**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе №2**

**по дисциплине «Основы искусственного интеллекта»**

Тема: Генетический алгоритм

|  |  |  |
| --- | --- | --- |
| Студенты гр. 9302 |  | Ширнин К.В. |
|  |  | Квитко Д.В. |
| Преподаватель |  | Новакова Н.Е. |

Санкт-Петербург

2022

**Оглавление**

[Цель работы 3](#_Toc120489052)

[Реализация алгоритма 3](#_Toc120489053)

[UML Диаграммы 4](#_Toc120489054)

[Ход работы 5](#_Toc120489055)

[Пример работы программы 7](#_Toc120489056)

[Вывод 8](#_Toc120489057)

[Список используемой литературы 8](#_Toc120489058)

[Листинг 9](#_Toc120489059)

# Цель работы

Изучить принцип работы генетического алгоритма. Написать ПО с пользовательским графическим интерфейсом, находящее максимум заданной функции, с использованием генетического алгоритма

# Реализация алгоритма

Для реализации генетического алгоритма ПО было разделено на две составляющие, а именно на GUI и сам программный код алгоритма. Программный комплекс был написан в рамках правил ООП и без особых трудностей может использоваться как отдельный виджет в других ПО или как библиотека.

В начале алгоритм генерирует популяцию, размер которых задан пользователем. Популяция состоит из хромосом, размер которых фиксирован и равен двум, так как для данной задачи этого будет более чем достаточно – в дальнейшем будем называть это “гены”. Хромосомы в одном своём гене будут хранить *x,* а в другом *y*. При инициализации алгоритма их значения заполняются случайным образом.

После генерации происходит оценка поколения на “выживаемость”. В нашем случае берется каждая хромосома со своими параметрами *x*’ов и *y*’ов и подставляется в заданную изначально функцию. Чем больше будет результат – тем лучше. (Так как наша задача найти максимум).  
 Далее происходит генерация следующего поколения. Хромосома, давшая лучший результат, переходит в следующее поколение без изменений. Остальные хромосомы начинают скрещиваться с заданной пользователем вероятностью. Скрещивание происходит случайным образом между хромосомами – хромосоме из следующего поколения может достаться случайный *x* ген и случайный *y* от текущего поколения.

После этого происходит мутация хромосом с заданной пользователем вероятностью. Мутация позволяет выйти алгоритму из зацикливаний – она разбавляет поколения новыми генами.   
 Вышеописанный алгоритм повторяется столько раз, сколько задал пользователь параметром “количество поколений”.

