Evolutionary Algorithms

Ali Asghar Kerai - ak06857

Muhammed Jazzel Mehmood - mm06886

February 12, 2024

CS451 - Computational Intelligence

Assignment 1

Contents

1 Code **1**

1.1 Evolutionary Algorithm . . . 1

1.1.1 Selection Schemes . . . 8

1.2 Travelling Salesman Problem . . . 11

1.2.1 Chromosome representation . . . 11

1.2.2 Fitness Function . . . 12

1.2.3 Automated Code for Plotting . . . 12

1.2.4 Results and Analysis . . . 16

1.2.5 Optimal Results and Conclusion . . . 19

1.3 Job Shop Scheduling Problem . . . 20

1.3.1 Chromosome representation . . . 20

1.3.2 Fitness Function . . . 21

1.3.3 Automated Code for Plotting . . . 22

1.3.4 Results and Analysis for dataset abz5 . . . 27

1.3.5 Results and Analysis for dataset abz6 . . . 29

1.3.6 Results and Analysis for dataset abz7 . . . 32

1.3.7 Gantt Charts for optimal solutions for all datasets . . . 34

1.3.8 Optimal Results and Conclusion . . . 36

1.4 Mona Lisa . . . 37

1.4.1 Chromosome representation . . . 37

1.4.2 Fitness Function . . . 38

1.4.3 Modification on EA . . . 39

1.4.4 Code for generating image . . . 40

1.4.5 Images at different iterations for Mona lisa . . . 41

1.4.6 Images at different iterations for Tom and Jerry . . . 43

1.4.7 Conclusion 45

1. Code

Complete code can be found at: `https://github.com/Jazzel/HU-CI-Assignment-1`

Instructions for the code can be found at: `https://jazzel.github.io/HU-CI-Assignment-1/`

1.1 Evolutionary Algorithm

```python
class EvolutionaryAlgorithm(SelectionScheme):
    """
    A class representing an evolutionary algorithm.

    Attributes:
    - population_size (int): The size of the population.
    - no_of_offsprings (int): The nunber of the offsprings
    - mutation_rate (float): The rate of mutation.

    Methods:
    - initialize_population(): Initializes the population with random individuals.
    - evaluate_fitness(): Evaluates the fitness of each individual in the population.
    - selection(): Performs selection to choose parents for reproduction.
    - crossover(): Performs crossover to create offspring.
    - mutation(): Performs mutation on the offspring.
    - replace_population(): Replaces the current population with the offspring.
    - run(): Runs the evolutionary algorithm.
    """

    def __init__(
        self,
        parent_selection_scheme: int = 1,
        survival_selection_scheme: int = 1,
        population_size: int = 30,
        no_of_generations: int = 50,
```

```
        no_of_offsprings: int = 10,
        mutation_rate: float = 0.5,
        no_of_iterations: int = 10,
    ) -> None:
        self.population_size = population_size
        self.no_of_offsprings = no_of_offsprings
        self.no_of_generations = no_of_generations
        self.mutation_rate = mutation_rate
        self.no_of_iterations = no_of_iterations
        self.parent_selection_scheme = parent_selection_scheme
        self.survivor_selection_scheme = survival_selection_scheme
};
```

Listing 1.1: Initializing the evolution class with attributes

```
    def initialize_population(self) -> None:
        # Initialize the population with random individuals
        self.population = {i: self.chromosome() for i in range(self.population_size)}
        return self.population
```

Listing 1.2: Initializing population

```
    # Parents selection scheme
    def parent_selection(self) -> None:
        """
        Performs selection to choose parents for reproduction of offsprings.
        """
        if self.parent_selection_scheme == 1:
            self.parents = self.fitness_proportionate_selection(self.no_of_offsprings
    )
        elif self.parent_selection_scheme == 2:
            self.parents = self.rank_based_selection(self.no_of_offsprings)
        elif self.parent_selection_scheme == 3:
            self.parents = self.binary_tournament_selection(self.no_of_offsprings)
```

```python
        elif self.parent_selection_scheme == 4:
            self.parents = self.truncation_selection(self.no_of_offsprings)
        elif self.parent_selection_scheme == 5:
            self.parents = self.random_selection(self.no_of_offsprings)
        else:
            print("Invalid selection scheme")
```

Listing 1.3: Parents Selection

```python
def crossover(self) -> None:
    """
    Takes two parents and produces two offsprings.
    """
        self.offsprings = {}
        # helper function
        def fillRest(arr, offspring) -> None:
            remaining_cities = []
            for i in range(end, length + start + end):
                index = i % length
                remaining = arr[index]
                if remaining not in offspring:
                    remaining_cities.append(remaining)
            remaining_cities.reverse()

            for i in range(end, length + start):
                index = i % length
                offspring[index % length] = remaining_cities.pop()

        # Perform crossover to create offspring. Parts of chromosomes of both parents are merged in such a way that two unique offspings are formed.
        d_index = len(self.population)
        for index in range(0, len(self.parents), 2):

            chromosome_parent1 = self.population[self.parents[index]]
            chromosome_parent2 = self.population[self.parents[index + 1]]
```

```python

            length = len(chromosome_parent1)
            start, end = sorted(random.sample(range(length), 2))
            offspring1 = [None] * length
            offspring2 = [None] * length

            offspring1[start:end] = chromosome_parent1[start:end]
            offspring2[start:end] = chromosome_parent2[start:end]

            fillRest(chromosome_parent2, offspring1)
            fillRest(chromosome_parent1, offspring2)

            self.offsprings[d_index] = offspring1
            self.offsprings[d_index + 1] = offspring2
            d_index += 2
```

Listing 1.4: Crossover

```python
    def mutation(self) -> None:
        """
        Genes are swapped to mutate chromosome according to the mutation rate.
        """
        if random.random() < self.mutation_rate:
            for individual in self.offsprings.keys():
                index1 = random.randint(0, len(self.offsprings[individual]) - 1)
                index2 = random.randint(0, len(self.offsprings[individual]) - 1)
                (
                    self.offsprings[individual][index1],
                    self.offsprings[individual][index2],
                ) = (
                    self.offsprings[individual][index2],
                    self.offsprings[individual][index1],
                )
```

Listing 1.5: Mutation

```python
    def survivor_selection(self) -> None:
        """
        - Performs selection to choose survivors in each generation.
        - The population is then updated accordingly.
        """
        survivers = []
        if self.survivor_selection_scheme == 1:
            survivers = self.fitness_proportionate_selection(self.population_size -
 1)
        elif self.survivor_selection_scheme == 2:
            survivers = self.rank_based_selection(self.population_size - 1)
        elif self.survivor_selection_scheme == 3:
            survivers = self.binary_tournament_selection(self.population_size - 1)
        elif self.survivor_selection_scheme == 4:
            survivers = self.truncation_selection(self.population_size - 1)
        elif self.survivor_selection_scheme == 5:
            survivers = self.random_selection(self.population_size - 1)
        else:
            print("Invalid selection scheme")

        updatedPopulation = []
        for index in survivers:
            updatedPopulation.append(self.population[index])
        self.population = {
            i: updatedPopulation[i] for i in range(len(updatedPopulation))
        }
```

Listing 1.6: Survivor Selection and updating population

```python
    def run(self):
        # Run the evolutionary algorithm
        iteration = 1
        fitttest_individual = []
        fitnesses = []
```

```python
        self.initialize_population()
        while iteration <= self.no_of_generations:
            # Computing fitness of the population
            self.fitness_dictionary = self.compute_population_fitness(self.population)
            # Using the Elitist approach and ensuring that the fittest individual is transferred to the next generation without being mutated.
            fittest_individual = self.population[
                min(self.fitness_dictionary, key=self.fitness_dictionary.get)
            ]
            self.parent_selection()
            self.crossover()
            self.mutation()
            # Adding offsprings to population
            updatedPopulation = list(self.population.values()) + list(
                self.offsprings.values()
            )
            self.population = {
                i: updatedPopulation[i] for i in range(0, len(updatedPopulation))
            }
            # Computing fitness of population
            self.fitness_dictionary = self.compute_population_fitness(self.population)
            # Adding fittest individual from elitist approach to population.
            self.population[len(self.population)] = fittest_individual

            self.fitness_dictionary = self.compute_population_fitness(self.population)

            avg_fitness = round(sum(self.fitness_dictionary.values()) / 30, 2)
            fitnesses.append(avg_fitness)
            print(
                "avg_fitness:",
                avg_fitness,
                "fittest:",
```

```
                min(self.fitness_dictionary.values()),
            )

            iteration += 1
        print(min(fitnesses))
        print(max(fitnesses))
```

Listing 1.7: Run the evolution

1.1.1 Selection Schemes

```python
def fitness_proportionate_selection(self, selection_size) -> list:
"""
Each chromosome will be assigned a probability of being selected according to
their fitness.
"""
    total_fitness = sum( 1/individual_fitness for individual_fitness in self.
fitness_dictionary.values())
    probabilities = [
        ((1/fitness) / total_fitness) for fitness in self.fitness_dictionary.
values()
    ]
    return random.choices(
        list(self.fitness_dictionary.keys()),
        weights=probabilities,
        k=selection_size,
    )
```

Listing 1.8: Fitness Proportionate Selection

```python
def rank_based_selection(self, selection_size) -> list:
"""
Each chromosome will be assigned a probability of being selected according to
their rank. ranks are assigned based on fitness.
"""
    temp_sorted = dict(
        sorted(
            self.fitness_dictionary.items(), key=lambda item: item[1], reverse=
True
        )
    )

    ranks = [i for i in range(1, len(temp_sorted) + 1)]
    probabilities = [rank / sum(ranks) for rank in ranks]
```

```python

        return random.choices(
            list(temp_sorted.keys()), weights=probabilities, k=selection_size
        )
```

Listing 1.9: Rank Based Selection

```python
        def binary_tournament_selection(self, selection_size) -> list:
    """
    Two chromosomes are selected from the population and the fittest of them is preffered for selection.
    """
        tournament_selected = []
        for i in range(selection_size):
            parent1, parent2 = random.choices(list(self.fitness_dictionary.keys()), k=2)
            if self.fitness_dictionary[parent1] > self.fitness_dictionary[parent2]:
                tournament_selected.append(parent1)
            else:
                tournament_selected.append(parent2)
        return tournament_selected
```

Listing 1.10: Binary Tournament selection

```python
    def truncation_selection(self, selection_size) -> list:
    """
    Chromosomes are selected based directly on their fitness.
    """
        trunc = dict(sorted(self.fitness_dictionary.items(), key=lambda item: item[1]))
        return list(trunc.keys())[:selection_size]
```

Listing 1.11: Truncation Selection

```
    def random_selection(self, selection_size) -> list:
    """
    Chormosomes are selected randomly.
    """
        return random.choices(list(self.fitness_dictionary.keys()), k=selection_size)
```

