# Evolutionary Algorithms

**Muhammad Meesum Ali Qazalbash - mq06861**
**Syed Ibrahim Ali Haider - sh06565**

February 8, 2023

CS451 - Computational Intelligence

Assignment 1

# Contents

# Chapter 1

# Abstract

The purpose of this assignment is to gather an insight of stochastic optimization using Evolutionary Algorithms (EA). This exercise will enable us to address some known computing problems that map to several real-world problems and are known to be computationally hard.

# Chapter 2

# Problems

## 2.1 Genetic Algorithm

The genetic algorithm is a search heuristic that replicates the process of natural evolution. This is represent the class with the following attributes,

```
    def __init__(self,
                 problem: Problem,
                 selection_method1: int = 0,
                 selection_method2: int = 0,
                 population_size: int = 30,
                 number_of_offsprings: int = 10,
                 number_of_generations: int = 100,
                 mutation_rate: float = 0.50) -> None:
```

Listing 2.1: Evolution class

The fittest chromosome is selected the chromosome maximized by the fittness function.

```
    def get_best_individual(self, population: list):
        """Get the fittest chromosome from the population

        Args:
            population (list): List of chromosomes
        """
        return max(population, key=self.fitness_function)
```

Listing 2.2: Fittest chromosome

The best fitness is the fitness of the best chromosome.

```python
    def get_best_fitness(self, population: list) -> float:
        """Get the best fitness of the population

        Args:
            population (list): List of chromosomes

        Returns:
            float: Best fitness of the population
        """
        best_chromosome = self.get_best_individual(population)
        return self.fitness_function(best_chromosome)
```

Listing 2.3: Best fitness

We randomly selects two parents and mate them to breed children. This process is repeated `number_of_offspring` times.

```python
    def breed_parents(self, population: list) -> list:
        """Breed the parents to get the offsprings

        Args:
            population (list): List of chromosomes

        Returns:
            list: List of chromosomes after breeding
        """
        for _ in range(self.number_of_offsprings):
            parents = random.sample(population, 2)
            child = self.get_offspring(parents[0], parents[1])
            population.append(child)
        return population
```

Listing 2.4: Select two parents and add their offspring to the population

First we select parents from population using `selection_method1` and then the parents breed and a new chromosome enters the population which are then selected using `selection_method2` as survivors.

```python
    def next_generation(self, population: list) -> list:
        """Get the next generation using selection methods and breeding of the parents

        Args:
            population (list): List of chromosomes

        Returns:
            list: List of chromosomes after selection and breeding
        """
        selection_methods = [
            self.fitness_proportionate, self.ranked_selection,
            self.tournament_selection, self.truncation, self.random_selection
        ]

        parents = selection_methods[self.selection_method1](population)
        new_population = self.breed_parents(parents)
        survivors = selection_methods[self.selection_method2](new_population)
        return survivors
```

Listing 2.5: Next generation

The `step` is returns the next generation and the best fitness. `run` runs the evolution process and return the fitness of all chromosomes.

```python
    def step(self, population: list) -> tuple[list, float]:
        """Get population of generation and best fitness of
        the population after selection and breeding of the parents

        Args:
            population (list): List of chromosomes

        Returns:
            tuple[list, float]: List of chromosomes after
            selection and breeding and best fitness of the population
        """
        return self.next_generation(population), self.get_best_fitness(
            population)

    def run(self) -> list:
        """Run the evolution

        Returns:
            list: List of best fitness of the population
        """
        population = self.initial_population()
        fitness_lst = []
        for _ in range(self.number_of_generations):
            population, best_fitness = self.step(population)
            fitness_lst.append(best_fitness)
        return fitness_lst
```

Listing 2.6: Run the evolution

### 2.1.1 Selection Schemes

The selection scheme class caters all the selection methods. It requires two attributes first the population size and the other is fitness function.

```python
    def __init__(self, fitness_function, population_size: int) -> None:
        """Initializes the selection schemes class with the fitness function
        and population size

        Args:
            fitness_function (function): fitness function
            population_size (int): population size

        Description:
            Population size is the number of chromosones to select from the
            population. fitness_function is the fitness function to calculate
            fitness.
        """
        self.population_size = population_size
        self.fitness_function = fitness_function
```

Listing 2.7: Selection Scheme class

This function would select the chromosome according to their fitness values.

```python
    def fitness_proportionate(self, population: list) -> list:
        """Selects chromosomes according to their fitness value (probability)

        Args:
            population (list): population of chromosomes

        Returns:
            list: new population of chromosomes after selection

        Description:
            Each chromosome will be assigned a probability of being selected.
            Such that the probability of a chromosome being selected is
            (fitness of chromosone) / (sum of all fitnesses).
        """
        fitness_lst = list(map(self.fitness_function, population))
        total_fitness = sum(fitness_lst)
        probabilities = list(map(lambda x: x / total_fitness, fitness_lst))
        return random.choices(population, probabilities, k=self.population_size)
```

Listing 2.8: Fitness proportion selection method

The tournament selection method relies on for every chromosome it selects two random chromosome and compete them according to the fitness function and the winner gets appended to the new population.

```python
    def tournament_selection(self, population: list) -> list:
        """Selects chromosomes according to tournament selection

        Args:
            population (list): population of chromosomes

        Returns:
            list: new population of chromosomes after selection

        Description:
            We will select two random chromosomes from the population and
            select the chromosome with the highest fitness and add that
            chromosone to the new population.
        """
        new_population = []
        for _ in range(self.population_size):
            tournament = random.sample(population, 2)
            winner = max(tournament, key=self.fitness_function)
            new_population.append(winner)
        return new_population
```

Listing 2.9: Tournament selection method

Rank based selection method priorotizes the chromosome with high fitness value over low fitness one, and then randomly selects them.

```
    def ranked_selection(self, population: list) -> list:
        """Selects chromosomes according to their rank

        Args:
            population (list): population of chromosomes

        Returns:
            list: new population of chromosomes after selection

        Description:
            The population will be sorted according to the fitness of each
            chromosome. The probability of a chromosome being selected is (rank
            of chromosome) / (total number of chromosomes). The fittest
            invidual will have the highest rank which will be N. The least fit
            will have the lowest rank which will be 1.
        """
        N = len(population)
        sorted_population = sorted(population, key=self.fitness_function)
        probabilities = list(map(lambda x: x / N, range(1, N + 1)))
        return random.choices(sorted_population,
                              probabilities,
                              k=self.population_size)
```