# UML Диаграммы

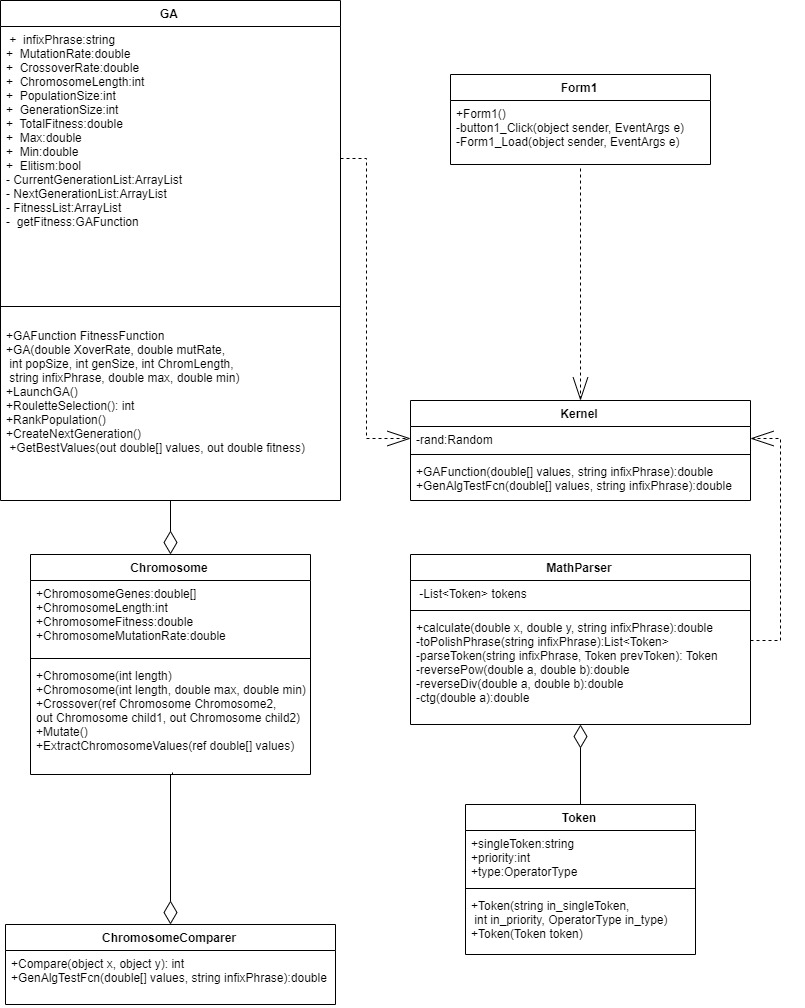


Рис.1. UML диаграмма

Таблица.1.Методы и их описание

|  |  |
| --- | --- |
| Метод | Описание |
| public GA(bool isBit, double XoverRate, double mutRate, int popSize, int genSize, int ChromLength, string infixPhrase, double max, double min) | Конструктор класса GA |
| public void LaunchGA() | Запуск генетического алгоритма |
| private int RouletteSelection() | Метод которые выполняет рулеточную селекцию |
| private void RankPopulation() | Высчитывает значение приспособленности |
| private void CreateNextGeneration() | Запускает одну итерацию цикла |
| public void GetBestValues(out double[] values, out double fitness) | Получает наибольшее значение приспособленности |
| public Chromosome(int length) | Конструктор класса Chromosome |
| public Chromosome(int length, double max, double min) | Конструктор класса Chromosome |
| public void Crossover(ref Chromosome Chromosome2, out Chromosome child1, out Chromosome child2) | Скрещивание хромосом |
| public void Mutate() | Мутация хромосом |
| public int Compare(object x, object y) | Сравнивание приспособленности хромосом |
| public Token(string in\_singleToken, int in\_priority, OperatorType in\_type) | Конструктор класса Token |
| public Token(Token token) | Конструктор класса Token |
| public static double calculate(double x, double y, string infixPhrase) | Вычисление значения заданной функции |
| private static List<Token> toPolishPhrase(string infixPhrase) | Переводит функцию в польскую нотацию |
| private static Token parseToken(string infixPhrase, Token prevToken) | Считывает токен из инфиксного математического выражения |
| private static double reversePow(double a, double b) | Возведение в квадрат |
| private static double reverseDiv(double a, double b) | Деление числа |
| private static double ctg(double a) | Вычисление ctg |

# Ход работы

Для выполнения данной лабораторной работы был выбран язык C#.

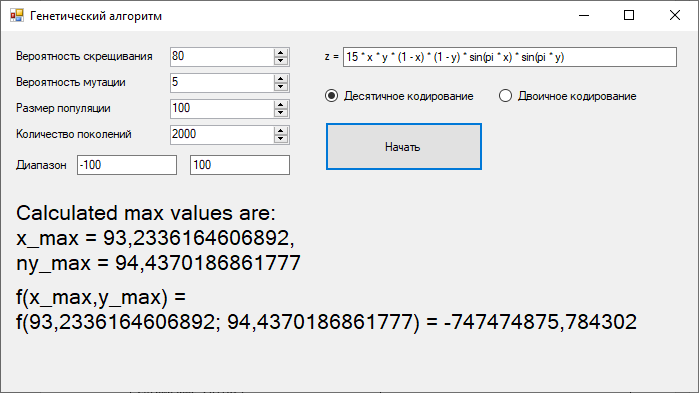


Рис.2. Главное окно приложения

При выполнении работы, были взяты основные используемые параметры:

* Вероятность скрещивания (0-100)
* Вероятность мутации (0-100)
* Размер популяции (3-5000)
* Количество поколений (1-5000)

В качестве решаемой задачи была взята функция

Для задачи параметров используются элементы WinForms “NumericUpDown”, что позволяет избежать ошибок с некорректным вводом данных.

Пользователю достаточно заполнить все предложенные поля и нажать на кнопку “Начать”, после чего программа выдаст рассчитанные значения для *x* и *y*, при выбранных параметрах.

При выполнении работы также были учтены крайние значения, вводимые пользователем в программу.

# Пример работы программы

Пример работы программы представлен на рисунках ниже.

