МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

АЭРОКОСМИЧЕСКОГО ПРИБО6РОСТРОЕНИЯ»

КАФЕДРА № 43

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Старший преподаватель |  |  |  | С.А. Рогачев |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ |
| «Генетическое программирование» |
| по курсу: ЭВОЛЮЦИОННЫЕ МЕТОДЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНО-ИНФОРМАЦИОННЫХ СИСТЕМ |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 4136 |  | 7.11.24 |  | А.С. Вилюмсон |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2024

**1. Задание по варианту**

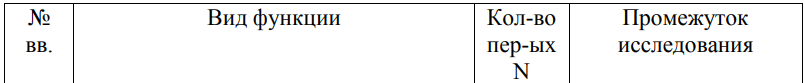
1. Разработать эволюционный алгоритм, реализующий ГП для нахождения

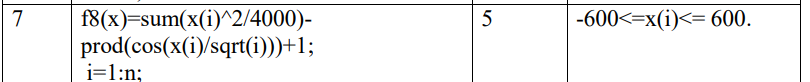
заданной по варианту функции (таб. 4.1).

* 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.***Вариант 7 — eil51.tsp***

******



**2. Основные сведения из теории**

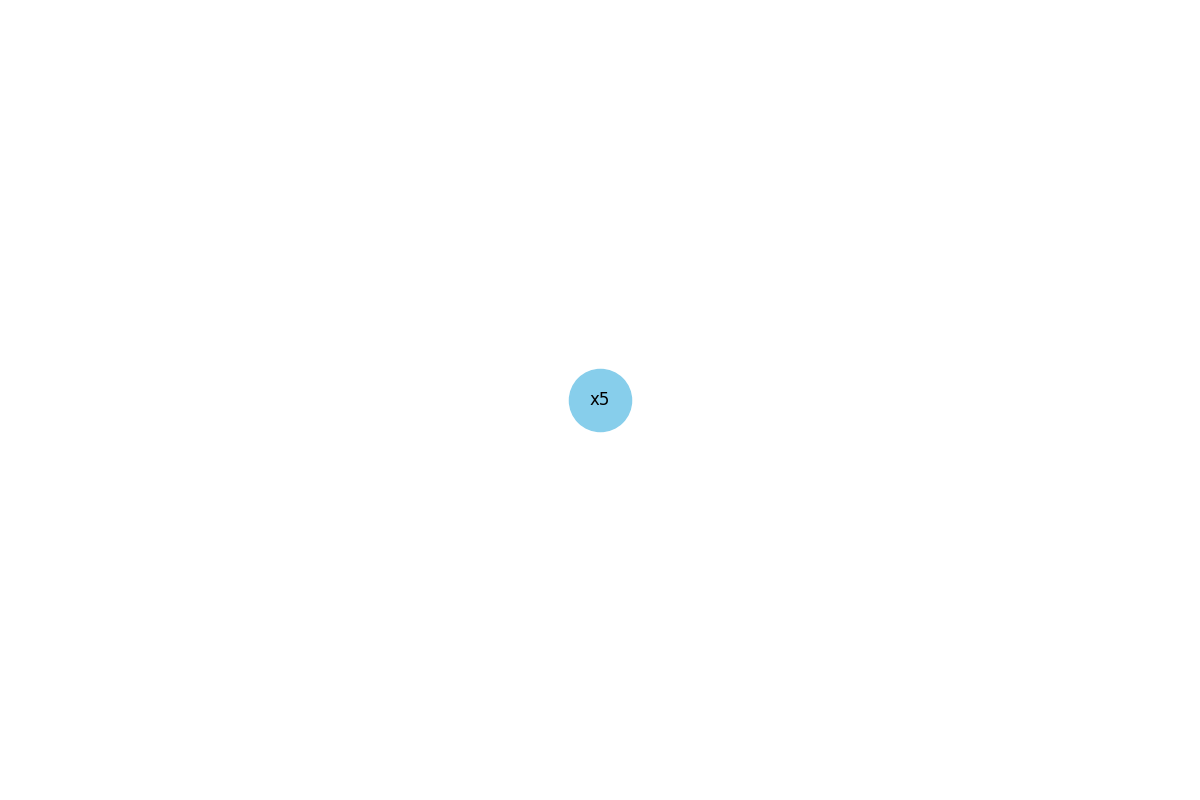
В генетическом программировании (ГП) в качестве особи выступает программа, представленная в определенном формате, которая решает некоторую задачу. Часто это выполняется с использованием обучающих данных и индуктивного вывода. ГП очень близко к машинному обучению и поэтому в качестве фитнесс-функции как правило выступают функции ошибки. ГП работает с генетическим материалом переменной длины, что требует нестандартной формы представления генома и соответствующих генетических операторов. Программы составляются из переменных, констант и функций, которые связаны некоторыми синтаксическими правилами. Поэтому определяется терминальное множество, содержащее константы и переменные, и функциональное множество, которое состоит, прежде всего, из операторов и необходимых элементарных функций (exp(x), sin(x) и т.п.). Следует отметить, что терминалы и функции играют различную роль. Терминалы обеспечивают входные значения в систему (программу), в то время как функции используются при обработке значений внутри системы. Термины «функции» и «терминалы» взяты из древовидного представления, и соответствуют узлам древовидных (или графоподобных) структур.

