**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет прикладной математики - процессов управления**

**Программа бакалавриата**

**“Большие данные и распределенная цифровая платформа”**

**ОТЧЕТ**

**по лабораторной работе №4**

**по дисциплине «Алгоритмы и структуры данных»**

**на тему «Генетический алгоритм»**

**Вариант – 9**

**Студент гр. 23Б15-пу**

**Кубякин Н.А.**

**Преподаватель**

**Дик А.Г.**

**Санкт-Петербург**

**2024 г.**

***Оглавление***

[1. Цель работы 3](#_Toc178797423)

[2. Описание задачи (формализация задачи) 3](#_Toc178797424)

[3. Теоретическая часть 4](#_Toc178797431)

[4. Основные шаги программы 5](#_Toc178797432)

[5. Блок-схема программы 6](#_Toc178797433)

[6. Описание программы 7](#_Toc178797442)

[7. Рекомендации пользователя 7](#_Toc178797443)

[8. Рекомендации программиста 8](#_Toc178797444)

[9. Исходный код программы 8](#_Toc178797445)

[10. Контрольный пример 8](#_Toc178797447)

[11. Вывод 11](#_Toc178797448)

[12. Источники 12](#_Toc178797450)

# Цель работы

# Исследование особенностей генетических алгоритмов для решения задач глобальной оптимизации.

# Описание задачи (формализация задачи)

*Цель работы* – исследование двух основных способов кодирования генотипа хромосом в генетическом алгоритме и проверка их эффективности.

Порядок выполнения лабораторной работы:

1. Изучить особенности кодирования генетических алгоритмов
2. Написать программу поиска минимума функции .
3. Протестировать программу на выбранной тестовой функции.

# Теоретическая часть

# Генетический алгоритм (ГА) представляет собой эволюционный метод оптимизации, моделирующий процессы естественного отбора и генетики. ГА оперирует популяцией индивидов, представляющих решения задачи оптимизации. В процессе эволюции, эти индивиды подвергаются генетическим операторам, таким как мутация и кроссовер, позволяя новым поколениям принимать черты успешных предыдущих поколений. Генетические алгоритмы успешно применяются в различных областях, включая оптимизацию параметров, обучение машин и другие задачи поиска глобальных экстремумов.

Генетический алгоритм (GA) - эвристический алгоритм поиска, используемый для решения задач оптимизации и моделирования путём случайного подбора, комбинирования и вариации искомых параметров с использованием механизмов, аналогичных естественному отбору в природе. Является разновидностью эволюционных вычислений, с помощью которых решаются оптимизационные задачи с использованием методов естественной эволюции, таких как наследование, мутации, отбор и кроссинговер. Отличительной особенностью генетического алгоритма является акцент на использование оператора «скрещивания», который производит операцию рекомбинации решений-кандидатов, роль которой аналогична роли скрещивания в живой природе. В рамках GA, каждый "ген" в пространстве поиска представляет собой потенциальное решение задачи оптимизации. Применяется для нахождения приближенных решений задачи оптимизации. В контексте генетического алгоритма используются следующие ключевые термины:

* **Ген (Gene):** Отдельный элемент в генетическом коде, представляющий параметр или характеристику потенциального решения задачи оптимизации.
* **Функция приспособленности (Fitness Function):** Оценочная функция, измеряющая качество решения в данной точке пространства параметров.
* **Мутация (Mutation):** Процесс изменения генетического кода для внесения случайных изменений и разнообразия в популяцию.
* **Скрещивание (Crossover):** Процесс комбинирования генетического материала от двух родительских хромосом для создания потомства.
* **Селекция (Selection):** Процесс выбора индивидов для участия в процессе мутации и скрещивания на основе их функции приспособленности.
* **Итерации (Generations):** Шаги оптимизационного процесса, на каждом из которых происходит обновление популяции.

