Erkan Dogan
2166262
2023-05-21

EE449 HW2

2. Experimental Results

Figures and images are sorted vertically from minimum parameter value to maximum parameter.

Images are sorted horizontally from first generation to last generation.

2.1. Default Parameters

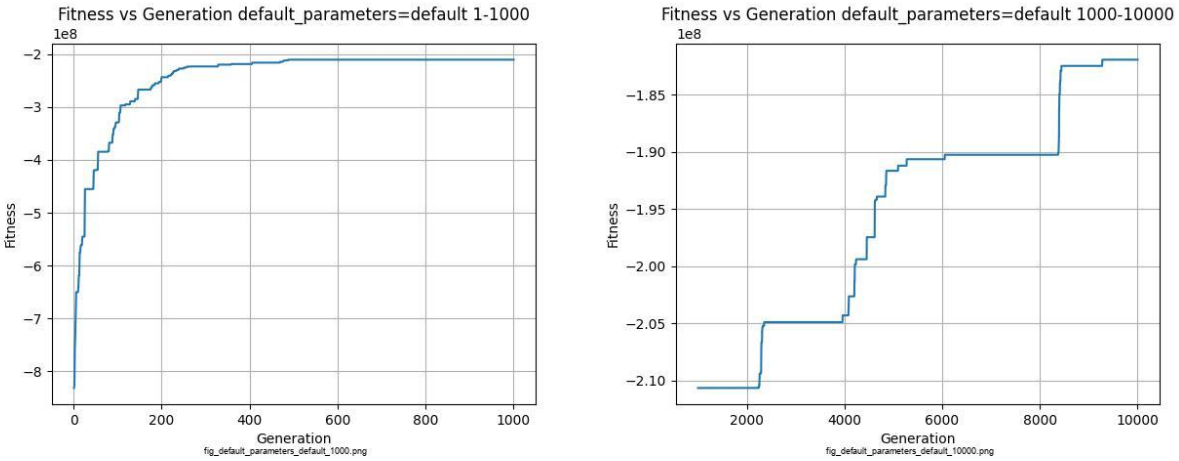

Figure 1 Fitness vs Generation Figures for default parameters

Figure 2 Images per Generation for default parameters

In the first 1000 generations, we see a significant fitness change since we initialized our population with random individuals, and it is highly possible to have improvements in each generation. After 1000 generations, we see an interesting change in 4000-6000 generations, which may mean that algorithms discovered good mutation points or crossovers.

2.2. Different num_inds (5, 10, 20, 40, 60)

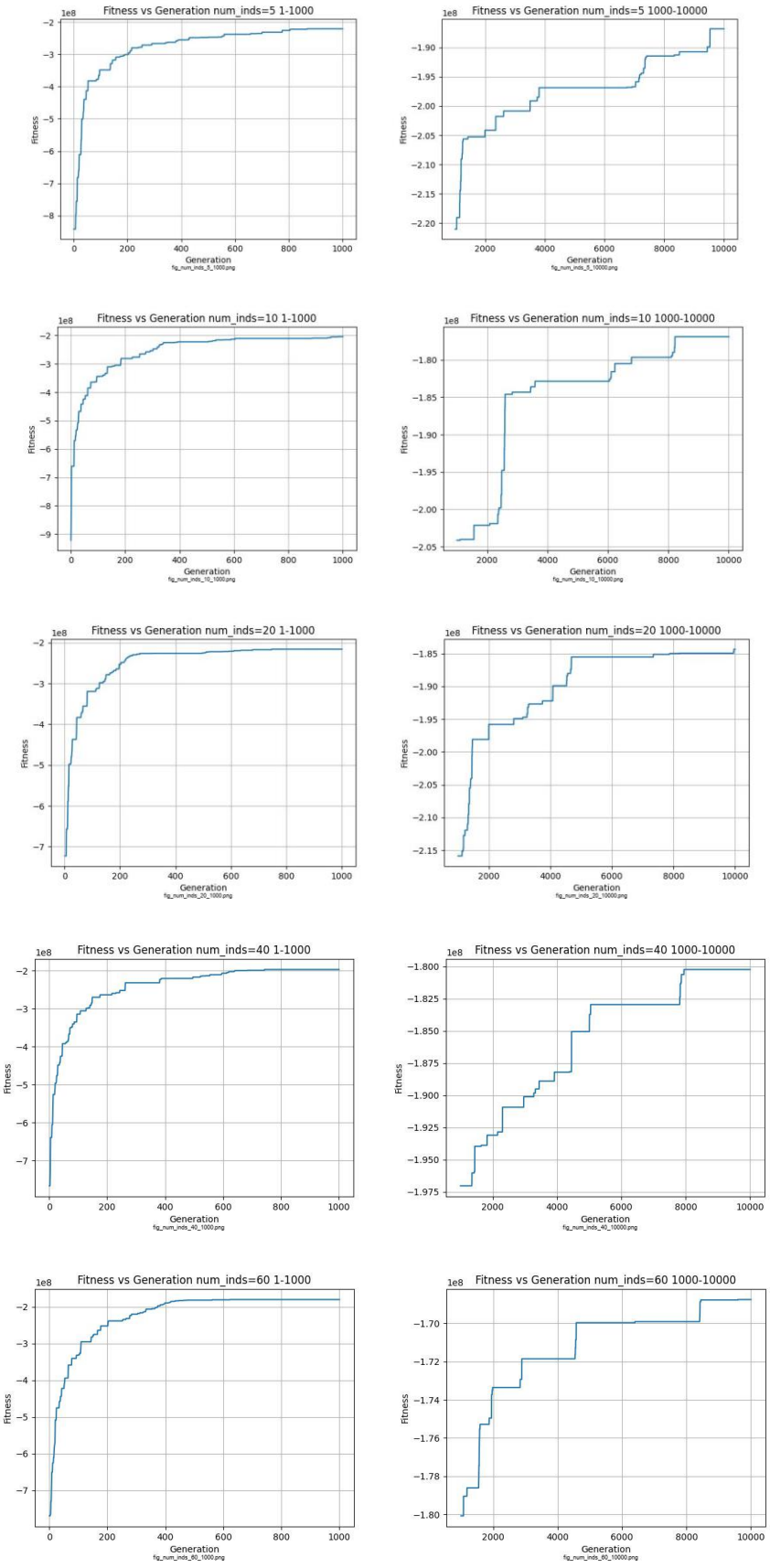

Figure 3 Fitness vs Generation Figures for num_inds (5, 10, 20, 40, 60)

Erkan Dogan
2166262
2023-05-21

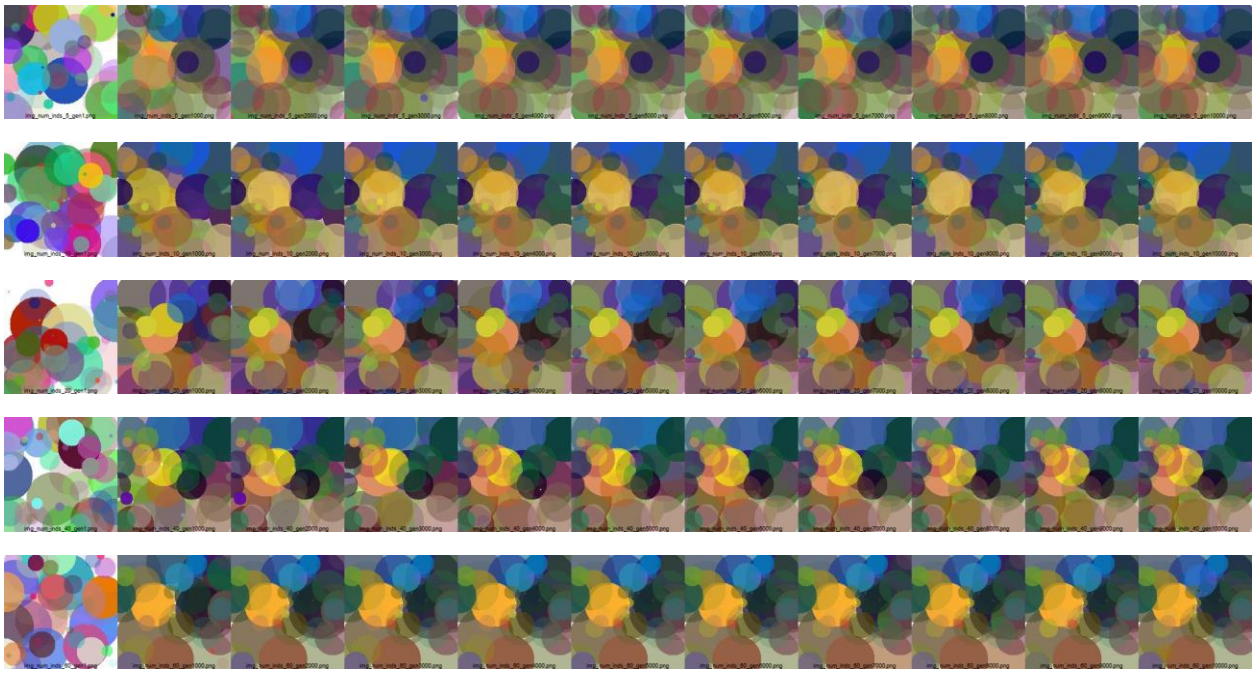

Figure 4 Images per Generation for num_inds (5, 10, 20, 40, 60)

As we improve our number of individuals, we see better fitness values since the possibility of finding a good individual increases. As our pool for mutation and parents increases, we have more chances to get better generations.

