МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение
высшего образования «Санкт-Петербургский политехнический
университет Петра Великого»

Институт компьютерных наук и кибербезопасности

Высшая школа технологий искусственного интеллекта

Направление: 02.03.01 «Математика и компьютерные науки»

Лабораторная работа №4
по дисциплине
«Генетические алгоритмы»
Вариант 18

Студент,
группы 5130201/20101 _____ Тищенко А. А.

Преподаватель _____ Большаков А. А.

«_____»_____ 2025г.

Санкт-Петербург, 2025

Содержание

1 Постановка задачи

В данной работе были поставлены следующие задачи:

- Разработать эволюционный алгоритм, реализующий ГП для нахождения заданной по варианту функции.
  - Структура для представления программы – древовидное представление.
  - Терминальное множество: переменные $x_1, x_2, x_3, \ldots, x_n$, 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.
- Представить графически найденное решение на каждой итерации.
- Сравнить найденное решение с представленным в условии задачи.

Индивидуальное задание вариант 18:

Дано: Функция

$$f(x) = \sum_{i=1}^{n} \sum_{j=1}^{i} x_j^2, \text{ где } x_j \in [-5.536, 5.536] \text{ для всех } j = 1, \ldots, n, \text{ а } n = 8.$$

2 Теоретические сведения

2.1 Генетическое программирование

Генетическое программирование (ГП) — разновидность эволюционных алгоритмов, в которых особь представляет собой программу, автоматически создаваемую для решения задачи. В отличие от генетических алгоритмов с фиксированной структурой хромосом, в ГП особи имеют переменную длину, что требует специальных методов кодирования, инициализации и генетических операторов. Ключевая идея ГП — представление программы на высоком уровне абстракции с учётом структуры компьютерных программ.

Оценка программ выполняется с помощью фитнесс-функции, отражающей степень соответствия решения требованиям задачи. Обычно используются метрики ошибки: среднеквадратичная ошибка, абсолютная ошибка или другие функции рассогласования между вычисленным и ожидаемым значением. Чем ниже ошибка, тем выше приспособленность особи.

2.2 Терминальное и функциональное множества

Программы формируются из **переменных**, **констант** и **функций**, связанных синтаксическими правилами. Для их описания необходимо определить два базовых множества:

- **Терминальное множество**, включающее константы и переменные.
- **Функциональное множество**, состоящее из операторов и элементарных функций, таких как $\exp(x)$, $\sin(x)$ и других.

2.2.1 Терминальное множество

Терминальное множество включает:

1. Внешние входы программы.
2. Константы, используемые в программе.
3. Функции без аргументов.

Термин «терминал» используется потому, что эти элементы соответствуют концевым (висячим) узлам в древовидных структурах и терминалам формальных грамматик. Терминал предоставляет численное значение, не требуя входных аргументов, то есть имеет нулевую арность. В классическом ГП на основе деревьев множество числовых констант выбирается для всей популяции и остается неизменным.

2.2.2 Функциональное множество

Функциональное множество состоит из операторов и различных функций. Оно может быть очень широким и включать типичные конструкции языков программирования, такие как:

- Логические функции: AND, OR, NOT;
- Арифметические операции: $+$, $-$, $\times$, $\div$;
- Трансцендентные функции: sin, cos, tan, log;
- Операции присваивания: $a := 2$;
- Условные операторы: if-then-else, switch/case;
- Операторы переходов: go to, jump, call;
- Операторы циклов: while, repeat-until, for;
- Подпрограммы и пользовательские функции.

2.3 Виды представления программ. Древовидное представление

Среди наиболее распространённых структур для представления особей (потенциальных решений) в современном генетическом программировании можно выделить:

1. **Древовидное представление** — классический подход, где программы представляются в виде деревьев с операторами в узлах и терминалами в листьях
2. **Линейная структура** — программы записываются как последовательности инструкций, аналогично ассемблерному коду
3. **Графоподобная структура** — расширенное представление, допускающее множественные связи и переиспользование компонентов

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

Сложность древовидных структур оценивается через максимальную глубину дерева D_m или общее количество узлов. Процесс инициализации древовидных структур основан на случайном выборе функциональных и терминальных символов при

заданном ограничении максимальной глубины. Рассмотрим пример с терминальным множеством:

Существуют два основных метода инициализации:

Полный метод (full)

На всех уровнях, кроме последнего, выбираются только функциональные символы. Терминальные символы размещаются исключительно на уровне максимальной глубины D_m. Это гарантирует создание сбалансированных деревьев регулярной структуры.

Растущий метод (grow)

На каждом шаге случайным образом выбирается либо функциональный, либо терминальный символ. Выбор терминала прекращает рост ветви, что приводит к формированию нерегулярных деревьев с различной глубиной листьев.

2.5 Оператор кроссинговера на древовидных структурах

Для древовидной формы представления программ в генетическом программировании применяются три основных типа операторов кроссинговера:

a) Узловой ОК

b) Кроссинговер поддеревьев

c) Смешанный

2.5.1 Узловой оператор кроссинговера

В узловом операторе кроссинговера выбираются два родителя (два дерева) и внутри них — узлы. Первый родитель называется доминантом, второй — рецессивом. Узлы могут различаться по типу, поэтому сначала необходимо проверить, что выбранные узлы взаимозаменяемы. Если типы не совпадают, выбирается другой узел во втором родителе, и проверка повторяется. После этого осуществляется обмен выбранных узлов между деревьями.

2.5.2 Кроссинговер поддеревьев

В кроссинговере поддеревьев не происходит обмен отдельными узлами, а определяется обмен поддеревьями. Он осуществляется следующим образом:

1. Выбираются два родителя (*один — доминантный, другой — рецессивный*). Необходимо убедиться, что выбранные узлы взаимозаменяемы, то есть принадлежат одному типу. В противном случае выбирается другой узел в рецессивном дереве.

2. Производится обмен соответствующими поддеревьями.