Основные шаги алгоритма:

1. **Инициализация:** 
   * Генерация генов со случайными значениями.
2. **Основной цикл оптимизации:**
   * Мутация некоторого количества генов
   * Скрещивание генов исходной популяции.
   * Селекция генов
3. **Повторение:**
   * Повторение шага 2 заданное количество раз (генераций).
4. **Вывод результатов:**
   * Вывод результатов

В данной работе реализована модификация алгоритма селекции, известная как стратегия элитарной селекции с частичной заменой популяции. Суть данной стратегии заключается в следующем: 10% лучших особей из предыдущего поколения переносятся в следующее поколение без изменений, независимо от их пригодности для достижения оптимального решения. Эта модификация способствует обеспечению разнообразия в популяции и сохранению наиболее успешных особей для следующих итераций алгоритма.

# Основные шаги программы

1. Инициализация генов со случайными значениями из заданного промежутка.
2. Мутация некоторого количества генов с заданной вероятностью.
3. Скрещивание генов исходной популяции.
4. Выполнение селекции (формирование нового поколения).
5. Повтор 2-4 заданное количество раз.
6. Вывод результатов в графическом интерфейсе.

# Блок-схема программы

# На рисунке представлена блок-схема алгоритма.

# 

# Рис. 5 Блок-схема генетического алгоритма.

# Описание программы

# Алгоритм реализован на языке Python 3.10 с использованием библиотеки PyQT для визуализации и некоторых встроенных библиотек. Так же программа выполнена в функциональном стиле и не содержит классов (за исключением GUI). Вся логика программы заключена в следующих функциях:

Таблица 6 Описание функций

|  |  |
| --- | --- |
| Имя модуля | Назначение |
| encode | Кодирует кортеж из двух вещественных чисел в бинарное представление. |
| decode | Декодирует бинарное представление вещественных чисел в кортеж из двух вещественных чисел. |
| initialize\_genes | Инициализирует список генетических индивидуумов случайными значениями в пределах заданных границ. |
| func | Вычисляет значение математической функции, заданной строкой f, для входных значений x1 и x2. |
| mutate | Производит мутацию генетического индивида. |
| crossover | Производит кроссовер между двумя генами. Отбор генов идет в функции algorithm. |
| algorithm | Выполняет генетический алгоритм для оптимизации функции. |

# Рекомендации пользователя

1. Запустить main.py для запуска графического интерфейса программы.
2. Ввести в поле нужную функцию (заранее введена функция , умножение - \*, возведение в степень - \*\*)
3. Выбрать в полях соответствующие величины.
4. Поставить галочки по желанию для выбора кодировки и отсутствия или присутствия модификации элитизма.
5. Нажать кнопку “Рассчитать хромосомы”.
6. Нажать кнопку “Рассчитать”. В поле под кнопкой отобразится лучшее значение и соответствующие значения переменных. В большом поле снизу отобразится таблица хромосом.

# Рекомендации программиста

Для запуска программы необходима 64-битная операционная система Windows, Linux или macOS. Для работы с кодом необходима среда разработки, совместимая с python 3.1 и библиотеки random, struct [[1]](#struct), copy [[2]](#copy), typing [[3]](#typing), PyQt5 [[4]](#pyqt5).

Для запуска программы необходимо установить Python версии не ниже 3.10, а так же библиотеки random, struct [[1]](#struct), copy [[2]](#copy), typing [[3]](#typing), PyQt5 [[4]](#pyqt5).

# Исходный код программы

# Исходный код программы и необходимые текстовые файлы доступны по ссылке: <https://github.com/NikiTaku1/spbu_alg/tree/main/lab4>

# Контрольный пример

В данном разделе представлен пример, демонстрирующий работу программы.

1) Запуск main.py (заранее введены контрольные значения).

