ГУАП

КАФЕДРА № 43

ОТЧЕТ  
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| профессор |  |  |  | Ю.А. Скобцов |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

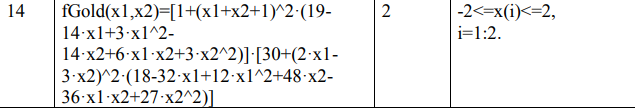
|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №4 |
|  |
| по курсу: ЭВОЛЮЦИОННЫЕ МЕТОДЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНО-ИНФОРМАЦИОННЫХ СИСТЕМ |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 4134 |  |  |  | Д. В. Самарин |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2024

**Вариант 14.**



# Задание:

1. Разработать эволюционный алгоритм, реализующий ГП для нахождения заданной по варианту функции (таб. 4.1).

− Структура для представления программы – древовидное представление.

− Терминальное множество: переменные х1, х2, х3, …, хn, и константы в соответствии с заданием по варианту.

− Функциональное множество: +, -, \*, /, abs(), sin(), cos(), exp(), возведение в степень,

− Фитнесс-функция – мера близости между реальными значениями выхода и требуемыми.

2. Представить графически найденное решение на каждой итерации.

3. Сравнить найденное решение с представленным в условии задачи.

Шаги выполнения алгоритма генетического программирования:

* Задать параметры эволюционного процесса.
* Сформировать начальную популяцию программ.
* Установить начальное значение счётчика поколений.
* Оценить качество всех программ (особей) в текущей популяции с использованием фитнесс-функции.
* Увеличить значение счётчика поколений.
* Провести отбор родителей из текущей популяции для участия в создании потомства.
* Сгенерировать потомков путём выполнения операции кроссинговера между выбранными родителями.
* Применить операцию мутации к новым особям (потомкам).
* Обновить популяцию, добавив новых особей в текущее множество.
* Уменьшить расширенную популяцию до изначального размера.
* Проверить критерий завершения алгоритма. Если он выполнен, выбрать лучшую программу из итоговой популяции как решение задачи. В противном случае повторить шаги с четвёртого пункта.

Графическое представление

На каждой итерации алгоритма построить визуализацию текущего лучшего решения для наглядного отслеживания процесса эволюции.

Сравнение результатов

Сравнить найденное решение с ожидаемым результатом, чтобы оценить эффективность работы алгоритма и его способность решать поставленную задачу.

# Программа

import random

import numpy as np

import matplotlib.pyplot as plt

from scipy.optimize import minimize

# Параметры алгоритма

params = {

    'generations': 100,        # Количество поколений

    'population\_size': 500,    # Размер популяции

    'max\_depth': 8,            # Максимальная глубина дерева

    'mutation\_chance': 0.3     # Вероятность мутации

}

# Функция для оценки (первоначальная функция)

def fGold(x1, x2):

    return (1 + (x1 + x2 + 1)\*\*2 \* (19 - 14\*x1 + 3\*x1\*\*2 - 14\*x2 + 6\*x1\*x2 + 3\*x2\*\*2)) \* \

           (30 + (2\*x1 - 3\*x2)\*\*2 \* (18 - 32\*x1 + 12\*x1\*\*2 + 48\*x2 - 36\*x1\*x2 + 27\*x2\*\*2))

# Поиск реального минимума функции

def find\_real\_minimum():

    bounds = [(-2, 2), (-2, 2)]

    result = minimize(lambda x: fGold(x[0], x[1]), [0, 0], bounds=bounds)

    return result.fun, result.x

# Класс для узла дерева

class Node:

    def \_\_init\_\_(self, value, left=None, right=None):

        self.value = value

        self.left = left

        self.right = right

    def evaluate(self, x1, x2):

        if self.value in ['+', '-', '\*', '/', 'abs', 'sin', 'cos', 'exp']:

            try:

                left\_value = self.left.evaluate(x1, x2) if self.left else None

                right\_value = self.right.evaluate(x1, x2) if self.right else None

                if left\_value is None or right\_value is None:

                    return None  # Пропускаем, если нет значений

                if self.value == '+':

                    return left\_value + right\_value

                elif self.value == '-':

                    return left\_value - right\_value

                elif self.value == '\*':

                    return left\_value \* right\_value

                elif self.value == '/':

                    if right\_value == 0:

