Министерство образования и науки РФ

Санкт-Петербургский Политехнический университет Петра Великого

Институт компьютерных наук и технологий

Высшая школа искусственного интеллекта

Направление 02.03.01 «Математика и компьютерные науки»

Дисциплина «Генетические алгоритмы»

Отчёт по лабораторной работе №1 «Простой генетический алгоритм»

Студент: Федотов С. Ю., гр. 5130201/00101

Преподаватель: Большаков А.А.

Санкт-Петербург — 2023

Содержание

1 Постановка задачи

В рамках данной лабораторной работы необходимо:

1. Разработать простой генетический алгоритм для нахождения максимума функции $\frac{sin(x)}{(1+e^{-x})}$ в диапазоне [0.5, 10]
2. Исследовать зависимость времени поиска, итераций, точности нахождения решения от основных параметров генетического алгоритма:
   - число особей в популяции
   - вероятность кроссинговера, мутации;
3. Вывести на экран график данной функции с указанием найденного экстремума для каждого поколения.
4. Сравнить найденное решение с действительным.

2 Теоретические сведения

2.1 Простой генетический алгоритм

ГА используют принципы и терминологию, заимствованные у биологической науки – генетики. В ГА каждая особь представляет потенциальное решение некоторой проблемы. В классическом ГА особь кодируется строкой двоичных символов – хромосомой, каждый бит которой называется геном. Множество особей – потенциальных решений составляет популяцию. Поиск (суб)оптимального решения проблемы выполняется в процессе эволюции популяции - последовательного преобразования одного конечного множества решений в другое с помощью генетических операторов репродукции, кроссинговера и мутации.

ГА берет множество параметров оптимизационной проблемы и кодирует их последовательностями конечной длины в некотором конечном алфавите (в простейшем случае двоичный алфавит «0» и «1») . Предварительно простой ГА случайным образом генерирует начальную популяцию стрингов (хромосом). Затем алгоритм генерирует следующее поколение (популяцию) с помощью трех основных генетических операторов:

1. Оператор репродукции (ОР);
2. Оператор скрещивания (кроссинговера, ОК);
3. Оператор мутации (ОМ).

Генетические операторы являются математической формализацией приведенных выше трех основополагающих принципов Дарвина, Менделя и де Вре естественной эволюции.

ГА работает до тех пор, пока не будет выполнено заданное количество поколений (итераций) процесса эволюции или на некоторой генерации будет получено заданное качество или вследствие преждевременной сходимости при попадании в некоторый локальный оптимум.

Схема простого генетического алгоритма представлен на рисунке 1.

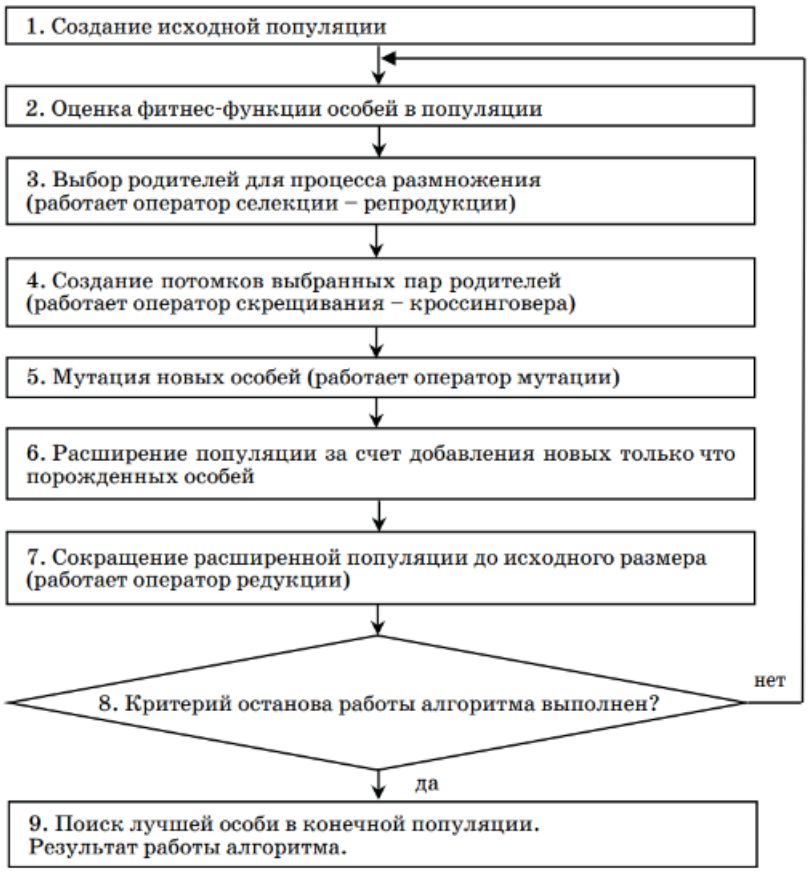

Рис. 1: Простой генетический алгоритм

2.2 Оператор репродукции

Для селекции хромосом используется случайный поиск на основе колеса рулетки. Хромосома, которой соответствует больший сектор рулетки, имеет большую вероятность попасть в следующее поколение. В результате выполнения оператора репродукции формируется промежуточная популяция, хромосомы которой будут использованы для построения поколения с помощью операторов скрещивания.

