Duncan Di Mauro
UID: 805163177
CEE/MAE M20
December 14, 2019

Final Project

1 Introduction

1.1 The Robotic Arm Problem

In this problem, we are given a 4-link robotic arm with given link-lengths L1, L2, L3, and L4. Additionally, the arm's joints make angles $\theta_1, \theta_2, \theta_3$, and θ_4 with the horizontal, as seen in Figure 1 below. The goal of this problem is to find the optimal joint-angles such that the robotic arm (starting at the origin) reaches the goal point located at a position (x_g, y_g) while also avoiding the 2 obstacles located at positions $(x_{o,1}, y_{o,1})$ and $(x_{o,2}, y_{o,2})$.

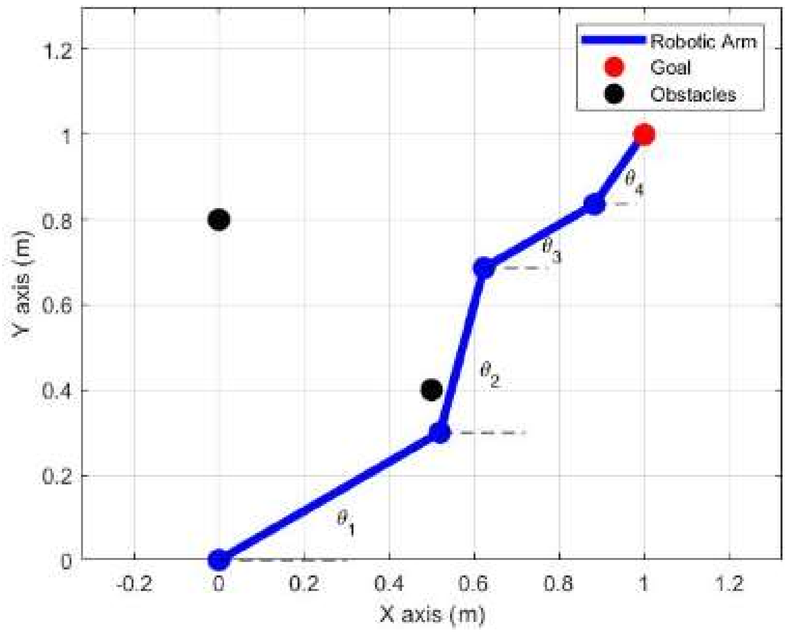

Figure 1: A 4-link robotic arm with a goal point and different obstacles. Note that the joint angles are defined relative to the horizontal. This figure is taken from CEE/MAE M20: Fall 2019 Final Project (ver. 3)[1].

There are two main fitness values that we can use to determine how well our solution (the four joint-angles) solves the problem. The first value is the distance of the arm's endpoint (x_e, y_e) from the goal (x_g, y_g). The smaller the distance, the better our solution. This distance (f_g) can be calculated with the following equation:

$$f_g = \sqrt{(x_g - x_e)^2 + (y_g - y_e)^2} \quad (1)$$

Along with reaching the goal, we also want the robotic arm to avoid touching all obstacles. Therefore, we should attempt to maximize the distance between each obstacle and each link of the robotic arm. We can use the sum of the distances from the *j*-th obstacle at $(x_{o,j}, y_{o,j})$ to the *i*-th link's center of mass at $(x_{c,i}, y_{c,i})$ to represent this value (the greater the number, the farther our arm is from both objects). The equation used to calculate this value (f_o) is shown below:

$$f_o = \sum_{j=1}^{2} \sum_{i=1}^{4} \sqrt{\left(x_{o,j} - x_{c,i}\right)^2 + \left(y_{o,j} - y_{c,i}\right)^2} \tag{2}$$

There are various methods we can use to optimize these fitness-values to find the best solution. One such method is known as the genetic algorithm.

