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
Генетическое программирование

по дисциплине: Эволюционные методы проектирования программно-информационных систем

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР.	4134к		Столяров Н.С.
		подпись, дата	инициалы, фамилия

Санкт-Петербург
2024

Цель работы:
Решение задачи символьной регрессии. Графическое отображение результатов оптимизации.

Вариант 2

2	f1a(x)=sum(i·x(i)^2), i=1:n;	9	-5.12<=x(i)<=5.12.

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 x1, x2, x3, …, xn, 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.

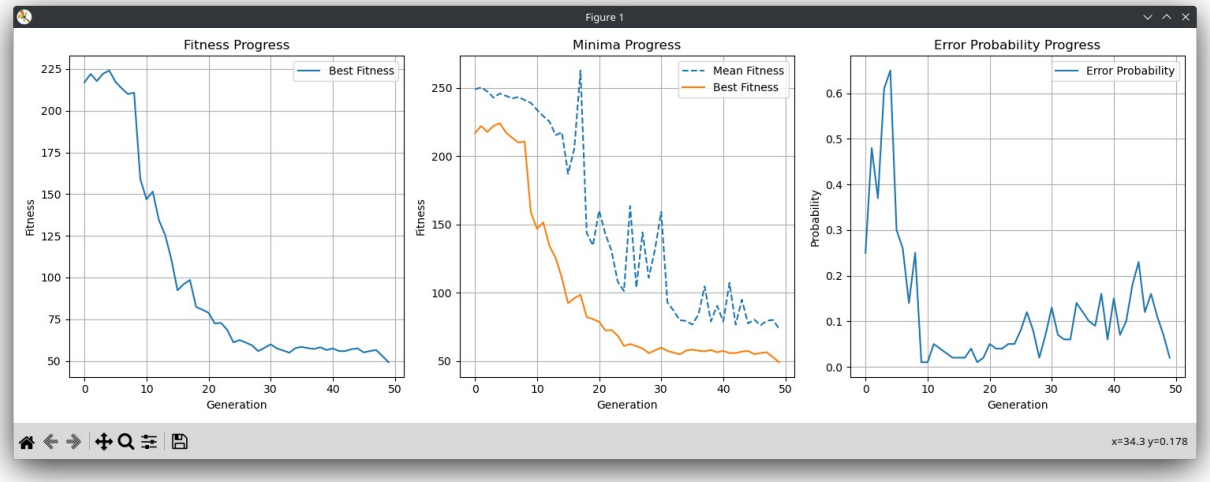
Общий алгоритм генетического программирования

