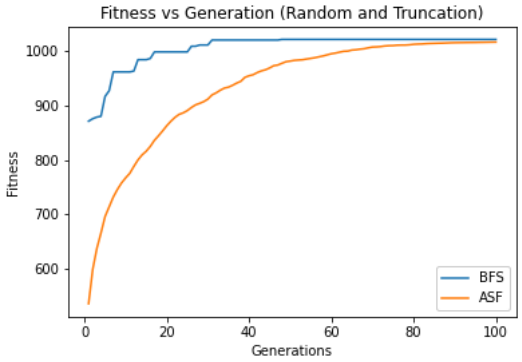

Figure 3.13: Fitness vs Generation plot

3.1.14 When PS = FPS and SS = Random:

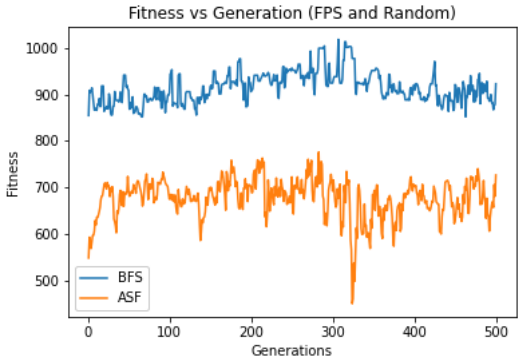

Figure 3.14: Fitness vs Generation plot

3.1.15 When PS = Random and SS = FPS:

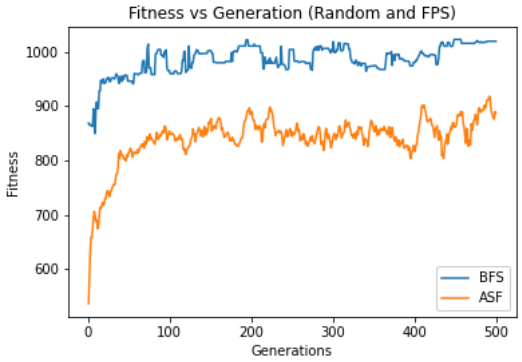

Figure 3.15: Fitness vs Generation plot

3.1.16 When PS = FPS and SS = Truncation:

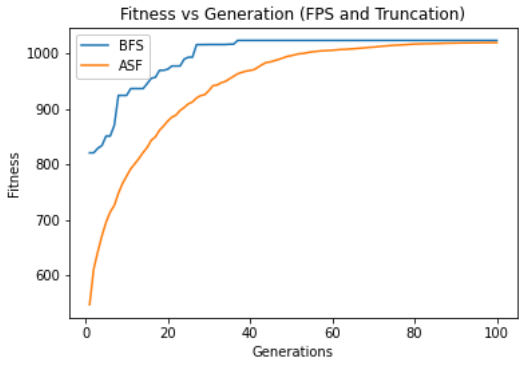

Figure 3.16: Fitness vs Generation plot

3.1.17 When PS = Truncation and SS = FPS:

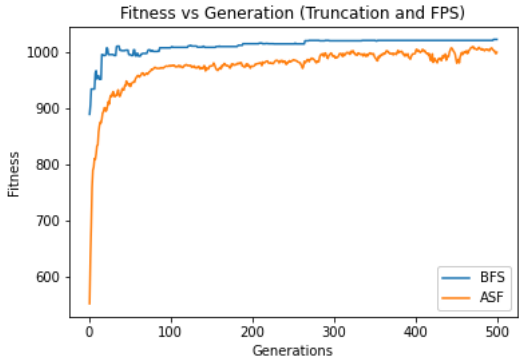

Figure 3.17: Fitness vs Generation plot

3.2 Analysis:

3.2.1 Chromosome Representation:

The Chromosome for this this problem is represented similar to the traditional 'bits' where 1 means the object is present in the knapsack and 0 means it isn't. Essentially, each individual chromosome consists of an array of length equal to the number of unique objects in the knapsack, with a 1 at the Nth index signifying that the (N+1)th object is present in the knapsack, and vice versa.