Вероятность выпадения каждой хромосомы $P(x_i) = \frac{f(x_i)}{\sum f(x_i)}$

2.3 Оператор кроссинговера

Одноточечный или простой оператор кроссинговера (ОК) с заданной вероятностью P_c выполняется в 3 этапа:

1-й этап. Две хромосомы (родители)

$$A = a_1 a_2 \dots a_k a_{k+1} \dots a_L$$

$$B = b_1 b_2 \dots b_k b_{k+1} \dots b_L$$

выбираются случайно из промежуточной популяции, сформированной при помощи оператора репродукции (ОР).

2-й этап. Случайно выбирается точка скрещивания - число k из диапазона [1, 2, . . . , n - 1], где n – длина хромосомы (число бит в двоичном коде).

3-й этап. Две новых хромосомы A', B' (потомки) формируются из A и B путем обмена подстрок после точки скрещивания:

$$A = a_1 a_2 \dots a_k b_{k+1} \dots b_L$$
$$B = b_1 b_2 \dots b_k a_{k+1} \dots a_L$$

Следует отметить, что ОК выполняется с заданной вероятностью P_c (отобранные два родителя не обязательно производят потомков). Обычно величина $P_c \approx 0.5$

2.4 Оператор мутации

Далее согласно схеме классического ГА с заданной вероятностью P_m выполняется оператор мутации. Иногда этот оператор играет вторичную роль. Обычно вероятность мутации мала -$P_m \approx 0.001$.

Оператор мутации (ОМ) выполняется в 2 этапа:

1-й этап. В хромосоме $A = a_1 a_2 \dots a_L$ случайно выбирается k-ая позиция (бит) мутации $(1 \leq k \leq n)$.

2-й этап. Производится инверсия значения гена в k-й позиции $a'_k = \bar{a}_k$.

2.5 Представление вещественных решений

Для представления вещественного решения (хромосомы) x будем использовать двоичный вектор. Его длина зависит от требуемой точности решения, которую в данном случае положим 3 знака после запятой. Поскольку интервалы области решения имеет длину 9.5, для достижения заданной точности $[0.5, 10]$ должны быть разбиты на равные части, число которых должно быть не менее $9.5 * 1000 - 1$. В качестве двоичного представления используется двоичный код номера отрезка. Этот код позволяет определить соответствующее ему вещественное число, если известны границы области решения. Отсюда следует, что двоичный вектор для кодирования вещественного решения должен иметь 14 бит, поскольку $8192 = 2^{13} < 15999 < 2^{14} = 16384$.

Это позволяет разбить интервалы $[0.5, 10]$ на 16384 частей и обеспечить необходую точность. Отображение из двоичного представления $(b_{13} \dots b_0)$, $b_i \in \{0, 1\}$ в вещественное число из $[0.5, 10]$ выполняется в два шага:

1) Перевод двоичного числа в десятичное:

$$(b_{12} \dots b_0)_2 = \left(\sum_{i=0}^{12} b_i * 2^i \right)_{10} = x'$$

2) Вычисление соответствующего вещественного числа x:

$$x = \frac{0.5 + x' * 10.5}{(2^{14} - 1)}$$

3 Программная реализация

Алгоритм реализован на языке выского уровня Python. Был создан класс Population, реализующий популяцию и ее эволюцию.

3.1 Описание полей класса

- population - массив, хранящий всех особей в популяции.
- power - мощность популяции.
- p_kross - вероятность оператора кроссинговера.
- p_mut - вероятност оператора мутации.
- func - фитнесс функция.

3.2 Создание популяции

Создание начальное популяции реализована в конструкторе класса Population. Популяция заполняется случайными особями.

```
# Создание начальной популяции
  for i in range(0, power):
      chromosome = ""
      for j in range(0, 14):
           gen = random.randint(0, 1)
           chromosome += str(gen)
       self.population.append(chromosome)
```

3.3 Представление вещественных решений

Для представление хромосомы в виде вещественного числа в диапазоне [0.5, 10] реализована функция value. Функция value также высчитывает значение фитнесс функции для 10-ого представления каждой хромосомы.

```
  def value(self, chromosome):
      decimal_value = int(chromosome, 2)  #перевод числа в 10-ый вид.
      return self.func(0.5 + (decimal_value * 10.5) / (2 ** 14 - 1))
```

3.4 Оператор репродукции

Оператор репродукции реализован с помощью функции reproduction. Формируются вероятности выпадения каждой особи, с помощью функции choice из библиотеки random, исходя из посчитанных вероятностей, выбираются N особей, где N - мощность популяции.

```
  def reproduction(self, sum):
     probabilities = []

     for individual in self.population:  # Формирование вероятностей
         probabilities.append((self.value(individual) + 1) / sum)
```

```
        new_population =
        random.choices(self.population,
                       probabilities,
                       k=self.power)  # Вращение колеса

        self.population = new_population
```

