Uppsala University

Modelling Complex Systems

1MA256

Lab6 report

Author:
Kristensen.Samuel
Place of publication: Uppsala

May 29, 2024

1 Task 1: Painter robot

For this task we will imagine we have a painter robot. The robot will then paint the floor of a room. To make it interesting, the painter starts at a random place in the room, and paints continuously.The robot has exactly enough paint to cover the floor. This means that it is wasteful to visit the same spot more than once or to stay in the same place. To see if there is an optimal set of rules for the painter to follow, we will create a genetic algorithm. The basis for this task is the code from **painter_play.py**.

The code will take in two inputs:

- *A chromosome*: A 1 × 54 array of numbers between 0 and 3 that shows how to respond (0: no turn, 1: turn left, 2: turn right, 3: random turn left/right) in each of the 54 possible states. The state is the state of the squares forward/left/right and the current square. Let [c, f, l, r] denote states of the current square, forward square, left square and right square respectively.
- *An environment*: A 2D array representing a rectangular room. Empty (paintable) space is represented by a zero, while furniture or extra walls are represented by ones. Outside walls are automatically created **by painter_play()**.

We will create 100 random chromosomes in a 100 × 54 matrix, as well as a 30 × 60 empty room. We then create a genetic algorithm to evolve this population over 200 generations, playing each chromosome 5 times and storing the chromosomes average efficiency as the fitness. The rules for picking the next generation are that first the parents are chosen based on the fitness score, higher fitness means higher probability to be chosen for the next generation. Then the crossover for the offspring happens, this is randomized with a set percentage of the population being crossovers and the crossover is a two-point crossover from the parents. Lastly the chromosomes mutates with a mutation rate of 0.005 per locus per generation.

We start with plotting the average fitness for each generation by taking the sum of the finesses for each chromosome and dividing it by the number of chromosomes.

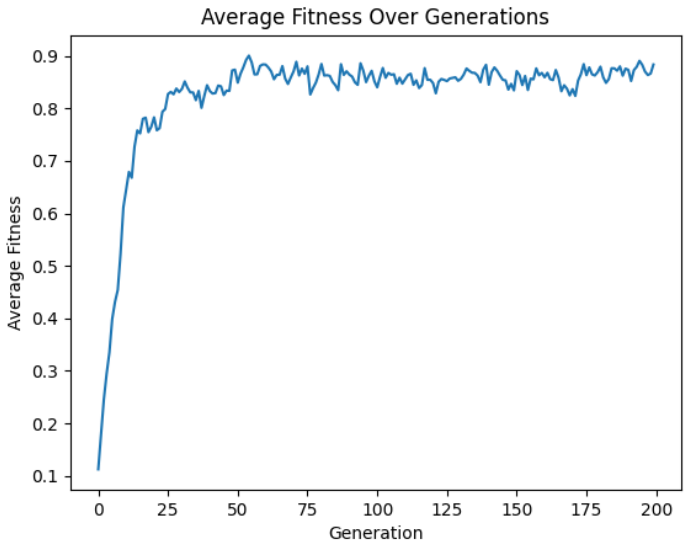

Figure 1: Average fitness for the chromosomes over generations

We can see from figure 1 that the average fitness improves very fast in the beginning and then stays at around 0.85. The reason that it does not reach 1 is probably since the chromosomes are initialized at random locations each time which makes it almost impossible for them to create one set of rules that works from all locations. That together with the fact that the chromosomes are mutated randomly can cause the chromosome set to be less effective.

We can also show the trajectory from the best chromosome in the last generation (or at least a example trajectory since the chromosome will be initialized somewhere else in the room).

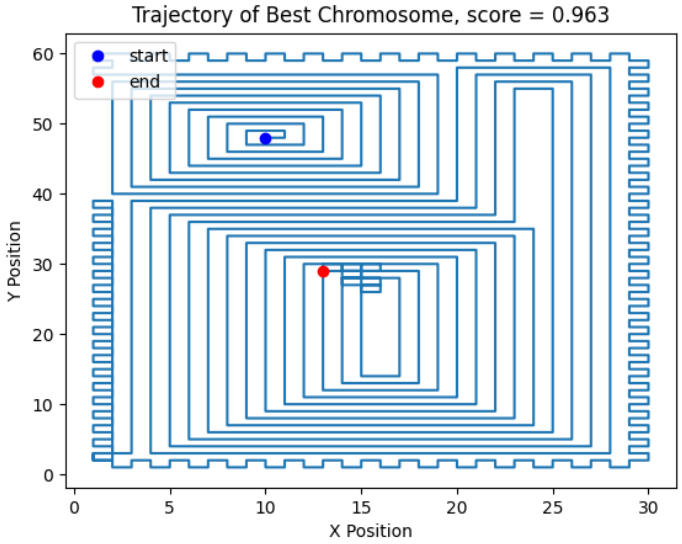

Figure 2: Example trajectory for the best chromosome

Figure 2 shows an example trajectory for the best chromosome from the last generation. The blue dot is the starting point and the red dot is the endpoint for that chromosome. For this particular example the chromosome scored a 0.963.

