Санкт-Петербургский Национальный Исследовательский
Университет Информационных Технологий, Механики и Оптики

Факультет инфокоммуникационных технологий

Лабораторная работа №2
Создание генетического алгоритма средствами C#

Выполнил
Стафеев И.А.

Группа
K3221

Проверил
Иванов С.Е.

Санкт-Петербург,
2025

СОДЕРЖАНИЕ

Стр.

ВВЕДЕНИЕ

Цель работы: изучить генетический алгоритм и реализовать алгоритм средствами ООП для решения Диофантова уравнения.

Для достижения цели были поставлены следующие задачи:

1. Изучить общую схему устройства генетических алгоритмов;
2. Реализовать класс и присущие ему методы для "особи отождествляющей диофантово уравнение;
3. Реализовать схему генетического алгоритма с учетом особенностей разработанной "особи включая целевую функцию, кроссинговер и мутацию;
4. Посмотреть результат работы при различных параметрах алгоритма, таких как число начальных особей, доля выживших и так далее.

1 Выполнение задания

Постановка задачи:

Генетический алгоритм основан на принципах естественного отбора и применяется для решения задач оптимизации. Решения представляются в виде генотипов — векторов генов, оцениваемых с помощью функции приспособленности. Алгоритм работает итерационно: на каждом шаге создается новое поколение с помощью операций кроссинговера, наследования, мутаций и селекции. Процесс продолжается до достижения оптимального решения или выполнения критерия остановки, например, по числу поколений или времени работы.

В рамках этой лабораторной работы надо разработать генетический алгоритм, позволяющий решить диофантово уравнение $a+2b+3c+4d = 30$;

Решение:

Вначале был создан вспомогательный класс **RandomOperations**, имеющий два метода: один для получения множества целых чисел из диапазона, и второй для получения подмножества целых чисел из диапазона. Эти методы будут применяться при выборе генов для мутации, а также для выбора, какие гены будут наследоваться от первого родителя, а какие - от второго. Код реализации этого класса показан на рисунке 1.

```
3   class RandomOperations
4   {
5       public static Random rnd = new Random();
6       public static HashSet<int> GetRandomFromRange(int min, int max, int count)
7       {
8           var numbers = new HashSet<int>();
9           while (numbers.Count < count)
10          {
11              numbers.Add(rnd.Next(min, max + 1));
12          }
13          return numbers;
14      }
15      public static HashSet<int> GetSetRange(int min, int max)
16      {
17          return new HashSet<int>(Enumerable.Range(min, max - min + 1));
18      }
19
20  }
21
```

Рисунок 1 — Класс RandomOperations

В качестве класса для "особи участвующей в эволюционном процессе, был создан класс **Eq**. Среди атрибутов класса - функция, отождествляющая уравнение, для которого ищется решение; требуемое значение функции; левая и правая граница допустимых значений параметров; максимальное число генов, которые могут мутировать и массив параметров. Поле **cur_result** возвращает значение функции при подстановке начальных параметров, а поле **fitting_f** - это целевая функция, в качестве которой была выбрана относительная ошибка от ожидаемого результата. Атрибуты и конструктор этого класса представлены на рисунке 2.

```
Ссылок: 15
class Eq
{
    public int[] nums;
    public int expected_result;
    public int left_border, right_border, num_of_mutations;
    public Func<int[], int> f;
    private static Random rnd = new Random();
    Ссылок: 1
    public int cur_result
    {
        get { return f(nums); }
    }
    Ссылок: 6
    public double fitting_f
    {
        get { return (double)Math.Abs(cur_result - expected_result) / (double)Math.Abs(expected_result); }
    }
    Ссылок: 2
    public Eq(Func<int[], int> f, int expected_result, int left_border,
                int right_border, int num_of_mutations, int[] nums)
    {
        this.f = f;
        this.expected_result = expected_result;
        this.nums = nums;
        this.left_border = left_border;
        this.right_border = right_border;
        this.num_of_mutations = num_of_mutations;
    }
    Ссылок: 1
```