                        return None  # Пропускаем деление на 0

                    return left\_value / right\_value

                elif self.value == 'abs':

                    return abs(left\_value)

                elif self.value == 'sin':

                    return np.sin(left\_value)

                elif self.value == 'cos':

                    return np.cos(left\_value)

                elif self.value == 'exp':

                    return np.exp(left\_value)

            except Exception:

                return None  # Возвращаем None при ошибке

        else:

            return float(self.value) if isinstance(self.value, (int, float)) else (x1 if self.value == 'x1' else x2)

    def \_\_str\_\_(self, level=0):

        ret = "\t" \* level + repr(self.value) + "\n"

        if self.left: ret += self.left.\_\_str\_\_(level + 1)

        if self.right: ret += self.right.\_\_str\_\_(level + 1)

        return ret

# Генерация случайного дерева

def generate\_tree(max\_depth, current\_depth=0):

    if current\_depth < max\_depth and random.random() > 0.5:

        func = random.choice(['+', '-', '\*', '/', 'abs', 'sin', 'cos', 'exp'])

        left = generate\_tree(max\_depth, current\_depth + 1)

        right = generate\_tree(max\_depth, current\_depth + 1)

        return Node(func, left, right)

    else:

        return Node(random.choice([random.uniform(-2, 2), 'x1', 'x2']))

# Оценка фитнеса особи

def fitness(individual, real\_minimum):

    output = individual.evaluate(0, 0)

    if output is None:

        return 0

    return 1 / (1 + abs(output - real\_minimum))

# Вероятность ошибки

def error\_probability(found\_value, real\_minimum):

    return abs(found\_value - real\_minimum) / abs(real\_minimum)

# Отбор родителей

def select\_parents(population, real\_minimum):

    weights = [fitness(ind, real\_minimum) for ind in population]

    return random.choices(population, weights=weights, k=2)

# Кроссинговер

def crossover(parent1, parent2):

    child1 = Node(parent1.value)

    child2 = Node(parent2.value)

    child1.left, child1.right = parent1.left, parent2.right

    child2.left, child2.right = parent2.left, parent1.right

    return child1, child2

# Мутация

def mutate(individual, max\_depth):

    if random.random() < params['mutation\_chance']:

        return generate\_tree(max\_depth)

    return individual

# Визуализация дерева с помощью Matplotlib

def plot\_tree(node, pos=(0, 0), level=1, delta\_x=1.0, ax=None):

    if node is None:

        return

    if ax is None:

        \_, ax = plt.subplots(figsize=(10, 8))

        ax.axis("off")

    ax.text(pos[0], pos[1], str(node.value), ha="center", va="center", fontsize=12, bbox=dict(facecolor="white", edgecolor="black"))

    if node.left:

        next\_pos = (pos[0] - delta\_x / 2, pos[1] - 1)

        ax.plot([pos[0], next\_pos[0]], [pos[1], next\_pos[1]], 'k-')

        plot\_tree(node.left, pos=next\_pos, level=level + 1, delta\_x=delta\_x / 2, ax=ax)

    if node.right:

        next\_pos = (pos[0] + delta\_x / 2, pos[1] - 1)

        ax.plot([pos[0], next\_pos[0]], [pos[1], next\_pos[1]], 'k-')

        plot\_tree(node.right, pos=next\_pos, level=level + 1, delta\_x=delta\_x / 2, ax=ax)

# Основной алгоритм генетического программирования

def genetic\_programming():

    population = [generate\_tree(params['max\_depth']) for \_ in range(params['population\_size'])]

    best\_fitnesses, error\_probabilities = [], []

    found\_minima = []

    real\_minimum, \_ = find\_real\_minimum()

    print(f"Real Minimum Value: {real\_minimum}")

    best\_overall\_individual, best\_overall\_fitness, best\_generation = None, 0, 0

    for generation in range(params['generations']):

        new\_population = []

        for \_ in range(params['population\_size'] // 2):

            parent1, parent2 = select\_parents(population, real\_minimum)

            child1, child2 = crossover(parent1, parent2)

            new\_population.extend([mutate(child1, params['max\_depth']), mutate(child2, params['max\_depth'])])

        population = new\_population

        best\_individual = max(population, key=lambda ind: fitness(ind, real\_minimum))

        best\_value = best\_individual.evaluate(0, 0)

        found\_minima.append(best\_value)

        best\_fitness = fitness(best\_individual, real\_minimum)

        best\_fitnesses.append(best\_fitness)