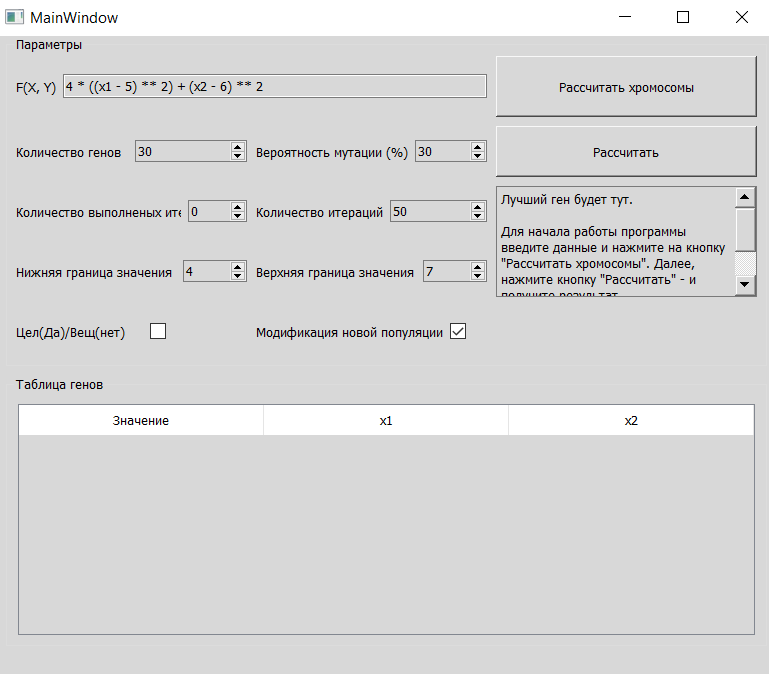


Рис 10.1 Графический интерфейс программы.

2) Нажатие кнопок “Рассчитать хромосомы” и “Рассчитать”

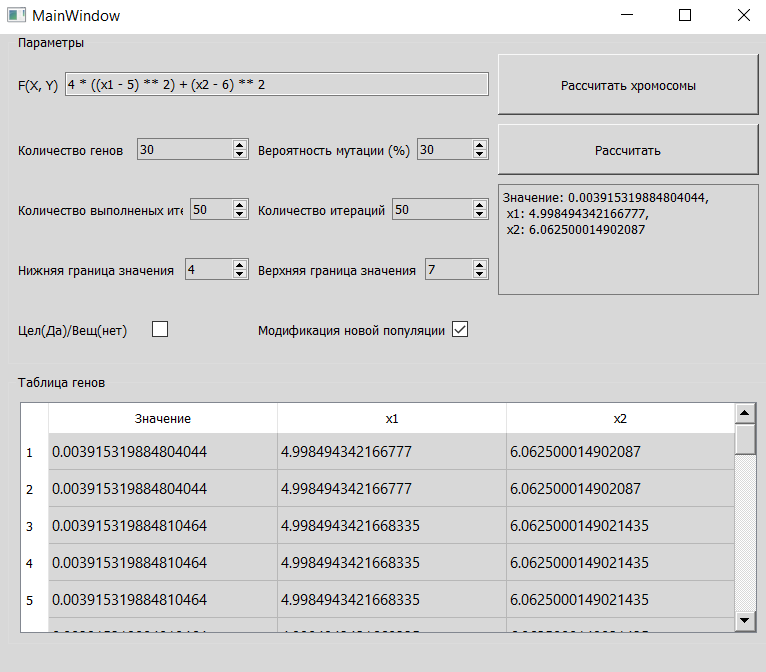


Рис 10.2 Графический интерфейс после нажатия кнопок.

3) Анализ точности вычислений в зависимости от параметров (10.3 - % мутации = 30%, N генов = 30; 10.4 – N итераций = 50, % мутации = 30%; 10.5 – N итераций = 50, N генов = 30)