Listing 1.12: Random selection

1.2 Travelling Salesman Problem

1.2.1 Chromosome representation

We have mapped every destination with a unique number. The distance of each destination with every other destination is computed and stored in a dictionary. A chromosome is initialized with random order of all destinations to complete the path.

```
    def read_file(self):
    # Coding for reading file and getting appropriate data
        with open(self.filename, "r") as f:
            self.name = f.readline().strip()
            self.comment1 = f.readline().strip()
            self.comment2 = f.readline().strip()
            self.type = f.readline().strip()
            self.dimension = int(f.readline().strip().split()[2])
            self.edgeWeightType = f.readline().strip()
            self.nodeCoordSelection = f.readline().strip()

            line = f.readline().strip()
            while line != "EOF":
                self.node_coords.append(list(map(float, line.split())))
                line = f.readline().strip()
        # Computing distance of each destination with another.
        self.distance_matrix = [
            [0 for _ in range(self.dimension)] for _ in range(self.dimension)
        ]
        for i in range(self.dimension):
            for j in range(self.dimension):
                self.distance_matrix[i][j] = self.euclidean_distance(
                    self.node_coords[i], self.node_coords[j]
                )
    # Helper function for calculating euclidean distance
    def euclidean_distance(self, p1, p2):
        return ((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2) ** 0.5
```

```python
    # shuffling destinations randomly to make a chromosome
    def chromosome(self) -> list:
        arr = [i for i in range(self.dimension)]
        random.shuffle(arr)
        return arr
```

Listing 1.13: Chromosome represntation

1.2.2 Fitness Function

We compute total distance of the path that is represented by the chromosome and store it in a dictionary which contains fitness of all individuals in the population.

```python
    def evaluate_fitness(self, chromosome) -> float:
    # Calculating distance of the path made by the order of the destination in the chromosomes
        fitness = 0
        for i in range(self.dimension - 1):
            fitness += self.distance_matrix[chromosome[i]][chromosome[i + 1]]
        fitness += self.distance_matrix[chromosome[self.dimension - 1]][chromosome[0]]
        return fitness

    def compute_population_fitness(self, population: dict) -> dict:
    #Computing fitness of all the individuals in the population
        fitness_dictionary = {}
        for individual, chromosome in population.items():
            fitness_dictionary[individual] = self.evaluate_fitness(chromosome)
        return fitness_dictionary
```

Listing 1.14: Fitness Function

1.2.3 Automated Code for Plotting

```python
import json
import pandas as pd
```

```
import matplotlib.pyplot as plt

files = [
    "tsp_qa194.tsp_p1_s2.json",
    "tsp_qa194.tsp_p1_s4.json",
    "tsp_qa194.tsp_p1_s5.json",
    "tsp_qa194.tsp_p2_s4.json",
    "tsp_qa194.tsp_p3_s4.json",
    "tsp_qa194.tsp_p4_s4.json",
    "tsp_qa194.tsp_p5_s5.json",
]

for file in files:

    algo = file.split(".")[1]
    parent = algo.split("_")[1]
    surviver = algo.split("_")[2]

    if parent == "p1":
        parent = "Fitness-Proportional"
    elif parent == "p2":
        parent = "Rank-Based"
    elif parent == "p3":
        parent = "Tournament"
    elif parent == "p4":
        parent = "Truncation"
    elif parent == "p5":
        parent = "Random"

    if surviver == "s1":
        surviver = "Fitness-Proportional"
    elif surviver == "s2":
        surviver = "Rank-Based"
    elif surviver == "s3":
        surviver = "Tournament"
```

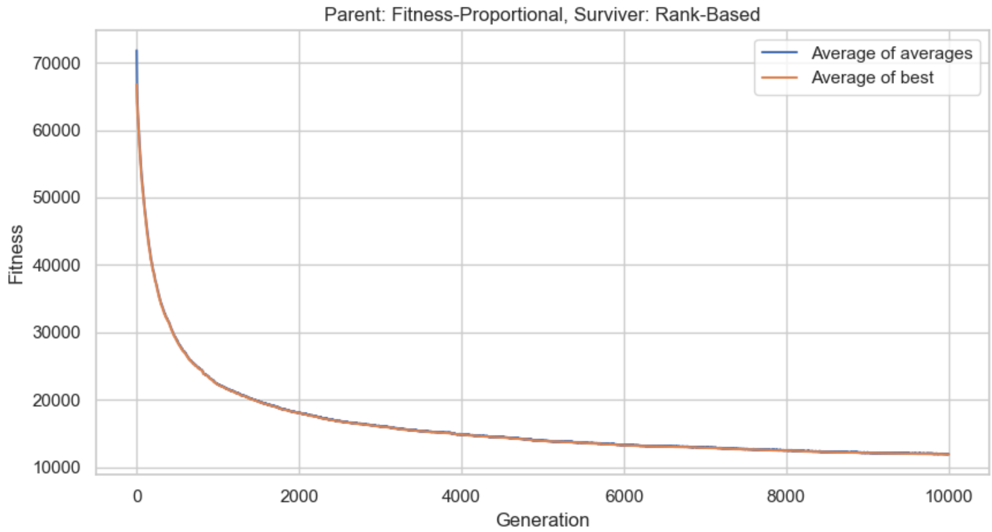
```python
    elif surviver == "s4":
        surviver = "Truncation"
    elif surviver == "s5":
        surviver = "Random"

    df = pd.DataFrame()
    results = pd.DataFrame()
    with open(file, "r") as f:
        data = json.load(f)

        rows = []

        for iteration, generation_data in data.items():
            iteration_number = int(iteration)
            for generation_info in generation_data:
                generation_number = generation_info["generation"]
                average = generation_info["average"]
                best = generation_info["best"]
                best_chromosome = generation_info["best_chromosome"]

                row = {
                    "iteration": iteration_number,
                    "generation": generation_number,
                    "average": average,
                    "best": best,
                    "best_chromosome": best_chromosome,
                }
                rows.append(row)

        df = pd.DataFrame(rows)

        pivot_df = df.pivot_table(
            index="generation", columns=["iteration"], values=["average", "best"]
        )

```

```
        avg_of_avgs = df.groupby(["generation"])["average"].mean()
        min_avg_of_avgs = avg_of_avgs.min()
        max_avg_of_avgs = avg_of_avgs.max()
        avg_of_best = df.groupby(["generation"])["best"].mean()
        min_avg_of_best = avg_of_best.min()
        max_avg_of_best = avg_of_best.max()
        pivot_df["Average of averages"] = avg_of_avgs
        pivot_df["Average of Best"] = avg_of_best

        avg_df = pd.DataFrame(
            {
                "generation": avg_of_avgs.index,
                "avg_of_avgs": avg_of_avgs.values,
                "avg_of_best": avg_of_best.values,
            }
        )

        print(f"Parent: {parent}, Surviver: {surviver}")
        print(avg_df[avg_df["generation"] == 0])
        print(avg_df[avg_df["generation"] == 10000])

        avg_df.plot(y=["avg_of_avgs", "avg_of_best"], linestyle="-", figsize=(10, 5))
        plt.xlabel("Generation")
        plt.ylabel("Fitness")

        plt.title(f"Parent: {parent}, Surviver: {surviver}")
        plt.legend(["Average of averages", "Average of best"])
        plt.grid(True)
        plt.tight_layout()

        plt.savefig(file.split(".")[1] + ".png")

        f.close()
```

Listing 1.15: Code for plotting

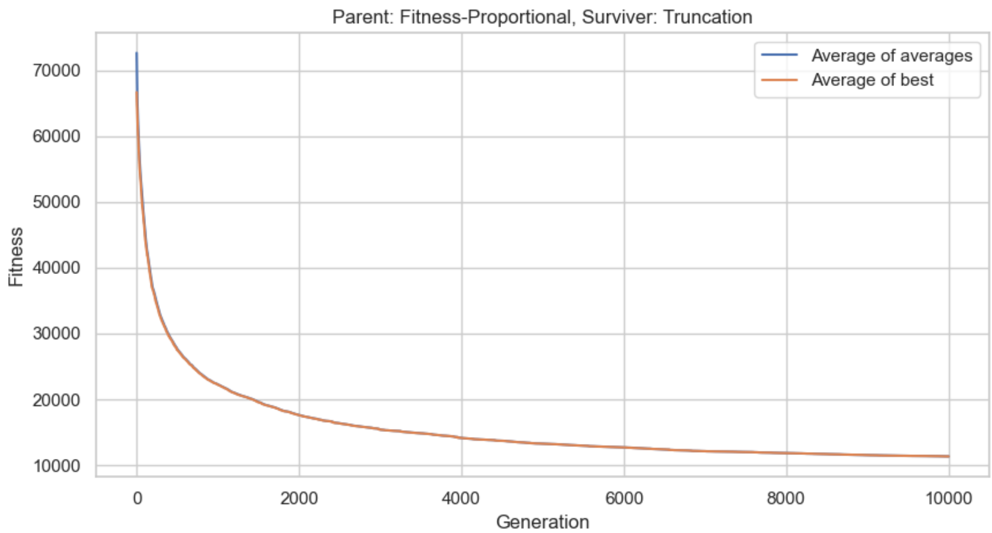
1.2.4 Results and Analysis

- FPS and RBS:



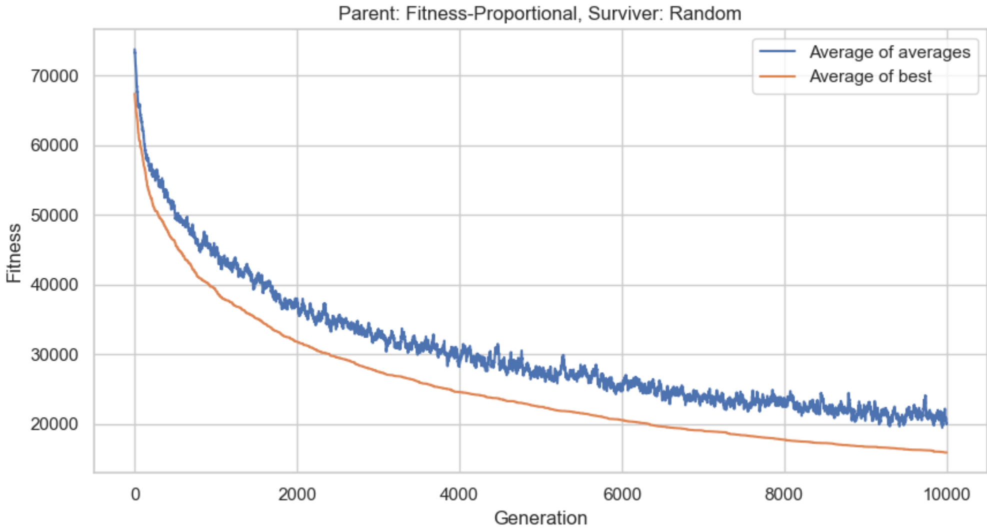
The initial fitness of the population was 71806.62. It was observed that after 10,000 iterations the average cycle fitness was reduced to 11950.47 while the best result achieved so far is 11931.54.