Рисунок 2 — Атрибуты и конструктор класса Eq

Далее были созданы несколько методов, непосредственно вызываемых во время работы генетического алгоритма. Это метод **Mutate**, который изменяет несколько генов на случайное отклонение и метод **Print** для вывода генотипа (параметров) уравнения. Код приведен на рисунке 3.

```
47      public void Mutate()
48      {
49          var param_nums = RandomOperations.GetRandomFromRange(0, nums.Length - 1, num_of_mutations);
50          foreach (int par_num in param_nums)
51          {
52              int delta = rnd.Next(-2, 3);
53              nums[par_num] = Math.Clamp(nums[par_num] + delta, left_border, right_border);
54          }
55      }
        Ссылок: 1
56      public void Print()
57      {
58          for (int i = 0; i < nums.Length; i++)
59          {
60              Console.Write("x{0}={1}, ", i + 1, nums[i]);
61          }
62          Console.WriteLine();
63      }
```

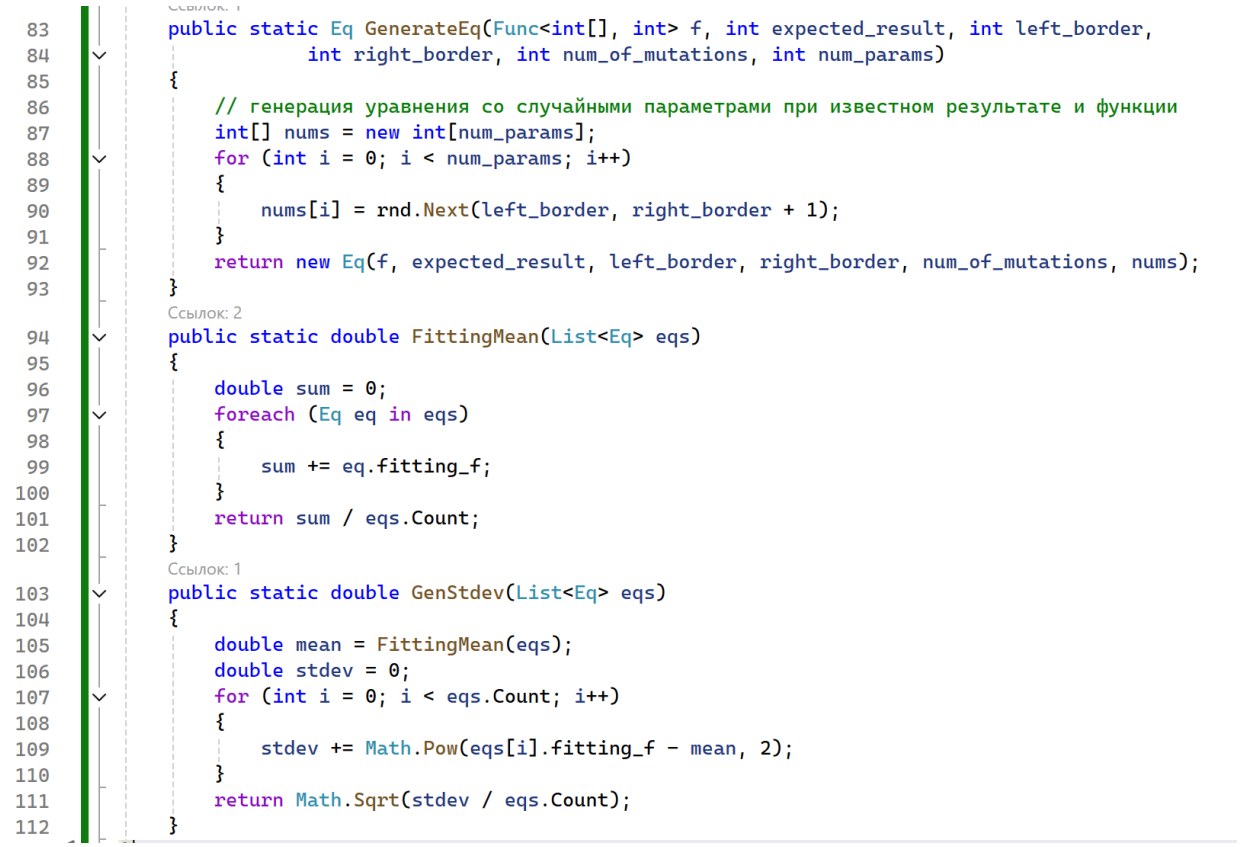
Рисунок 3 — Методы класса Eq

Также был создан оператор сложения, отвечающий за скрещивание двух особей. В результате применения оператора создается новая особь, которая получает часть признаков случайно от первого родителя и часть - от второго. Код методов и оператор приведены на рисунке 4.

```
        Ссылок: 1
64      public static Eq operator +(Eq father, Eq mother)
65      {
66          int n = father.nums.Length;
67          var father_genes = RandomOperations.GetRandomFromRange(0, n - 1, n / 2);
68          var mother_genes = RandomOperations.GetSetRange(0, n - 1);
69          mother_genes.ExceptWith(father_genes);
70          int[] new_genes = new int[n];
71          foreach (int fgen in father_genes)
72          {
73              new_genes[fgen] = father.nums[fgen];
74          }
75          foreach (int mgen in mother_genes)
76          {