2.3. Different num_genes (15, 30, 50, 80, 120)

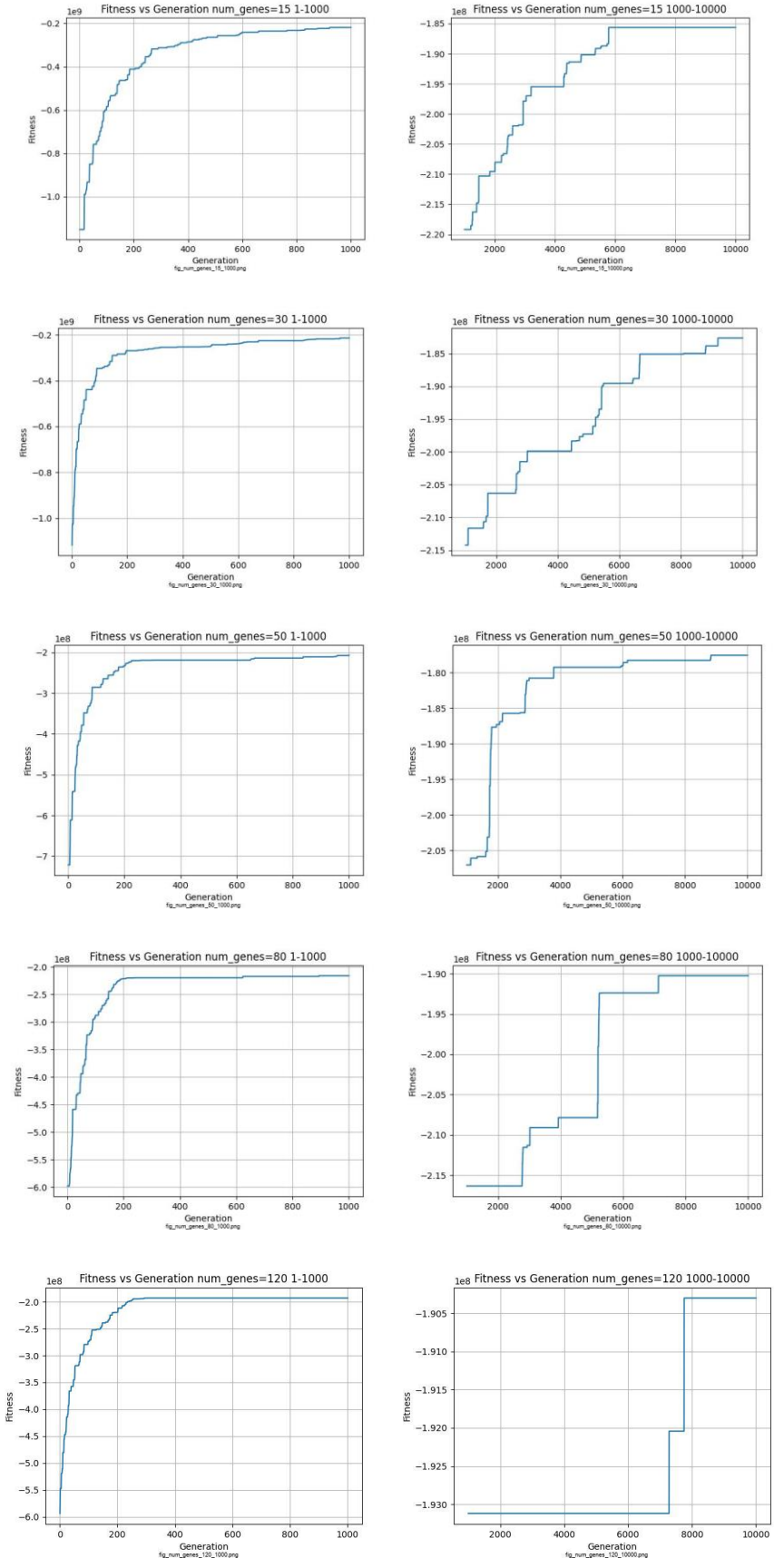

Figure 5 Fitness vs Generation Figures for num_genes (15, 30, 50, 80, 120)

Erkan Dogan
2166262
2023-05-21

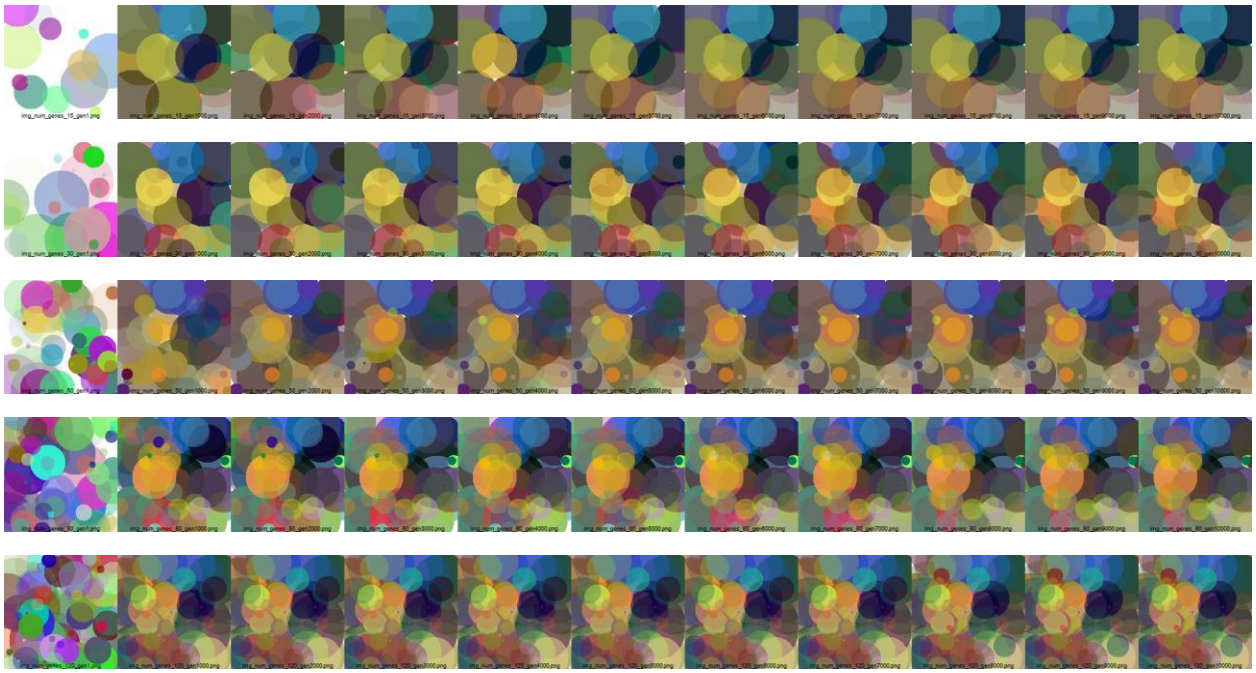

Figure 6 Images per Generation for num_genes (15, 30, 50, 80, 120)

As we increase our genes, our result looks more similar to the exact image. This can be explained as a quality increase, and it can be compared as having more pixels in an image.

