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 №6 |
| Оптимизация путей на графах с помощью муравьиных алгоритмов |
| по курсу: ЭВОЛЮЦИОННЫЕ МЕТОДЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНО-ИНФОРМАЦИОННЫХ СИСТЕМ |
|  |
|  |

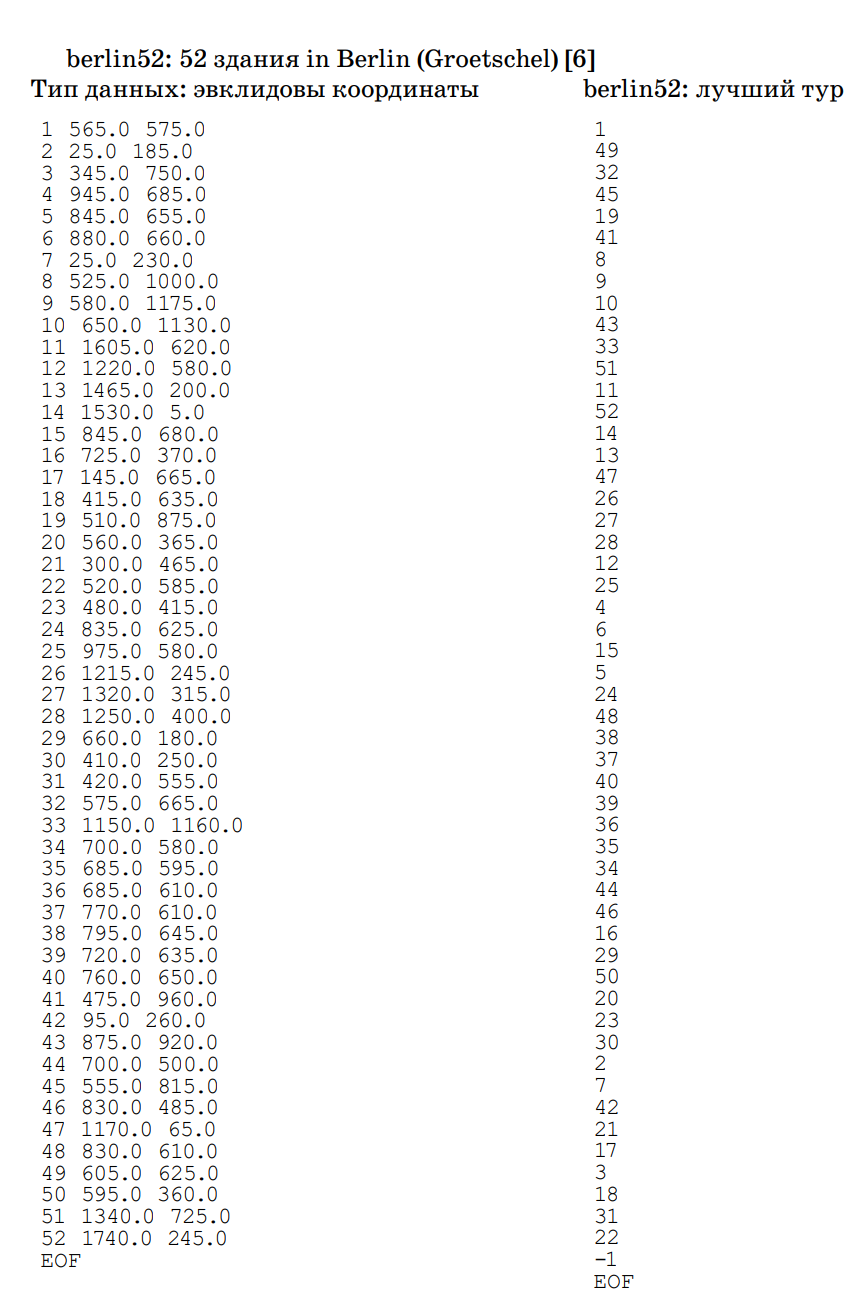
РАБОТУ ВЫПОЛНИЛ

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

Санкт-Петербург 2024

**Индивидуальное задание по варианту:**

Вариант 22



**Цель работы:**

Решение задач комбинаторной оптимизации с помощью муравьиных алгоритмов на примере задачи коммивояжера.

Графическое отображение результатов оптимизации.