3. Далее вычисляется предполагаемый размер потомков. Если он не превышает установленный порог, то обмен ветвями запоминается.

При смешанном операторе кроссинговера для некоторых узлов выполняется узловой ОК, а для других - кроссинговер поддеревьев. В целом ОК выполняется следующим образом:

1. Выбор точек скрещивания P_1, P_2 в обоих родителях

2. Выбор типа кроссинговера с заданной вероятностью:

   - Первый тип (обмен подграфами) с вероятностью P_G
   - Второй тип (линейный обмен) с вероятностью $1 - P_G$

3. Если выбран первый тип и размер потомка не превышает порог, выполняется кроссинговер подграфами

4. Если выбран второй тип и размер потомка не превышает порог, выполняется линейный кроссинговер

2.6 Мутационные операторы для древовидных структур

В контексте древовидного представления программ применяются следующие мутационные операторы:

a) Мутация узлов (узловая)

b) Мутация с усечением (усекающая)

c) Мутация с ростом (растущая)

d) Hoist-мутация

Процедура узловой мутации включает следующие шаги:

1. Случайный выбор целевого узла в дереве программы и идентификация его типа

2. Случайный выбор заменяющего узла того же типа из соответствующего множества (функционального или терминального)

3. Замена исходного узла на выбранный вариант

Алгоритм усекающей мутации реализуется следующим образом:

1. Выбор узла, который будет подвергнут мутации
2. Случайный выбор терминального символа из допустимого множества
3. Удаление поддерева, корнем которого является выбранный узел
4. Замена удаленного поддерева терминальным символом

Алгоритм растущей мутации реализуется следующим образом:

1. Определение узла, подвергаемого мутации
2. Если узел является терминальным, выбирается другой узел; для нетерминального узла производится удаление всех исходящих ветвей
3. Вычисление размера и сложности оставшейся части дерева
4. Генерация нового случайного поддерева, размер которого не превышает заданного порогового значения, и его размещение вместо удалённой части

Алгоритм Hoist-мутации предназначен для борьбы с избыточным ростом деревьев (bloat) и реализуется следующим образом:

1. Случайный выбор поддерева с функциональным узлом в корне
2. Выбор случайного узла внутри этого поддерева (исключая корень выбранного поддерева)
3. Замена исходного поддерева на поддерево, начинающееся с выбранного внутреннего узла
4. В результате дерево становится короче, сохраняя при этом часть исходной структуры

Данная мутация всегда уменьшает размер дерева, что помогает контролировать сложность программ и предотвращает неконтролируемый рост деревьев в процессе эволюции.

Комбинированная мутация. В реализованном алгоритме используется стратегия комбинированной мутации, которая на каждом шаге случайно выбирает один из четырёх описанных операторов с заданными вероятностями:

- Растущая мутация: $p = 0.40$
- Узловая мутация: $p = 0.30$
- Hoist-мутация: $p = 0.15$
- Усекающая мутация: $p = 0.15$

Такой подход обеспечивает баланс между увеличением разнообразия популяции (растущая мутация), локальными изменениями (узловая мутация) и контролем размера деревьев (Hoist-мутация и усекающая мутация).

2.7 Фитнес-функции в генетическом программировании

В отличие от генетических алгоритмов, где фитнес-функция часто совпадает с исходной целевой функцией, в генетическом программировании фитнес-функция обычно измеряет степень соответствия между фактическими выходными значениями y_i и целевыми значениями d_i. В качестве фитнес-функций часто используются метрики ошибок, такие как абсолютное отклонение или среднеквадратичная ошибка.

3 Особенности реализации

В рамках работы создана библиотека `gp` для генетического программирования с древовидным представлением программ. Реализация выполнена на языке Python с использованием NumPy для векторизованных вычислений.

3.1 Примитивы и операции (primitive.py, ops.py)

Базовый класс `Primitive` представляет атомарные элементы дерева программы:

```
@dataclass(frozen=True)
class Primitive:
    name: str
    arity: int  # арность: 0 для терминалов, >0 для операций
    operation_fn: OperationFn | None
```

Реализованы конструкторы для создания терминалов и операций: `Var(name: str)`, `Const(name: str, val: Value)`, `Operation(name: str, arity: int, fn)`.

Модуль `ops.py` содержит набор безопасных векторизованных операций. Функция `make_safe` оборачивает операции для обработки некорректных значений:

```
def make_safe(fn: Callable) -> Callable:
    def wrapped(args: Sequence[Value]) -> Value:
        with np.errstate(over="ignore", invalid="ignore",
                         divide="ignore", under="ignore"):
            res = fn(args)
        res = np.nan_to_num(res, nan=0.0, posinf=1e6, neginf=-1e6)
        return np.clip(res, -1e6, 1e6)
    return wrapped
```

Реализованы унарные операции (`NEG, SIN, COS, SQUARE, EXP`) и бинарные (`ADD, SUB, MUL, DIV, POW`). Для деления используется защита от деления на ноль, для возведения в степень – ограничение показателя.

3.2 Узлы дерева (node.py)

Класс `Node` представляет узел дерева программы:

```
class Node:
    value: Primitive
    parent: Node | None
    children: list[Node]
```

Реализованы методы для манипуляций с деревом: `add_child`, `replace_child`, `copy_subtree`. Метод `list_nodes` возвращает список всех узлов поддерева (обход в глубину). Для контроля размера реализован метод `prune`, который усекает дерево до заданной глубины, заменяя операции на случайные терминалы.

Вычисление программы выполняется методом `eval`, который рекурсивно вычисляет значения поддеревьев и применяет операцию узла:

```
def eval(self, context: Context) -> Value:
```

```
2       return self.value.eval(
3           [child.eval(context) for child in self.children],
4           context
5       )
```

Для кроссовера реализована функция `swap_subtrees(a: Node, b: Node)`, которая обменивает два поддерева, корректно обновляя ссылки на родителей.

