ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ ИМ. ПРОФ. М.А. БОНЧ-БРУЕВИЧА» (СПбГУТ)

ОТЧЕТ
О ПРОДЕЛАННОЙ РАБОТЕ НА ФАКУЛЬТАТИВЕ
«Разработка объектно-ориентированного программного обеспечения автоматизированных систем на языке C#»

Разработка программы, реализующей генетический алгоритм для поиска минимума функций

Выполнила:
Студентка гр. ИКПИ-93
Колтунова Е.В.

Научный руководитель:
доцент кафедры АПС,
к.т.н. Акимов С.В.

Санкт-Петербург
2019

Содержание

Введение

Целью проекта, реализуемого в рамках факультатива, является разработка программы, реализующей генетический алгоритм для поиска минимума функций.

Генетические алгоритмы — это адаптивные методы поиска, которые в последнее время используются для решения задач оптимизации. В них используются как аналог механизма генетического наследования, так и аналог естественного отбора. При этом сохраняется биологическая терминология в упрощенном виде и основные понятия линейной алгебры.

Постановка задачи

В рамках данного факультатива создаются программные классы, обеспечивающие создание популяции, скрещивание особей, мутацию и отбор наиболее приспособленных особей.

Основные понятия

Введем основные понятия, применяемые в генетических алгоритмах.

Организм(особь) — строка из каких-либо чисел. В нашей работе этот вектор представлен бинарной строкой из нулей и единиц. Каждая позиция (бит) хромосомы называется геном.

Кроссинговер(кроссовер) — операция, при которой две хромосомы обмениваются своими частями.

Мутация— случайное изменение одной или нескольких позиций в хромосоме.

Популяция— совокупность организмов.

Пригодность(приспособленность) — критерий или функция, минимум которой следует найти.

Размер особи — длина вектора; количество битов, из которых состоит особь.

Вероятность мутации — вероятность, с которой происходит мутация особи.

Критерий окончания процесса - схождение популяции.

Схождение - такое состояние популяции, когда все особи популяции почти одинаковы и находятся в области некоторого экстремума. В такой ситуации кроссинговер практически никак не изменяет популяции, так как создаваемые при нем потомки представляют собой копии родителей с перемененными участками хромосом. Таким образом, схождение популяции обычно означает, что найдено лучшее или близкое к минимуму решение.

Основные принципы работы ГА

1. Генерируем начальную популяцию из заданного количества организмов.
2. Проводим поочередно кроссинговер для всех особей из популяции.
3. Проводим мутацию потомков (если вероятность мутации < 0.99).
4. Добавляем потомков в нашу популяцию.
5. Сортируем организмы в популяции в соответствии с условием задачи.
6. Оставляем в полученной популяции только первые n особей.(Количество особей в полученной популяции равно заданному количеству особей в изначальной популяции).
7. Повторяем шаги 2-6 заданное количество раз.