**Теоретические сведения:**

Муравьиные алгоритмы (МА) основаны на использовании популяции потенциальных решений и разработаны для решения задач комбинаторной оптимизации, прежде всего, поиска различных

путей на графах. Кооперация между особями (искусственными

муравьями) здесь реализуется на основе моделирования. При этом

каждый агент, называемый искусственным муравьем, ищет решение поставленной задачи. Искусственные муравьи последовательно

строят решение задачи, передвигаясь по графу, откладывают феромон и при выборе дальнейшего участка пути учитывают концентрацию этого фермента. Чем больше концентрация феромона в последующем участке, тем больше вероятность его выбора.

**Код программы:**

#ifndef LAB6\_HPP

#define LAB6\_HPP

#include <iostream>

#include <utility>

#include <vector>

#include <cmath>

#include <algorithm>

#include <random>

#include <numeric>

#include <set>

namespace lab6

{

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

typedef std::vector<City> Tour;

typedef uint16\_t CityNumber;

typedef std::map<CityNumber, City> CityMap;

typedef std::map<City, CityNumber> InvCityMap;

using CityPair = std::pair<CityNumber, CityNumber>; // Пара индексов городов

typedef std::map<CityPair, double> CityGraph; // Мапа для хранения расстояний / флогистона между городами

struct Ant {

Tour tour;

float tourLength;

};

typedef std::vector<Ant> Population;

namespace k

{

const size\_t N = 1000;

const float ALPHA = 1.0; // Влияние феромона

const float BETA = 4.0; // Влияние эвристики

const float RHO = 0.6; // Испарение феромона

const float Q = 1.0; // Количество феромона, оставляемого муравьем

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);

}

float fitness(const Tour& tour) {

if (tour.size() < 2) return 0.0f;

float totalDistance = 0.0f;

for (size\_t i = 0; i < tour.size() - 1; ++i) {

const City& from = tour[i];

const City& to = tour[i + 1];

totalDistance += calculateDistance(from, to);

}

return totalDistance;

}

float fitness(const Ant& ant)

{

return fitness(ant.tour);

}

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

{

std::vector<float> out;

for (const auto& ant: population)

out.push\_back(fitness(ant));

return out;

}

} // namespace k

class AntColonyOptimization {

public:

AntColonyOptimization(CityMap cityMap,

InvCityMap cityInvMap,

CityGraph pheromoneGraph,

CityGraph heuristicGraph,

CityGraph distanceGraph,

size\_t numAnts, size\_t maxIterations)

: cityMap(std::move(cityMap)),

cityInvMap(std::move(cityInvMap)),

pheromoneGraph(std::move(pheromoneGraph)),

heuristicGraph(std::move(heuristicGraph)),

distanceGraph(std::move(distanceGraph)),

numAnts(numAnts),

maxIterations(maxIterations),

bestTourLength(std::numeric\_limits<float>::max()) {

ants.resize(numAnts);

}

void run() {

const size\_t numCities = cityMap.size();

std::random\_device rd;

std::mt19937 gen(rd());

for (size\_t iteration = 0; iteration < maxIterations; ++iteration) {

placeAnts(gen);

constructTours(gen);

updatePheromones();

if (iteration % 10 == 0)

printIterationResult(iteration);

}

printFinalResult();

}

private:

CityMap cityMap;

InvCityMap cityInvMap;

CityGraph pheromoneGraph;

CityGraph heuristicGraph;

CityGraph distanceGraph;

size\_t numAnts;

size\_t maxIterations;

Population ants;

Tour bestTour;

float bestTourLength;

void placeAnts(std::mt19937& gen) {

const size\_t numCities = cityMap.size();

for (auto& ant : ants) {

ant.tour.clear();

ant.tourLength = 0.0f;

ant.tour.push\_back(cityMap.at(1));

}

}

void constructTours(std::mt19937& gen) {

for (auto& ant : ants) {

std::set<CityNumber> visited;

visited.insert(ant.tour.front().first);

while (ant.tour.size() < cityMap.size()) {

CityNumber currentCity = cityInvMap.at(ant.tour.back());

chooseNextCity(ant, visited, currentCity, gen);

localPheromoneUpdate(ant); // Локальное обновление феромонов

}

ant.tour.push\_back(cityMap.at(1));

ant.tourLength = k::fitness(ant);

updateBestTour(ant);

}

}

void chooseNextCity(Ant& ant, std::set<CityNumber>& visited, CityNumber currentCity, std::mt19937& gen) {

std::vector<std::pair<CityNumber, float>> probabilities;

double probabilitySum = 0.0f;

for (const auto& [nextCity, \_] : cityMap) {

if (visited.find(nextCity) != visited.end()) continue;

float pheromone = pheromoneGraph.at({currentCity, nextCity});

float heuristic = heuristicGraph.at({currentCity, nextCity});

double probability = std::pow(pheromone, k::ALPHA) \* std::pow(heuristic, k::BETA);

probabilities.emplace\_back(nextCity, probability);

probabilitySum += probability;

