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 №3 |
| Генетический алгоритм решения задачи коммивояжера |
| по курсу: ЭВОЛЮЦИОННЫЕ МЕТОДЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНО-ИНФОРМАЦИОННЫХ СИСТЕМ |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

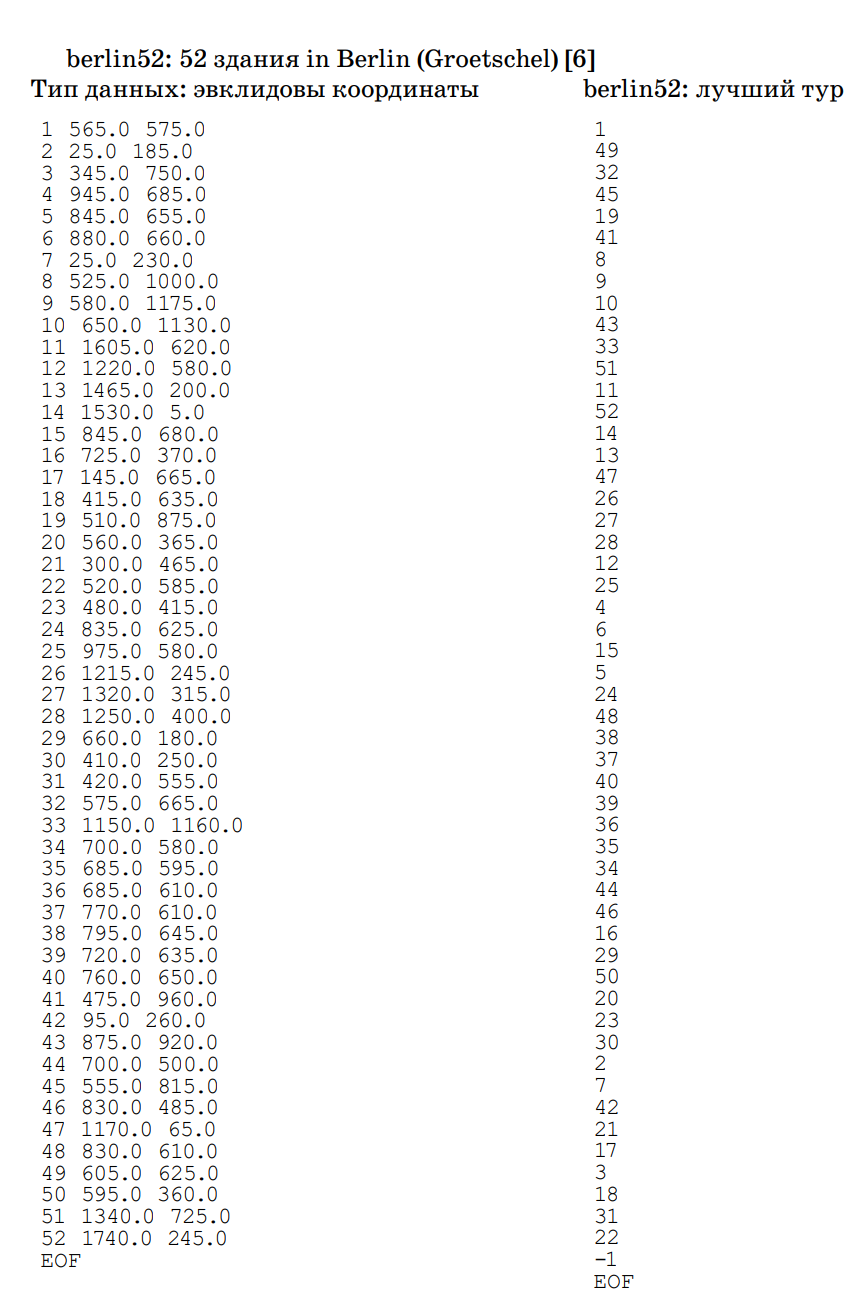
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 4136 |  |  |  | Якшин С. Е. |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2024

**Индивидуальное задание по варианту:**

Вариант 22

IMG_256



**Цель работы:**

Решение задач комбинаторной оптимизации с помощью генетических алгоритмов на примере задачи коммивояжера. Графическое отображение результатов оптимизации.