Схема генетического алгоритма

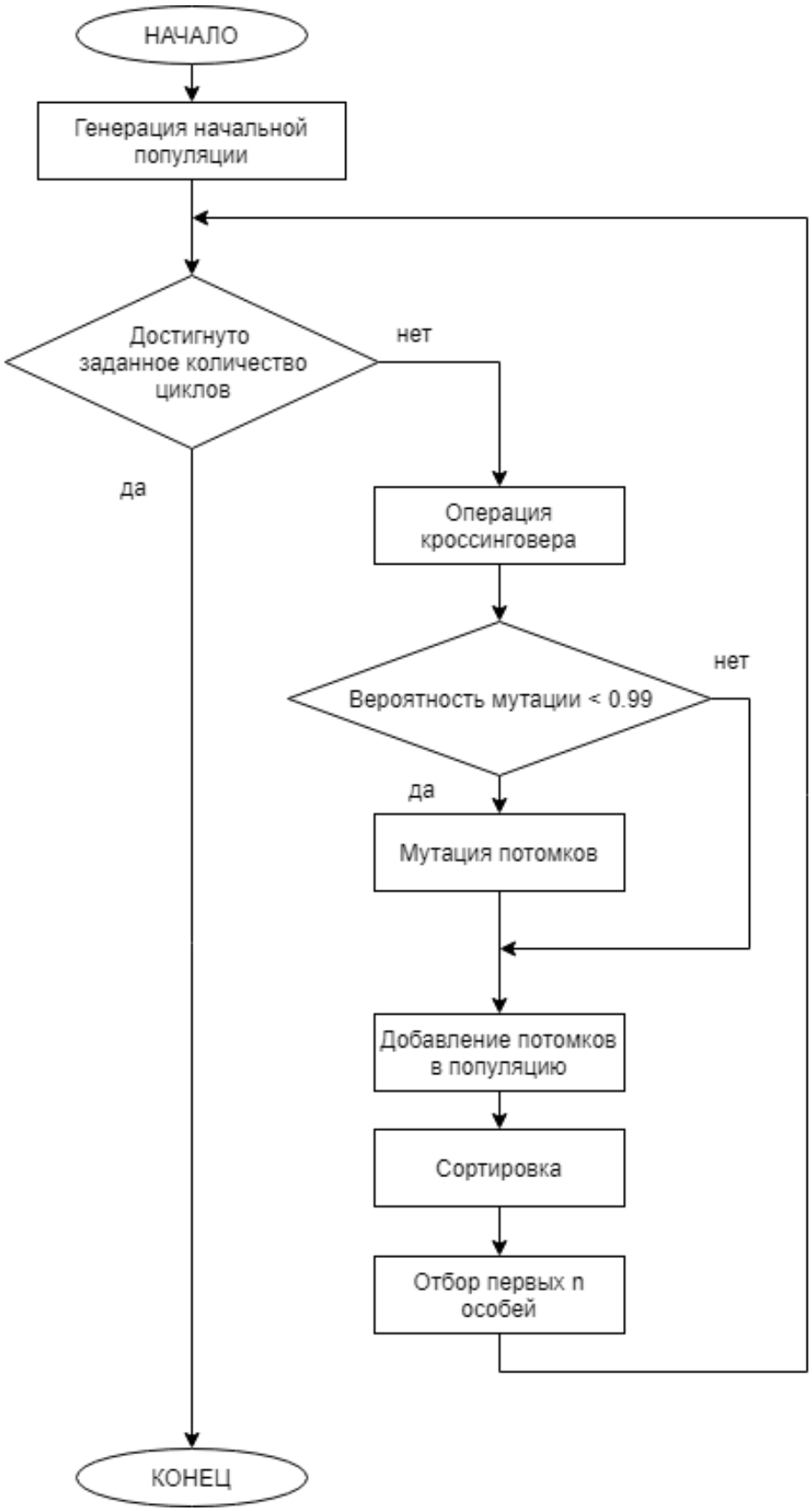

Рисунок 1. Схема генетического алгоритма

Основные операторы ГА

1. Создание нового организма(CreateChromosome)

Задаем размер организма*(size)*: желаемое количество битов в организме + 1(дополнительный бит для хранения знака числа).

Создаем список, который рандомно заполняем 0 и 1 от 0 до size — получаем новый организм.

```
public class Organism
   {
      private static Random _rnd = new Random();

      public const int size = 33;

      public Organism()
      {
         chromosome = new List<int>();
         CreateChromosome();
      }

      public Organism(List<int> list)
      {
         this.chromosome = list;
      }

      public List<int> chromosome { get; set; }

      public void CreateChromosome()
      {
         for (int n = 0; n < size; n++)
         {
            chromosome.Add(_rnd.Next(2));
         }
      }
…
}
```

2. Создание начальной популяции(CreateNewPopulation)

Задается количество организмов в популяции(*initial_count*). Далее создается это количество организмов и добавляется в список популяции.

```
namespace genetic_algorithm
{
   public class GenerationKeeper
   {

      ...

      public List<Organism> population { get; private set; }

      public void CreateNewPopulation(int initial_count)
      {
         this.initial_count = initial_count;

         population = new List<Organism>();
         for (int n = 0; n < initial_count; n++)
         {
            population.Add(new Organism());
         }
      }

        ...

   }
}
```

3. Скрещивание(Crossingover)

В данной программе используется одноточечный кроссинговер, при котором родительские организмы разделяются только в одной случайной точке. Берутся две родительские особи. Рандомным образом определяется точка разрыва, в которой обе особи делятся на две части и обмениваются ими.

```
namespace genetic_algorithm
{
   public class GenerationKeeper
   {

      private static Random _rnd = new Random();
      private int initial_count;

      public List<Organism> Crossingover(Organism x, Organism y)
      {
         //dlya randomnoj tochki razriva
         int a = _rnd.Next(1, Organism.size - 1);
         int b = Organism.size - a;

         Organism x0y1 = new Organism(x.chromosome.GetRange(0,
a).Concat(y.chromosome.GetRange(a, b)).ToList());

         Organism y0x1 = new Organism(y.chromosome.GetRange(0,
a).Concat(x.chromosome.GetRange(a, b)).ToList());

         return new List<Organism>(){ x0y1, y0x1}; //vozvrashaet dvuh potomkov
      }

      …

   }
}
```

4. Мутация(Mutation)