}

float randomValue = random::\_double(0, probabilitySum);

CityNumber nextCity = 0;

for (const auto& [city, probability] : probabilities) {

randomValue -= probability;

if (randomValue <= 0) {

nextCity = city;

break;

}

}

visited.insert(nextCity);

ant.tour.push\_back(cityMap.at(nextCity));

}

void localPheromoneUpdate(const Ant& ant) {

const float localRho = 0.1; // Локальный коэффициент испарения

for (size\_t i = 0; i < ant.tour.size() - 1; ++i) {

CityNumber from = ant.tour[i].first;

CityNumber to = ant.tour[i + 1].first;

pheromoneGraph[{from, to}] \*= (1 - localRho);

pheromoneGraph[{to, from}] \*= (1 - localRho);

}

}

void updateBestTour(const Ant& ant) {

if (ant.tourLength < bestTourLength) {

bestTour = ant.tour;

bestTourLength = ant.tourLength;

}

}

void updatePheromones() {

for (auto& [edge, pheromone] : pheromoneGraph) {

pheromone \*= (k::RHO);

}

for (const auto& ant : ants) {

float pheromoneDeposit = k::Q / ant.tourLength;

for (size\_t i = 0; i < ant.tour.size() - 1; ++i) {

CityNumber from = ant.tour[i].first;

CityNumber to = ant.tour[i + 1].first;

pheromoneGraph[{from, to}] += pheromoneDeposit;

pheromoneGraph[{to, from}] += pheromoneDeposit;

}

}

}

void printIterationResult(size\_t iteration) const {

std::cout << "Iteration " << iteration + 1 << ": Best length so far = " << bestTourLength << std::endl;

}

void printFinalResult() const {

std::cout << "Final best tour length: " << bestTourLength << std::endl;

std::cout << "END" << std::endl;

plot::city\_graph(bestTour);

}

};

int 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);

};

CityMap berlin52\_map = [] (const std::vector<City>& \_)

{

CityMap \_\_;

for (size\_t number = 1; number <= \_.size(); ++number)

\_\_[number] = \_[number - 1];

return \_\_;

}(berlin52);

InvCityMap berlin52\_invmap = [] (const std::vector<City>& \_)

{

InvCityMap \_\_;

for (size\_t number = 1; number <= \_.size(); ++number)

\_\_[\_[number - 1]] = number;

return \_\_;

}(berlin52);

CityGraph distanceGraph = [](const CityMap& \_)

{

CityGraph \_\_;

for (const auto& col: \_)

for (const auto& line: \_)

\_\_[{col.first, line.first}] = k::calculateDistance(col.second, line.second);

return \_\_;

}(berlin52\_map);

CityGraph pheromoneGraph = [&distanceGraph](const CityMap& \_)

{

CityGraph \_\_;

for (const auto& col: \_)

for (const auto& line: \_)

\_\_[{col.first, line.first}] = random::\_float(0.001, 0.01);

return \_\_;

}(berlin52\_map);

CityGraph heuristicGraph = [&distanceGraph](const CityMap& \_)

{

CityGraph \_\_;

for (const auto& col: \_)

for (const auto& line: \_)

{

auto dist = distanceGraph[{col.first, line.first}];

\_\_[{col.first, line.first}] = dist == 0 ? 0 : 1 / distanceGraph[{col.first, line.first}];

}

return \_\_;

}(berlin52\_map);

AntColonyOptimization aco(berlin52\_map, berlin52\_invmap, pheromoneGraph, heuristicGraph, distanceGraph, k::N, 100);

aco.run();

return 0;

