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 №8 |
| Оценка сложности программных проектов |
| по курсу: ЭВОЛЮЦИОННЫЕ МЕТОДЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНО-ИНФОРМАЦИОННЫХ СИСТЕМ |
|  |
|  |

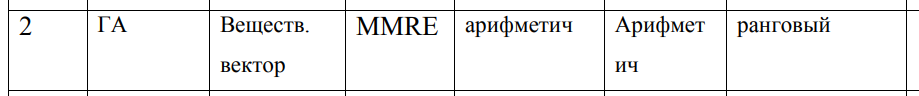
РАБОТУ ВЫПОЛНИЛ

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

Санкт-Петербург 2024

**Индивидуальное задание по варианту:**

22 (2)



**Цель работы:**

Разработка эволюционного алгоритма оценки стоимости программных проектов. Графическое отображение результатов.

**Теоретические сведения:**

Одной из самых популярных является модель COCOMO (Constructive Cost Model), предложенная Боэмом [12]. Эта модель разработана на основе фактической статистики 63 программных проектов НАСА путем сбора и анализа информации об их реализации. В результате выведены формулы, которые наилучшим образом аппроксимируют полученные экспериментальные данные.

В последнее десятилетие при решении задач оптимизации все шире используются новые методы, фактически примыкающие к эволюционным вычислениям по своей сути и основанные на моделировании социального поведения живых организмов. К ним относятся прежде всего роевые алгоритмы (Particle Swarm Optimization – PSO), которые в основном применяются в численной оптимизации, и муравьиные алгоритмы (Ant Colony Optimization – ACO), применяемые, как правило, при решении задач комбинаторной оптимизации (преимущественно на графах).

Роевые алгоритмы (РА), как и эволюционные, используют популяцию особей – потенциальных решений проблемы и метод стохастической оптимизации, навеянный наблюдениями за социальным поведением птиц и рыб в стае, а также насекомых в рое. Как и в эволюционных алгоритмах, здесь начальная популяция потенциальных решений тоже генерируется случайным образом, и затем ищется субоптимальное решение проблемы в процессе выполнения РА. Первоначально в РА была предпринята попытка смоделировать поведение стаи птиц, способной внезапно и синхронно перегруппировываться и изменять направление полета при выполнении некоторой задачи

**Код программы:**

#ifndef LAB8\_HPP

#define LAB8\_HPP

#include <iostream>

#include <random>

#include <vector>

#include <string>

#include <cstdlib>

#include <ctime>

#include <algorithm>

#include "Random.hpp"

namespace lab8

{

bocchie::mark selection\_mark("selection"), crossover\_mark("crossover"), mutate\_mark("mutate"),

insert\_mark("population.insert"), EPOCH\_MARK("EPOCH"), list\_fitness\_mark("list fitness");

struct Entity

{

double a, b;

[[nodiscard]] double evaluate\_Ef(double L) const

{

return a + std::pow(L, b);

}

} bestEntity;

typedef std::vector<Entity> Population;

struct Project

{

struct Feature

{

double L; // В килостроках

} feature;

struct Target

{

double Ef; // Реальная стоимость

} target;

};

namespace dataset

{

std::vector<Project> projects\_dataset;

std::vector<Project> train\_set;

std::vector<Project> test\_set;

} // namespace dataset

namespace k

{

constexpr uint16\_t N = 1000;

struct P

{

double c, m;

};

constexpr double MUTATION\_RANGE = 0.1;

constexpr double MUTATION\_OFFSET = 0.05;

namespace cost

{

double MMRE(const std::vector<double> &a, const std::vector<double> &b)

{

if (a.size() != b.size() || a.empty()) {

throw std::invalid\_argument("Vectors sizes not equal or empty =(");

}

double totalRelativeError = 0.0;

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

totalRelativeError += std::abs((a[i] - b[i]) / a[i]);

}

return totalRelativeError / (double) a.size();

}

} // namespace cost

double fitness(const Entity &entity, const std::vector<Project> &set = dataset::train\_set)

