VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

MACHINE LEARNING (CO3117)

Homework 5:

Genetic Algorithms

Team LHPD2

Semester 2, Academic Year 2024 - 2025

Teacher: Nguyen An Khuong
Students: Nguyen Quang Phu - 2252621 (***Leader***)
Nguyen Thanh Dat - 2252145 (Member)
Pham Huynh Bao Dai - 2252139 (Member)
Nguyen Tien Hung - 2252280 (Member)
Nguyen Thien Loc - 2252460 (Member)

HO CHI MINH CITY, FEBRUARY 2025

Contents

Chapter 1

Introduction

1.1 Purpose of This Document

This chapter serves as an introduction to our team, LHPD2, for the Machine Learning (CO3117) course in Semester 2, Academic Year 2024 - 2025. The purpose of this writing is to formally present our team members and confirm our participation in solving the given homework exercise, "Homework 5: Genetic Algorithms".

1.2 Workload

Specifically, our contributions included:

- **Understanding the Problem:** Each member participated in analyzing and discussing the given exercise to ensure a shared understanding of the requirements.
- **Exploring Concepts:** We collectively researched and reviewed relevant concepts, theories, and methods necessary for solving the problems.
- **Reasoning for the Solution:** The team worked together to identify the rationale behind each approach, ensuring that all steps were logical and well-justified.
- **Implementing the Solution:** The solutions were implemented with equal collaboration, where every member contributed to coding, calculations, and documentation.
- **Final Review:** The team jointly reviewed the solutions to verify their correctness, clarity, and coherence before submission.

We believe that this exercise has enhanced our understanding of the concepts involved and strengthened our ability to work collaboratively as a team.

Chapter 2

Problem Description - Our Solution

2.1 Question 9.1

2.1.1 Description

Design a genetic algorithm to learn conjunctive classification rules for the PlayTennis problem described in Chapter 3. Describe precisely the bit-string encoding of hypotheses and a set of crossover operators.

2.1.2 Solution

To design a genetic algorithm for learning conjunctive classification rules for the PlayTennis problem, we can use the following approach:

2.1.2.1 Bit-string encoding

Each hypothesis can be represented as a fixed-length bit string with 9 bits, corresponding to the possible attribute values:

[Outlook_S | Outlook_O | Outlook_R | Temp_H | Temp_M |

Temp_C | Humidity_H | Humidity_N | Windy_T | Windy_F]

Where:

- Outlook: Sunny (S), Overcast (O), Rainy (R)
- Temperature: Hot (H), Mild (M), Cool (C)

- Humidity: High (H), Normal (N)
- Windy: True (T), False (F)

A '1' in a position indicates the presence of that attribute value in the rule, while a '0' indicates its absence.

2.1.2.2 Crossover operators

a) One-point crossover: Select a random crossover point and exchange the bits after that point between two parent chromosomes.

Parent 1: [101|011010]
Parent 2: [010|101101]
Offspring 1: [101|101101]
Offspring 2: [010|011010]

b) Two-point crossover: Select two random crossover points and exchange the bits between these points.

Parent 1: [10|101|1010]
Parent 2: [01|010|1101]
Offspring 1: [10|010|1010]
Offspring 2: [01|101|1101]

c) Uniform crossover: For each bit position, randomly select which parent will contribute its bit to the offspring.

Parent 1: [101011010]
Parent 2: [010101101]
Offspring: [111101100]

These crossover operators will help maintain diversity in the population while allowing the algorithm to explore the space of possible conjunctive classification rules for the PlayTennis problem.

2.2 Question 9.2

2.2.1 Description

Implement a simple GA for Exercise 9.1. Experiment with varying population size p, the fraction r of the population replaced at each generation, and the mutation rate m.

