Lab-7

classmate
Date
Page

Gene Expression Algorithm

* Algorithm

1) Initialization

Define constants and parameters:

Set population size P, no of generation G, mutation rate M, crossover rate C and max tree depth D.

Define the function set F(es, +, *, /) and terminal set T

Initialize the population

Generate P random individuals, each represented by a mathematical expression of depth upto 'D'.

2) Evaluate fitness

① For each individual in the population:

Replace the variable x in the individual's expression with a specific value. (eg:- x = 3)

② Evaluate the mathematical expression to calculate fitness

③ If the expression is invalid, assign a high fitness value.

3) Selection

① Identify the best individual in the population

② Store or output the best individual's fitness for the current generation

4) Generate new population, with crossover and mutation

5) Find best individual & its fitness.

Code:-

```
import random
import operator
import math

POPULATION_SIZE = 100
GENERATIONS = 5
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.7
MAX_TREE_DEPTH = 5
FUNCTIONS = ['+', '*', '/']
TERMINALS = ['x', 'y', '2', '3']

class Individual:
    def __init__(self, expression):
        self.expression = expression
        self.fitness = float('inf')

    def evaluate_fitness(self, x_value):
        try:
            expr = self.expression.replace('x', str(x_value))
            self.fitness = eval(expr)
        except Exception as e:
            self.fitness = float('inf')

def generate_random_individual():
    expression = generate_random_expression(MAX_TREE_DEPTH)
    return Individual(expression)
```

```
def generate_random_expression(depth):
    if depth == 0 or random.randint() < 0.7:
        return random.choice(TERMINALS)
    else:
        function = random.choice(FUNCTIONS)
        left = generate_random_expression(depth-1)
        right = generate_random_expression(depth-1)
        return f"({left} {function} {right})"

def crossover(parent1, parent2):
    expr1, expr2 = parent1.expression, parent2.expression
    split1 = random.choice(expr1.split())
    split2 = random.choice(expr2.split())
    offspring_expr = expr1.replace(split1, split2, 1)
    individual.expression = mutated_expr

def mutate(individual):
    if random.random() < MUTATION_RATE:
        mutated_expr = individual.expression
        split_expr = mutated_expr.split()
        mutated_expr = mutated_expr.replace(
            random.choice(split_expr), generate_random_
        expression(MAX_TREE_DEPTH), 1)
        individual.expression = mutated_expr

def select_best_individual(population, x_value):
    best_individual = min(population, key=lambda
                          ind: ind.fitness)
    best_individual.evaluate_fitness(x_value
    return best_individual
```

Output:

Generation 1 : Best fitness = 0.0042

Generation 2 : Best fitness = 0.0066

Generation 3 : Best fitness = 0.04

Generation 4 : Best fitness = 8.74

Generation 5 : Best fitness = 1

18/12/24