Listing 2.10: Rank based selection method

Random selection methods is just randomly selecting desired number of chromosomes.

```
    def random_selection(self, population: list) -> list:
        """Selects chromosomes randomly from the population

        Args:
            population (list): population of chromosomes

        Returns:
            list: new population of chromosomes after selection
        """
        return random.choices(population, k=self.population_size)
```

Listing 2.11: Random selection method

## 2.2 Travelling Salesman Problem

A chromosome is represented by a list of different cities.

```
    @staticmethod
    def chromosome() -> list:
        """Returns a random route of cities

        Returns:
            list: A random route of cities
        """
        return random.sample(list(range(num_cities)), num_cities)
```

Listing 2.12: Chromosome representation

The fitness function is the inverse of the sum of the distances between the cities in order.

```
    @staticmethod
    def fitness_function(route: list) -> float:
        """Calculates the distance covered in the route

        Args:
            route (list): different routes of cities

        Returns:
            float: distance covered in the route
        """
        N = len(graph) - 1
        distances = list(map(lambda x: graph[route[x]][route[x + 1]],
 range(N)))
        return 1 / sum(distances)
```

Listing 2.13: Fitness function

The mutation is happening by swapping the cities in the route.

```
    @staticmethod
    def mutate(individual: list) -> list:
        """Mutates the route by swapping two cities

        Args:
            individual (list): list of cities

        Returns:
            list: list of cities after mutation
        """
        indexes = random.sample(list(range(len(individual))), 2)
        swap1, swap2 = indexes[0], indexes[1]
        individual[swap1], individual[swap2] = individual[swap2],
    individual[
            swap1]
        return individual
```

Listing 2.14: Mutation

The crossover is done by randomly selecting two cities and then placing all the cities that are intermediate to reach them and placing them in the start of the tour. Rest of the cities comes in the end of the route.

```
    @staticmethod
    def crossover(parent1: list, parent2: list) -> list:
        """Returns a offspring after breeding from two parents

        Args:
            parent1 (list): first parent
            parent2 (list): second parent

        Returns:
            list: offspring after breeding from two parents
        """
        gene1 = int(random.random() * len(parent1))
        gene2 = int(random.random() * len(parent1))

        start_gene = min(gene1, gene2)
        end_gene = max(gene1, gene2)
        child1 = parent1[start_gene:end_gene]
        child2 = [gene for gene in parent2 if gene not in child1]
        child = child1 + child2
        return child
```

Listing 2.15: Crossover

## 2.3 Graph Coloring Problem

A chromosome is represented using the matrix representation for graphs.

```python
    @staticmethod
    def chromosome() -> list:
        """Returns a list of random colors

        Returns:
            list: list of random colors

        Description:
            A random solution would be assign a different color to each vertex
        """
        return random.sample(range(num_nodes), num_nodes)
```

Listing 2.16: Chromosome representation

Fitness function is defined as inverse of the number of unique individuals we have if they are valid other wise it is zero.

```python
    @staticmethod
    def fitness_function(individual: list) -> float:
        """Calculates the fitness of a solution

        Args:
            individual (list): list of colors

        Returns:
            float: fitness of a solution (number of colors used)

        Description:
            If the solution is valid, return the number of colors used. else,
            fitness is zero.
        """
        if is_valid(individual):
            return 1 / len(set(individual))
        return 0.0
```

Listing 2.17: Fitness function

Crossover is done by selecting a random index and placing all the elements after that index the start of the list.

```python
    @staticmethod
    def crossover(parent1: list, parent2: list) -> list:
        """Returns a child solution by crossing over two parents

        Args:
            parent1 (list): first parent
            parent2 (list): second parent

        Returns:
            list: child solution
        """
        position = random.randint(0, num_nodes - 1)
        child = parent1[:position] + parent2[position:]
        return child
```

Listing 2.18: Crossover

Mutation is done by assigning a random color to a random chromosome.

```python
    @staticmethod
    def mutate(individual: list) -> list:
        """Mutates the individual by changing a color

        Args:
            individual (list): list of colors

        Returns:
            list: list of colors after mutation
        """
        position = random.randint(0, num_nodes - 1)
        individual[position] = random.randint(0, num_nodes - 1)
        return individual
```

Listing 2.19: Mutation

## 2.4 Knapsack Problem

Chromosome are just a list of ones and zeros, that refers the selection of that wieght.

```
    @staticmethod
    def chromosome() -> list:
        """Returns a list of randomized binary numbers

        Returns:
            list: list of randomized binary numbers
        """
        return random.choices([0, 1], k=number_of_items)
```

Listing 2.20: Chromosome representation

Fitness function return zero if the total wieght is greater than the threshold otherwise it returns the wieght.

```
    @staticmethod
    def fitness_function(solution: list) -> int:
        """Calculates the fitness of a solution

        Args:
            solution (list): list of binary numbers

        Returns:
            int: fitness of a solution (total profit)

        Description:
            We loop over the solution. If the solution has a 1 in the ith
            position, we add the profit of the ith item to the total profit.
            However, if we exceeed the capacity of the knapsack, the fitness
            is 0.
        """
        total_profit, total_weight = 0, 0
        for binary, profit, weight in zip(solution, profits, weights):
            if binary == 1:
                total_profit += profit
                total_weight += weight
        return total_profit * (total_weight <= threshold)
```

Listing 2.21: Fitness function

Mutation is done by randomly switching the state of the wieght.

```
    @staticmethod
    def mutate(individual: list) -> list:
        """Mutates the individual by flipping a bit

        Args:
            individual (list): list of binary numbers

        Returns:
            list: list of binary numbers after mutation
        """
        index = random.randint(0, len(individual) - 1)
        individual[index] = int(not individual[index])
        return individual
```

Listing 2.22: Mutation

Crossover is done by selecting a random index and placing all the elements after that index the start of the list.

```
    @staticmethod
    def crossover(parent1: list, parent2: list) -> list:
        """Returns a child after breeding from two parents

        Args:
            parent1 (list): first parent
            parent2 (list): second parent

        Returns:
            list: child after breeding from two parents
        """
        geneA = int(random.random() * len(parent1))
        geneB = int(random.random() * len(parent1))
        startGene, endGene = min(geneA, geneB), max(geneA, geneB)
        childP1 = parent1[startGene:endGene]
        childP2 = [gene for gene in parent2 if gene not in childP1]
        return childP1 + childP2
```

