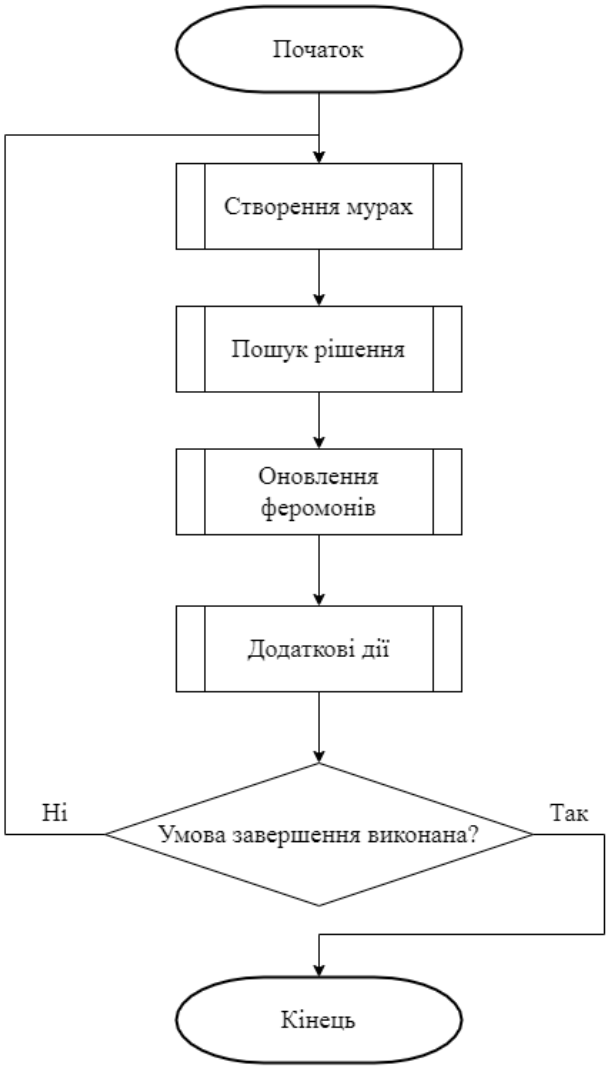
ДОДАТОК А
ГРАФІЧНІ МАТЕРІАЛИ

ГЮИК.506120.001

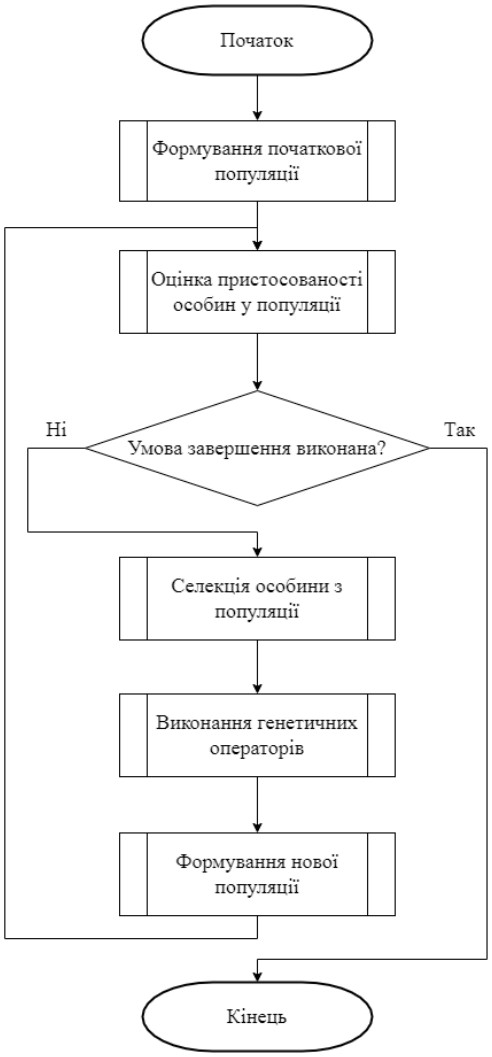
БЛОК-СХЕМА МУРАШИНОГО АЛГОРИТМУ



					ГЮИК.506120.001	
Змін.	*Арк*	*№ докум.*	*Підпис*	*Дата*		
Разраб.		*Бронза Є.С.*			Блок-схема мурашиного алгоритму	Літера: — / Арк.: *1* / Аркушів: *1*
Перевірив.		*Імангулова З.А.*				*Лист 1* / *Листів 1*
Т. Контр.						*ХНУРЭ*
Н. Контр.		*Імангулова З.А.*				*Кафедра СТ*
Реценз.						
Затвер..		*Гребеннік І.В.*				

ГЮИК.506120.001

БЛОК-СХЕМА ГЕНЕТИЧНОГО АЛГОРИТМУ



					ГЮИК.506120.001		
Змін.	Арк	№ докум.	Підпис	Дата			
Разраб.		Бронза Є.С.			Блок-схема генетичного алгоритму	Літера	Арк. 1 / Аркушів 1
Перевірив.		Імангулова З.А.					
Т. Контр.						Лист 1	Листів 1
Н. Контр.		Імангулова З.А.				ХНУРЭ Кафедра СТ	
Реценз.							
Затвер..		Гребеннік І.В.					

