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		Костяков Н.А.
		подпись, дата	инициалы, фамилия

Санкт-Петербург
2024

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

Вариант 4

4	f2(x)=sum(100·(x(i+1)-x(i)^2)^2+(1-x(i))^2), i=1:n-1;	7	-2.048<=x(i)<=2.048.

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.

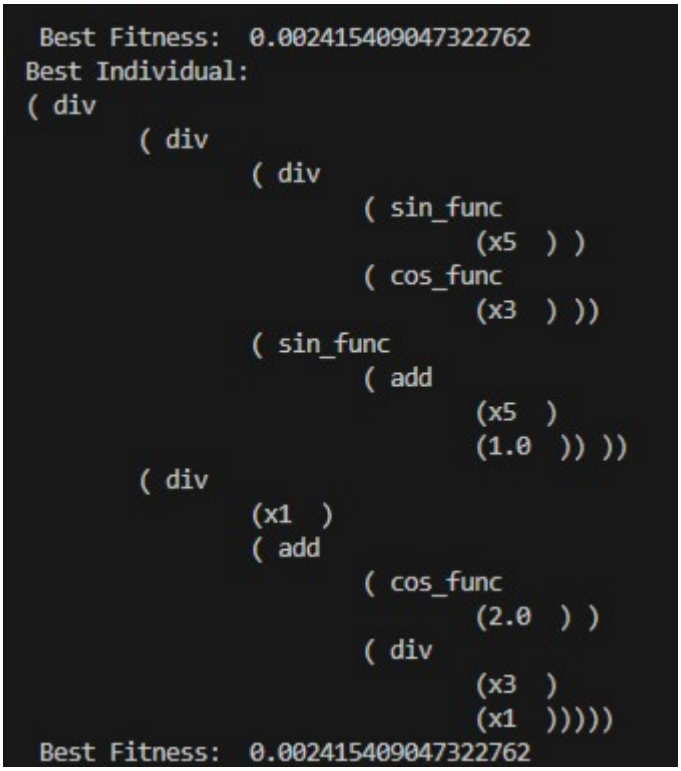
Общий алгоритм генетического программирования

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
```

```python
# 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)
    #print("R",total_nodes,random_index)
    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:
      # Возвращаем терминал
      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]:
            # Двухаргументные функции
            left = self._create_tree(depth + 1, max_depth,grow)
            right = self._create_tree(depth + 1, max_depth,grow)
            return Node(func, left, right)
        else:
            # Одноаргументные функции
            left = self._create_tree(depth + 1, max_depth,grow)
            return Node(func, left)

    def _count_nodes(self, node: Node):
        # Рекурсивный подсчёт узлов в поддереве
        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):
        # Рекурсивный поиск случайного узла с данным индексом
        if node is None:
            return None
        left_size = self._count_nodes(node.left)

        if index == left_size:  # Мы нашли нужную вершину
            return node
        elif index < left_size:  # Ищем в левом поддереве
            return self._get_random_node(node.left, index)
        else:  # Ищем в правом поддереве, корректируем индекс
            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):
        children = self._count_nodes(node)
        print(depth*"\t",node.value,":", children, "-", 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 len(value.split("function"))!=1:
      value = value.split("function")[1].split(" at")[0]
    depth = _get_node_height(self.root,node)
    return "\n" + "\t"*depth + "("+ value + " " + self._print_function(node.left)+
" " + self._print_function(node.right)+")"
def _get_node_height(root: Node, target_node: Node):
  # Рекурсивно определяет высоту целевого узла в дереве
  if root is None:
    return -1
  if root == target_node:
    return 0
  left_height = _get_node_height(root.left, target_node)
  if left_height >= 0:
    return left_height + 1
  right_height = _get_node_height(root.right, target_node)
  if right_height >= 0:
    return right_height + 1
  return -1
def is_compatible(node1, node2):
  """Проверяем совместимость двух поддеревьев (по типу узлов)."""
  # Проверка на бинарные узлы
  if (node1.left is not None ) and (node2.left is not None ):
    return 1  # Оба бинарные узлы
  if (node1.left is not None and node1.right is not None) and (node2.left is not
None and node2.right is not None):
    return 2  # Оба бинарные узлы
  # Проверка на унарные узлы (с одним дочерним узлом)
  if (node1.left is None and node1.right is None) and (node2.left is None and
node2.right is None):
    return 2  # Оба терминальные узлы
  # Проверка на унарные узлы
  if (node1.left is None and node1.right is not None) and (node2.left is None and
node2.right is not None):
    return 2  # Оба унарные узлы
  return 0  # Узлы несовместимы
from copy import deepcopy
def subtree_crossover(tree1: Tree, tree2: Tree,maxHeight : int):
  # Получаем случайные узлы (поддеревья) в каждом из деревьев
  # Получаем случайный узел в первом дереве
  node1 = tree1.get_random_node()