Listing 2.23: Crossover

## 2.5 Optimization Class

Optimization class handles the testing of the algorithms. It requires attributes mentioned below along with their default arguments.

```python
    def __init__(
        self,
        problem: Problem,
        population_size: int = 30,
        number_of_offsprings: int = 10,
        number_of_generations: int = 100,
        mutation_rate: float = 0.50,
        number_of_iterations: int = 10,
        selection_case: tuple = (0, 0)) -> None:
        """Initializes the Optimization class with the given parameters

        Args:
            problem (Problem): Problem class from Evolution.problem
            population_size (int, optional): population size. Defaults to 30.
            number_of_offsprings (int, optional): number of offsprings. Defaults to 10.
            number_of_generations (int, optional): number of generations. Defaults to 100.
            mutation_rate (float, optional): mutation rate. Defaults to 0.50.
            number_of_iterations (int, optional): number of iterations. Defaults to 10.
            selection_case (tuple, optional): selection case. Defaults to (0, 0).
        """
        self.population_size = population_size
        self.number_of_generations = number_of_generations
        self.number_of_iterations = number_of_iterations

        self.number_of_offsprings = number_of_offsprings
        self.problem = problem

        self.mutation_rate = mutation_rate
        self.selection_case = selection_case
```

Listing 2.24: Parameters

This function would evolve the generation according to the values provided.

```python
    def evolve(self) -> Evolution:
        """Returns an Evolution object

        Returns:
            Evolution: Evolution object
        """
        return Evolution(problem=self.problem,
                         mutation_rate=self.mutation_rate,
                         population_size=self.population_size,
                         selection_method1=self.selection_case[0],
                         selection_method2=self.selection_case[1],
                         number_of_generations=self.number_of_generations,
                         number_of_offsprings=self.number_of_offsprings)
```

Listing 2.25: Evolution

The title for the plot is generated by the following function.

```python
    def get_title(self, fitness_type: str) -> str:
        """Returns the title of the plot

        Args:
            fitness_type (str): fitness type (BSF or ASF)

        Returns:
            str: title of the plot
        """
        selection_cases = ["FPS", "RBS", "Tournament", "Truncation", "Random"]
        title = f"{fitness_type} - {self.problem.__name__} - "
        title += f"{selection_cases[self.selection_case[0]]} &"
        title += f"{selection_cases[self.selection_case[1]]}\n"
        title += f"Pop Size: {self.population_size}; "
        title += f"Num Offsprings: {self.number_of_offsprings}; "
        title += f"Mutation Rate: {self.mutation_rate}"

        return title
```

Listing 2.26: Getter for title

The name of the file is generated by the following function.

```
    def get_filename(self, fitness_type: str) -> str:
        """Returns the filename of the plot to be saved

        Args:
            fitness_type (str): fitness type (BSF or ASF)

        Returns:
            str: filename of the plot to be saved
        """
        filename = f"{fitness_type}_{self.problem.__name__}"
        filename += f"_{self.selection_case[0]}"
        filename += f"_{self.selection_case[1]}"
        filename += f"_{self.population_size}"
        filename += f"_{self.number_of_offsprings}"

        return filename
```

Listing 2.27: Getter for filename

The best so far is ploted with the following function.

```
    def plot_BSF(self) -> None:
        """Plots the best fitness of the evolution"""
        evolution = self.evolve()
        fitness_lst = evolution.run()

        if self.problem.inverse_fitness:
            fitness_lst = [(1 / fitness) if fitness != 0 else 100
                           for fitness in fitness_lst]

        print("Initial fitness: ", fitness_lst[0])
        print("Final fitness: ", fitness_lst[-1])

        x = list(range(len(fitness_lst)))
        y = fitness_lst

        title = self.get_title("Best Fitness")

        plt.title(title)
        plt.plot(x, y)

        filename = self.get_filename("BSF")
        plt.savefig("Analysis/" + filename + ".png")
        plt.close()
```

Listing 2.28: Plotter for best so far

The average so far is ploted with the following function.

```
    def plot_ASF(self) -> None:
        """Plots the average fitness of the evolution"""
        runs: dict[str, list] = dict()

        for iteration in range(self.number_of_iterations):
            runs["Run #" + str(iteration + 1)] = self.evolve().run()

        df = pd.DataFrame(runs)

        def invert(val):
            if val == 0:
                return 100
            return 1 / val

        df["Average"] = df.mean(axis=1)

        print("Best Average Fitness: ")

        df.index.name = "Generation #"

        title = self.get_title("Average Fitness")

        plt.title(title)
        filename = self.get_filename("ASF")

        if self.problem.inverse_fitness:
            df["Average"] = df["Average"].apply(invert)
            print(df["Average"].min())
        else:
            print(df["Average"].max())

        plt.plot(df["Average"])
        plt.savefig("Analysis/" + filename + ".png")
        plt.close()
```

Listing 2.29: Plotter for average so far

# Chapter 3

# Analysis

We tried 10 different combinations of the 5 selection schemes that are,

1. Fitness Proportional Selection
2. Rank based Selection
3. Binary Tournament
4. Truncation
5. Random

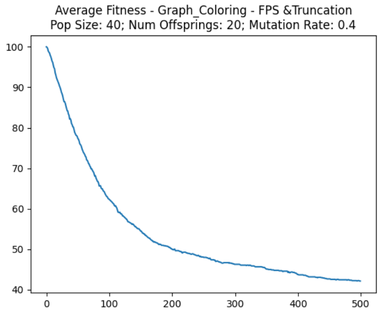
and those combinations are,

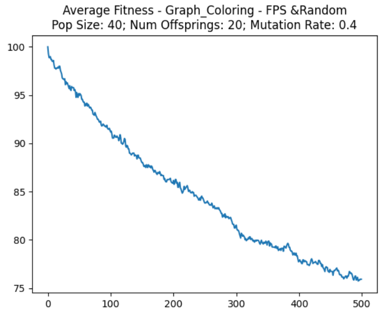
1. FPS and Random
2. Binary Tournament and Truncation
3. Truncation and Truncation
4. Random and Random
5. FPS and Truncation
6. RBS and Binary Tournament
7. Random and Truncation
8. Binary Tournament and FPS
9. Binary Tournament and RBS
10. Truncation and FPS

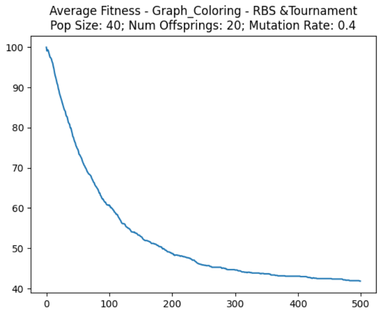
## 3.1 Graph Coloring Problem

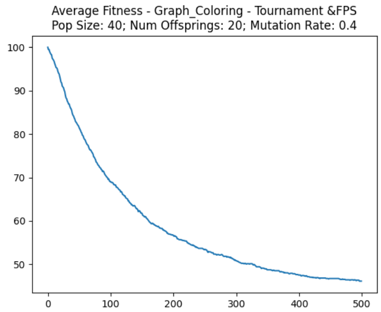
### 3.1.1 Average So Far

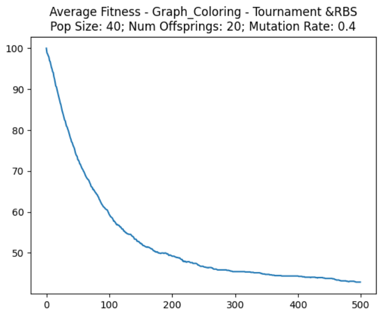
Again we found 2 best selection scheme combinations, which were Tournament & Truncation, and Truncation and Truncation, with their fitness being 40. The worst selection scheme combination was Random & Random with the fitness being 100, we also saw that FPS & Random did not give us good results too.