1.2 Genetic Algorithm Outline

For purposes of this problem, a solution (a set of 4 angle-values in radians $\theta_1, \theta_2, \theta_3$, and θ_4) can be represented by a single uint32 number. In this problem, the function `decodeChromosome` will be used to extract the 4 angle values (radians) from a single uint32 number. In the genetic algorithm, these uint32 numbers will be referred to as "chromosomes." The genetic algorithm starts by creating a population of random chromosomes (i.e., a population of random solutions). It then uses a `fitness` function to calculate the fitness (how well a solution solves the given problem) of each chromosome. The genetic algorithm updates the population's chromosome values through a well-defined series of operations, repeating this series until the population includes a chromosome with a sufficient fitness-value. The functions used within this series of operations include, `fitness`, `selection`, `crossover`, and `mutation`. Additionally, for this problem, it is necessary to use the function `decodeChromosome` for reasons previously mentioned. All of these functions and their implementations will be described in the following section.

The pseudocode for the genetic algorithm can be seen below:

```
Inputs:
   fitness: function handle for evaluating the fitness of a solution in the original solution space
   decodeChromosome: function handle for decoding a 32-bit chromosome into a solution in the
                     original solution space (in our case, an array of four angles in radians)
   populationSize: number of chromosomes used in the genetic algorithm
   pCrossover: probability of crossover
   pMutation: probability of mutation
Outputs:
   xOpt: optimal solution as expressed in the original solution space

1. Initialization:
   Randomly initialize populationSize chromosomes into a vector, chromosomePopulation
2. while the fittest chromosome does not converge perform the following series of operations:
3:     Use selection to select populationSize-1 chromosomes from
       chromosomePopulation into a subpopulation chromosomeSubpopulation
4:     Perform crossover on members in chromosomeSubpopulation with probability
       pCrossover
5:     Perform mutation on members in chromosomeSubpopulation with probability
```

```
     pMutation
6:   Use fitness to find the fittest chromosome from chromosomePopulation
7:   Elitism: update chromosomeSubpopulation =
     chromosomePopulation(fittestChromosome) ∪ chromosomeSubpopulation
8:   Update chromosomePopulation = chromosomeSubpopulation
9: return xOpt = decodeChromosome(chromosomePopulation(fittestChromosome))
```

2 Inner Functions – Overview & Methods

2.1 Overview of Each Function

The inner-functions (the functions used to perform specific operations within the genetic algorithm) all serve a different purpose. A brief overview of each function, along with their purposes, can be seen below:

`fitness` – This function takes an array of 4 angle values ($[\theta_1 \;\; \theta_2 \;\; \theta_3 \;\; \theta_4]$) as an input and outputs a normalized fitness-value (`score`) between 0 and 1. This value is determined using the following equation, which includes the previously described goal-fitness and object-fitness values f_g and f_o:

$$\texttt{score} = 1 - f_g - \frac{5.1730 - f_o}{5.1730} \tag{3}$$

`selection` – This function takes the `chromosomePopulation` as an input and returns a `chromosomeSubpopulation`. This subpopulation is created by applying the Binary Tournament Selection policy. This is achieved by creating a loop in which two random chromosomes from the input array are chosen, with the fitter one being appended to the `chromosomeSubpopulation` until it has the required number of elements. These fitness scores are calculated using the `fitness` function. This will allow for the resulting subpopulation to be filled with fitter chromosomes overall. It is also important to note that this subpopulation has one less chromosome than the input population, since elitism is to be implemented later (elitism is explained in a later section of the report).

`crossover` – This function performs crossover on two chromosomes with a probability `pCrossover`. In this function, a crossover point is obtained, located after a random bit in a uint32 value. Crossover involves taking two parent chromosomes as an input and swapping their respective bits located before the crossover point (so if the crossover point is located after the 8th bit, bits 1-8 will be swapped). The resulting chromosomes with swapped bits (daughter chromosomes) are then returned.

`mutation` – This function takes a single chromosome as an input and performs mutation on it with a probability `pMutation`. During mutation, a random bit in a uint32 number is flipped. The function then returns the resulting mutated chromosome.