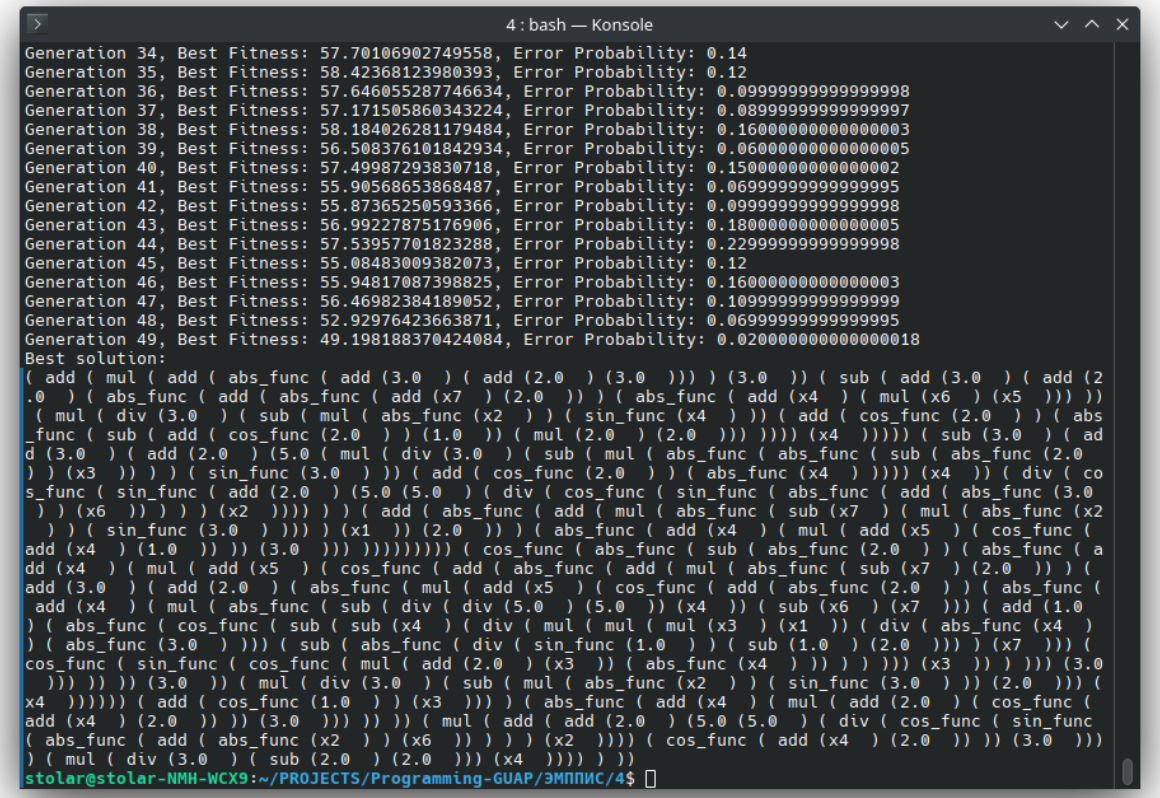
Таким образом, для решения задачи с помощью ГП необходимо выполнить описанные выше предварительные этапы:
1) Определить терминальное множество;
2) Определить функциональное множество;
3) Определить фитнесс-функцию;
4) Определить значения параметров, такие как мощность популяции, максимальный размер особи, вероятности кроссинговера и мутации, способ отбора родителей, критерий окончания эволюции (например, максимальное число поколений) и т. п. После этого можно разрабатывать непосредственно сам эволюционный алгоритм, реализующий ГП для конкретной задачи. Например, решение задачи на основе ГП можно представить следующей последовательностью действий.
1) установка параметров эволюции;
2) инициализация начальной популяции;
3) t: = 0;
4) оценка особей, входящих в популяцию;
5) t: = t+1;
6) отбор родителей;
7) создание потомков выбранных пар родителей – выполнение оператора кроссинговера;
3
8) мутация новых особей;
9) расширение популяции новыми порожденными особями;
10) сокращение расширенной популяции до исходного размера;
11) если критерий останова алгоритма выполнен, то выбор лучшей особи в конечной популяции – результат работы алгоритма.
Иначе переход на шаг 4.

Опишите линейное представление программы

Линейное представление программы — это способ организации и записи программы в виде последовательности команд или инструкций, расположенных в строго определённом порядке. В таком представлении программа выполняется шаг за шагом, без ветвлений или циклов, если не считать их явной развёртки.

Ход Работы





Выводы:
В данной работе был реализован алгоритм генетического программирования для поиска минимума функции, используя деревья для представления математических выражений. Проведён анализ полученных решений, где удалось найти выражение, приближающееся к реальному минимуму функции, что подтвердило эффективность использованного подхода и его потенциал для решения задач оптимизации в различных областях