Рандомным образом выбирается значение вероятности мутации от 0 до 1(probability). Если probability < 0.99, то выполняется операция мутации. При мутации случайным образом выбирается ген(бит), который будет мутировать. Значение выбранный гена заменяется на противоположное(с 0 на 1 или с 1 на 0).

```
namespace genetic_algorithm
{
   public class Organism
   {
      private static Random _rnd = new Random();  // dlya randomnogo vibora bita
      public const int size = 33;
        public void Mutate()      // metod mutacii
      {
         double probability = _rnd.NextDouble();

         if (probability > 0.99)
         {
            return;
         }

         int index = _rnd.Next(size);   //8 - tak kak 8 bit

         if (chromosome[index] == 0)
         {
            chromosome[index] = 1;
         }
         else
         {
            chromosome[index] = 0;
         }
      }

     …
   }
}
```

5. Генерация новой популяции(Generation)

В существующей популяции поочередно скрещиваются все особи. В зависимости от вероятности мутации получившиеся потомки мутируют или нет. Далее потомки добавляются в исходную популяцию.

```
namespace genetic_algorithm
{
   public class GenerationKeeper
   {
        public void Generation()
       {
          var copy = population.ToArray<Organism>();

          for (int parent1_index = 0; parent1_index < copy.Length - 1; parent1_index++)
          {
             for (int parent2_index = parent1_index + 1 ; parent2_index < copy.Length;
parent2_index++)
             {
                var childs = Crossingover(copy[parent1_index], copy[parent2_index]);

                foreach (var child in childs)
                {
                   child.Mutate();
                   population.Add(child);
                }
             }
          }

      …
    }

}
```

6. Сортировка и отбор(Selection)

Сортируем организмы в популяции согласно указанному критерию.

При использовании метода сортировки Sort в начале списка содержатся наименьшие элементы, в конце — наибольшие. После сортировки оставляем то же количество особей, которое было в начальной популяции.

```
namespace genetic_algorithm
{
    public class MinimumComparer : IComparer<Organism>
    {
        private Func<double, double> function;

        public MinimumComparer(Func<double, double> function)
        {
            this.function = function;
        }

        public static IComparer<Organism> Create(Func<double, double> function)
        {
            return new MinimumComparer(function);
        }

        public int Compare(Organism x, Organism y)
        {
            var value1 = function(x.Value);
            var value2 = function(y.Value);
            return value1 > value2 ? 1 : value2 > value1 ? -1 : 0;
        }
    }
    public class GenerationKeeper
    {
            public void Selection(IComparer<Organism> comparer)
        {
            population.Sort(comparer);

            population = population.GetRange(0, initial_count);
        }
       …
    }

}
```

Код программы

Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;



namespace genetic_algorithm
{

    public class MinimumComparer : IComparer<Organism>
    {
        private Func<double, double> function;

        public MinimumComparer(Func<double, double> function)
        {
            this.function = function;
        }

        public static IComparer<Organism> Create(Func<double, double> function)
        {
            return new MinimumComparer(function);
        }

        public int Compare(Organism x, Organism y)
        {
            var value1 = function(x.Value);
            var value2 = function(y.Value);
            return value1 > value2 ? 1 : value2 > value1 ? -1 : 0;
        }
    }

    class Program
    {

        static void Main(string[] args)
        {
            Console.WriteLine($"ArgMin((x + 1)^2) = {FindMinimum((x) => (x + 1) * (x +
1))}");

            Console.WriteLine($"ArgMin((x - 1)^2) = {FindMinimum((x) => (x - 1) * (x -
1))}");

            Console.WriteLine($"ArgMin(x^2) = {FindMinimum((x) => x * x)}");
        }
```

```
        static double FindMinimum(Func<double, double> func) {
            var comparer = new MinimumComparer(func);
            GenerationKeeper keeper = new GenerationKeeper();

            keeper.CreateNewPopulation(10);

            for (var n = 0; n < 20; n++)
            {
                keeper.Generation();
                keeper.Selection(comparer);
                Console.WriteLine($"[{n}]: {keeper.population[0].Value}
({keeper.population.Count})");
            }

            return keeper.population[0].Value;
        }
    }
}
```

Organism.cs

```
Using System;
using System.Collections.Generic;

namespace genetic_algorithm
{
    public class Organism
    {
        private static Random _rnd = new Random();
        public const int size = 33;

        public Organism()
        {
            chromosome = new List<int>();
            CreateChromosome();
        }


        public Organism(List<int> list)
        {
            this.chromosome = list;
        }


        public List<int> chromosome { get; set; }


        public void CreateChromosome()
        {
            for (int n = 0; n < size; n++)
            {
                chromosome.Add(_rnd.Next(2));
            }
        }

        public void Mutate()      // metod mutacii
        {
            double probability = _rnd.NextDouble();

            if (probability > 0.99)
            {
                return;
            }

            int index = _rnd.Next(size);

            if (chromosome[index] == 0)
            {
                chromosome[index] = 1;
            }
            else
            {
                chromosome[index] = 0;
```