2.2 Fitness (Method)

The goal in this function is to create a function `score = fitness(theta)` that evaluates the fitness of a chromosome using Eq. (3). However, we first need to use Eqs. (1) and (2) to get the necessary values to use in Eq. (3). First, the function assigns all given values (link lengths and object/goal coordinates) to variables. In this problem, the link-lengths were $[L1, L2, L3, L4] = [0.6, 0.4, 0.3, 0.2]m$, the object coordinates were $(x_{o,1}, y_{o,1}) = (0.0, 0.8)$ and $(x_{o,2}, y_{o,2}) = (0.5, 0.4)$, and the goal was located at $(x_g, y_g) = (1.0, 1.0)$. Next, the code finds the distances each arm-link travels in the x and y directions by using trigonometric functions with the given arm-lengths and their respective angles. Once these distances are obtained, it is trivial to find the end-points of each arm, along with the location of each arm's center of mass. For example, to find the 3rd link's center of mass x-coordinate, add the 1st and 2nd link's distances in the x-direction, then add half of the 3rd link's distance in the x-direction. To find the 3rd link's endpoint x-coordinate, just add the first 3 link's x-distances. The code is then able to plug in the necessary values into Eqs. (1) and (2) which can then be used to obtain a fitness value from Eq. (3).

2.3 Selection (Method)

The function `chromosomeSubpopulation = selection(chromosomePopulation, fitness, decodeChromosome)` first pre-allocates space for the subpopulation, creating an array of uint32 values with `length(chromosomePopulation)-1` entries. Next, a `for` loop is created with `length(chromosomeSubpopulation)` amount of iterations. Two different random numbers are then chosen, each representing an index value (column number) of the input population. The function `decodeChromosome` is then applied to `chromosomePopulation(random indices)` in order to obtain the `theta` values of the two different chromosomes. Next, the `fitness` function is applied to these `theta` values, and a conditional statement is used to append the chromosome with the larger fitness value to the subpopulation (if the fitness values are equal, it doesn't matter which one is appended). For example: `chromosomeSubpopulation(iteration) = chromosomePopulation(index of chromosome with larger fitness).`

2.4 Crossover (Method)

The function `[daughter1, daughter2] = crossover(chromosome1, chromosome2, pCrossover)` first uses a conditional statement in order to ensure that the function is performed with a probability `pCrossover`:

```
if rand() <= pCrossover
perform crossover operations
else
return inputs as outputs
```

Next, a variable `m` is assigned to a random number between 1 and 32, representing a crossover point located directly after the mth bit of a uint32 value. The relevant bits for `chromosome1` and `chromosome2` are extracted via the `bitget` function. Then, the `bitset` function is used to replace the relevant `chromosome1` bits with the extracted `chromosome2` bits and vice versa. Finally, these new chromosomes with swapped bits are returned as `daughter1` and `daughter2`.

2.5 Mutation (Method)

The function `mutatedChromosome = mutation(chromosome, pMutation)` first uses a conditional statement similar to the one used in `crossover` to ensure that the function is performed with a probability pMutation. Next, a random number between 1 and 32 is chosen, representing a single bit. Then, the script flips the value of the input chromosome's relevant bit. If the bit's value is 1 (value obtained via bitget), that bit's value is set to 0 (via bitset) and vice versa. Finally, the function outputs the mutated chromosome.

3 Genetic Algorithm & Main Script - Methods

3.1 Genetic Algorithm

The pseudocode-outline for the genetic algorithm is presented in section 1.2 of the report. The bolded numbers in this section represent the analogous parts of the pseudocode.

The function `xOpt = geneticAlgorithm(fitness, decodeChromosome, populationSize, pCrossover, pMutation)` generates a random chromosome population of size populationSize and performs the genetic algorithm optimizer with a pCrossover probability of performing the crossover function and a pMutation probability of performing the mutation function. It then returns the optimal solution (defined by the fitness function) xOpt in the original solution space, which is achieved by using decodeChromosome on the most optimal chromosome.