77              new_genes[mgen] = mother.nums[mgen];
78          }
79          return new Eq(father.f, father.expected_result, father.left_border, father.right_border,
80              father.num_of_mutations, new_genes);
81
82      }
        Ссылок: 1
```

Рисунок 4 — Оператор сложения для класса Eq

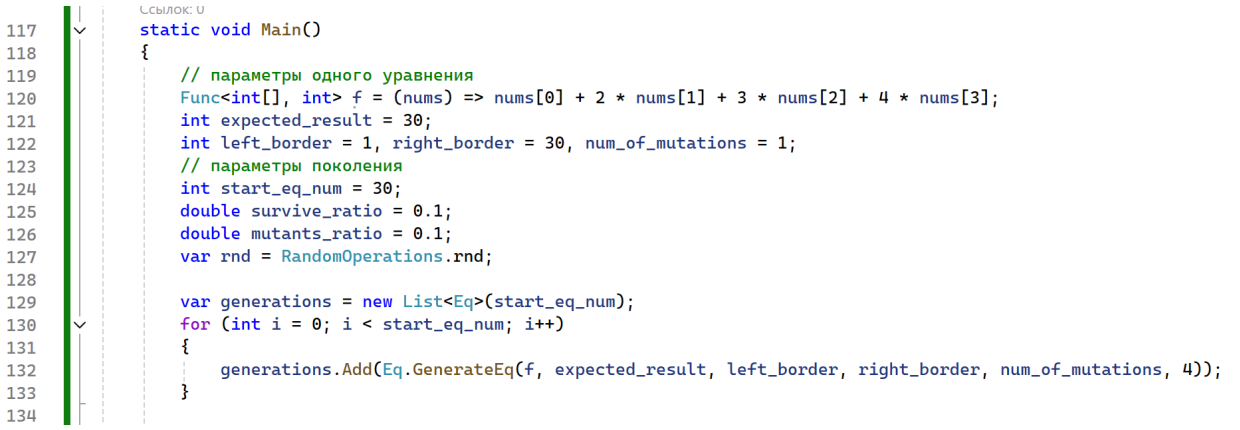
Следом были реализованы статические методы класса **Eq** - для генерации особи со случайными параметрами при заданной функции уравнения и границах допустимых значений, а также для нахождения выборочного среднего и среднеквадратического отклонения для набора однотипных уравнений. Код этих методов показан на рисунке 5.



```
83      public static Eq GenerateEq(Func<int[], int> f, int expected_result, int left_border,
84                  int right_border, int num_of_mutations, int num_params)
85      {
86          // генерация уравнения со случайными параметрами при известном результате и функции
87          int[] nums = new int[num_params];
88          for (int i = 0; i < num_params; i++)
89          {
90              nums[i] = rnd.Next(left_border, right_border + 1);
91          }
92          return new Eq(f, expected_result, left_border, right_border, num_of_mutations, nums);
93      }
        Ссылок: 2
94      public static double FittingMean(List<Eq> eqs)
95      {
96          double sum = 0;
97          foreach (Eq eq in eqs)
98          {
99              sum += eq.fitting_f;
100         }
101         return sum / eqs.Count;
102     }
        Ссылок: 1
103     public static double GenStdev(List<Eq> eqs)
104     {
105         double mean = FittingMean(eqs);
106         double stdev = 0;
107         for (int i = 0; i < eqs.Count; i++)
108         {
109             stdev += Math.Pow(eqs[i].fitting_f - mean, 2);
110         }
111         return Math.Sqrt(stdev / eqs.Count);
112     }
```