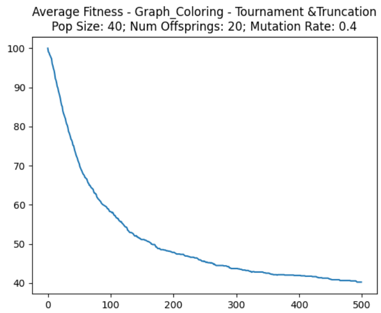

Average Fitness - Graph_Coloring - FPS &Truncation
Pop Size: 40; Num Offsprings: 20; Mutation Rate: 0.4

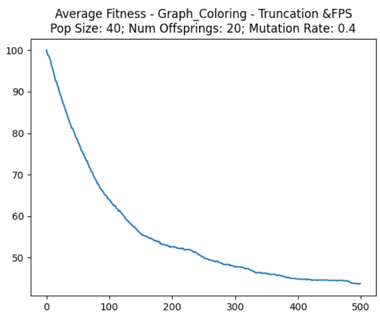

Average Fitness - Graph_Coloring - FPS &Random
Pop Size: 40; Num Offsprings: 20; Mutation Rate: 0.4

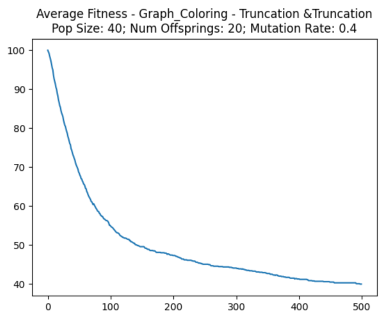

Average Fitness - Graph_Coloring - RBS &Tournament
Pop Size: 40; Num Offsprings: 20; Mutation Rate: 0.4

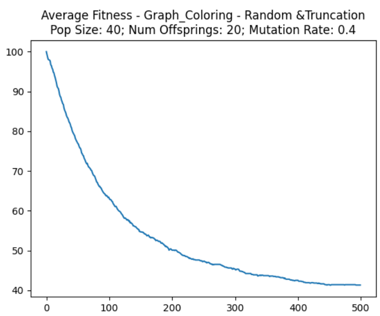

Average Fitness - Graph_Coloring - Tournament &FPS
Pop Size: 40; Num Offsprings: 20; Mutation Rate: 0.4

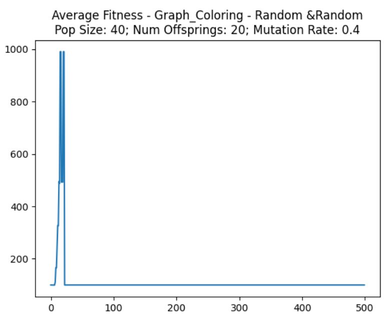

Average Fitness - Graph_Coloring - Tournament &RBS
Pop Size: 40; Num Offsprings: 20; Mutation Rate: 0.4


Average Fitness - Graph_Coloring - Tournament &Truncation
Pop Size: 40; Num Offsprings: 20; Mutation Rate: 0.4


Average Fitness - Graph_Coloring - Truncation &FPS
Pop Size: 40; Num Offsprings: 20; Mutation Rate: 0.4


Average Fitness - Graph_Coloring - Truncation &Truncation
Pop Size: 40; Num Offsprings: 20; Mutation Rate: 0.4

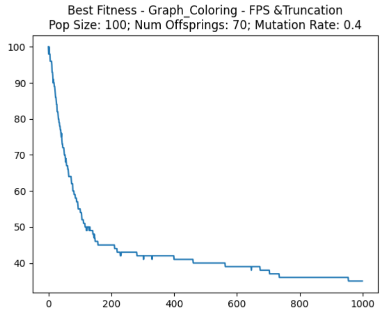

Average Fitness - Graph_Coloring - Random &Truncation
Pop Size: 40; Num Offsprings: 20; Mutation Rate: 0.4

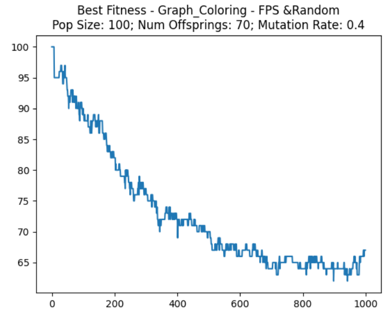

Average Fitness - Graph_Coloring - Random &Random
Pop Size: 40; Num Offsprings: 20; Mutation Rate: 0.4

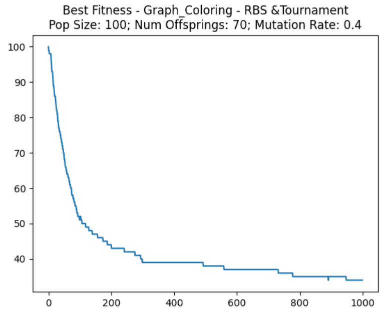
### 3.1.2 Best So Far

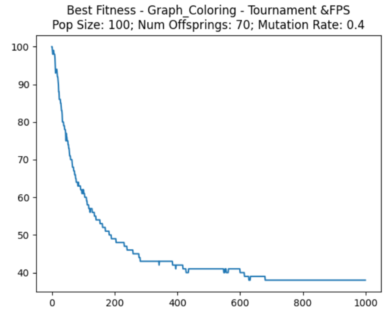
We found 2 of the schemes to be best among our selection scheme combinations, Random & Truncation and RBS & Tournament, both of these combinations gave us a fitness level of around 34. Whereas the worst selection scheme combination was Random and Random in which the fitness that we received was 100.

