МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт компьютерных наук и кибербезопасности

Высшая школа технологий искусственного интеллекта

Отчёт по дисциплине «Генетические алгоритмы»

Практическая работа №4

«Генетическое программирование»
Вариант №17

Студент: ______________ Салимли Айзек Мухтар Оглы

Преподаватель: ______________ Большаков Александр Афанасьевич

«_____»____________ 20___ г.

Санкт-Петербург, 2025

Содержание

Введение

Генетические алгоритмы относятся к классу эволюционных методов оптимизации и активно применяются для решения сложных задач, где традиционные подходы оказываются малоэффективными. Эти алгоритмы вдохновлены природными процессами естественного отбора и наследственности, что позволяет им эффективно исследовать пространство решений и находить близкие к оптимальным результаты. В данной лабораторной работе реализован эволюционный алгоритм на основе генетического программирования для аппроксимации функции.

1 Постановка задачи

В лабораторной работе требовалось:

1. Разработать эволюционный алгоритм, реализующий генетическое программирование (ГП) для нахождения (символьной регрессии) заданной по варианту функции.
   - Структура для представления программы — древовидное представление.
   - Терминальное множество: переменные $x_1, \ldots, x_n$ и эфемерные случайные константы.
   - Функциональное множество: $+$, $-$, $*$, $/$, $\sin()$, $\cos()$, $\exp()$, возведение в степень (ˆ или).
   - Фитнесс-функция — мера близости между реальными значениями выхода и требуемыми, основанная на сумме квадратов ошибок (SSE).
2. Представить графически найденное решение на каждой 10-й итерации и в конце работы алгоритма.
3. Сравнить найденное решение с представленным в условии задачи.

Исходные данные

- Функция:

$$f_{1a}(x) \;=\; \sum_{i=1}^{n} i \cdot x_i^2$$

- Количество переменных: $N = 9$.
- Промежуток исследования: $x_i \in [\,-5.12,\; 5.12\,]$.

2 Теоретические сведения

2.1 Терминальное множество

Терминальное множество включает в себя "листья"дерева программы — элементы, которые не принимают аргументов. К ним относятся:

1. Внешние входы в программу: переменные (в данной работе x_1, x_2, x_3, x_4).
2. Используемые в программе константы: числовые значения, которые могут быть как предопределёнными, так и случайно генерируемыми (эфемерные случайные константы).
3. Функции, которые не имеют аргументов.

2.2 Функциональное множество

Функциональное множество состоит из операторов и функций, которые являются внутренними узлами дерева и обрабатывают значения от дочерних узлов. В данной работе оно включает:

- **Арифметические функции:** сложение $(+)$, вычитание $(-)$, умножение $(\cdot)$, деление $(/)$.
- **Трансцендентные функции:** синус $\sin$, косинус $\cos$, экспонента $\exp$.
- **Другие функции:** возведение в степень (ˆ или).

2.3 Древовидные структуры

Древовидная форма представления является классической для ГП. Программа представляется в виде дерева, где внутренние узлы — это функции из функционального множества, а листья (терминальные узлы) — это переменные и константы из терминального множества. Такая структура позволяет гибко работать с выражениями различной длины и сложности.

2.4 Инициализация древовидных структур

Начальная популяция деревьев создаётся с помощью двух основных методов:

- **Полный (full):** генерируются деревья, у которых все листья находятся на максимально заданной глубине.
- **Растущий (grow):** генерируются деревья неправильной формы, где терминалы могут находиться на разной глубине, не превышающей максимальную.

На практике часто используется их комбинация (*ramped half-and-half*) для обеспечения разнообразия структур в начальной популяции.

2.5 Кроссинговер на древовидных и графоподобных структурах

В генетическом программировании для графоподобных структур применяются два основных вида оператора кроссинговера. Эти методы комбинируют генетический материал из двух родительских программ путём обмена их частями.

2.6 Кроссинговер подграфов

Этот способ аналогичен кроссинговеру поддеревьев, который используется для древовидных структур. Процесс обмена происходит следующим образом:

1. В каждой родительской программе (графе) случайным образом выбирается множество смежных узлов, образующих подграф.

2. Производится обмен этими двумя подграфами между родительскими особями, в результате чего создаются две новые дочерние программы.

Этот метод позволяет обмениваться целыми функциональными блоками программы.

2.7 Линейный кроссинговер

Второй тип кроссинговера оперирует на уровне линейных сегментов кода внутри узлов графа:

1. В каждом из родительских графов выбирается один узел.
2. Внутри линейной программы этого узла выбирается сегмент кода, который определяется случайной начальной позицией и случайной длиной.
3. Происходит обмен этими линейными сегментами между родителями.
4. Если размер хотя бы одного из потомков превышает установленный порог, результаты кроссинговера могут быть аннулированы, и вместо этого выполняется обмен сегментами меньшей длины.

Обычно данный вид кроссинговера выполняется с определённой вероятностью, например, $P_l = 0.1$.

Как правило, в процессе эволюции для графоподобных структур используются оба типа кроссинговера для обеспечения как обмена крупными блоками (кроссинговер подграфов), так и более тонкой настройки внутри этих блоков (линейный кроссинговер).

2.8 Кроссинговер поддеревьев (Subtree Crossover) для древовидных структур

Кроссинговер поддеревьев является основным генетическим оператором, используемым для рекомбинации в генетическом программировании (ГП), когда программы представлены в виде древовидных структур. Он создаёт новые дочерние программы (потомков) путём обмена случайно выбранными поддеревьями между двумя родительскими программами.