ДОДАТОК Б
ТЕКСТ ПРОГРАМИ

Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

«ЗАТВЕРДЖУЮ»
Керівник атестаційної роботи
_____ доц. Імангулова. З.А.
(підпис)
«____»_____2021 р.

ДОСЛІДЖЕННЯ ТА РОЗРОБКА МЕТОДІВ МАРШРУТИЗАЦІЇ ТРАНСПОРТНИХ ЗАСОБІВ В СИСТЕМІ МЕРЕЖЕВОГО РІТЕЙЛУ

Текст програми
ЛИСТ ЗАТВЕРДЖЕННЯ
ГЮИК.506120.001 01 12 01 -ЛЗ;

Виконавець:
студент групи СПРм-20-1
Бронза Євген Семенович
«___» _____ 2021 р.

2021

Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Затверджено

ГЮИК.506120.001 01 12 01

ДОСЛІДЖЕННЯ ТА РОЗРОБКА МЕТОДІВ МАРШРУТИЗАЦІЇ ТРАНСПОРТНИХ ЗАСОБІВ В СИСТЕМІ МЕРЕЖЕВОГО РІТЕЙЛУ

Текст програми

ГЮИК.506120.001 01 12 01

Листів _42_

2021

ЗМІСТ

Б.1 Формування плану постачання

```
const double coeff_A = 0.4;
const double coeff_B = 0.15f;
const double coeff_C = 0.45;
for (size_t j = 0; j < pointsAmount; ++j) {
    std::pair<int, std::vector<int>> point_plan;
    std::vector<int> plan;
    point_plan.first = j + 1;
    for (size_t i = 0; i < productsAmount; ++i) {
        a = coeff_A *
(static_cast<float>(last_month_history[j].second[i]) /
static_cast<float>(standart_plan[i]));
        b = coeff_B *
(static_cast<float>(last_month_history[j].second[i]) /
static_cast<float>(last_month_year_history[j].second[i]));
        c = coeff_C *
(static_cast<float>(next_month_history[j].second[i]) /
static_cast<float>(standart_plan[i]));
        count = static_cast<int>(standart_plan[i] * (a + b + c)) +
1;
        plan.push_back(count);
    }
    point_plan.second = plan;
    computed_plan.push_back(point_plan);
}
```

Б.2 Підготовка даних доріг

```
std::vector<std::vector<int>> points_array;
std::vector<std::vector<int>> roads_array;
int getPointIndexByID(const int id);
int main() {
    std::ifstream points("../data/map_points.txt");
    std::ifstream roads("../data/map_roads.txt");
    std::ofstream
computed_roads("../computed_data/computed_map_roads.txt");
    std::ofstream telemetry_out("../output/telemetry.txt",
std::ios::out | std::ios::app);
```

```
    std::string line, token;
    if (points.is_open()) {
        while (getline(points, line)) {
            std::stringstream ss;
            std::vector<int> array;
            ss << line;
            while (!ss.eof()) {
                ss >> token;
                array.push_back(std::stoi(token));
            }
            points_array.push_back(array);
        }
    }
    if (roads.is_open()) {
        while (getline(roads, line)) {
            std::stringstream ss;
            std::vector<int> array;
            ss << line;
            while (!ss.eof()) {
                ss >> token;
                int point_idx =
getPointIndexByID(std::stoi(token));
                array.push_back(points_array[point_idx][1]);
                array.push_back(points_array[point_idx][2]);
            }
            roads_array.push_back(array);
        }
    }
    if (computed_roads.is_open()) {
        unsigned int id = 1;
        for (std::vector<int> road : roads_array) {
            computed_roads << id << " " << road[0] << " " <<
road[1] << " " << road[2] << " " << road[3] << "\n";
            id++;
        }
    }
    points.close();
    roads.close();
    computed_roads.close();
    return 0;
}
```

```
int getPointIndexByID(const int id) {
    int index;
    for (size_t i = 0; i < points_array.size(); ++i) {
        if (points_array[i][0] == id) {
            index = i;
        }
    }
    return index;
}
```