For example, for the chromosome [0, 1, 0, 1] has non-zero indices at 1 and 3, and thus states that the 2nd and the 4th object are present in the knapsack.

Alongside the chromosome, we also maintain arrays sorted by object order, which contain the value and weight of each object of the knapsack. We can refer between Nth index of this list and the Nth index of a chromosome to find the value and weight of a given object placed in the knapsack.

Initially, chromosomes are randomly generated which might also result in the weight being higher than the maximum capacity of the knapsack but this is problem is dealt by the fitness function, once the algorithm starts running.

3.2.2 Fitness Function:

To compute fitness of each individual we compute weighted sum of list of values in a chromosome by multiplying each bit by its corresponding weight of the item and then we add them. If the weighted sum exceeds the maximum capacity of the knapsack i.e. 878 then that individual's fitness is zero; otherwise, it is equal to the total value of items in the knapsack. Mathematically:

$$Fitness = \sum_{i=1}^{n} c_i * v_i$$

$$if : \sum_{i=1}^{n} c_i * w_i \leq cap$$

Otherwise, fitness = 0

where,
n = length of chromosome
c_i = ith bit of chromosome
v_i = value of the ith item
w_i = weight of the ith item
cap = knapsack capacity

Once the fitness is computed for every individual in the population then various selection schemes are used to select parents which undergo crossover and mutation resulting in off-springs.

We have implemented single point crossover where an index number is randomly generated which marks the point where both the parents' chromosomes split. The resultant split portions from two different parents are combined resulting in a pair of off-springs which have some portion of the chromosome of each parent. Considering the complexity of the problem we decided to use single point crossover because it is simple and hence, reduces the computation power and time required as compared to other advance crossover techniques. Whiled doing crossover we still have to ensure that the knapsack capacity is not exceeded.

For mutation amongst the off-springs, we randomly switch any two bits while ensuring that the weight value does not exceed the knapsack capacity.

3.2.3 Varying Schemes and Parameters:

We also tuned quite a few hyperparamters to improve the convergence rate of our algorithm and observed several interesting behaviours.

For instance, when we increased the size of population, where parent selection scheme was FPS and survivor selection scheme was truncation, so it reduced the convergence rate such that it converged at 60 generation. Similarly, when we increased the mutation rate while keeping rest of the parameters constant, then the algorithm converged much faster at 20 generations but it converged to 1022 which is non optimal.

The worst results were obtained using the schemes of FPS&Random, Random&FPS, Truncation&Random, Random&Random, and FPS&FPS which were not converging to the correct optimal value and also displaying highly volatile behaviour.

The Best Solution:

Unlike TSP, in this problem multiple combinations of parent and survivor selection scheme give the optimal result which is 1024 in their BFS graphs. These schemes are FPS&Truncation, Random&Truncation, RBS&BT, BT&RBS, FPS& RBS etc.

In our opinion, the scheme for parent selection using BT and survivor selection using Truncation seem to not only give the optimal result but displays a more stable & smoother curve and converges much faster than others.

4 A Comment on Generic Implementation:

We thought it might be helpful for the reader who might go through the code to understand what aspects of it are repeated or reused and which are specific to the problem.

We have defined a class called EA (standing for Evolutionary Algorithms) to solve our Knapsack and TSP problems. For both problems, we have some specialized functions in the class and some functions which are being reused.

The function which deals with the selection of different parents & survivor schemes is the same in both the problems. As is the function that mutates the chromosomes of the off-springs (by simply switching two values at random). The code of chunk that runs the many different iterations/generations for evolution to happen, and the code that deals with plotting graphs and storing the data to a text file, are all the same as well.

The specialized functions for each of the Knapsack and the TSP problems are the fitness calculation function, the crossover function, and the chunk of code representing the chromosome.

We believe that it is possible to have a common crossover function for both the problems as well, but for that we will have to change the representation of the Knapsack problem. We were unable to do so because of time limitations. The difference in representation and thus the need for different crossover functions occurs because the work was divided between the group partners, who approached the problem in their own ways.