- FPS and Truncation:



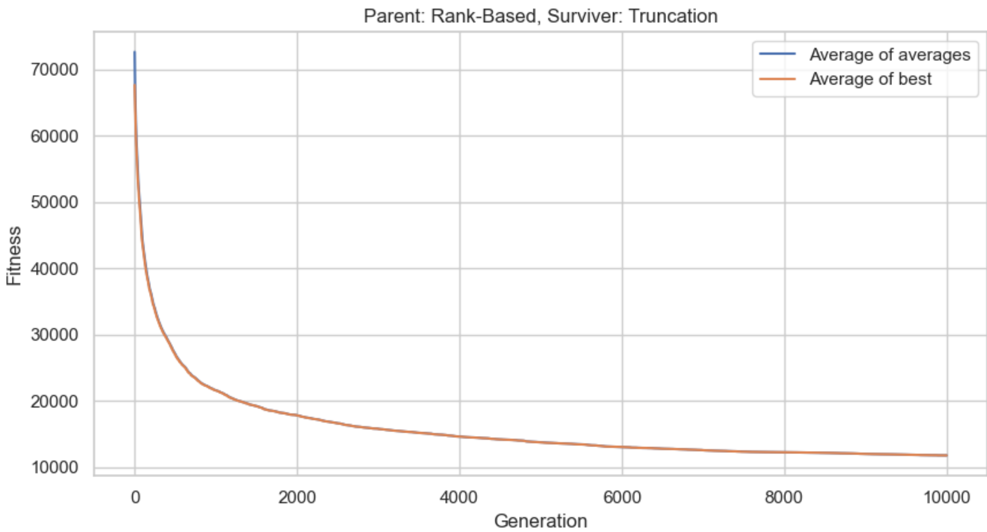
The initial fitness of the population was 72627.211. It was observed that after 10,000 iterations the average cycle fitness was reduced to 11332.911 while the best result achieved so far is 11332.44.

- FPS and Random :



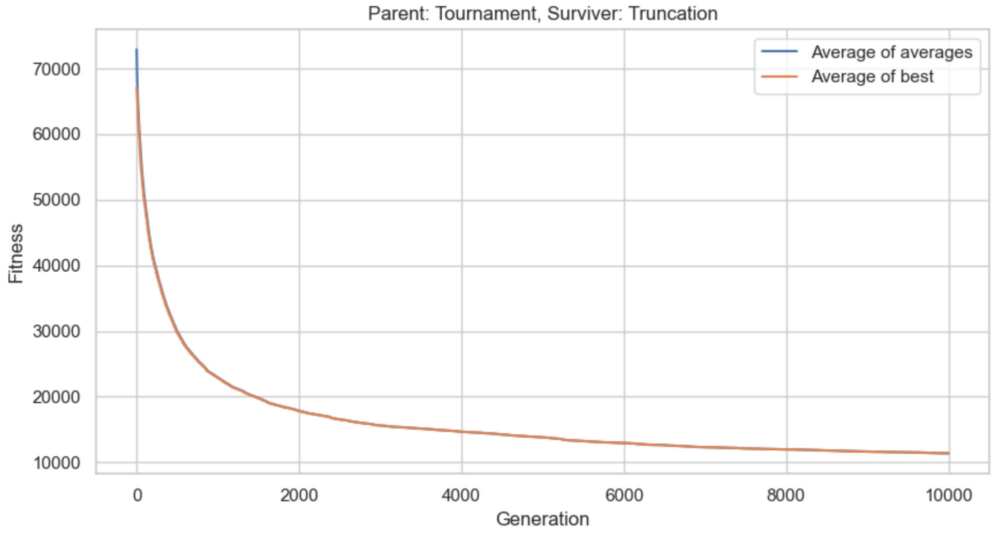
The initial fitness of the population was 73762.37. It was observed that after 10,000 iterations the average cycle fitness was reduced to 20020.02 while the best result achieved so far is 15945.68.

- RBS and Truncation:



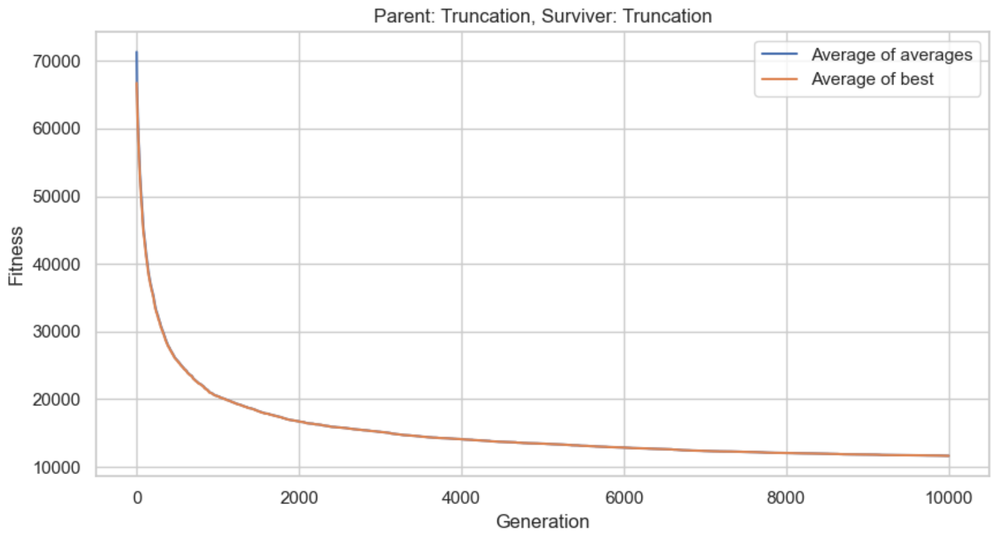
The initial fitness of the population was 72675.94. It was observed that after 10,000 iterations the average cycle fitness was reduced to 11799.74 while the best result achieved so far is 11797.92.

- Tournament and Truncation:



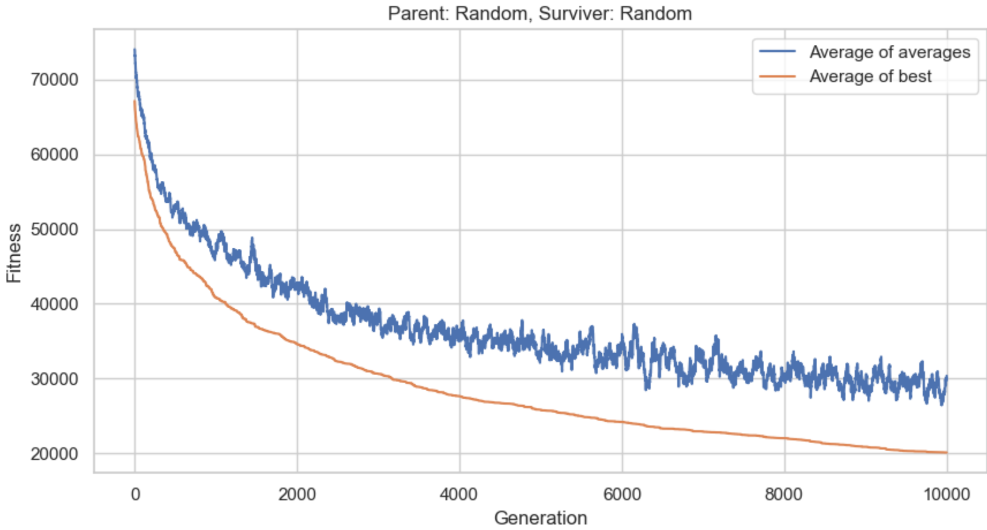
The initial fitness of the population was 72913.783. It was observed that after 10,000 iterations the average cycle fitness was reduced to 11382.87 while the best result achieved so far is 11379.79.

- Truncation and Truncation:



The initial fitness of the population was 71341.46. It was observed that after 10,000 iterations the average cycle fitness was reduced to 11632.99 while the best result achieved so far is 11631.71.

- Random and Random:



The initial fitness of the population was 74056.35. It was observed that after 10,000 iterations the average cycle fitness was reduced to 30370.40 while the best result achieved so far is 20108.88.

1.2.5 Optimal Results and Conclusion

It was realized that optimal results were achieved when we used following schemes: FPS and truncation and tournament and truncation.

1.3 Job Shop Scheduling Problem

1.3.1 Chromosome representation

The chromosome contains operations which are represented as a tuple of machines and jobs. The sequence of operation is varied and the fitness is evaluated in terms of total time.

```
# Reading data
def read_file(self):
    with open(self.filename, "r") as f:
        self.comment = f.readline().strip()
        self.operations = f.readline().strip().split()
        self.total_machines = int(self.operations[1])
        self.total_jobs = int(self.operations[0])
        self.operations_data = {}
        job_no = 0

        for line in f:
            if not line.strip():
                continue
            numbers = line.split()
            # Operations data is a dictionary in which keys are operations and their corresponding value is processing time.
            for i in range(0, len(numbers), 2):
                self.operations_data[(job_no, int(numbers[i]))] = int(
                    numbers[i + 1]
                )
            job_no += 1
# Making a chromosome by shuffling data.
def chromosome(self) -> list:
    temp = list(self.operations_data.keys())
    arr = [i for i in temp]
    random.shuffle(arr)
    return arr
```

Listing 1.16: Chromosome represntation

1.3.2 Fitness Function

To calculate fitness two dictionaries were used which contained information about last processing time of the jobs and machines respectively. For any given operation the maximum time of respective job and machine was picked. Current processing time was added to it. A dictionary named timings store all information about the timings of the particular operation sequences and was used to make Gantt charts.

```
def evaluate_fitness(self, chromosome) -> float:
# Storing machine and jobs latest operating time
    self.machine_process_time = {
        machine: 0 for machine in range(self.total_machines)
    }
    self.job_process_time = {job: 0 for job in range(self.total_jobs)}
    #Storing starting and ending times of all operations.
    self.timings = {time: [0, 0] for time in self.operations_data.keys()}
    for i in range(len(chromosome)):
        current_process_time = self.operations_data[chromosome[i]]
        job, machine = chromosome[i]
        end_process_time = (
            max(self.machine_process_time[machine], self.job_process_time[job])
            + current_process_time
        )
        self.timings[chromosome[i]] = [
            max(self.machine_process_time[machine], self.job_process_time[job]),
            end_process_time,
        ]
        self.machine_process_time[machine] = end_process_time
        self.job_process_time[job] = end_process_time

    return float(
        max(
```

```
                max(self.machine_process_time.values()),
                max(self.job_process_time.values()),
            )
        )
# Fitness of each individual in the population was computed.
    def compute_population_fitness(self, population: dict) -> dict:
        fitness_dictionary = {}
        for individual, chromosome in population.items():
            fitness_dictionary[individual] = self.evaluate_fitness(chromosome)
        return fitness_dictionary
```