3.5 Оператор Кроссинговера

Оператор кроссинговера реализован с помощью функции krossingover. Случайно выбираются 2 особи, случайно выбирается число k, хромосомы каждой особи меняются по алгоритму кроссинговера. Оператор кроссинговера выполняется с вероятностью, заданной на этапе создания популяции.

```
    def krossingover(self, p):
        random_number = random.random()
        if random_number < p:
            # Выбор первой особи
            individual1 = random.choice(self.population)
            self.population.remove(individual1)

            # Выбор второй особи
            individual2 = random.choice(self.population)
            self.population.remove(individual2)

            k = random.randint(0, 13)

            # Изменение хромосом особей
            for i in range(0, 14):
                if i < k:
                    char_list = list(individual1)
                    char_list[i] = individual2[i]
                    individual1 = "".join(char_list)
                else:
                    char_list = list(individual2)
                    char_list[i] = individual1[i]
                    individual2 = "".join(char_list)

            self.population.append(individual1)
            self.population.append(individual2)
```

3.6 Оператор мутации

Оператор кроссинговера реализован с помощью функции mutation. Для каждой хромосомы, с вероятностью, заданной на этапе создания популяции изменяется случайный ген.

```
    def mutation(self, p):
        for chromosome in self.population:
            random_number = random.random()
            if random_number < p:
```

```
self.population.remove(chromosome)
k = random.randint(0, 13)
if chromosome[k] == "0":
    char_list = list(chromosome)
    char_list[k] = "1"
    chromosome = "".join(char_list)
else:
    char_list = list(chromosome)
    char_list[k] = "0"
    chromosome = "".join(char_list)
self.population.append(chromosome)
```

4 Результат работы программы

На рис. 2 - 5 приведен результат эволюции популяции из 20 особей с вероятностью кроссинговера 0.5, вероятность мутации 0.01.