3.3 Хромосомы (chromosome.py)

Класс `Chromosome` инкапсулирует дерево программы вместе с множествами терминалов и операций:

```
1  class Chromosome:
2      terminals: Sequence[Primitive]
3      operations: Sequence[Primitive]
4      root: Node
```

Реализованы два метода инициализации случайных деревьев:

- `full_init(terminals, operations, max_depth)` – полная инициализация, где на каждом уровне до максимальной глубины выбираются только операции, а на последнем – только терминалы.
- `grow_init(terminals, operations, max_depth, terminal_probability)` – растущая инициализация с вероятностным выбором терминалов на каждом уровне, что создаёт деревья различной формы.

Комбинация этих методов (*ramped half-and-half*) реализована в функции `ramped_initializati` которая создаёт начальную популяцию из деревьев различных глубин, используя оба метода поровну.

3.4 Кроссовер (crossovers.py)

Реализован оператор кроссовера поддеревьев:

```
1  def crossover_subtree(parent1: Chromosome, parent2: Chromosome,
2                        max_depth: int) -> tuple[Chromosome, Chromosome]:
```

Алгоритм выбирает случайные узлы в каждом родителе (кроме корня) и обменивает соответствующие поддеревья. Если глубина потомков превышает `max_depth`, деревья усекаются методом `prune`.

3.5 Мутации (mutations.py)

Все мутации наследуются от базового класса `BaseMutation` с методом `mutate`. Реализованы четыре типа мутаций:

- `NodeReplacementMutation` – заменяет узел на другой той же арности

- `ShrinkMutation` – заменяет случайную операцию на терминал (усечение)
- `GrowMutation` – заменяет узел на случайное поддерево с контролем глубины
- `HoistMutation` – заменяет поддерево на его случайную внутреннюю часть (уменьшает размер)

Класс `CombinedMutation` позволяет комбинировать мутации с заданными вероятностями, случайно выбирая одну из них на каждом шаге.

3.6 Фитнес-функции (fitness.py)

Базовый класс `BaseFitness` определяет интерфейс для вычисления ошибки:

```
class BaseFitness(ABC):
    def __call__(self, chromosome: Chromosome) -> float:
        test_points = self.test_points_fn()
        context = {t: test_points[:, i]
                   for i, t in enumerate(chromosome.terminals)}
        predicted = chromosome.root.eval(context)
        true_values = self.target_function(test_points)
        return self.fitness_fn(chromosome, predicted, true_values)
```

Реализованы метрики ошибок: `MSEFitness` (среднеквадратичная), `RMSEFitness` (корень из MSE), `MAEFitness` (средняя абсолютная), `NRMSEFitness` (нормализованная RMSE). Класс `PenalizedFitness` добавляет штраф за размер и глубину дерева для борьбы с bloat.

3.7 Селекция (selection.py)

Реализованы три метода селекции:

- `roulette_selection` – селекция рулеткой со сдвигом для обработки отрицательных значений
- `tournament_selection(k)` – турнирная селекция размера k
- `stochastic_tournament_selection(k, p_best)` – стохастическая турнирная с вероятностью выбора лучшего

Для минимизации фитнес-функции используется инверсия знака при передаче фитнесов в селекцию.

3.8 Генетический алгоритм (ga.py)

Основная функция `genetic_algorithm(config: GARunConfig)` реализует классический цикл ГА:

1. Вычисление фитнеса: `eval_population(population, fitness_func)`
2. Сохранение элиты (если `config.elitism > 0`)
3. Селекция родителей: `config.selection_fn(population, fitnesses)`
4. Кроссовер с вероятностью p_c: попарный обмен поддеревьями
5. Мутация с вероятностью p_m
6. Замещение популяции с восстановлением элиты

Поддерживаются критерии остановки: по числу поколений, повторению лучшего результата, достижению порогового значения. История поколений сохраняется в виде списка объектов `Generation`.

Функция `save_generation` использует библиотеку Graphviz для визуализации лучшего дерева поколения. Функция `plot_fitness_history` строит графики динамики лучших и средних значений фитнеса по поколениям и сохраняет их отдельно в `fitness_best.png` и `fitness_avg.png`.

4 Результаты работы

На Рис. 3–11 представлены результаты работы генетического алгоритма со следующими параметрами:

- $N = 400$ – размер популяции.
- 10 – максимальная глубина дерева.
- $p_c = 0.85$ – вероятность кроссинговера поддеревьев.
- $p_m = 0.15$ – вероятность мутации, при этом использовалась комбинация различных вариантов:
  - Растущая мутация: $p = 0.40$
  - Узловая мутация: $p = 0.30$
  - Hoist-мутация: $p = 0.15$
  - Усекающая мутация: $p = 0.15$
- 200 – максимальное количество поколений.
- 15 – количество "элитных"особей, переносимых без изменения в следующее поколение.
- 3 – размер турнира для селекции.

На Рис. 1 и Рис. 2 показаны графики изменения среднего и лучшего значения фитнеса по поколениям.