Листинг программы

```
import copy
import random
import math

from decimal import Decimal, getcontext
from decimal import InvalidOperation
from decimal import Overflow
getcontext().prec = 10
getcontext().traps[InvalidOperation] = False
getcontext().traps[Overflow] = False

# Функции для представления операций
def add(x: float, y: float) -> float:
    return x + y
```

```
def sub(x: float, y: float) -> float:
    return x - y


def mul(x: float, y: float) -> float:
    return x * y


def div(x: float, y: float) -> float:
    if y != float(0):
        return x / y
    else:
        return float(1)


def abs_func(x: float, y: float) -> float:
    return abs(x)


def sin_func(x: float, y: float) -> float:
    return float(math.sin(float(x)))


def cos_func(x: float, y: float) -> float:
    return float(math.cos(float(x)))


def exp_func(x: float, y: float) -> float:
    return float(math.exp(float(x)))


def power(x: float, y: float) -> float:
    if x == float(0):
        return float(0)
    y = y.quantize(float('1'))
    return float(x ** y)

# Типы узлов
FUNCTIONS = [add, sub, mul, div, abs_func, sin_func, cos_func]
TERMINALS = ['x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', float(1), float(2),
float(3), float(5)]  # Переменные и константы


class Node:
    def __init__(self, value=None, left=None, right=None):
        self.value = value  # Это будет либо функция, либо терминал
        self.left = left
        self.right = right  # для бинарных операторов
```

```
    def evaluate(self, variables):
        """Рекурсивная функция для вычисления значения дерева"""
        try:
            if self.value in TERMINALS:
                if isinstance(self.value, str):
                    return variables[self.value]  # возвращаем значение
переменной
                return self.value  # возвращаем константу
            else:
                # Применяем функцию на основе значения
                left = self.left.evaluate(variables) if self.left is not
None else None
                right = self.right.evaluate(variables) if self.right is not
None else None
                return self.value(left, right)
        except (OverflowError, ZeroDivisionError):
            return None  # Возвращаем None в случае ошибки

    def type(self):
        if self.value in TERMINALS:
            return "terminal"
        else:
            return "function"


class Tree:
    def __init__(self):
        self.root = None

    def create(self, grow=True, max_depth=5):
        self.root = self._create_tree(0, max_depth, grow)

    def get_random_node(self):
        total_nodes = self._count_nodes(self.root)
        random_index = random.randint(0, total_nodes - 1)
        return self._get_random_node(self.root, random_index)

    def evaluate(self, variables):
        result = self.root.evaluate(variables)
        return result

    def _create_tree(self, depth, max_depth, grow=False):
        """Рекурсивно создаем дерево с максимальной глубиной max_depth"""
        if depth == max_depth:
            value = random.choice(TERMINALS)
            return Node(value)
        else:
            if grow:
                node_is_terminal = random.random()
```

```
                if node_is_terminal < 0.4:
                    value = random.choice(TERMINALS)
                    return Node(value)
            func = random.choice(FUNCTIONS)
            if func in [add, sub, mul, div, power]:
                left = self._create_tree(depth + 1, max_depth, grow)
                right = self._create_tree(depth + 1, max_depth, grow)
                return Node(func, left, right)
            else:
                left = self._create_tree(depth + 1, max_depth, grow)
                return Node(func, left)

    def _count_nodes(self, node: Node):
        if node is None:
            return 0
        left_size = self._count_nodes(node.left)
        right_size = self._count_nodes(node.right)
        return 1 + left_size + right_size

    def _get_random_node(self, node, index):
        if node is None:
            return None
        left_size = self._count_nodes(node.left)
        if index == left_size:
            return node
        elif index < left_size:
            return self._get_random_node(node.left, index)
        else:
            return self._get_random_node(node.right, index - left_size - 1)

    def print(self):
        self._print(self.root)

    def print_function(self):
        return self._print_function(self.root)

    def _print(self, node: Node, depth=0):
        print(depth * "\t", node.value, "-", depth)
        if node.left is not None:
            self._print(node.left, depth + 1)
        if node.right is not None:
            self._print(node.right, depth + 1)

    def _print_function(self, node: Node):
        if node is None:
            return ""
        value = str(node.value)
        if "function" in value:
            value = value.split("function")[1].split(" at")[0]
        return "(" + value + " " + self._print_function(node.left) + " " +
```

```
self._print_function(node.right) + ")"

#№№№№№№№№№№№№№№№№№№№№№№№№№№№№№№№№

# Генерация данных
def generate_variables():
    """Генерация набора переменных в диапазоне [-5.12, 5.12]"""
    return {f'x{i+1}': random.uniform(-5.12, 5.12) for i in range(7)}

# Целевая функция
def target_function(variables):
    """Целевая функция f(x) = Σ(i * x(i)^2), где -5.12 <= x(i) <= 5.12"""
    return sum((i + 1) * variables[f'x{i+1}']**2 for i in
range(len(variables)))

# Оценка фитнеса
def fitness_function(tree, target_function, variables):
    """Вычисляем фитнес для дерева, сравнивая его результат с целевой
функцией"""
    predicted = tree.evaluate(variables)
    if predicted is None:  # Если дерево не смогло вычислить результат
(например, деление на 0)
        return float('inf')  # Задаем плохой фитнес
    target = target_function(variables)
    return abs(predicted - target)  # Фитнес как разница между
предсказанием и целевой функцией

# Расчет фитнеса для всей популяции
def calculate_fitness(population):
    for individual in population:
        fitness = 0
        samples = 100
        for _ in range(samples):
            variables = generate_variables()  # Генерируем случайный набор
переменных
            fitness += fitness_function(individual.tree, target_function,
variables)
        individual.fitness = fitness / samples  # Усредняем фитнес по всем
примерам
    return population

# Инициализация популяции
def initialize_population(size, max_depth):
    population = []
    for _ in range(size):
        individual = Individual()
        individual.tree.create(max_depth=max_depth,
grow=random.choice([True, False]))
        population.append(individual)
    return population
```

```
class Individual:
    def __init__(self):
        self.tree = Tree()
        self.fitness = float('inf')

# №№№№№№№№№№№№№№№№№№№№№№№

# Генерация данных
def generate_variables():
    """Генерация набора переменных в диапазоне [-5.12, 5.12]"""
    return {f'x{i+1}': random.uniform(-5.12, 5.12) for i in range(7)}

# Целевая функция
def target_function(variables):
    """Целевая функция f(x) = Σ(i * x(i)^2), где -5.12 <= x(i) <= 5.12"""
    return sum((i + 1) * variables[f'x{i+1}']**2 for i in
range(len(variables)))

# Оценка фитнеса
def fitness_function(tree, target_function, variables):
    """Вычисляем фитнес для дерева, сравнивая его результат с целевой
функцией"""
    predicted = tree.evaluate(variables)
    if predicted is None:  # Если дерево не смогло вычислить результат
(например, деление на 0)
        return float('inf')  # Задаем плохой фитнес
    target = target_function(variables)
    return abs(predicted - target)  # Фитнес как разница между
предсказанием и целевой функцией

# Расчет фитнеса для всей популяции
def calculate_fitness(population):
    for individual in population:
        fitness = 0
        samples = 100
        for _ in range(samples):
            variables = generate_variables()  # Генерируем случайный набор
переменных
            fitness += fitness_function(individual.tree, target_function,
variables)
        individual.fitness = fitness / samples  # Усредняем фитнес по всем
примерам
    return population

# Инициализация популяции
def initialize_population(size, max_depth):
    population = []
    for _ in range(size):
        individual = Individual()
```

```
        individual.tree.create(max_depth=max_depth,
grow=random.choice([True, False]))
        population.append(individual)
    return population

class Individual:
    def __init__(self):
        self.tree = Tree()
        self.fitness = float('inf')

# №№№№№№№№№№№№№№№№№№№№№№№№№№№№№№

# Селекция методом турнира
def tournament_selection(population, tournament_size=3):
    tournament = random.sample(population, tournament_size)
    winner = min(tournament, key=lambda individual: individual.fitness)
    return copy.deepcopy(winner)

# Кроссовер (одноточечный)
def crossover(parent1, parent2):
    child1, child2 = copy.deepcopy(parent1), copy.deepcopy(parent2)

    node1 = child1.tree.get_random_node()
    node2 = child2.tree.get_random_node()

    # Меняем поддеревья местами
    node1.value, node2.value = node2.value, node1.value
    node1.left, node2.left = node2.left, node1.left
    node1.right, node2.right = node2.right, node1.right

    return child1, child2

# Мутация
def mutation(individual, max_depth):
    mutant = copy.deepcopy(individual)