2.8.1 Алгоритм выполнения

Процесс кроссинговера поддеревьев выполняется в несколько шагов:

1. **Выбор двух родителей.** Из текущей популяции для скрещивания выбираются две особи (программы-деревья).
2. **Выбор точки скрещивания в каждом родителе.** В каждом из двух родительских деревьев случайным образом выбирается один узел. Этот узел становится корнем поддерева, которое будет участвовать в обмене. Точкой скрещивания может быть как функциональный узел (внутренний узел дерева), так и терминальный узел (лист).
3. **Обмен поддеревьями.** Поддерево, начинающееся в точке скрещивания первого родителя, меняется местами с поддеревом из точки скрещивания второго родителя.
4. **Создание потомков.** В результате этого обмена создаются две новые дочерние программы. Каждая из них содержит часть генетического материала от одного родителя и «привитое» поддерево от другого.

2.8.2 Пример

Рассмотрим две родительские программы, представленные в виде математических выражений:

$$\text{Родитель 1: } \min\bigl(x-6,\ x+y\cdot 2\bigr), \qquad \text{Родитель 2: } 3\cdot x+\frac{y}{2}.$$

Процесс кроссинговера:

- В Родителе 1 в качестве точки скрещивания выбирается узел $(+)$. Поддерево, соответствующее этому узлу, — это выражение $x+y\cdot 2$.
- В Родителе 2 в качестве точки скрещивания выбирается узел $(\cdot)$. Соответствующее поддерево — $3\cdot x$.

После обмена этими поддеревьями получаются два потомка:

$$\text{Потомок 1: } \min\bigl(x-6,\ 3\cdot x\bigr), \qquad \text{Потомок 2: } \bigl(x+y\cdot 2\bigr)+\frac{y}{2}.$$

2.9 Выполнение мутации на древовидных структурах

Оператор мутации вносит случайные изменения в особь. Основные виды для деревьев:

- **Узловая:** замена одного узла на другой того же типа.
- **Усекающая:** замена целого поддерева на один случайный терминал.
- **Растущая:** замена случайного узла на новое, случайно сгенерированное поддерево.

2.10 Фитнесс-функция в генетическом программировании

В ГП фитнесс-функция почти всегда определяет меру ошибки программы. Она вычисляется на наборе тестовых данных и показывает, насколько сильно выходные значения программы отличаются от эталонных. Часто используется сумма квадратов ошибок (SSE). Цель эволюции — минимизировать эту ошибку, что эквивалентно максимизации фитнесс-функции, например:

$$F = \frac{1}{1+\mathrm{SSE}}.$$

3 Программная реализация

Программа решает задачу символьной регрессии для

$$f_{1a}(\mathbf{x}) = \sum_{i=1}^{n} i\, x_i^2, \quad x_i \in [-5.12,\, 5.12],$$

используя эволюционный алгоритм с древовидным представлением, терминалами $\{x_1, \ldots, x_9, \mathrm{ERC}\}$ и функциональным множеством $\{+, -, *, /, \sin, \cos, \exp, \hat{}\}$. Алгоритм минимизирует SSE (с дополнительной парсимонией по размеру дерева), выводит дерево лучшей особи через Graphviz, а также графики сходимости, средней длины программ и распределения ошибок.

3.1 Импорт и конфигурация

Основные импорты: `math`, `random`, `numpy`, `matplotlib.pyplot`, `dataclasses`, `typing`; опционально `graphviz` (`GRAPHVIZ_AVAILABLE`).
Константы (CONFIG): `N_VARS` $= 9$, `DOMAIN` $= (-5.12, 5.12)$, `POP_SIZE`, `GENERATIONS`, турнир $k =$ `TOURNAMENT_K`, вероятности операторов (`P_CROSS`, `P_MUT_NODE`, `P_MUT_SHRINK`, `P_MUT_GROW`), предельные глубины (`MAX_INIT_DEPTH`, `MAX_TREE_DEPTH`), диапазон ERC и штраф за сложность $\lambda =$ `LAMBDA_COMPLEXITY`.

3.2 Защищённые операции (числовая стабильность)

`p_div` (защищённое деление), `p_pow` (защищённая степень: клип показателя, модуль для нецелых), `p_exp` (клип аргумента), `p_sin`, `p_cos`.
Назначение: предотвратить NaN/∞ и переполнения при вычислении деревьев.

3.3 Примитивы и узлы дерева

`@dataclass Primitive`: имя, арность, функция. Словарь `FUNCTIONS` для $(+, -, *, /, \sin, \cos, \exp, \hat{})$.
`@dataclass Terminal`: терминал-переменная (`kind='var'`, `index`) или ERC (`kind='erc'`, `value`), метод `eval`.
`@dataclass Node`: `prim` $\in$ {`Primitive`, `Terminal`}, `children`; служебные методы: `is_function`/`is_terminal`, `copy`, `depth`, `size`, `eval` (векторизованная оценка на X), `to_string` (инфиксная запись).

3.4 Инициализация деревьев

`random_terminal` (переменная или ERC), `random_function` (по арности), `generate_full`, `generate_grow`, `generate_ramped`.
Метод *ramped half-and-half*: баланс разнообразия форм и глубин в начальной популяции.

3.5 Генетические операторы

Селекция. `tournament(pop, fitness, k)` — турнирная селекция; минимизируется SSE.
Кроссинговер. `all_nodes_with_parents` собирает узлы с родителями; `subtree_crossover` обменивает случайные поддеревья (с проверкой `MAX_TREE_DEPTH`).
Мутации. `mutate_node_replace` (замена функции/терминала на однотипный), `mutate_shrink` (усечение: функция $\to$ терминал), `mutate_grow` (вставка случайного поддерева ограниченной глубины).

3.6 Целевая функция и датасеты

`target_function(X)`: векторизованно считает f_{1a} на матрице X. `make_dataset` генерирует обучающую и валидационную выборки в $[-5.12, 5.12]^9$.