Listing 1.17: Fitness Function

1.3.3 Automated Code for Plotting

```
import json
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

files = [
    "jssp_abz5.txt_p1_s2.json",
    "jssp_abz5.txt_p1_s4.json",
    "jssp_abz5.txt_p3_s4.json",
    "jssp_abz5.txt_p4_s4.json",
    "jssp_abz5.txt_p5_s5.json",
    "jssp_abz6.txt_p1_s2.json",
    "jssp_abz6.txt_p1_s4.json",
    "jssp_abz6.txt_p3_s4.json",
    "jssp_abz6.txt_p4_s4.json",
    "jssp_abz6.txt_p5_s5.json",
     "jssp_abz7.txt_p1_s2.json",
     "jssp_abz7.txt_p1_s4.json",
    "jssp_abz7.txt_p3_s4.json",
    "jssp_abz7.txt_p4_s4.json",
    "jssp_abz7.txt_p5_s5.json",
```

```
]

for file in files:

    algo = file.split(".")[1]
    parent = algo.split("_")[1]
    surviver = algo.split("_")[2]

    if parent == "p1":
        parent = "Fitness-Proportional"
    elif parent == "p2":
        parent = "Rank-Based"
    elif parent == "p3":
        parent = "Tournament"
    elif parent == "p4":
        parent = "Truncation"
    elif parent == "p5":
        parent = "Random"

    if surviver == "s1":
        surviver = "Fitness-Proportional"
    elif surviver == "s2":
        surviver = "Rank-Based"
    elif surviver == "s3":
        surviver = "Tournament"
    elif surviver == "s4":
        surviver = "Truncation"
    elif surviver == "s5":
        surviver = "Random"

    df = pd.DataFrame()
    pivot_df = pd.DataFrame()
    with open(file, "r") as f:
        data = json.load(f)

```

```python
        rows = []

        for iteration, generation_data in data.items():
            iteration_number = int(iteration)
            for generation_info in generation_data:
                generation_number = generation_info["generation"]
                average = generation_info["average"]
                best = generation_info["best"]
                best_chromosome = generation_info["best_chromesome"]

                row = {
                    "iteration": iteration_number,
                    "generation": generation_number,
                    "average": average,
                    "best": best,
                    "best_chromosome": best_chromosome,
                }
                rows.append(row)

        df = pd.DataFrame(rows)

        pivot_df = df.pivot_table(
            index="generation",
            columns=["iteration"],
            values=["average", "best", "best_chromosome"],
            aggfunc="first",
        )

        avg_of_avgs = df.groupby(["generation"])["average"].mean()
        avg_of_best = df.groupby(["generation"])["best"].mean()
        pivot_df["Average of averages"] = avg_of_avgs
        pivot_df["Average of Best"] = avg_of_best

        avg_df = pd.DataFrame(
            {
```

```
                "generation": avg_of_avgs.index,
                "avg_of_avgs": avg_of_avgs.values,
                "avg_of_best": avg_of_best.values,
            }
        )

        print(f"Parent: {parent}, Surviver: {surviver}")
        print(avg_df[avg_df["generation"] == 0])
        print(avg_df[avg_df["generation"] == 5000])

        avg_df.plot(y=["avg_of_avgs", "avg_of_best"], linestyle="-", figsize=(10, 5))
        plt.xlabel("Generation")
        plt.ylabel("Fitness")

        plt.title(f"Parent: {parent}, Surviver: {surviver}")
        plt.legend(["Average of averages", "Average of best"])

        plt.savefig(".".join(file.split(".")[0:2]) + ".png")

        minimum_avg_best = pivot_df["Average of Best"].idxmin()

        min_row = pivot_df.loc[minimum_avg_best]

        # Get the best chromosome corresponding to the minimum value
        best_chromosome_of_min_best = min_row["best_chromosome"].values[0]
        plot_data = {}
        for data in best_chromosome_of_min_best.values():
            plot_data[(data[0], data[1])] = [data[2], data[3]]

        # Convert data to a format suitable for DataFrame
        formatted_data = [
            (job, machine, start, end)
            for (job, machine), (start, end) in plot_data.items()
        ]
        df = pd.DataFrame(formatted_data, columns=["Job", "Machine", "Start", "End"])
```

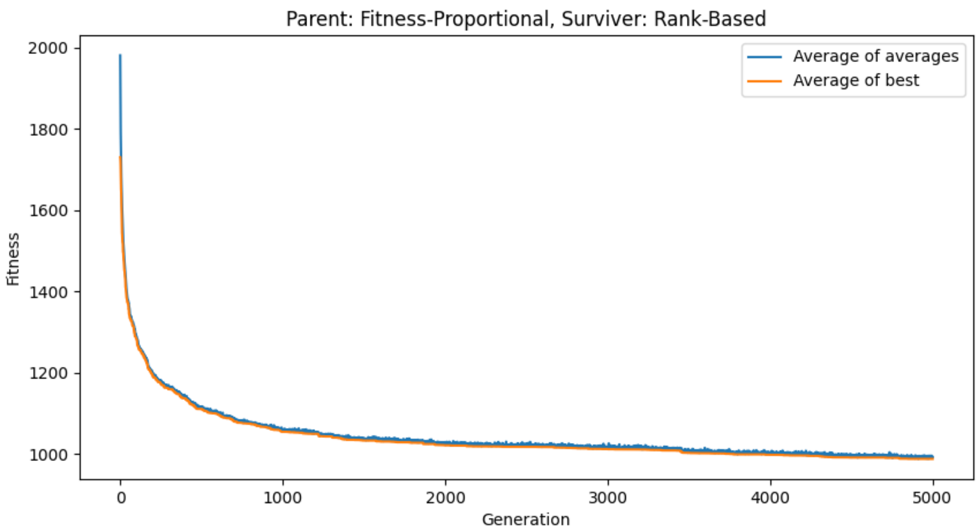
```python

        # Get unique jobs
        unique_jobs = df["Job"].unique()

        # Generate unique colors for each job
        colors = plt.cm.tab10(np.linspace(0, 1, len(unique_jobs)))

        # Map each job to a unique color
        job_colors = {job: color for job, color in zip(unique_jobs, colors)}

        # Create a Gantt chart
        fig, ax = plt.subplots(figsize=(10, 6))

        # Plot the bars with unique colors
        for row in df.itertuples(index=False):
            ax.barh(
                y=row.Machine,
                width=row.End - row.Start,
                left=row.Start,
                height=0.5,
                align="center",
                alpha=0.6,
                color=job_colors[row.Job],
            )

        # Set labels and title
        ax.set_xlabel("Time")
        ax.set_ylabel("Machine")
        ax.set_title(f"Best Chromosome | Parent: {parent}, Surviver: {surviver}")

        # Show grid
        ax.grid(True)

        plt.savefig(".".join(file.split(".")[0:2]) + "_gantt_chart.png")

```

```
        f.close()
```

Listing 1.18: Code for plotting

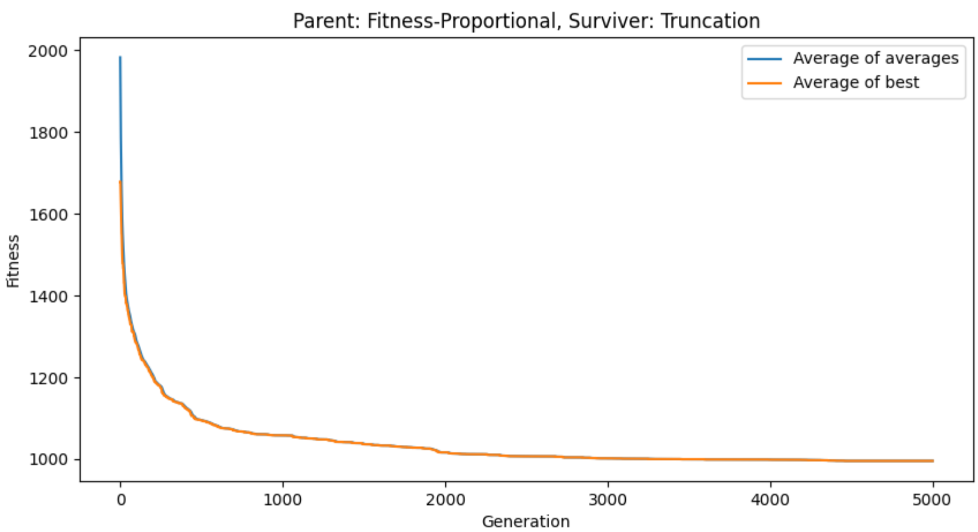
1.3.4 Results and Analysis for dataset abz5

- FPS and RBS:



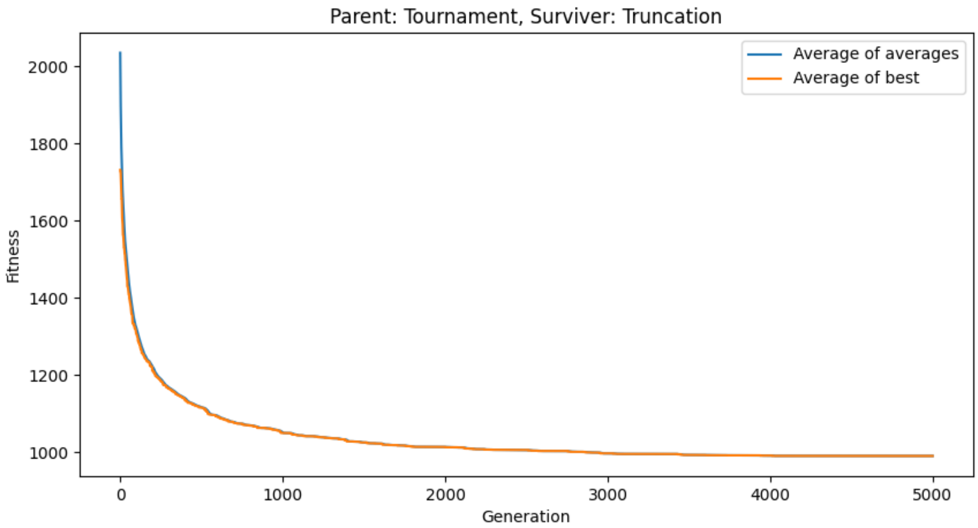
The initial fitness of the population was 1980.61. It was observed that after 5000 iterations the average cycle fitness was reduced to 992.304 while the best result achieved so far is 987.5.