**Теоретические сведения:**

Задача коммивояжера (ЗК) считается классической задачей генетических алгоритмов. Она заключается в следующем: путешественник (или коммивояжер) должен посетить каждый из базового

набора городов и вернуться к исходной точке. Имеется стоимость

проезда из одного города в другой. Необходимо составить план путешествия, чтобы сумма затраченных средств была минимальной.

Поисковое пространство для ЗК-множество из N городов. Любая

комбинация из N городов, где города не повторяются, является решением. Оптимальное решение – такая комбинация, стоимость которой (сумма из стоимостей переезда между каждыми из городов в

комбинации) является минимальной.

ЗК – достаточно старая задача, она была сформулирована еще

в 1759 году (под другим именем). Термин «Задача коммивояжера»

был использован в 1932 г. в немецкой книге «The traveling salesman,

how and what he should to get commissions and be successful in his

business», написанную старым коммивояжером.

Задача коммивояжера была отнесена к NP-сложным задачам.

Существуют строгие ограничения на последовательность, и количество городов может быть очень большим (существуют тесты, включающие несколько тысяч городов).

Кажется естественным, что представление тура – последовательность (i1, i2, …, in), где (i1, i2, …, in) – числа из множества (1 …

n), представляющие определенный город. Двоичное представление

городов неэффективно, так как требует специального ремонтирующего алгоритма: изменение одиночного бита может повлечь неправильность тура.

В настоящее время существует три основных представления пути:

соседское, порядковое и путевое. Каждое из этих представлений имеет собственные полностью различные операторы рекомбинации.

**Код программы:**

#ifndef LAB3\_HPP

#define LAB3\_HPP

#include "Random.hpp"

namespace lab3

{

typedef std::pair<float, float> City;

typedef std::vector<City> Tour;

typedef std::vector<float> Fitness;

struct Entity

{

Tour tour;

};

typedef std::vector<Entity> Population;

namespace k

{

const size\_t EPOCH\_CRITERIA = 1000;

const size\_t EPOCH\_LIMIT = 200'000;

const size\_t FLOAT\_EQ\_PRECISION = 10;

const uint16\_t N = 1000;

struct P

{

float \_CROSSING = 0.5; // Вероятность скрещивания

float \_MUTATION = 0.0001; // Вероятность мутации

};

// Функция для вычисления евклидова расстояния между двумя точками

float calculateDistance(const City& a, const City& b) {

float dx = a.first - b.first;

float dy = a.second - b.second;

return std::sqrt(dx \* dx + dy \* dy);

}

// Вычисление фитнеса для одного Entity (общая длина маршрута)

float fitness(const Tour& tour) {

if (tour.size() < 2) return 0.0f; // Если меньше двух городов, длина маршрута равна 0

float totalDistance = 0.0f;

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

const City& from = tour[i];

const City& to = tour[(i + 1) % tour.size()]; // Кольцевой маршрут

totalDistance += calculateDistance(from, to);

}

return totalDistance;

}

// Вычисление фитнеса для всей популяции

std::vector<float> fitness(const Population& population) {

std::vector<float> fitnessValues;

fitnessValues.reserve(population.size()); // Резервируем место для оптимизации

for (const Entity& entity : population) {

fitnessValues.push\_back(fitness(entity.tour)); // Вычисляем фитнес каждого Entity

}

return fitnessValues;

}

// Функция для сравнения двух чисел с плавающей запятой с точностью до n знаков

bool are\_equal\_with\_precision(float a, float b, int precision = FLOAT\_EQ\_PRECISION) {

return a == b;

// Степень 10 для проверки точности

float epsilon = std::pow(10.0f, -precision);