Lastly we can see how the locuses for all the chromosomes of the last generation are.

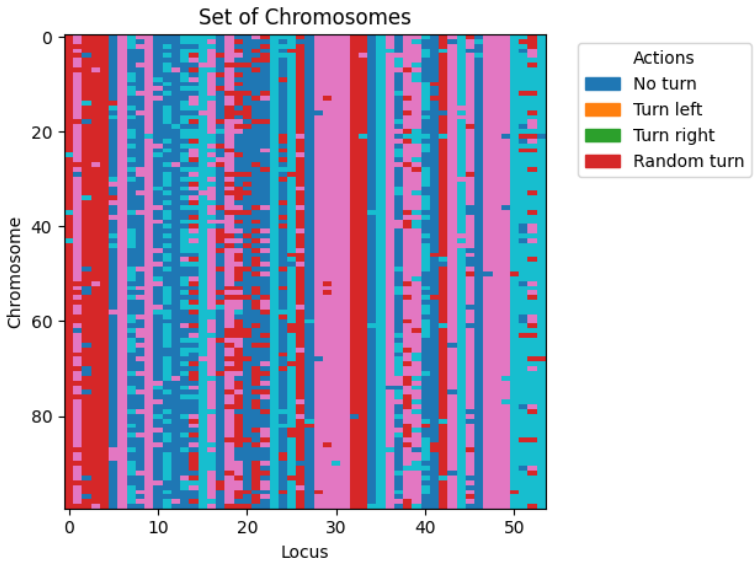

Figure 3: Set of chromosomes for the last generation

Figure 3 shows the locus for each chromosome in the last generation (slight mismatch with the colours but couldn't seem to get it right). We see that the behaviours of the last generation are fairly close to each other with parts of the locus almost being the same colour.

2 Task 2: Fractals

For task 2 we write a code that does the following:

1. Take any three points in a plane to form a triangle.
2. Randomly select any point inside the triangle and consider that your current position.
3. Randomly select any one of the three vertex points (with some probabilities $p_1 + p_2 + p_3 = 1$).
4. Move half the distance from your current position to the selected vertex.
5. After first hundred steps, start plotting the current position.
6. Repeat from step 3).

If we set that $p_1 = p_2 = p_3$ we get the image

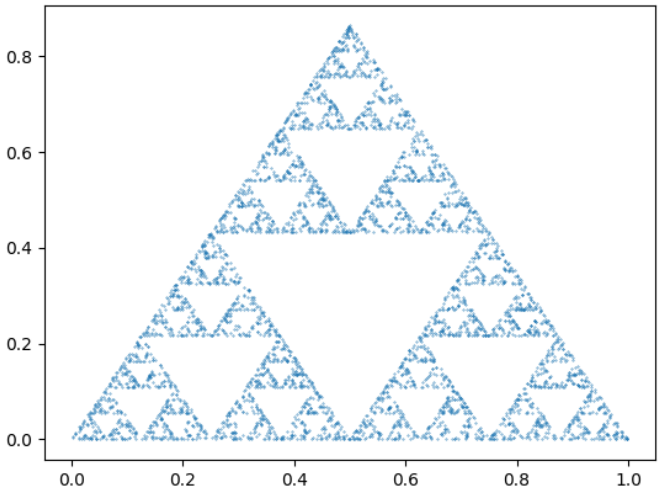

Figure 4: $p_1 = p_2 = p_3$

The Sierpiński triangle. Now lets change so that there is not equal probability to choose vertex points, for example $p_1 = p_2 = 0.1, p_3 = 0.8$