2.2.2 Solution

To implement a simple genetic algorithm for the PlayTennis problem, we can use the following Python code:

```
def initialize_population(pop_size, rule_length):
    return [random.choices([0, 1], k=rule_length) for _ in range(pop_size)
        ]

def fitness(individual, examples):
    correct = 0
    for example in examples:
        if evaluate_rule(individual, example) == (example['PlayTennis'] ==
            'Yes'):
            correct += 1
    return correct / len(examples)

def evaluate_rule(rule, example):
    attributes = ['Outlook', 'Temperature', 'Humidity', 'Wind']
    value_map = {
        'Outlook': ['Sunny', 'Overcast', 'Rainy'],
        'Temperature': ['Hot', 'Mild', 'Cool'],
        'Humidity': ['High', 'Normal'],
        'Wind': ['Strong', 'Weak']
    }

    index = 0
    for attr in attributes:
        num_values = len(value_map[attr])

        if any(rule[index:index + num_values]):
            attr_value = example[attr]
            valid_values = [value_map[attr][j] for j in range(num_values)
                if rule[index + j] == 1]
            if attr_value not in valid_values:
                return False
        index += num_values
    return True

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

```
    return child1, child2

def mutate(individual, mutation_rate):
    return [1 - bit if random.random() < mutation_rate else bit for bit in
        individual]

def genetic_algorithm(examples, pop_size, generations, replacement_rate,
   mutation_rate):
    rule_length = 10
    population = initialize_population(pop_size, rule_length)

    for _ in range(generations):
        fitnesses = [fitness(individual, examples) for individual in
           population]
        sorted_population = [x for _, x in sorted(zip(fitnesses,
           population), reverse=True)]

        new_population = sorted_population[:int((1 - replacement_rate) *
           pop_size)]

        while len(new_population) < pop_size:
            parent1, parent2 = random.choices(sorted_population[:int(
               pop_size / 2)], k=2)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1, mutation_rate)
            child2 = mutate(child2, mutation_rate)
            new_population.extend([child1, child2])

        population = new_population[:pop_size]

    best_individual = max(population, key=lambda x: fitness(x, examples))
    return best_individual, fitness(best_individual, examples)

def interpret_rule(rule):
    attributes = ['Outlook', 'Temperature', 'Humidity', 'Wind']
    value_map = {
        'Outlook': ['Sunny', 'Overcast', 'Rainy'],
        'Temperature': ['Hot', 'Mild', 'Cool'],
        'Humidity': ['High', 'Normal'],
        'Wind': ['Strong', 'Weak']
    }

    interpretation = []
    index = 0

    for attr in attributes:
        num_values = len(value_map[attr])
        if any(rule[index:index + num_values]):
            selected_values = [value_map[attr][j] for j in range(
               num_values) if rule[index + j] == 1]
            interpretation.append(f"{attr} is {' or '.join(selected_values
               )}")
        index += num_values
    return " AND ".join(interpretation)
```

Example usage

```
examples = [
    {'Outlook': 'Sunny', 'Temperature': 'Hot', 'Humidity': 'High', 'Wind':
        'Weak', 'PlayTennis': 'No'},
    {'Outlook': 'Sunny', 'Temperature': 'Hot', 'Humidity': 'High', 'Wind':
        'Strong', 'PlayTennis': 'No'},
    {'Outlook': 'Overcast', 'Temperature': 'Hot', 'Humidity': 'High', '
       Wind': 'Weak', 'PlayTennis': 'Yes'},
    {'Outlook': 'Rain', 'Temperature': 'Mild', 'Humidity': 'High', 'Wind':
        'Weak', 'PlayTennis': 'Yes'},
    {'Outlook': 'Rain', 'Temperature': 'Cool', 'Humidity': 'Normal', 'Wind
       ': 'Weak', 'PlayTennis': 'Yes'},
    {'Outlook': 'Rain', 'Temperature': 'Cool', 'Humidity': 'Normal', 'Wind
       ': 'Strong', 'PlayTennis': 'No'},
    {'Outlook': 'Overcast', 'Temperature': 'Cool', 'Humidity': 'Normal', '
       Wind': 'Strong', 'PlayTennis': 'Yes'},
    {'Outlook': 'Sunny', 'Temperature': 'Mild', 'Humidity': 'High', 'Wind'
       : 'Weak', 'PlayTennis': 'No'},
    {'Outlook': 'Sunny', 'Temperature': 'Cool', 'Humidity': 'Normal', '
       Wind': 'Weak', 'PlayTennis': 'Yes'},
    {'Outlook': 'Rain', 'Temperature': 'Mild', 'Humidity': 'Normal', 'Wind
       ': 'Weak', 'PlayTennis': 'Yes'},
    {'Outlook': 'Sunny', 'Temperature': 'Mild', 'Humidity': 'Normal', '
       Wind': 'Strong', 'PlayTennis': 'Yes'},
    {'Outlook': 'Overcast', 'Temperature': 'Mild', 'Humidity': 'High', '
       Wind': 'Strong', 'PlayTennis': 'Yes'},
    {'Outlook': 'Overcast', 'Temperature': 'Hot', 'Humidity': 'Normal', '
       Wind': 'Weak', 'PlayTennis': 'Yes'},
    {'Outlook': 'Rain', 'Temperature': 'Mild', 'Humidity': 'High', 'Wind':
        'Strong', 'PlayTennis': 'No'},
]