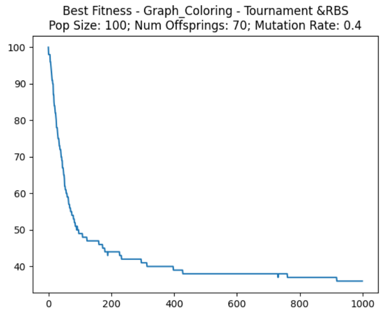
The overall majority of the fitnesses were in near 36-35, these proved to be quite impressive as compared to our initial fitness values.

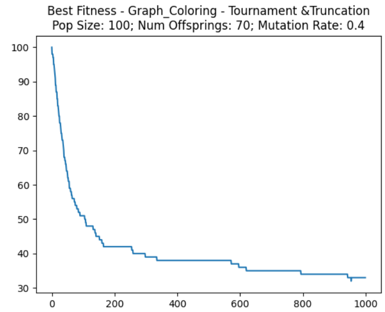

Best Fitness - Graph_Coloring - FPS &Truncation
Pop Size: 100; Num Offsprings: 70; Mutation Rate: 0.4
100 90 80 70 60 50 40
0 200 400 600 800 1000

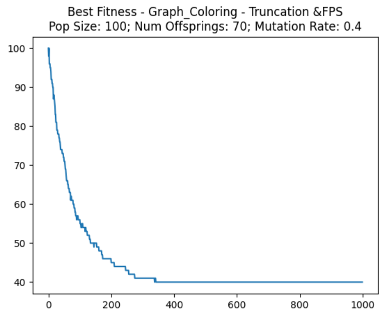

Best Fitness - Graph_Coloring - FPS &Random
Pop Size: 100; Num Offsprings: 70; Mutation Rate: 0.4
100 95 90 85 80 75 70 65
0 200 400 600 800 1000

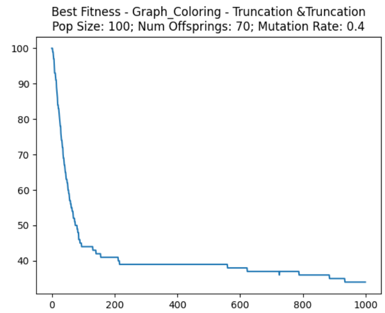

Best Fitness - Graph_Coloring - RBS &Tournament
Pop Size: 100; Num Offsprings: 70; Mutation Rate: 0.4
100 90 80 70 60 50 40
0 200 400 600 800 1000

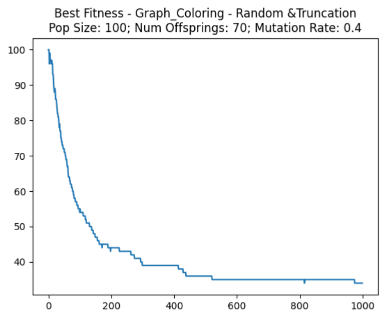

Best Fitness - Graph_Coloring - Tournament &FPS
Pop Size: 100; Num Offsprings: 70; Mutation Rate: 0.4
100 90 80 70 60 50 40
0 200 400 600 800 1000

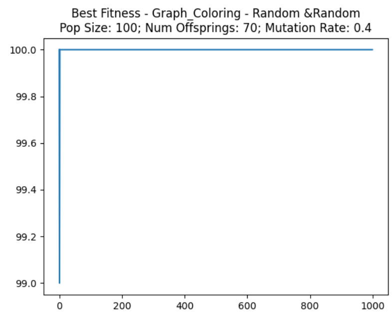

Best Fitness - Graph_Coloring - Tournament &RBS
Pop Size: 100; Num Offsprings: 70; Mutation Rate: 0.4
100 90 80 70 60 50 40
0 200 400 600 800 1000


Best Fitness - Graph_Coloring - Tournament &Truncation
Pop Size: 100; Num Offsprings: 70; Mutation Rate: 0.4
100 90 80 70 60 50 40 30
0 200 400 600 800 1000


Best Fitness - Graph_Coloring - Truncation &FPS
Pop Size: 100; Num Offsprings: 70; Mutation Rate: 0.4
100 90 80 70 60 50 40
0 200 400 600 800 1000


Best Fitness - Graph_Coloring - Truncation &Truncation
Pop Size: 100; Num Offsprings: 70; Mutation Rate: 0.4
100 90 80 70 60 50 40
0 200 400 600 800 1000


Best Fitness - Graph_Coloring - Random &Truncation
Pop Size: 100; Num Offsprings: 70; Mutation Rate: 0.4
100 90 80 70 60 50 40
0 200 400 600 800 1000

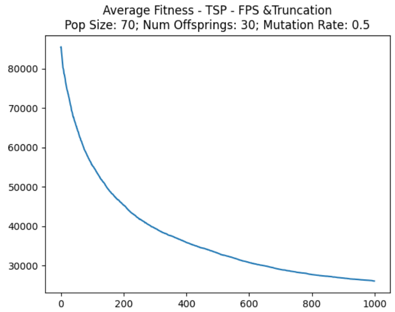

Best Fitness - Graph_Coloring - Random &Random
Pop Size: 100; Num Offsprings: 70; Mutation Rate: 0.4
100.0 99.8 99.6 99.4 99.2 99.0
0 200 400 600 800 1000

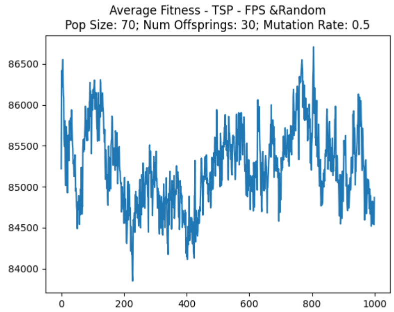
## 3.2 Travelling Salesman Problem

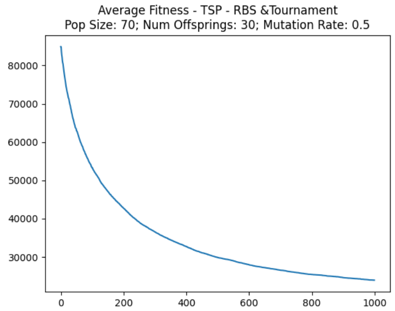
### 3.2.1 Average So Far

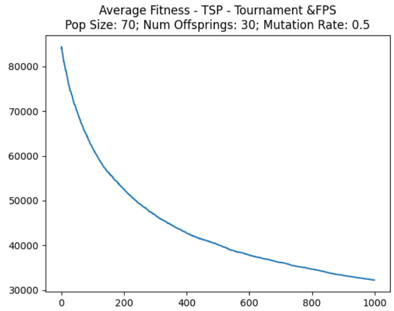
The best results were given by FPS & Truncation, and RBS & Tournament, with both of the selected combinations having a greater depth in their trend. The worst results were given by FPS & Random, and Random & Random, which had chaotic results with almost no convergence

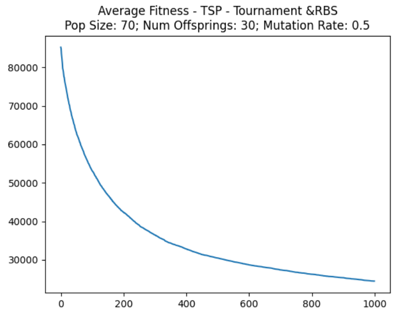
If we look at the plots of ASF on various selection schemes we again see that most of the combinations have achieved impressive fitness results, and are similar to each other as compared to their initial fitness scores.