Б.3 Підготовкаданих до кластеризації

```
int main() {
    std::ifstream map_points_in("../data/map_points.txt");
    std::ofstream
computed("../computed_data/pure_map_points.txt");
    std::ofstream telemetry_out("../output/telemetry.txt",
std::ios::out | std::ios::app);
    std::string line, token;
    std::vector<std::pair<int, int>> points;
    if (map_points_in.is_open()) {
        getline(map_points_in, line); //skip warehouse
        while (getline(map_points_in, line)) {
            std::stringstream ss;
            std::pair<int, int> point;
            ss << line;
            ss >> token; // skip id
            ss >> token;
            point.first = std::stoi(token);
            ss >> token;
            point.second = std::stoi(token);
            points.push_back(point);
        }
    }
    map_points_in.close();
    if (computed.is_open()) {
        for (auto point : points) {
            computed << point.first << " " << point.second <<
"\n";
        }
```

```
    }
    computed.close();
    return 0;
}}
```

Б.4 Кластеризація методом k-середніх

```
int iters = 100;
KMeans kmeans(K, iters, output_dir);
kmeans.run(all_points);

class Point {
private:
    int pointId, clusterId;
    int dimensions;
    std::vector<double> values;
    std::vector<double> lineToVec(std::string &line) {
        std::vector<double> values;
        std::string tmp = "";
        for (int i = 0; i < static_cast<int>(line.length()); i++)
{
            if ((48 <= int(line[i]) && int(line[i]) <= 57) ||
line[i] == '.' || line[i] == '+' || line[i] == '-' || line[i] ==
'e') {
                tmp += line[i];
            }
            else if (tmp.length() > 0) {
                values.push_back(stod(tmp));
                tmp = "";
            }
        }
        if (tmp.length() > 0) {
            values.push_back(stod(tmp));
            tmp = "";
        }
        return values;
    }
public:
    Point(int id, std::string line) {
        pointId = id;
```

```
        values = lineToVec(line);
        dimensions = values.size();
        clusterId = 0; // Initially not assigned to any cluster
    }
    int getDimensions() { return dimensions;}
    int getCluster() { return clusterId; }
    int getID() { return pointId; }
    void setCluster(int val) { clusterId = val; }
    double getVal(int pos) { return values[pos]; }
};

class Cluster {
private:
    int clusterId;
    std::vector<double> centroid;
    std::vector<Point> points;
public:
    Cluster(int clusterId, Point centroid) {
        this->clusterId = clusterId;
        for (int i = 0; i < centroid.getDimensions(); i++) {
            this->centroid.push_back(centroid.getVal(i));
        }
        this->addPoint(centroid);
    }
    void addPoint(Point p) {
        p.setCluster(this->clusterId);
        points.push_back(p);
    }
    bool removePoint(int pointId) {
        int size = points.size();
        for (int i = 0; i < size; i++) {
            if (points[i].getID() == pointId) {
                points.erase(points.begin() + i);
                return true;
            }
        }
        return false;
    }
    void removeAllPoints() { points.clear(); }
    int getId() { return clusterId; }
    Point getPoint(int pos) { return points[pos]; }
```

```
    int getSize() { return points.size(); }
    double getCentroidByPos(int pos) { return centroid[pos]; }
    void setCentroidByPos(int pos, double val) { this-
>centroid[pos] = val; }
};

class KMeans {
private:
    int K, iters, dimensions, total_points;
    std::vector<Cluster> clusters;
    std::string output_dir;
    void clearClusters() {
        for (int i = 0; i < K; i++) {
clusters[i].removeAllPoints(); }
    }