pop_size = 50
generations = 100
replacement_rate = 0.5
mutation_rate = 0.01

best_rule, best_fitness = genetic_algorithm(examples, pop_size,
    generations, replacement_rate, mutation_rate)
print(f"Best rule: {best_rule}")
print(f"Fitness: {best_fitness}")
print(f"Interpretation: IF {interpret_rule(best_rule)} THEN PlayTennis =
    Yes")
```

After run this, we get the result:

```
Best rule: [0, 0, 0, 0, 0, 0, 0, 1, 1, 1]
Fitness: 0.7142857142857143
Interpretation: IF Humidity is Normal AND Wind is Strong or Weak THEN
    PlayTennis = Yes
```

To experiment with varying population size, replacement rate, and mutation rate, you can run the algorithm multiple times with different parameters:

```
pop_sizes = [20, 50, 100]
replacement_rates = [0.3, 0.5, 0.7]
mutation_rates = [0.01, 0.05, 0.1]
generations = 100

results = []
for p, r, m in itertools.product(pop_sizes, replacement_rates,
    mutation_rates):
     best_rule, best_fitness = genetic_algorithm(examples, p, generations,
         r, m)
     results.append({'Population Size': p, 'Replacement Rate': r, 'Mutation
         Rate': m, 'Best Fitness': best_fitness})