{

std::vector<double> predictions, reality;

predictions.reserve(set.size());

reality.reserve(set.size());

for (const auto &project : set) {

predictions.push\_back(entity.evaluate\_Ef(project.feature.L));

reality.push\_back(project.target.Ef);

}

return cost::MMRE(predictions, reality);

}

std::vector<double> fitness(const Population &population, const std::vector<Project> &set = dataset::train\_set)

{

std::vector<double> out;

out.reserve(population.size());

for (const auto &entity : population)

out.emplace\_back(fitness(entity, set));

return out;

}

} // namespace k

namespace op

{

// Ранговый оператор селекции

Population selection(const Population &population, const std::vector<double> &fitness, double crossing\_rate)

{

Population selected\_population;

std::vector<std::pair<double, Entity>> ranked\_population;

// Ранжируем популяцию по фитнес-функции

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

ranked\_population.emplace\_back(fitness[i], population[i]);

}

// Сортируем популяцию по фитнес-функции

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

{

return a.first < b.first;

});

// Выбираем лучшие особи

for (size\_t i = 0; i < size\_t(population.size() \* crossing\_rate); ++i) {

selected\_population.push\_back(ranked\_population[i].second);

}

bestEntity = selected\_population.front();

return selected\_population;

}

// Арифметический оператор скрещивания

Population crossover(const Population &population)

{

Population new\_population;

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

if (i + 1 < population.size()) {

const Entity &parent1 = population[i];

const Entity &parent2 = population[i + 1];

Entity child1, child2;

double alpha = random::\_double(0.0, 1.0);

child1.a = alpha \* parent1.a + (1 - alpha) \* parent2.a;

child1.b = alpha \* parent1.b + (1 - alpha) \* parent2.b;

child2.a = (1 - alpha) \* parent1.a + alpha \* parent2.a;

child2.b = (1 - alpha) \* parent1.b + alpha \* parent2.b;

new\_population.push\_back(child1);

new\_population.push\_back(child2);

}

else {

new\_population.push\_back(population[i]);

}

}

return new\_population;

}

// Арифметический оператор мутации

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

{

Population mutated\_population = population;

for (auto &entity : mutated\_population) {

if (random::\_double(0.0, 1.0) < mutation\_rate) {

entity.a +=

random::\_double(0.0, 1.0) \* k::MUTATION\_RANGE - k::MUTATION\_OFFSET; // Мутация с небольшим изменением

entity.b +=

random::\_double(0.0, 1.0) \* k::MUTATION\_RANGE - k::MUTATION\_OFFSET; // Мутация с небольшим изменением

}

}

return mutated\_population;

}

} // namespace op

Entity \_main(Population population, k::P p, size\_t epochs = 1000, size\_t log\_epoch = 100)

{

for (size\_t epoch = 0; epoch < epochs; ++epoch)

{

EPOCH\_MARK.run([&]()

{

Population parents = selection\_mark.run(op::selection, population, list\_fitness\_mark.run([&](){return k::fitness(population);}), p.c);

Population childs = crossover\_mark.run(op::crossover,parents);

childs = mutate\_mark.run(op::mutate,childs, p.m);

insert\_mark.run([&](){return population.insert(population.end(), childs.begin(), childs.end());});

population = selection\_mark.run(op::selection, population, list\_fitness\_mark.run([&](){return k::fitness(population);}), (double)k::N / population.size());

if (log\_epoch != 0 && epoch % log\_epoch == 0)

std::cout << "Epoch " << epoch << ": best fitness (MMRE) = " << list\_fitness\_mark.run([&](){return k::fitness(bestEntity);}) << std::endl;

return true;

});

}

return bestEntity;

}

void init\_dataset()

