CEE/MAE M20
Fall 2019 Final Project
Genetic Algorithm

Amy Yu
UID: 905138432
December 14, 2019

1. Introduction

The goal in this problem is to find the best joint angles for a four-link robotic arm so that it will reach the goal point and avoid all obstacles. In order to find a solution, the distance between the end of the robotic arm to the goal has to be minimized. We will be using a genetic algorithm to do this. A genetic algorithm uses the idea the most fit chromosomes are more likely to breed and pass on their genes, so they will survive to the next generation. To do this, we will need to implement selection, crossover, mutation, and elitism.

2. Models and Methods

2.1 Fitness

The `fitness` function is first implemented. This function takes in an array of link angles which is obtained using the given `decodeChromosome.p` file and outputs a normalized fitness score.

This is done as shown in pseudocode below:

```
Input:
theta – array of four link angles
Output:
score – a normalized fitness score
    1: Initialization:
         Save the given values (arm lengths, goal point, obstacle points) in arrays
    2: Find joint locations
    3: Find centers of mass using angles and lengths
    4: Find end point by adding components of each arm
    5: Calculate $f_g$ and $f_0$
    6: return fitness score
```

To find the joint locations, the following equations were used:

$$x_k = x_{k-1} + L_{k-1}\cos(\theta_{k-1})$$
$$y_k = y_{k-1} + L_{k-1}\sin(\theta_{k-1})$$

x_1 and y_1 are set to 0 because the robotic arm begins at the point (0,0), and a `for` loop is used to set the remaining locations from k = 2 to k = 4.

To find the centers of mass, the following equations were used:

$$x_{c,k} = x_k + \frac{L_k}{2}\cos(\theta_k)$$
$$y_{c,k} = y_k + \frac{L_k}{2}\sin(\theta_k)$$

A `for` loop is used from k = 1 to k = 4 to set the center of masses.

The location of the endpoint of the arm is calculated by adding the x-components of each arm to find x_e and adding the y-components of each arm to find y_e. Lastly, f_o is calculated using a `for` loop and Equation (2), and f_g is calculated using Equation (1). These values are plugged into Equation (3) and returned by the function.

2.2 Selection

The `selection` function takes an array of chromosomes, which is an array of uint32, and returns a new array of chromosomes that have been selected using the Binary Tournament Selection policy.

This is done as shown in pseudocode below:

```
Input:
chromosomePopulation – array of uint32
fitness – function handle to evaluate fitness
decode – function handle to decode a chromosome
Output:
chromosomeSubpopulation – a new array of uint32
    1: Initialization:
         Find number of chromosomes in chromosomePopulation, n
         Initialize chromosomeSubpopulation as a uint32 array that has one less element than
         chromosomePopulation
    2: for each chromosome:
    3:          Generate two random integers between 1 and n
    4:          Decode two random chromosomes
    5:          Add the fitter chromosome into chromosomeSubpopulation
    6: return chromosomeSubpopulation
```

The size of chromosomeSubpopulation is n-1, but due to elitism, the fittest chromosome in each generation is guaranteed to survive onto the next generation, so this gets added on and keeps the population at n chromosomes.

2.3 Crossover

The `crossover` function takes two parent chromosomes and returns two daughter chromosomes. There is also a probability that crossover will not occur, and in that case, the daughter chromosomes and copies of the parents.

This is implemented as shown in pseudocode below:

```
Input:
chromosome1 – uint32
chromosome2 – second uint32
pCrossover – probability that crossover will occur
Output:
```

daughter1 – resulting uint32
daughter2 – second resulting uint32

```
1:  **If** crossover does occur
2:        generate random number between 1 and 32, k
3:        Create masks for the first k bits and the last 32-k bits
4:        Decode two random chromosomes
5:        Add the fitter chromosome into chromosomeSubpopulation
6:  **return** chromosomeSubpopulation
```

In order to check that crossover does occur, a random number between 0 and 1 is first generated. If that number is less than the probability of crossover, then it will occur. If the number is greater, the daughters will just be the parent chromosomes.