1. The function first randomly generates the chromosomePopulation (uint32 values) of size populationSize with the randi function, using the numbers from the uint32 space (0 to 2^32 – 1).

2. Next, a while loop is created with the condition that the fittest chromosome must have a fitness value greater than 0.99. If a chromosome has a fitness value of > 0.99, it is considered to converge, and the while loop ends.

3. In the first part of the loop, the `selection(chromosomePopulation, fitness, decodeChromosome)` function is called to create the chromosomeSubpopulation.

4. Next, a for loop is used to perform crossover on the entire subpopulation. The loop goes from `n = 1:2:length(chromosomeSubpopulation)` in order to perform crossover on the nth and nth + 1 chromosomes of subpopulation. The step size is 2 in order to ensure that crossover is applied on every chromosome only once. Additionally, once the crossover function is performed, the nth and nth + 1 chromosomes of the subpopulation are replaced by the daughter1 and daughter2 values obtained via crossover.

5. Another loop (`for n = 1:length(chromosomeSubpopulation)`) is created that applies mutation to every chromosome of the subpopulation and updates that chromosome's value to reflect the mutated version.

6. Next, we implement elitism by adding the fittest chromosome from the original population to the current subpopulation. An array is created containing all of the fitness values from the original chromosomePopulation. This is simply achieved with a for loop and the use of

```
PopulationFitness(n) = ...
fitness(decodeChromosome(chromosomePopulation(n))).
```

The function then finds the highest fitness value and its relevant index (indexes are the same between chromosomes in chromosomePopulation and their matching fitness values in PopulationFitness).

7. Since we now know the index of the fittest chromosome, we can add it to the subpopulation via

```
chromosomeSubpopulation(populationSize) = ...
chromosomePopulation(index).
```

8. In the last part of the `while` loop, the variable `chromosomePopulation` is set equal to `chromosomeSubpopulation` so that the loop can run again if needed. A check is also performed at the end of the `while` loop that terminates the loop if the greatest fitness value is > 0.99.

9. Finally, the optimal solution `xOpt` is obtained by using `decodeChromosome` on the fittest chromosome of the population.

The `geneticAlgorithm` function should also produce a plot of the maximum fitness score in a population vs. iteration number. This can be achieved by creating two vectors – one that stores the maximum fitness score every iteration and another that stores the iteration number. Since we do not know when the genetic algorithm will end, we allow these vectors to expand on every iteration until reaching their needed size. We can then plot the values in these vectors against one another.

Lastly, the `geneticAlgorithm` function should also create a video showcasing the distribution of chromosomes evolving over time according to the normalized fitness values. This can be done by creating a histogram on every iteration using the `PopulationFitness` vector with `probability` as the specified normalization. The figure can be stored in a vector as an image. After the `while` loop has completed, the `VideoWriter` function can be used to produce a video with each image acting as a frame.

3.2 Main Script

The main script first lists all required given values, which in this problem include `populationSize = 1001`, `pCrossover = 0.8`, and `pMutation = 0.3`. It then calls the geneticAlgorithm function such that `xOpt = geneticAlgorithm(@fitness, @decodeChromosome, populationSize, pCrossover, pMutation)`. The values of the four angles contained within the solution `xOpt` are then printed to the command window. Next, a plot is created showing the robotic arm using the angles obtained from `xOpt`. Additionally, the objects and goal are plotted at their respective locations. Each link of the robotic arm is plotted individually as a line going from its starting location to its endpoint. The starting locations and endpoints of each link are obtained in exactly the same fashion as they were obtained in the previously described `fitness` function.