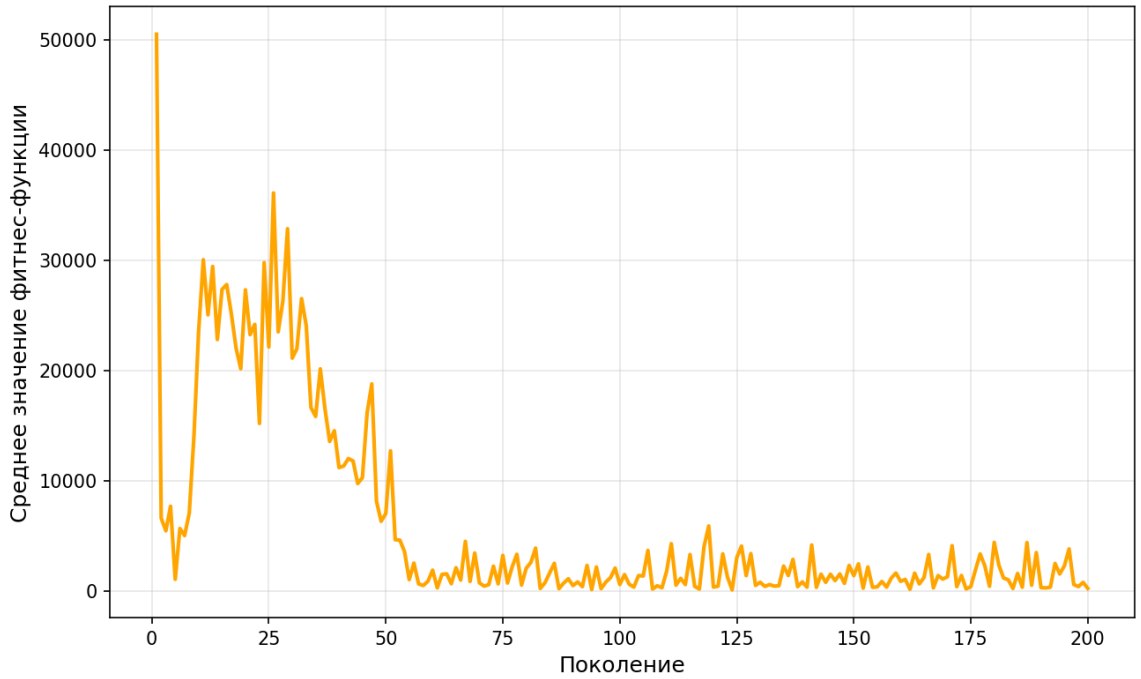

Рис. 1. График среднего значения фитнеса по поколениям

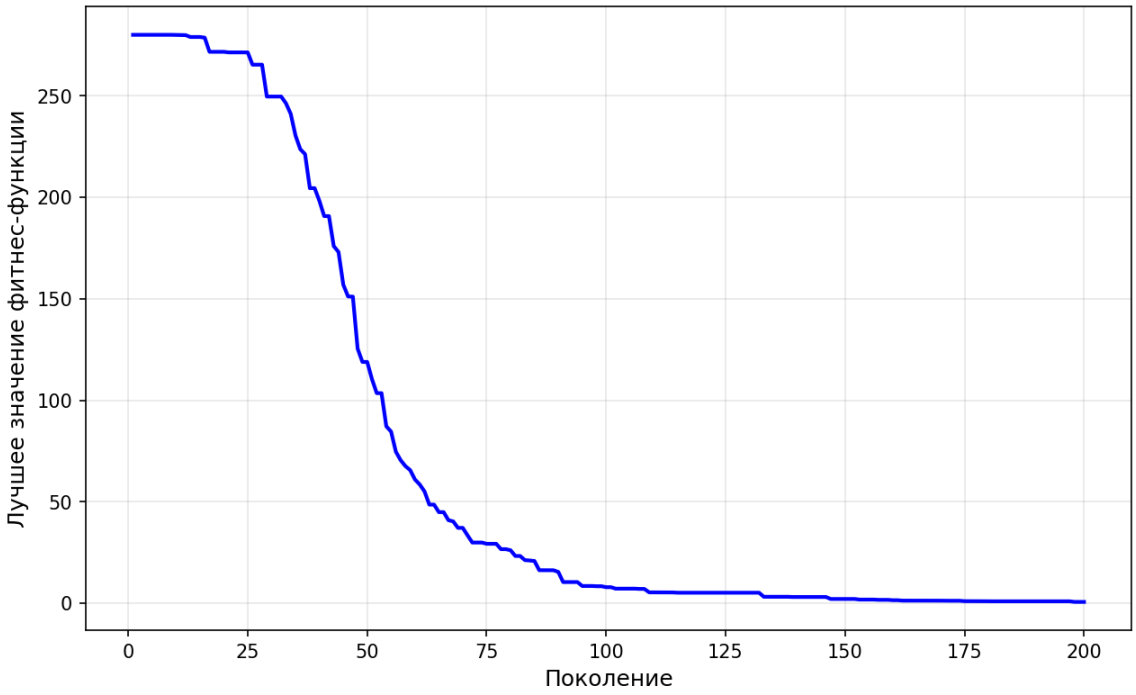

Рис. 2. График лучшего значения фитнеса по поколениям

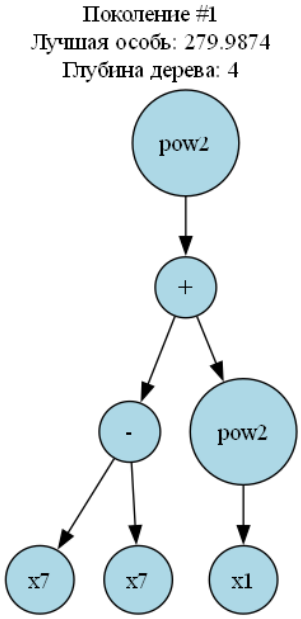

Рис. 3. Лучшая особь поколения №1

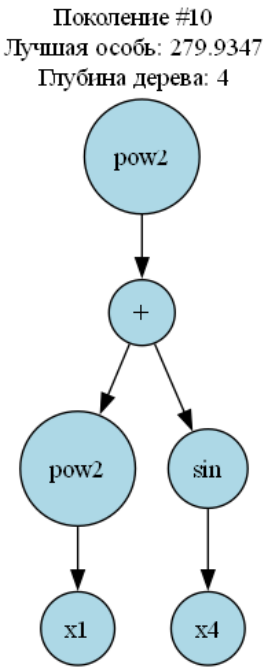

Рис. 4. Лучшая особь поколения №10