The `bitshift` function is used to create the first mask, and `bitcmp` is used to create the second mask, which is the opposite of the first mask. The `bitand` function is used to grab the section of the chromosome, and the `bitor` function is used to combine the parts and form the daughters.

The size of chromosomeSubpopulation is n-1, but due to elitism, the fittest chromosome in each generation is guaranteed to survive onto the next generation, so this gets added on and keeps the population at n chromosomes.

2.4 Mutation

The `mutation` function takes a chromosome and flips a random bit.

This is implemented as shown in pseudocode below:

Input:
chromosome – uint32
pMutation – probability that mutation will occur
Output:
mutatedChromosome – resulting uint32

```
1:  **If** mutation does occur
2:        generate random number between 1 and 32, k
3:        create masks for the kth bit
4:        toggle the bit
5: **return** mutatedChromosome
```

The probability of mutation occurring is checked in the same way as for the crossover function. The `bitset` function is used to create the mask, and the `bitxor` function is used to toggle the bit. If mutation does not occur, the original chromosome is returned.

2.5 Genetic Algorithm

The `geneticAlgorithm` function uses the previous functions to find the evolution of the population over a certain number of generations. The function outputs the four joint link angles by decoding the final array of chromosomes. Evolution was stopped when the highest fitness score was greater than 0.99. A condition was also created so that the loop will break if the highest fitness score has not reached 0.99 after 1000 generations. The pseudocode is given in the problem statement.

Random chromosomes are initialized by using the `randi` function to generate numbers between 1 and 4,294,967,295 and converting it to a `uint32`, since 4,294,967,295 is the largest number that can be taken by `uint32`. The `histogram` function is used to create a histogram, and it is updated after each generation. The x and y limits are set to between 0 and 1, and the `pause` function is used to display the generations of the histogram. The max fitness score is also plotted. This is done by creating two arrays: one for the number of iterations of the while loop and one for the highest score in each generation. This is plotted using the `plot` function.

The `videoWriter` function is used to save the histograms for all generations as a video.

2.6 Main Script

The main script is used to run the `geneticAlgorithm` function and plot the configuration of the robot using the final angle values. The population size is set to 1001, the probability of crossover is 0.8, and the probability of mutation is 0.3.

The configuration of the robot is plotted by first finding the joint locations using the following equations, with x_1 and y_1 equal to 0:

$$x_k = x_{k-1} + L_{k-1}\cos(\theta_{k-1})$$
$$y_k = y_{k-1} + L_{k-1}\sin(\theta_{k-1})$$

Then, the obstacles and goal are plotted as points and lines connect the joint locations using the `plot` function.

3. Results

When the main script is run, the following is printed to the screen:

```
0.4558      0.7515      1.0718      1.4414
```

These are the final joint angles in radians of each of the four robotic arms.

Two figures are also generated.

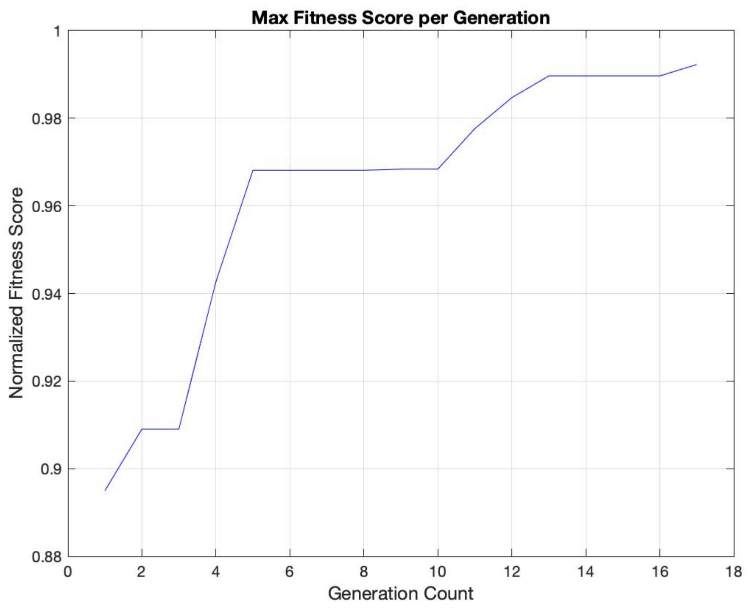

Figure 1: The maximum fitness score in a population over several generations.

