**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

**Содержание**

[Цель работы 3](#_Toc121439270)

[Теория 3](#_Toc121439271)

[Реализация алгоритма 5](#_Toc121439272)

[UML Диаграммы 7](#_Toc121439273)

[Описание классов 8](#_Toc121439274)

[Ход работы 10](#_Toc121439275)

[Пример работы программы 12](#_Toc121439276)

[Вывод 14](#_Toc121439277)

[Список используемой литературы 14](#_Toc121439278)

[Листинг 15](#_Toc121439279)

# Цель работы

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

# Теория

Задача формализуется таким образом, чтобы её решение могло быть закодировано в виде [вектора](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%BA%D1%82%D0%BE%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) («[генотипа](https://ru.wikipedia.org/wiki/%D0%93%D0%B5%D0%BD%D0%BE%D1%82%D0%B8%D0%BF)») генов, где каждый ген может быть [битом](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82), числом или неким другим объектом. В классических реализациях генетического алгоритма (ГА) предполагается, что генотип имеет фиксированную длину. Однако существуют вариации ГА, свободные от этого ограничения.

Некоторым, обычно случайным, образом создаётся множество генотипов начальной популяции. Они оцениваются с использованием «[функции приспособленности](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_%D0%BF%D1%80%D0%B8%D1%81%D0%BF%D0%BE%D1%81%D0%BE%D0%B1%D0%BB%D0%B5%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B8)», в результате чего с каждым генотипом ассоциируется определённое значение («приспособленность»), которое определяет насколько хорошо [фенотип](https://ru.wikipedia.org/wiki/%D0%A4%D0%B5%D0%BD%D0%BE%D1%82%D0%B8%D0%BF), им описываемый, решает поставленную задачу.

При выборе «[функции приспособленности](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_%D0%BF%D1%80%D0%B8%D1%81%D0%BF%D0%BE%D1%81%D0%BE%D0%B1%D0%BB%D0%B5%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B8)» (или fitness function в англоязычной литературе) важно следить, чтобы её «рельеф» был «гладким».

Из полученного множества решений («поколения») с учётом значения «приспособленности» выбираются решения (обычно лучшие особи имеют большую вероятность быть выбранными), к которым применяются «генетические операторы» (в большинстве случаев «[скрещивание](https://ru.wikipedia.org/wiki/%D0%A1%D0%BA%D1%80%D0%B5%D1%89%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)» — crossover и «[мутация](https://ru.wikipedia.org/wiki/%D0%9C%D1%83%D1%82%D0%B0%D1%86%D0%B8%D1%8F)» — mutation), результатом чего является получение новых решений. Для них также вычисляется значение приспособленности, и затем производится отбор («селекция») лучших решений в следующее поколение.

Этот набор действий повторяется итеративно, так моделируется «эволюционный процесс», продолжающийся несколько жизненных циклов (поколений), пока не будет выполнен критерий остановки алгоритма. Таким критерием может быть:

* нахождение глобального, либо субоптимального решения;
* исчерпание числа поколений, отпущенных на эволюцию;
* исчерпание времени, отпущенного на эволюцию.

Генетические алгоритмы служат, главным образом, для поиска решений в многомерных пространствах поиска.

Таким образом, можно выделить следующие этапы генетического алгоритма:

1. Задать целевую функцию (приспособленности) для особей популяции
2. Создать начальную популяцию

* (Начало цикла)

1. Размножение (скрещивание)
2. Мутирование
3. Вычислить значение целевой функции для всех особей
4. Формирование нового поколения (селекция)
5. Если выполняются условия остановки, то (конец цикла), иначе (начало цикла).

Выбор способа кодирования является одним из важнейших этапов при использовании эволюционных алгоритмов. В частности, должно выполняться следующее условие: должна быть возможность закодировать (с допустимой погрешностью) в хромосоме любую точку из рассматриваемой области пространства поиска. Невыполнение этого условия может привести как к увеличению времени эволюционного поиска, так и к невозможности найти решение поставленной задачи. Как правило, в хромосоме кодируются численные параметры решения. Для этого возможно использование целочисленного и вещественного кодирования. Целочисленное кодирование. В классическом генетическом алгоритме хромосома представляет собой битовую строку, в которой закодированы параметры решения поставленной задачи.

Вещественное кодирование. Часто бывает удобнее кодировать в гене не целое число, а вещественное. Это позволяет избавиться от операций кодирования/декодирования, используемых в целочисленном кодировании, а также увеличить точность найденного решения.