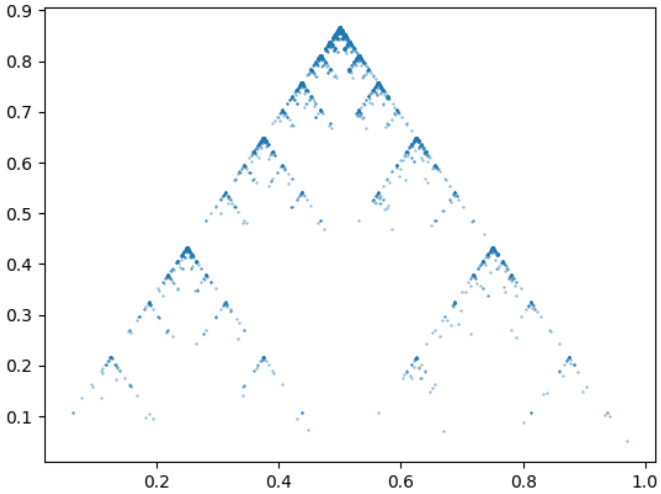

Figure 5: $p_1 = p_2 = 0.1, p_3 = 0.8$

We can see that the pattern remains but the third vortex have a much higher chance of getting chosen and thus the figure looks like it "fades" towards the bottom. The same thing happens if we let one of the other vertices have higher probability with that side being more predominant.

Iterated Function System Formulation:

Given the vertices $A = (x_1, y_1)$, $B = (x_2, y_2)$ and $C = (x_3, y_{3)}$, and their associated probabilities p_A, p_B, and p_c, the iterated function system consists of three contraction mappings:

Let $\mathbf{X}$=(x,y) be the current position.

- $f_A(\mathbf{X}) = \frac{1}{2}\mathbf{X} + \frac{1}{2}$A, with probability p_A.
- $f_B(\mathbf{X}) = \frac{1}{2}\mathbf{X} + \frac{1}{2}$B, with probability p_B.
- $f_C(\mathbf{X}) = \frac{1}{2}\mathbf{X} + \frac{1}{2}$C, with probability p_C.

Where $p_A + p_B + p_C = 1$.