// Проверяем разницу между числами

return std::fabs(a - b) < epsilon;

}

} // namespace k

namespace op

{

Population selection(const Population &population,

const Fitness &fitness,

int num\_selected)

{

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);

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 offspring;

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

{

// Выбираем двух случайных родителей

const auto &parent1 = population[random::\_int(0, population.size() - 1)].tour;

const auto &parent2 = population[random::\_int(0, population.size() - 1)].tour;

// Создаём потомка, равного размеру родителей

Tour child(parent1.size());

std::fill(child.begin(), child.end(), City{-1, -1});

child.front() = parent1.front();

child.back() = parent1.back();

// Выбираем случайный промежуток для обмена, исключая первый и последний город

size\_t start = random::\_int(1, parent1.size() - 2); // Исключаем первый и последний города

size\_t end = random::\_int(start, parent1.size() - 2); // Исключаем первый и последний города

if (start > end)

std::swap(start, end);

// Копируем часть от первого родителя

for (size\_t j = start; j <= end; ++j)

{

child[j] = parent1[j];

}

// Заполняем оставшиеся города из второго родителя

for (size\_t j = 1; j < parent2.size() - 1; ++j) // Исключаем первый и последний города

{

if (std::find(child.begin(), child.end(), parent2[j]) == child.end())

{

auto it = std::find(child.begin(), child.end(), City{-1, -1});

if (it != child.end())

{

\*it = parent2[j];

}

}

}

offspring.push\_back({child});

}

return offspring;

}

// Мутация методом перестановки (swap mutation)

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

{

Population mutated\_population = population;

for (auto &entity : mutated\_population)

{

if (random::\_float(0, 1) < mutation\_rate)

{

size\_t idx1 = random::\_int(1, entity.tour.size() - 2);

size\_t idx2 = random::\_int(1, entity.tour.size() - 2);

std::swap(entity.tour[idx1], entity.tour[idx2]);

}

}

return mutated\_population;

}

} // namespace op

Entity createRandomEntity(std::vector<City> cityPool)

{

static std::random\_device rd;

static std::mt19937 gen(rd());

// Перемешиваем города

std::shuffle(cityPool.begin(), cityPool.end(), gen);

cityPool.push\_back(cityPool.front());

return Entity { cityPool };

}

Entity createRandomEntityWithFixedStart(const std::vector<City>& cityPool)

{

if (cityPool.empty())

throw std::invalid\_argument("City pool cannot be empty.");

static std::random\_device rd;

static std::mt19937 gen(rd());

// Извлекаем первый город

City fixedCity = cityPool.front();

// Копируем города (кроме первого) для случайного перемешивания

std::vector<City> shuffledCities(cityPool.begin() + 1, cityPool.end());

// Перемешиваем оставшиеся города

std::shuffle(shuffledCities.begin(), shuffledCities.end(), gen);

// Формируем тур

std::vector<City> tour = {fixedCity};

tour.insert(tour.end(), shuffledCities.begin(), shuffledCities.end());

tour.push\_back(fixedCity);

return Entity{tour};

}

Population createRandomPopulation(const std::vector<City>& cityPool, bool fixedStart = true)

{

Population out;

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

out.push\_back(fixedStart ? createRandomEntityWithFixedStart(cityPool) : createRandomEntity(cityPool));

return out;

};

void printTour(const Tour& cityPool, const Tour& tour) {

for (const auto& city : tour) {

// Ищем индекс города в пуле

auto it = std::find(cityPool.begin(), cityPool.end(), city);

if (it != cityPool.end()) {

// Индекс найден

int index = std::distance(cityPool.begin(), it);

std::cout << index + 1 << "-";

}

}

std::cout << std::endl; // Печатаем новую строку после завершения вывода

}

auto min = [](const Population& population)

{

return \*std::min\_element(population.begin(), population.end(),

[](const auto &a, const auto &b)

