ГУАП

КАФЕДРА № 43

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| д-р техн. наук, профессор |  |  |  | Ю.А. Скобцов |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

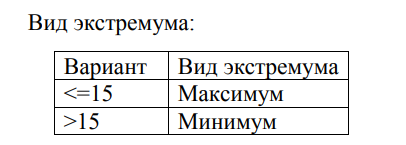
|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №1 |
| Простой генетический алгоритм |
| по курсу: ЭВОЛЮЦИОННЫЕ МЕТОДЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНО-ИНФОРМАЦИОННЫХ СИСТЕМ |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 4136 |  |  |  | Якшин С.Е. |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2024

**Индивидуальное задание по варианту:**



Вариант 22



**Теоретические сведения:**

ГА используют принципы и терминологию, заимствованные у

биологической науки – генетики. В ГА каждая особь представляет

потенциальное решение некоторой проблемы. В классическом ГА особь

кодируется строкой двоичных символов – хромосомой, каждый бит которой

называется геном. Множество особей – потенциальных решений составляет

популяцию. Поиск (суб)оптимального решения проблемы выполняется в

процессе эволюции популяции - последовательного преобразования одного

конечного множества решений в другое с помощью генетических операторов

репродукции, кроссинговера и мутации.

ЭВ используют следующие механизмы естественной эволюции:

1) Первый принцип основан на концепции выживания сильнейших

и естественного отбора по Дарвину, который был сформулирован им в 1859

году в книге «Происхождение видов путем естественного отбора». Согласно

Дарвину особи, которые лучше способны решать задачи в своей среде,

выживают и больше размножаются (репродуцируют). В генетических

алгоритмах каждая особь представляет собой решение некоторой проблемы.

По аналогии с этим принципом особи с лучшими значениями целевой

(фитнесс) функции имеют большие шансы выжить и репродуцировать.

Формализация этого принципа, как мы увидим далее, дает оператор

репродукции.

2) Второй принцип обусловлен тем фактом, что хромосома потомка

состоит из частей полученных из хромосом родителей. Этот принцип был

открыт в 1865 году Менделем. Его формализация дает основу для оператора

скрещивания (кроссинговера).

3) Третий принцип основан на концепции мутации, открытой в 1900

году де Вре. Первоначально этот термин использовался для описания

существенных (резких) изменений свойств потомков и приобретение ими

свойств, отсутствующих у родителей. По аналогии с этим принципом

генетические алгоритмы используют подобный механизм для резкого

изменения свойств потомков и тем самым, повышают разнообразие

(изменчивость) особей в популяции (множестве решений). Эти три принципа составляют ядро ЭВ. Используя их, популяция

(множество решений данной проблемы) эволюционирует от поколения к

поколению.

Эволюцию искусственной популяции – поиска множества решений

некоторой проблемы формально можно описать алгоритмом, который

представлен на рис.1.1.

ГА берет множество параметров оптимизационной проблемы и

кодирует их последовательностями конечной длины в некотором конечном

алфавите (в простейшем случае двоичный алфавит «0» и «1») .

Предварительно простой ГА случайным образом генерирует

начальную популяцию стрингов (хромосом). Затем алгоритм генерирует

следующее поколение (популяцию), с помощью трех основных генетических

операторов:

1) Оператор репродукции (ОР);

2) Оператор скрещивания (кроссинговера, ОК);

3) Оператор мутации (ОМ).

Генетические операторы являются математической формализацией

приведенных выше трех основополагающих принципов Дарвина, Менделя и

де Вре естественной эволюции.ГА работает до тех пор, пока не будет выполнено заданное количество

поколений (итераций) процесса эволюции или на некоторой генерации будет

получено заданное качество или вследствие преждевременной сходимости

при попадании в некоторый локальный оптимум. В каждом поколении множество искусственных особей создается с

использованием старых и добавлением новых с хорошими свойствами.

Генетические алгоритмы - не просто случайный поиск, они эффективно

используют информацию, накопленную в процессе эволюции.

В отличие от других методов оптимизации ГА оптимизируют

различные области пространства решений одновременно и более

приспособлены к нахождению новых областей с лучшими значениями

целевой функции за счет объединения квазиоптимальных решений из разных

популяций.