3.7 Фитнесс-функция

`fitness_sse(ind, X, y)`: $\mathrm{SSE} = \sum (y - \hat{y})^2$ с парсимонией $+\lambda \cdot \mathrm{size}^2$; защита от NaN/$\infty$.

3.8 Визуализация дерева (Graphviz)

`plot_tree_graphviz(root, filename, title)`: строит `graphviz.Digraph`; функциональные узлы — эллипсы, терминалы/константы — прямоугольники; сохраняет `PNG`. Используется каждые 10 поколений и в конце.

3.9 Главный эволюционный цикл

`evolve()`:

1. Генерация обучающего/валидационного наборов.
2. Инициализация популяции `generate_ramped`.
3. Для каждого поколения: оценка SSE, элитизм (`ELITE`), селекция, кроссинговер, мутации; обновление лучшей особи. Ведётся журнал `best_hist` (лучший SSE) и `avg_size_hist` (средний размер дерева).
4. Каждые `PLOT_EVERY` поколений — `plot_tree_graphviz`.
5. Финальный отчёт и три графика: (i) сходимость SSE, (ii) средняя длина программ, (iii) гистограмма остатков на валидации.

3.10 Выходные визуализации

- **Сходимость:** линия $\min \mathrm{SSE}$ по поколениям — контроль прогресса оптимизации.
- **Средняя длина программ:** $\frac{1}{|\mathcal{P}|} \sum \mathrm{size}(\text{индивид})$ — мониторинг bloat.
- **Распределение ошибок:** гистограмма $r = y - \hat{y}$ на валидации — проверка смещения/разброса.
- **Дерево решения:** `tree_gen_#`.png (срезы) и `tree_final.png` (финал).

3.11 Ключевые параметры для тонкой настройки

- **Сложность/парсимония:** `LAMBDA_COMPLEXITY`, `MAX_TREE_DEPTH`.
- **Поиск:** `POP_SIZE`, `GENERATIONS`, `TOURNAMENT_K`.
- **Баланс операторов:** `P_CROSS`, `P_MUT_NODE`, `P_MUT_SHRINK`, `P_MUT_GROW`.
- **Стохастика:** диапазон `ERC_RANGE`.

4 Результаты

Алгоритм был реализован на языке Python и запущен на 120 поколений для задачи символьной регрессии

$$f_{1a}(\mathbf{x}) = \sum_{i=1}^{9} i\, x_i^2, \qquad x_i \in [-5.12,\, 5.12].$$

Ниже приведены ключевые параметры запуска и сводные наблюдения по логам работы.

Параметры эксперимента

- Размер популяции: 300 особей.
- Количество поколений: 120.
- Вероятность кроссинговера: 0.9.
- Вероятности мутаций: узловая 0.2, усечение 0.1, рост 0.2.
- Метод отбора: турнирный, размер турнира $k = 5$.
- Элитизм: 2 лучшие особи переходят в следующее поколение.
- Ограничения структуры: начальная глубина до 6, максимальная глубина дерева 16.
- Парсимония: малый штраф за сложность $\lambda = \texttt{LAMBDA\_COMPLEXITY} = 10^{-4}$.

Анализ динамики фитнесса Оптимизация велась по сумме квадратов ошибок (SSE), при этом фитнесс интерпретировался как $F = 1/(1 + \mathrm{SSE})$. По журналам видно характерную для ГП картину:

- Уже на первых поколениях достигается заметное снижение SSE за счёт быстрого распространения удачных фрагментов деревьев.
- В интервале примерно до 50-го поколения наблюдается монотонное, но замедляющееся улучшение лучшего решения.
- Далее улучшения становятся редкими и небольшими; к $\sim$ 90–100 поколению SSE стабилизируется на плато, что соответствует практически неизменному значению фитнесса.
- Средний фитнесс популяции растёт быстрее на ранних этапах и выходит на высокий стабильный уровень к завершающей трети эволюции, что указывает на успешное распространение «генетического материала» лучших особей механизмами отбора и кроссинговера.

Такая динамика подтверждает работоспособность поиска: алгоритм выходит из локальных минимумов и постепенно дорабатывает решение, пока не достигает зоны стагнации.

Анализ сложности программ (Bloat) Запуск демонстрирует типичное явление «вздутия» кода:

- **Нейтральные замены лучшей особи.** Даже при почти неизменной SSE структура лучшей программы заметно меняется от поколения к поколению: размер дерева то растёт, то падает, что свидетельствует о замене одной близкой по качеству программы на другую, зачастую более крупную. Это эффект "*survival of the fattest*" — крупные деревья более устойчивы к деструктивным мутациям/кроссинговеру.
- **Рост среднего размера.** График средней длины программ показывает восходящий тренд в течение запуска: популяция аккумулирует "интроны" (участки, мало влияющие на выход), что увеличивает сложность без эквивалентного выигрыша в качестве после определённого момента времени.

Общий вывод по результатам Запуск можно считать успешным: алгоритм быстро нашёл решения с низкой ошибкой и стабилизировал качество к финалу эволюции. Одновременно отчётливо проявился **bloat**, что подчёркивает важность приёмов контроля сложности (усиление штрафа за размер, более жёсткие ограничения глубины, упрощение выражений) для получения не только точных, но и компактных формул.