        error\_prob = error\_probability(best\_value, real\_minimum)

        error\_probabilities.append(error\_prob)

        if best\_fitness > best\_overall\_fitness:

            best\_overall\_fitness, best\_overall\_individual, best\_generation = best\_fitness, best\_individual, generation + 1

        print(f"\nGeneration {generation + 1}: Best Fitness = {best\_fitness:.4f}, Best Value = {best\_value:.4f}, Error Probability = {error\_prob:.4f}")

    print(f"\nBest Overall Individual (Generation {best\_generation}):\n{best\_overall\_individual}")

    print(f"Best Overall Value: {best\_overall\_individual.evaluate(0, 0):.4f}")

    print(f"Best Overall Fitness: {best\_overall\_fitness:.4f}")

    # Построение графиков

    plt.figure(figsize=(15, 5))

    plt.subplot(1, 3, 1)

    plt.plot(best\_fitnesses, label='Best Fitness')

    plt.xlabel('Generations')

    plt.ylabel('Best Fitness')

    plt.title('Fitness Progress')

    plt.grid()

    plt.legend()

    plt.subplot(1, 3, 2)

    plt.plot(found\_minima, label='Found Minima', color='orange')

    plt.axhline(real\_minimum, color='red', linestyle='--', label='Real Minimum')

    plt.xlabel('Generations')

    plt.ylabel('Function Value')

    plt.title('Minima Progress')

    plt.grid()

    plt.legend()

    plt.subplot(1, 3, 3)

    plt.plot(error\_probabilities, label='Error Probability', color='green')

    plt.xlabel('Generations')

    plt.ylabel('Error Probability')

    plt.title('Error Probability Progress')

    plt.grid()

    plt.legend()

    plt.tight\_layout()

    plt.show()

    # Визуализация лучшего дерева

    plot\_tree(best\_overall\_individual)