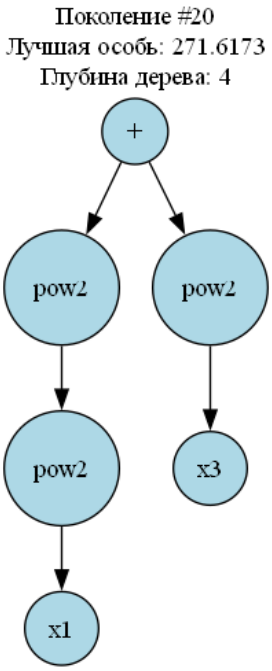

Рис. 5. Лучшая особь поколения №20

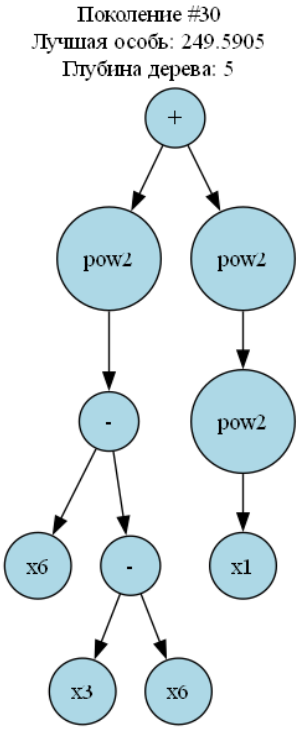

Рис. 6. Лучшая особь поколения №30

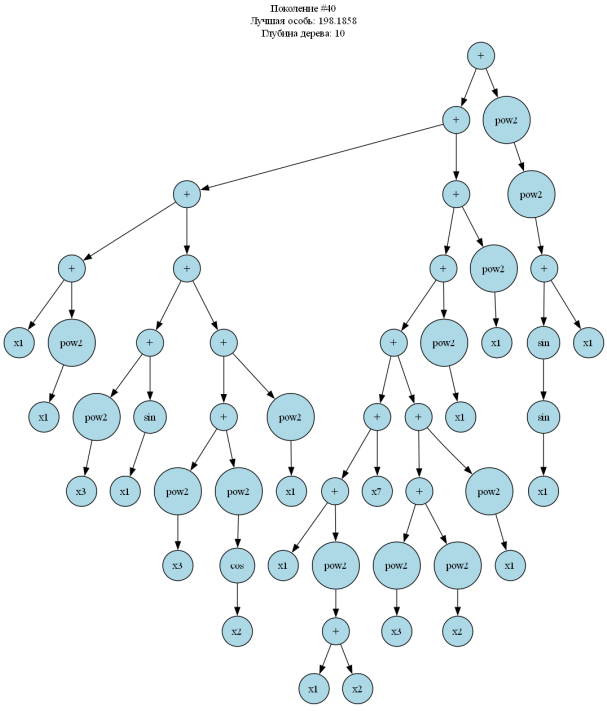

Рис. 7. Лучшая особь поколения №40

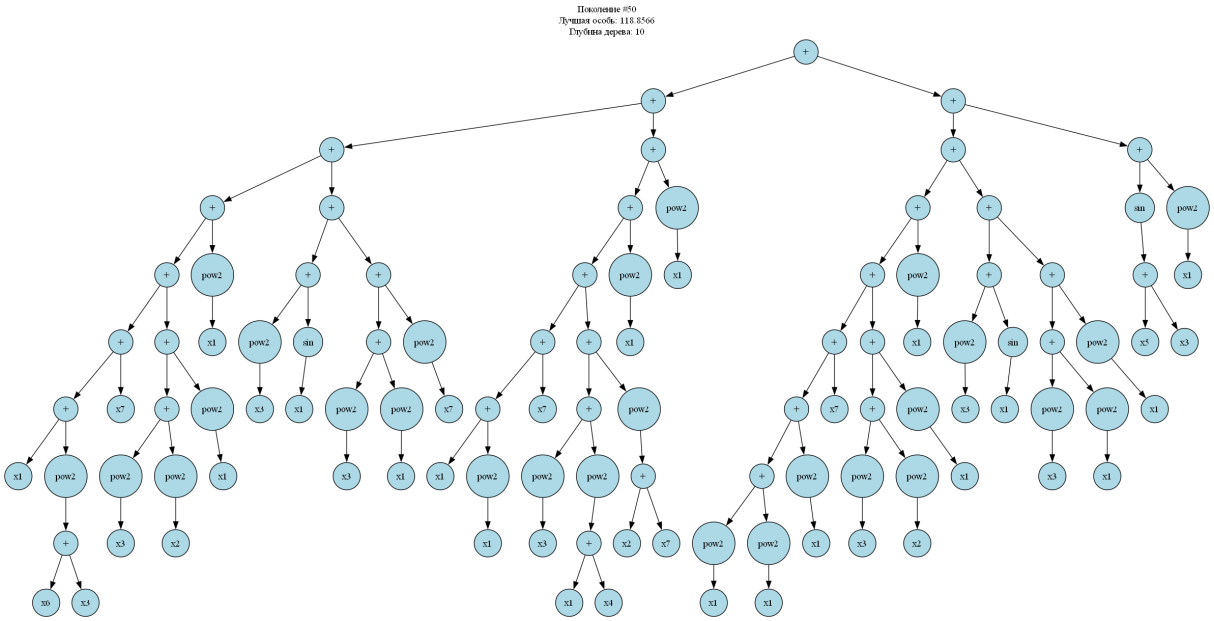

Рис. 8. Лучшая особь поколения №50