df_results = pd.DataFrame(results)
```

This implementation allows you to experiment with different population sizes, replacement rates, and mutation rates to observe their effects on the algorithm's performance in learning conjunctive classification rules for the PlayTennis problem.

When run the last block of code, we get:

# Population Size	# Replacement Rate	# Mutation Rate	# Best Fitness
20	0.3	0.01	0.6428571428571429
20	0.3	0.01	0.6428571428571429
100	0.7	0.01	0.6428571428571429
20	0.5	0.01	0.6428571428571429
20	0.5	0.05	0.6428571428571429
20	0.5	0.1	0.6428571428571429
20	0.7	0.01	0.6428571428571429
50	0.7	0.01	0.6428571428571429
100	0.5	0.01	0.7142857142857143
100	0.5	0.05	0.7142857142857143

Table 2.1: Genetic Algorithm Experiment Results

2.3 Question 9.3

2.3.1 Description

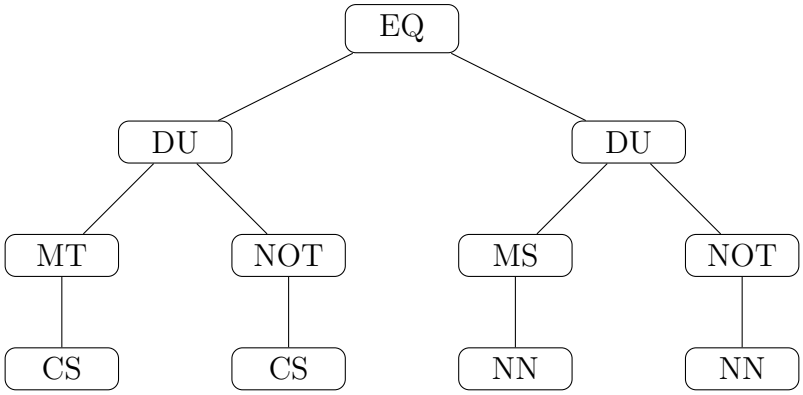
Represent the program discovered by the GP (described in Section 9.5.2) as a tree. Illustrate the operation of the GP crossover operator by applying it using two copies of your tree as the two parents.

2.3.2 Solution

The Genetic Programming (GP) discovered the following program:

$$(EQ(DU(MT\ CS)(NOT\ CS))(DU(MS\ NN)(NOT\ NN)))$$

This program can be represented as a tree:

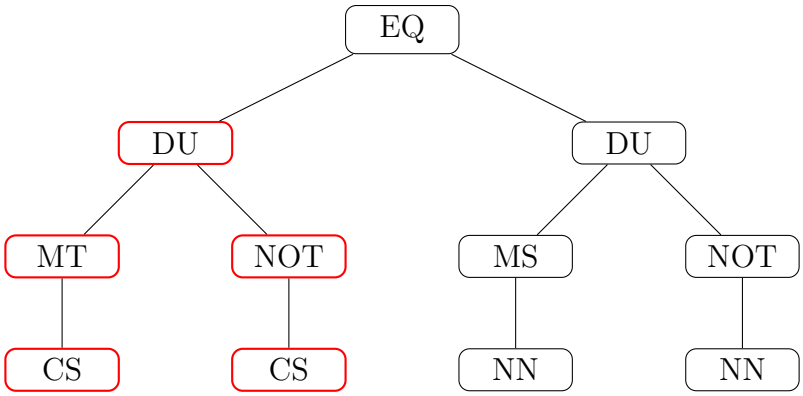


The tree structure shows the hierarchical composition of functions in the program.

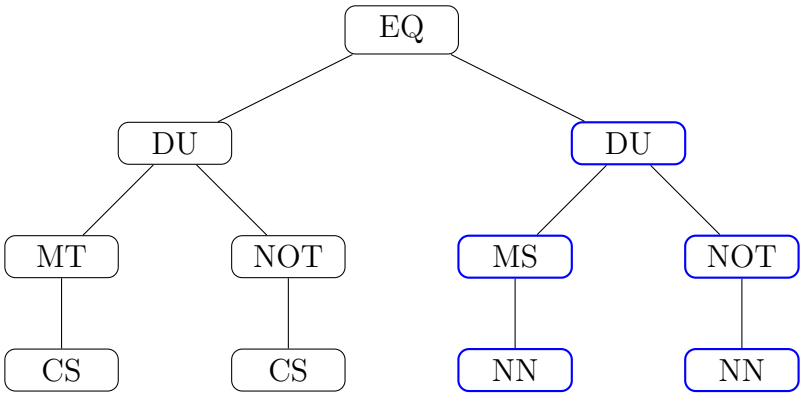
Illustration of the GP Crossover Operation

The crossover operation in Genetic Programming involves swapping subtrees between two parent trees to create offspring.

Parent 1:

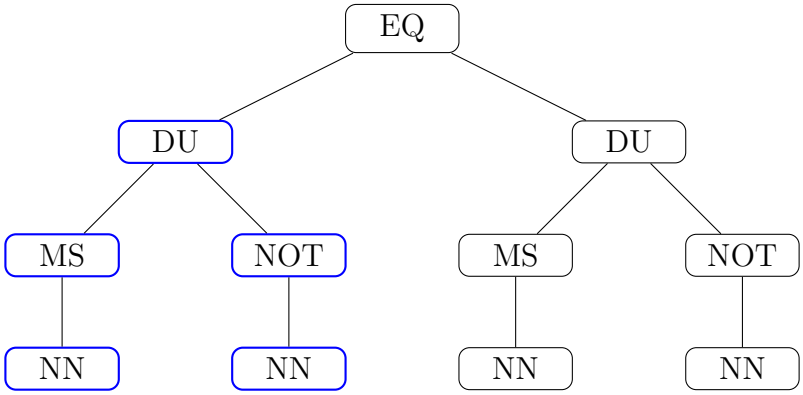


Parent 2:

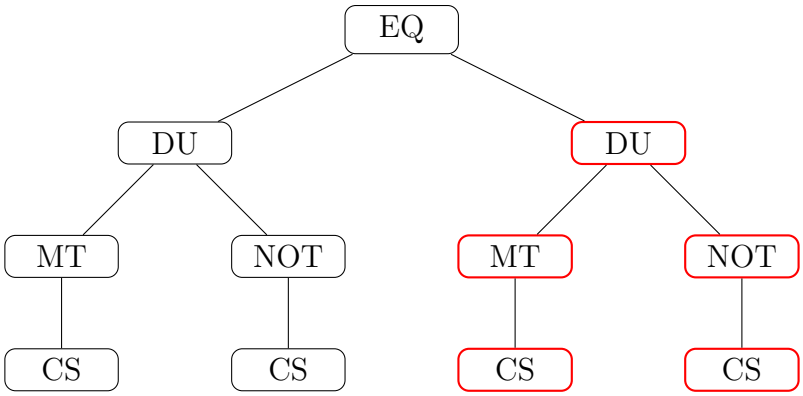


After performing the crossover, the two offspring would be:

Offspring 1:



Offspring 2:



This illustrates how genetic programming crossover can create new offspring by swapping subtrees, potentially leading to improved or diverse solutions in subsequent generations.

2.4 Question 9.4

2.4.1 Description

Consider applying GAs to the task of finding an appropriate set of weights for an artificial neural network (in particular, a feedforward network identical to those trained by BACKPROPAGATION (Chapter 4)). Consider a 3 x 2 x 1 layered, feedforward network. Describe an encoding of network weights as a bit string, and describe an appropriate set of crossover operators. (Hint: Do not allow all possible crossover operations on bit strings.) Finally, state one advantage and one disadvantage of using GAs in contrast to BACKPROPAGATION to train network weights.

2.4.2 Solution

2.4.2.1 Network Architecture and Weight Counting

We have a three-layer feedforward network:

$$\text{Input layer with 3 units,} \quad \text{Hidden layer with 2 units,} \quad \text{Output layer with 1 unit.}$$