2.4. Different tm_size (2, 5, 8, 16)

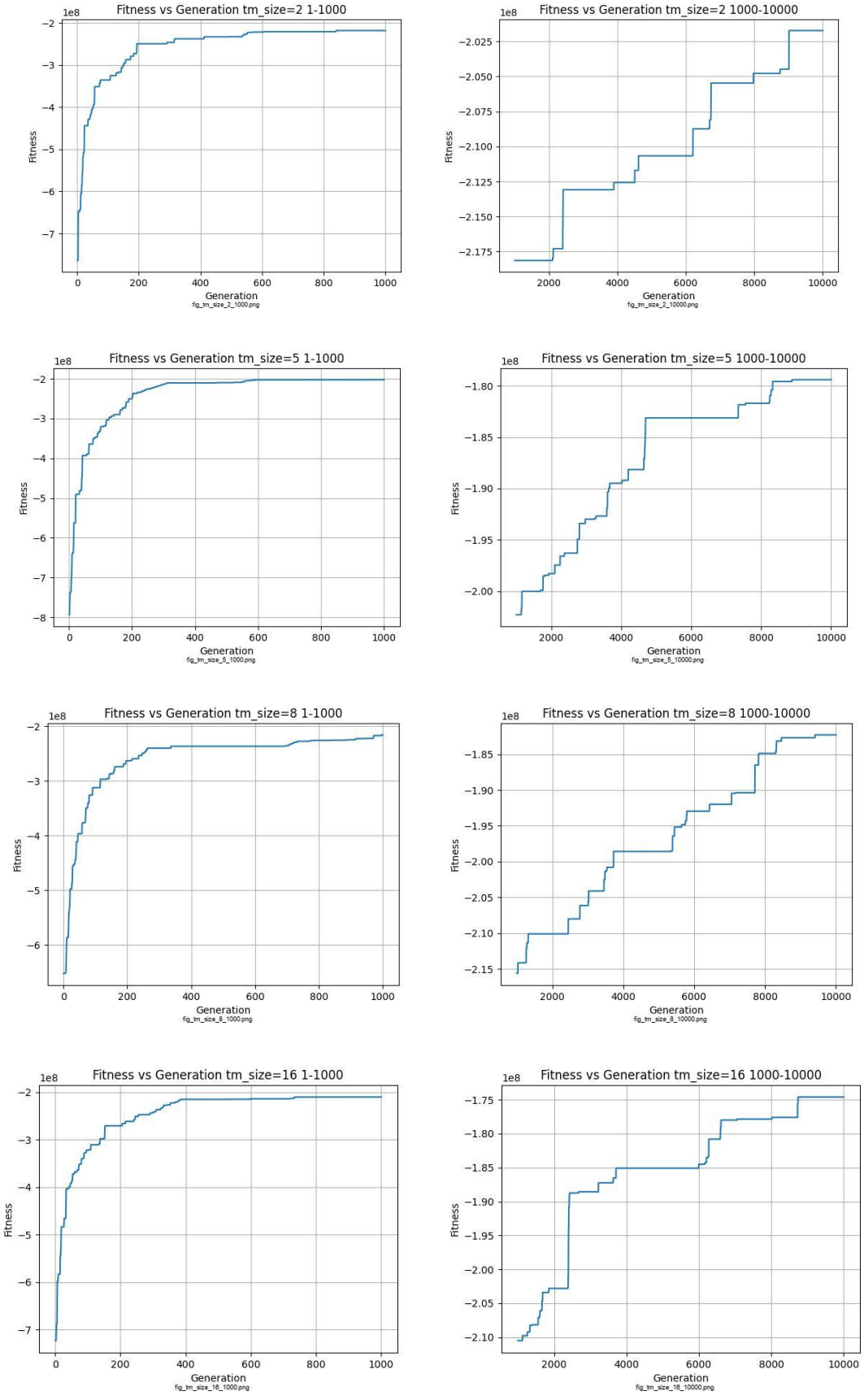

Figure 7 Fitness vs Generation Figures for tm_size (2, 5, 8, 16)

Erkan Dogan
2166262
2023-05-21

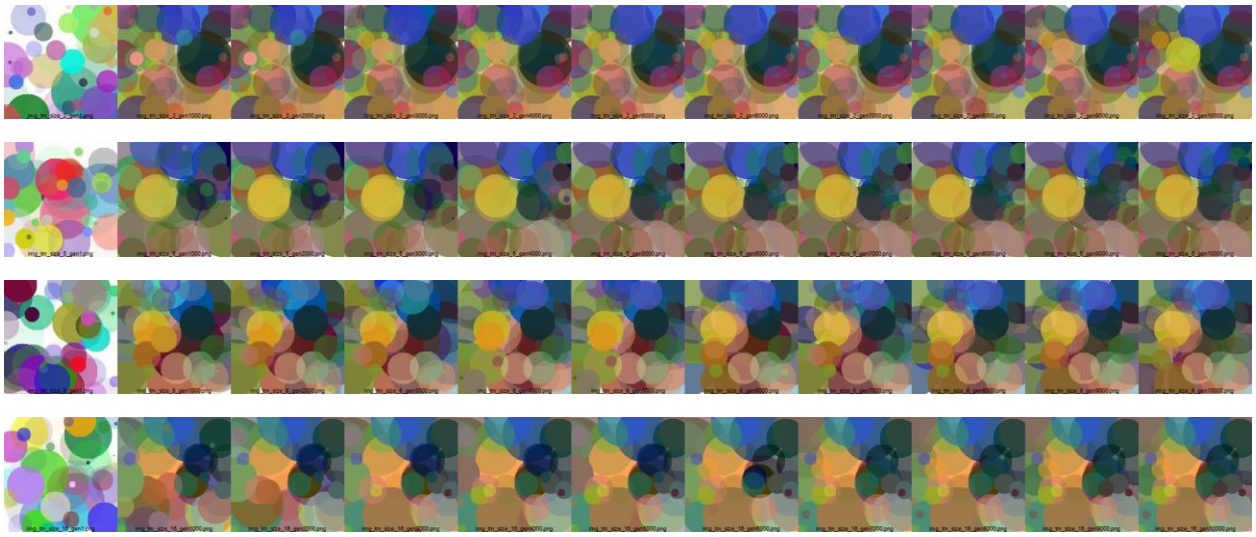

Figure 8 Images per Generation for tm_size (2, 5, 8, 16)

Increasing our tournament size let us to choose more individuals to fight and have individuals that have better fitness in our tournament pool. Due to that, our fitness value is better with increased tournament size.

Erkan Dogan
2166262
2023-05-21

2.5. Different frac_elites (0.04, 0.2, 0.35)

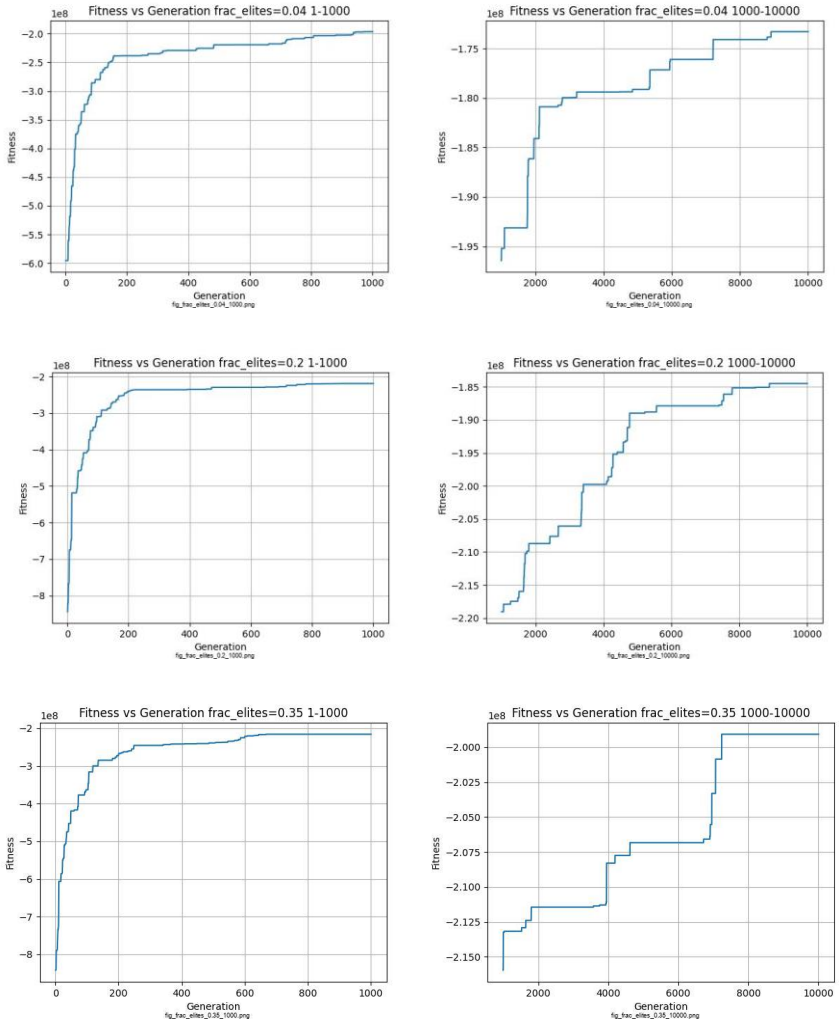

Figure 9 Fitness vs Generation Figures for frac_elites (0.04, 0.2, 0.35)