{

return k::fitness(a.tour) < k::fitness(b.tour);

});

};

Entity \_main(Population population, k::P p)

{

bool exit = false;

size\_t epoch\_counter = 0;

auto fine\_optimum = [&epoch\_counter](const Population &population) -> bool

{

static size\_t best\_optimum\_counter = 0;

static float last\_optimum = 0xFFFFFF;

float optimum = k::fitness(min(population).tour);

if (not k::are\_equal\_with\_precision(last\_optimum, optimum))

{

last\_optimum = optimum;

best\_optimum\_counter = 0;

}

else best\_optimum\_counter++;

return best\_optimum\_counter == k::EPOCH\_CRITERIA || epoch\_counter == k::EPOCH\_LIMIT;

};

for (size\_t i = 0; i < 10'000; ++i)

// while (not exit)

{

epoch\_counter++;

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

Population parents = op::selection(population, k::fitness(population), k::N \* 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, 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(population), k::N);

if (i % 1000 == 0)

{

std::cout << "Epoch " << i << ", tour length: " << k::fitness(min(population).tour) << std::endl;

// printTour(berlin52, min(population).tour);

plot::city\_graph(min(population).tour);

}

// exit = fine\_optimum(population);

}

std::cout << "Educated by " << epoch\_counter << " epochs\n";

return min(population);

}

void main()

{

std::vector<City> berlin52;

{

berlin52.emplace\_back(565.0, 575.0);

berlin52.emplace\_back(25.0, 185.0);

berlin52.emplace\_back(345.0, 750.0);

berlin52.emplace\_back(945.0, 685.0);

berlin52.emplace\_back(845.0, 655.0);

berlin52.emplace\_back(880.0, 660.0);

berlin52.emplace\_back(25.0, 230.0);

berlin52.emplace\_back(525.0, 1000.0);

berlin52.emplace\_back(580.0, 1175.0);

berlin52.emplace\_back(650.0, 1130.0);

berlin52.emplace\_back(1605.0, 620.0);

berlin52.emplace\_back(1220.0, 580.0);

berlin52.emplace\_back(1465.0, 200.0);

berlin52.emplace\_back(1530.0, 5.0);

berlin52.emplace\_back(845.0, 680.0);

berlin52.emplace\_back(725.0, 370.0);

berlin52.emplace\_back(145.0, 665.0);

berlin52.emplace\_back(415.0, 635.0);

berlin52.emplace\_back(510.0, 875.0);

berlin52.emplace\_back(560.0, 365.0);

berlin52.emplace\_back(300.0, 465.0);

berlin52.emplace\_back(520.0, 585.0);

berlin52.emplace\_back(480.0, 415.0);

berlin52.emplace\_back(835.0, 625.0);

berlin52.emplace\_back(975.0, 580.0);

berlin52.emplace\_back(1215.0, 245.0);

berlin52.emplace\_back(1320.0, 315.0);

berlin52.emplace\_back(1250.0, 400.0);

berlin52.emplace\_back(660.0, 180.0);

berlin52.emplace\_back(410.0, 250.0);

berlin52.emplace\_back(420.0, 555.0);

berlin52.emplace\_back(575.0, 665.0);

berlin52.emplace\_back(1150.0, 1160.0);

berlin52.emplace\_back(700.0, 580.0);

berlin52.emplace\_back(685.0, 595.0);

berlin52.emplace\_back(685.0, 610.0);

berlin52.emplace\_back(770.0, 610.0);

berlin52.emplace\_back(795.0, 645.0);

berlin52.emplace\_back(720.0, 635.0);

berlin52.emplace\_back(760.0, 650.0);

berlin52.emplace\_back(475.0, 960.0);

berlin52.emplace\_back(95.0, 260.0);

berlin52.emplace\_back(875.0, 920.0);

berlin52.emplace\_back(700.0, 500.0);

berlin52.emplace\_back(555.0, 815.0);

berlin52.emplace\_back(830.0, 485.0);

berlin52.emplace\_back(1170.0, 65.0);

berlin52.emplace\_back(830.0, 610.0);

berlin52.emplace\_back(605.0, 625.0);

berlin52.emplace\_back(595.0, 360.0);

berlin52.emplace\_back(1340.0, 725.0);

berlin52.emplace\_back(1740.0, 245.0);

