Cairo University

Faculty of Computer and Artificial Intelligence

Computational Intelligence

Genetic Algorithm

(Simple)

Dr: Ayman Sabry Ghoneim

Prepared by

Omar Yousef Mohammed

ID

20210591

Introduction

The algorithm is implemented in Python programming language with. I used the structured programming technique which the code is composed from a collection of functions.

We go through each function, and I'll try to explain it as much as possible.

Generate Probabilities

- The function takes the population (Solutions) which is a 2D array as a parameter. And compute the fitness of each solution and store it in a list and make another list of probabilities from that fitness list (number of ones). Returns the fitness list and probabilities.

```
def Generate_Probabilities(array):
    shape = array.shape  # The dimensions of the array
    fitness = np.zeros(shape[0], dtype=int)
    for i in range(shape[0]):
        for j in range(shape[1]):
            fitness[i] += array[i][j]

    probabilities = fitness / fitness.sum()
    return fitness , probabilities
```

Selection

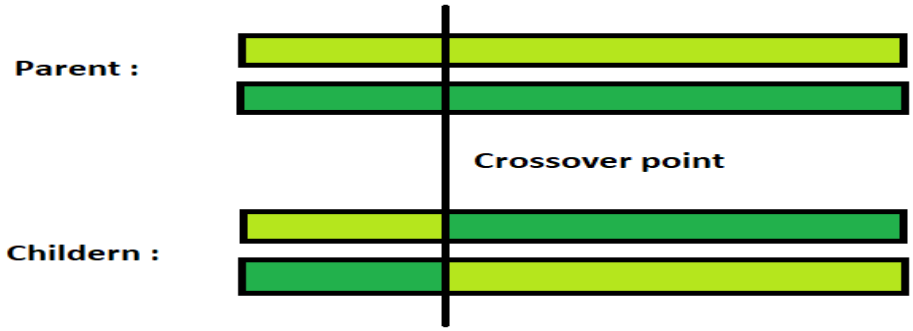
- The mission of this function is to select individuals from the current population to become parents for the next generation.
- The selection is done randomly based on the probability of the solution where the solution with big probability has more probability to be chosen.
- To make this we must make a cumulative probability of solutions.
- Generates two random numbers to select two parents and return their indices.
- This technique called **“Roulette wheel selection”.**

```python
def Selection(array):
    # Generate cumulative probabilities
    cumulative_probabilities = np.cumsum(array)
    num1 = random.random()
    i = 0
    while num1 > cumulative_probabilities[i]:
        i += 1
    j = 0
    while i == j:
        num2 = random.random()
        while num2 > cumulative_probabilities[j]:
            j += 1

    return i, j
```

Crossover Operation

- This function takes two parents and do **“One-Point Crossover”** and returns two children.
- There is a probability of crossover operation, and we generate random number if the number exceeded this probability then the crossover is not done otherwise do crossover.
- Simply the explain of one-point crossover will be in this image:



- The cutting index is selected randomly.

```python
def crossover(parent1, parent2, PCross):

    pCross = random.random()
    if pCross < PCross:
        # Take the index of separation randomly
        split_point = random.randint(0, len(parent1) - 1)
        child1 = parent1[:split_point] + parent2[split_point:]
        child2 = parent2[:split_point] + parent1[split_point:]
        return child1, child2
    else:
        return parent1, parent2

```

Mutation Method

- Mutation method is applied on the whole new generation together. Bit flip mutation is the algorithm that is used in this function.

- Usually, the probability of mutation is very small.

```
1
2  # Mutation Function takes a 2d array that contains the solutions
3  # and perform mutation on them with parameter = 0.01
4  def Mutation(array, PMuation):
5      for i in range(len(array)):
6          if random.random() < PMuation:  # Then there is a mutation
7              flib_index = int(random.uniform(0, 5))
8              if array[i][flib_index] == 1:
9                  array[i][flib_index] = 0
10             else:
11                 array[i][flib_index] = 1
12     # No need for returning it its values will be changed
```

- It iterates on the new generation and at each solution generates random number and if this number is smaller than the probability of mutation then, flip a random bit of the solution otherwise, do nothing.