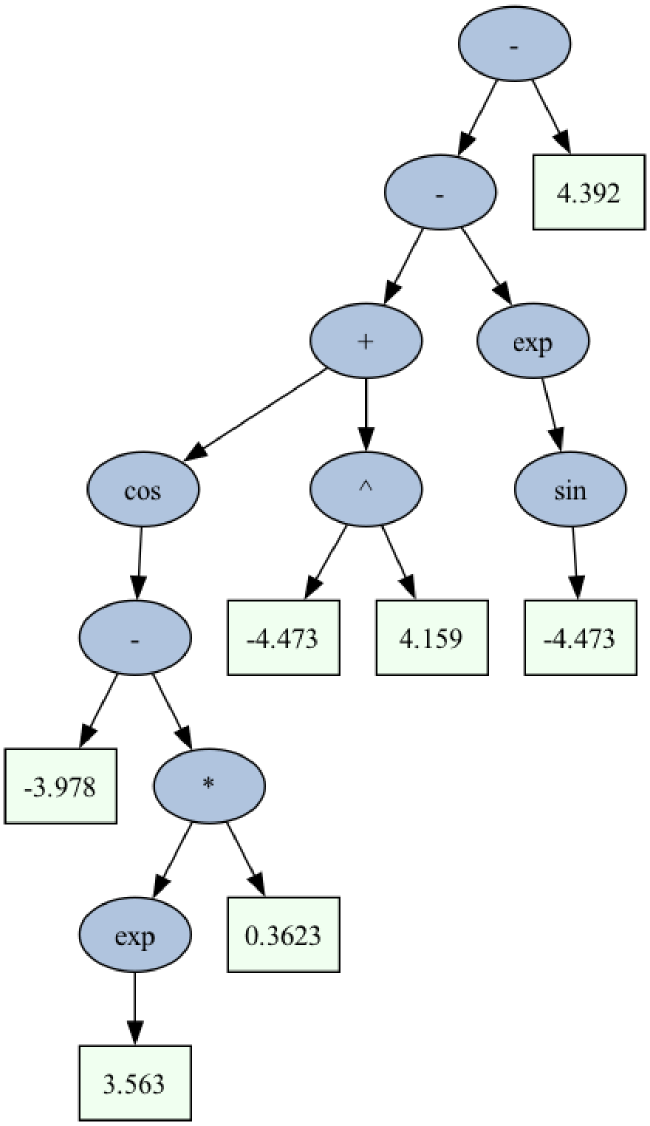

Рис. 1: Дерево 10-го поколения

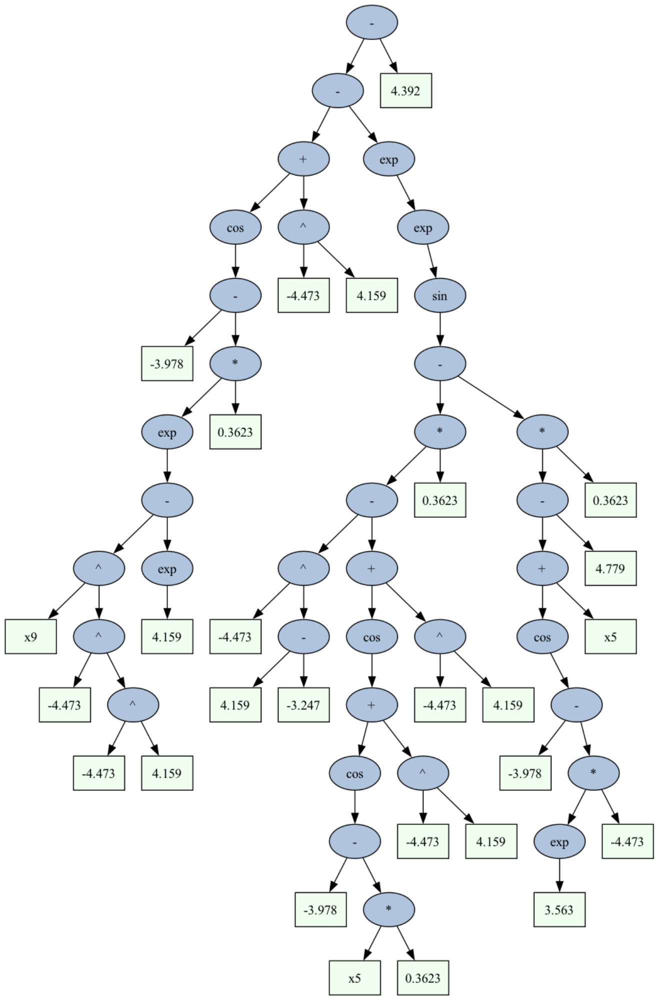

Рис. 2: Дерево 20-го поколения

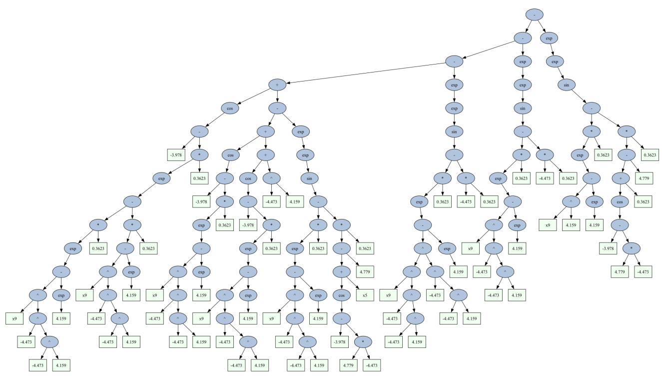

Рис. 3: Дерево 50-го поколения

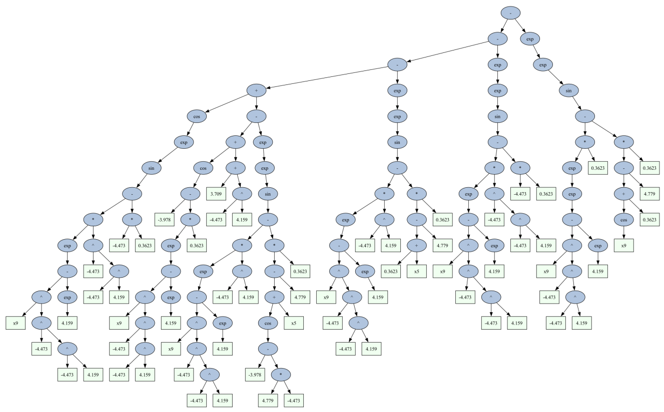

Рис. 4: Дерево 70-го поколения

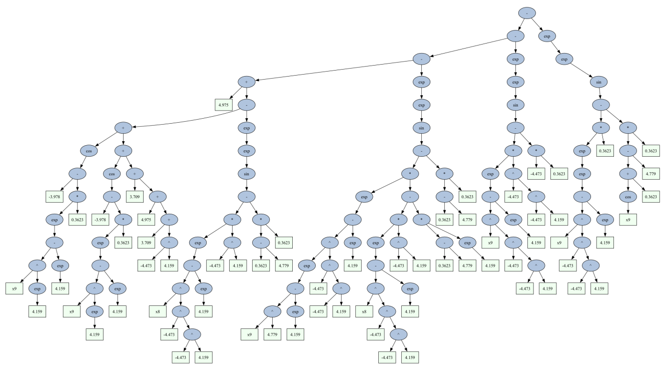

Рис. 5: Дерево 100-го поколения

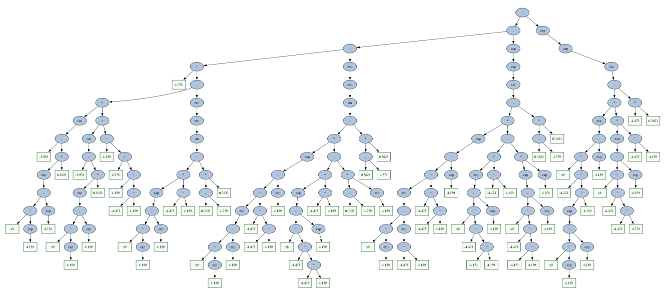

Рис. 6: Финальное дерево

Рис. 7: Сходимость алгоритма (лучший SSE по поколениям)

Рис. 8: Средний размер дерева (число узлов) по поколениям

Рис. 9: Распределение ошибок на валидации

5 Анализ исследования

Качество аппроксимации и сходимость

- Алгоритм нашёл решения с низкой ошибкой на ранних поколениях, после чего демонстрировал постепенное улучшение.
- Снимки дерева лучшей особи на промежуточных этапах и финальная структура подтверждают, что улучшение шло за счёт замены и перестройки поддеревьев (структурная эволюция формулы).

Сложность программ (bloat)

- По графику среднего размера деревьев виден устойчивый рост сложности популяции: эффект *bloat*, при котором программы увеличиваются по длине без эквивалентного выигрыша в точности на поздних стадиях.

Поведение ошибок

- Гистограмма остатков имеет заметное смещение вправо: преобладают положительные значения. При этом форма распределения остаётся близкой к симметричной относительно некоторого положительного среднего.

Итоговый вывод

- Запуск можно считать успешным: достигнута низкая ошибка и устойчивость качества к концу эволюции.
- Основной практический недостаток — рост сложности выражений на поздних поколениях.