4 Calculations and Results

During one execution of the main script, the following text was printed to the command window and the following figures were created (note that the histogram shown below is the final histogram produced, and the script produced a video containing the histogram at each generation):

```
Final angle values (radians) for angles 1, 2, 3, 4 are:
0.5051 0.7269 0.9486 1.5646
```

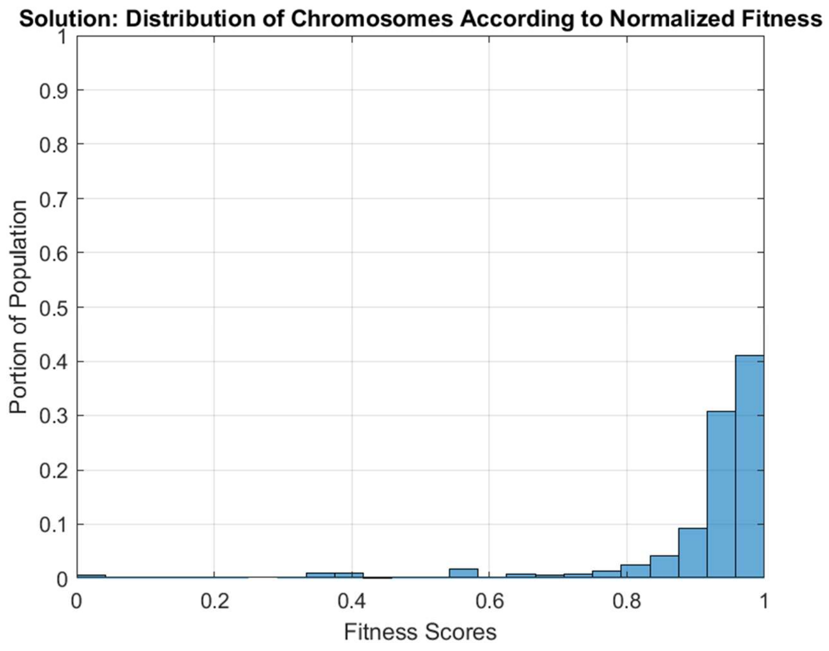

Figure 2: This figure shows the distribution of fitness scores for the final generation of chromosomes within `chromosomePopulation`. The distribution of higher fitness scores increases per generation (shown in video).

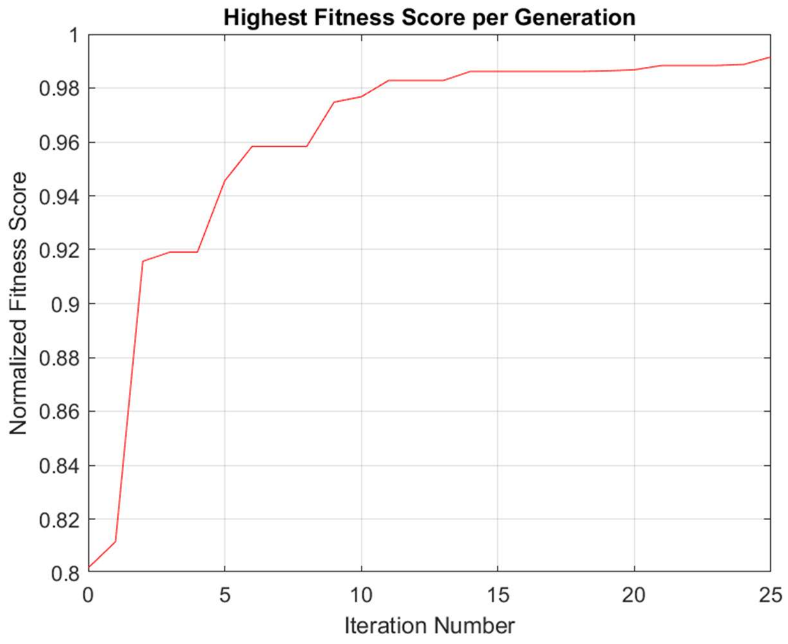

Figure 3: This figure showcases the highest fitness score obtained per generation of `chromosomePopulation`. The highest fitness score either increases or stays the same with each iteration.

