МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский Томский государственный университет»

Институт прикладной математики и компьютерных наук

Лабораторная работа

по дисциплине «Интеллектуальные системы»

на тему «Генетические алгоритмы»

Работу выполнили:

студенты 2 курса 932001 гр.  
Д. В. Малясов  
Д. Д. Степаньков  
Е. К. Куюмджиев  
Е. А. Орлов

Проверила:

Бакланова Ольга Евгеньевна

Томск  
2022

**1. Цель работы**

Создать программу для нахождения минимума функции y(x) = , используя целочисленное и вещественное кодирование.

# 2. Задачи работы

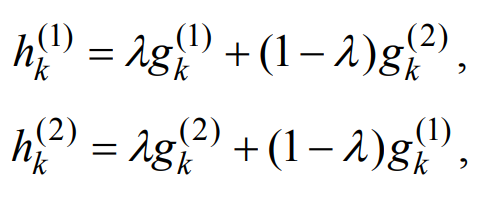
Создать генетический алгоритм, в котором будет оценена популяция и определен критерий оценки особей с помощью функции приспособленности, реализованы: селекция особей для скрещивания, операция скрещивания, оператор мутации.

# 3. Метод решения задачи Был реализован класс отдельной особи, класс популяции и функции непосредственно генетического алгоритма. Класс Prim характеризует конкретную особь и содержит конструкторы, метод установки значения особи, а также метод вычисления приспособленности данной особи.

Класс **Population** содержит конструкторы, метод(FindBest), вычисляющий наиболее приспособленную особь, и метод(CalculateFitnessPopulation), вычисляющий среднее значение приспособленности данной популяции.

Функция турнирного отбора выбирает две особи из популяции, выбирает лучшую и кладет её в следующее поколение. Так повторяется, пока новое поколение не будет сформировано.

Функция ArithmeticCrossing, используя формулы ниже, скрещивает пары родительских особей с вероятностью P\_Crossing(P\_Crossing = 90%)



Функция мутации с вероятность P\_Mutation прибавляет к имеющемуся значению гена число, лежащее в интервале [0.25, 0.25].

В результате турнирного отбора, одна и та же особь может попасть в новое поколение, и тогда две ссылки будут ссылаться на один объект. Чтобы не происходило ошибок, связанных с памятью, используется функция Clone, создающая для каждой особи новый экземпляр. Таким образом, каждой особи соответствует свой объект.

В цикле генетического алгоритма последовательно вызываются указанные функции 50 раз. Наблюдается сходимость приспособленности к точке 4, что соответствует значению x = 0.

# 4. Структурная схема алгоритма

# 

**Формирование начальной популяции:** Начальная популяция формируется случайным образом.

Оценивание популяции: С помощью целевой функции подсчитывается приспособленность каждой особи, таким образом выявляются более и менее приспособленные особи.

*fi* = *f* (**G***i*),

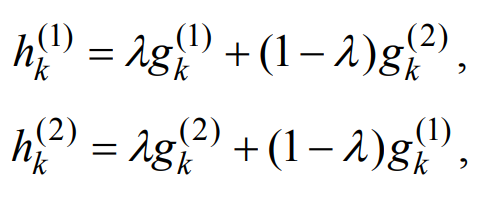
где **G***i* = {*gik* : *k* = 1,2,...,*N* } – хромосома *i*-й особи, g*ik* – значение *k*-го гена *i*-й особи, *N* – количество генов в хромосоме. При целочисленном кодировании необходимо преобразовать закодированные в хромосоме целочисленные значения к вещественным числам.

**Селекция:** отбор наиболее приспособленных особей для скрещивания. Метод селекции, примененный в программе – турнирный отбор. Принцип работы: случайно выбираются t особей (в нашем случае t = 2), и самая приспособленная допускается к скрещиванию. Операция повторяется n раз.

**Скрещивание и формирование нового поколения:**

В нашем случае используется функция арифметического кроссинговера.

обмен информацией между родительскими особями и потомками производится с учетом значений генов родителей по формуле:



Где gk – значение k-го гена родителя (в нашем случае он один), а лямбда равна лежит в диапазоне [0, 1] (в нашем случае лямбда равна 0.55).

**Мутация:** Оператор мутации для вещественного кодирования изменяет содержимое каждого гена с вероятностью *PM*. При этом величина изменения выбирается случайно в некотором диапазоне [– ξ; + ξ], в нашем случае, [−0,25; 0,25].