5 Evolutionary Art:

Disclaimer: Evolutionary Art is Difficult.

5.1 Approach:

We have unfortunately had very limited success with the problem of attempting to regenerate the Mona Lisa picture using polygons. In this section, we will explain our approach to the problem, and detail our obtained results, such as they are.

We pursued writing the code in Python, looking at various online implementations for inspiration. We drew on the same model that we used to solve the Knapsack and TSP problems, and attempted to scale to this problem.

We defined one class for polygons, which were represented using an array of vertices, and their colors in 'RGBA' format.

We generated a population of N number of individuals. Each individual represented an image. For each individual, We generated 50 polygons, of various randomly selected vertex coordinates, sides, colors, assigning them all to be semi-transparent. This was our initialization.

We then calculated the fitness values of our test solutions by taking the absolute difference of the polygon image with the original image, and noting down the resultant sum of differences of all the pixels, normalized by the numbers of pixels.

We then attempted to implement mutation and crossover, alongside the same process for parent/survivor selection schemes that we did in the TSP and Knapsack problems.

Crossover was attempted at different levels with different resolutions. The simplest way was to take a random number of polygon from one parent and (50-random) number from the other parent to get the children. We also switched the polygon vertex coordinates between each other. We added coordinates to one polygon while removing from the other at random. We switched the RGBA values at random between polygons while constructing the child image.

Mutation was also attempted at different resolutions. One way was to exchange the vertices of any two polygons in an image. Another way was to switch the RGBA values. We also generated an offset in the polygon coordinates w.r.t the origin at random.

Another tangent that we pursued in addition to all the above steps, was to periodically add polygons at random if they would have a significant impact on the fitness, measured by X amount of difference.

5.2 Results:

However, we were unable to get anything resembling the Mona Lisa in all of our attempts.

We attempted to run our algorithm with a large population (50), but it took about 2 hours to simply run 15 generations giving unsatisfactory results. These are shown in Fig. 5.1.

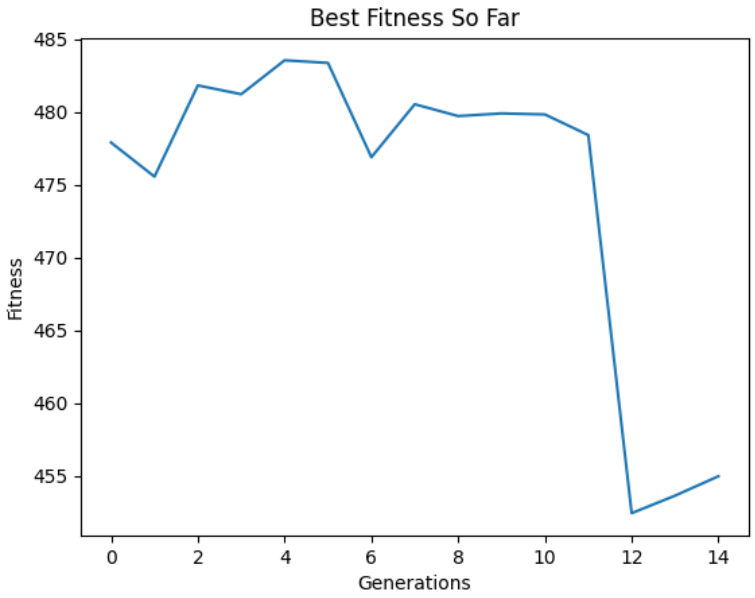

Figure 5.1: Fitness vs Generations Plot

We also attempted to run the algorithm with 5 population and 2 off-springs for 1500 generations. The results were obtained quicker but were not much better as shown in Fig. 5.4