- FPS and Truncation:



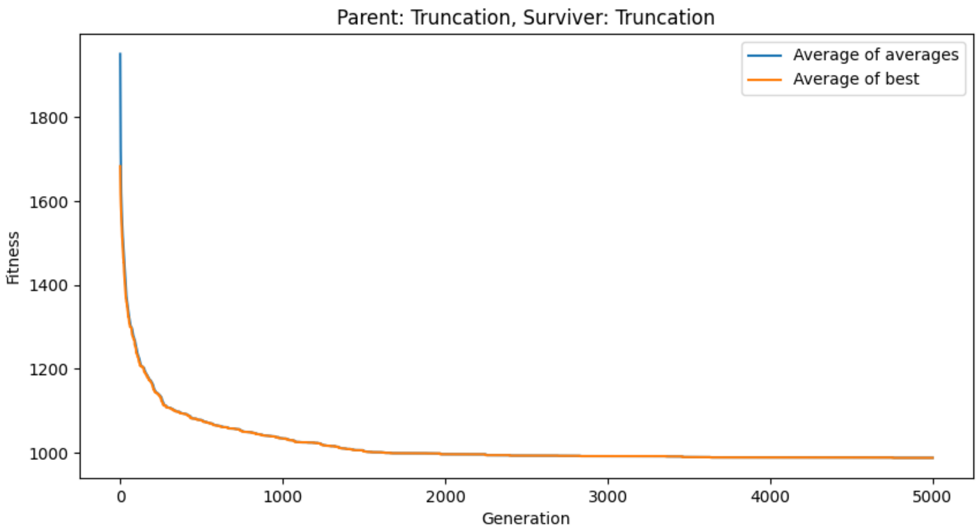
The initial fitness of the population was 1982.87. It was observed that after 5000 iterations the average cycle fitness was reduced to 995.7 while the best result achieved so far is 995.7.

- Tournament and Truncation:



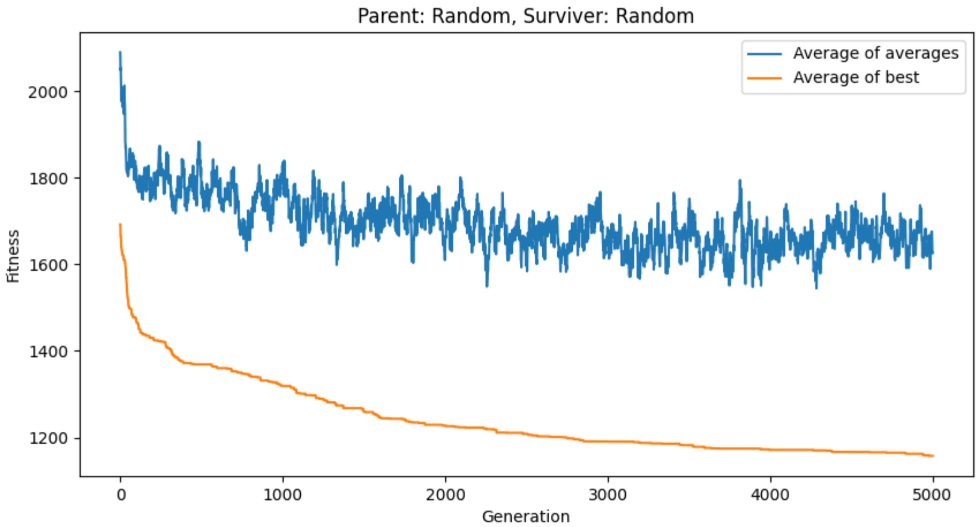
The initial fitness of the population was 2034.512. It was observed that after 5000 iterations the average cycle fitness was reduced to 990 while the best result achieved so far is 990.9.

- Truncation and Truncation:



The initial fitness of the population was 1950.786. It was observed that after 5000 iterations the average cycle fitness was reduced to 988.1 while the best result achieved so far is 988.1.

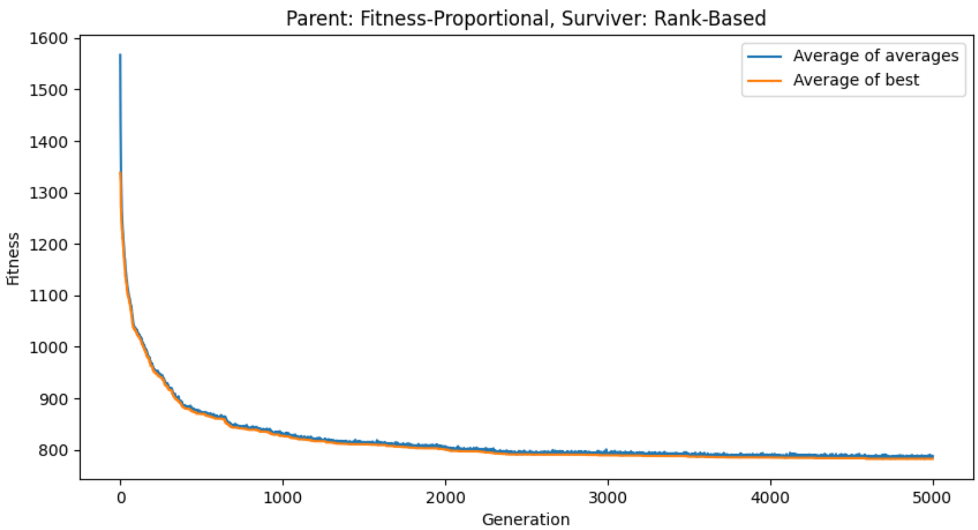
- Random and Random:



The initial fitness of the population was 2089.68. It was observed that after 5000 iterations the average cycle fitness was reduced to 1626.254 while the best result achieved so far is 1157.4.

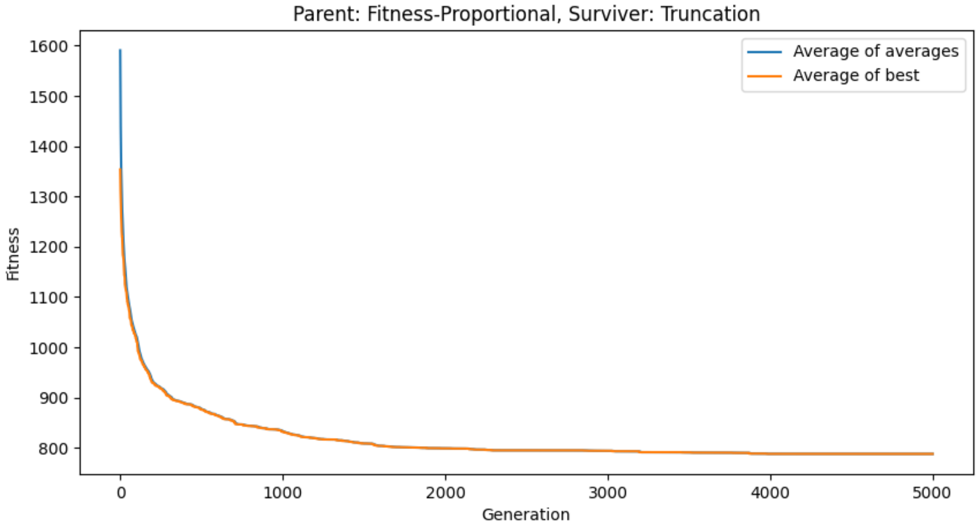
1.3.5 Results and Analysis for dataset abz6

- FPS and RBS:



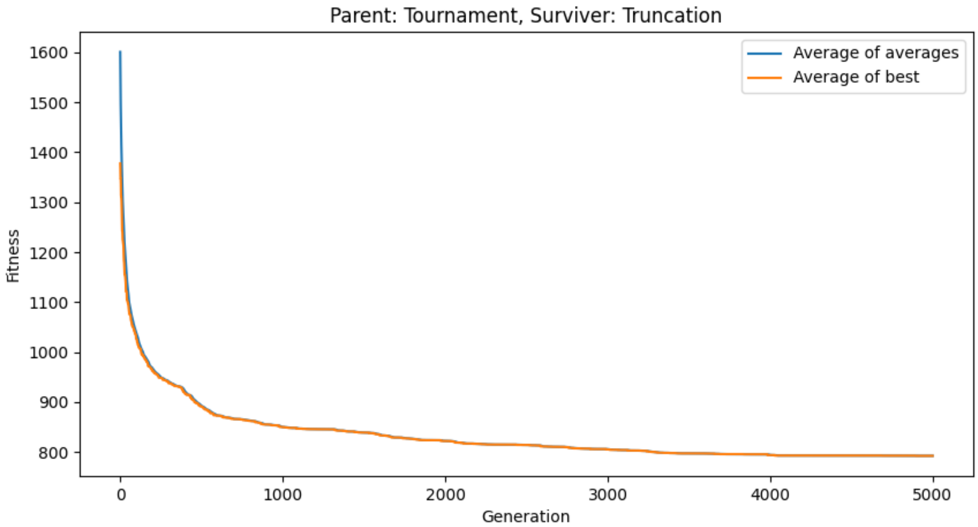
The initial fitness of the population was 1567.115. It was observed that after 5000 iterations the average cycle fitness was reduced to 788.05 while the best result achieved so far is 782.4.

- FPS and Truncation:



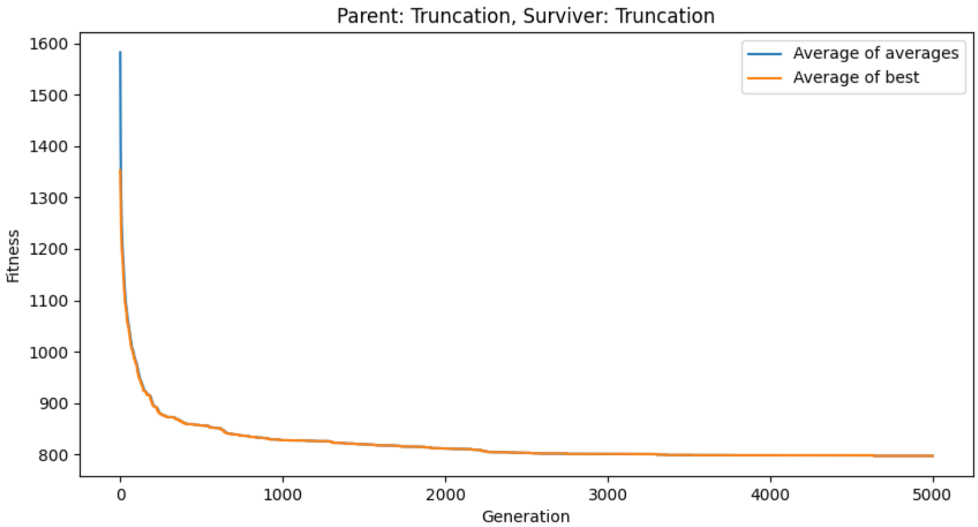
The initial fitness of the population was 1590.69. It was observed that after 5000 iterations the average cycle fitness was reduced to 788 while the best result achieved so far is 788.