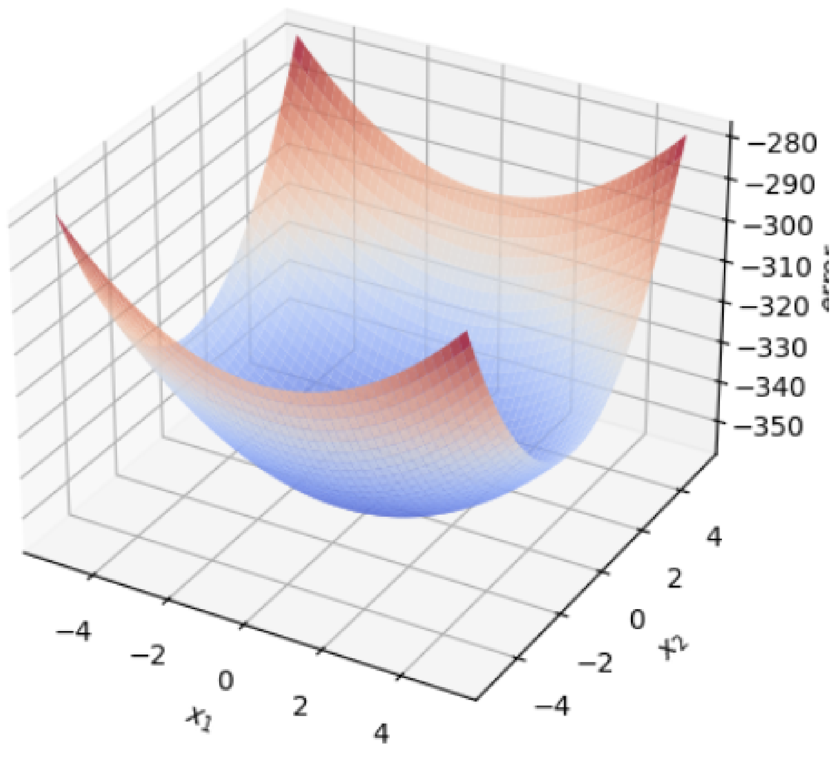

Рис. 10: График функции получившимся из дерева

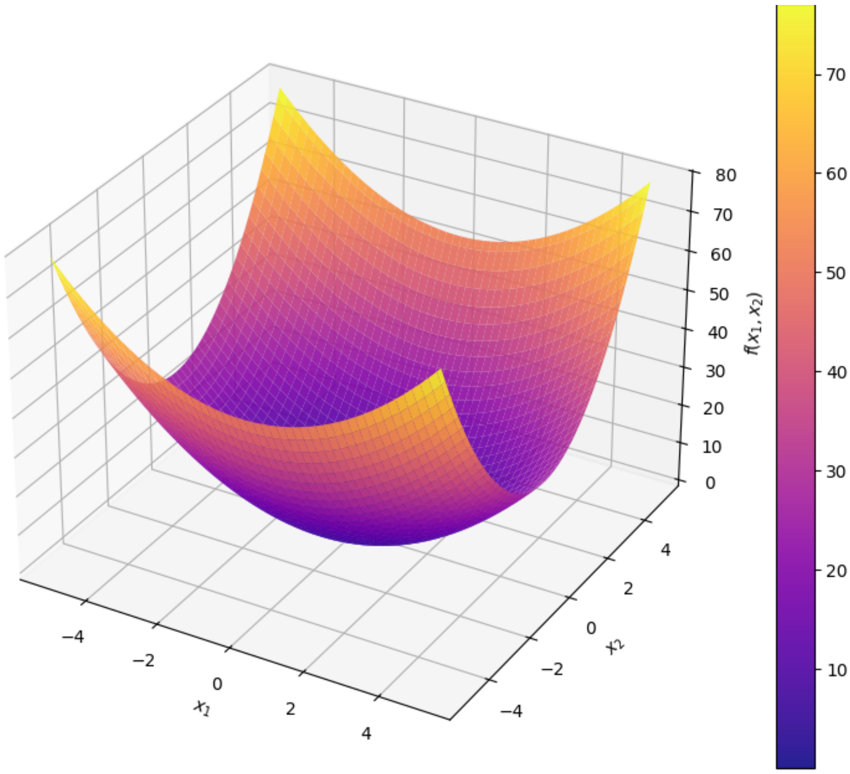

Рис. 11: Эталонный график функции

6 Ответ на контрольный вопрос

Вопрос: Какие структуры используются для представления программ в ГП? **Ответ:**

- **Древовидное представление:** Программа представляется в виде дерева, где листья (терминальные узлы) соответствуют переменным, константам или функциям без аргументов, а внутренние узлы — функциям (операторам).
- **Линейная структура:** Программа представляется в виде линейной последовательности операторов. Функциональное множество включает арифметические операции, условные операторы и вызовы функций, а терминальное — переменные и константы.
- **Графоподобная структура:** Программа представляется в виде графа. Это более общая структура, которая позволяет естественным образом моделировать операторы ветвления и циклы.

Заключение

В результате выполнения лабораторной работы №4 были достигнуты следующие результаты:

- освоен теоретический материал;
- создана программа на языке `Python` с использованием среды `Jupyter Notebook`;
- реализован эволюционный алгоритм ГП для функции $f_{1a}(\mathbf{x}) = \sum_{i=1}^{n} i\, x_i^2$ и получены приближения на диапазоне $[-5.12, 5.12]^9$;
- выполнена визуализация: срезы деревьев (10, 20, 50, 70, 100 поколения), финальное дерево, графики сходимости, среднего размера программ и распределения ошибок;

Список литературы

1. Методические указания по выполнению лабораторных работ к курсу «Генетические алгоритмы», стр. 119.

```
  import math
import random
import numpy as np
import matplotlib.pyplot as plt
from dataclasses import dataclass, field
from typing import Callable, List, Tuple, Union, Optional, Dict

try:
    import graphviz
    GRAPHVIZ_AVAILABLE = True
except Exception:
    GRAPHVIZ_AVAILABLE = False

SEED = 42
N_VARS = 9
DOMAIN = (-5.12, 5.12)
N_SAMPLES = 2000
POP_SIZE = 300
GENERATIONS = 120
TOURNAMENT_K = 5
ELITE = 2
P_CROSS = 0.9
P_MUT_NODE = 0.2
P_MUT_SHRINK = 0.1
P_MUT_GROW = 0.2
MAX_INIT_DEPTH = 6
MAX_TREE_DEPTH = 16
ERC_RANGE = (-5.0, 5.0)
PLOT_EVERY = 10
LAMBDA_COMPLEXITY = 1e-4
random.seed(SEED)
np.random.seed(SEED)

def p_div(x, y):
    denom = np.where(np.abs(y) < 1e-12, 1e-12, y)
    return x / denom

def p_pow(x, y):
    y_clip = np.clip(y, -4.0, 4.0)
    y_round = np.round(y_clip)
    is_int = np.isclose(y_clip, y_round)
    base = np.where(is_int, x, np.abs(x))
    try:
        out = np.power(base, np.where(is_int, y_round, y_clip))
    except FloatingPointError:
        out = np.power(np.clip(base, -1e6, 1e6), np.where(is_int, y_round, y_clip))
    return np.clip(out, -1e6, 1e6)

def p_exp(x):
    return np.exp(np.clip(x, -50, 50))

def p_sin(x):
    return np.sin(x)

def p_cos(x):
    return np.cos(x)