# 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 *,* а в другом . При инициализации алгоритма их значения заполняются случайным образом.

После генерации происходит оценка поколения на “выживаемость”. В нашем случае берется каждая хромосома со своими параметрами ’ов и ’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

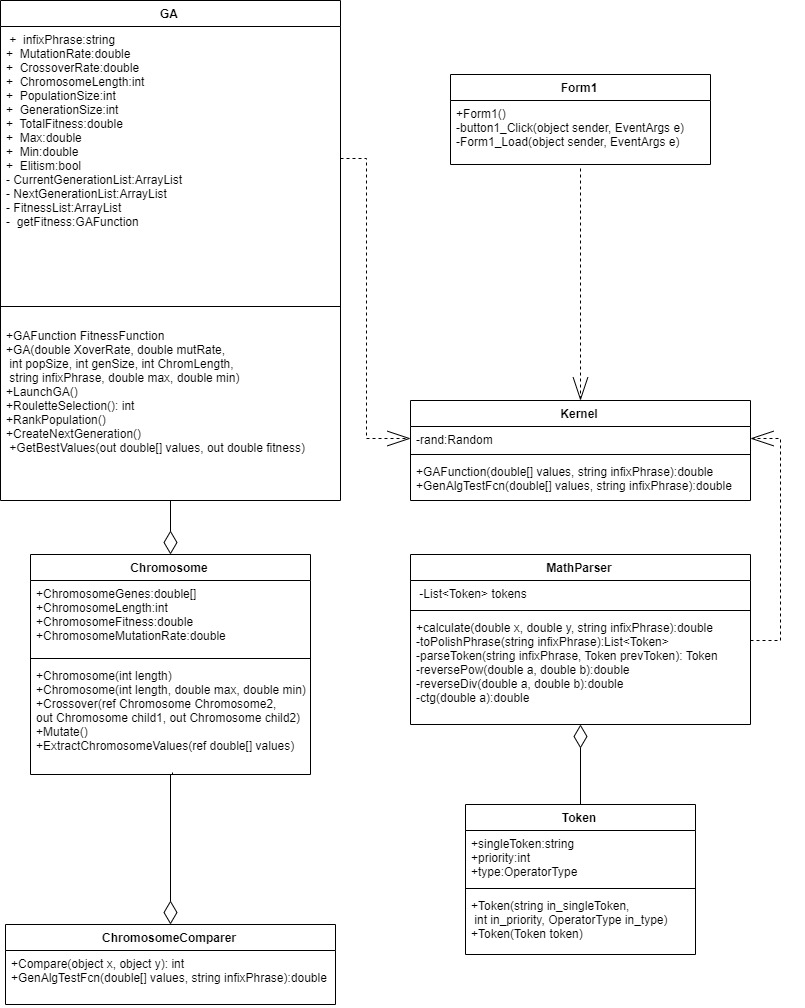


Рис.1. UML диаграмма

# Описание классов

Таблица 1 – Описание методов класса GA

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Выходные параметры | Назначение |
| GA | - | public | double XoverRate, double mutRate, int popSize, int genSize, int ChromLength, string infixPhrase, double max, double min | - | Конструктор |
| LaunchGA | void | public | - | - | Запуск алгоритма |
| RouletteSelection | int | private | - | Индекс селекции | Выбор селекции |
| RankPopulation | void | private | - | - | Оценка популяции |
| GetBestValues | void |  | out double[] values, out double fitness | лучший результат записывается в входные параметры | Получить лучший результат |

Таблица 2 Описание полей и свойств класса GA

|  |  |  |  |
| --- | --- | --- | --- |
| Имя | Тип | Модификатор доступа | Назначение |
| isBit | bool | public | Вид кодирования |
| infixPhrase | string | public | Функция в виде строки |
| MutationRate | double | public | Вероятность мутации |
| CrossoverRate | double | public | Вероятность скрещивания |
| ChromosomeLength | int | public | Длина хромосомы |
| PopulationSize | int | public | Размер популяции |
| GenerationSize | int | public | Кол-во поколений |
| Max | double | public | Максимальный диапазон расчетов |
| Min | double | public | Минимальный диапазон расчетов |
| Elitism | bool | public | Флаг перехода генов в след. поколение |
| CurrentGenerationList | ArrayList | private | Список текущего поколения |
| NextGenerationList | ArrayList | private | Список следующего поколения |