- Tournament and Truncation:



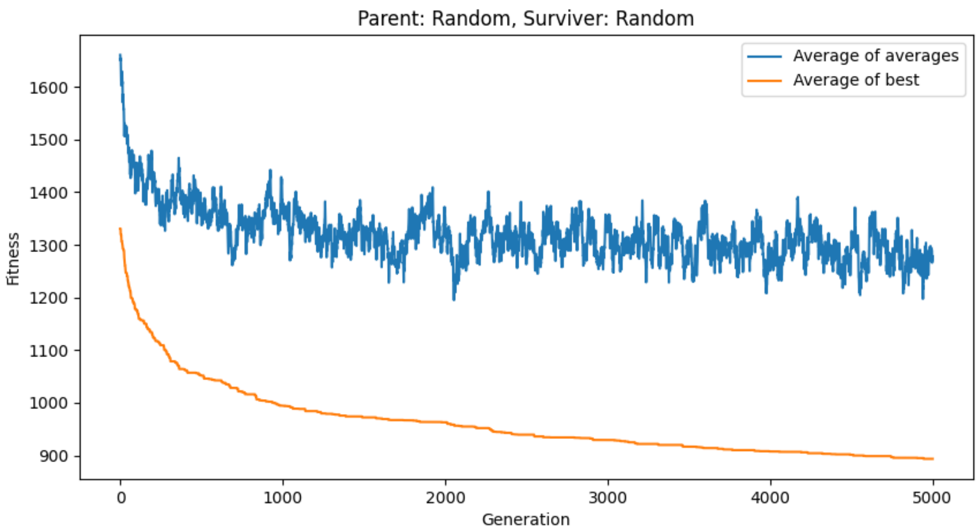
The initial fitness of the population was 1600.833. It was observed that after 5000 iterations the average cycle fitness was reduced to 792.6 while the best result achieved so far is 792.6.

- Truncation and Truncation:



The initial fitness of the population was 1582.26. It was observed that after 5000 iterations the average cycle fitness was reduced to 797.4 while the best result achieved so far is 797.4.

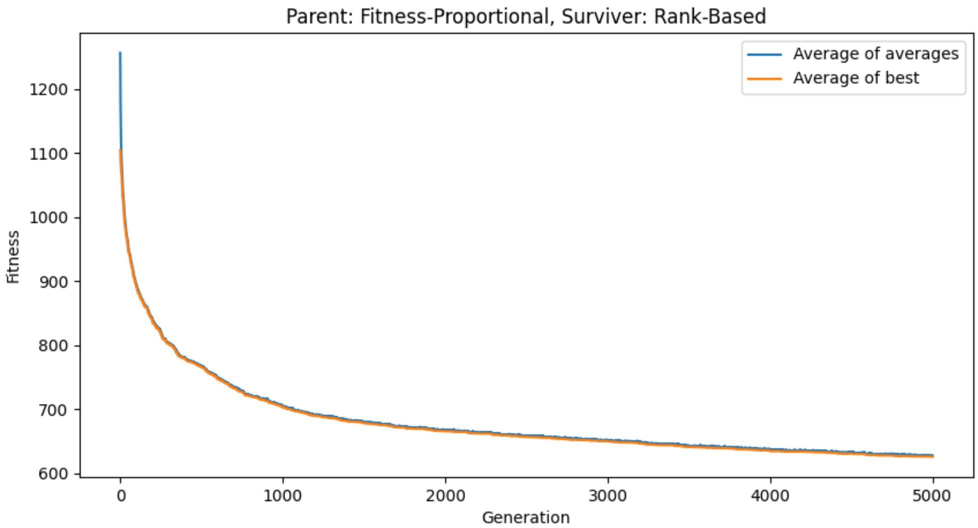
- Random and Random:



The initial fitness of the population was 1660.91. It was observed that after 5000 iterations the average cycle fitness was reduced to 1271.63 while the best result achieved so far is 893.5.

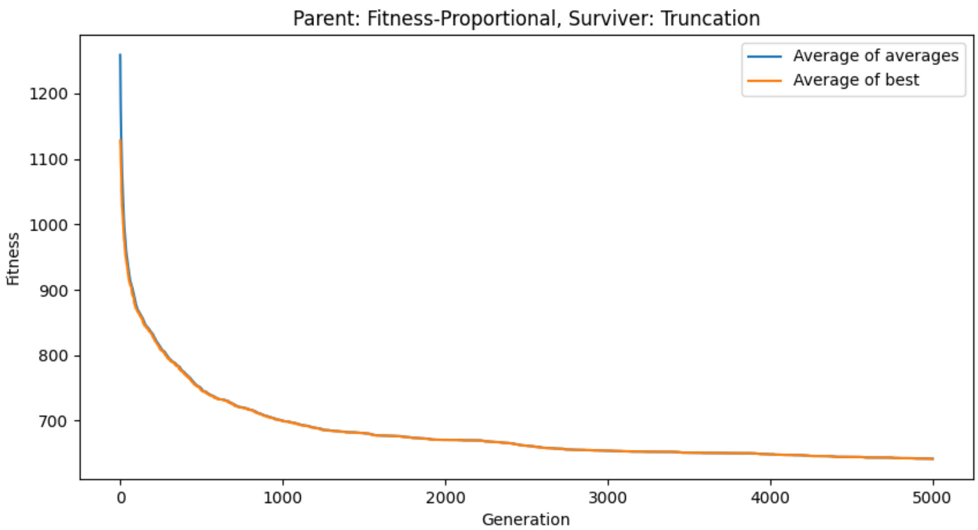
1.3.6 Results and Analysis for dataset abz7

- FPS and RBS:



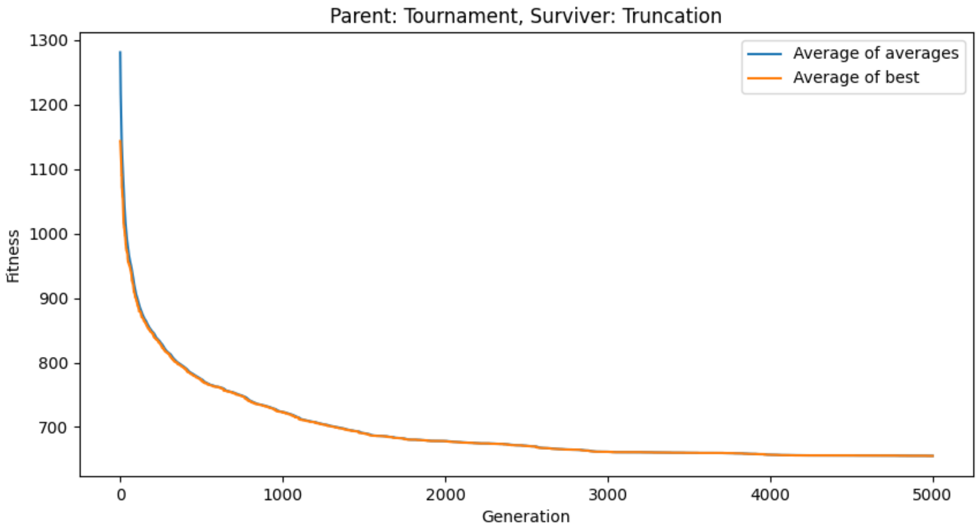
The initial fitness of the population was 1256.073. It was observed that after 5000 iterations the average cycle fitness was reduced to 627.447 while the best result achieved so far is 625.8.

- FPS and Truncation:



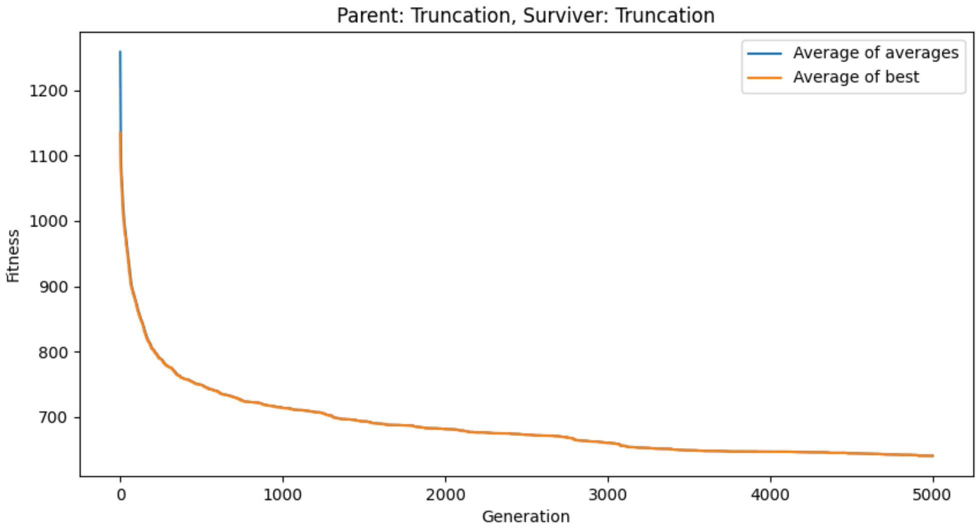
The initial fitness of the population was 1258.84. It was observed that after 5000 iterations the average cycle fitness was reduced to 641.897 while the best result achieved so far is 641.8.

- Tournament and Truncation:



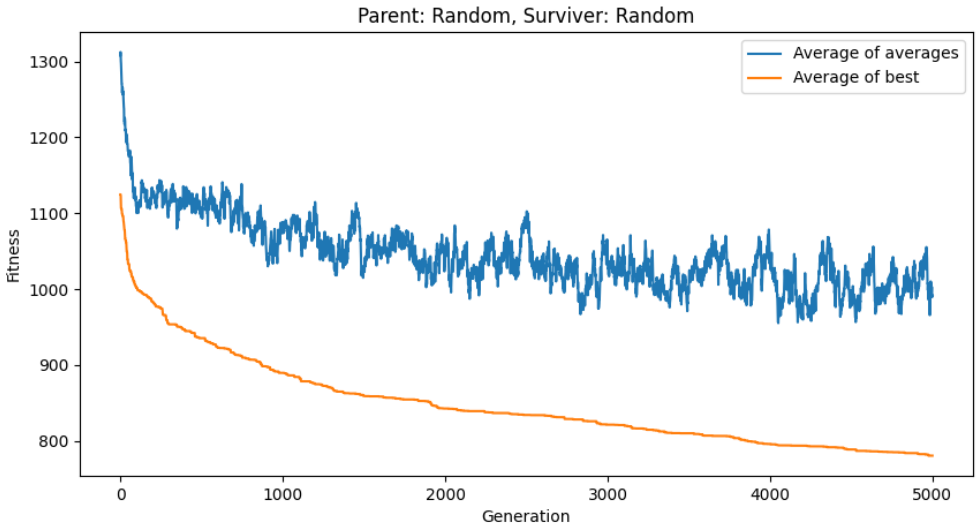
The initial fitness of the population was 1281.09. It was observed that after 5000 iterations the average cycle fitness was reduced to 655.2 while the best result achieved so far is 655.2.

- Truncation and Truncation:



The initial fitness of the population was 1258.39. It was observed that after 5000 iterations the average cycle fitness was reduced to 640.7 while the best result achieved so far is 640.7.