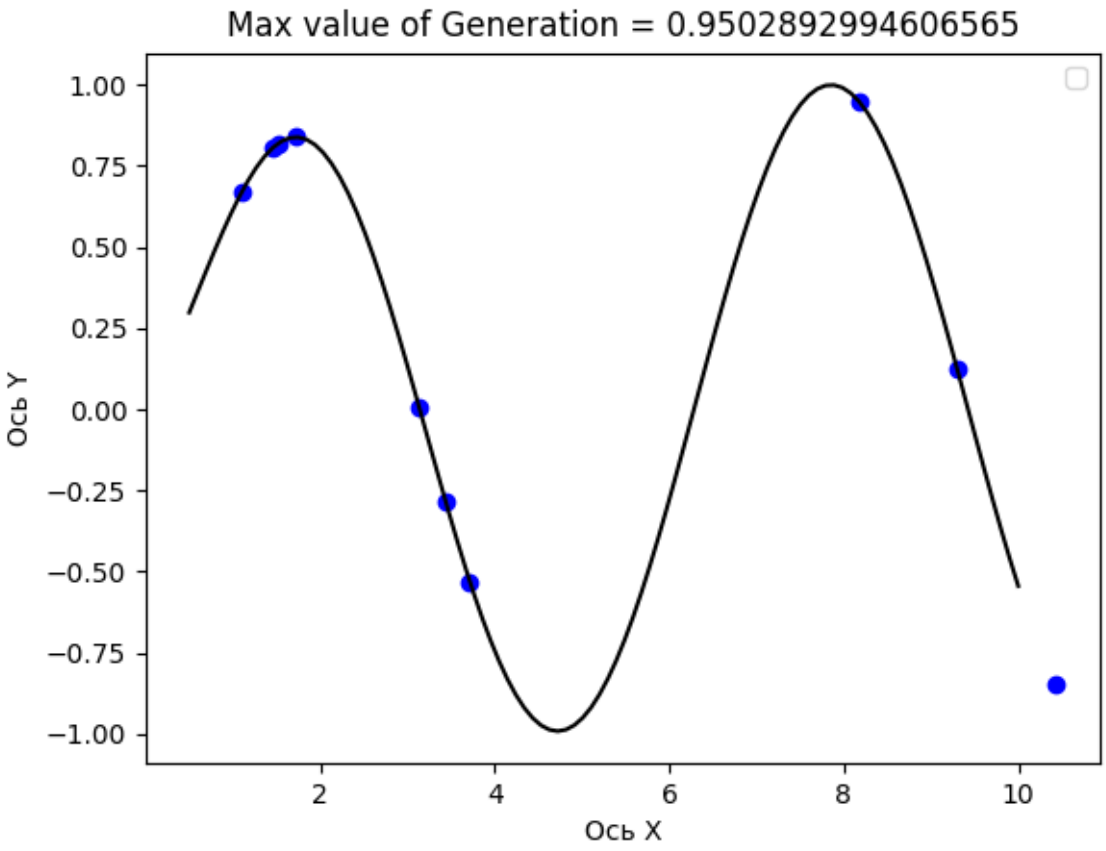

Рис. 2: Нулевая итерация

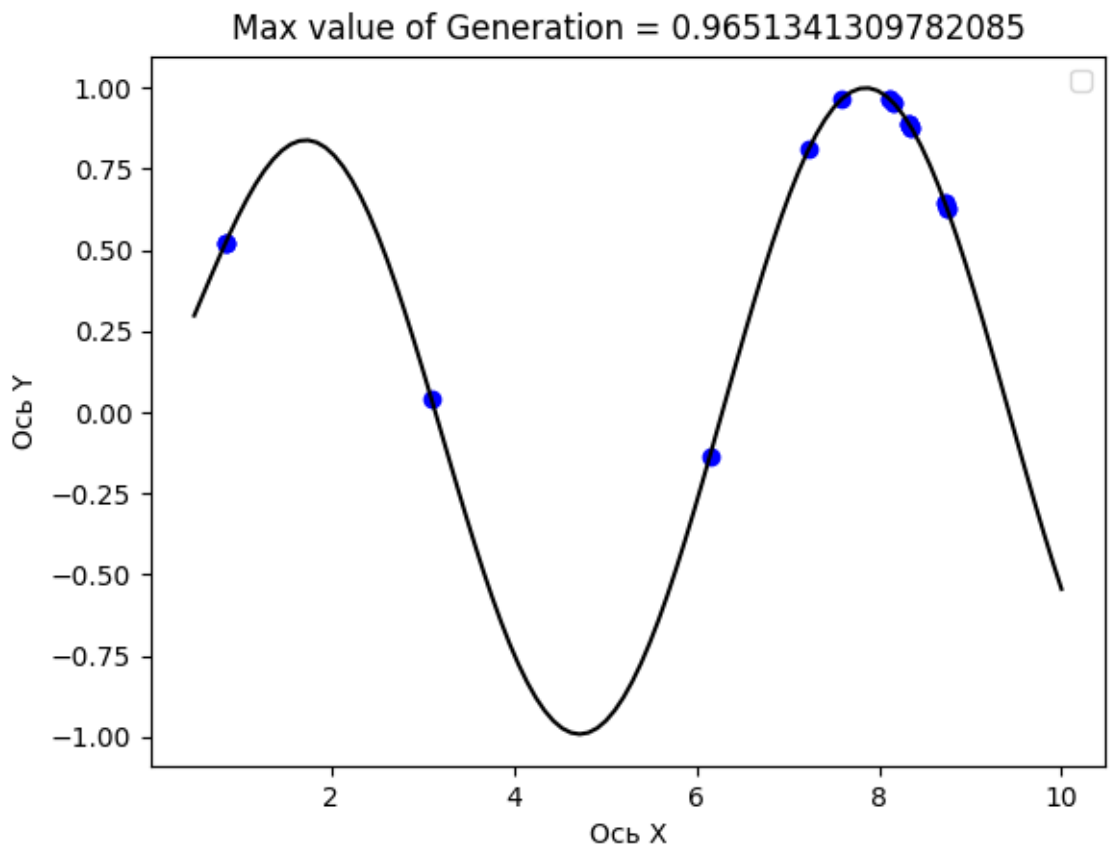

Рис. 3: Первая итерация

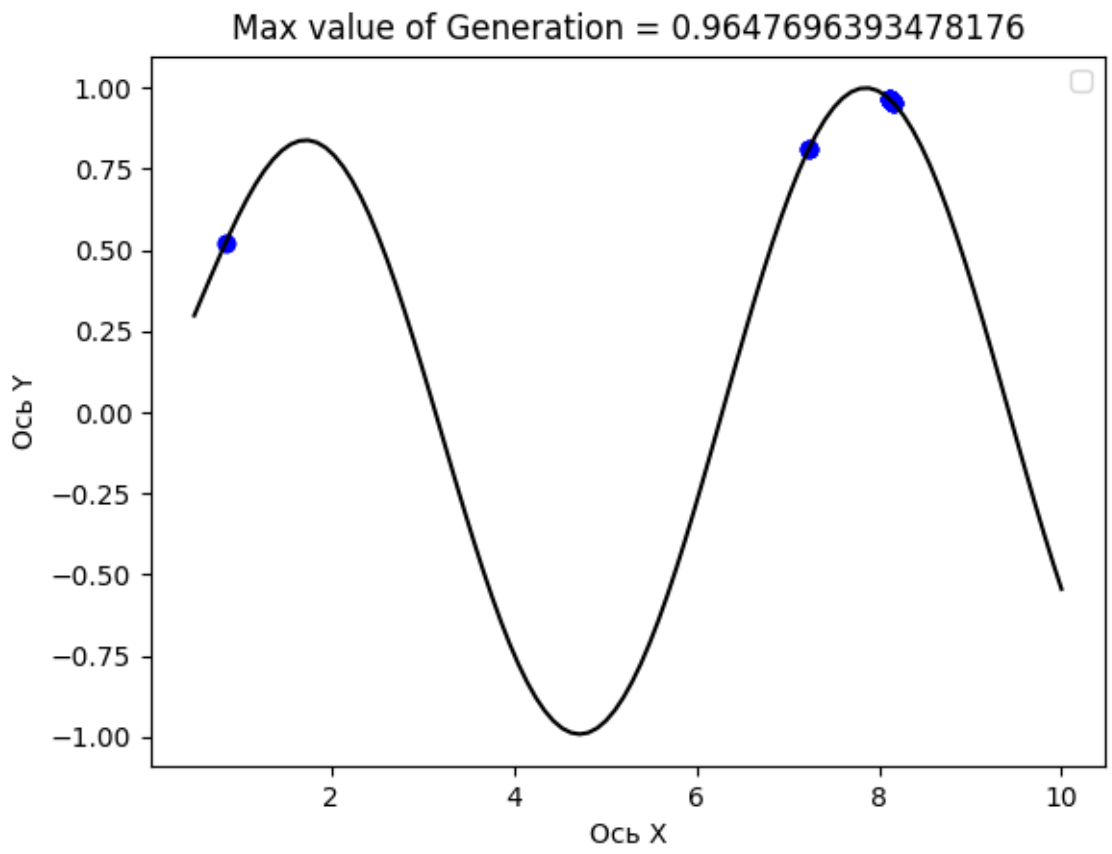

Рис. 4: 5-я итерация

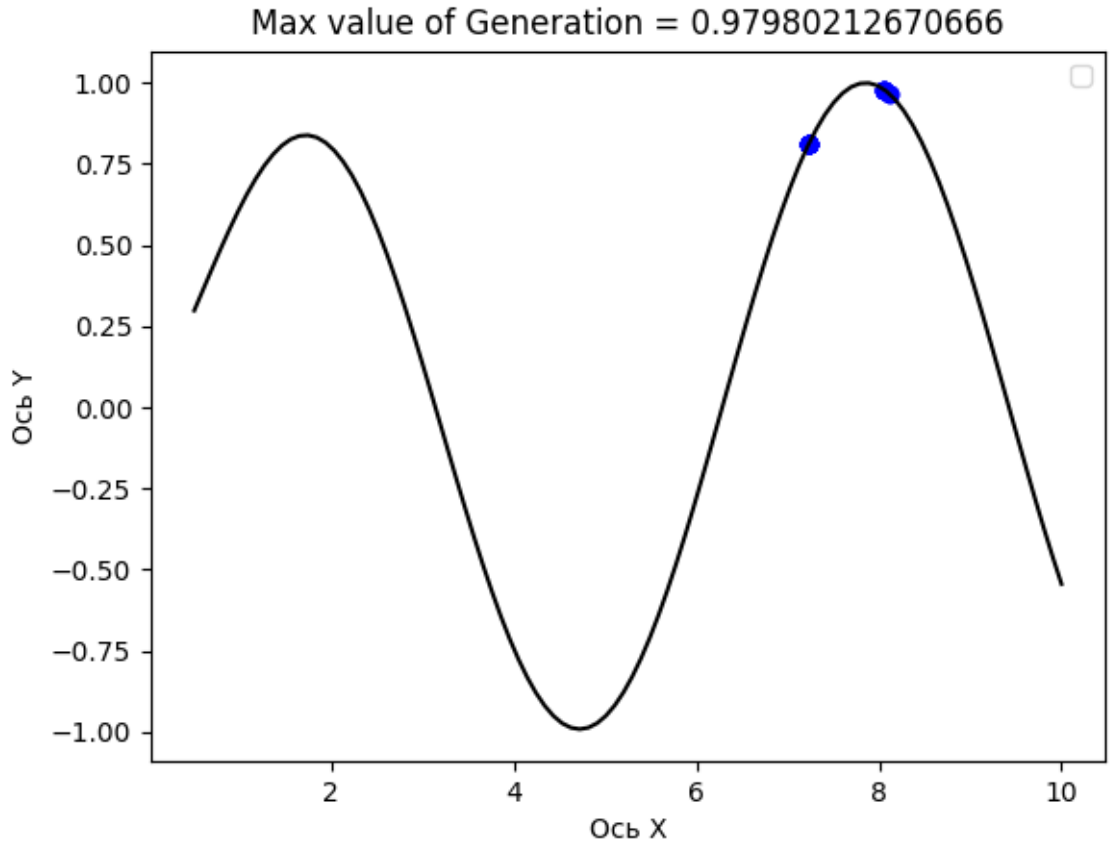

Рис. 5: 15-я итерация