  # Если узел пустой, возвращаем исходные деревья без изменений
  if node1 is None:
    return tree1, tree2
  nodes_count = tree2._count_nodes(tree2.root)*100
  # Получаем узел в tree2, подходящий по типу
  node2 = tree2.get_random_node()
  while node2 is not None and  node1.type() != node2.type() and nodes_count > 0:
    node2 = tree2.get_random_node()
    nodes_count -= 1
```

```
    if nodes_count == 0:
        return tree1, tree2


    # Временные деревья для проверки высоты после кроссовера
    temp_tree1 = deepcopy(tree1)
    temp_tree2 = deepcopy(tree2)

    # Выполняем пробный обмен поддеревьями
    _replace_node(temp_tree1, node1, node2)
    _replace_node(temp_tree2, node2, node1)

    # Проверяем высоту обоих временных деревьев
    #if _get_tree_height(temp_tree1.root) <= maxHeight and
_get_tree_height(temp_tree2.root) <= maxHeight:
    #    # Если высота допустима, выполняем кроссинговер на оригиналах
    #    _replace_node(tree1, node1, node2)
    #    _replace_node(tree2, node2, node1)

    if _get_tree_height(temp_tree1.root)<=maxHeight:
        _replace_node(tree1, node1, node2)
    if _get_tree_height(temp_tree2.root)<=maxHeight:
        _replace_node(tree2, node2, node1)
    # Если высота превышает допустимую, возвращаем исходные деревья
    return deepcopy(tree1), deepcopy(tree2)

def _get_tree_height(node: Node):
    # Рекурсивное вычисление высоты дерева
    if node is None:
        return 0
    left_height = _get_tree_height(node.left)
    right_height = _get_tree_height(node.right)
    return 1 + max(left_height, right_height)

def _replace_node(tree: Tree, target: Node, new_subtree: Node):
    # Функция для замены узла в дереве
    if tree.root == target:
        tree.root = new_subtree
    else:
        _replace_node_recursive(tree.root, target, new_subtree)

def _replace_node_recursive(current: Node, target: Node, new_subtree: Node):
    # Рекурсивный поиск целевого узла для замены
    if current.left == target:
        current.left = new_subtree
    elif current.right == target:
        current.right = new_subtree
    else:
        if current.left:
            _replace_node_recursive(current.left, target, new_subtree)
```

```
    if current.right:
        _replace_node_recursive(current.right, target, new_subtree)



def fitness_function(tree, target_function, variables):
    """Вычисляем фитнес для дерева, сравнивая с целевой функцией"""
    predicted = 0
    predicted += tree.evaluate(variables)
    return abs(predicted - target_function(variables))


def node_mutation(tree: Tree):
    # Выбираем случайный узел для замены
    target_node = tree.get_random_node()
    if target_node is None:
        return tree

    # Замена функции или терминала
    if target_node.value in FUNCTIONS:
        if target_node.value in [abs_func, sin_func, cos_func]:
            target_node.value = random.choice([abs_func, sin_func, cos_func])
        else:
            target_node.value = random.choice([add, sub, mul, div])

    elif target_node.value in TERMINALS:
        target_node.value = random.choice(TERMINALS)

    return tree


def pruning_mutation(tree: Tree):
    # Выбираем случайный узел для усечения
    target_node = tree.get_random_node()
    if target_node is None:
        return tree

    # Превращаем узел в терминал
    target_node.value = random.choice(TERMINALS)
    target_node.left = None
    target_node.right = None

    return tree


def growing_mutation(tree: Tree, max_height: int):
    # Выбираем случайный узел для замены
    target_node = tree.get_random_node()
    if target_node is None:
        return tree
```

```
    # Определяем текущую высоту целевого узла и оставшуюся допустимую высоту
    current_height = _get_node_height(tree.root, target_node)
    remaining_height = max_height - current_height

    # Если оставшаяся высота позволяет рост, создаем новое поддерево
    if remaining_height > 0:
        # Создаем новое поддерево с ограничением по оставшейся высоте
        new_subtree = tree._create_tree(current_height, max_height, grow=False)

        # Заменяем целевой узел новым поддеревом
        target_node.value = new_subtree.value
        target_node.left = new_subtree.left
        target_node.right = new_subtree.right

    return tree


def get_tree_size(node):
    """Вычисляем размер дерева (количество узлов)."""
    if node is None:
        return 0
    return 1 + get_tree_size(node.left) + get_tree_size(node.right)



def selection(population: list[Tree], fitness: list[float], k=3) -> list[Tree]:
    candidates_indices = random.choices(range(len(population)), k=k)


def crossover(parent1, parent2, max_size,chance=0.5):
    """Оператор кроссинговера поддеревьев"""
    if random.random() > chance:
        return parent1, parent2
    new_parent1, new_parent2 = subtree_crossover(parent1, parent2, max_size)
    return new_parent1, new_parent2


def mutation(tree, max_size, chance=0.1):
    if random.random() > chance:
        return tree