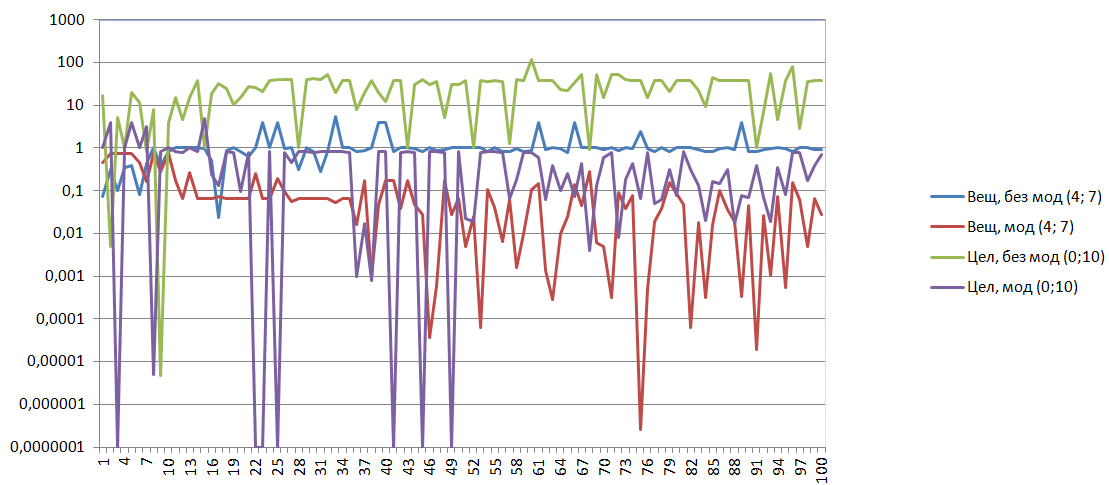


Рис 10.3 График зависимости точности от количества итераций

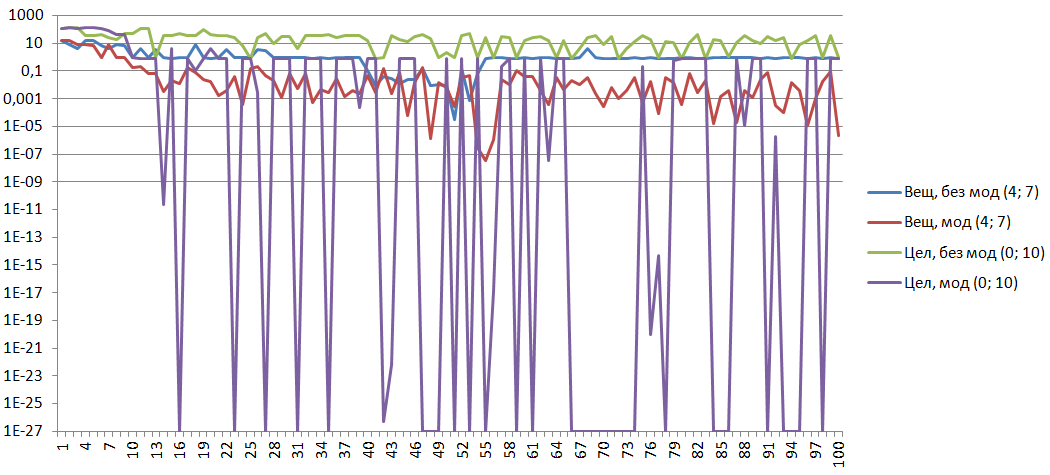


Рис 10.4 График зависимости точности от количества генов

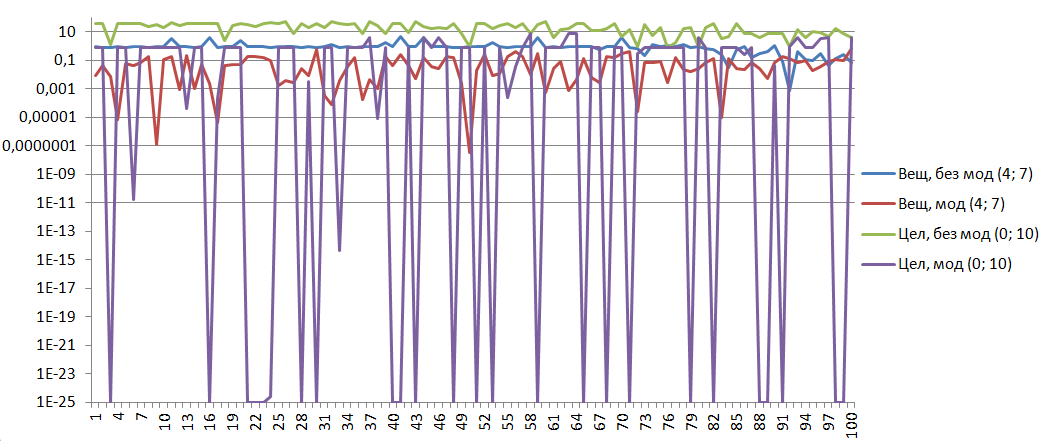


Рис 10.5 График зависимости точности от процента мутации

# Листинг

# 1) genetic\_algorithm.py

import random

import struct

from copy import deepcopy

from typing import TypedDict

def encode(x: tuple[float, float]) -> tuple[str, str]:

"""  
 Переводит кортеж из двух float в двоичную систему.  
  