Таблица 3 – Описание методов класса Chromosome

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Выходные параметры | Назначение |
| Chromosome | - | public | int length | - | Конструктор |
| Chromosome | - | public | int length, double max, double min, bool isBit | - | Конструктор |
| Crossover | void | public | ref Chromosome Chromosome2, out Chromosome child1, out Chromosome child2 | - | Скрещивание |
| Mutate | void | public | - | - | Мутация |
| ExtractChromosomeValues | void | public | ref double[] values | - | Перенос полученных значение в локальное поле |

Таблица 4 Описание полей и свойств класса Chromosome

|  |  |  |  |
| --- | --- | --- | --- |
| Имя | Тип | Модификатор доступа | Назначение |
| ChromosomeGenes | double[] | public | Гены хромосомы |
| ChromosomeLength | int | public | Длина хромосомы |
| ChromosomeFitness | double | public | Оценка хромосомы |
| ChromosomeMutationRate | double | public | Вероятность мутации |

Таблица 5 – Описание методов класса Kernel

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Метод | Возвращаемый тип | Модификатор доступа | Входные параметры | Выходные параметры | Назначение |
| GenAlgTestFcn | double | public | double[] values, string infixPhrase | Значение функции в точке | Посчитать функцию |
| bitToDecimal | double | private | double[] bitValue, int amountOfBits | Десятичное число | Перевод двоичного числа в десятичное |
| DecimalToBit | double[] | private | double value | Двоичное число | Перевод десятичного числа в двоичное |

Таблица 6 Описание полей и свойств класса Kernel

|  |  |  |  |
| --- | --- | --- | --- |
| Имя | Тип | Модификатор доступа | Назначение |
| rand | Random | private | Объект класса псевдорандома |

# Ход работы

Для выполнения данной лабораторной работы был выбран язык C#.

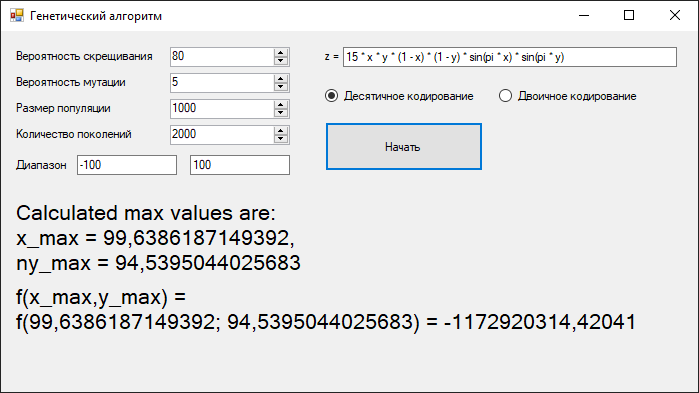


Рис.2. Главное окно приложения

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 и , 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. Пользователю требуется ввести функцию в текстовое поле «z=», а также параметры в левой части окна программы. После чего нажать кнопку «Начать»