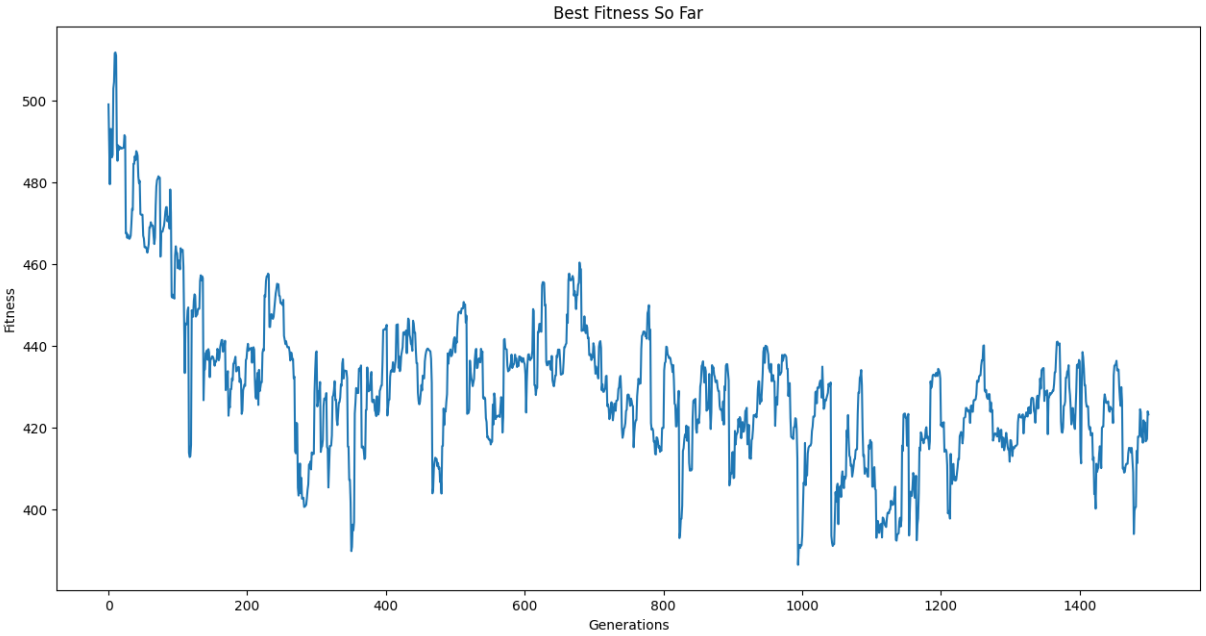

Figure 5.2: Reciprocal of Fitness vs Generation plot

For the aforementioned combination, we obtained the following image, which has little resemblance to the Mona Lisa.

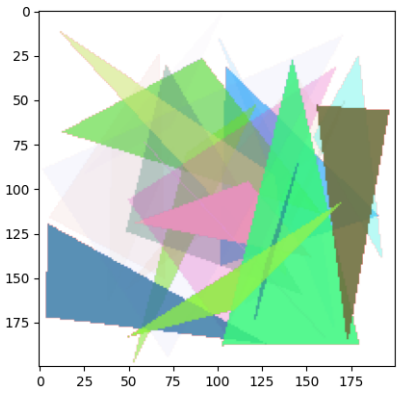

Figure 5.3: Recreation of the Mona Lisa: An Attempt

When we were trying to add polygons to achieve the same, we got the following result for about 500 polygons.

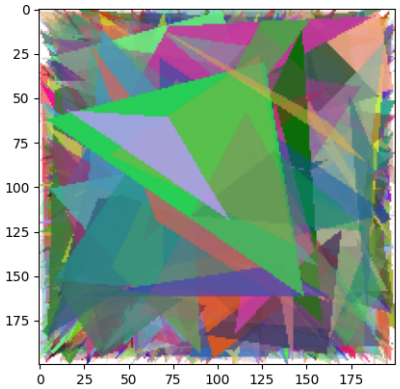

Figure 5.4: Using 500 polygons to recreate the Mona Lisa

6 Appendix:

In this section we have attached results for TSP achieved from tuning other hyper-parameters like population size, generation, and number of off-springs where schemes for parent and survivor selection were kept constant. These results were from the initial iterations of our work when we were plotting the reciprocal of the fitness instead of the fitness itself. The fitness in the plots below can be computed by taking 1/(y-axis magnitude).