3 Codes

3.1 Code for Task 1

```
import numpy as np
import random
import matplotlib.pyplot as plt
from painter_play import painter_play
from matplotlib.animation import FuncAnimation
import matplotlib.patches as mpatches

PLOT = 1
ANIMATE = 0

def initialize_population(pop_size, chromosome_length):
    '''Initializes the population'''
    return np.random.randint(0, 4, (pop_size, chromosome_length))

def evaluate_population(population, room):
    '''Evaluates the population, takes the average fitness for 5 tests for each
       cromosone'''
    nr_tests = 5
    fitness = np.zeros(population.shape[0])
    for i, chromosome in enumerate(population):
        tot_score = 0
        for _ in range(nr_tests):
            score, _, _ = painter_play(chromosome, room)
            tot_score += score
        fitness[i] = tot_score/nr_tests
    return fitness

def select_parents(population, fitness):
    '''Selects the parents based on probability, higher fitness gets higher
       probability'''
    idx = np.random.choice(np.arange(len(population)), size=len(population),
       p=fitness/fitness.sum())
    return population[idx]

def crossover(parents, crossover_rate=0.5):
    '''Does a 2 point crossovers for "crossover_rate" of the population, the
       rest reamins the same"'''
    offspring = np.empty(parents.shape)
    for k in range(0, len(parents), 2):
        parent1, parent2 = parents[k], parents[k+1]
        if np.random.rand() < crossover_rate:
            pt1, pt2 = sorted(random.sample(range(len(parent1)), 2))
            offspring[k, :pt1] = parent1[:pt1]
            offspring[k, pt1:pt2] = parent2[pt1:pt2]
            offspring[k, pt2:] = parent1[pt2:]
            offspring[k+1, :pt1] = parent2[:pt1]
            offspring[k+1, pt1:pt2] = parent1[pt1:pt2]
```

```
            offspring[k+1, pt2:] = parent2[pt2:]
        else:
            offspring[k], offspring[k+1] = parent1, parent2
    return offspring

def mutate(offspring, mutation_rate=0.005):
    '''Mutates the population with a 0.005 mutation rate per locus'''
    for chromosome in offspring:
        for i in range(len(chromosome)):
            if np.random.rand() < mutation_rate:
                chromosome[i] = np.random.randint(0, 4)
    return offspring

def genetic_algorithm(pop_size, chromosome_length, generations, room):
    '''Runs the generic algorithm, initalizes population and simulates for
        given number of generations'''
    population = initialize_population(pop_size, chromosome_length)
    average_fitness_over_generations = []

    for i in range(generations):
        print("Gen " + str(i))
        fitness = evaluate_population(population, room)
        print("Average fitness: " + str(np.average(fitness)))
        average_fitness_over_generations.append(np.average(fitness))

        parents = select_parents(population, fitness)
        offspring = crossover(parents)
        population = mutate(offspring)

    best_chromosome = population[np.argmax(fitness)]
    return best_chromosome, average_fitness_over_generations, population

# Parameters
pop_size = 100
chromosome_length = 54
generations = 200
room = np.zeros((30, 60))

# Run Genetic Algorithm
best_chromosome, fitness_over_time, population = genetic_algorithm(pop_size,
    chromosome_length, generations, room)

if PLOT:
    # Plot fitness over generations
    plt.plot(fitness_over_time)
    plt.xlabel('Generation')
    plt.ylabel('Average Fitness')
    plt.title('Average Fitness Over Generations')
    plt.show()

    # Plot example trajectory of the best chromosome
```

```
    score, xpos, ypos = painter_play(best_chromosome, room)

    plt.plot(xpos, ypos)
    plt.plot(xpos[0], ypos[0], 'o', color='blue', label='start')
    plt.plot(xpos[-1], ypos[-1], 'o', color='red', label='end')
    plt.xlabel('X Position')
    plt.ylabel('Y Position')
    plt.title('Trajectory of Best Chromosome, score = {:.3f}'.format(score))
    plt.legend()

    fig, ax = plt.subplots()

    # Plot the chromosomes
    ax.imshow(population, cmap='tab10', aspect='auto')

    # Set labels and title
    ax.set_xlabel('Locus')
    ax.set_ylabel('Chromosome')
    ax.set_title('Set of Chromosomes')

    # Create custom legend
    colors = ['tab:blue', 'tab:orange', 'tab:green', 'tab:red'] # Colors in
        'tab10' colormap for values 0, 1, 2, 3
    labels = ['No turn', 'Turn left', 'Turn right', 'Random turn']
    patches = [mpatches.Patch(color=colors[i], label=labels[i]) for i in
        range(len(labels))]
    ax.legend(handles=patches, title='Actions', bbox_to_anchor=(1.05, 1),
        loc='upper left')
    plt.tight_layout() # Adjust layout to fit the legend
    plt.show()

if ANIMATE:

    score, xpos, ypos = painter_play(best_chromosome, room)
    # Create an animation of the trajectory
    fig, ax = plt.subplots()
    ax.set_xlim(0, room.shape[0]+1)
    ax.set_ylim(0, room.shape[1]+1)
    line, = ax.plot([], [], lw=2)
    start_point, = ax.plot([], [], 'o', color='blue', label='start')
    end_point, = ax.plot([], [], 'o', color='red', label='end')

    def init():
        line.set_data([], [])
        start_point.set_data([], [])
        end_point.set_data([], [])
        return line, start_point, end_point

    def update(frame):
        line.set_data(xpos[:frame], ypos[:frame])
        start_point.set_data(xpos[0], ypos[0])
```

```
        end_point.set_data(xpos[frame-1], ypos[frame-1])
        return line, start_point, end_point

    ani = FuncAnimation(fig, update, frames=len(xpos), init_func=init,
        blit=True, repeat=False, interval=100)
    plt.xlabel('X Position')
    plt.ylabel('Y Position')
    plt.title('Trajectory of Best Chromosome, score = {:.3f}'.format(score))
    plt.legend()
    plt.show()
```

3.2 Code for Task 2

```
import numpy as np
import matplotlib.pyplot as plt
import random

# Define the three points of the triangle
point1 = np.array([0, 0])
point2 = np.array([1, 0])
point3 = np.array([0.5, np.sqrt(3)/2])

# Function to generate a random point inside the triangle
def random_point_in_triangle(p1, p2, p3):
    s, t = sorted([random.random(), random.random()])
    return s * p1 + (t - s) * p2 + (1 - t) * p3

# Initial current position inside the triangle
current_position = random_point_in_triangle(point1, point2, point3)

# Define probabilities for each vertex
p1, p2, p3 = 0.1, 0.1, 0.8

# List to store the points after 100 steps
points = []

# Perform the iteration
for i in range(5000): # Let's do 5000 iterations for a good plot
    # Randomly select one of the three vertices based on the probabilities
    vertex = random.choices([point1, point2, point3], weights=[p1, p2, p3],
        k=1)[0]

    # Move half the distance from the current position to the selected vertex
    current_position = (current_position + vertex) / 2

    # After the first 100 steps, start storing the points
    if i >= 100:
        points.append(current_position)

# Convert the list of points to a numpy array for plotting
```

```
points = np.array(points)

# Plot the points
plt.scatter(points[:, 0], points[:, 1], s=0.1)
plt.show()
```