    int getNearestClusterId(Point point) {
        double sum = 0.0, min_dist;
        int NearestClusterId;
        if (dimensions == 1) {
            min_dist = abs(clusters[0].getCentroidByPos(0) -
point.getVal(0));
        }
        else {
            for (int i = 0; i < dimensions; i++) { sum +=
pow(clusters[0].getCentroidByPos(i) - point.getVal(i), 2.0); }
            min_dist = sqrt(sum);
        }
        NearestClusterId = clusters[0].getId();
        for (int i = 1; i < K; i++) {
            double dist;
            sum = 0.0;
            if (dimensions == 1) { dist =
abs(clusters[i].getCentroidByPos(0) - point.getVal(0)); }
            else {
                for (int j = 0; j < dimensions; j++) { sum +=
pow(clusters[i].getCentroidByPos(j) - point.getVal(j), 2.0); }
                dist = sqrt(sum);
            }
            if (dist < min_dist) {
                min_dist = dist;
                NearestClusterId = clusters[i].getId();
```

```
            }
        }
        return NearestClusterId;
    }
public:
    KMeans(int K, int iterations, std::string output_dir) {
        this->K = K;
        this->iters = iterations;
        this->output_dir = output_dir;
    }
    void run(std::vector<Point> &all_points) {
        total_points = all_points.size();
        dimensions = all_points[0].getDimensions();
        std::vector<int> used_pointIds;
        for (int i = 1; i <= K; i++) {
            while (true) {
                int index = rand() % total_points;
                if (find(used_pointIds.begin(),
used_pointIds.end(), index) ==
                    used_pointIds.end()) {
                    used_pointIds.push_back(index);
                    all_points[index].setCluster(i);
                    Cluster cluster(i, all_points[index]);
                    clusters.push_back(cluster);
                    break;
                }
            }
        }
        int iter = 1;
        while (true) {
            bool done = true;
#pragma omp parallel for reduction(&&: done) num_threads(16)
            for (int i = 0; i < total_points; i++) {
                int currentClusterId = all_points[i].getCluster();
                int nearestClusterId =
getNearestClusterId(all_points[i]);
                if (currentClusterId != nearestClusterId) {
                    all_points[i].setCluster(nearestClusterId);
                    done = false;
                }
            }
```

```
            clearClusters();
            for (int i = 0; i < total_points; i++) {
clusters[all_points[i].getCluster() - 1].addPoint(all_points[i]);
}
            for (int i = 0; i < K; i++) {
                int ClusterSize = clusters[i].getSize();
                for (int j = 0; j < dimensions; j++) {
                    double sum = 0.0;
                    if (ClusterSize > 0) {
#pragma omp parallel for reduction(+: sum) num_threads(16)
                        for (int p = 0; p < ClusterSize; p++) {
sum += clusters[i].getPoint(p).getVal(j); }
                        clusters[i].setCentroidByPos(j, sum /
ClusterSize);
                    }
                }
            }
            if (done || iter >= iters) {
                std::cout << "Clustering completed in iteration :
" << iter << std::endl << std::endl;
                break;
            }
            iter++;
        }
        for (int i = 0; i < total_points; i++) {
            pointsFile << all_points[i].getCluster() << std::endl;
        }
        for (int i = 0; i < K; i++) {
            for (int j = 0; j < dimensions; j++) {
                outfile << clusters[i].getCentroidByPos(j) << " ";
            }
            outfile << std::endl;
        }
    }
};
```

