JawadAhmed_20P-0165

May 10, 2023

1 Lab 12: Genetic Algorithms

1.0.1 Name: Jawad Ahmed

1.0.2 Roll No: 20P-0165

1.0.3 Section: BCS-6A

In [1]:
```
import numpy as np
```

In [2]:
```
# Initialize a 2D numpy array with 3 rows and 4 columns filled with integer zeros
arr = np.zeros((4, 4), dtype=int)
```

In [3]:
```
arr
```

Out[3]:
```
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```

1.0.4 Step 1: Initial Population

Select any random states as initial population i.e.
No. of population is a random or your choice.
Values in each population represents a random state.

In [4]:
```
def fitness_function(arr):
    return np.count_nonzero(arr == 1)
```

In [5]:
```
def generate_initial_chromosomes(n):
    arr = np.zeros((4,4), dtype=int)
    arr = arr.reshape(16)
    chromosomes = []
    for i in range(n):
        random_numbers = np.random.randint(0, 16, size=10)
        for random_number in random_numbers:
            arr[random_number] = 1
        chromosomes.append(arr)
        arr = np.zeros((4, 4), dtype=int)
        arr = arr.reshape(16)
    return chromosomes
```

```
In [6]: chromosomes = generate_initial_chromosomes(30)
```

```
In [7]: chromosomes
```

```
Out[7]: [array([1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1]),
         array([1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1]),
         array([1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1]),
         array([0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0]),
         array([0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1]),
         array([1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0]),
         array([0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0]),
         array([0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0]),
         array([0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0]),
         array([1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0]),
         array([1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0]),
         array([1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0]),
         array([0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1]),
         array([0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0]),
         array([0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0]),
         array([0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1]),
         array([0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0]),
         array([1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1]),
         array([0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0]),
         array([0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0]),
         array([1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0]),
         array([1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0]),
         array([1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0]),
         array([0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1]),
         array([1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1]),
         array([1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0]),
         array([0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1]),
         array([0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1]),
         array([1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0]),
         array([0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1])]
```

1.0.5 Step 2: Selection of Parents

1. Parents will be selected random from the population in the first step.
2. High fitness value population will have high chances of selection as a parent i.e. P4 will have high weightage to be selected as randomly Selection_Probability P(i) = Fitness of P(i)/Total Fitness of all Populations

```
In [8]: def calculate_probabilities(chromosomes):
        #     Calculate Total Fitness of all the population
            total_fitness_all_population = 0
            for chromosome in chromosomes:
                total_fitness_all_population += fitness_function(chromosome)
        #     Calculate selection probabitlies
            selection_probabilities = []
```

```
        for chromosome in chromosomes:
            selection_probabilities.append((fitness_function(chromosome)) / total_fitness_a
        return selection_probabilities
```

```
In [9]: def convert_chromosomes_to_indices(chromosomes):
            indices_chromosomes = []
            for i in range(len(chromosomes)):
                indices_chromosomes.append(i)
            return indices_chromosomes
```

```
In [10]: def roulette_wheel_selection(chromosomes):
             selection_probabilities = calculate_probabilities(chromosomes)
             indices_chromosomes = convert_chromosomes_to_indices(chromosomes)
             selected_parents_indices = np.random.choice(indices_chromosomes, 2, p=selection_p
             selected_parents = []
             selected_parents.append(chromosomes[selected_parents_indices[0]])
             selected_parents.append(chromosomes[selected_parents_indices[1]])
             return selected_parents
```

```
In [11]: selected_parents = roulette_wheel_selection(chromosomes)
```

```
In [12]: selected_parents
```

```
Out[12]: [array([0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0]),
          array([0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1])]
```

1.0.6 Step 3: Modification

Crossover: 1. This is called convergence step because it will generate children (new states) 2. Create children from meeting of the parents 3. Generate two children/successor from two parents (new version of genetic algorithm produce one child instead)

```
In [13]: def generate_childs_using_crossover(selected_parents):
             crossover_point = np.random.randint(0, len(selected_parents[0]))

         #     Create child
             child_one = np.concatenate((selected_parents[0][:crossover_point], selected_paren
             child_two = np.concatenate((selected_parents[1][:crossover_point], selected_paren

             childs = []
             childs.append(child_one)
             childs.append(child_two)

             return childs
```

```
In [14]: childs = generate_childs_using_crossover(selected_parents)
```

```
In [15]: childs
```

```
Out[15]: [array([0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1]),
          array([0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0])]
```

1.0.7 Mutation:

1. Apply mutation on children in order to get the new/updated state (assuming it will quickly lead us to goal)
2. Mutation probability or mutation rate is fixed and chosen a very small value i.e. 0.01 means generate a random no. between (1-100), mutate the child if random no. is 1 else skip. i.e. 0.2 means generate a random no. between (1-10), mutate the child if random no. is 1 or 2. Skip otherwise
3. Mutation rate will be applied and checked for each digit/char in a population i.e. You will keep repeating this process for each array value in a single population means 16 times random no. will be generated and checked respectively.

```
In [16]: def mutation(childs, mutation_rate = 0.2):
             random_number = 0
             if (mutation_rate == 0.01):
                 random_number = np.random.randint(1, 100)
                 for child in childs:
                     for i in range(len(child)):
         #               If random number is 1 mutate the child
                         if random_number is 1:
                             if (childs[i] is 1):
                                 childs[i] = 0
                             else:
                                 childs[i] = 1

             else:
                 random_number = np.random.randint(1, 10)
                 for child in childs:
                     for i in range(len(child)):
         #               If random number is 1 or 2 mutate it
                         if random_number is 1 or random_number is 2:
                             if childs[i] is 1:
                                 childs[i] = 0
                             else:
                                 childs[i] = 1
```

```
In [17]: childs
```

```
Out[17]: [array([0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1]),
          array([0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0])]
```

```
In [18]: mutation(childs)
```

```
In [19]: childs
```

```
Out[19]: [array([0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1]),
          array([0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0])]
```

1.0.8 Step 4: Evaluation

1. Compute the fitness values of newly generated children
2. Apply the goal test
3. Replace the old population with newly created population having new children
4. Repeat the steps 1-4 if goal is not found.

```
In [20]: def evaluation(goal_fitness_value = 16, no_of_iterations = 10000):
             a = 0
             new_fitness_value = 0
             goal_state = np.array([])
             while(new_fitness_value <= goal_fitness_value and a < no_of_iterations):
                 chromosomes = generate_initial_chromosomes(30)
                 selected_parents = roulette_wheel_selection(chromosomes)
                 childs = generate_childs_using_crossover(selected_parents)
                 if (fitness_function(childs[0]) > fitness_function(childs[1])) and (fitness_fu
                     new_fitness_value = fitness_function(childs[0])
                     goal_state = childs[0]
                 elif (fitness_function(childs[1]) > new_fitness_value):
                     new_fitness_value = fitness_function(childs[1])
                     goal_state = childs[1]
                 a += 1
             return goal_state, new_fitness_value
```

```
In [21]: evaluation()
```

```
Out[21]: (array([0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1]), 13)
```

```
In [ ]:
```

```
In [ ]:
```