GA Function

- Now, we go throw the genetic algorithm. I'll try to explain it as much as possible.

```python
def Do_GA(Pop_Size, Num_Generations, Chromosome_Length, Prop_Crossover, Prop_Mutation):
    #Create 2D array with random values zeros or ones with dimensions of 20*5
    Intial_Population = np.random.randint(0, 2, size=(Pop_Size, Chromosome_Length))
    Population = Intial_Population
    #New Generation initializes with zero values
    new_generation = np.zeros((Pop_Size, Chromosome_Length), dtype=int)

    Best_Hist_Fitness = []#Best fitness in each generation

    for i in range(Num_Generations):
        #Generates a probability of each solution based on its fitness
        fitness , probabilities = Generate_Probabilities(Population)

        #Sort the indices of the probability list acsending
        sorted_prob_indices = np.argsort(probabilities)

        #Rearrange the population in a new list by the indices above
        sorted_population = Population[sorted_prob_indices][::-1]

        Best_Hist_Fitness.append(fitness.max())

        # Add best two solutions in the new generation
        new_generation[0] = sorted_population[0]
        new_generation[1] = sorted_population[1]

        # And start adding children from index 2 and jumps step = 2
        j = 2
        while j < Pop_Size - 1:
            # Returns the index of two parents which are married
            parent1_index, parent2_index = Selection(probabilities)

            # Get the two parents from the population by their index
            parent1 = Population[parent1_index, :]
            parent2 = Population[parent2_index, :]

            #Get the new children by crossover function which takes two lists and returns two lists
            child1, child2 = crossover(parent1.tolist(), parent2.tolist(), Prop_Crossover)

            # Add child in this index and one in the next index
            new_generation[j] = child1
            new_generation[j + 1] = child2

            j += 2  # jumps two steps

        Mutation(new_generation, Prob_Muation)
        Population = new_generation

    #The result of the algorithm as a list
    return Intial_Population , Population , Best_Hist_Fitness
```

- First, the parameters that function takes are population size, chromosome length, probability of crossover and of mutation and the number of generations.

- The first step of genetic algorithm is initializing the population randomly as an initial solution. And declare a population array that contains the solutions of each generation and finally declare a new generation 2D array to accept the new children of the current generation.

- The new generation array is initialized by zeros.

```
#Create 2D array with random values zeros or ones with dimensions of 20*5
    Intial_Population = np.random.randint(0, 2, size=(Pop_Size, Chromosome_Length))
    Population = Intial_Population
    #New Generation initializes with zero values
    new_generation = np.zeros((Pop_Size, Chromosome_Length), dtype=int)

    Best_Hist_Fitness = []#Best fitness in each generation
```

- And I make a 1D array storing the best fitness of each generation.
- And then iterate on the number of generations.
- Next step is generating the fitness of each solution.

Elitism

- Elitism is to add best N solutions of the current generation to the new generation without making crossover to preserve the quality of them.
- To make it I sort the population descending based on the fitness and choose the first two solutions which are the best and add them into the next generation at index 0,1 and start adding new children from index 2.

```
#Sort the indices of the probability list acsending
        sorted_prob_indices = np.argsort(probabilities)

        #Rearrange the population in a new list by the indices above
        sorted_population = Population[sorted_prob_indices][::-1]

        Best_Hist_Fitness.append(fitness.max())

        # Add best two solutions in the new generation
        new_generation[0] = sorted_population[0]
        new_generation[1] = sorted_population[1]
```

- Now the step of getting the new generation with the new children came from crossover function.
- To make it we need to iterate on the new generation array until it fills and adds new children.

```
# And start adding children from index 2 and jumps step = 2
        j = 2
        while j < Pop_Size - 1:
            # Returns the index of two parents which are married
            parent1_index, parent2_index = Selection(probabilities)
            # Get the two parents from the population by their index
            parent1 = Population[parent1_index, :]
            parent2 = Population[parent2_index, :]
            #Get the new children by crossover function which takes two lists and returns two lists
            child1, child2 = crossover(parent1.tolist(), parent2.tolist(), Prop_Crossover)
            # Add child in this index and one in the next index
            new_generation[j] = child1
            new_generation[j + 1] = child2

            j += 2  # jumps two steps
```

- And the last step of the algorithm is to apply the mutation on the new generation and swap the values of it with population and do that until the number of generations. And the function returns the history of best fitness and the final generation.

```
1
2       Mutation(new_generation, Prob_Muation)
3       Population = new_generation
4
5   #The result of the algorithm as a list
6   return Intial_Population , Population , Best_Hist_Fitness
```

Finally, Call the function and pass to it the values that it needs to run and print the solution and some another information about the operation.

```
1  Pop_Size = 6
2  Chromosome_Length = 5
3  generations = 10
4  Prob_Crossover = 0.6
5  Prob_Muation = 0.05
6
7  Initial_Solution , Result , Best_Hist_Fitness = Do_GA(Pop_Size , generations , Chromosome_Length , Prob_Crossover , Prob_Muation)
8  AVG_Best_Fitness = sum(Best_Hist_Fitness)/len(Best_Hist_Fitness)
9
10 print(f"Initial Population:\n{Initial_Solution}")
11 print(f"Final Solutions:\n{Result}")
12 print(f"The best solutions over {generations} generations:\n{Best_Hist_Fitness}")
13 print(f"The average of all best solutions over generations: {AVG_Best_Fitness}")
```