**5. Листинг программы.**

#include <iostream>

#include <math.h>

using namespace std;

int maximum = 100, minimum = -100; //Рассматриваемый интервал

double CalculateFitness(double x) { return x \* x + 4; }

class Prim

{

double gen;

int gen2;

public:

Prim()

{

gen = 0;

gen2 = 0;

}

Prim(double x)

{

gen = x;

gen2 = Encode(x);

}

Prim(const Prim& obj)

{

gen = obj.gen;

gen2 = obj.gen2;

}

double GetGen() { return gen; }

int GetGenInt() { return gen2; }

void OutputGenInConsole()

{

for (int i = 0; i < 15; i++)

{

cout << ((gen2 >> i) & 1);

}

cout << endl;

}

void SetGen(double x) { gen = x; gen2 = Encode(x); }

double SelfFitness() { return CalculateFitness(gen); }

Prim& operator =(const Prim& obj)

{

if (this == &obj)

return \*this;

gen = obj.gen;

gen2 = obj.gen2;

return \*this;

}

};

class Population

{

Prim\* mas;

int size;

double\* MasFitness;

public:

Population(int primNumber)

{

mas = new Prim[primNumber];

size = primNumber;

MasFitness = new double[primNumber];

}

//Конструктор копий

Population(const Population& obj)

{

mas = new Prim[obj.size];

size = obj.size;

MasFitness = new double[obj.size];

for (int i = 0; i < obj.size; i++)

this->mas[i] = obj.mas[i];

for (int i = 0; i < obj.size; i++)

this->MasFitness[i] = obj.MasFitness[i];

}

Prim& operator [] (int i) { return mas[i]; }

int Size() { return size; }

//Функция оценки популяции

double CalculateFitnessPopulation(double& x)

{

double f = 0;

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

{

MasFitness[i] = CalculateFitness(mas[i].GetGen()); //Целевая функция f = x \* x + 4;

f += MasFitness[i];

x += mas[i].GetGen();

}

f = f / size;

x = x / size;

return f;

}

Population& operator = (const Population& obj)

{

if (this == &obj)

return \*this;

for (int i = 0; i < obj.size; i++)

this->mas[i] = obj.mas[i];

for (int i = 0; i < obj.size; i++)

this->MasFitness[i] = obj.MasFitness[i];

return \*this;

}

friend ostream& operator <<(ostream& stream, Population& population)

{

for (int i = 0; i < population.size; i++)

{

stream << population[i].GetGen() << endl;

}

return stream;

}

double FindBest(double &x)

{

CalculateFitnessPopulation(x); //Заполняем массив MasFitness внутри функции

double z = 0;

for (int i = 0; i < size - 1; i++)

{

if (MasFitness[i] <= MasFitness[i + 1])

{

z = MasFitness[i];

x = mas[i].GetGen();

}

else

{

z = MasFitness[i + 1];

x = mas[i + 1].GetGen();

}

}

return z;

}

};

//Функция турнирного отбора

void TournamentSelection(Population& population, int populationSize, Population& offspring)

{

for (int j = 0; j < populationSize; j++)

{

int i1 = 0, i2 = 0;

while (i1 == i2)

{

i1 = rand() % populationSize;

i2 = rand() % populationSize;

}

if (population[i1].SelfFitness() < population[i2].SelfFitness())

offspring[j] = population[i1];

else

offspring[j] = population[i2];

}

}

//Функция арифметического кроссинговера

void ArithmeticCrossing(Prim& parent1, Prim& parent2)

{

Prim child1, child2;

double lambda = 0.55;

child1.SetGen(lambda \* parent1.GetGen() + (1 - lambda) \* parent2.GetGen()); //ген первого потомка

child2.SetGen(lambda \* parent2.GetGen() + (1 - lambda) \* parent1.GetGen()); //ген второго потомка

parent1 = child1; parent2 = child2; //родители заменяются на потомков

}

//Функция мутации индивида

void Mutation(Prim& individual, short P\_Mutation)