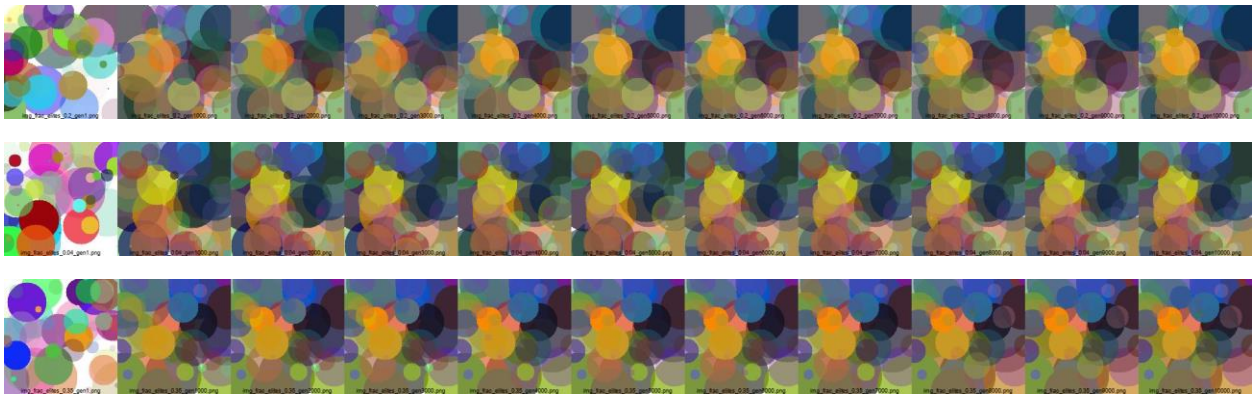

Figure 10 Images per Generation for frac_elites (0.04, 0.2, 0.35)

We see better results when our elite fraction is less. This can be explained by having a larger pool for mutation and parents. If we select low fitness levels as elites due to a large elite fraction, our results get worse. This parameter can be adaptive with generations to have better results.

2.6. Different frac_parents (0.15, 0.3, 0.6, 0.75)

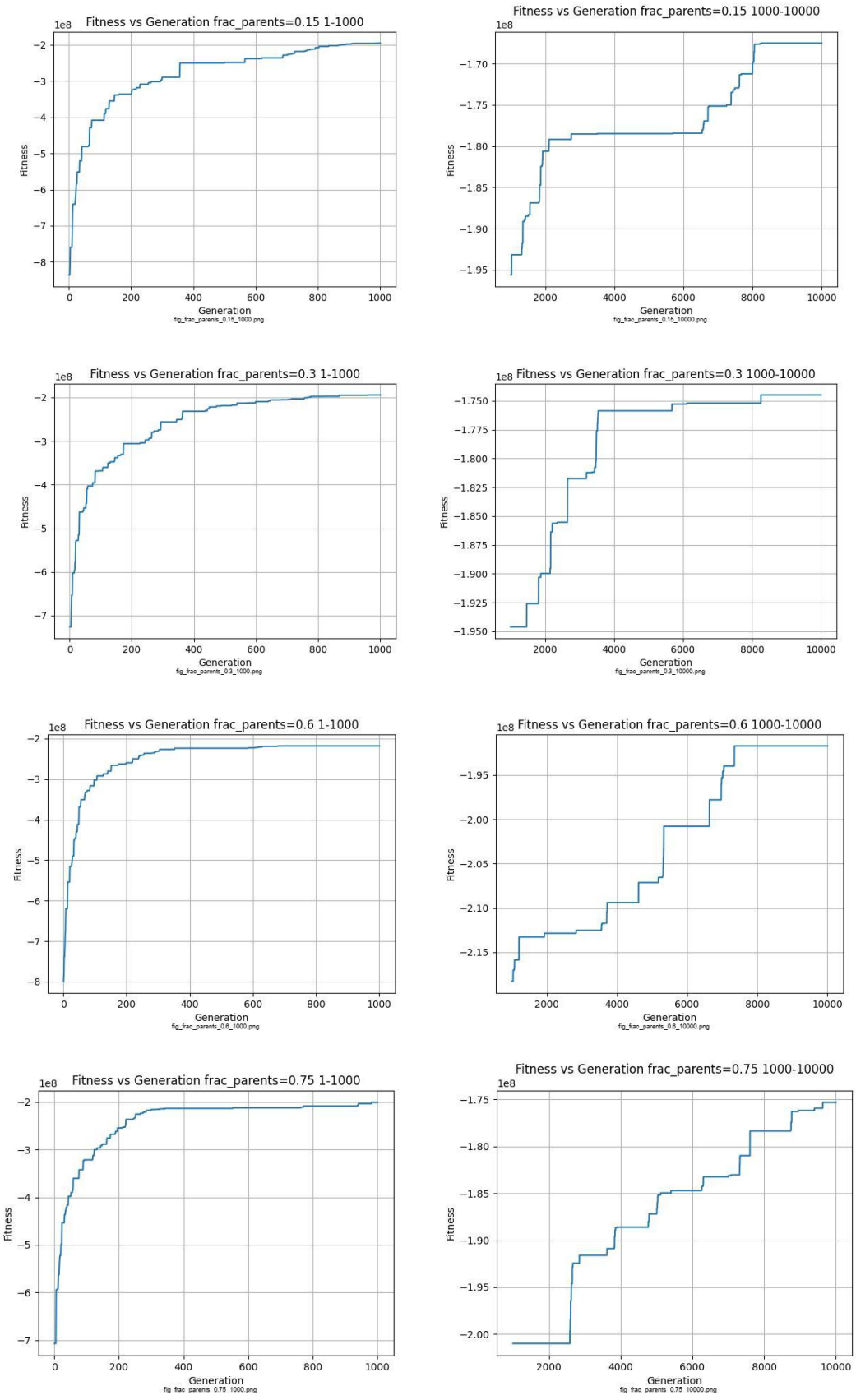

Figure 11 Fitness vs Generation Figures for frac_parents (0.15, 0.3, 0.6, 0.75)

Erkan Dogan
2166262
2023-05-21

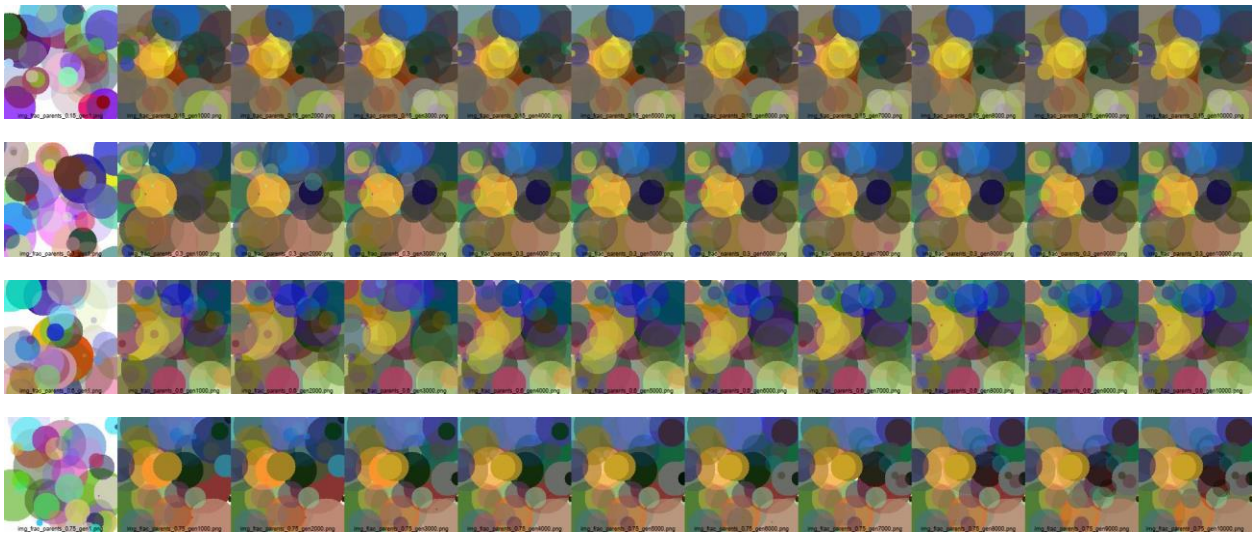

Figure 12 Images per Generation for frac_parents (0.15, 0.3, 0.6, 0.75)

We don’t see a significant change in fitness change with fraction of parents, except for 0.6, which decreases our best fitness value. In general, we can say that as we increase our parents, we are having less favorable genes in our offsprings. We can use adaptive fraction of parents according to offspring fitness levels, so that we can have better improvement in fitness levels.