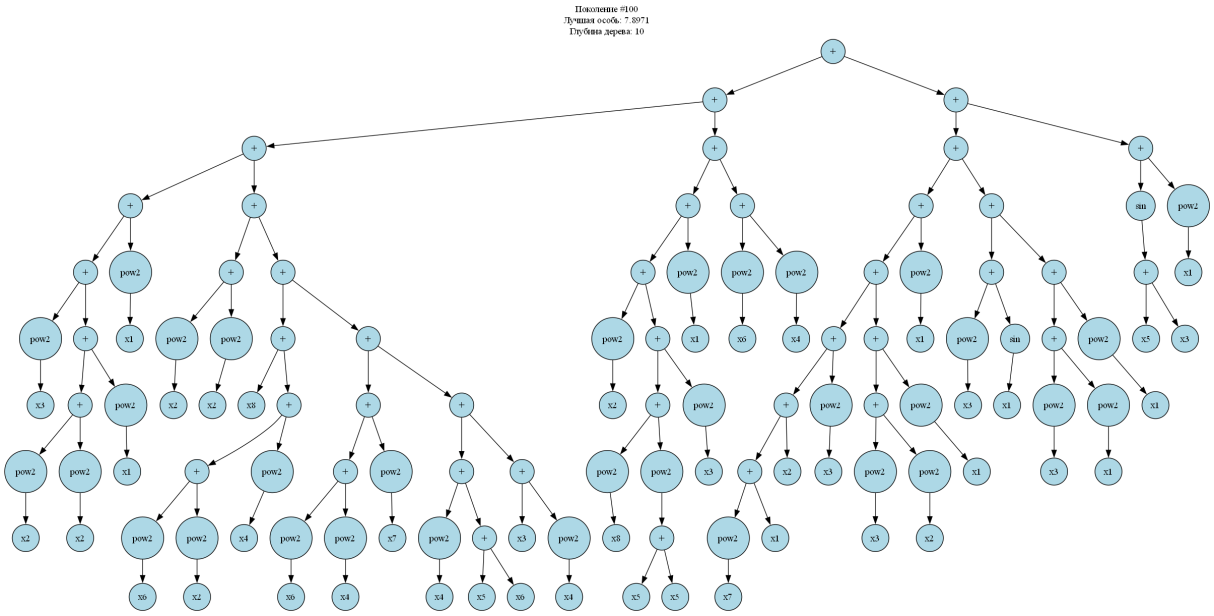

Рис. 9. Лучшая особь поколения №100

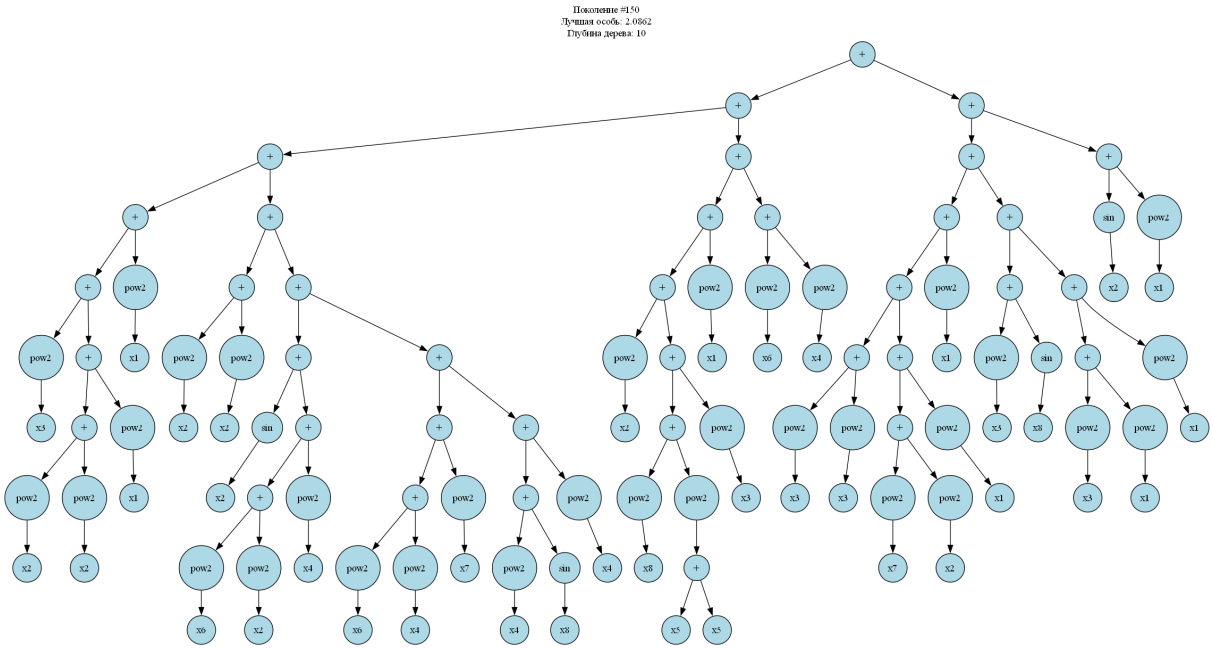

Рис. 10. Лучшая особь поколения №150

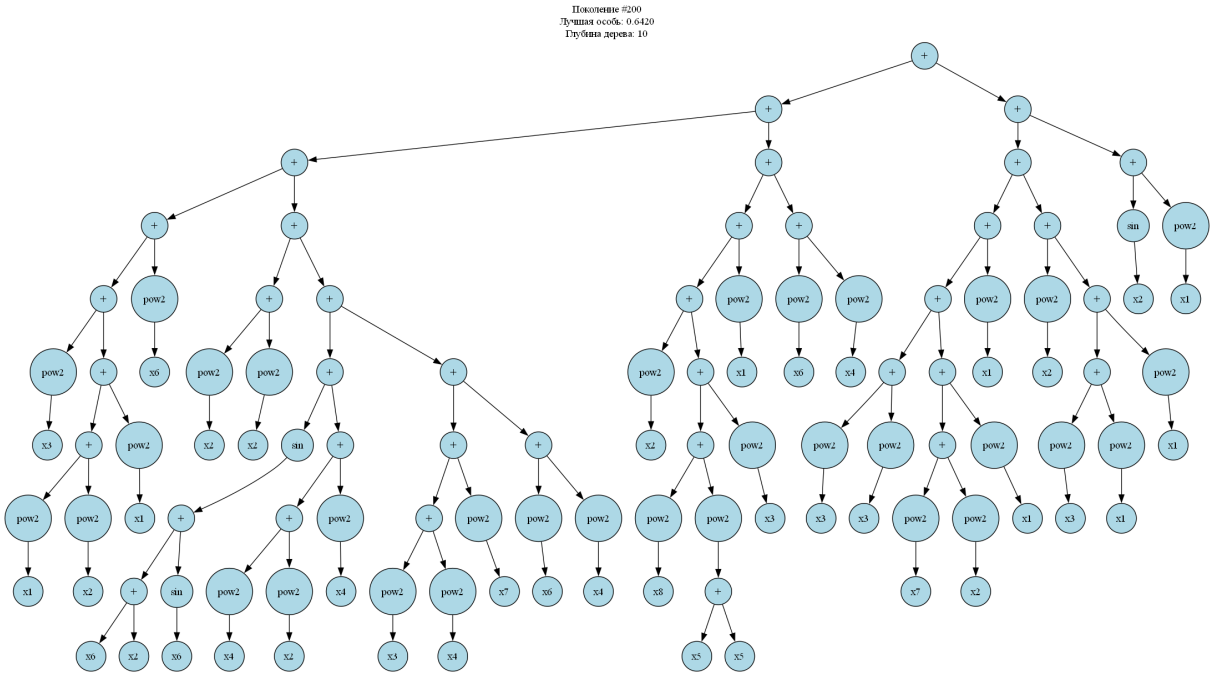

Рис. 11. Лучшая особь поколения №200

4.1 Анализ результатов

Сравнение полученных деревьев

На Рис. 12 представлено исходное дерево, на Рис. 13 представлено лучшее дерево, найденное алгоритмом.