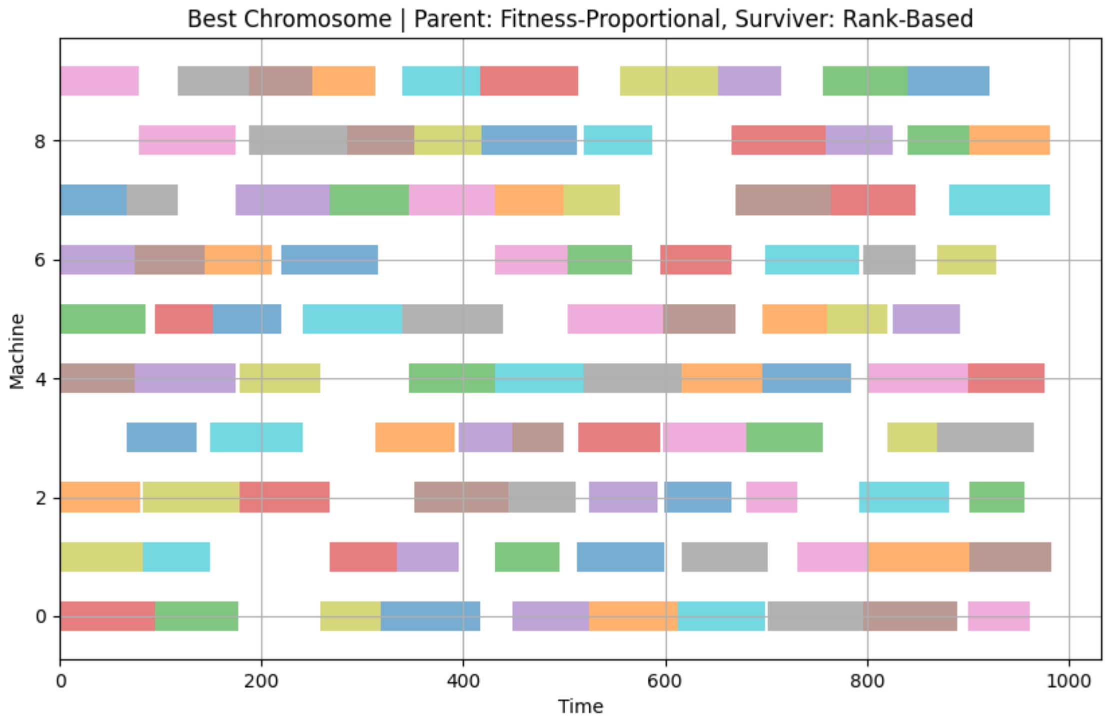
- Random and Random:



The initial fitness of the population was 1308.04. It was observed that after 5000 iterations the average cycle fitness was reduced to 991.89 while the best result achieved so far is 780.6.

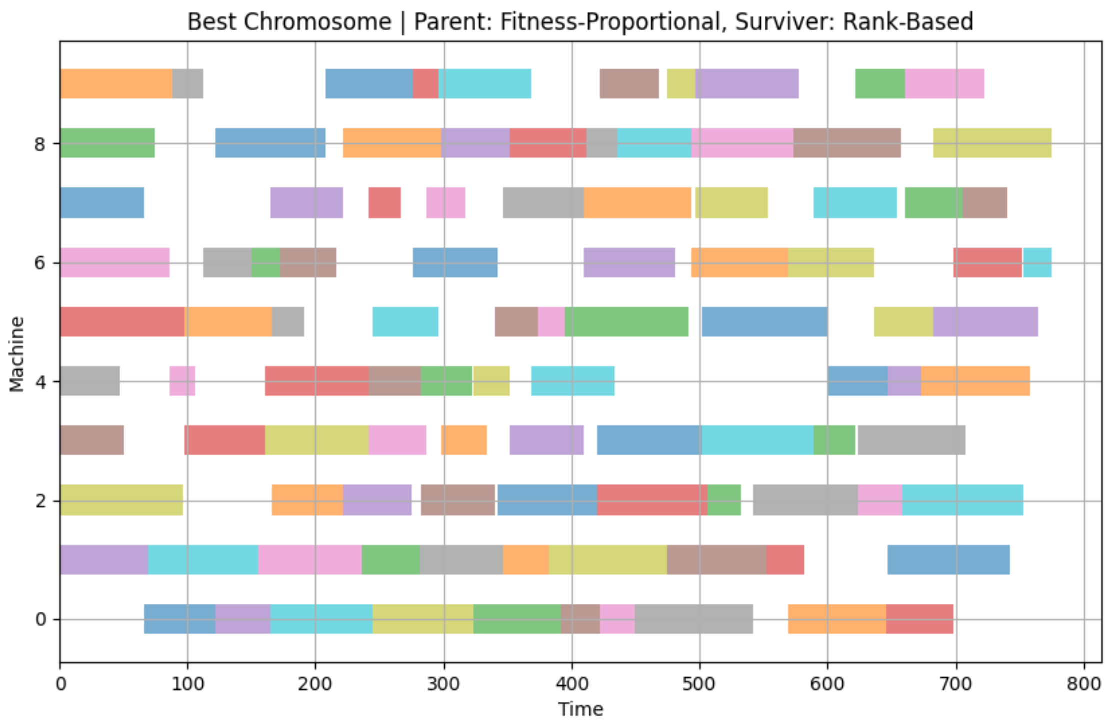
1.3.7 Gantt Charts for optimal solutions for all datasets

- **abz5.txt**



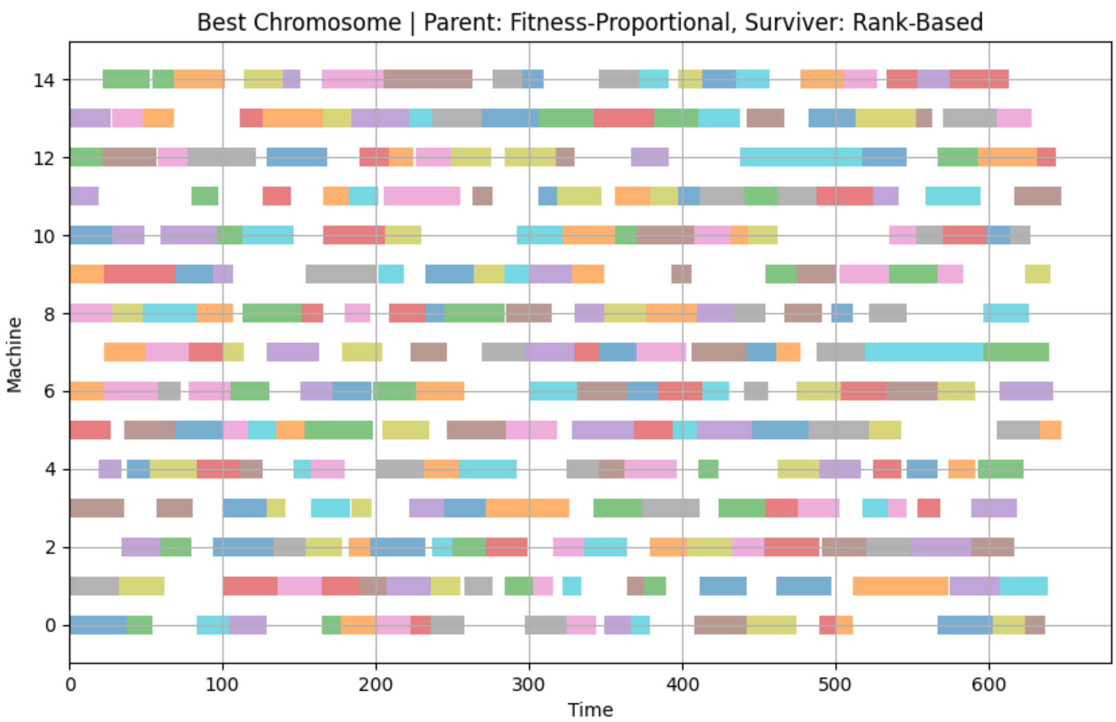
Using fitness selection and rank based selection for parents and survivor selection respectively.

- **abz6.txt**



Using fitness selection and rank based selection for parents and survivor selection respectively.

- **abz7.txt**



Using fitness selection and rank based selection for parents and survivor selection respectively.

1.3.8 Optimal Results and Conclusion

It was realized that by using fitness selection and rank based selection for parents and survivor selection respectively optimal results were achieved which can also be shown from gantt charts above.

1.4 Mona Lisa

1.4.1 Chromosome representation

Firstly we are reading the reference image and resizing it. We are also finding colours from the image and using those only while generating polygons. By using this we are able to converge faster. A chromosome contains multiple polygons, each polygons have 3 attributes, vertices, colors and transparency. The latter is fixed to 50%.

```
def read_file(self):
"""
Reading image, resizing it, extracting colours from it.
"""
        image = Image.open(self.target_human_image)
        image = image.resize((IMAGE_WIDTH, IMAGE_HEIGHT))
        self.target_human_image = image

        # Convert the image to RGB mode
        target_image = image.convert("RGB")

        # Get the image data as a numpy array
        image_array = np.array(target_image)

        # Reshape the image data to a 2D array of pixels (rows) x RGB values (columns)
        pixels = image_array.reshape(-1, 3)

        # Randomly sample colors or use clustering algorithm to find dominant colors
        unique_colors = np.unique(pixels, axis=0)

        self.target_image_colors = unique_colors.tolist()

        # image.show()

    def chromosome(self) -> list:
```

```
    """
    For each chromosome, there are multiple polygons, each containing vertices, color
    of the polygon and its transparency level.
    """
        chromosome = []
        for _ in range(self.num_polygons):
            num_vertices = random.randint(3, self.max_vertices)
            color = random.choice(self.target_image_colors)
            polygon = {
                "vertices": [
                    (random.randint(0, IMAGE_WIDTH), random.randint(0, IMAGE_HEIGHT))
                    for _ in range(num_vertices)
                ],
                "color": color,
                "transparency": float(0.5),
            }
            chromosome.append(polygon)
        return chromosome
```

Listing 1.19: Chromosome represntation

1.4.2 Fitness Function

Using help of python's pillow library the image was rendered and difference with reference image was computed. ImageChops function computes abs difference between the pixels of the image. The fitness represents the difference between the two images. The least the fitness, more the similarity between the two images.

```
def compute_fitness(self, rendered_image, target_human_image) -> float:
"""
Computing difference between reference and generated image.
"""
        diff = ImageChops.difference(rendered_image, target_human_image)
        totdiff = np.array(diff.getdata()).sum()

        return totdiff
```

```

    def evaluate_fitness(self, chromosome) -> float:
    """
    Calling the helper function which generates the image with drawn polygons on
   white canvas
    """
        rendered_image = self.render_individual(
            chromosome, self.target_human_image.size
        )

        return self.compute_fitness(rendered_image, self.target_human_image)

    def compute_population_fitness(self, population: dict) -> dict:
    """
    Computing fitness of each individual in population
    """
        fitness_dictionary = {}
        for individual, chromosome in population.items():
            fitness_dictionary[individual] = self.evaluate_fitness(chromosome)
        return fitness_dictionary
```

Listing 1.20: Fitness Function

1.4.3 Modification on EA

The code for mutation was modified the mutation rate was 100%. Two new random polygons were replacing any two polygons in offsprings.

```

    def mutation(self) -> None:
        if random.random() < self.mutation_rate:
            num_vertices = random.randint(3, self.max_vertices)
            for individual in self.offsprings.keys():
                index1 = random.randint(0, len(self.offsprings[individual]) - 1)
                index2 = random.randint(0, len(self.offsprings[individual]) - 1)
                self.offsprings[individual][index1] = {
```

```
                    "vertices": [
                        (random.randint(0, IMAGE_WIDTH), random.randint(0,
IMAGE_HEIGHT))
                        for _ in range(num_vertices)
                    ],
                    "color": random.choice(self.target_image_colors),
                    "transparency": float(0.5),
                    }
                self.offsprings[individual][index2] = {
                    "vertices": [
                        (random.randint(0, IMAGE_WIDTH), random.randint(0,
IMAGE_HEIGHT))
                        for _ in range(num_vertices)
                    ],
                    "color": random.choice(self.target_image_colors),
                    "transparency": float(0.5),
                    }

```

1.4.4 Code for generating image

Drawing polygons of different colours and 50% transparency level on a white canvas and returns the image.