2.7. Different mutation_prob (0.1, 0.2, 0.4, 0.75)

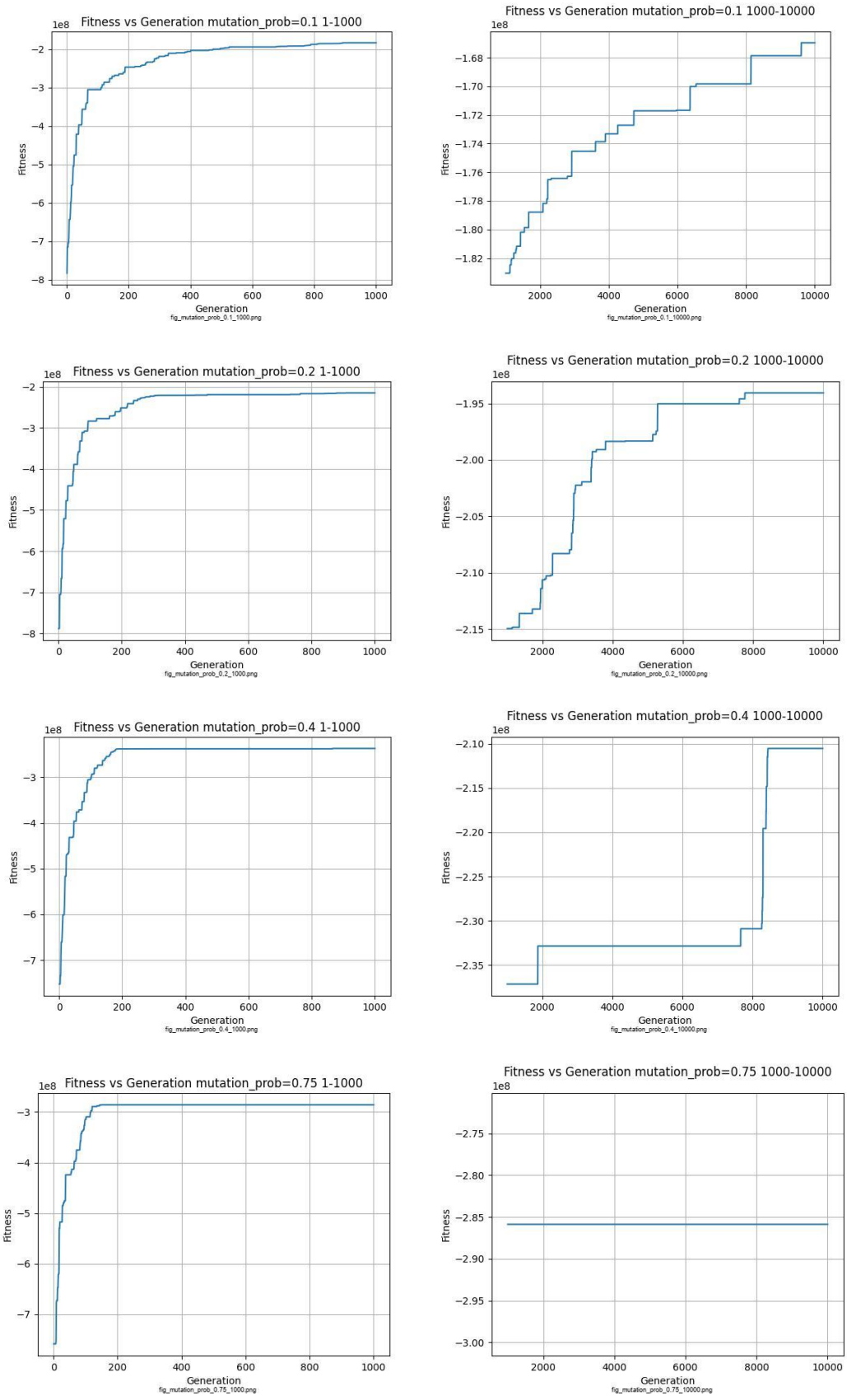

Figure 13 Fitness vs Generation Figures for mutation_prob (0.1, 0.2, 0.4, 0.75)

Erkan Dogan
2166262
2023-05-21

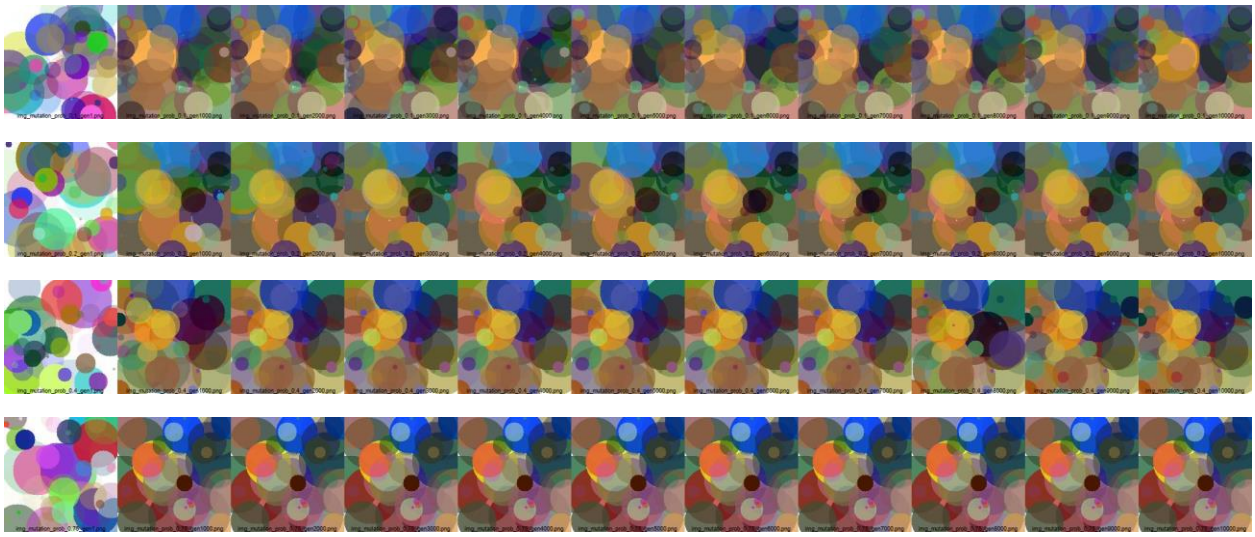

Figure 14 Images per Generation for mutation_prob (0.1, 0.2, 0.4, 0.75)

We see a significant decrease in best fitness with increasing mutation rate. This can be explained with the disruptive effect of randomness of mutations. As we get offsprings, having other individuals in population to mutate, we are having more randomness, which hinder our improvement in fitness. This parameter can be adaptive with comparison to the fitness levels before and after mutation to have better results.