Figure 1 shows the maximum normalized fitness score over a certain number of generations, until the score becomes greater than 0.99. We can see that the maximum fitness score only increases or stays the same over generations and never decreases.

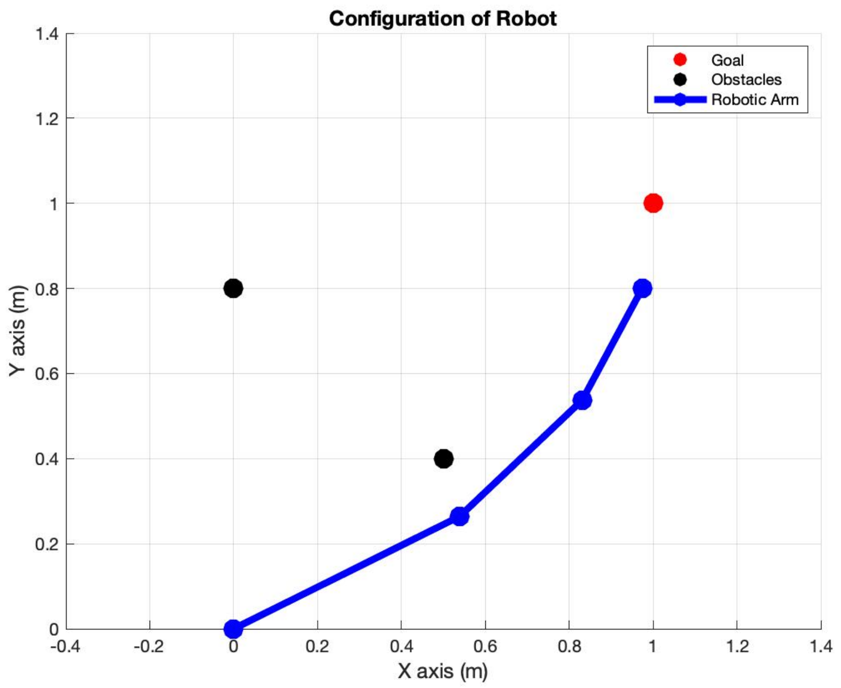

Figure 2: The positions of the robotic arms so that the distance between the end point and the goal is minimized, while avoiding obstacles.

Figure 2 shows the configuration of the robot arms with lengths 0.6, 0.4, 0.3, and 0.4 meters, and joint angles of 0.4558,0.7515, 1.0718, and 1.4414 radians on the x-y plane. The obstacles are placed at (0.0, 0.8) and (0.5, 0.4), and the goal point is placed at (1.0, 1.0). It can be seen that the arms avoid both obstacles and is close to the goal point.

A video of the histograms was also generation and can be seen at the following link: https://youtu.be/X1l6m3XU9wo

The video shows that the portion of the population with higher fitness scores increase over time. This makes sense, as only the chromosomes with higher fitness scores survive onto the next generation.

4. Discussion

Typically, the highest fitness score converges to 0.99 in under 30 generations. In this case, it converged after 17 generations. However, occasionally, it will not converge so a condition was set in `geneticAlgorithm` to stop the loop after 1000 iterations.

When the main script was run again, the maximum fitness score converged after 22 generations and the following angles were printed to the screen:

```
0.4558      0.7515      1.0718      1.4414
```

This is similar to the angles given previously, showing that the algorithm is consistent.