EE449 HW2

# 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.

## Default Parameters

A picture containing text, line, diagram, plot

Description automatically generated

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.

## Different num\_inds (5, 10, 20, 40, 60)

A picture containing text, line, diagram, plot

Description automatically generated

A picture containing text, line, diagram, plot

Description automatically generated

A picture containing text, line, diagram, plot

Description automatically generated

A picture containing text, line, diagram, plot

Description automatically generated

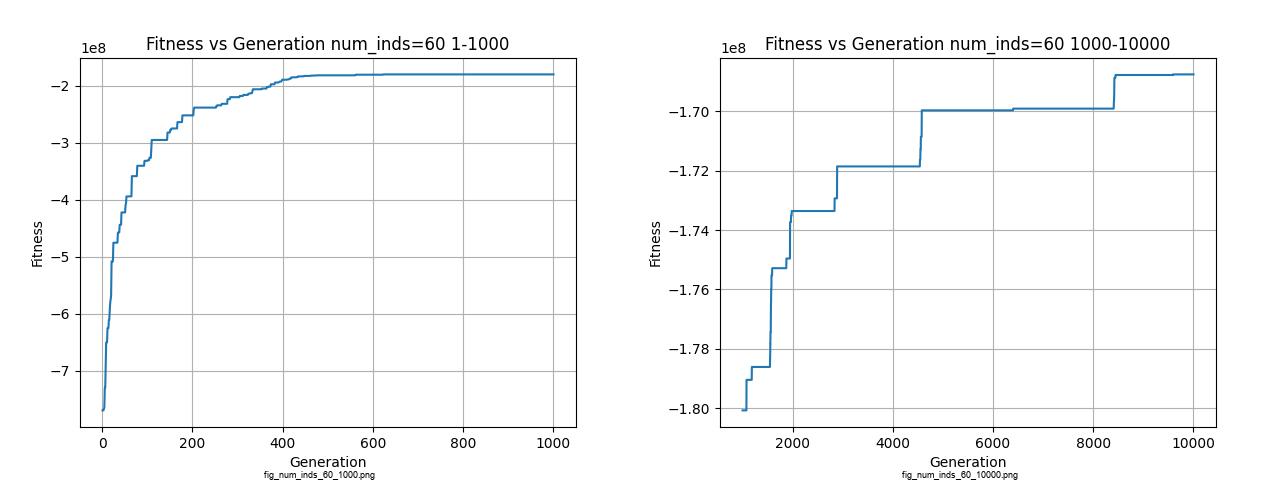


Figure 3 Fitness vs Generation Figures for num\_inds (5, 10, 20, 40, 60)











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.

## Different num\_genes (15, 30, 50, 80, 120)

A picture containing text, diagram, line, plot

Description automatically generated

A picture containing text, diagram, line, plot

Description automatically generated

A picture containing text, line, diagram, plot

Description automatically generated

A picture containing text, line, diagram, plot

Description automatically generated

A picture containing text, line, receipt, plot

Description automatically generated

Figure 5 Fitness vs Generation Figures for num\_genes (15, 30, 50, 80, 120)











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.

## Different tm\_size (2, 5, 8, 16)

A picture containing text, line, diagram, plot

Description automatically generated

A picture containing text, diagram, line, plot

Description automatically generated

A picture containing text, line, diagram, plot

Description automatically generated

A picture containing text, line, diagram, plot

Description automatically generated

Figure 7 Fitness vs Generation Figures for tm\_size (2, 5, 8, 16)









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.

## Different frac\_elites (0.04, 0.2, 0.35)

A picture containing text, diagram, line, plot

Description automatically generated

A picture containing text, diagram, line, plot

Description automatically generated

A picture containing text, diagram, line, plot

Description automatically generated

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.

## Different frac\_parents (0.15, 0.3, 0.6, 0.75)

A picture containing text, line, diagram, plot

Description automatically generated

A picture containing text, line, diagram, plot

Description automatically generated

A picture containing text, diagram, line, plot

Description automatically generated

A picture containing text, diagram, line, plot

Description automatically generated

Figure 11 Fitness vs Generation Figures for frac\_parents (0.15, 0.3, 0.6, 0.75)









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.

## Different mutation\_prob (0.1, 0.2, 0.4, 0.75)

A picture containing text, line, plot, diagram

Description automatically generated

A picture containing text, line, diagram, plot

Description automatically generated

A picture containing text, line, diagram, receipt

Description automatically generated

A picture containing text, line, receipt, plot

Description automatically generated

Figure 13 Fitness vs Generation Figures for mutation\_prob (0.1, 0.2, 0.4, 0.75)









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.

## Different mutation\_type (“unguided”, “guided”)

A picture containing text, diagram, line, plot

Description automatically generated

A picture containing text, line, diagram, plot

Description automatically generated

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.

# 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.
   1. 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.
   1. 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.
   1. 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.

A picture containing text, line, diagram, plot

Description automatically generated

A picture containing text, line, diagram, plot

Description automatically generated

Figure 17 Fitness vs Generation Figures for default and discussion





Figure 18 Images per Generation for default and discussion

# 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\_prob\_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

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\_{name}\_{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")