{

// Данные из таблицы 8.2

{

using namespace dataset;

projects\_dataset.emplace\_back(Project::Feature{2.2}, Project::Target{8.4});

projects\_dataset.emplace\_back(Project::Feature{3.5}, Project::Target{10.8});

projects\_dataset.emplace\_back(Project::Feature{5.5}, Project::Target{18});

projects\_dataset.emplace\_back(Project::Feature{6.0}, Project::Target{24});

projects\_dataset.emplace\_back(Project::Feature{9.7}, Project::Target{25.2});

projects\_dataset.emplace\_back(Project::Feature{7.7}, Project::Target{31.2});

projects\_dataset.emplace\_back(Project::Feature{11.3}, Project::Target{36});

projects\_dataset.emplace\_back(Project::Feature{8.2}, Project::Target{36});

projects\_dataset.emplace\_back(Project::Feature{6.5}, Project::Target{42});

projects\_dataset.emplace\_back(Project::Feature{8.0}, Project::Target{42});

projects\_dataset.emplace\_back(Project::Feature{20.0}, Project::Target{48});

projects\_dataset.emplace\_back(Project::Feature{10.0}, Project::Target{48});

projects\_dataset.emplace\_back(Project::Feature{15.0}, Project::Target{48});

projects\_dataset.emplace\_back(Project::Feature{10.4}, Project::Target{50});

projects\_dataset.emplace\_back(Project::Feature{13.0}, Project::Target{60});

projects\_dataset.emplace\_back(Project::Feature{14.0}, Project::Target{60});

projects\_dataset.emplace\_back(Project::Feature{19.7}, Project::Target{60});

projects\_dataset.emplace\_back(Project::Feature{32.5}, Project::Target{60});

projects\_dataset.emplace\_back(Project::Feature{31.5}, Project::Target{60});

projects\_dataset.emplace\_back(Project::Feature{12.5}, Project::Target{62});

projects\_dataset.emplace\_back(Project::Feature{15.4}, Project::Target{70});

projects\_dataset.emplace\_back(Project::Feature{20.0}, Project::Target{72});

projects\_dataset.emplace\_back(Project::Feature{7.5}, Project::Target{72});

projects\_dataset.emplace\_back(Project::Feature{16.3}, Project::Target{82});

projects\_dataset.emplace\_back(Project::Feature{15.0}, Project::Target{90});

projects\_dataset.emplace\_back(Project::Feature{11.4}, Project::Target{98.8});

projects\_dataset.emplace\_back(Project::Feature{21.0}, Project::Target{107});

projects\_dataset.emplace\_back(Project::Feature{16.0}, Project::Target{114});

projects\_dataset.emplace\_back(Project::Feature{25.9}, Project::Target{117.6});

projects\_dataset.emplace\_back(Project::Feature{24.6}, Project::Target{117.6});

projects\_dataset.emplace\_back(Project::Feature{29.5}, Project::Target{120});

projects\_dataset.emplace\_back(Project::Feature{19.3}, Project::Target{155});

projects\_dataset.emplace\_back(Project::Feature{32.6}, Project::Target{170});

projects\_dataset.emplace\_back(Project::Feature{35.5}, Project::Target{192});

projects\_dataset.emplace\_back(Project::Feature{38.0}, Project::Target{210});

projects\_dataset.emplace\_back(Project::Feature{48.5}, Project::Target{239});

projects\_dataset.emplace\_back(Project::Feature{47.5}, Project::Target{252});

projects\_dataset.emplace\_back(Project::Feature{70.0}, Project::Target{278});

projects\_dataset.emplace\_back(Project::Feature{66.6}, Project::Target{300});

projects\_dataset.emplace\_back(Project::Feature{66.6}, Project::Target{352.8});

projects\_dataset.emplace\_back(Project::Feature{50.0}, Project::Target{370});

projects\_dataset.emplace\_back(Project::Feature{79.0}, Project::Target{400});

projects\_dataset.emplace\_back(Project::Feature{90.0}, Project::Target{450});

projects\_dataset.emplace\_back(Project::Feature{78.0}, Project::Target{571.4});

projects\_dataset.emplace\_back(Project::Feature{100.0}, Project::Target{215});

projects\_dataset.emplace\_back(Project::Feature{150.0}, Project::Target{324});

projects\_dataset.emplace\_back(Project::Feature{100.0}, Project::Target{360});

projects\_dataset.emplace\_back(Project::Feature{100.0}, Project::Target{360});

projects\_dataset.emplace\_back(Project::Feature{190.0}, Project::Target{420});

projects\_dataset.emplace\_back(Project::Feature{115.8}, Project::Target{480});