**3. Листинг программы**

import copy  
import random  
import math  
import matplotlib.pyplot as plt  
import networkx as nx  
  
  
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: Decimal, y: Decimal) -> Decimal:  
 return x + y  
  
  
def sub(x: Decimal, y: Decimal) -> Decimal:  
 return x - y  
  
  
def mul(x: Decimal, y: Decimal) -> Decimal:  
 return x \* y  
  
  
def div(x: Decimal, y: Decimal) -> Decimal:  
 if y != Decimal(0):  
 return x / y  
 else:  
 return Decimal(1)  
  
  
def abs\_func(x: Decimal,y: Decimal) -> Decimal:  
 return abs(x)  
  
  
def sin\_func(x: Decimal,y: Decimal) -> Decimal:  
 return Decimal(math.sin(float(x)))  
  
  
def cos\_func(x: Decimal,y: Decimal) -> Decimal:  
 return Decimal(math.cos(float(x)))  
  
  
def exp\_func(x: Decimal,y: Decimal) -> Decimal:  
 return Decimal(math.exp(float(x)))  
  
  
def power(x: Decimal, y: Decimal) -> Decimal:  
 if x == Decimal(0):  
 return Decimal(0)  
 y=y.quantize(Decimal('1'))  
 return Decimal(x \*\* y)  
  
# Типы узлов  
FUNCTIONS = [add, sub, mul, div, abs\_func, sin\_func, cos\_func]  
TERMINALS = ['x1', 'x2', 'x3', 'x4', 'x5',Decimal(1), Decimal(2)] # 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 plot\_graph(self):  
 *"""  
 Визуализирует дерево с помощью библиотеки NetworkX и Matplotlib  
 """* graph = nx.DiGraph()  
 positions = {}  
  
 def add\_edges(node, parent\_id=None, depth=0, pos=0):  
 if node is None:  
 return  
 node\_id = id(node)  
 positions[node\_id] = (pos, -depth) # Расположение узлов на графике  
 label = str(node.value)  
  
 if callable(node.value): # Если это функция, преобразуем её в читаемую форму  
 label = node.value.\_\_name\_\_  
 graph.add\_node(node\_id, label=label)  
  
 if parent\_id is not None:  
 graph.add\_edge(parent\_id, node\_id)  
  
 # Рекурсивно добавляем дочерние узлы  
 left\_pos = pos - 1 / (2 \*\* (depth + 1)) # Смещение для левого поддерева  
 right\_pos = pos + 1 / (2 \*\* (depth + 1)) # Смещение для правого поддерева  
 add\_edges(node.left, node\_id, depth + 1, left\_pos)  
 add\_edges(node.right, node\_id, depth + 1, right\_pos)  
  
 # Построение графа начиная с корня дерева  
 add\_edges(self.root)  
  
 # Визуализация графа  
 plt.figure(figsize=(12, 8))  
 labels = nx.get\_node\_attributes(graph, 'label')  
 nx.draw(graph, pos=positions, labels=labels, with\_labels=True, node\_size=2000, node\_color="skyblue")  
 plt.title("Tree Visualization")  
 plt.show()  
  