};

Tour best\_tour;

{

best\_tour.emplace\_back(berlin52[1 - 1]);

best\_tour.emplace\_back(berlin52[49 - 1]);

best\_tour.emplace\_back(berlin52[32 - 1]);

best\_tour.emplace\_back(berlin52[45 - 1]);

best\_tour.emplace\_back(berlin52[19 - 1]);

best\_tour.emplace\_back(berlin52[41 - 1]);

best\_tour.emplace\_back(berlin52[8 - 1]);

best\_tour.emplace\_back(berlin52[9 - 1]);

best\_tour.emplace\_back(berlin52[10 - 1]);

best\_tour.emplace\_back(berlin52[43 - 1]);

best\_tour.emplace\_back(berlin52[33 - 1]);

best\_tour.emplace\_back(berlin52[51 - 1]);

best\_tour.emplace\_back(berlin52[11 - 1]);

best\_tour.emplace\_back(berlin52[52 - 1]);

best\_tour.emplace\_back(berlin52[14 - 1]);

best\_tour.emplace\_back(berlin52[13 - 1]);

best\_tour.emplace\_back(berlin52[47 - 1]);

best\_tour.emplace\_back(berlin52[26 - 1]);

best\_tour.emplace\_back(berlin52[27 - 1]);

best\_tour.emplace\_back(berlin52[28 - 1]);

best\_tour.emplace\_back(berlin52[12 - 1]);

best\_tour.emplace\_back(berlin52[25 - 1]);

best\_tour.emplace\_back(berlin52[4 - 1]);

best\_tour.emplace\_back(berlin52[6 - 1]);

best\_tour.emplace\_back(berlin52[15 - 1]);

best\_tour.emplace\_back(berlin52[5 - 1]);

best\_tour.emplace\_back(berlin52[24 - 1]);

best\_tour.emplace\_back(berlin52[48 - 1]);

best\_tour.emplace\_back(berlin52[38 - 1]);

best\_tour.emplace\_back(berlin52[37 - 1]);

best\_tour.emplace\_back(berlin52[40 - 1]);

best\_tour.emplace\_back(berlin52[39 - 1]);

best\_tour.emplace\_back(berlin52[36 - 1]);

best\_tour.emplace\_back(berlin52[35 - 1]);

best\_tour.emplace\_back(berlin52[34 - 1]);

best\_tour.emplace\_back(berlin52[44 - 1]);

best\_tour.emplace\_back(berlin52[46 - 1]);

best\_tour.emplace\_back(berlin52[16 - 1]);

best\_tour.emplace\_back(berlin52[29 - 1]);

best\_tour.emplace\_back(berlin52[50 - 1]);

best\_tour.emplace\_back(berlin52[20 - 1]);

best\_tour.emplace\_back(berlin52[23 - 1]);

best\_tour.emplace\_back(berlin52[30 - 1]);

best\_tour.emplace\_back(berlin52[2 - 1]);

best\_tour.emplace\_back(berlin52[7 - 1]);

best\_tour.emplace\_back(berlin52[42 - 1]);

best\_tour.emplace\_back(berlin52[21 - 1]);

best\_tour.emplace\_back(berlin52[17 - 1]);

best\_tour.emplace\_back(berlin52[3 - 1]);

best\_tour.emplace\_back(berlin52[18 - 1]);

best\_tour.emplace\_back(berlin52[31 - 1]);

best\_tour.emplace\_back(berlin52[22 - 1]);

best\_tour.emplace\_back(berlin52[1 - 1]);