{

//Пусть величина мутации будет в диапазоне от [-0.25; 0.25]

if (rand() % 100 < P\_Mutation) //Мутация происходит с вероятностью P\_Mutation

{

double newGen = (rand() % 51 - 25.0) / 100; //Генерируем случайное число в указанном интервале

newGen += individual.GetGen(); //Прибавляем его к исходному гену

individual.SetGen(newGen); //Теперь это ген i-ой особи

}

}

Population Clone(Population& population, int populationSize)

{

Population newPop(populationSize);

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

newPop[i] = population[i];

return newPop;

}

int main()

{

setlocale(LC\_ALL, "ru");

srand(time(0));

int populationSize, generationCounter = 0;

const short maxGeneration = 50, P\_Crossing = 90, P\_Mutation = 5;

cout << "Введите количество особей в популяции:\n";

cin >> populationSize; cout << endl; //Ввод размера популяции

//Переменные для целочисленного кодирования

//Генерация начальной популяции

Population population(populationSize);

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

//i-й особи присваивается значение в диапазоне [-100, 100]

population[i].SetGen((rand() % (maximum - minimum + 1)) - ((maximum - minimum)/2));

double\* bestFitnessValues = new double[maxGeneration]; //значения наиболее приспособленной особи в i-ой популяции

double\* bestValues = new double[maxGeneration];

double\* meanFitnessValues = new double[maxGeneration]; //средние значения приспособленности особей в i-ой популяции

double\* meanValues = new double[maxGeneration];

double x = 0;

bestFitnessValues[0] = population.FindBest(x);

bestValues[0] = x;

meanFitnessValues[0] = population.CalculateFitnessPopulation(x);

meanValues[0] = x;

Population offspring(populationSize); //Промежуточная переменная

//Собственно генетический алгоритм для ВЕЩЕСТВЕННОГО КОДИРОВАНИЯ

cout << "ВЕЩЕСТВЕННОЕ КОДИРОВАНИЕ:\n";

//cout << population;

for (generationCounter = 1; generationCounter < maxGeneration; generationCounter++)

{

TournamentSelection(population, populationSize, offspring);

offspring = Clone(offspring, populationSize);

//Цикл скрещивания

for (int i = 0; i < populationSize / 2; i++)

{

if (rand() % 100 < P\_Crossing) //Скрещивание происходит с вероятностью 90%

ArithmeticCrossing(offspring[i], offspring[populationSize - i]); //Скрещивается i с n-i особью, где n- размер популяции

}

if (populationSize % 2 == 1) //Если кол-во особей нечётно, оставшаяся особь скрещивается со случайной

{

int i1 = populationSize / 2 + 1;

int i2 = rand() % populationSize;

ArithmeticCrossing(offspring[i1], offspring[i2]);

}

//Цикл мутаций

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

{

Mutation(offspring[i], P\_Mutation);

}

population = offspring;

bestFitnessValues[generationCounter] = population.FindBest(x);

bestValues[generationCounter] = x;

meanFitnessValues[generationCounter] = population.CalculateFitnessPopulation(x);

meanValues[generationCounter] = x;

}

cout << "BEST FITNESS:\n";

for (int i = 0; i < 50; i++)

{

cout << i + 1 << "; " << bestFitnessValues[i] << ";" << endl;

}

cout << "\n\nMEAN FITNESS:\n";

for (int i = 0; i < 50; i++)

{

cout << i + 1 << "; " << meanFitnessValues[i] << ";" << endl;

}

cout << "\n\nBEST X:\n";

for (int i = 0; i < 50; i++)

{

cout << i + 1 << "; " << bestValues[i] << ";" << endl;

}

cout << "\n\nMEAN X:\n";

for (int i = 0; i < 50; i++)

{

cout << i + 1 << "; " << meanValues[i] << ";" << endl;

}

return 0;