**Код программы:**

#include <iostream>

#include <cinttypes>

#include <cmath>

#include <algorithm>

#include <numeric>

#include <exception>

#include <random>

#include "Plot.h"

namespace lab1

{

typedef uint16\_t Entity;

typedef std::vector<Entity> Population;

typedef std::vector<float> Fitness ;

namespace k

{

const uint32\_t PRECISION = 1 << 14;

const uint8\_t START = 1, END = 10;

float P\_MUTATION = 0.0001; // Вероятность мутации

float P\_CROSSING = 0.5; // Вероятность скрещивания

uint16\_t N = 100;

float f(float x)

{

return std::log(x) \* std::cos(3 \* x - 15);

}

template<typename T>

std::vector<float> f(std::vector<T> \_x)

{

std::vector<float> \_\_x;

\_\_x.reserve(\_x.size());

for (const auto &x : \_x)

\_\_x.push\_back(f(float(x)));

return \_\_x;

}

float fitness(float x)

{

return - std::log(x) \* std::cos(3 \* x - 15);

}

template<typename T>

Fitness fitness(std::vector<T> \_x)

{

std::vector<float> \_\_x;

\_\_x.reserve(\_x.size());

for (const auto &x : \_x)

\_\_x.push\_back(fitness(float(x)));

return \_\_x;

}

// Функция для перевода вещественного числа из промежутка в целочисленный эквивалент

uint16\_t chunk(float value)

{

float step = (float)(k::END - k::START) / k::PRECISION;

return uint16\_t((value - k::START) / float(step));

}

// Функция для обратного оперевода

float value(uint16\_t chunk)

{

float step = (float)(k::END - k::START) / k::PRECISION;

return k::START + (float)chunk \* step;

}

}

template<typename T1, typename T2>

void is\_aboba(std::vector<T1> \_, std::vector<T2> \_\_)

{

if (\_.size() != \_\_.size())

throw std::exception();

}

namespace op

{

Population selection(const Population &population,

const Fitness &fitness,

int num\_selected)

{

is\_aboba(population, fitness);

std::vector<std::pair<float /\* <- fit \*/ , uint16\_t /\* <- source ind \*/>> sorted\_fitness;

for (size\_t i = 0; i < fitness.size(); ++i)

sorted\_fitness.emplace\_back(fitness[i], i);

// auto min = \*std::min\_element(fitness.begin(), fitness.end());

//

// auto positive\_fitness = fitness;

// for (auto &fit : positive\_fitness)

// fit += std::abs(min);

//

// auto sum = std::accumulate(positive\_fitness.begin(), positive\_fitness.end(), 0.0f);

//

// std::vector<std::pair<float /\* <- prob \*/ , uint16\_t /\* <- source ind \*/>> probabilities;

// for (size\_t i = 0; i < positive\_fitness.size(); ++i)

// probabilities.push\_back({positive\_fitness[i] / float(sum), i});

std::sort(sorted\_fitness.begin(), sorted\_fitness.end(), [](const auto& a, const auto& b)

{

return a.first > b.first;

});

Population selected\_population;

for (int i = 0; i < num\_selected; ++i) {

selected\_population.push\_back(population[sorted\_fitness[i].second]);

}

return selected\_population;

}

Population crossover(const Population &population)

{

Population new\_population;

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dist(1, 15);

for (size\_t i = 0; i < population.size(); i += 2) {

uint16\_t parent1 = population[i];

uint16\_t parent2 = (i + 1 < population.size()) ? population[i + 1] : population[i];

int crossover\_point = dist(gen);

uint16\_t mask1 = (1 << crossover\_point) - 1;

uint16\_t mask2 = ~mask1;

uint16\_t child1 = (parent1 & mask1) | (parent2 & mask2);

new\_population.push\_back(child1);

}

return new\_population;

}

Population mutate(const Population &population, float mutation\_rate)

{

Population mutated\_population;

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_real\_distribution<> dist(0.0, 1.0);

for (uint16\_t entity : population) {

for (int i = 0; i < 14; ++i) {

if (dist(gen) < mutation\_rate) {

entity ^= (1 << i); // Инвертируем i-й бит

}

}

// Ограничиваем значение в пределах 2\*\*14

if (entity > (1 << 14) - 1) {

entity = (1 << 14) - 1;

}

mutated\_population.push\_back(entity);

}

return mutated\_population;

}

}

int main()

{

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_real\_distribution<> dist(1, 10);

Population population;

population.reserve(k::N);

for (size\_t i = 0; i < k::N; ++i) {

population.push\_back(k::chunk((float)dist(gen)));

}

Population source\_population = population;

std::vector<float> x(k::N);

double step = float(k::END - k::START) / k::N;

double current\_value = 1.0;

for (int i = 0; i < k::N; ++i) {

x[i] = current\_value;

current\_value += step;

}

std::vector<float> y;

for (auto \_ : x)

y.push\_back(k::f(\_));

{

std::vector<float> \_x;

for (const auto& \_p: population)

\_x.push\_back(k::value(\_p));

std::vector<float> p;

for (const auto &\_p : population)

p.push\_back(k::f(k::value(\_p)));

plot::print(x, y, \_x, p);

}

auto get\_unchunked\_population = [](const Population& \_){

std::vector<float> out;

out.reserve(\_.size());

for (const auto& chunk: \_)

out.push\_back(k::value(chunk));

return out;