}

std::cout << "Best tour length: " << k::fitness(best\_tour) << std::endl;

printTour(berlin52, best\_tour);

plot::city\_graph(best\_tour);

auto population = createRandomPopulation(berlin52);

std::cout << "Begin best tour length: " << k::fitness(min(population).tour) << std::endl;

printTour(berlin52, min(population).tour);

plot::city\_graph(min(population).tour);

auto result = \_main(population, k::P {0.7, 0.01});

std::cout << "Final best tour length: " << k::fitness(result.tour) << std::endl;

printTour(berlin52, result.tour);

plot::city\_graph(min(population).tour);

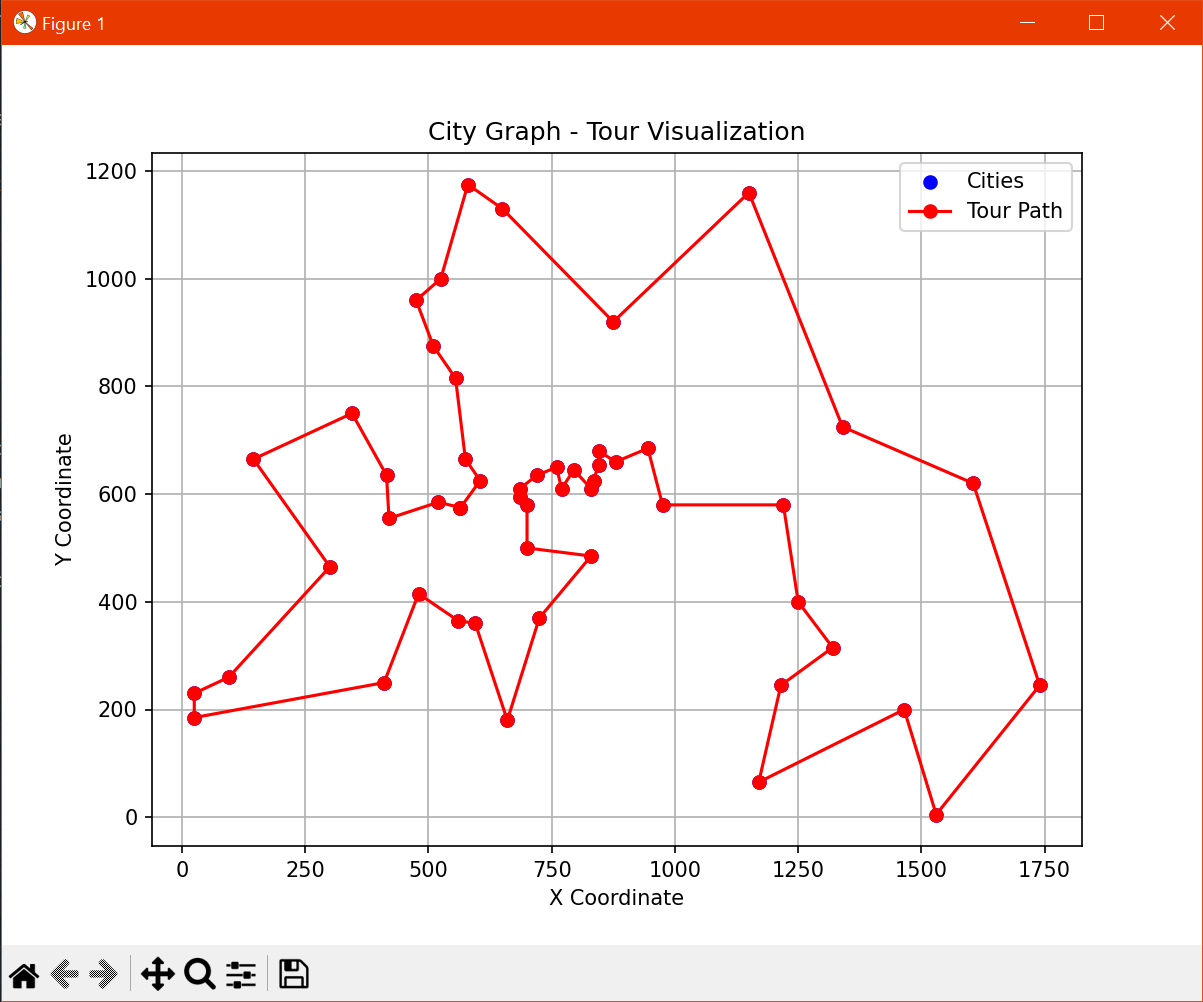
}

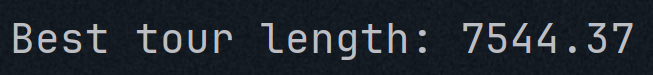
} // namespace lab3

#endif //LAB3\_HPP