Erkan Dogan
2166262
2023-05-21

2.8. Different mutation_type (“unguided”, “guided”)

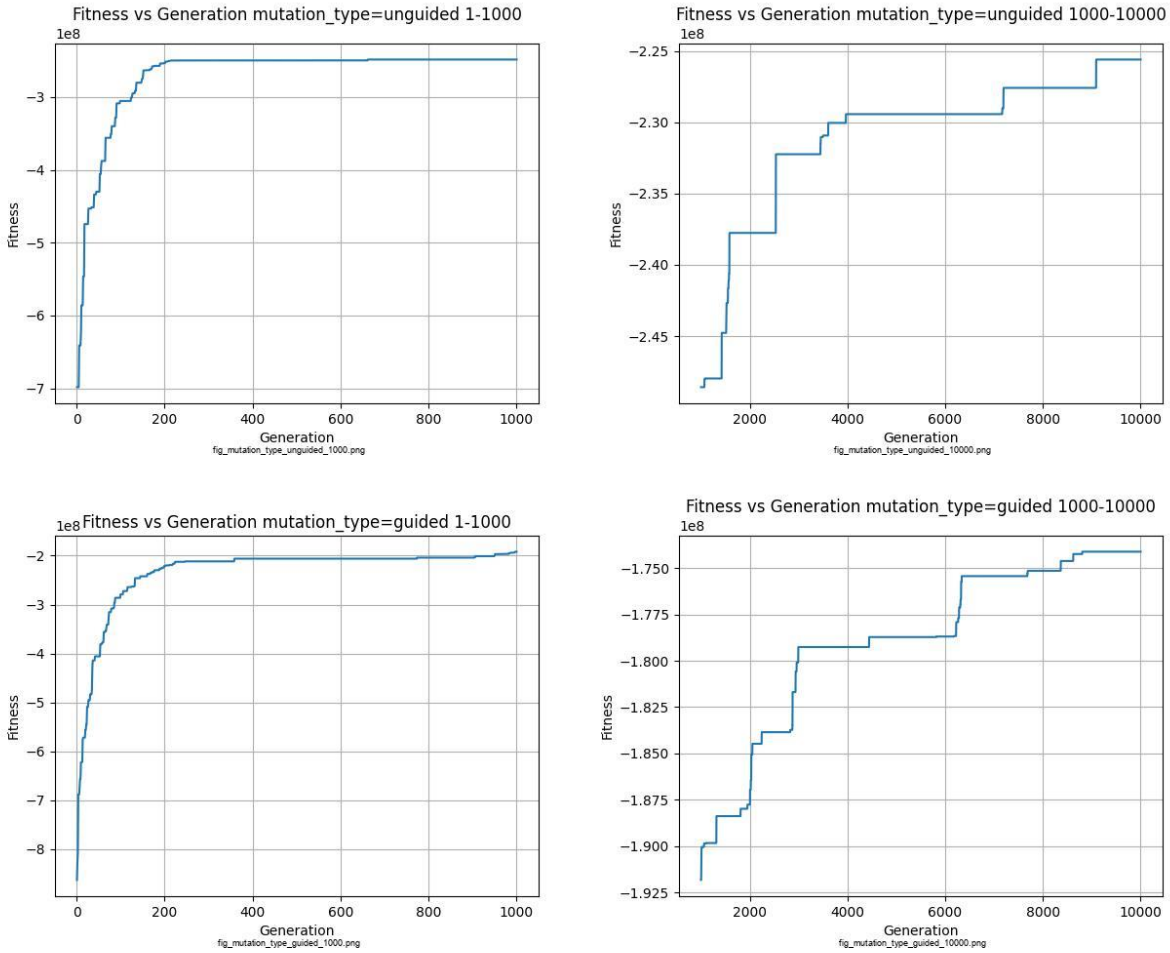

Figure 15 Fitness vs Generation Figures for mutation_type (“unguided”, “guided”)

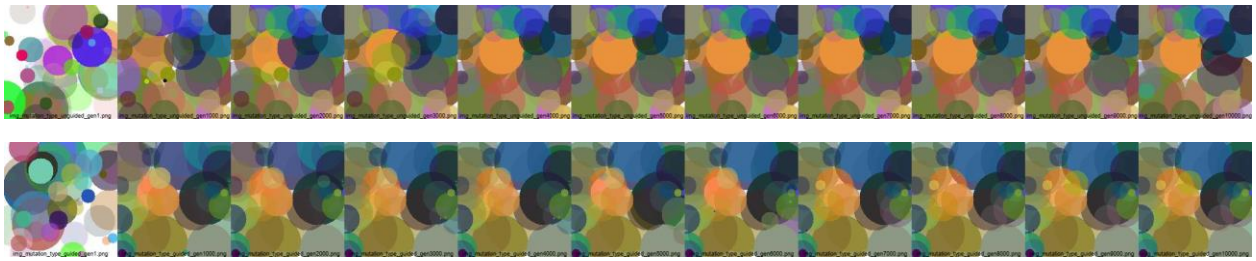

Figure 16 Images per Generation for mutation_type (“unguided”, “guided”)

We see a significant difference between unguided and guided mutation types. Guided mutation provides better results, because each generation have similar features to each other. For unguided mutation, we randomize our population for the mutation fraction, without any reference and this decreases our performance. We can utilize unguided mutation if we are similar individuals in our population frequently and if our fitness is not increasing in next generations.

Erkan Dogan
2166262
2023-05-21

3. Discussion

We can utilize adaptive mutation rate, fitness-based parent selection or fitness landscape analysis.

1. Adaptive Mutation Rate: We can start with high mutation rate in the beginning to have better individuals since we created our population randomly in the beginning. After a predefined threshold of best fitness, we can decrease the mutation rate to low levels.
    a. In the code, the mutation rate is the initial amount until 1000th generation. After that threshold, the mutation rate is decreased to 10% of its value. With this way, we jumpstart our fitness with a high mutation and then decrease the impact of mutation when it is compared to other parameters.
2. Fitness-based parent selection: Instead of using a fixed fraction for parent selection, we can assign different probabilities to parent candidates so that better parent candidates will be in our pool.
    a. In the code, instead of tournament selection, the parents are selected according to their fitness level. They have larger probabilities if their fitness are higher. With this way, instead of random tournament groups, we give more chance on better individuals.
3. Fitness landscape analysis: To increase the chances of improvement in fitness at some levels, we can change our mutation or parent parameters at those levels. By comparing a specific window of fitnesses, we can adjust our parameters to have better improvement in fitness. As we reach our threshold improvement percentage, we can decrease our parameter values to their previous value.
    a. In the code, each 100 generation is checked with the best fitness value. If the value is same after 100 generation, we trigger convergence and increase the mutation rate and fraction of parents. To escape convergence, we are looking for more diverse and improved fitnesses.

When our results are compared with default parameters, we see an improvement in generation results at the 10000th generation. This is due to our adaptive approach for our model.