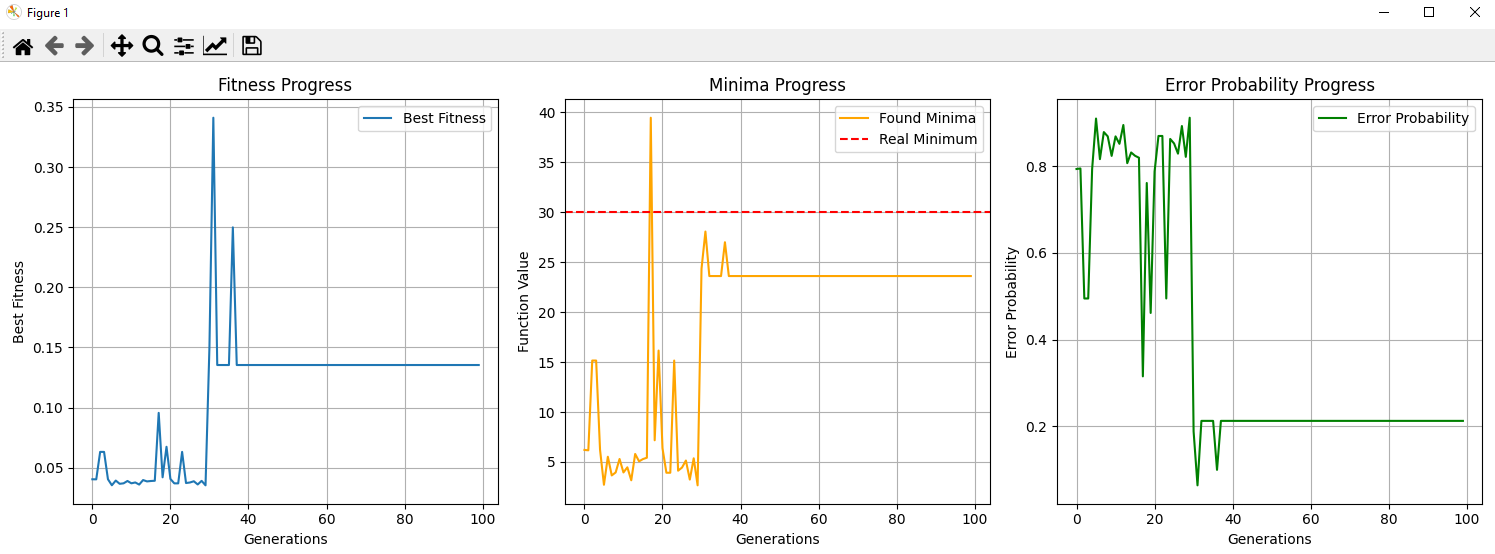
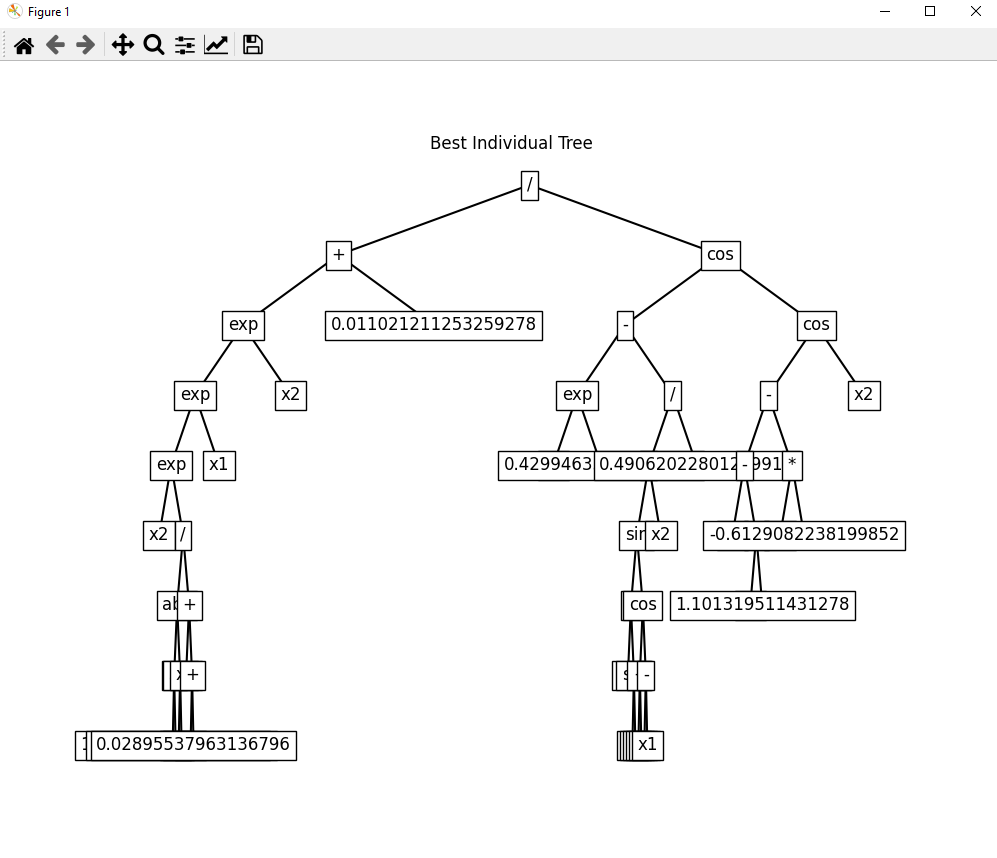
    plt.title('Best Individual Tree')

    plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

    genetic\_programming()

# Результат работы программы

**Результат исследований:**

|  |
| --- |
| Generation 100: Best Fitness = 0.1354, Best Value = 23.6129, Error Probability = 0.2129  Best Overall Individual (Generation 32):  '/'  '+'  'exp'  'exp'  'exp'  'x2'  '/'  'abs'  '+'  1.7139448692844077  'x1'  'sin'  'x1'  -0.681127581437829  '+'  'x2'  '+'  'x2'  0.02895537963136796  'x1'  'x2'  0.011021211253259278  'cos'  '-'  'exp'  'x1'  0.42994633512263425  '/'  '-'  'sin'  '\*'  'x2'  'sin'  'x2'  'x1'  'cos'  '+'  'x1'  'x1'  '-'  'x1'  'x1'  'x2'  0.49062022801279914  'cos'  '-'  '-'  'x2'  '+'  'x1'  1.101319511431278  '\*'  'x1'  -0.6129082238199852  'x2'  Best Overall Value: 28.0681  Best Overall Fitness: 0.3411 |

# Вывод

Выполнив ЛР, я реализовал ГА для поиска минимума функции, используя деревья для представления математических выражений. Удалось найти выражение, приближающееся к реальному минимуму функции, что подтвердило эффективность использованного подхода и его потенциал для решения задач оптимизации в различных областях.