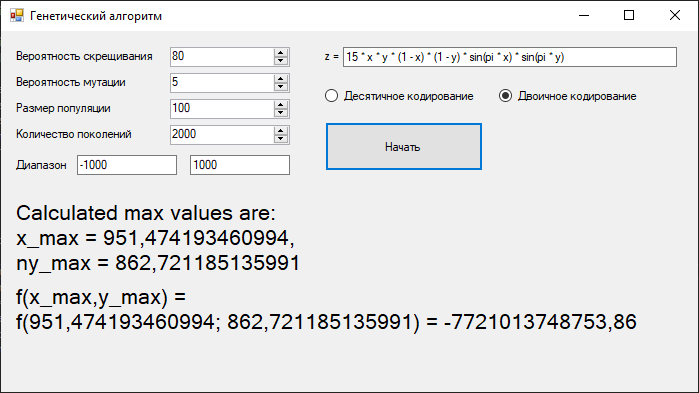


Рис. 3. Пример работы программы 1

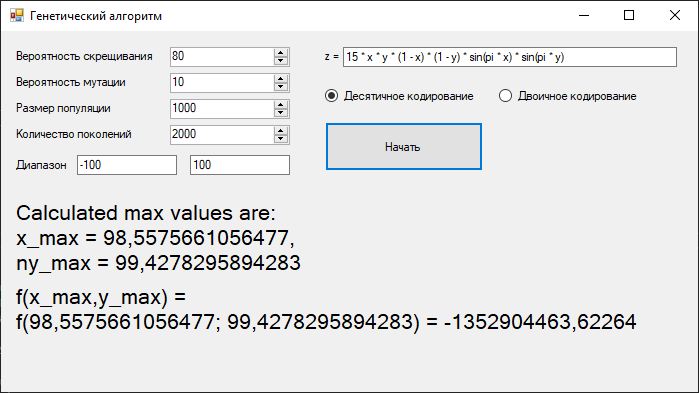


Рис.4. Пример работы программы 2

# Вывод

В ходе выполнения работы стало ясно, что генетический алгоритм очень сильно зависим от заданных параметров. К каждой функции придется прдбирать свои индивидуальные параметры. Если подобрать конфигурацию неверно – алгоритм может “свалиться” в локальный максимум, что не имеет никакой практической ценности для решения поставленных задач. Алгоритм был успешно реализован на языке C#. В лабораторной работе были достигнуты все цели.

# Список используемой литературы

1. Генетический алгоритм // Википедия URL: <https://ru.wikipedia.org/wiki/%D0%93%D0%B5%D0%BD%D0%B5%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC> (дата обращения: 0410.2022).

# Листинг

using System;

using System.Collections;

using System.Collections.Generic;

using System.Globalization;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace GeneticAlgorithm

{

internal class Kernel

{

private static Random rand = new Random();

public delegate double GAFunction(double[] values, string infixPhrase);

public class GA

{

public string infixPhrase;

public double MutationRate;

public double CrossoverRate;

public int ChromosomeLength;

public int PopulationSize;

public int GenerationSize;

public double TotalFitness;

public double Max;

public double Min;

public bool Elitism;

private ArrayList CurrentGenerationList;

private ArrayList NextGenerationList;

private ArrayList FitnessList;

static private GAFunction getFitness;

public GAFunction FitnessFunction

{

get { return getFitness; }

set { getFitness = value; }

}

public GA(double XoverRate, double mutRate, int popSize, int genSize, int ChromLength, string infixPhrase, double max, double min)

{

Elitism = false;

MutationRate = mutRate;

CrossoverRate = XoverRate;

PopulationSize = popSize;

GenerationSize = genSize;

ChromosomeLength = ChromLength;

this.infixPhrase = infixPhrase;

Max = max;

Min = min;

}

public void LaunchGA()

{

FitnessList = new ArrayList();

CurrentGenerationList = new ArrayList(GenerationSize);

NextGenerationList = new ArrayList(GenerationSize);

Chromosome.ChromosomeMutationRate = MutationRate;

for (int i = 0; i < PopulationSize; i++)

{

Chromosome g = new Chromosome(ChromosomeLength,this.Max,this.Min);

CurrentGenerationList.Add(g);

}

RankPopulation();

for (int i = 0; i < GenerationSize; i++)

{

CreateNextGeneration();

RankPopulation();

}

}

private int RouletteSelection()

{

double randomFitness = rand.NextDouble() \* TotalFitness;

int idx = -1;

int mid;

int first = 0;

int last = PopulationSize - 1;

mid = (last - first) / 2;

while (idx == -1 && first <= last)

{

if (randomFitness < (double)FitnessList[mid])

{ last = mid; }

else if (randomFitness > (double)FitnessList[mid])

{ first = mid; }

mid = (first + last) / 2;

if ((last - first) == 1) idx = last;

}

return idx;

}

private void RankPopulation()