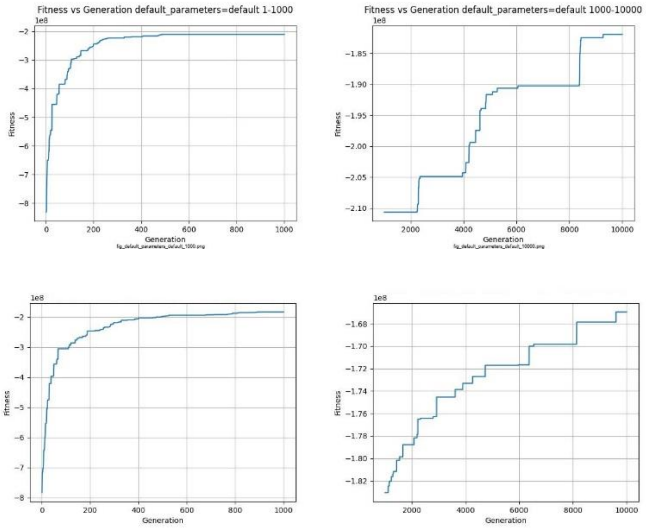

Figure 17 Fitness vs Generation Figures for default and discussion

Figure 18 Images per Generation for default and discussion

Erkan Dogan
2166262
2023-05-21

Appendix A. Code

```
import cv2
import numpy as np
import random
import math
import pandas as pd
from copy import deepcopy
import matplotlib.pyplot as plt
import time
import json

source_image = cv2.imread("painting.png")
width = source_image.shape[1]
height = source_image.shape[0]

max_radius = 45
num_generations = 10000

num_inds = 20
num_genes = 50
tm_size = 5
frac_elites = 0.2
frac_parents = 0.6
mutation_prob = 0.2
mutation_type = "guided"
default_list = [num_inds, num_genes, tm_size, frac_elites, frac_parents, mutation_prob,
mutation_type]

num_inds_list = [5, 10, 20, 40, 60]
num_genes_list = [15, 30, 50, 80, 120]
tm_size_list = [2, 5, 8, 16]
frac_elites_list = [0.04, 0.2, 0.35]
frac_parents_list = [0.15, 0.3, 0.6, 0.75]
mutation_prob_list = [0.1, 0.2, 0.4, 0.75]
mutation_type_list = ["unguided", "guided"]
parameters =
[num_inds_list,num_genes_list,tm_size_list,frac_elites_list,frac_parents_list,mutation_pr
ob_list,mutation_type_list]
names =
["num_inds","num_genes","tm_size","frac_elites","frac_parents","mutation_prob","mutation_
type"]
class Gene:
    def __init__(self, x, y, radius, R, G, B, A):
        self.x = x
        self.y = y
        self.radius = radius
        self.R = R
        self.G = G
```

```
        self.B = B
        self.A = A

class Individual:
    def __init__(self, num_genes):
        self.ID = -1  # ID of the individual
        self.chromosome = []  # List of genes representing circles
        self.fitness = -99999999999  # Fitness value of the individual
        for _ in range(num_genes):
            outside = True
            while outside:
                gene = Gene(
                    x=random.randint(0 - max_radius, width + max_radius),  # Random x-
coordinate
                    y=random.randint(0 - max_radius, height + max_radius),  # Random y-
coordinate
                    radius=random.randint(1, max_radius),  # Random radius
                    R=random.randint(0, 255),  # Random red value
                    G=random.randint(0, 255),  # Random green value
                    B=random.randint(0, 255),  # Random blue value
                    A=random.uniform(0, 1),  # Random alpha value
                )
                outside = is_outside(gene.x, gene.y, gene.radius)
            self.chromosome.append(gene)
        self.chromosome.sort(key=lambda gene: gene.radius, reverse=True)

def Population(num_individuals, num_genes):
    individuals = []  # List of individuals
    for i in range(num_individuals):
        individual = Individual(num_genes)
        individual.ID = i
        individuals.append(individual)
    individuals.sort(key=lambda ind: ind.fitness, reverse=True)
    return individuals

def evaluate_individual(individual, source_image):
    individual.chromosome.sort(key=lambda gene: gene.radius, reverse=True)
    #image = np.zeros_like(source_image)
    image = np.zeros_like(source_image, dtype=np.uint8) # Initialize image with zeros
    image.fill(255)
    for gene in individual.chromosome:
        overlay = deepcopy(image)  # Create a copy of the image

        # Extract gene attributes
        x = gene.x
        y = gene.y
        radius = gene.radius
        R = gene.R
```

```
        G = gene.G
        B = gene.B
        A = gene.A

        # Draw the circle on the overlay
        cv2.circle(overlay, (x, y), radius, (B, G, R), -1)

        # Apply alpha blending to overlay the circle on the image
        image = cv2.addWeighted(overlay, A, image, 1 - A, 0)


    # Calculate fitness value by comparing the generated image with the source image
    srcimg_img=np.subtract(np.array(source_image, dtype=np.int64), np.array(image,
dtype=np.int64))
    fitness = np.sum(-1*np.power(srcimg_img, 2))

    # Update the individual's fitness attribute
    individual.fitness = fitness

def is_outside(x, y, radius):
    outside = True
    # Check if the circle is outside the image
    # 1. inside, middle, middle
    if (x >= 0 and x <= width) and (y >= 0 and y <= height):
        outside = False
    # 2. outside, left, middle
    elif (x < 0) and (y > 0 and y < height):
        if (x + radius<0):
            outside = True
    # 3. outside, right, middle
    elif (x > width) and (y > 0 and y < height):
        if (x - radius>width):
            outside = True
    # 4. outside, bottom, middle
    elif (y < 0) and (x > 0 and x < width):
        if (y + radius<0):
            outside = True
    # 5. outside, top, middle
    elif (y > height) and (x > 0 and x < width):
        if (y - radius>height):
            outside = True
    # 6. outside, left, bottom
    elif (x < 0) and (y < 0):
        if (radius**2 < (x - 0)**2 + (y - 0)**2):
            outside = True
    # 7. outside, left, top
    elif (x < 0) and (y > height):
        if (radius**2 < (x - 0)**2 + (y - height)**2):
```

```
            outside = True
    # 8. outside, right, bottom
    elif (x > width) and (y < 0):
        if (radius**2 < (x - width)**2 + (y - 0)**2):
            outside = True
    # 9. outside, right, top
    elif (x > width) and (y > height):
        if (radius**2 < (x - width)**2 + (y - width)**2):
            outside = True
    else:
        outside = False
    return outside

def selection(population, elites_IDs, num_parents):
    selected_parents = []
    selected_ids = []
    parent_candidates = deepcopy([ind for ind in population if ind.ID not in elites_IDs])
    parent_candidates_ids = [ind.ID for ind in parent_candidates]
    # Calculate selection probabilities based on fitness values
    if discussion == True:
        fitness_values = [ind.fitness for ind in parent_candidates]
        fitness_sum = sum(fitness_values)
        selection_probs = [fitness / fitness_sum for fitness in fitness_values]
        for _ in range(num_parents):
            best_cand = np.random.choice(parent_candidates, p=selection_probs)
            best_cand_id = best_cand.ID
            selected_ids.append(best_cand_id)
            parent_candidates_ids.remove(best_cand_id)
            parent_candidates.remove(best_cand)
            fitness_values = [ind.fitness for ind in parent_candidates]
            fitness_sum = sum(fitness_values)
            selection_probs = [fitness / fitness_sum for fitness in fitness_values]

        selected_parents = deepcopy([ind for ind in population if ind.ID in
selected_ids])

    else:
        for _ in range(num_parents):
            best_cand = random.choice(parent_candidates)
            best_cand_id = best_cand.ID
            for i in range(tm_size):
                cand = random.choice(parent_candidates)
                if cand.fitness > best_cand.fitness:
                    best_cand_id = cand.ID
                    best_cand = cand
            selected_ids.append(best_cand_id)
            parent_candidates_ids.remove(best_cand_id)
            parent_candidates.remove(best_cand)
```

```
        selected_parents = deepcopy([ind for ind in population if ind.ID in
selected_ids])
    return selected_parents, selected_ids

def crossover(parent1, parent2):
    chromosome_length = len(parent1.chromosome)
    cand1 = Individual(chromosome_length)
    cand2 = Individual(chromosome_length)
    # Perform crossover
    for gene in range(chromosome_length):
        coinflip = random.randint(0, 1)
        if coinflip == 0:
            cand1.chromosome[gene] = deepcopy(parent1.chromosome[gene])
            cand2.chromosome[gene] = deepcopy(parent2.chromosome[gene])

        else:
            cand1.chromosome[gene] = deepcopy(parent2.chromosome[gene])
            cand2.chromosome[gene] = deepcopy(parent1.chromosome[gene])

    evaluate_individual(cand1, source_image)
    evaluate_individual(cand2, source_image)
    inds = [cand1, cand2, parent1, parent2]
    inds.sort(key=lambda ind: ind.fitness, reverse=True)
    child1 = deepcopy(inds[0])
    child2 = deepcopy(inds[1])
    child1.ID = parent1.ID
    child2.ID = parent2.ID
    return child1, child2

def mutate(individual):
    while True:
        prev_fitness = individual.fitness
        temp_ind = deepcopy(individual)
        for gene in range(len(temp_ind.chromosome)):
            if random.random() < mutation_prob:
                if mutation_type == "unguided":
                    mutate_unguided(temp_ind.chromosome[gene])
                elif mutation_type == "guided":
                    mutate_guided(temp_ind.chromosome[gene])
        evaluate_individual(temp_ind, source_image)
        if temp_ind.fitness > prev_fitness:
            individual.chromosome = deepcopy(temp_ind.chromosome)
            individual.fitness = temp_ind.fitness
            break
        else:
            break
    return individual
```