}

}

#endif //LAB6\_HPP

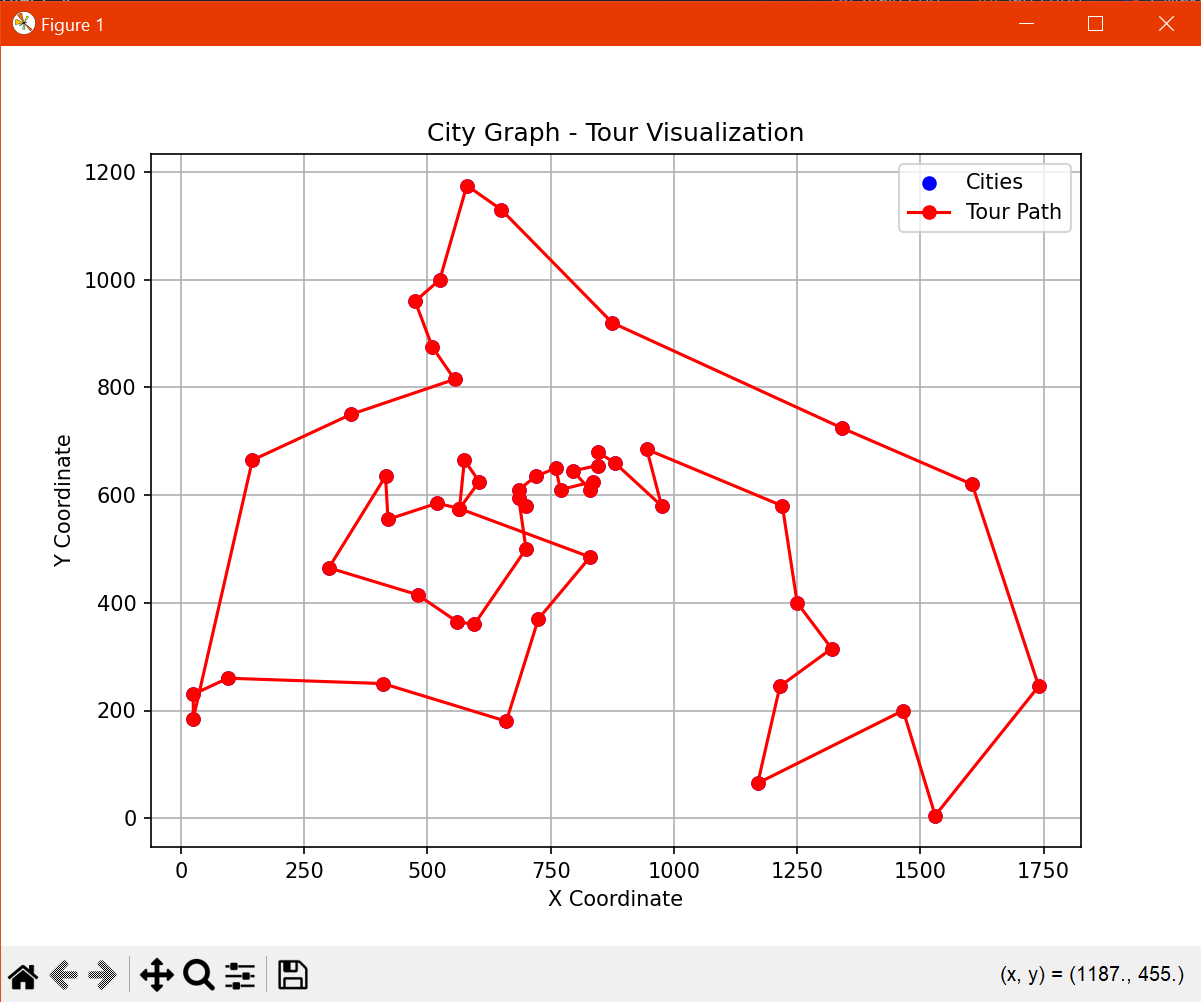
**Результаты выполнения:**

Начальная генерация:

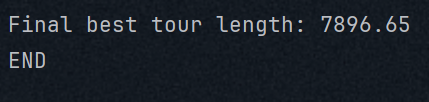




Лучший путь в конце работы алгоритма:



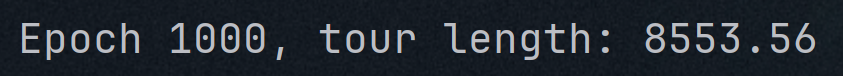
Длина лучшего маршрута:

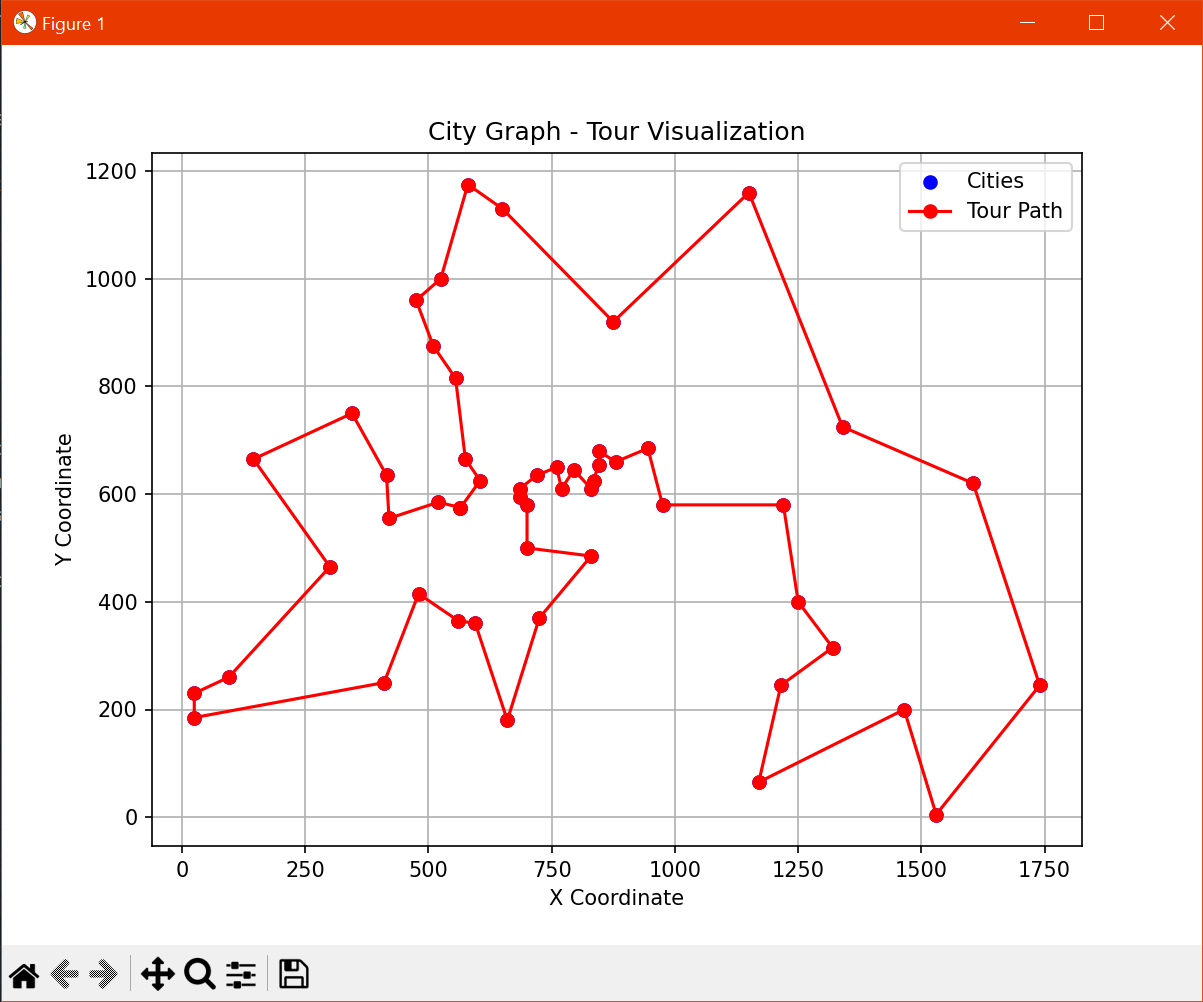


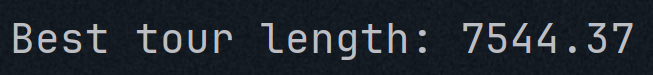
Результаты 3 лабораторной работы:



Длина:



Оптимальный тур по условию:  




Можно сделать вывод, что муравьиный алгоритм работает лучше, так как найденный путь короче, в сравнении с решением задачи через генетические алгоритмы. Также выполнение программы по времени у муравьиного алгоритма быстрее.

**Ответ на контрольный вопрос:**

**10.**  **Что общего между генетическими и муравьиными алгоритмами.**

 **Поиск в пространстве решений**: И ГА, и МА стремятся найти наилучшее решение в большом пространстве возможных решений. В обоих случаях решения постепенно улучшаются через итерации, используя эволюционные принципы.

 **Многократное обновление популяции**: В обоих алгоритмах используется популяция (в ГА это популяция индивидуумов, в МА — муравьев), которая проходит через несколько итераций. Каждый муравей или индивидуум в популяции генерирует решение, и это решение обновляется в зависимости от результатов предыдущих шагов.

 **Использование случайности**: Оба алгоритма применяют случайные элементы для генерации новых решений. В МА это случайный выбор следующего пути для муравья, а в ГА — случайное мутационное изменение или кроссовер между хромосомами.