Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Assignment

Course No: CSE 4203

Course Title: Neural Networks and Fuzzy Systems

Submitted By

Saifur Rahman

Roll: 1703018

Sec: A

Dept. of CSE, RUET

Submitted To

Dr. Md. Rabiul Islam

Professor

Dept. of CSE, RUET

Date of Submission: 10th June, 2023

Contents

Assignment Name

Implementation of Genetic Algorithm to find out the bit sets that can optimize the search of minimum number.

Objectives

- To find the combination of bits in a chromosome that results in the lowest possible decimal value.
- To iteratively evolve the population through selection, crossover, and mutation operations to improve the fitness.
- To identify the minimum number and the corresponding best chromosome after a certain number of generations.

Problem Definition

This problem involves a genetic algorithm to find out the bit sets that can optimize the search of minimum number. The fitness function or objective function must be defined to find out the minimum number. The basic genetic operators such as reproduction or selection, crossover and mutation should be used to optimize the searching criteria. The chromosome (string) with minimum 20 genes (bits) and population 30. The number of generations may be used for the breaking condition. Followings might be reported:

- Initial random strings and fitness function
- Output of reproduction, crossover and mutation for each generation
- Calculate the efficiency of each generation

Methodology

1. Initialize the genetic algorithm parameters: chromosome length, population size, and number of generations.
2. Generate an initial random population consisting of binary strings.
3. Print the initial population along with their fitness values.
4. Set the minimum number as infinity and the best chromosome as None.
5. Repeat the following steps for the specified number of generations:
    a. Create a new population by selecting parent chromosomes, performing crossover, and introducing mutations.
    b. Update the population with the new population by including children.
    c. Display the current generation and the fitness values of the chromosomes.
    d. Check if any chromosome has a lower fitness value (minimum number) than the current minimum number. If yes, update the minimum number and store the corresponding best chromosome.
    e. Calculate and display the efficiency of the current population.
6. Print the minimum number and the best chromosome found by the genetic algorithm.
7. End the algorithm.

Implementation

```python
# -*- coding: utf-8 -*-
"""Genetic Algorithm CT #4 Assignment

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1h_eF-h3rhCTOwweVum9JkV6lltf7_jVz

"""

import random

# Genetic Algorithm parameters
chromosome_length = 20
population_size = 30
num_generations = 10

"""Fitness Function"""

def fitness_function(chromosome):
    decimal_value = int(chromosome, 2)  # Convert binary string to decimal
    return decimal_value

"""Generate Initial Population"""

def generate_population():
    population = []
    for _ in range(population_size):
        chromosome = ''.join(random.choice(['0', '1']) for _ in range(chromosome_length)) # generate random chromosome
        population.append(chromosome)
    return population

"""Roulette Wheel Selection"""

def selection(population):
    total_fitness = sum(fitness_function(chromosome) for chromosome in population)
    probabilities = [fitness_function(chromosome) / total_fitness for chromosome in population]
    selected = random.choices(population, probabilities, k=2) # select 2 chromosome randomly by using probabilities
    return selected[0], selected[1]

"""Single-Point Crossover"""

def crossover(parent1, parent2):
    crossover_point = random.randint(1, chromosome_length - 1)
    child1 = parent1[:crossover_point] + parent2[crossover_point:]
    child2 = parent2[:crossover_point] + parent1[crossover_point:]
    return child1, child2

"""Bit Flip Mutation"""

def mutation(chromosome, mutation_rate):
    mutated_chromosome = ''
    for bit in chromosome:
        if random.random() < mutation_rate:
            mutated_chromosome += '0' if bit == '1' else '1' # inverted
        else:
```

```
            mutated_chromosome += bit # unchanged
    return mutated_chromosome

"""Each Generation Efficiency Calculation"""

def calculate_efficiency(population):
    return min(fitness_function(chromosome) for chromosome in population)

"""Genetic Algorithm"""

def genetic_algorithm():

    # generate random population
    population = generate_population()
    print("Initial Population:")

    # print population with fitness values
    for chromosome in population:
        print(chromosome, fitness_function(chromosome))
    print()

    minimum_number = float('inf')
    best_chromosome = None

    for generation in range(num_generations):
        new_population = []

        for _ in range(population_size // 2): # no of parent pair
            parent1, parent2 = selection(population) # selection
            child1, child2 = crossover(parent1, parent2) # crossover
            child1 = mutation(child1, mutation_rate=0.01) # mutation
            child2 = mutation(child2, mutation_rate=0.01) # mutation
            new_population.extend([child1, child2])

        population = new_population

        print("Generation", generation + 1)
        for chromosome in population:
            fitness = fitness_function(chromosome)
            print(chromosome, fitness) # fitness value
            if fitness < minimum_number:
                minimum_number = fitness
                best_chromosome = chromosome

        print("Efficiency:", calculate_efficiency(population)) # efficiency of current generation
        print()

    print("Minimum Number:", minimum_number)
    print("Best Chromosome:", best_chromosome)

genetic_algorithm()
```

Results & Performance Analysis

Initial Population:
01001011110010101101
310445
10101011000100010010
700690
11111100001110110010
1033138
01101101001110110100
447412
11000111111111001000
819144
11110111000110010101
1012117
11000000010011110000
787696
11111110101111110110
1043446
11100111110110111010
949690
00010011001001101110
78446
10000011001111000001
537537
01001000100110100100
297380
00000001100001111100
6268
11111110100011111000
1042680
11001111000111110111
848375
11011001011110111110
890814
10010101110000111111
613439
10011110011010110110
648886
00010100001011100011
82659
11110010011010111101
992957
11100111011101100101
948069
00010011110110111101
81341
10101000000001001010
688202
10100111101010011010
686746
11011111010010111001
914617
10100110110010000101
683141
10001011111110101100
573356
00110011000111011110
209374
01010110011111101001
354281
01110100111010110100
478900