Erkan Dogan
2166262
2023-05-21

```
def mutate_unguided(gene):
        outside = True
        while outside:
            gene.x = random.randint(0 - max_radius, width + max_radius)
            gene.y = random.randint(0 - max_radius, height + max_radius)
            gene.radius = random.randint(1, max_radius)
            outside = is_outside(gene.x, gene.y, gene.radius)
        gene.R = random.randint(0, 255)
        gene.G = random.randint(0, 255)
        gene.B = random.randint(0, 255)
        gene.A = random.uniform(0, 1)

def mutate_guided(gene):
    # Mutate the gene attributes without exceeding the boundaries
    x = gene.x
    y = gene.y
    radius = gene.radius
    R = gene.R
    G = gene.G
    B = gene.B
    A = gene.A
    temp_x = x
    temp_y = y
    temp_radius = radius
    outside = True
    while outside:
        temp_x = x + random.randint(-width // 4, width // 4)
        if (temp_x < x):
            temp_x = max(temp_x, 0 - max_radius)
        else:
            temp_x = min(temp_x, width + max_radius)

        temp_y = y + random.randint(-height // 4, height // 4)
        if (temp_y < y):
            temp_y = max(temp_y, 0 - max_radius)
        else:
            temp_y = min(temp_y, height + max_radius)

        temp_radius = radius + random.randint(-10, 10)
        if (temp_radius < 0):
            temp_radius = 1
        else:
            temp_radius = min(temp_radius, max_radius)
        outside = is_outside(temp_x, temp_y, temp_radius)

    gene.x = temp_x
    gene.y = temp_y
    gene.radius = temp_radius
```

```
    R = gene.R + random.randint(-64, 64)
    if (R >= 0 and R <= 255):
        gene.R = R
    elif (R < 0):
        gene.R = 0
    elif (R > 255):
        gene.R = 255

    G = gene.G + random.randint(-64, 64)
    if (G >= 0 and G <= 255):
        gene.G = G
    elif (G < 0):
        gene.G = 0
    elif (G > 255):
        gene.G = 255

    B = gene.B + random.randint(-64, 64)
    if (B >= 0 and B <= 255):
        gene.B = B
    elif (B < 0):
        gene.B = 0
    elif (B > 255):
        gene.B = 255

    A = gene.A + random.uniform(-0.25, 0.25)
    if (A >= 0 and A <= 1):
        gene.A = A
    elif (A < 0):
        gene.A = 0
    elif (A > 1):
        gene.A = 1

def draw_circle(individual, name, value, generation):
    individual.chromosome.sort(key=lambda gene: gene.radius, reverse=True)
    image = np.zeros_like(source_image, dtype=np.uint8) # Initialize image with zeros
    image.fill(255)
    for gene in individual.chromosome:
        overlay = deepcopy(image)
        x = gene.x
        y = gene.y
        radius = gene.radius
        color = (gene.B, gene.G, gene.R)
        A = gene.A
        thickness = -1  # Filled circle

        cv2.circle(overlay, (x, y), radius, color, thickness)
        cv2.addWeighted(overlay, A, image, 1 - A, 0, image)
```

```
    cv2.imwrite(f"C:/Users/erkan/Desktop/EE/e2022_2/EE449/2023/HW2/Code/{name}/img_{name}
_{value}_gen{generation}.png", image)

def draw_fig(fitness_list, name, value):
    part1 = int(num_generations/10)
    part2 = num_generations
    parts = [part1, part2]
    print(len(fitness_list[0:10]))
    print(len(fitness_list[10:]))
    for part in parts:
        if part == part1:
            generations = range(1, part1+1)
            plt.plot(generations, fitness_list[0:part1])
            plt.title(f'Fitness vs Generation {name}={value} {1}-{part1}')
        else:
            generations = range(part1+1, part2+1)
            plt.plot(generations, fitness_list[part1:])
            plt.title(f'Fitness vs Generation {name}={value} {part1}-{part2}')
        plt.xlabel('Generation')
        plt.ylabel('Fitness')
        plt.grid()
        plt.savefig(f"C:/Users/erkan/Desktop/EE/e2022_2/EE449/2023/HW2/Code/{name}/fig_{n
ame}_{value}_{part}.png")
        plt.close()

def genetic_algorithm(name, item):
    global num_inds, num_genes, tm_size, frac_elites, frac_parents, mutation_prob,
mutation_type, discussion
    # Step 6.1: Initialize the population with random individuals
    population = Population(num_inds, num_genes)
    fitness_list = []
    population_sorted = []
    mutation_prob_low = mutation_prob / 10.0
    mutation_threshold = 0.1 * num_generations
    # Step 6.2: Iterate over the specified number of generations
    for generation in range(num_generations):
        if(generation == mutation_threshold and discussion == True):
            mutation_prob = mutation_prob_low
            print(f"Mutation probability decreased to {mutation_prob} at threshold
{generation}")

        if (generation+1) % (num_generations/100) == 0:
            print(f"Generation {generation+1}/{num_generations}, Best Fitness:
{population[0].fitness}, Parameters: num_inds={num_inds}, num_genes={num_genes},
tm_size={tm_size}, frac_elites={frac_elites}, frac_parents={frac_parents},
mutation_prob={mutation_prob}, mutation_type={mutation_type}")
            print(f"Time: parent={round(parent_time,3)},
crossover={round(crossover_time,3)}, mutation={round(mutation_time,3)}")
```

```
            if fitness_list[-1] == fitness_list[-99]:
                mutation_prob = mutation_prob_list[-1]
                frac_parents = frac_parents_list[-1]
                print(f"Convergence detected, increasing mutation_prob={mutation_prob}
and frac_parents={frac_parents}")
            else:
                mutation_prob = default_list[5]
                frac_parents = default_list[4]
                print("No convergence detected, resetting mutation_prob and
frac_parents")
        if (generation+1) % (num_generations/10) == 0 or (generation == 0):
            draw_circle(population[0], name, item, generation+1)

        # Step 6.3: Evaluate all individuals in the population
        for individual in population:
            evaluate_individual(individual, source_image)
        population.sort(key=lambda ind: ind.fitness, reverse=True)

        fitness_list.append(population[0].fitness)

        # Step 6.4: Select elites to directly pass to the next generation
        num_elites = int(frac_elites * num_inds)
        elites = deepcopy(population[:num_elites])
        elites_IDs = [ind.ID for ind in elites]

        # Step 6.5: Perform tournament selection to select parents for crossover
        num_parents = int(frac_parents * num_inds)
        if num_parents % 2 != 0:
            num_parents += 1
        parent_start = time.time()
        parents, parents_IDs = selection(population, elites_IDs, num_parents)
        parent_end = time.time()
        parent_time = parent_end - parent_start

        nonparents_IDs = [ind.ID for ind in population if ind.ID not in parents_IDs]
        nonparents_IDs = [npID for npID in nonparents_IDs if npID not in elites_IDs]
        nonparents = deepcopy([ind for ind in population if ind.ID in nonparents_IDs])

        # Step 6.6: Apply crossover to create new individuals
        offspring = []
        crossover_start = time.time()
        for i in range(0, num_parents, 2):
            # Perform crossover on adjacent parents
            parent1 = parents.pop(random.randint(0,len(parents)-1))
            parent2 = parents.pop(random.randint(0,len(parents)-1))
            #parent1 = parents.pop(0)
            #parent2 = parents.pop(0)
            # parent2 = parents[i+1]
```

```
            child1, child2 = crossover(parent1, parent2)
            offspring.extend([child1, child2])
        crossover_end = time.time()
        crossover_time = crossover_end - crossover_start
        # Step 6.7: Perform mutation on some individuals
        mutation_candidates = deepcopy(offspring + nonparents)
        mutation_results = []
        mutation_start = time.time()
        for individual in mutation_candidates:
            individual = mutate(individual)
            mutation_results.append(individual)
        mutation_end = time.time()
        mutation_time = mutation_end - mutation_start
        # Step 6.7: Update the population with elites, mutation results
        population = deepcopy(elites + mutation_results)

        # Sort the final population based on fitness values in descending order
        population.sort(key=lambda ind: ind.fitness, reverse=True)
        # Print population IDs

        #print(f"Generation {generation+1}/{num_generations}, times:
parent={parent_time}, crossover={crossover_time}, mutation={mutation_time}")
    # Return the final population
    return population, fitness_list
discussion = False
print(f"Running for default parameters")
population, fitness_list = genetic_algorithm("default_parameters", "default")
best_individual = population[0]
draw_fig(fitness_list, "default_parameters", "default")

discussion = True
print(f"Running for default parameters, discussion enabled")
population, fitness_list = genetic_algorithm("default_parameters", "default")
best_individual = population[0]
draw_fig(fitness_list, "default_parameters", "default")


for param, name in zip(parameters,names):
    num_inds, num_genes, tm_size, frac_elites, frac_parents, mutation_prob, mutation_type
= default_list
    for item in param:
        if name == "num_inds":
            num_inds = item
        elif name == "num_genes":
            num_genes = item
        elif name == "tm_size":
            tm_size = item
        elif name == "frac_elites":
```

```
            frac_elites = item
        elif name == "frac_parents":
            frac_parents = item
        elif name == "mutation_prob":
            mutation_prob = item
        elif name == "mutation_type":
            mutation_type = item
        print(f"Running for num_inds={num_inds}, num_genes={num_genes},
tm_size={tm_size}, frac_elites={frac_elites}, frac_parents={frac_parents},
mutation_prob={mutation_prob}, mutation_type={mutation_type}")
        population, fitness_list = genetic_algorithm(name, item)

        # Find the best individual from the final population
        best_individual = population[0]

        # Plot the fitness graph
        draw_fig(fitness_list, name, item)

# Done
print("Done")
```