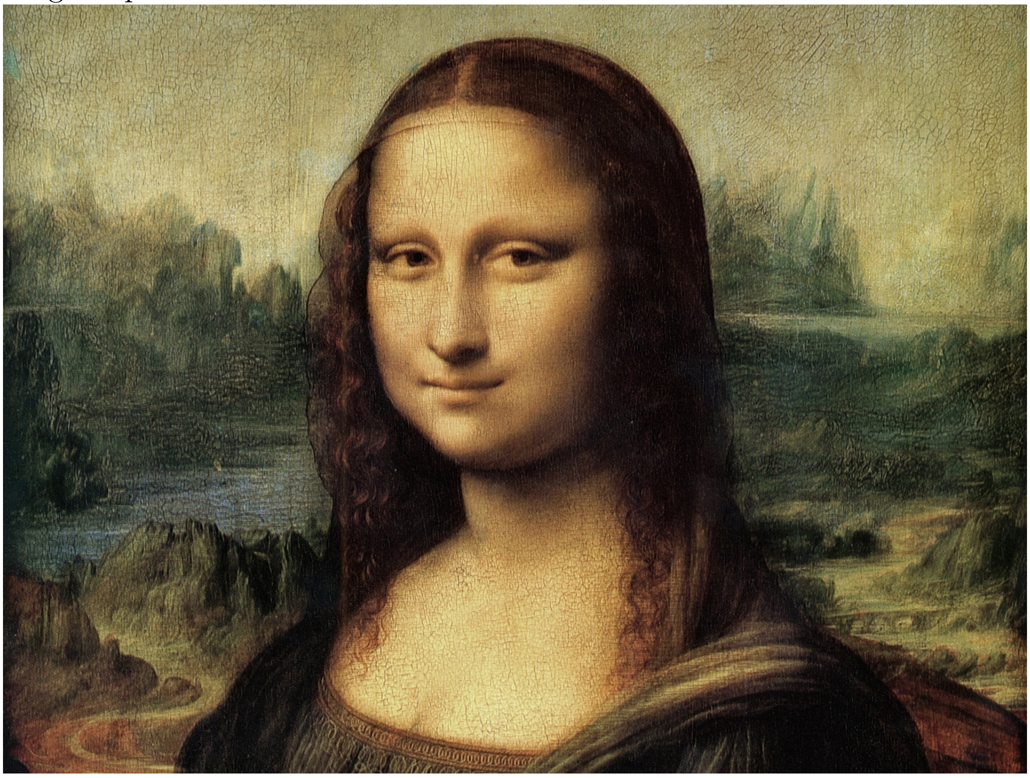
```
    def render_individual(self, chromosome, image_size) -> Image:
        image = Image.new("RGB", image_size, color="white")
        draw = ImageDraw.Draw(image,"RGBA")
        for polygon in chromosome:
            vertices = [(x, y) for x, y in polygon["vertices"]]
            color = tuple(polygon["color"])
            transparency = int(255 * polygon["transparency"])
            draw.polygon(vertices, fill=color + (transparency,))

        image.save(f"images/image_{self.counter}.png")
        self.counter += 1
```

```python
        # image.show()

        return image
```

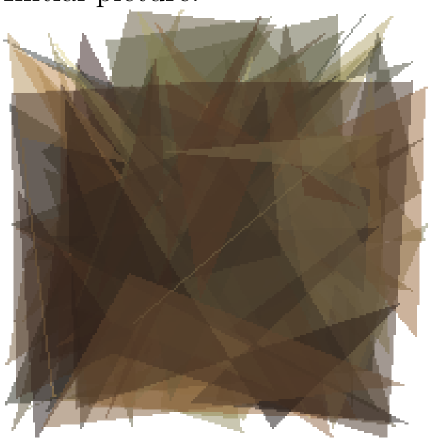
Listing 1.21: Code for generating image

1.4.5 Images at different iterations for Mona lisa

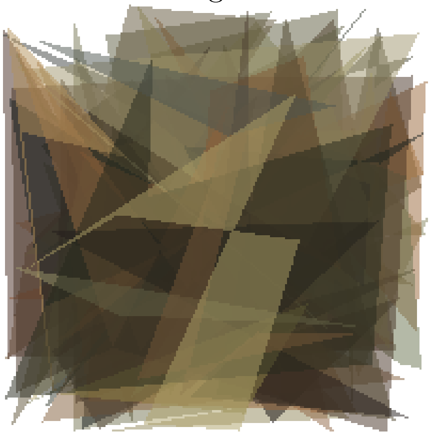
- Original picture:



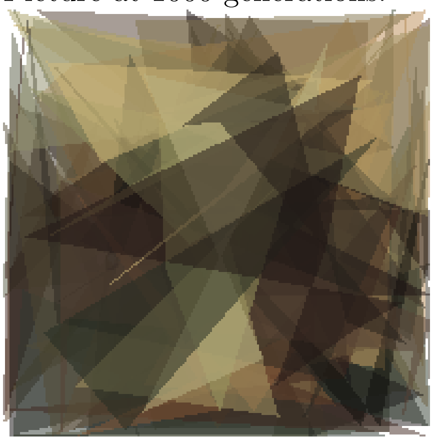
- Initial picture:



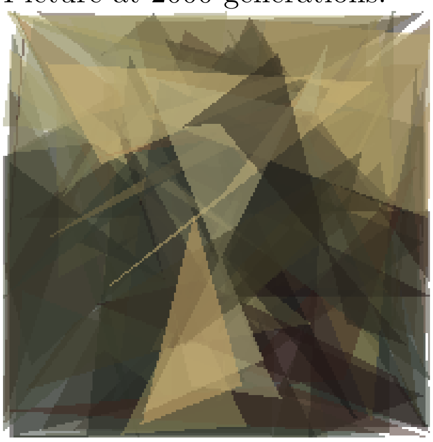
- Picture at 200 generations:



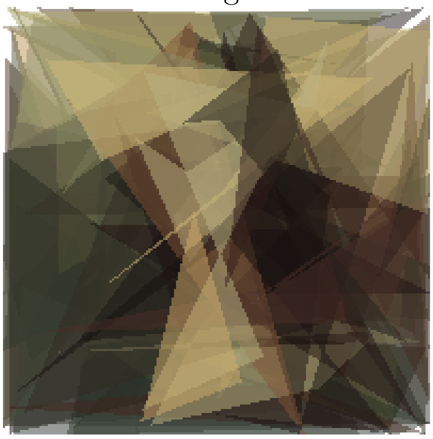
- Picture at 1000 generations:



- Picture at 2000 generations:



- Picture at 3500 generations:

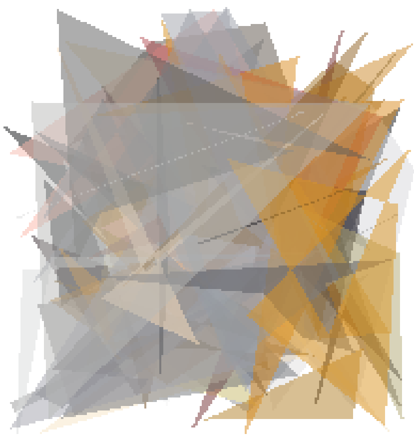


- Picture at 5000 generations:

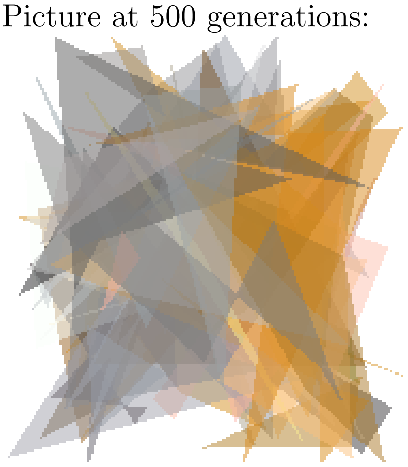
1.4.6 Images at different iterations for Tom and Jerry

- Original picture:

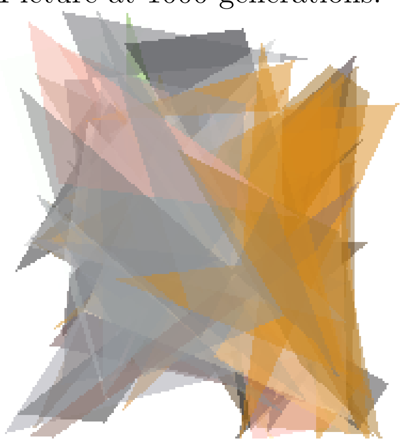
- Initial picture:



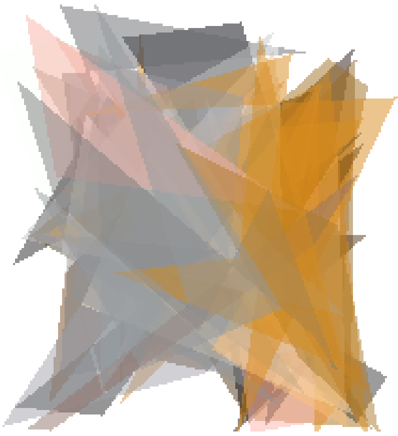
- Picture at 500 generations:



- Picture at 1000 generations:



- Picture at 2000 generations:



1.4.7 Conclusion

It can be seen that the results are converging and if we run it for many iterations around 100K, it can be expected that we will receive a almost perfect pictures. Hence it can be concluded that the following code would work on any generic image.