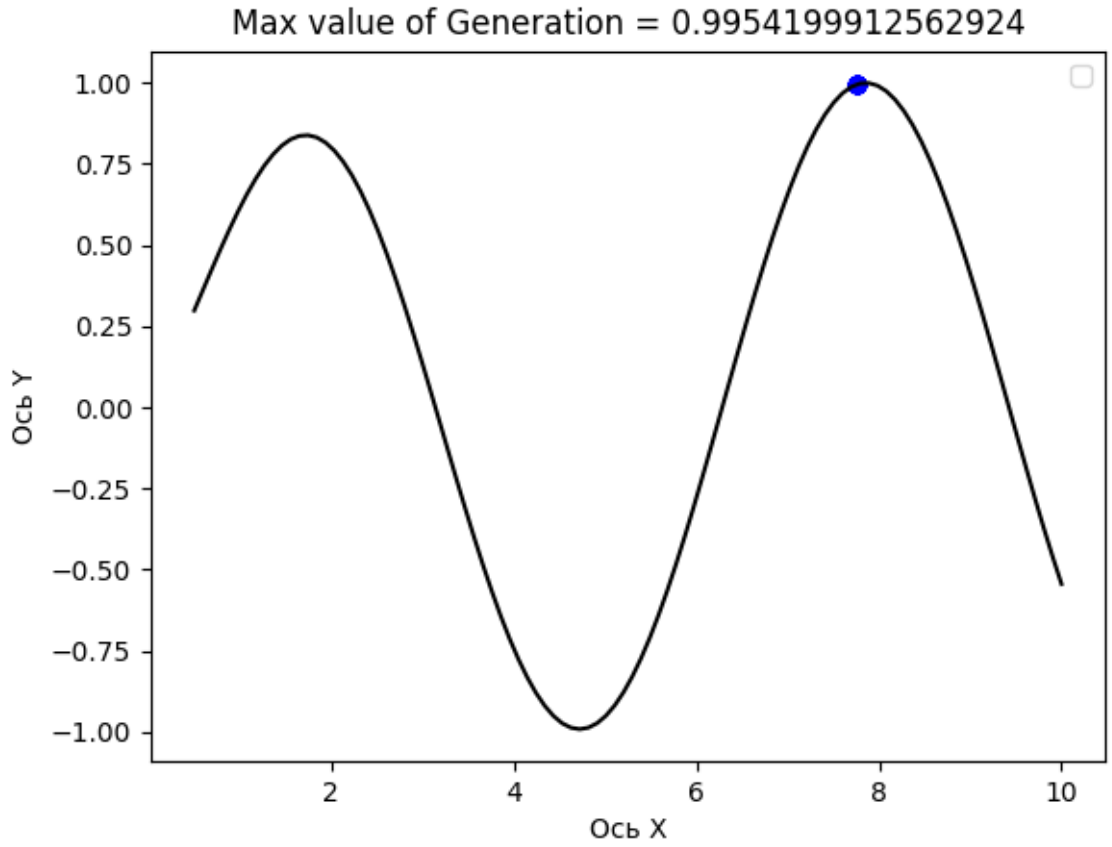

Рис. 6: 50-я итерация

5 Исследования

5.1 Исследования зависимости времени выполнения генерации от мощности популяции и количества итераций

На рис. 7-9 изображены графики зависимости времени выполнения генерации от мощности популяции и количества итераций, очевидно, что при росте количества итераций и мощности популяции время выполенения увеличивается. Также можно заметить, что увеличение мощности популяции сильнее влияет на время выполнения генерации.