{

TotalFitness = 0;

for (int i = 0; i < PopulationSize; i++)

{

Chromosome g = ((Chromosome)CurrentGenerationList[i]);

g.ChromosomeFitness = FitnessFunction(g.ChromosomeGenes, infixPhrase);

TotalFitness += g.ChromosomeFitness;

}

CurrentGenerationList.Sort(new ChromosomeComparer());

double fitness = 0.0;

FitnessList.Clear();

for (int i = 0; i < PopulationSize; i++)

{

fitness += ((Chromosome)CurrentGenerationList[i]).ChromosomeFitness;

FitnessList.Add((double)fitness);

}

}

private void CreateNextGeneration()

{

NextGenerationList.Clear();

Chromosome g = null;

if (Elitism)

g = (Chromosome)CurrentGenerationList[PopulationSize - 1];

for (int i = 0; i < PopulationSize; i += 2)

{

int pidx1 = RouletteSelection();

int pidx2 = RouletteSelection();

Chromosome parent1, parent2, child1, child2;

parent1 = ((Chromosome)CurrentGenerationList[pidx1]);

parent2 = ((Chromosome)CurrentGenerationList[pidx2]);

if (rand.NextDouble() < CrossoverRate)

{ parent1.Crossover(ref parent2, out child1, out child2); }

else

{

child1 = parent1;

child2 = parent2;

}

child1.Mutate();

child2.Mutate();

NextGenerationList.Add(child1);

NextGenerationList.Add(child2);

}

if (Elitism && g != null) NextGenerationList[0] = g;

CurrentGenerationList.Clear();

for (int i = 0; i < PopulationSize; i++)

CurrentGenerationList.Add(NextGenerationList[i]);

}

public void GetBestValues(out double[] values, out double fitness)

{

Chromosome g = ((Chromosome)CurrentGenerationList[PopulationSize - 1]);

values = new double[g.ChromosomeLength];

g.ExtractChromosomeValues(ref values);

fitness = (double)g.ChromosomeFitness;

}

}

public class Chromosome

{

public double[] ChromosomeGenes;

public int ChromosomeLength;

public double ChromosomeFitness;

public static double ChromosomeMutationRate;

public Chromosome(int length)

{

ChromosomeLength = length;

ChromosomeGenes = new double[length];

}

public Chromosome(int length, double max, double min)

{

ChromosomeLength = length;

ChromosomeGenes = new double[length];

for (int i = 0; i < ChromosomeLength; i++)

ChromosomeGenes[i] = rand.NextDouble() \* (Math.Abs(max) + Math.Abs(min)) - (Math.Abs(max) + Math.Abs(min))/2;

}

public void Crossover(ref Chromosome Chromosome2, out Chromosome child1, out Chromosome child2)

{

int position = (int)(rand.NextDouble() \* (double)ChromosomeLength);

child1 = new Chromosome(ChromosomeLength);

child2 = new Chromosome(ChromosomeLength);

for (int i = 0; i < ChromosomeLength; i++)

{

if (i < position)

{

child1.ChromosomeGenes[i] = ChromosomeGenes[i];

child2.ChromosomeGenes[i] = Chromosome2.ChromosomeGenes[i];

}

else

{

child1.ChromosomeGenes[i] = Chromosome2.ChromosomeGenes[i];

child2.ChromosomeGenes[i] = ChromosomeGenes[i];

}

}

}

public void Mutate()

{

for (int position = 0; position < ChromosomeLength; position++)

{

if (rand.NextDouble() < ChromosomeMutationRate)

ChromosomeGenes[position] = (ChromosomeGenes[position] + rand.NextDouble()) / 2.0;

}

}

public void ExtractChromosomeValues(ref double[] values)

{

for (int i = 0; i < ChromosomeLength; i++)

values[i] = ChromosomeGenes[i];

}

}

public sealed class ChromosomeComparer : IComparer

{

public int Compare(object x, object y)

{

if (!(x is Chromosome) || !(y is Chromosome))

throw new ArgumentException("Not of type Chromosome");

if (((Chromosome)x).ChromosomeFitness > ((Chromosome)y).ChromosomeFitness)

return -1;

else if (((Chromosome)x).ChromosomeFitness == ((Chromosome)y).ChromosomeFitness)

return 0;

else

return 1;

}

}

public static double GenAlgTestFcn(double[] values, string infixPhrase)

{

if (values.GetLength(0) != 2)

throw new Exception("should only have 2 args");

double x = values[0]; double y = values[1];

return MathParserSpace.MathParser.calculate(x, y, infixPhrase);

}

}

}