};

for (int i = 0; i < 10; ++i)

{

// 1. Выбор родителей для процесса размножения (работает оператор селекции - репродукции)

Population parents = op::selection(population, k::fitness(get\_unchunked\_population(population)), k::N \* k::P\_CROSSING);

// 2. Создание потомков выбранных пар родителей (работает оператор скрещивания - кроссинговера)

Population childs = op::crossover(parents);

// 3. Мутация новых особей (работает оператор мутации)

childs = op::mutate(childs, k::P\_MUTATION);

// 4. Расширение популяции за счет добавления новых только что порожденных особей

population.insert(population.end(), childs.begin(), childs.end());

// 5. Сокращение расширенной популяции до исходного размера (работает оператор редукции)

population = op::selection(population, k::fitness(get\_unchunked\_population(population)), k::N);

}

{

std::vector<float> \_x;

for (const auto& \_p: population)

\_x.push\_back(k::value(\_p));

std::vector<float> p;

for (const auto &\_p : population)

p.push\_back(k::f(k::value(\_p)));

plot::print(x, y, \_x, p);

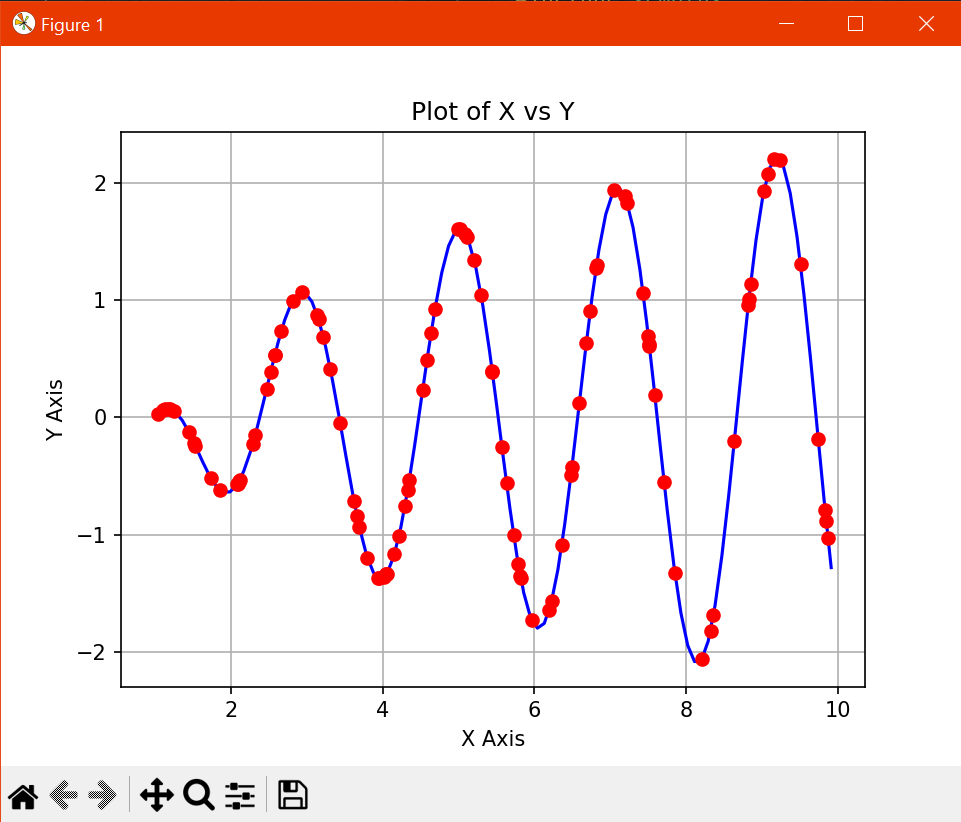
}

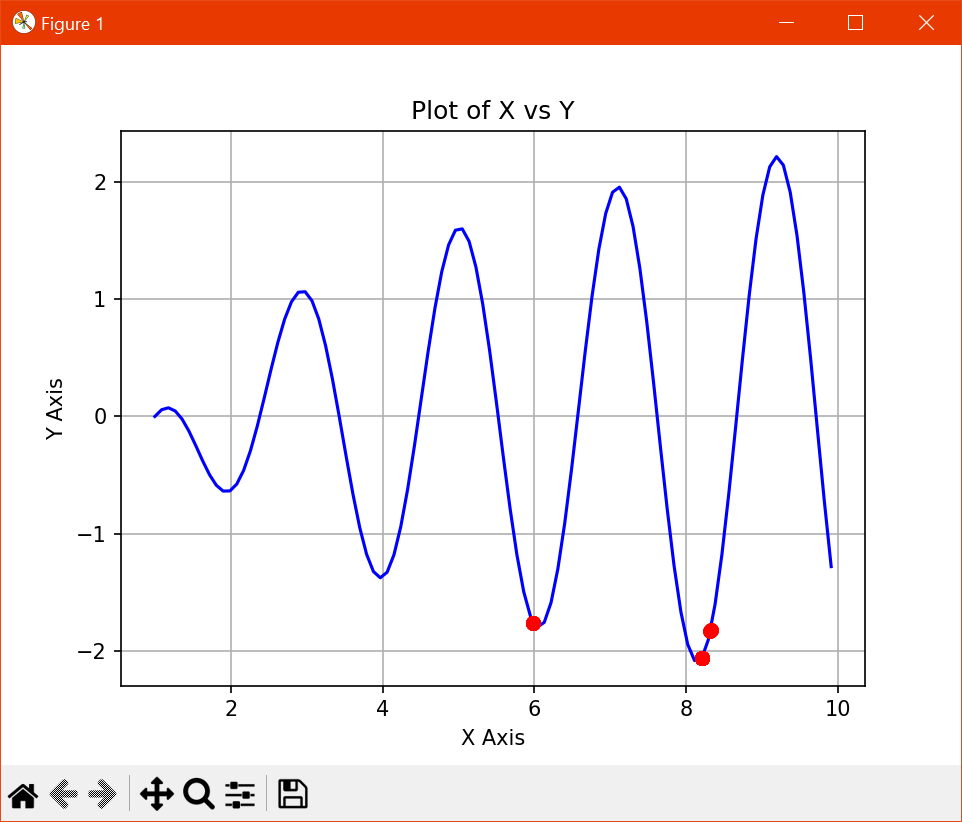
return 0;

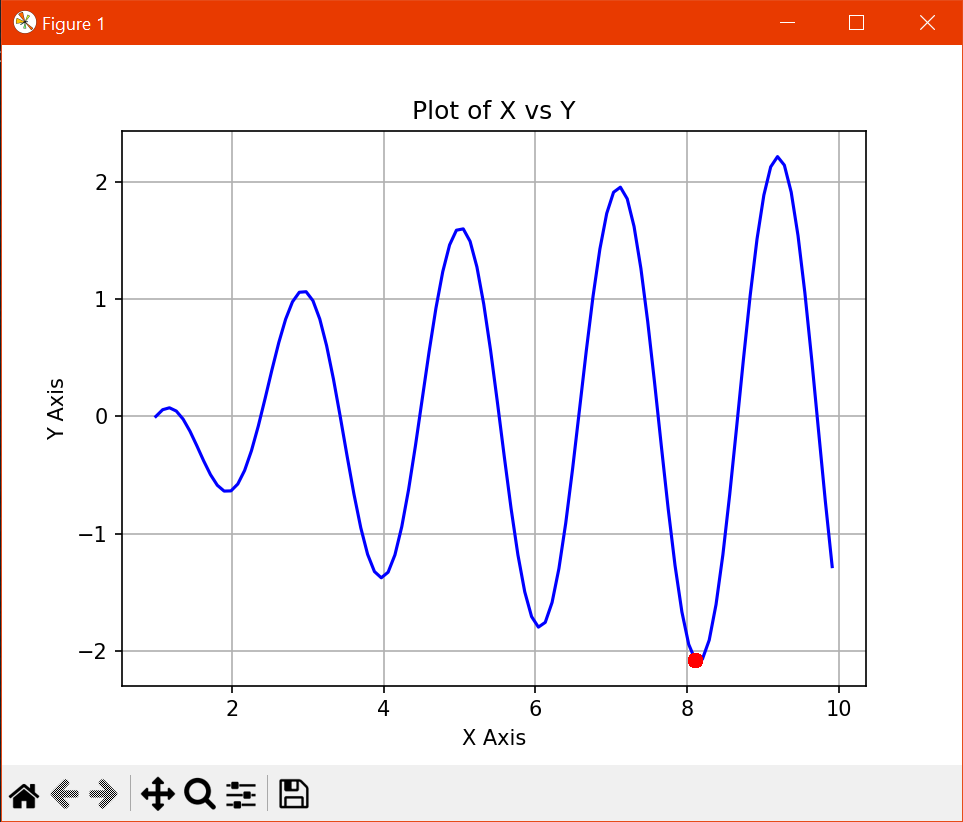
}

}

**Результаты выполнения (100 особей и 10, 50 поколений):**

Начальное положение особей

  
Положение через 10 эпох



Положение через 50 эпох

**Ответ на контрольный вопрос:**

**10.Придумайте другую реализацию ОМ.**

Оператор мутации в генетических алгоритмах можно реализовать не инверсией случайного бита, а, например, смещением всего слова (побитового сдвига) влево или вправо на случайное число (<длины хромосомы). Или же можно перевернуть строку, получив другую хромосому.