Рисунок 5 — Статические методы класса Eq

В методе **Main** были определены начальные условия: функция диофантова уравнения, границы допустимых значений, доля выживающих особей и доля мутирующих. Далее создается выборка из случайных особей. Код показан на рисунке 6.



```
        Ссылок: 0
117     static void Main()
118     {
119         // параметры одного уравнения
120         Func<int[], int> f = (nums) => nums[0] + 2 * nums[1] + 3 * nums[2] + 4 * nums[3];
121         int expected_result = 30;
122         int left_border = 1, right_border = 30, num_of_mutations = 1;
123         // параметры поколения
124         int start_eq_num = 30;
125         double survive_ratio = 0.1;
126         double mutants_ratio = 0.1;
127         var rnd = RandomOperations.rnd;
128
129         var generations = new List<Eq>(start_eq_num);
130         for (int i = 0; i < start_eq_num; i++)
131         {
132             generations.Add(Eq.GenerateEq(f, expected_result, left_border, right_border, num_of_mutations, 4));
133         }
134
```

Рисунок 6 — Начальные условия в методе Main

На рисунке 7 показан сам генетический алгоритм. В бесконечном цикле особи сортируются для выбор лучших, причем в качестве ключа сортиров-

ки выступает сигма-отсечение $p_i = 1 - \frac{f_i - \langle f \rangle}{2\sigma}$ (f - целевая функция), чтобы отсекать выбросные значения и алгоритм сходился медленнее, но без наличия "удобных"случайных особей. Если среди особей есть та, у которой значение целевой функции минимально и равно нулю, то это решение диофантова уравнения. Если таковой особи нет, берутся лучшие особи, а дефицит восполняется скрещиванием случайных особей (не обязательно лучших, чтобы не возник преобладающий генотип). В конце происходит случайная мутация.

```
134
135         for (int i = 0; ; i++)
136         {
137             var cur_mean = Eq.FittingMean(generations);
138             var cur_stdev = Eq.GenStdev(generations);
139             var best_gens = generations.OrderBy(g => 1 + (g.fitting_f - cur_mean) / (2 * cur_stdev)).ToList();
140             var med = best_gens[start_eq_num / 2];
141
142             Console.WriteLine("Generation {0}, FitMin {1}, FitMax {2}, FitMean {3}", i + 1, best_gens[0].fitting_f, best_
143
144             if (best_gens.Any(g => g.fitting_f == 0))
145             {
146                 best_gens[0].Print();
147                 break;
148             }
149
150             var new_generations = new List<Eq>(best_gens.Take((int)(survive_ratio * start_eq_num)));
151             while (new_generations.Count < start_eq_num)
152             {
153                 var father = generations[rnd.Next(start_eq_num)];
154                 var mother = generations[rnd.Next(start_eq_num)];
155                 var child = father + mother;
156                 new_generations.Add(child);
157             }
158             int mutants = (int)(mutants_ratio * start_eq_num);
159             foreach (int eq_num in RandomOperations.GetRandomFromRange(0, start_eq_num - 1, mutants))
160             {
161                 new_generations[eq_num].Mutate();
162             }
163             generations = new_generations;
164         }
165
```

Рисунок 7 — Реализация генетического алгоритма

Примеры работы генетического алгоритма при разных начальных параметрах приведены на рисунках 8, 9 и 10.