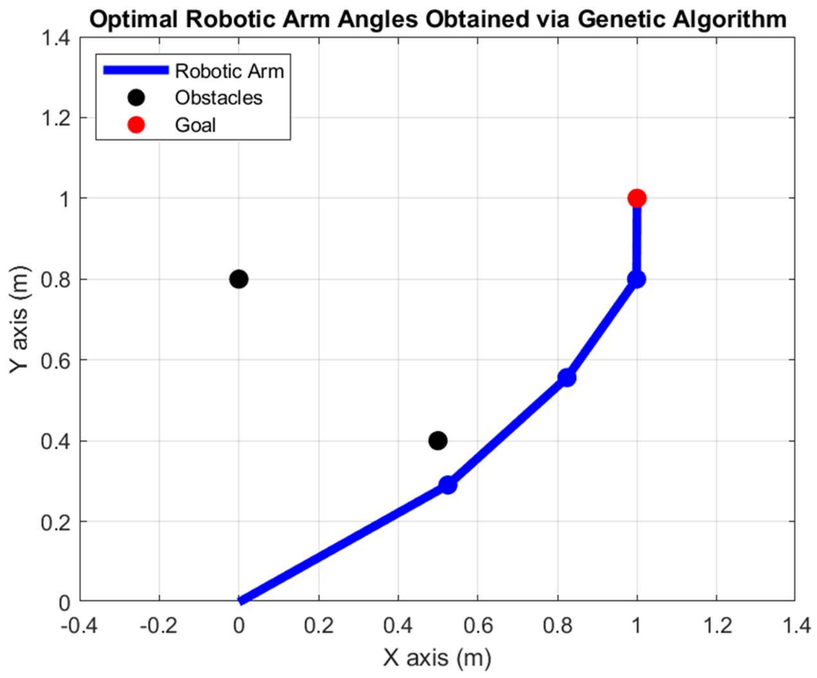

Figure 4: A 4-link robotic arm with a goal point and two obstacles. The joint angles used in this plot were obtained via the genetic algorithm. The arm is shown to reach the goal while avoiding both obstacles.

5 Discussion

If we look at the above plots, we can note several trends of the genetic algorithm and confirm that it is working properly. Looking at Figure 2, we can notice that most of the population has a high fitness score. If we look at the video that our code generated, we can see that the majority of the population had a very low fitness score on the first generation and that the concentrations of fitness scores kept improving over time. Sometimes, there would be a very small increase in lower fitness-scores (most likely due to random mutation), but all of the other fitness scores improved. This means that our genetic algorithm is working properly and can probably be attributed mostly to the selection portion of the algorithm. During selection, the subpopulation is created with only the fittest chromosomes, so it makes sense that after each generation there are more chromosomes with higher fitness values.

Looking at Figure 3, we can immediately notice that the max fitness score gradually got higher and reached a value above 0.99 at the 25th generation. Additionally, we notice that the maximum fitness score never goes down, but sometimes stays the same. This means that the elitism portion of the code is working properly, and that the chromosome with the highest fitness score is always going into the next generation. After many executions of the main script, it is shown that it usually takes between 15 and 30 generations to converge. Additionally, it is interesting to note that usually there is a very large increase in max fitness for the first few (~5) generations, followed by smaller increases in fitness until convergence.

Looking at Figure 4, we can observe that the robotic arm successfully reaches the goal while avoiding all obstacles. This main script has been executed a large amount of times and the arm has been successful every time. This means that our genetic algorithm is working properly and that our criteria for convergence (fitness score > 0.99) produces a successful result. It is interesting to note that the arm is always located on the right side of both obstacles and never in-between. This is likely due to the nature of the `fitness` function. The `fitness` function considers the distance that each arm-link is from each object (f_o). If the arm passes between the two objects, the value of f_o is most likely smaller (worse) than if the arm were to be located on the right side.

Bibliography

1. CEE/MAE M20: Fall 2019 Final Project (ver. 3). (Univ. California Los Angeles, Los Angeles, California).