Рис. 7: 100 итераций

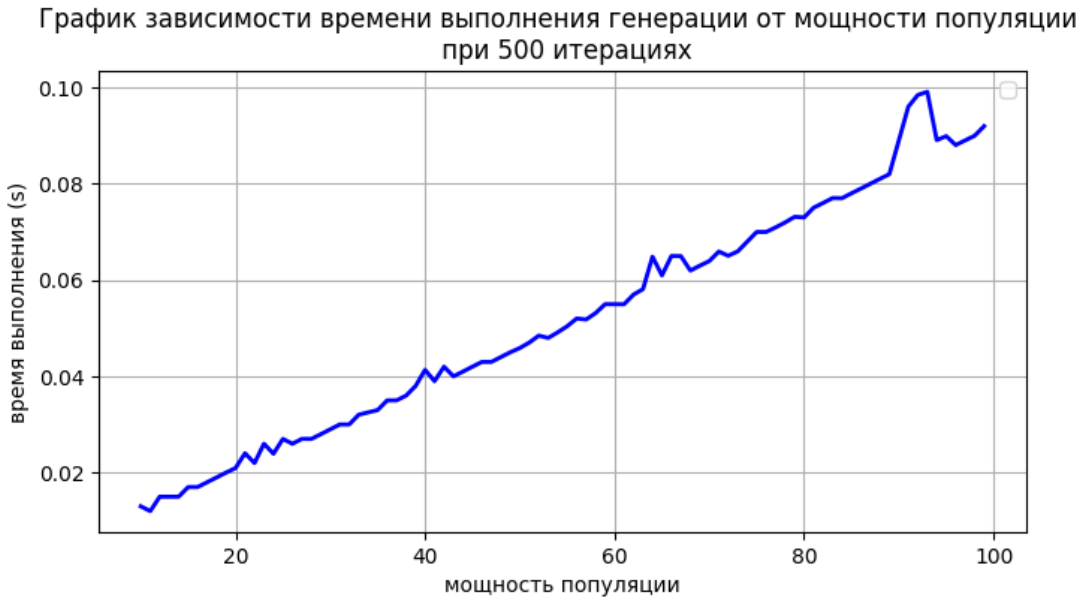

Рис. 8: 500 итераций

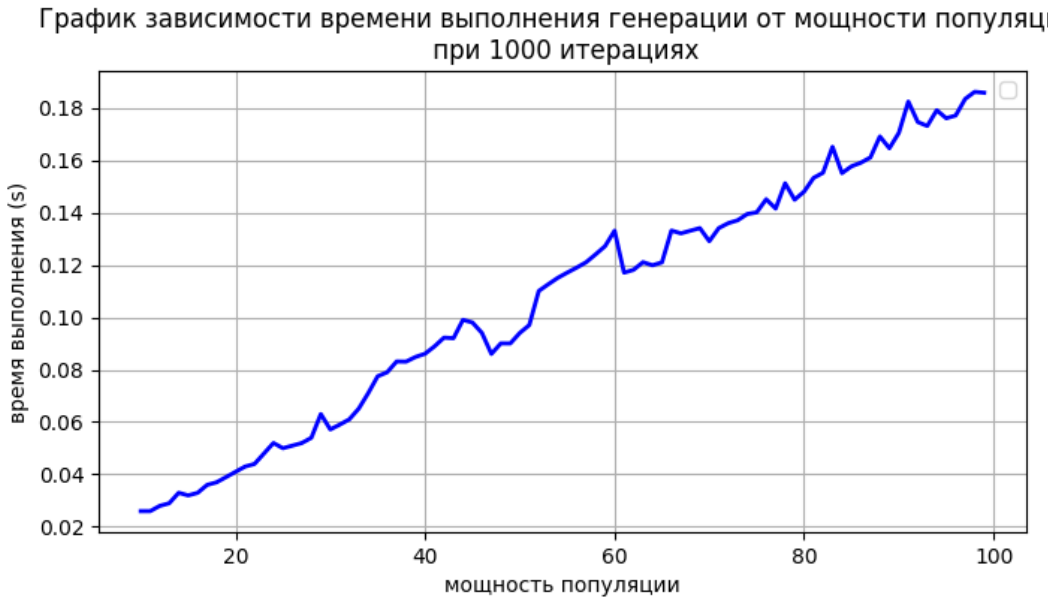

Рис. 9: 1000 итераций

5.2 Исследования зависимости точности результата от мощности популяции и количества итераций

На рис. 10-13 изображены графики зависимости точности результата от мощности популяции и количества итераций. Все вычисления были проделаны с вероятность кроссинговера, равной 0.5 и вероятностью мутации, равной 0.1. Точность результата была посчитана после 1000 запусков алгоритма, после чего было посчитано среднее отклонение от правильного значения. Очевидно, что при росте количества итераций и мощности популяции точность результата повышается. Можно заметить, что увеличение мощности популяции сильнее гораздо сильнее влияет на точность полученного результата. Уже при популяциях с более чем 50 особями алгоритм вычисляет результат с точностью +- 0.04 при любом из представленных количеством итераций.

Рис. 10: 10 итераций

Рис. 11: 50 итераций

Рис. 12: 100 итераций

Рис. 13: 500 итераций

5.3 Исследования зависимости точности результата от вероятности кроссинговера

На рис.14 изображен график зависимости точности результата от вероятности кроссинговера. Все вычисления были проведены при популяции с 50 особями и 100 итерациями. Точность результата была посчитана после 1000 запусков алгоритма, после чего было посчитано среднее отклонение от правильного значения. Исходя из графика, можно сказать, что в данной задаче оператор кроссинговера не влияет на точность результата.

Рис. 14: 500 итераций

5.4 Исследования зависимости точности результата от вероятности мутации

На рис.15 изображен график зависимости точности результата от вероятности мутации. Все вычисления были проведены при популяции с 50 особями и 100 итерациями. Точность результата была посчитана после 1000 запусков алгоритма, после чего было посчитано среднее отклонение от правильного значения. Исходя из графика, можно сказать, что для данной задачи при увеличении вероятности мутации, точность результата возрастает.