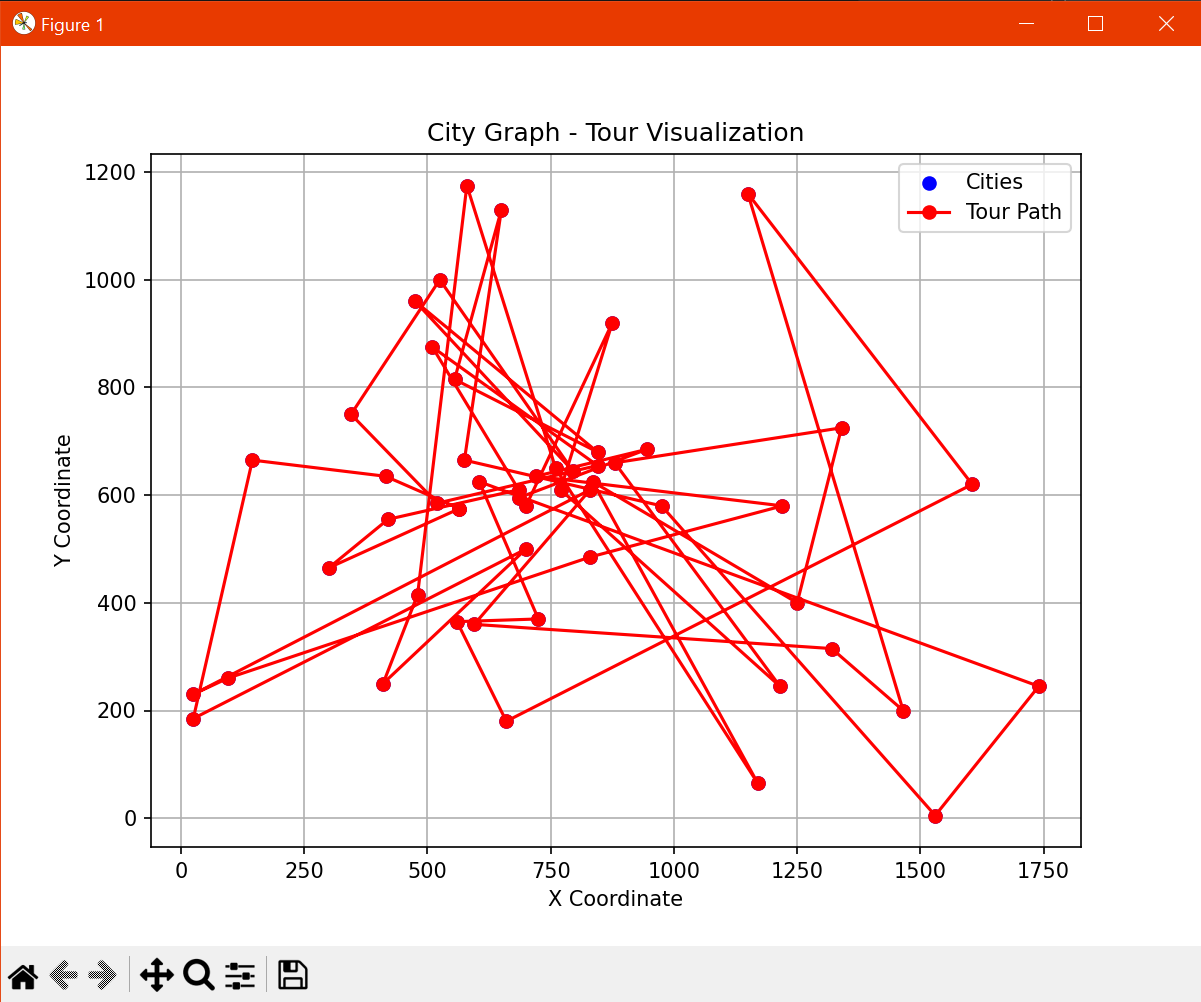
**Результаты выполнения:**

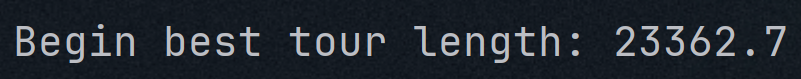
**Лучший тур по условию:**





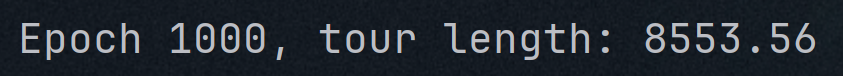
**Лучший тур из 1000 случайных особей:**



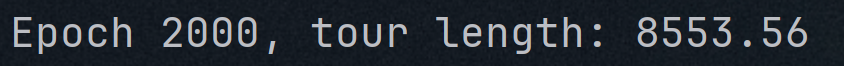


Лучший тур после 1000 эпох:



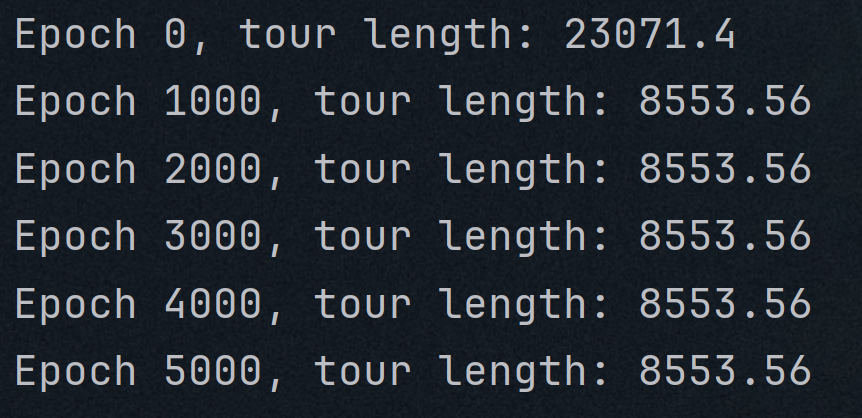


Лучший тур через 2000 эпох:  

**Экспериментальные изменения:**

Приведенные выше результаты были получены при вероятности скрещивания = 0.7 и мутации = 0.01, то есть с достаточно высокими вероятностями. Популяция обучалась заданное количество эпох, и смогла быстро достугнуть некоторого оптимума:



Однако не самого лучшего. При обучении по критерию повторения лучшей особи N раз результаты были еще хуже - обучение было быстрым, но результат далек от оптимального

(при критерии в 500 повторений обучение длилось 600 - 800 эпох, а результат (длина тура) был в среднем около 13 000. При критерии в 1000 повторений обучение длилось примерно 2500 эпох, а результат был около 9500 ). Думаю можно попробовать увеличить разнообразие популяции, чтобы находить новые оптимумы, если популяция уткнулась в некий локальный минимум.

**Ответ на контрольный вопрос:**

**10.Почему неэффективно двоичное кодирование хромосомы при решении**

**задачи коммивояжера?**Двоичное представление городов неэффективно, т.к. требует специального ремонтирующего алгоритма: изменение одиночного бита может повлечь неправильность тура.