    mutation_node = mutant.tree.get_random_node()

    if mutation_node:
        mutation_node.value = random.choice(FUNCTIONS + TERMINALS)
        if mutation_node.type() == "function":
            mutation_node.left = mutant.tree._create_tree(depth=0,
max_depth=max_depth, grow=True)
            mutation_node.right = mutant.tree._create_tree(depth=0,
max_depth=max_depth, grow=True)
    return mutant

import matplotlib.pyplot as plt

def plot_progress(best_fitness_history, mean_fitness_history,
error_probability_history):
```

```
    """Отображение графиков Fitness Progress, Minima Progress и Error
Probability Progress."""
    generations = range(len(best_fitness_history))

    # Fitness Progress
    # plt.figure(figsize=(15, 5))
    plt.figure(figsize=(15, 5))
    plt.subplot(1, 3, 1)
    plt.plot(generations, best_fitness_history, label="Best Fitness")
    plt.title("Fitness Progress")
    plt.xlabel("Generation")
    plt.ylabel("Fitness")
    plt.legend()
    plt.grid()

    # Minima Progress
    plt.subplot(1, 3, 2)
    plt.plot(generations, mean_fitness_history, label="Mean Fitness",
linestyle="--")
    plt.plot(generations, best_fitness_history, label="Best Fitness")
    plt.title("Minima Progress")
    plt.xlabel("Generation")
    plt.ylabel("Fitness")
    plt.legend()
    plt.grid()

    # Error Probability Progress
    plt.subplot(1, 3, 3)
    plt.plot(generations, error_probability_history, label="Error
Probability")
    plt.title("Error Probability Progress")
    plt.xlabel("Generation")
    plt.ylabel("Probability")
    plt.legend()
    plt.grid()

    plt.tight_layout()
    plt.show()


# Основной цикл с обновлением истории
def genetic_algorithm(population_size, max_depth, generations,
crossover_prob, mutation_prob):
    # Инициализация популяции
    population = initialize_population(population_size, max_depth)

    # История для графиков
    best_fitness_history = []
    mean_fitness_history = []
    error_probability_history = []
```

```
    # Основной цикл
    for generation in range(generations):
        # Расчет фитнеса
        calculate_fitness(population)
        population.sort(key=lambda individual: individual.fitness)

        # Обновление истории
        best_fitness = population[0].fitness
        mean_fitness = sum(ind.fitness for ind in population) /
len(population)
        # error_probability = sum(1 for ind in population if ind.fitness >
1e-6) / len(population)
        # error_probability = sum(1 for ind in population if ind.fitness >
1e-6) / len(population)
        # error_probability = 1 - (sum(1 for ind in population if
ind.fitness <= 1e-6) / len(population))
        error_probability = 1 - (sum(1 for ind in population if ind.fitness
> best_fitness * 1.1) / len(population))

        best_fitness_history.append(best_fitness)
        mean_fitness_history.append(mean_fitness)
        error_probability_history.append(error_probability)

        # Вывод информации
        print(f"Generation {generation}, Best Fitness: {best_fitness},
Error Probability: {error_probability}")
        # print(f"Generation {generation}, Best Fitness: {best_fitness}")

        # Если достигли удовлетворительного результата, завершаем
        if best_fitness < 1e-6:
            break

        # Создание нового поколения
        new_population = []
        while len(new_population) < population_size:
            # Селекция
            parent1 = tournament_selection(population)
            parent2 = tournament_selection(population)

            # Кроссовер
            if random.random() < crossover_prob:
                child1, child2 = crossover(parent1, parent2)
            else:
                child1, child2 = parent1, parent2

            # Мутация
            if random.random() < mutation_prob:
                child1 = mutation(child1, max_depth)
            if random.random() < mutation_prob:
```

```
                child2 = mutation(child2, max_depth)

            new_population.extend([child1, child2])

        # Обновляем популяцию
        population = new_population[:population_size]

    # Возвращаем лучшего индивида
    calculate_fitness(population)
    best_individual = min(population, key=lambda individual:
individual.fitness)

    # Отображение графиков
    plot_progress(best_fitness_history, mean_fitness_history,
error_probability_history)

    return best_individual

from tree import visualize_tree

# Запуск алгоритма
if __name__ == "__main__":
    best = genetic_algorithm(
        population_size=100,
        max_depth=5,
        generations=50,
        crossover_prob=0.7,
        mutation_prob=0.2
    )
    print("Best solution:")
    print(best.tree.print_function())
    visualize_tree(best.tree)
```