 Параметры:  
 - x: Кортеж из двух float.  
  
 Вывод:  
 Кортеж двух строк.  
 """

    binary\_representation = format(struct.unpack(">Q", struct.pack(">d", x[0]))[0], "064b"), format(struct.unpack(">Q", struct.pack(">d", x[1]))[0], "064b")

    return binary\_representation

def decode(binary\_str: tuple[str, str]) -> tuple[float, float]:

"""  
 Декодирует кортеж из двух строк в двоичной системе в float.  
  
 Параметры:  
 - binary\_str: Кортеж из двух строк.  
  
 Вывод:  
 Кортеж из двух float.  
 """

    float\_value1 = struct.unpack(">d", int(binary\_str[0], 2).to\_bytes(8, "big"))[0]

    float\_value2 = struct.unpack(">d", int(binary\_str[1], 2).to\_bytes(8, "big"))[0]

    return float\_value1, float\_value2

def initialize\_genes(flag: bool, bounds, amount):

"""  
 Инициализирует список хромосом.  
  
 Параметры:

- flag: Флаг на выбор кодировки.  
 - bounds: Кортеж содержаший нижнюю и верхнюю границу значений.  
 - amount: Количество хромосом для генерации.  
  
 Вывод:  
 Список кортежей обозначающих хромосомы.  
 """

    genes = []

    if flag == True:

        for \_ in range(amount):

            genes.append(

                (random.randint(bounds[0], bounds[1]+1), random.randint(bounds[0], bounds[1]+1))

            )

    else:

        for \_ in range(amount):

            genes.append(

                (random.uniform(bounds[0], bounds[1]), random.uniform(bounds[0], bounds[1]))

            )

    return genes

def func(f, x1, x2):

"""  
 Вычисляет математическую функцию.  
  
 Параметры:  
 - f: Функция в виде строки.  
 - x1: Первая переменная.  
 - x2: Вторая переменная.  
  
 Вывод:  
 Значение функции при данных переменных.  
 """

    return eval(f)

def mutate(gene, bounds) -> tuple[float, float]:

"""  
 Выполняет мутацию над геном.  
  
 Параметры:  
 - gene: Кортеж обозначающий ген.  
 - bounds: Кортеж содержащий нижнюю и верхнюю границу.  
  
 Вывод:  
 Мутировавший ген.  
 """

    mutated\_gene = list(encode(gene))

    bit\_position = random.randint(0, 63)

    for i in range(len(mutated\_gene)):

        # Convert binary string to list of characters

        binary\_list = list(mutated\_gene[i])

        binary\_list[bit\_position] = "1" if binary\_list[bit\_position] == "0" else "0"

        # Convert back to a binary string

        mutated\_gene[i] = "".join(binary\_list)

    # Convert back to binary string and then decode to float values

    mutated\_gene = (

        max(bounds[0], min(bounds[1], decode(mutated\_gene)[0])),

        max(bounds[0], min(bounds[1], decode(mutated\_gene)[1])),

    )

    return mutated\_gene

def crossover(gene1, gene2, bounds):

"""  
 Выполняет кроссовер.  
  
 Параметры:  
 - gene1: Первый ген.  
 - gene2: Второй ген.  
 - bounds: Кортеж содержащий нижнюю и верхнюю границу.  
  