  
  
  
  
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  
 x1,x2,x3,x4,x5 = random.sample(range(-600,600),5)  
 for \_ in range(samples):  
  
 predicted = individual.tree.evaluate({'x1': Decimal(x1), 'x2': Decimal(x2), 'x3': Decimal(x3), 'x4': Decimal(x4), 'x5': Decimal(x5)})  
  
 target = target\_function(Decimal(x1), Decimal(x2), Decimal(x3), Decimal(x4), Decimal(x5)) # Целевая функция (x1, x2, x3, x4, x5)  
  
 fitness += ((predicted - target) \*\* 2)  
  
 individual.fitness = fitness  
  
  
 return population  
  
def target\_function(x1, x2, x3, x4, x5):  
 return x1\*x1+x2+2\*x3  
  
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, Decimal(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}")  
 if(generation%20==0):  
 best\_individual.tree.plot\_graph()  
  
 # 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, 4)  
 best\_individual = genetic\_algorithm(population, 100, 4, 5, 3, 1, 1)  
 print("Best Individual: ", best\_individual.tree,"\n Best Fitness: ", best\_individual.fitness)  
  
  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

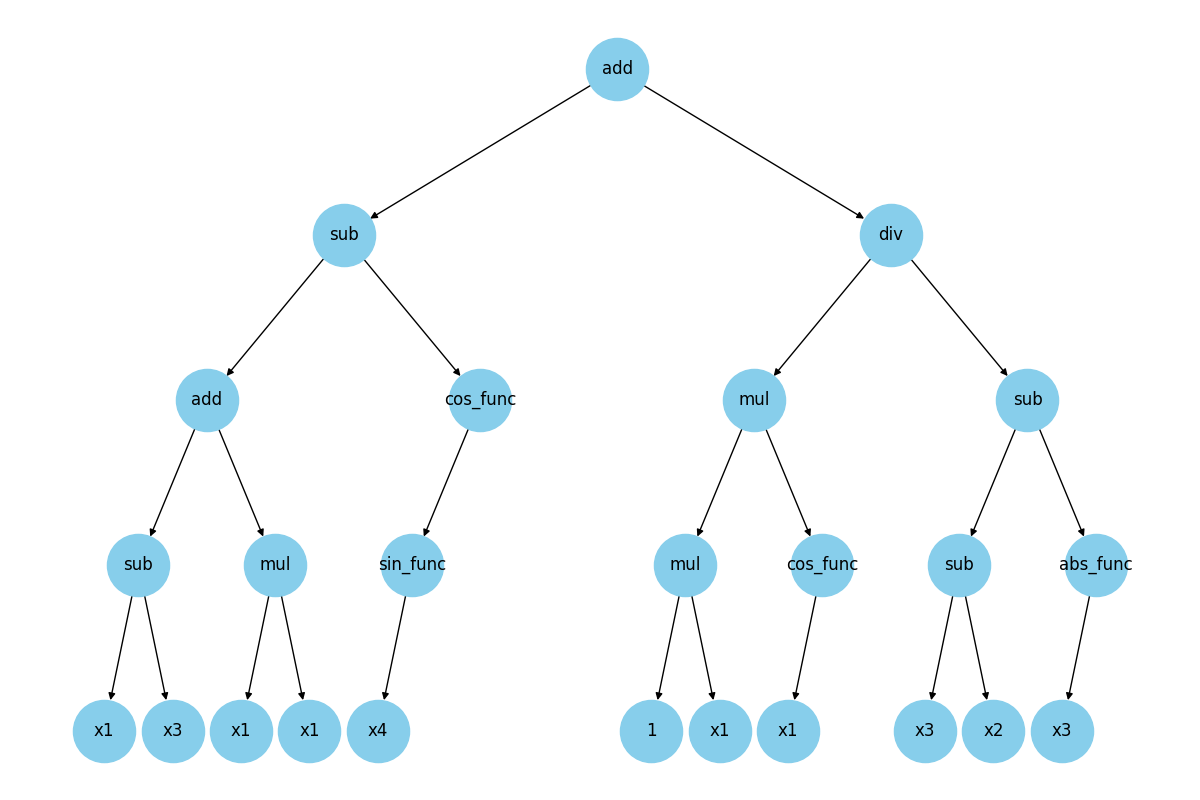
**4. Выполнение программы**

**Шаг 1**

****Лучшее решение в начальной популяции

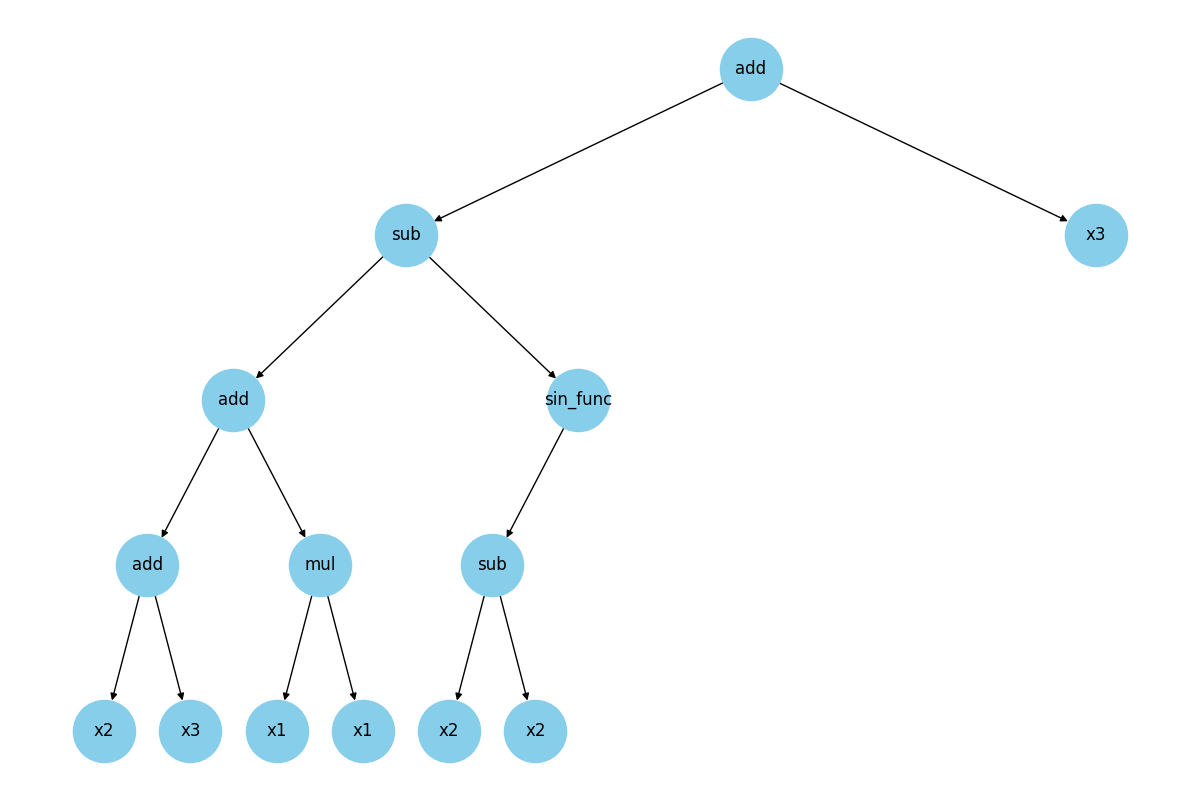
Fitness: 6400

**Шаг 20**

****Лучшая особь спустя 20 шагов

Fitness: 40.52595600

**Шаг 40**

****Популяция спустя 40 шагов

Fitness: 0

В дальнейшем (до 100 поколения) лучшее решение не менялось, так как имеет наименьший фитнес.

Сравним с целевой функцией:

Решение соответствует заданному уравнению, хоть и содержит интроны

**5. Контрольный вопрос**

**Как производится инициализация линейных структур?**

* Программа разбивается на 4 части (Заголовок, тело, подвал, возврат). Все части кроме тела заготавливаются заранее.
* Алгоритм инициализации:
  1. Выбор случайного числа в качестве длина
  2. Копирование заголовка
  3. Заполнение тела до указанной длины случайными операторами
  4. Копирование подвала и выхода