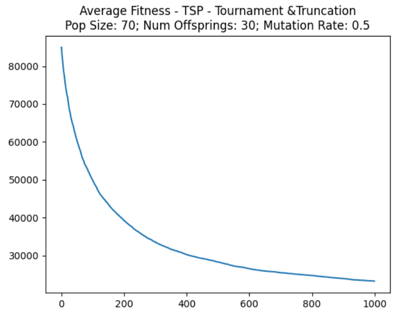

Average Fitness - TSP - FPS &Truncation
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5

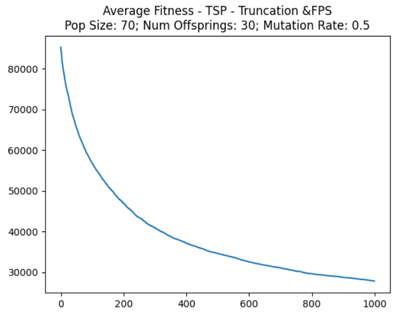

Average Fitness - TSP - FPS &Random
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5

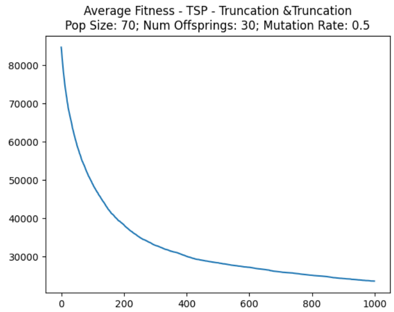

Average Fitness - TSP - RBS &Tournament
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5

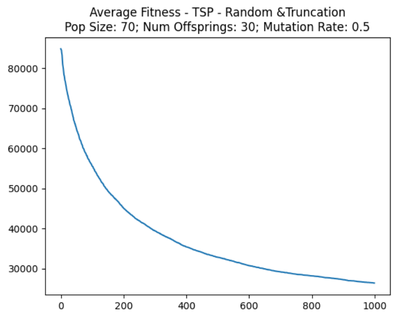

Average Fitness - TSP - Tournament &FPS
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5

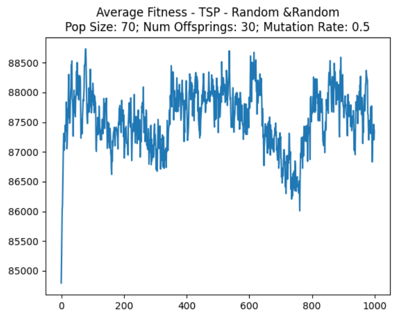

Average Fitness - TSP - Tournament &RBS
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5


Average Fitness - TSP - Tournament &Truncation
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5


Average Fitness - TSP - Truncation &FPS
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5


Average Fitness - TSP - Truncation &Truncation
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5

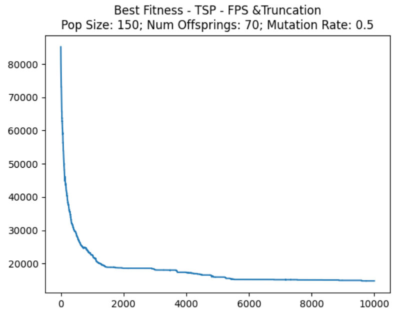

Average Fitness - TSP - Random &Truncation
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5

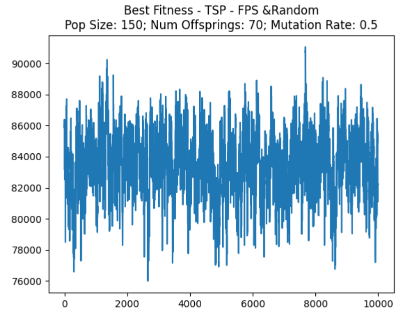

Average Fitness - TSP - Random &Random
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5

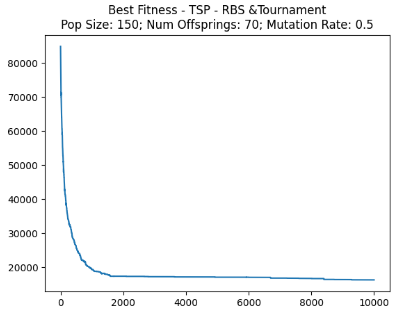
### 3.2.2 Best So Far

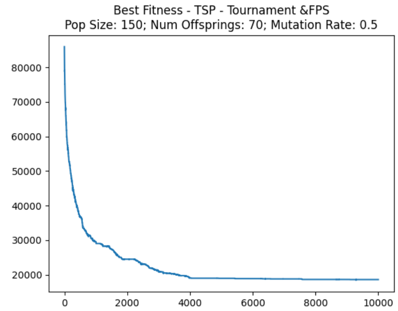
The best results we got from BSF were from the selection schemes FBS & Truncation, and Tournament and RBS, where both have pretty similar and excellent results. The worst case results were from Random & Random, FBS & Random.

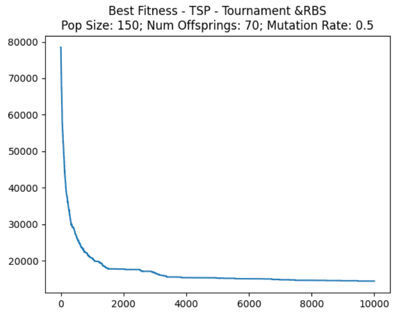
Overall if we look at the trends of the plots of our BSF plots we can see that most of our selection scheme combinations have performed.