 Вывод:  
 Кортеж двух новых генов после кроссовера.  
 """

    gene1 = encode(gene1)

    gene2 = encode(gene2)

    crossover\_point = random.randint(2, 12)

    new\_gene1 = (

        gene1[0][:crossover\_point] + gene2[0][crossover\_point:],

        gene1[1][:crossover\_point] + gene2[1][crossover\_point:],

    )

    new\_gene2 = (

        gene2[0][:crossover\_point] + gene1[0][crossover\_point:],

        gene2[1][:crossover\_point] + gene1[1][crossover\_point:],

    )

    gene1 = (

        max(bounds[0], min(bounds[1], decode(new\_gene1)[0])),

        max(bounds[0], min(bounds[1], decode(new\_gene1)[1])),

    )

    gene2 = (

        max(bounds[0], min(bounds[1], decode(new\_gene2)[0])),

        max(bounds[0], min(bounds[1], decode(new\_gene2)[1])),

    )

    return gene1, gene2

def algorithm(flag: bool, genes, generations, f, mutation, bounds):

"""  
 Выполняет генетический алгоритм.  
  
 Параметры:

- flag: Флаг на включение модификации.  
 - genes: Список генов.  
 - generations: Количество поколений.  
 - f: Функция в виде строки.  
 - mutation: Вероятность мутации в процентах.  
 - bounds: Кортеж содержащий нижнюю и верхнюю границу.  
  
 Вывод:  
 Список, содержащий историю генов по поколениям.  
 """

    history = [deepcopy(sorted(genes, key=lambda x: func(f, \*x)))]

    for \_ in range(generations):

        for i in range(len(genes)):

            if random.uniform(0, 1) > mutation / 100:

                genes[i] = mutate(genes[i], bounds)

        for i in range(0, len(genes) - len(genes) % 2, 2):

            genes[i], genes[i + 1] = crossover(genes[i], genes[i + 1], bounds)

        if flag == True:

            genes = (

                sorted(genes, key=lambda x: func(f, \*x))[

                : len(genes) // 10 \* 9 + len(genes) % 10

                ]

                + history[-1][: len(genes) // 10]

            )

        genes = sorted(genes, key=lambda x: func(f, \*x))

        history.append(deepcopy(genes))

    return history

# 2) main.py

from PyQt5 import QtWidgets, uic

from genetic\_algorithm import initialize\_genes, algorithm, func

from copy import deepcopy

class GeneticAlgorithmGUI(QtWidgets.QMainWindow):

"""  
 Создает графический интерфейс.  
  
 """

    def \_\_init\_\_(self):

        super(GeneticAlgorithmGUI, self).\_\_init\_\_()

        uic.loadUi("ui.xml", self)

        self.setStyleSheet(

            """

            background-color: #D8D8D8;

            color: #000000;

            QPushButton {

                background-color: #D8D8D8;

                color: #000000;

            }

            QPushButton:hover {

                background-color: #D8D8D8;

            }

            """

        )

        # Подключение функций к кнопкам

        self.pushButton.clicked.connect(self.calculate\_chromosomes)

        self.pushButton\_2.clicked.connect(self.calculate)

        self.tableWidget.setColumnCount(3)

        self.tableWidget.setHorizontalHeaderLabels(["Значение", "x1", "x2"])

        self.tableWidget.horizontalHeader().setSectionResizeMode(

            QtWidgets.QHeaderView.Stretch

        )

        self.genes = []

        self.funcQLE.setText("4 \* ((x1 - 5) \*\* 2) + (x2 - 6) \*\* 2")

        self.spinBox.setMinimum(0)

        self.spinBox.setMaximum(10000)

        self.spinBox\_2.setMinimum(0)

        self.spinBox\_2.setMaximum(100)

        self.spinBox\_4.setMinimum(0)

        self.spinBox\_4.setMaximum(10000)

        self.spinBox\_5.setMinimum(0)

        self.spinBox\_5.setMaximum(100000)

        self.spinBox\_6.setMinimum(-10000)

        self.spinBox\_6.setMaximum(10000)

        self.spinBox\_7.setMinimum(-10000)

        self.spinBox\_7.setMaximum(10000)

        self.spinBox.setValue(30)

        self.spinBox\_2.setValue(30)

        self.spinBox\_4.setValue(0)

        self.spinBox\_5.setValue(50)

        self.spinBox\_6.setValue(4)

        self.spinBox\_7.setValue(7)

        self.checkBox\_1.setChecked(False)

        self.checkBox\_2.setChecked(True)

    def calculate\_chromosomes(self):

"""  
 Вычисляет хромосомы.  
  