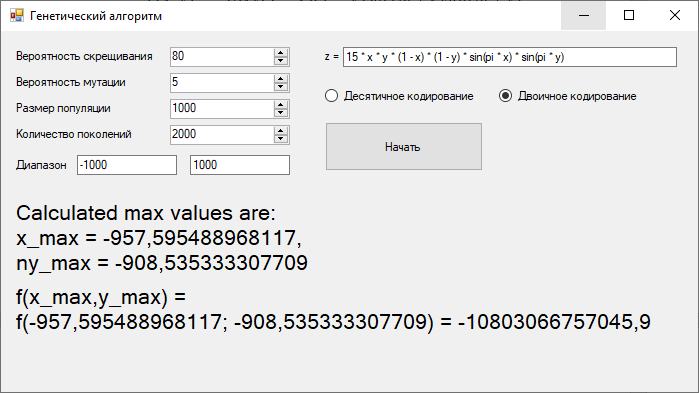


Рис. 3. Пример работы программы 1

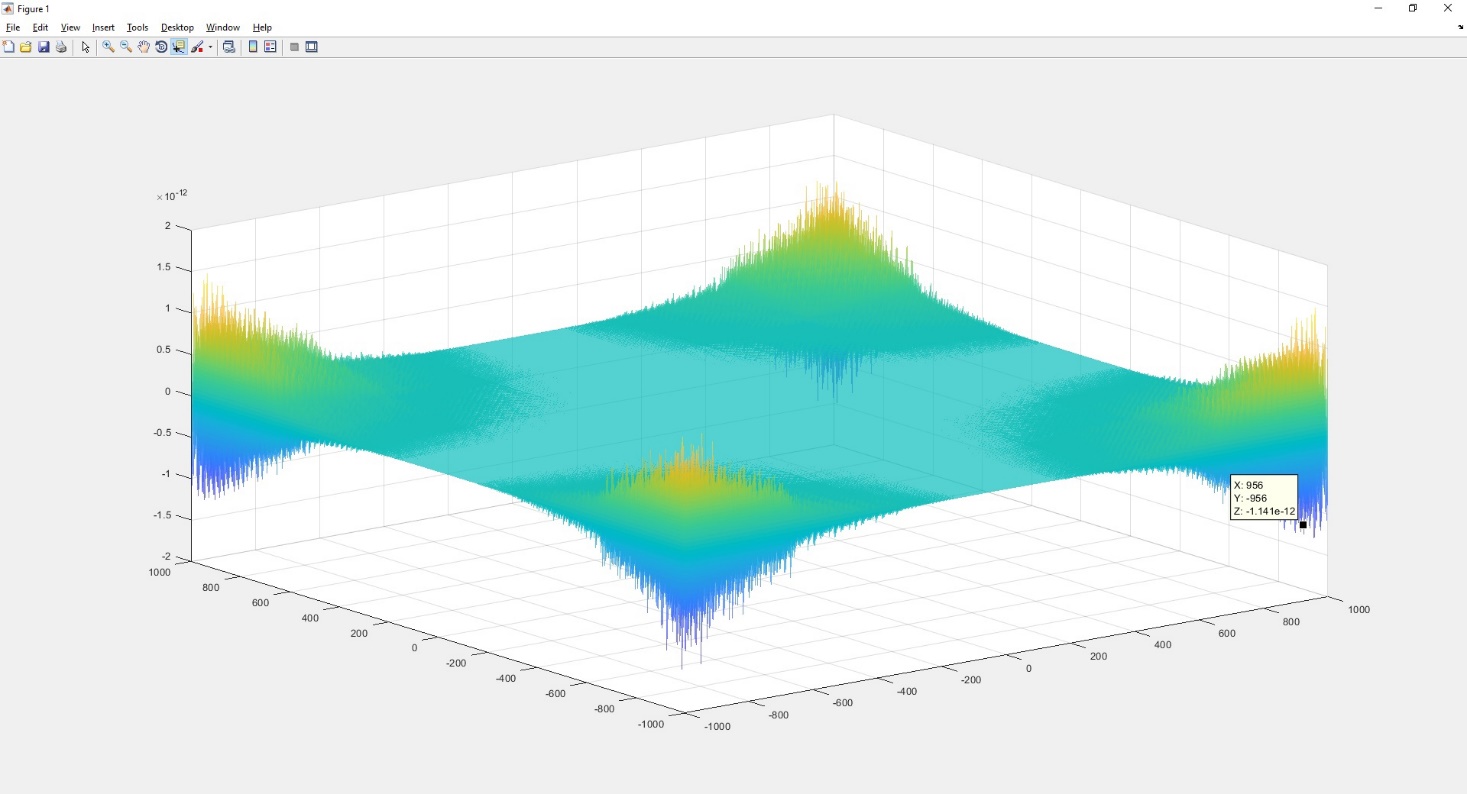


Рис. 4. График функции для примера 1

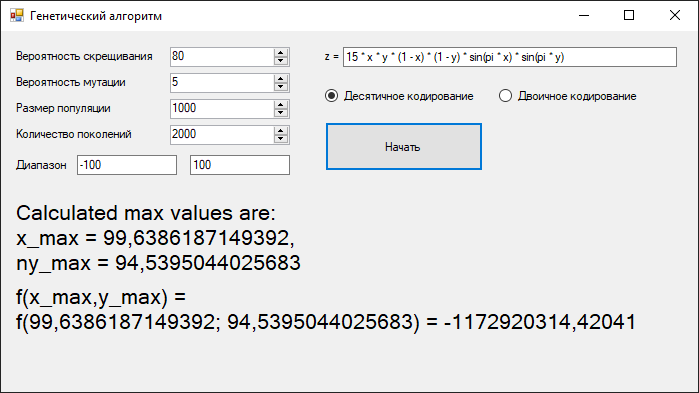


Рис.5. Пример работы программы 2

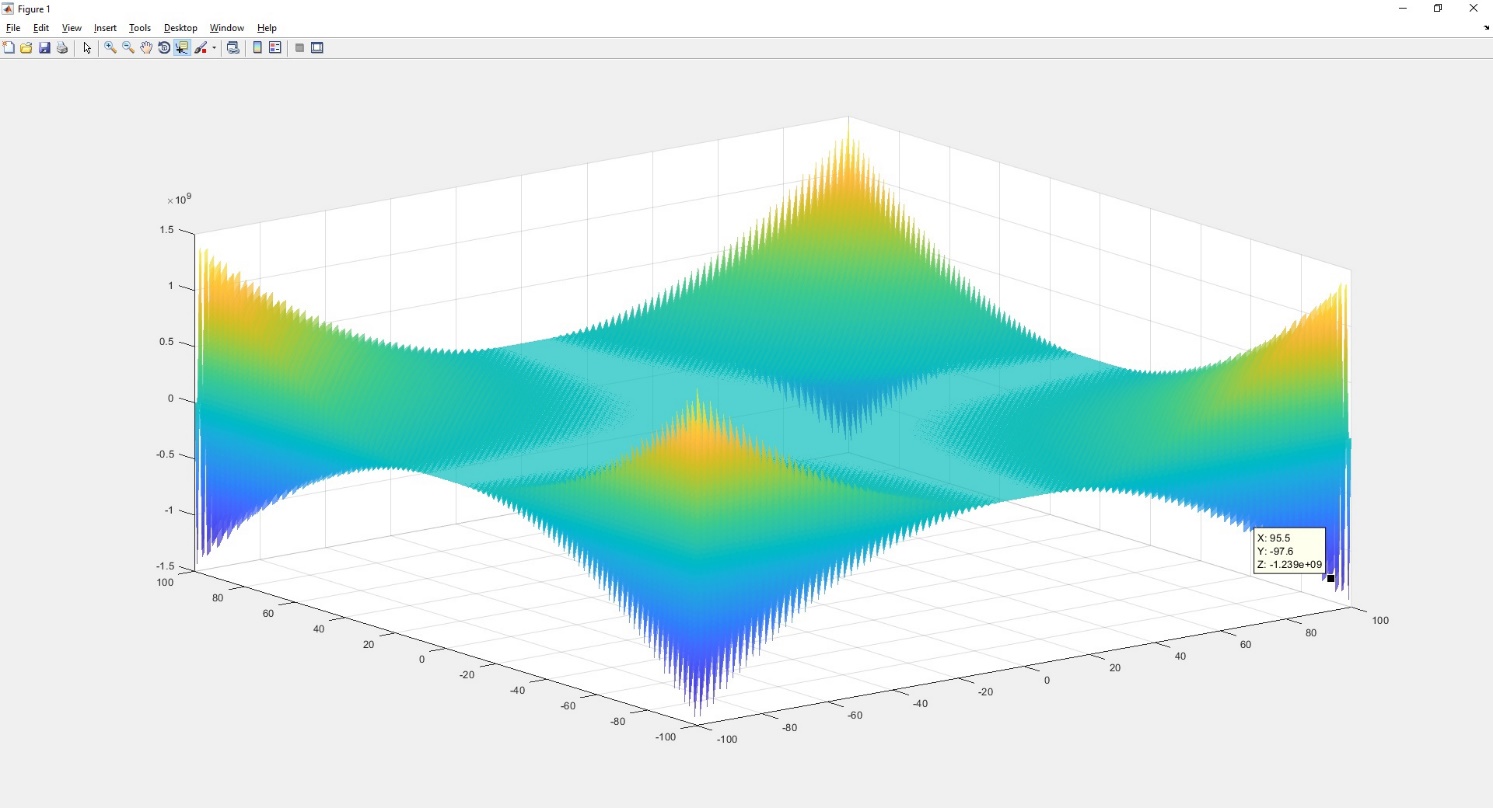


Рис. 6. График функции для примера 2

# Вывод

В ходе выполнения работы стало ясно, что генетический алгоритм очень сильно зависим от заданных параметров. К каждой функции придется подбирать свои индивидуальные параметры. Если подобрать конфигурацию неверно – алгоритм может “свалиться” в локальный максимум, что не имеет никакой практической ценности для решения поставленных задач. В ходе тестирования было также установлено, что бинарный способ кодирования генов менее точный. 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);

}

}

}