```
            }

        }


        public override string ToString()
        {
            string s = "";

            foreach (int item in chromosome)
            {
                s += item;
            }

            return $"{Value}[{s}]";
        }


        public double Value
        {
            get
            {
                double n = 0;

                for (int i = 0, j = size - 2; i < size - 2 || j >= 0; i++, j--)
                {
                    n += chromosome[i] * Math.Pow(2, j);
                }

                return n * (chromosome[size - 1] == 0 ? 1.0 : -1.0);
            }
        }
    }
}
```

GenerationKeeper.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace genetic_algorithm
{
    public class GenerationKeeper
    {
        private static Random _rnd = new Random();
        private int initial_count;

        //vozvrashaet potomkov
        public List<Organism> Crossingover(Organism x, Organism y)
        {
            //dlya randomnoj tochki razriva
            int a = _rnd.Next(1, Organism.size - 1);
            int b = Organism.size – a;

            Organism x0y1 = new Organism(x.chromosome.GetRange(0,
a).Concat(y.chromosome.GetRange(a, b)).ToList());

            Organism y0x1 = new Organism(y.chromosome.GetRange(0,
a).Concat(x.chromosome.GetRange(a, b)).ToList());

            return new List<Organism>(){ x0y1, y0x1};
        }

        public List<Organism> population { get; private set; }

        public void CreateNewPopulation(int initial_count)
        {
            this.initial_count = initial_count;

            population = new List<Organism>();
            for (int n = 0; n < initial_count; n++)
            {
                population.Add(new Organism());
            }
        }

        public void Generation()
        {
            var copy = population.ToArray<Organism>();
```

```
            for (int parent1_index = 0; parent1_index < copy.Length - 1; parent1_index+
+)
            {
                for (int parent2_index = parent1_index + 1 ; parent2_index <
copy.Length; parent2_index++)
                {
                    var childs = Crossingover(copy[parent1_index],
copy[parent2_index]);

                    foreach (var child in childs)
                    {
                        child.Mutate();
                        population.Add(child);
                    }
                }
            }
        }

        public void Selection(IComparer<Organism> comparer)
        {
            population.Sort(comparer);

            population = population.GetRange(0, initial_count);
        }

        public override string ToString()
        {
            var ret = "";
            foreach(var organism in population)
            {
                ret += $"{organism} ";
            }
            return ret;
        }
    }
}
```

Пример результатов программы

[Номер цикла]: значение минимума (кол-во особей в популяции)

```
[0]: -138412451 (10)
[1]: -4425117 (10)
[2]: -326 (10)
[3]: -64 (10)
[4]: 56 (10)
[5]: -6 (10)
[6]: -4 (10)
[7]: -4 (10)
[8]: 0 (10)
[9]: -1 (10)
[10]: -1 (10)
[11]: -1 (10)
[12]: -1 (10)
[13]: -1 (10)
[14]: -1 (10)
[15]: -1 (10)
[16]: -1 (10)
[17]: -1 (10)
[18]: -1 (10)
[19]: -1 (10)
ArgMin((x + 1)^2) = -1
```

```
[0]: -34132047 (10)
[1]: -34099240 (10)
[2]: -544808 (10)
[3]: -20776 (10)
[4]: -4392 (10)
[5]: -4128 (10)
[6]: -32 (10)
[7]: -32 (10)
[8]: 8 (10)
[9]: 0 (10)
[10]: 1 (10)
[11]: 1 (10)
[12]: 1 (10)
[13]: 1 (10)
[14]: 1 (10)
[15]: 1 (10)
[16]: 1 (10)
[17]: 1 (10)
[18]: 1 (10)
[19]: 1 (10)
ArgMin((x - 1)^2) = 1
```

```
[0]: -125247695 (10)
[1]: 63964431 (10)
[2]: -30677382 (10)
[3]: 13632783 (10)
[4]: 5243114 (10)
[5]: 5243114 (10)
[6]: 1573135 (10)
[7]: -1048810 (10)
[8]: 524559 (10)
[9]: 236 (10)
[10]: 15 (10)
[11]: 11 (10)
[12]: -6 (10)
[13]: -4 (10)
[14]: 0 (10)
[15]: 0 (10)
[16]: 0 (10)
[17]: 0 (10)
[18]: 0 (10)
[19]: 0 (10)
ArgMin(x^2) = 0
```

Заключение

В процессе проделанной работы на факультативе была реализована программа для поиска минимума функции.

Используемые источники

1. Батищев, Д.И. Генетические алгоритмы решения экстремальных задач [Текст]/ Д.И. Батищев ; Нижегородский госуниверситет. — Нижний Новгород : 1995.с. — 62с
2. Панченко, Т. В. Генетические алгоритмы [Текст] : учебно-методическое пособие / под ред. Ю. Ю. Тарасевича. — Астрахань : Издательский дом «Астраханский университет», 2007. — 87 [3] с.