```

```python
@dataclass(frozen=True)
class Primitive:
    name: str
    arity: int
    func: Callable

FUNCTIONS: List[Primitive] = [
    Primitive('+', 2, lambda a, b: a + b),
    Primitive('-', 2, lambda a, b: a - b),
    Primitive('*', 2, lambda a, b: a * b),
    Primitive('/', 2, p_div),
    Primitive('sin', 1, p_sin),
    Primitive('cos', 1, p_cos),
    Primitive('exp', 1, p_exp),
    Primitive('^', 2, p_pow),
]

@dataclass
class Terminal:
    kind: str
    index: Optional[int] = None
    value: Optional[float] = None

    def eval(self, X: np.ndarray) -> np.ndarray:
        if self.kind == 'var':
            return X[:, self.index]
        else:
            return np.full(X.shape[0], self.value, dtype=float)

    def __str__(self):
        if self.kind == 'var':
            return f"x{self.index+1}"
        else:
            return f"{self.value:.4g}"

@dataclass
class Node:
    prim: Union[Primitive, Terminal]
    children: List['Node'] = field(default_factory=list)

    def is_function(self) -> bool:
        return isinstance(self.prim, Primitive)

    def is_terminal(self) -> bool:
        return isinstance(self.prim, Terminal)

    def copy(self) -> 'Node':
        return Node(self.prim, [c.copy() for c in self.children])

    def depth(self) -> int:
        if self.is_terminal():
            return 1
        return 1 + max((ch.depth() for ch in self.children), default=0)

    def size(self) -> int:
        return 1 + sum(ch.size() for ch in self.children)

    def eval(self, X: np.ndarray) -> np.ndarray:
        if self.is_terminal():
            return self.prim.eval(X)
        args = [c.eval(X) for c in self.children]
```

```
        try:
            if self.prim.arity == 1:
                return self.prim.func(args[0])
            elif self.prim.arity == 2:
                return self.prim.func(args[0], args[1])
        except FloatingPointError:
            pass
        return np.full(X.shape[0], np.nan)

    def to_string(self) -> str:
        if self.is_terminal():
            return str(self.prim)
        name = self.prim.name
        if self.prim.arity == 1:
            return f"{name}({self.children[0].to_string()})"
        left = self.children[0].to_string()
        right = self.children[1].to_string()
        if name in ['+', '-', '*', '/', '^']:
            return f"({left} {name} {right})"
        return f"{name}({left}, {right})"

def random_terminal() -> Node:
    if random.random() < 0.5:
        idx = random.randrange(N_VARS)
        return Node(Terminal('var', index=idx))
    else:
        val = random.uniform(*ERC_RANGE)
        return Node(Terminal('erc', value=val))

def random_function(arity: Optional[int] = None) -> Primitive:
    if arity is None:
        return random.choice(FUNCTIONS)
    cands = [f for f in FUNCTIONS if f.arity == arity]
    return random.choice(cands)

def generate_full(depth: int) -> Node:
    if depth <= 1:
        return random_terminal()
    prim = random_function(arity=random.choice([1, 2]))
    return Node(prim, [generate_full(depth - 1) for _ in range(prim.arity)])

def generate_grow(depth: int) -> Node:
    if depth <= 1 or (depth > 1 and random.random() < 0.3):
        return random_terminal()
    prim = random_function(arity=random.choice([1, 2]))
    return Node(prim, [generate_grow(depth - 1) for _ in range(prim.arity)])

def generate_ramped(max_depth: int) -> Node:
    d = random.randint(2, max_depth)
    return generate_full(d) if random.random() < 0.5 else generate_grow(d)

def tournament(pop: List[Node], fitness: List[float], k: int) -> Node:
    idxs = random.sample(range(len(pop)), k)
    best = min(idxs, key=lambda i: fitness[i])
    return pop[best].copy()

def all_nodes_with_parents(root: Node) -> List[Tuple[Optional[Node], int,
  Node]]:
    out = []
    def dfs(parent, idx, node):
        out.append((parent, idx, node))
```

```python
        for i, ch in enumerate(node.children):
            dfs(node, i, ch)
    dfs(None, -1, root)
    return out

def replace_child(parent: Optional[Node], child_index: int, new_child: Node,
    root: Node) -> Node:
    if parent is None:
        return new_child
    parent.children[child_index] = new_child
    return root

def subtree_crossover(a: Node, b: Node, max_depth: int) -> Tuple[Node, Node
    ]:
    a = a.copy()
    b = b.copy()
    a_nodes = all_nodes_with_parents(a)
    b_nodes = all_nodes_with_parents(b)
    _, _, a_sub = random.choice(a_nodes)
    _, _, b_sub = random.choice(b_nodes)

    a2 = a.copy()
    b2 = b.copy()
    astr = a_sub.to_string()
    bstr = b_sub.to_string()
    a2_nodes = all_nodes_with_parents(a2)
    b2_nodes = all_nodes_with_parents(b2)
    cand_a = [(p, i, n) for (p, i, n) in a2_nodes if n.to_string() == astr]
    cand_b = [(p, i, n) for (p, i, n) in b2_nodes if n.to_string() == bstr]
    if not cand_a or not cand_b:
        return a, b
    pa2, ia2, a_sub2 = random.choice(cand_a)
    pb2, ib2, b_sub2 = random.choice(cand_b)

    new_a = replace_child(pa2, ia2, b_sub2.copy(), a2)
    new_b = replace_child(pb2, ib2, a_sub2.copy(), b2)
    if new_a.depth() > max_depth or new_b.depth() > max_depth:
        return a, b
    return new_a, new_b

def mutate_node_replace(tree: Node) -> Node:
    t = tree.copy()
    parent, idx, node = random.choice(all_nodes_with_parents(t))
    if node.is_function():
        ar = node.prim.arity
        node.prim = random_function(arity=ar)
    else:
        node.prim = Terminal('var', index=random.randrange(N_VARS)) if
            random.random() < 0.5 \