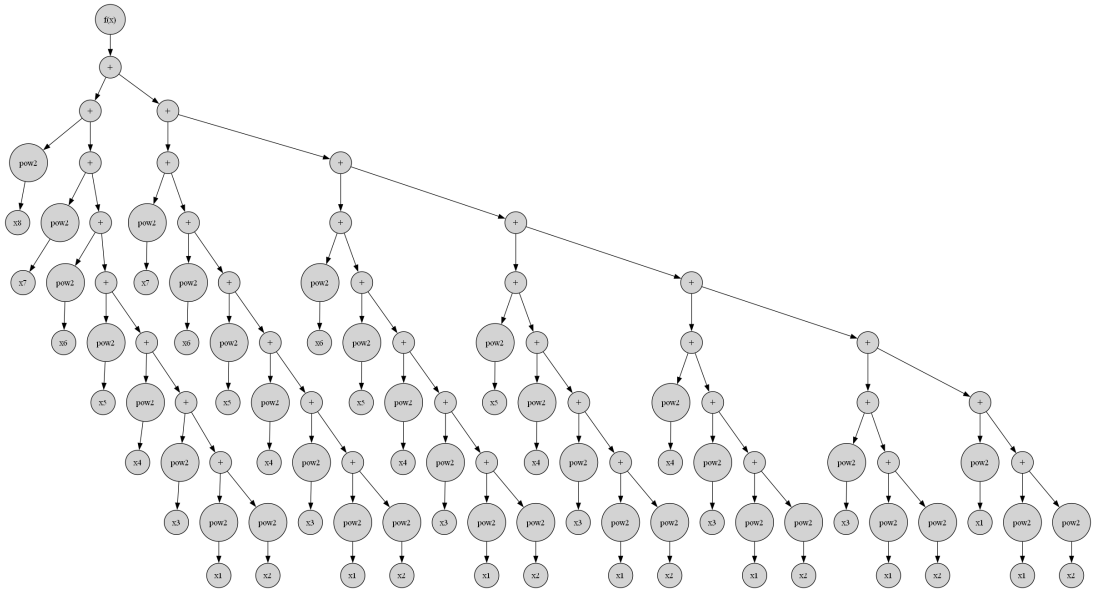

Рис. 12. Дерево целевой функции

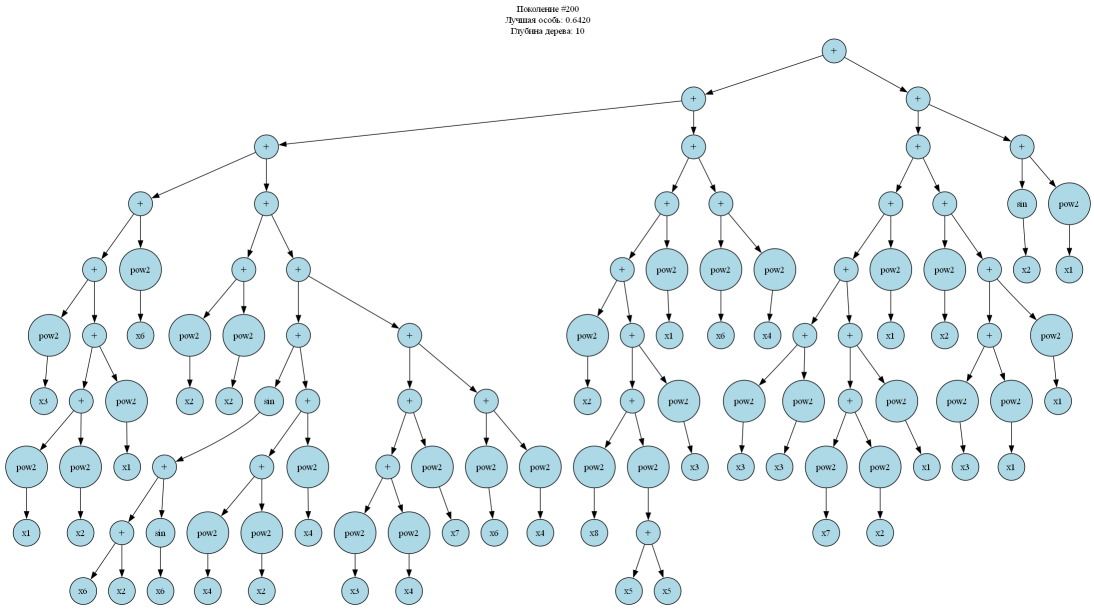

Рис. 13. Лучшая особь, найденная алгоритмом

Сравнение полученных формул

Перед сравнением, упростим исходную формулу, раскрыв знаки суммирования и перегруппировав слагаемые.