 """

        num\_genes = self.spinBox.value()

        lower\_bound = self.spinBox\_6.value()

        upper\_bound = self.spinBox\_7.value()

        area = (lower\_bound, upper\_bound)

        if self.checkBox\_1.isChecked():

            self.genes = initialize\_genes(True, area, num\_genes)

        else:

            self.genes = initialize\_genes(False, area, num\_genes)

    def calculate(self):

"""  
 Вычисляет необходимые значения.  
"""

        if not self.genes:

            self.calculate\_chromosomes()

        func\_str = self.funcQLE.text()

        mutation\_probability = self.spinBox\_2.value()

        previous\_iterations = self.spinBox\_4.value()

        total\_iterations = self.spinBox\_5.value()

        shown\_iterations = previous\_iterations + total\_iterations

        previous\_iterations = max(previous\_iterations, total\_iterations)

        self.spinBox\_4.setValue(shown\_iterations)

        lower\_bound = self.spinBox\_6.value()

        upper\_bound = self.spinBox\_7.value()

        area = (lower\_bound, upper\_bound)

        if self.checkBox\_2.isChecked():

            result = algorithm(

                True, deepcopy(self.genes), shown\_iterations, func\_str, mutation\_probability, area

            )

        else:

            result = algorithm(

                False, deepcopy(self.genes), shown\_iterations, func\_str, mutation\_probability, area

            )

        self.display\_genes\_in\_table(result[previous\_iterations])

        self.display\_best\_gene(result[-1])

    def display\_genes\_in\_table(self, contents):

"""  
 Выводит значения в таблицу.  
"""

        self.tableWidget.setRowCount(0)

        for i, gene in enumerate(contents):

            func\_str = self.funcQLE.text()

            value = func(func\_str, \*gene)

            position = gene

            # Добавляем новую строку в таблицу

            row\_position = self.tableWidget.rowCount()

            self.tableWidget.insertRow(row\_position)

            # Заполняем ячейки таблицы

            self.tableWidget.setItem(

                row\_position, 1, QtWidgets.QTableWidgetItem(str(position[0]))

            )

            self.tableWidget.setItem(

                row\_position, 2, QtWidgets.QTableWidgetItem(str(position[1]))

            )

            self.tableWidget.setItem(

                row\_position, 0, QtWidgets.QTableWidgetItem(str(value))

            )

    def display\_best\_gene(self, genes):

"""  
 Выводит лучшие значения.  
"""

        best\_gene = genes[0]

        func\_str = self.funcQLE.text()

        value = func(func\_str, \*best\_gene)

        result\_text = f"Значение: {value},\n x1: {best\_gene[0]},\n x2: {best\_gene[1]}"

        self.plainTextEdit.setPlainText(result\_text)

if \_\_name\_\_ == "\_\_main\_\_":

    app = QtWidgets.QApplication([])

    window = GeneticAlgorithmGUI()

    window.show()

    app.exec\_()

# Вывод

Были получены все нужные навыки для реализации генетических алгоритмов, а также разработана программа, реализующая генетический алгоритм с графическим интерфейсом. Проанализированы результаты работы алгоритма, выполнено сравнение кодировок и параметров.

# Источники

# struct — struct documentation // Documentation URL: <https://docs.python.org/3/library/struct.html> (дата обращения: 08.11.2024).

# copy — copy documentation // Documentation URL: <https://docs.python.org/3/library/copy.html> (дата обращения: 08.11.2024).

# typing — typing documentation // Documentation URL: <https://docs.python.org/3/library/typing.html> (дата обращения: 08.11.2024).

# PyQt5 — PyQt5 documentation // Documentation URL: <https://doc.qt.io/qtforpython-5/contents.html> (дата обращения: 08.11.2024).