                      else Terminal('erc', value=random.uniform(*ERC_RANGE))
    return t

def mutate_shrink(tree: Node) -> Node:
    t = tree.copy()
    funcs = [triple for triple in all_nodes_with_parents(t) if triple[2].
        is_function()]
    if not funcs:
        return t
    parent, idx, node = random.choice(funcs)
    return replace_child(parent, idx, random_terminal(), t)
```

```
def random_subtree(max_depth: int) -> Node:
    return generate_ramped(max_depth)

def mutate_grow(tree: Node, max_depth: int) -> Node:
    t = tree.copy()
    parent, idx, node = random.choice(all_nodes_with_parents(t))
    new_st = random_subtree(max_depth=min(5, max_depth))
    candidate = replace_child(parent, idx, new_st, t)
    if candidate.depth() > max_depth:
        return t
    return candidate

def target_function(X: np.ndarray) -> np.ndarray:
    idxs = np.arange(1, N_VARS + 1, dtype=float)
    return (X**2 @ idxs)

def make_dataset(n_samples: int, low: float, high: float) -> Tuple[np.
    ndarray, np.ndarray]:
    X = np.random.uniform(low, high, size=(n_samples, N_VARS))
    y = target_function(X)
    return X, y

def fitness_sse(ind: Node, X: np.ndarray, y: np.ndarray) -> float:
    pred = ind.eval(X)
    if np.any(np.isnan(pred)) or np.any(np.isinf(pred)):
        return 1e50
    err = y - pred
    sse = float(np.sum(err * err))
    if LAMBDA_COMPLEXITY > 0:
        sse += LAMBDA_COMPLEXITY * (ind.size() ** 2)
    return sse

def plot_tree_graphviz(root: Node, filename: str, title: Optional[str] =
    None):
    if not GRAPHVIZ_AVAILABLE:
        return
    dot = graphviz.Digraph(comment=title or 'tree', format='png')
    def add(node: Node) -> str:
        node_id = str(id(node))
        if node.is_terminal():
            label = str(node.prim)
            shape = 'box'
            fill = 'honeydew'
        else:
            label = node.prim.name
            shape = 'ellipse'
            fill = 'lightsteelblue'
        dot.node(node_id, label, shape=shape, style='filled', fillcolor=fill
            )
        for ch in node.children:
            ch_id = add(ch)
            dot.edge(node_id, ch_id)
        return node_id
    add(root)
    out = dot.render(filename, view=False)
    print(f"[Graphviz] {out}")

def evolve():
    X, y = make_dataset(N_SAMPLES, *DOMAIN)
    X_val, y_val = make_dataset(1000, *DOMAIN)
    pop = [generate_ramped(MAX_INIT_DEPTH) for _ in range(POP_SIZE)]
```

```python
    fitness = [fitness_sse(ind, X, y) for ind in pop]
    sizes = [ind.size() for ind in pop]

    best_idx = int(np.argmin(fitness))
    best = pop[best_idx].copy()
    best_fit = fitness[best_idx]

    best_hist = [best_fit]
    avg_size_hist = [float(np.mean(sizes))]

    print(f"Init best SSE: {best_fit:.6g} | depth={best.depth()} | size={best.size()}")
    print("Best expr:", best.to_string())

    for gen in range(1, GENERATIONS + 1):
        new_pop: List[Node] = []

        elite_idx = np.argsort(fitness)[:ELITE]
        for ei in elite_idx:
            new_pop.append(pop[ei].copy())

        while len(new_pop) < POP_SIZE:
            if random.random() < P_CROSS and len(new_pop) + 1 < POP_SIZE:
                p1 = tournament(pop, fitness, TOURNAMENT_K)
                p2 = tournament(pop, fitness, TOURNAMENT_K)
                c1, c2 = subtree_crossover(p1, p2, MAX_TREE_DEPTH)
                new_pop.extend([c1, c2])
            else:
                p = tournament(pop, fitness, TOURNAMENT_K)
                c = p
                if random.random() < P_MUT_NODE:
                    c = mutate_node_replace(c)
                if random.random() < P_MUT_SHRINK:
                    c = mutate_shrink(c)
                if random.random() < P_MUT_GROW:
                    c = mutate_grow(c, MAX_TREE_DEPTH)
                new_pop.append(c)

        pop = new_pop[:POP_SIZE]
        fitness = [fitness_sse(ind, X, y) for ind in pop]
        sizes = [ind.size() for ind in pop]

        gen_best_idx = int(np.argmin(fitness))
        gen_best = pop[gen_best_idx]
        gen_best_fit = fitness[gen_best_idx]
        if gen_best_fit < best_fit:
            best_fit = gen_best_fit
            best = gen_best.copy()

        best_hist.append(best_fit)
        avg_size_hist.append(float(np.mean(sizes)))

        if gen % 1 == 0:
            print(f"Gen {gen:4d} | best SSE={best_fit:.6g} | depth={best.depth():2d} | size={best.size():4d} | avg_size={avg_size_hist[-1]:.1f}")
        if gen % PLOT_EVERY == 0:
            title = f"Gen {gen} | SSE(val)={fitness_sse(best, X_val, y_val):.4g}"
            plot_tree_graphviz(best, filename=f"tree_gen_{gen}", title=title)
```

```python

    final_sse_train = fitness_sse(best, X, y)
    final_sse_val = fitness_sse(best, X_val, y_val)

    plot_tree_graphviz(best, filename='tree_final', title=f"FINAL | SSE(val)
        ={final_sse_val:.4g}")

    plt.figure(figsize=(7,4.5))
    plt.plot(best_hist, linewidth=2)
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()

    plt.figure(figsize=(7,4.5))
    plt.plot(avg_size_hist, linewidth=2)
    plt.xlabel("Evo")
    plt.ylabel("Mean")
    plt.title("Tree cap")
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()

    yhat_val = best.eval(X_val)
    residuals = y_val - yhat_val
    plt.figure(figsize=(7,4.5))
    plt.hist(residuals, bins=50, alpha=0.8)
    plt.xlabel("Err")
    plt.ylabel("Val")
    plt.title("Valid")
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.show()

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