$$f(x) = \sum_{i=1}^{n} \sum_{j=1}^{i} x_j^2, \text{ для всех } j = 1, \ldots, n, \text{ при этом n } = 8.$$

$$f(x) = \underbrace{(x_1^2) + (x_1^2 + x_2^2) + \ldots + (x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 + x_6^2 + x_7^2 + x_8^2)}_{\text{всего } n{=}8 \text{ слагаемых}}$$

$$f(x) = 8x_1^2 + 7x_2^2 + 6x_3^2 + 5x_4^2 + 4x_5^2 + 3x_6^2 + 2x_7^2 + x_8^2$$

В программе реализован метод преобразования особи (дереве) в строковую формулу. Вывод программы для лучшей особи представлен ниже:

```
(((((pow2(x3) + ((pow2(x1) + pow2(x2)) + pow2(x1))) + pow2(x6)) +
((pow2(x2) + pow2(x2)) + ((sin(((x6 + x2) + sin(x6))) + ((pow2(x4) +
pow2(x2)) + pow2(x4))) + (((pow2(x3) + pow2(x4)) + pow2(x7)) + (pow2(x6) +
pow2(x4)))))) + (((pow2(x2) + ((pow2(x8) + pow2((x5 + x5))) + pow2(x3))) +
pow2(x1)) + (pow2(x6) + pow2(x4)))) + (((((pow2(x3) + pow2(x3))
+ ((pow2(x7) + pow2(x2)) + pow2(x1))) + pow2(x1)) + (pow2(x2) + ((pow2(x3) +
pow2(x1)) + pow2(x1)))) + (sin(x2) + pow2(x1))))
```

Программный метод автоматически обрамляет функции и переменные в скобки, чтобы правильно расставить приоритеты операций. Однако в данном случае они избыточны, поэтому их можно убрать:

```
pow2(x3) + pow2(x1) + pow2(x2) + pow2(x1) + pow2(x6) + pow2(x2) + pow2(x2) +
sin(x6 + x2) + sin(x6) + pow2(x4) + pow2(x2) + pow2(x4) + pow2(x3) + pow2(x4) +
pow2(x7) + pow2(x6) + pow2(x4) + pow2(x2) + pow2(x8) + pow2(x5 + x5) + pow2(x3) +
pow2(x1) + pow2(x6) + pow2(x4) + pow2(x3) + pow2(x3) + pow2(x7) + pow2(x2) + pow2(x1) +
pow2(x1) + pow2(x2) + pow2(x3) + pow2(x1) + pow2(x1) + sin(x2) + pow2(x1)
```

Переставим слагаемые:

```
pow2(x1) + pow2(x1) + pow2(x1) + pow2(x1) + pow2(x1) + pow2(x1) + pow2(x1) + pow2(x1) +
pow2(x2) + pow2(x2) + pow2(x2) + pow2(x2) + pow2(x2) + pow2(x2) + pow2(x2) +
pow2(x3) + pow2(x3) + pow2(x3) + pow2(x3) + pow2(x3) + pow2(x3) +
pow2(x4) + pow2(x4) + pow2(x4) + pow2(x4) + pow2(x4) +
pow2(x5 + x5) +
pow2(x6) + pow2(x6) + pow2(x6) +
pow2(x7)  + pow2(x7) +
pow2(x8) +
sin(x6 + x2) + sin(x6) + sin(x2)
```

Заметим, что $(x_5 + x_5)^2 = (2x_5)^2 = 4x_5^2$, а также сгруппируем слагаемые, чтобы получить финальный вид формулы, найденной алгоритмом:

$$\hat{f}(x) = 8x_1^2 + 7x_2^2 + 6x_3^2 + 5x_4^2 + 4x_5^2 + 3x_6^2 + 2x_7^2 + x_8^2 + sin(x_6 + x_2) + sin(x_6) + sin(x_2)$$

Найденная формула полностью включает в себя целевую и содержит лишь несколько лишних слагаемых.

5 Ответ на контрольный вопрос

Вопрос: Опишите древовидное представление.

Ответ:

Древовидное представление — классический подход в генетическом программировании, где программы представляются в виде синтаксических деревьев. Внутренние узлы содержат функции из функционального множества (арифметические операции, математические функции), а листья — терминалы из терминального множества (переменные и константы). Вычисление происходит рекурсивно от листьев к корню. Сложность дерева оценивается через максимальную глубину D_m (расстояние от корня до самого дальнего листа) или общее количество узлов.

Основные преимущества: естественное отображение синтаксической структуры математических выражений, гибкость в работе с выражениями различной длины и сложности, простота реализации генетических операторов (кроссовер поддеревьев, узловая мутация, растущая и усекающая мутации), автоматическое соблюдение синтаксической корректности при генерации и модификации программ. Инициализация выполняется полным методом (full) или растущим методом (grow), либо их комбинацией (ramped half-and-half).

Заключение

В ходе четвёртой лабораторной работы была успешно решена задача нахождения формулы целевой функции вида $f(x) = \sum_{i=1}^{n} \sum_{j=1}^{i} x_j^2$ с использованием генетического программирования:

1. Изучен теоретический материал о представлениях программ в генетическом программировании (древовидное, линейное, графовое) и специализированных операторах кроссинговера и мутации для древовидных структур;
2. Создана программная библиотека `gp` на языке Python с реализацией древовидного представления хромосом, кроссовера поддеревьев, четырёх типов мутаций (узловая, усекающая, растущая, Hoist-мутация), турнирной селекции и безопасных векторизованных операций;
3. Реализованы методы инициализации популяции (full, grow, ramped half-and-half), фитнес-функции на основе метрик ошибок (MSE, RMSE, MAE, NRMSE), механизм элитизма и визуализация деревьев с помощью Graphviz;
4. Проведён эксперимент с популяцией из 400 особей на 10000 тестовых точках для 8 переменных. За 200 поколений (5.9 минут) получено решение с MSE = 0.412 и RMSE = 0.642, полностью включающее целевую функцию с небольшими дополнительными слагаемыми.

Список литературы

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