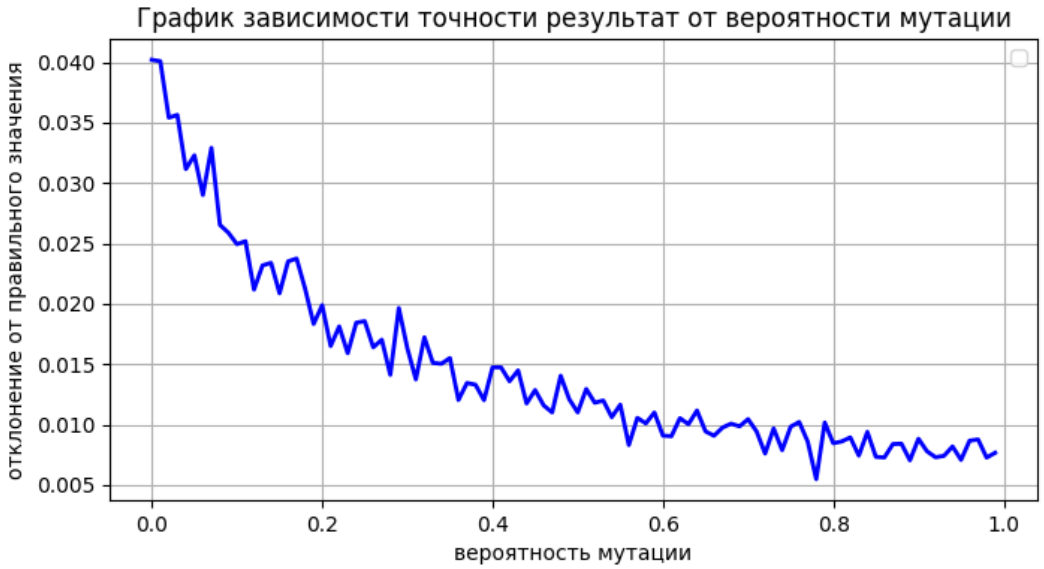

Рис. 15: 500 итераций

6 Контрольный вопрос

Контрольный вопрос: Придумайте другую реализацию ОМ.

Ответ: ОМ может быть реализован методом инверсии. В этом методе случайный участок генома инвертируется. Формальная запись:

G - случайная особь с хромосомой H, N - случайное число, определяющая количество инвертируемых генов, L - начало инвертируемого участка, причем $N + L < |H|$. Участок хромосомы H [L, L + N] инвертируется.

Заключение

В результате выполнения работы разработан простой генетический алгоритм для нахождения максимума функции

$$\frac{sin(x)}{1+e^{-x}}$$

на интервале $x \in [0.5, 10]$.

В отчете представлены результаты работы программы.

Исследована зависимость времени поиска, числа поколений, точности нахождения решения от числа особей в популяции и вероятностей кроссинговера, мутации.

Список литературы

[1] [1] Эволюционные вычисления: Учебное пособие / Ю. А. Скобцов, Д. В. Сперанский— М.: Национальный Открытый Университет «ИНТУИТ» 2012. —331с., ил.

Приложение

Population.py

```
import random

import matplotlib.pyplot as plt
import numpy as np

import time


class Population:

    def __init__(self, power, iteration, p_kross, p_mut, function):
        self.population = []
        self.power = power
        self.iteration = iteration
        self.p_kross = p_kross
        self.p_mut = p_mut
        self.func = function

        # Создание начальной популяции
        for i in range(0, power):
            chromosome = ""
            for j in range(0, 14):
                gen = random.randint(0, 1)
                chromosome += str(gen)
            self.population.append(chromosome)

    def value(self, chromosome):
        decimal_value = int(chromosome, 2)
        return self.func(0.5 + (decimal_value * 10.5) / (2 ** 14 - 1))

    def reproduction(self, sum):
        probabilities = []

        for individual in self.population:  # Формирование вероятностей
            probabilities.append((self.value(individual) + 1) / sum)

        new_population
        = random.choices(self.population,
                         probabilities,
                         k=self.power)  # Вращение колеса

        self.population = new_population

    def krossingover(self, p):
        random_number = random.random()
        if random_number < p:
            # Выбор первой особи
```

```
            individual1 = random.choice(self.population)
            self.population.remove(individual1)

            # Выбор второй особи
            individual2 = random.choice(self.population)
            self.population.remove(individual2)

            k = random.randint(0, 13)

            # Изменение хромосом особей
            for i in range(0, 14):
                if i < k:
                    char_list = list(individual1)
                    char_list[i] = individual2[i]
                    individual1 = "".join(char_list)
                else:
                    char_list = list(individual2)
                    char_list[i] = individual1[i]
                    individual2 = "".join(char_list)

            self.population.append(individual1)
            self.population.append(individual2)

    def mutation(self, p):
        for chromosome in self.population:
            random_number = random.random()
            if random_number < p:
                self.population.remove(chromosome)
                k = random.randint(0, 13)
                if chromosome[k] == "0":
                    char_list = list(chromosome)
                    char_list[k] = "1"
                    chromosome = "".join(char_list)
                else:
                    char_list = list(chromosome)
                    char_list[k] = "0"
                    chromosome = "".join(char_list)
                self.population.append(chromosome)

    def start(self):
        start_time = time.time()
        for i in range(0, self.iteration):
            sum = 0

            for individual in self.population:
                sum += (1 + self.value(individual))

            avg = sum / len(self.population)

            self.reproduction(sum)
```

```
        self.krossingover(self.p_kross)
        self.mutation(self.p_mut)
    end_time = time.time()
    return (max(map(self.value, self.population)))
```