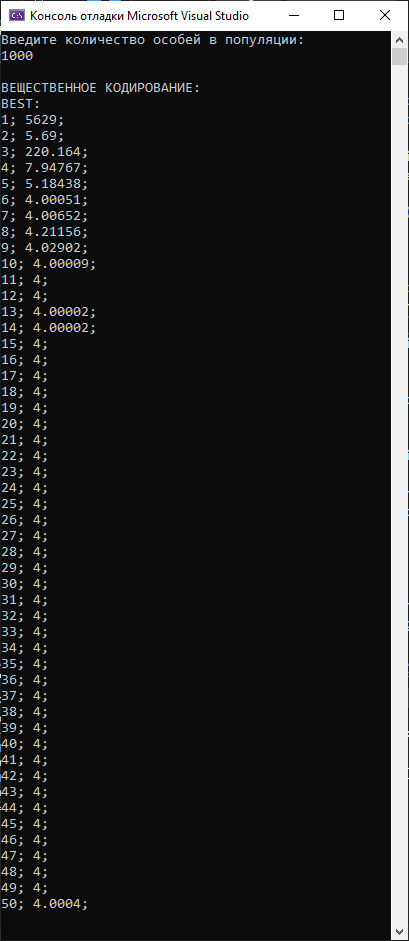
}

Начало формы

Конец формы

# 6. Результат работы алгоритма

BEST – лучшая особь в каждой популяции



MEAN – среднее значение приспособленности популяции.

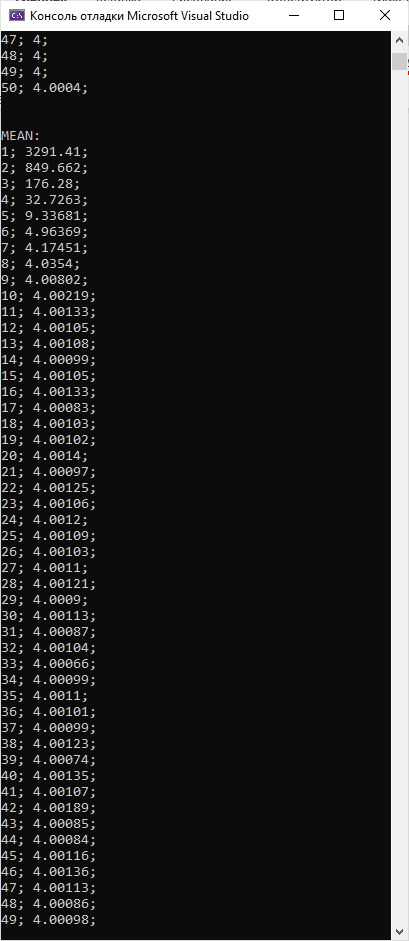
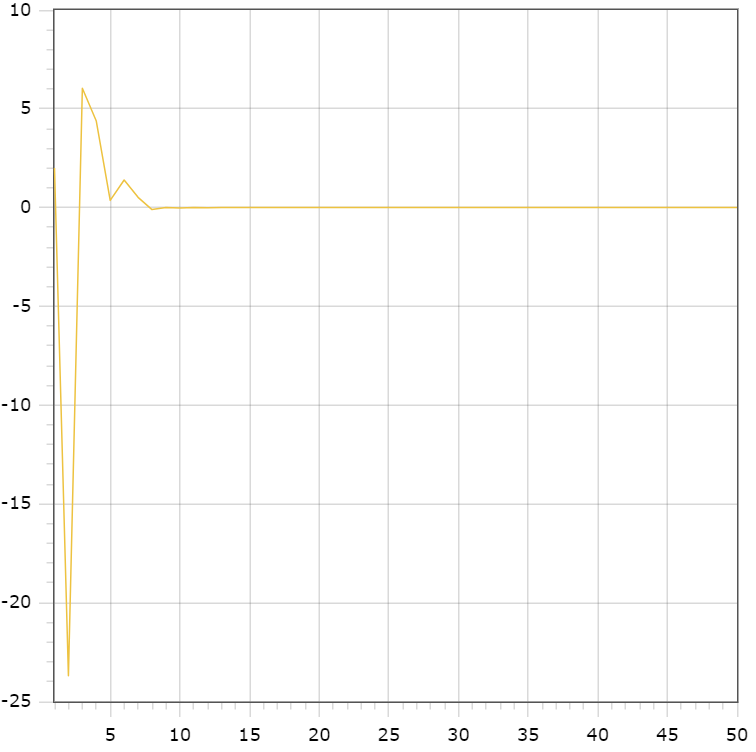
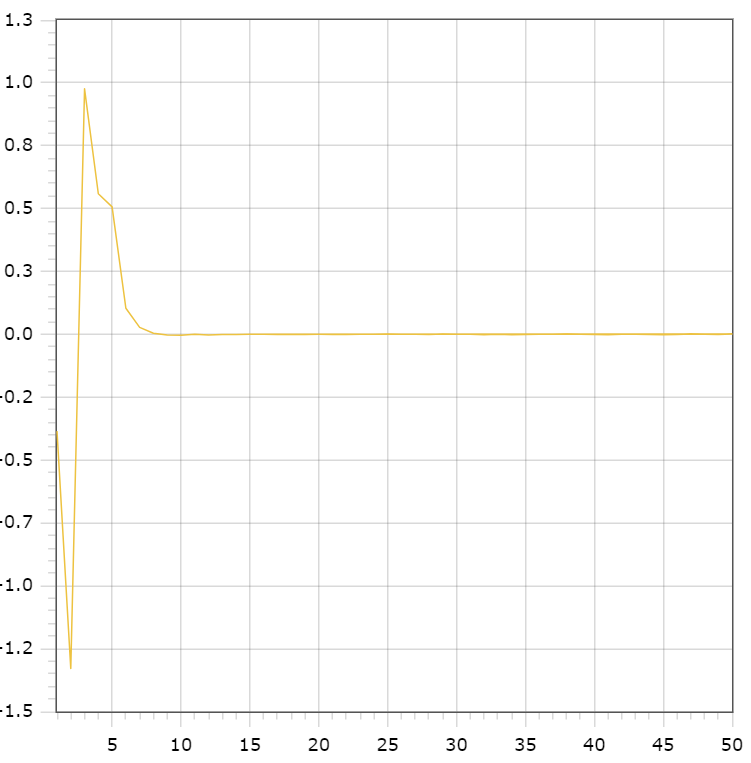