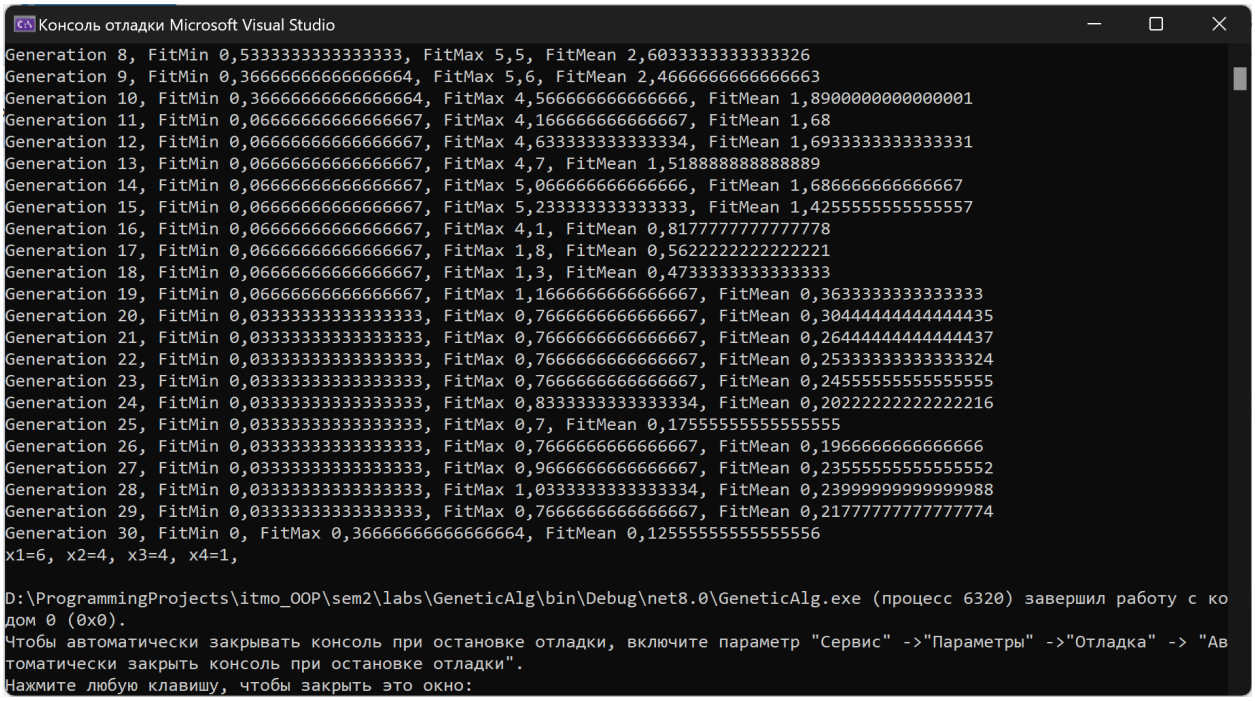


Рисунок 8 — Результат выполнения при 30 особях и 0.1 доле выживших

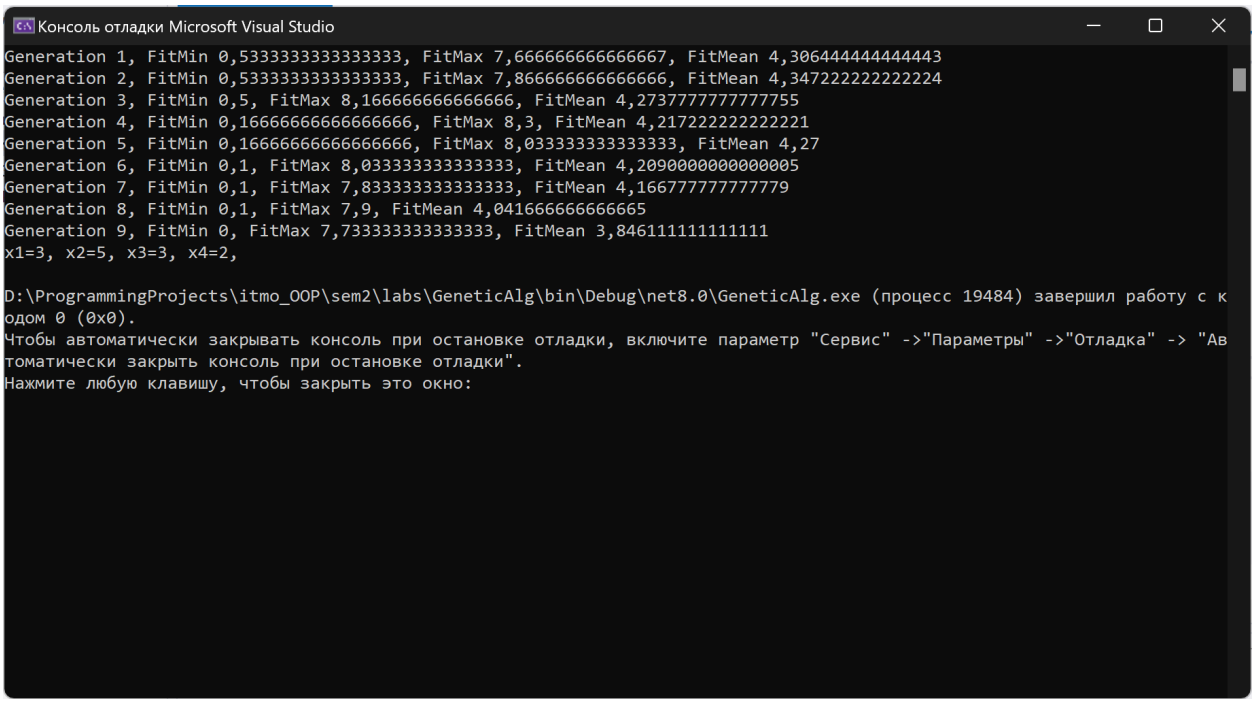


Рисунок 9 — Результат выполнения при 300 особях и 0.01 доле выживших

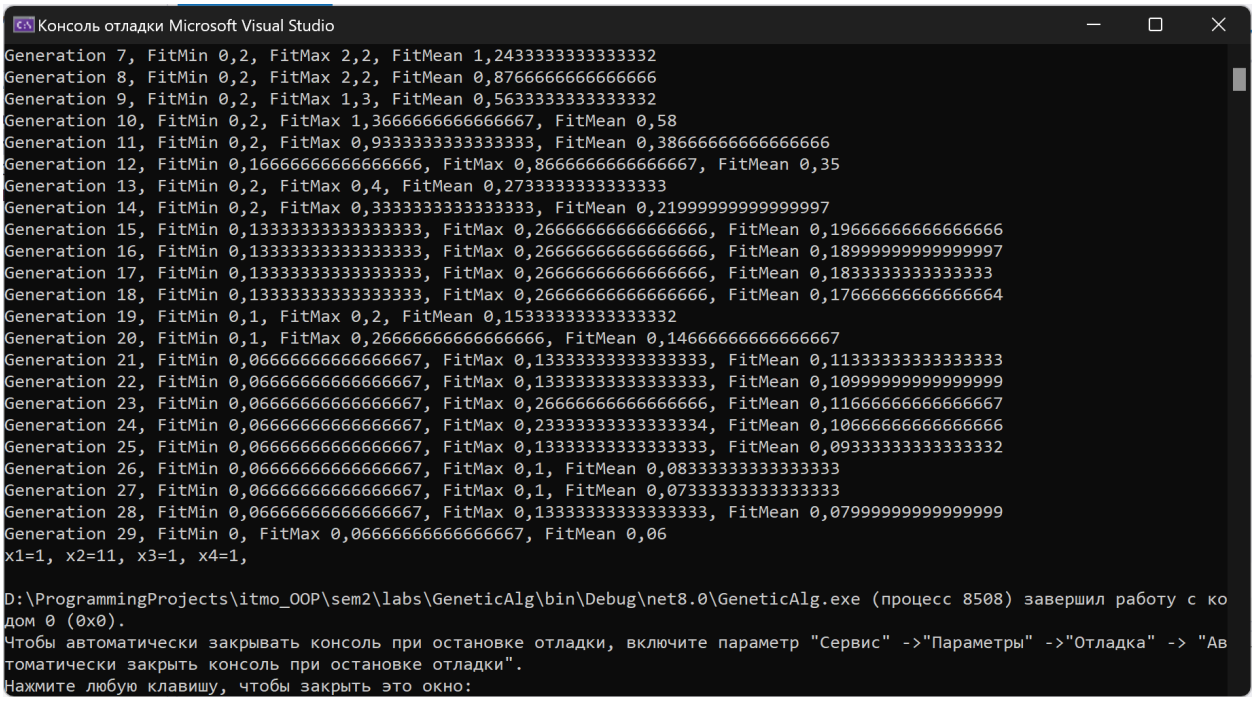


Рисунок 10 — Результат выполнения при 10 особях и 0.5 доле выживших

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были выполнены все задачи, поставленная цель ддостигнута. Было изучено общее строение генетических алгоритмов, и на его основе был создан генетический алгоритм для нахождения решений диофантова уравнения. Полученные знания и навыки заложили основу понимания генетических алгоритмов и разработки и применения их в реальных промышленных задачах.