// Перемешиваем данные для случайного разделения

std::srand(static\_cast<unsigned>(std::time(nullptr)));

std::shuffle(projects\_dataset.begin(), projects\_dataset.end(), std::mt19937(std::random\_device()()));

// Разделяем на обучающую и тестовую выборки

size\_t test\_size = projects\_dataset.size() - 40;

size\_t train\_size = projects\_dataset.size() - test\_size;

train\_set.insert(train\_set.end(), projects\_dataset.begin(), projects\_dataset.begin() + train\_size);

test\_set.insert(test\_set.end(), projects\_dataset.begin() + train\_size, projects\_dataset.end());

};

}

void main()

{

init\_dataset();

Population population = []()

{

Population population;

population.reserve(k::N);

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

population.emplace\_back(random::\_double(0, 2), random::\_double(0, 2));

return population;

}();

auto best = \_main(population, k::P{0.7, 0.1}, 3000, 100);

std::cout << selection\_mark.to\_json<bocchie::accuracy::milliseconds>() << std::endl;

std::cout << crossover\_mark.to\_json<bocchie::accuracy::milliseconds>() << std::endl;

std::cout << mutate\_mark.to\_json<bocchie::accuracy::milliseconds>() << std::endl;

std::cout << insert\_mark.to\_json<bocchie::accuracy::milliseconds>() << std::endl;

std::cout << list\_fitness\_mark.to\_json<bocchie::accuracy::milliseconds>() << std::endl;

std::cout << EPOCH\_MARK.to\_json<bocchie::accuracy::milliseconds>() << std::endl;

std::cout << "Best entity have a = " << best.a << ", b = " << best.b << ".\n"

<< "Check on test set:\n";

for (const auto &project : dataset::test\_set)

std::cout << "L = "<< std::setw(7) << project.feature.L

<< "\tPredicted value = "<< std::setw(7) << best.evaluate\_Ef(project.feature.L)

<< "\tReality value = " << std::setw(7) << project.target.Ef << std::endl;

std::cout << "Fitness (MMRE) on test set = " << k::fitness(best, dataset::test\_set) << std::endl;

}

} // namespace lab8

#endif //LAB8\_HPP

**Результаты выполнения:**

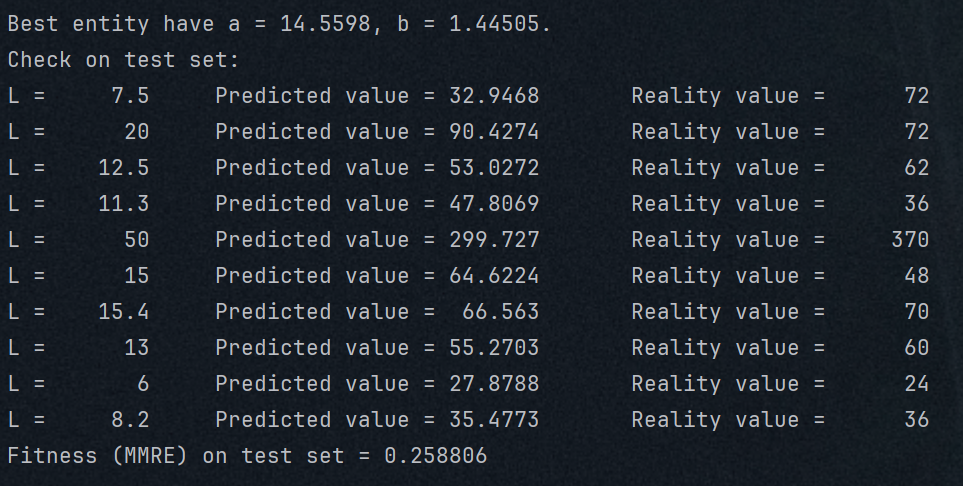


Рис. 1 – Результат обучения модели

**Ответ на контрольный вопрос:**

**10.** **Приведите основную формулу метрики абсолютных значений**