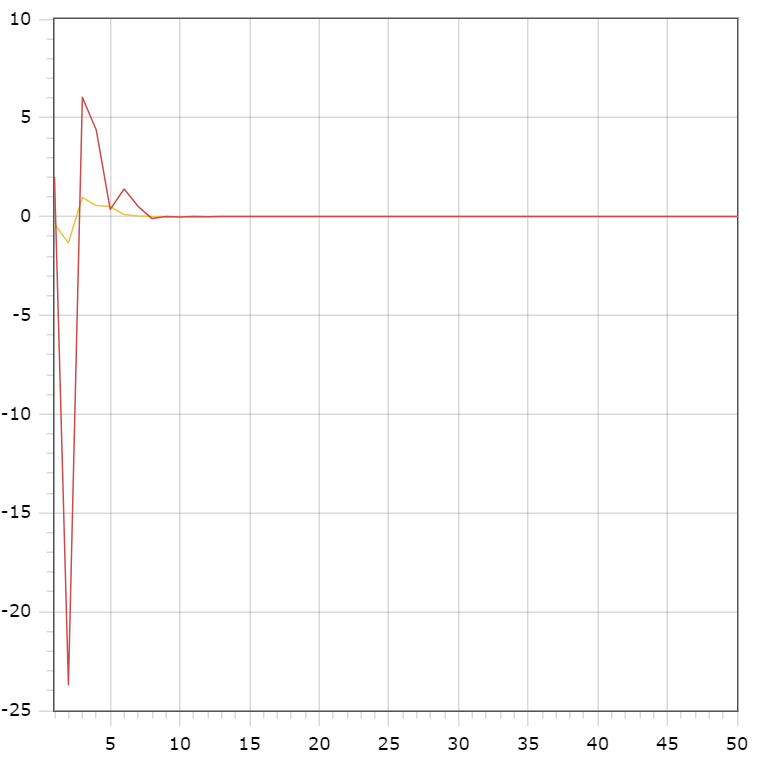
График зависимости значений x от поколений:



Среднее значение:



Совмещенный график:



# Ссылка на GitHub:

https://github.com/FrantsuzovDmitry/Programs/blob/734491deb75afbf25554b5632fffd63cf8d90187/Lab3\_IS\_binary.cpp

# Выводы

Итак, нашей командой был рассмотрен генетический алгоритм как один из самых распространенных эволюционных алгоритмов.

Нам удалось получить достаточно точное решение уже на 10 итерации.

В нашей работе генетический алгоритм решает задачу нахождения минимума функции. На примере этой работы была проиллюстрирована универсальность алгоритма и его непосредственная практическая польза.

Подвергнув анализу полученные данные в ходе проведения лабораторной работы, можно сделать вывод, доказывающий, что особенностью генетических алгоритмов является то, что они обеспечивают сходимость к глобальному оптимуму, а не являются случайным поиском решения функции.