This is a fully connected topology with biases. Specifically:

- *Input to Hidden:* 3 inputs $\times$ 2 hidden neurons = 6 weights, plus 2 biases (one per hidden neuron).
- *Hidden to Output:* 2 inputs $\times$ 1 output neuron = 2 weights, plus 1 bias term.

Hence, the total number of weights and biases is:

$$6 + 2 \quad \text{(input-to-hidden layer)} \; + \; 2 + 1 \quad \text{(hidden-to-output layer)} \; = \; 11.$$

Each of these 11 real parameters will be encoded as a segment of the genetic string in our GA.

2.4.2.2 Encoding Weights in a Bit String

One common method is to allocate a fixed-size binary substring (e.g., 16 bits) for each real-valued weight. For instance:

- Restrict each weight to lie in a known numeric range, say $[-5.0,\ 5.0]$.
- Use 16-bit binary representations (two's complement or another scheme) to represent each real-valued weight in that range.

Hence, the chromosome is composed of 11 concatenated 16-bit substrings, one for each weight and bias. This leads to a chromosome of length $11 \times 16 = 176$ bits in total. For example:

$$\underbrace{b_1 b_2 \dots b_{16}}_{\text{weight 1}} \parallel \underbrace{b_{17} \dots b_{32}}_{\text{weight 2}} \parallel \dots \parallel \underbrace{b_{(16\times 10+1)} \dots b_{176}}_{\text{weight 11}}.$$

2.4.2.3 Crossover Operators

Since direct bit-level crossover can disrupt numeric fields in undesirable ways if the split occurs mid-weight, we prefer crossover operators that only swap entire weight blocks:

- **Aligned One-Point or Two-Point Crossover:** Crossover boundaries are allowed *only* between successive 16-bit segments, ensuring that entire weight segments get exchanged.
- **Block-Based Uniform Crossover:** Instead of flipping bits individually, we regard each 16-bit weight block as a unit. With probability p, we swap the corresponding weight block between the two parent chromosomes. This keeps each 16-bit encoding fully intact while combining parent solutions.

2.4.2.4 Advantage of GAs Over Backpropagation

Global Exploration and No Need for Differentiability: GAs do not require gradient information and can jump out of local minima more easily due to population-based global search. This can be particularly useful in problems with non-smooth or discrete aspects, where gradient-based methods (like backpropagation) may struggle.

2.4.2.5 Disadvantage of GAs Over Backpropagation

Slower Convergence and Higher Computational Cost: Compared to gradient-based learning, GAs generally require many generations of population evaluations and can be computationally more expensive to converge. Backpropagation often exploits direct gradient information to train networks efficiently, whereas GAs rely on broader random exploration and selection.