    """Оператор мутации"""
    mutation_type = random.choice(['node', 'pruning', 'growing'])  # Выбор типа
мутации
    if mutation_type == 'node':
        return node_mutation(tree)
    elif mutation_type == 'pruning':
        return pruning_mutation(tree)
    elif mutation_type == 'growing':
```

```
        return growing_mutation(tree, max_size)
    return tree


# Оценка фитнеса: сумма квадратов отклонений от целевой функции
def calculate_fitness(population):
    for individual in population:

        fitness = 0
        samples = 100
        # Генерация случайных чисел с плавающей точкой
        x1, x2, x3, x4, x5, x6, x7 = [random.uniform(-2, 2) for _ in range(7)]
        for _ in range(samples):

            predicted = individual.tree.evaluate({'x1': float(x1), 'x2': float(x2), 'x3':
float(x3), 'x4': float(x4), 'x5': float(x5), 'x6': float(x6), 'x7': float(x7)})

            target = target_function(float(x1), float(x2), float(x3), float(x4),
float(x5), float(x6), float(x7))  # Целевая функция (x1, x2, x3, x4, x5)

            fitness += ((predicted - target) ** 2)

        individual.fitness = fitness


    return population
def target_function(x1, x2, x3, x4, x5, x6, x7):
    queue=[x1, x2, x3, x4, x5, x6, x7]
    for i in range(len(queue)-1):
        res = 100*(queue[i+1]-queue[i]**2)**2+(1-queue[i])**2

    return res
class Individual:
    def __init__(self, tree: Tree, fitness: float):
        self.tree = tree
        self.fitness = fitness

def initialize_population(pop_size, max_depth) -> list[Individual]:
    """Создаем популяцию деревьев"""
    population = []
    depth = 0
    is_grow = 1
    for _ in range(pop_size):
        tree = Tree()
        tree.create(is_grow, max_depth=depth)
        population.append(Individual(tree, float(0.0)))
        is_grow = not is_grow
        depth = depth % max_depth
    return population

def tournament_selection(population, tournament_size):
```

```
    tournament = random.sample(population, tournament_size)
    sorted_tournament = deepcopy(sorted(tournament, key=lambda individual:
individual.fitness))
    return sorted_tournament[0], sorted_tournament[1]

def elitism_selection(population, elite_size):
    sorted_population = sorted(population, key=lambda individual: individual.fitness)
    return deepcopy(sorted_population[:elite_size])


def genetic_algorithm(population, max_generations, max_size, tournament_size,
elite_size, crossover_chance=0.5,
                      mutation_chance=0.1):
    # Шаг 1: Оценка фитнеса каждого индивида
    population = calculate_fitness(population)

    # Главный цикл генетического алгоритма
    for generation in range(max_generations):
        # Шаг 2: Отбор с использованием турнира
        selected_individuals = []
        while len(selected_individuals) < len(population) - elite_size:
            parent1, parent2 = tournament_selection(population, tournament_size)
            selected_individuals.append(parent1)
            selected_individuals.append(parent2)

        # Шаг 3: Элита - выбираем лучшие особи
        elite_individuals = elitism_selection(population, elite_size)
        selected_individuals.extend(elite_individuals)

        # Шаг 4: Создание новой популяции
        next_generation = []

        # Применение кроссовера и мутации
        for i in range(0, len(selected_individuals), 2):
            parent1 = selected_individuals[i]
            parent2 = selected_individuals[i + 1] if i + 1 < len(selected_individuals)
else selected_individuals[i]

            # Применяем кроссовер
            parent1.tree, parent2.tree = crossover(parent1.tree, parent2.tree, max_size,
crossover_chance)

            # Применяем мутацию
            parent1.tree = mutation(parent1.tree, max_size, mutation_chance)
            parent2.tree = mutation(parent2.tree, max_size, mutation_chance)

            # Добавляем в следующее поколение
            next_generation.append(parent1)
            next_generation.append(parent2)
```

```
        # Обновляем популяцию
        population.extend(next_generation)
        population = calculate_fitness(population)

        sorted_population = copy.copy(sorted(population, key=lambda individual:
individual.fitness))
        population = (sorted_population[:len(sorted_population)//2])

        # Печать состояния на текущем шаге (например, фитнес лучшего индивида)
        best_individual = min(population, key=lambda individual: individual.fitness)
        print(f"Best Individual: {best_individual.tree.print_function()} - Best
Fitness: {best_individual.fitness}")

    # Возвращаем лучший результат после всех поколений
    best_individual = min(population, key=lambda individual: individual.fitness)
    return best_individual


def main():
    print("Hello")
    getcontext().prec = 10
    print(getcontext())
    population = initialize_population(500, 2)
    best_individual = genetic_algorithm(population, 100, 4, 5, 3, 0.5, 0.3)
    print("Best Individual: ", best_individual.tree.print_function(),"\n Best
Fitness: ", best_individual.fitness)
    print("Best Individual: ", best_individual.tree.print_function(),"\n Best
Fitness: ", best_individual.fitness)



if __name__ == "__main__":
    main()
```