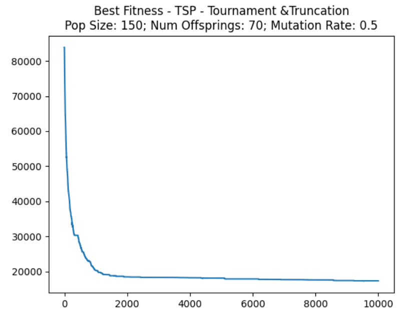

Best Fitness - TSP - FPS &Truncation
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.5

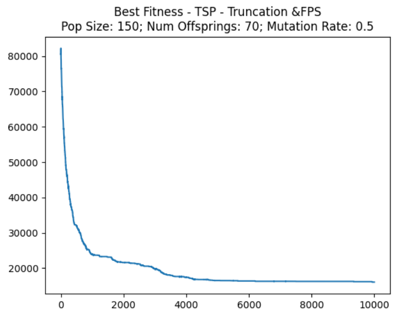

Best Fitness - TSP - FPS &Random
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.5

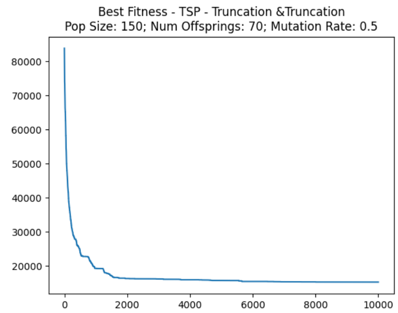

Best Fitness - TSP - RBS &Tournament
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.5

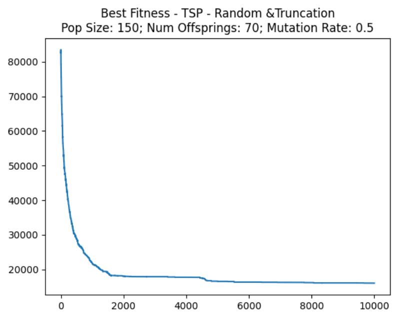

Best Fitness - TSP - Tournament &FPS
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.5

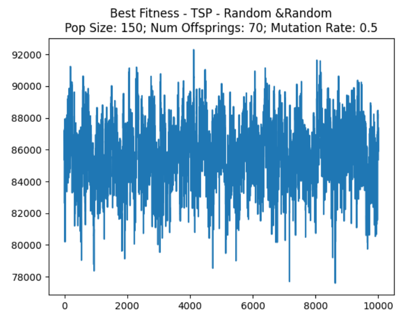

Best Fitness - TSP - Tournament &RBS
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.5


Best Fitness - TSP - Tournament &Truncation
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.5


Best Fitness - TSP - Truncation &FPS
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.5


Best Fitness - TSP - Truncation &Truncation
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.5

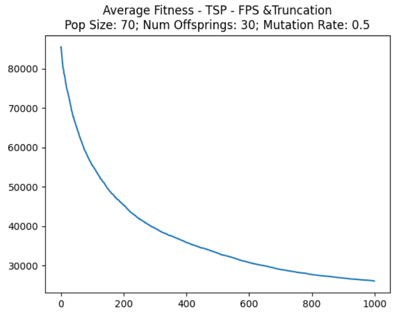

Best Fitness - TSP - Random &Truncation
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.5

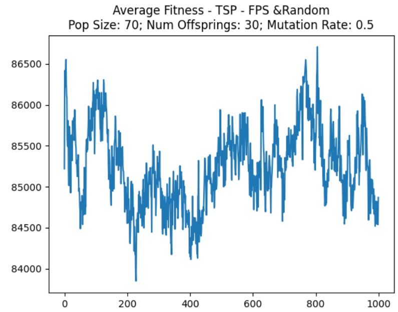

Best Fitness - TSP - Random &Random
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.5

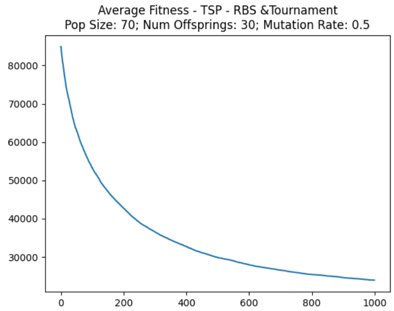
## 3.3 Knapsack Problem

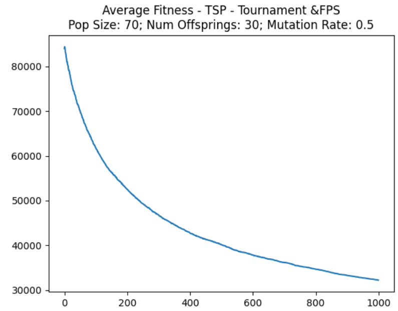
### 3.3.1 Average So Far

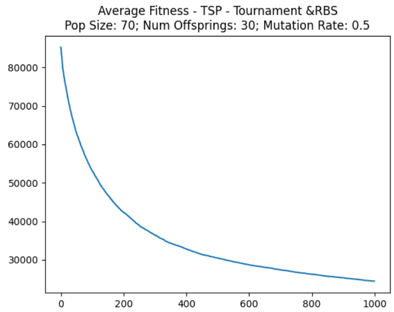
Majority of the results from the selection scheme combinations were quite similar, with very few differences from each other, however, Truncation & Truncation and Tournament & FPS were the best, with their fitness being 9758. On the other hand, the worst selection scheme combination was of Random & Random, moreover FPS & Random did not have good results as compared to our other fitnesses achieved.

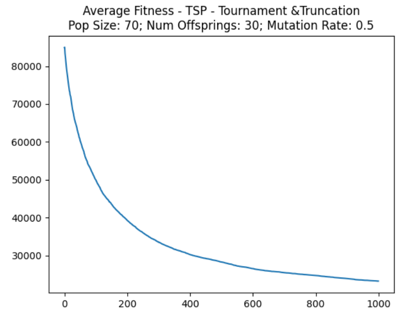

Average Fitness - TSP - FPS &Truncation
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5

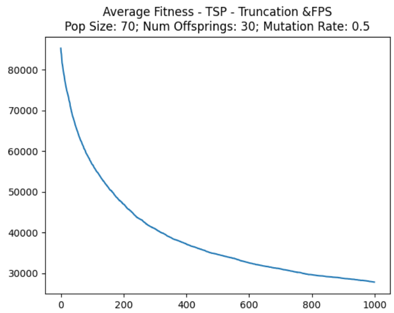

Average Fitness - TSP - FPS &Random
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5

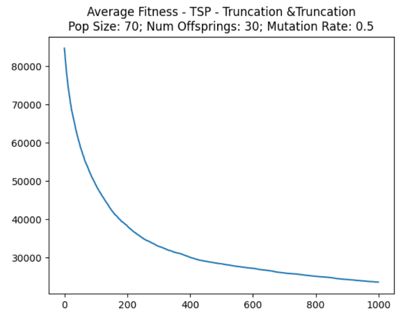

Average Fitness - TSP - RBS &Tournament
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5