When generation = 2000, iteration = 1, population = 100 and number of off-springs = 70

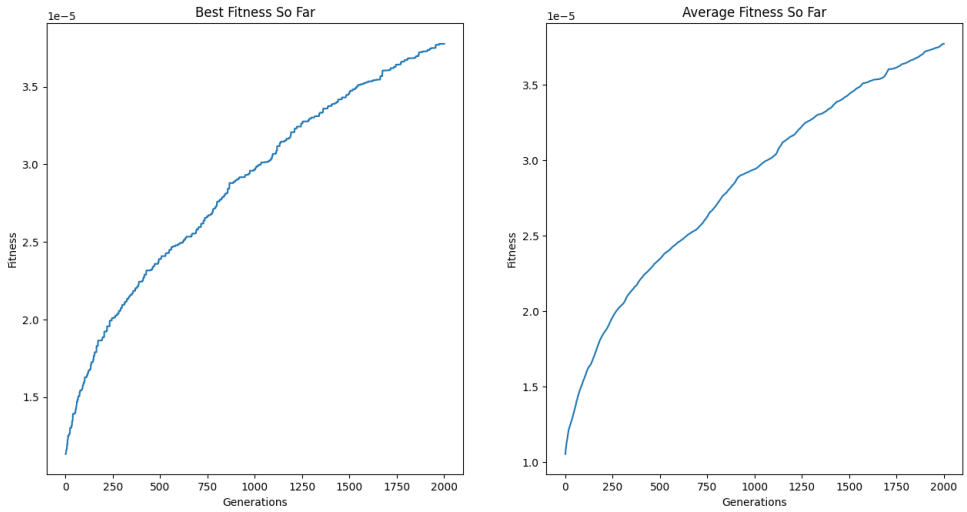

Figure 6.1: Reciprocal of Fitness vs Generation plot

When generation = 2000, iteration = 1, population = 500 and number of off-springs = 350

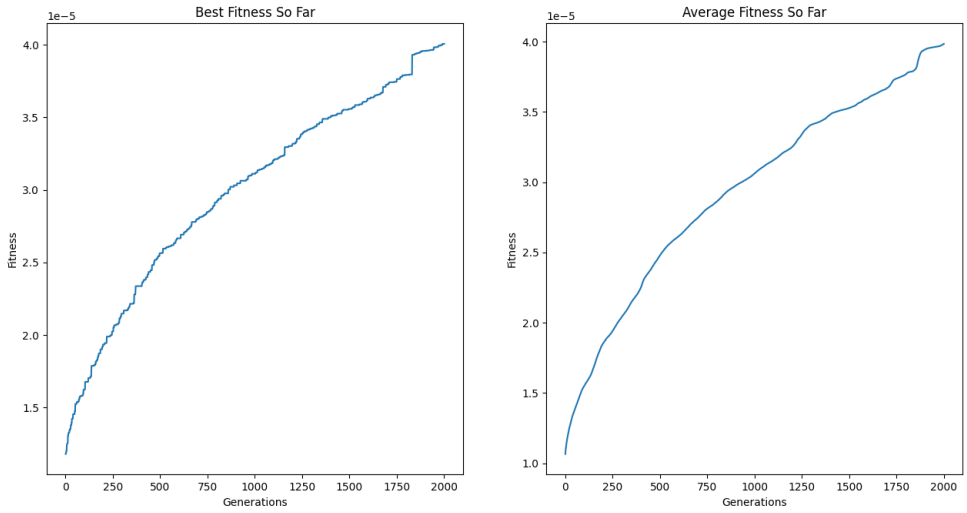

Figure 6.2: Reciprocal of Fitness vs Generation plot

When generation = 2000, iteration = 1, population = 1000 and number of off-springs = 800