Generation 1
11100111110010101101
949421
01001010011101100101
304997
10000011111111101001
540649
01010110011110101100
354220
01111110101101110101
519029
11110111000110010110
1012118
11001111100011111000
850168
11111110000111110111
1040887
11000000010011110000
787696
10101000000001001010
688202
10001110101111110110
584694
11111011111110101100
1032108
10010101110000111010
613434
10100111101010011111
686751
11011001000111110111
889335
11001111011110111110
849854
11111111000010111001
1044665
11011110001111110110
910326
11011111010010011010
914586
10100111101010111001
686777
11111101001110110100
1037236
01100111000110010101
422293
01010110011110101100
354220
10001011111111101001
573417
11100111010111110111
947703
11001111001101100101
848741
11000000011111101001
788457
01010110000011110000
352496
11100111000111110111
946679
11001111110110111010
851386
Efficiency: 304997

Generation 2
11111101001110101100
1037228
01010110011110110100
354228
11000000010011110000
787696
11000000010011110000
787696
11000111000010010110
815254

1111111011110111110
1046462
0111101111110101100
507820
1111110101101110101
1043317
1100000011111110000
788464
1100000010011111001
787705
1100111001101100101
848741
0100101001100100101
304741
1111011100011001 0110
1012118
1000111010111110110
584694
1111011110110111010
1015226
1100111100011001 0110
848278
1100000000011111000
786680
1100111111111101001
851945
1110011101011110111
947703
1110011100011110111
946679
0110011101110110 0001
423777
0100101001011001 0101
304533
1000111010111110111
584695
1111111000111110110
1040886
1010111000011111 0111
713207
1111011110101001 1111
1014431
1100000001001010 1101
787629
1110011111111101001
950249

1111111100001011 1001
1044665
1000101111111101001
573417
Efficiency: 304533

Generation 3

1111011110111010 1100
1014700
1111110100101001 1111
1036959
0111101111110100001
507809
0110011101010110 1100
423276
1111111101101001 1111
1046175
1111011110111011 1110
1014718
1111011100011001 0110
1012118
0000000000001101 1000
216
1100111100011001 0111
848279
1100000001001111 1000
787704
1111111010011001 0110
1042838
1100111100110111 0101
848757
0111101111110101001
507817
1000101111111101100
573420
1010111000011111 0110
713206
1100011100001001 0111
815255
1100111100110101 0110
848726
1100111100011010 0101
848293
1100000111011011 1010
794042
1111011001001111 1001
1008889

1110001010110111 0101
928629
1111100001001111 0000
1017072
1100100101001111 1001
824569
1100011111111101001
819177
1100111001001111 0000
845040
0100000111111101001
270313
1100000000011111 0111
786935
1110011101001111 1000
947448
1110001100011111 1001
930297
1000101110111110 0111
572391
Efficiency: 216

Generation 4

0100000111111101111
270319
1111110100101001 1001
1036953
1111011110111011 0111
1014711
1100011100001001 1110
815262
1111111101101011 1111
1046207
1111011110111010 1100
1014700
1111011001001111 1001
1008889
1111110100101001 1111
1036959
0111011110111010 1100
490412
1111101111111010 1001
1032105
1110111010011001 0110
977302
1111111100011001 0111
1044887

11000000010010010110
787606
11110111000111111000
1012216
11110000000111100111
983527
11000111101110111110
818110
11111100100110010110
1034646
11111111001010011111
1045151
11110110010011110101
1008885
11100010101101111001
928633
11100010101101110101
928629
11100010101101110101
928629
01000011111111101100
278508
10001001111111101001
565225
11000000000111110110
786934
11000001110110111011
794043
11111101000110100101
1036709
11001111101011011111
850655
11000111000011110111
815351
11000000000110010111
786839
Efficiency: 270319

Generation 5
11100010101101110101
928629
11111101000110100101
1036709
11110110010011110111
1008887
11110111101110111001
1014713
11000001010110111011
791995
11000001110110011011
794011
11100010101101111001
928633
11000000010010010110
787606
11111101101110101001
1039273
11110111110110010110
1031574
11000111001101111001
815993
11100010100010011110
927902
11110000000111101110
983534
11101110100110010111
977303
11100010101101110001
928625
11110110010011111101
1008893
11001111101011011111
850655
11000011110110111011
802235
11111111101110101100
1047468
11110100100110010110
1001878
11111110100110010111
1042839
11111111000110010110
1044886
11100000000110010111
917911
11011011111110101001
901033
11111101000110100100
1036708
11101110100110010111
977303
11111101000110010110
1036694
11101110100110100101
977317
11111111000111111000
1044984
11110111000111111000
1012216
Efficiency: 787606

Minimum Number: 216
Best Chromosome: 00000000000011011000

Conclusion & Observation

- The selection process uses the Roulette Wheel Selection method.
- Single-Point Crossover technique is used.
- Bit Flip Mutation is applied.
- Efficiency of each generation is calculated based on the minimum fitness value in the population.
- The algorithm aims to optimize the search for the minimum number by evolving the population over generations.
- The results obtained from the algorithm provide valuable insights into the efficiency and effectiveness of the genetic algorithm.