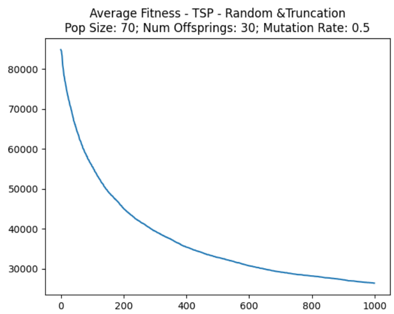

Average Fitness - TSP - Tournament &FPS
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5

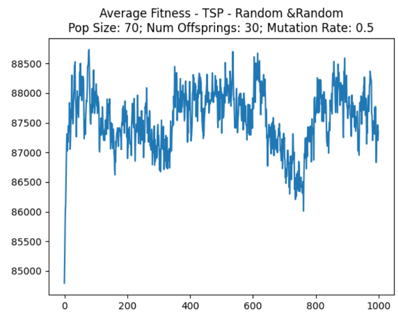

Average Fitness - TSP - Tournament &RBS
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5


Average Fitness - TSP - Tournament &Truncation
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5


Average Fitness - TSP - Truncation &FPS
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5

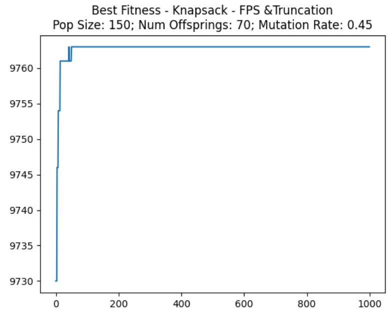

Average Fitness - TSP - Truncation &Truncation
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5

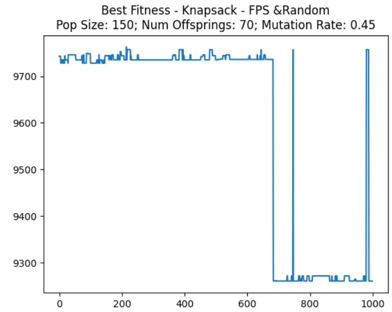

Average Fitness - TSP - Random &Truncation
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5

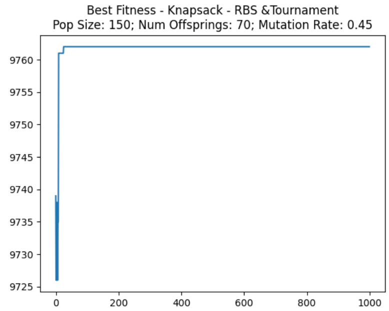

Average Fitness - TSP - Random &Random
Pop Size: 70; Num Offsprings: 30; Mutation Rate: 0.5

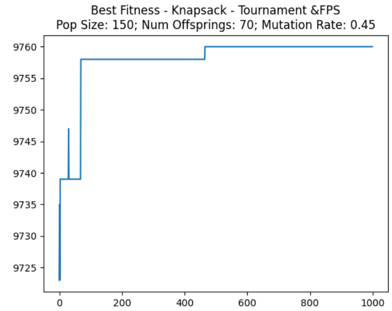
### 3.3.2 Best So Far

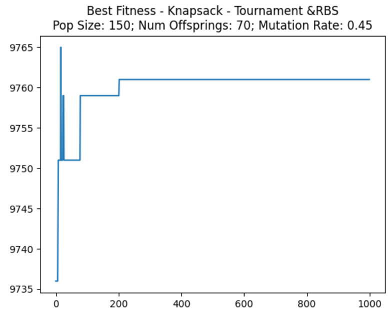
The best selection combination that we found was RBS & Tournament, with the fitness value being 9763. The worst selection scheme combination was Random & Random, which had no convergence , with the fitness value fluctuating between 4000-2000 throughout multiple generations

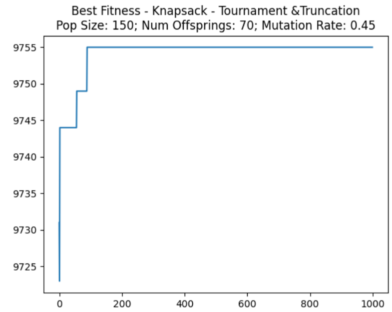

Best Fitness - Knapsack - FPS &Truncation
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.45

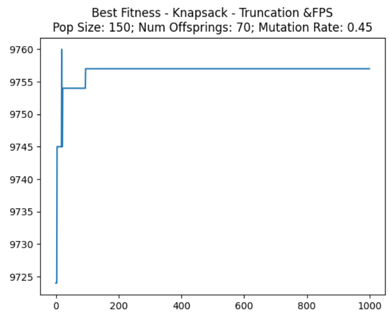

Best Fitness - Knapsack - FPS &Random
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.45

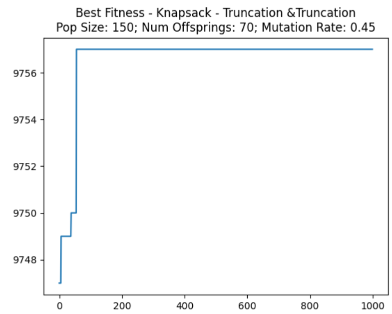

Best Fitness - Knapsack - RBS &Tournament
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.45

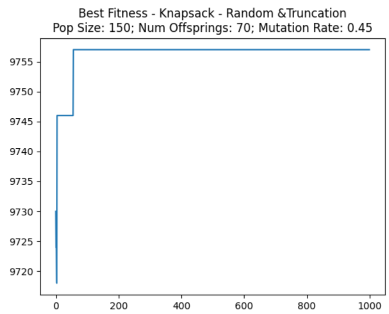

Best Fitness - Knapsack - Tournament &FPS
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.45

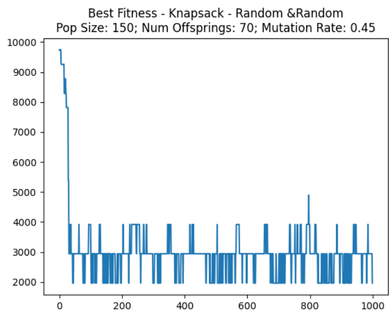

Best Fitness - Knapsack - Tournament &RBS
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.45


Best Fitness - Knapsack - Tournament &Truncation
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.45


Best Fitness - Knapsack - Truncation &FPS
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.45


Best Fitness - Knapsack - Truncation &Truncation
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.45


Best Fitness - Knapsack - Random &Truncation
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.45


Best Fitness - Knapsack - Random &Random
Pop Size: 150; Num Offsprings: 70; Mutation Rate: 0.45