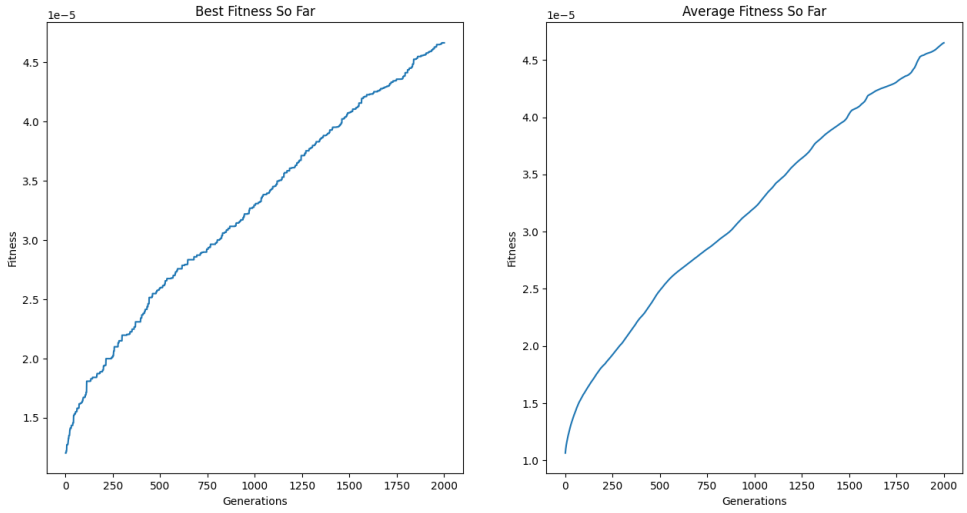

Figure 6.3: Reciprocal of Fitness vs Generation plot

When generation = 3000, iteration = 1, population = 250 and number of off-springs = 200

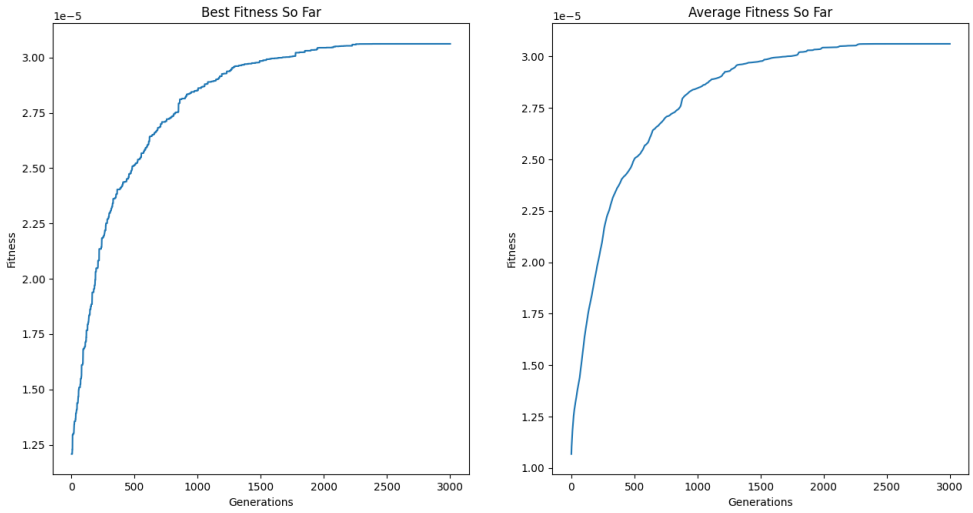

Figure 6.4: Reciprocal of Fitness vs Generation plot

6.1 Naming convention for text files attached:

File named as $ASF\_iteration\_gen(xandy)$ means that it holds the tabular data of EA implementation of knapsack (if found in $knapsack\_graphs$ folder) or TSP (if found in TSP_graphs folder) where x is the parent selection scheme and y is the survivor selection scheme. This contains Average Fitness so Far (ASF) in each iteration and generation. Vertical axis signifies number of generation and each column signifies ASF at each iteration. Similarly, if instead of ASF it starts with BFS then everthing is same except it stores BFS instead of AFS.

File named as $gen\_BFS\_ASF(xandy)$ stores the BFS and AFS values in each generation where x signifies the parent selection scheme and y signifies survivor selection scheme.