Б.5 Розрахунок початкових значень центроїдів кластерів для покращеного методу кластеризацій k-середніх

```
def initialize(data, k):
```

```
    centroids = []
    centroids.append(data[np.random.randint(
            data.shape[0]), :])
    plot(data, np.array(centroids))
    for c_id in range(k - 1):
        dist = []
        for i in range(data.shape[0]):
            point = data[i, :]
            d = sys.maxsize
            for j in range(len(centroids)):
                temp_dist = distance(point, centroids[j])
                d = min(d, temp_dist)
            dist.append(d)
        dist = np.array(dist)
        next_centroid = data[np.argmax(dist), :]
        centroids.append(next_centroid)
        dist = []
        plot(data, np.array(centroids))
    return centroids
centroids = initialize(data, k = 4)
```

Б.6 Нечітка кластеризація методом С-середніх

```
const unsigned int min_value = 10;
const unsigned int max_value = 40;
unsigned int randomCoord() {
    return rand() % (max_value - min_value) + min_value;
}
for (size_t i = 0; i < clustersAmount; ++i) {
    centroids.push_back(Centroid(randomCoord(), randomCoord()));
}
CMeans alg(points, centroids, 2);
int iterations = alg.run(std::pow(10, -5));
double** matrix = alg.getU();
points = alg.getPoints();
centroids = alg.getCentroids();

class Point {
private:
    double x;
```

```
    double y;
    double cluster_index;
public:
    Point() { this->cluster_index = -1; }
    Point(double x, double y) {
        this->x = x;
        this->y = y;
        this->cluster_index = -1;
    }
    void setX(double value) { this->x = value; }
    void setY(double value) { this->y = value; }
    void setClusterIndex(double value) { this->cluster_index =
value; }
    double getX() { return this->x; }
    double getY() { return this->y; }
    double getClusterIndex() { return this->cluster_index; }
};

class Centroid : public Point {
public:
    Centroid(double x, double y) : Point(x, y) {}
};

class CMeans final {
private:
    std::vector<Point> points;
    std::vector<Centroid> clusters;
    double fuzzyness;
    double eps = std::pow(10, -5);
    double J;
    std::string log;
    void recalculateClusterIndexes() {
        for (size_t i = 0; i < this->points.size(); i++) {
            double max = -1.0;
            for (size_t j = 0; j < this->clusters.size(); j++) {
                if (max < U[i][j]) {
                    max = U[i][j];
                    this->points[i].setClusterIndex((max == 0.5) ?
0.5 : j);
                }
            }
```

```
        }
    }
    double calculateEulerDistance(Point p, Centroid c) { return
std::sqrt(std::pow(p.getX() - c.getX(), 2) + std::pow(p.getY() -
c.getY(), 2)); }
    double calculateObjectiveFunction() {
        double Jk = 0;
        for (size_t i = 0; i < this->points.size(); i++) { for
(size_t j = 0; j < this->clusters.size(); j++) { Jk +=
std::pow(U[i][j], this->fuzzyness) * std::pow(this-
>calculateEulerDistance(points[i], clusters[j]), 2); } }
        return Jk;
    }
    void calculateClusterCenters() {
        for (size_t j = 0; j < this->clusters.size(); j++) {
            double uX = 0.0;
            double uY = 0.0;
            double l = 0.0;
            for (size_t i = 0; i < this->points.size(); i++) {
                double uu = std::pow(U[i][j], this->fuzzyness);
                uX += uu * this->clusters[j].getX();
                uY += uu * this->clusters[j].getY();
                l += uu;
            }
            this->clusters[j].setX(uX / l);
            this->clusters[j].setY(uY / l);
            std::ostringstream stream;
            stream << "Cluster Centroid: (" << this-
>clusters[j].getX() << "; " << this->clusters[j].getY() << ")\n";
            this->log += stream.str();
        }
    }
public:
    void setLogMsg(std::string msg) { this->log = msg; }
    std::string getLogMsg() { return this->log; }
    double** U;
    double** getU() { return this->U; }
    std::vector<Point> getPoints() { return this->points; }
    std::vector<Centroid> getCentroids() { return this->clusters;
}
```

```
    CMeans(std::vector<Point> points, std::vector<Centroid>
clusters, float fuzzy) {
        if (points.empty()) {
            std::cerr << "points is NULL" << std::endl;
        }
        if (clusters.empty()) {
            std::cerr << "clusters is NULL" << std::endl;
        }
        this->points = points;
        this->clusters = clusters;
        U = new double*[this->points.size()];
        for (size_t i = 0; i < this->points.size(); ++i) {
            U[i] = new double[this->clusters.size()];
        }
        this->fuzzyness = fuzzy;
        double diff;
        for (size_t i = 0; i < this->points.size(); i++) {
            double sum = 0.0;
            for (size_t j = 0; j < this->clusters.size(); j++) {
                diff = std::sqrt(std::pow(this->points[i].getX() -
this->clusters[j].getX(), 2.0) + std::pow(this->points[i].getY() -
this->clusters[j].getY(), 2.0));
                U[i][j] = (diff == 0) ? eps : diff;
                sum += U[i][j];
            }
            double sum2 = 0.0;
            for (size_t j = 0; j < this->clusters.size(); j++) {
                U[i][j] = 1.0 / std::pow(U[i][j] / sum, 2.0 /
(fuzzyness - 1.0));
                sum2 += U[i][j];
            }
            for (size_t j = 0; j < this->clusters.size(); j++) {
U[i][j] = U[i][j] / sum2; }
        }
        this->recalculateClusterIndexes();
    }
    void step() {
        for (size_t cl = 0; cl < clusters.size(); cl++) {
            for (size_t h = 0; h < points.size(); h++) {
                double top = calculateEulerDistance(points[h],
clusters[cl]);
```

```
                if (top < 1.0)
                    top = eps;
                double sumTerms = 0.0;
                for (size_t ck = 0; ck < clusters.size(); ck++) {
                    double thisDistance =
calculateEulerDistance(points[h], clusters[ck]);
                    if (thisDistance < 1.0)
                        thisDistance = eps;
                    sumTerms += std::pow(top / thisDistance, 2.0 /
(this->fuzzyness - 1.0));
                }
                U[h][cl] = static_cast<double>(1.0 / sumTerms);
            }
        }
        this->recalculateClusterIndexes();
    }
    int run(double accuracy) {
        int i = 0;
        int maxIterations = 20;
        do {
            i++;
            this->J = this->calculateObjectiveFunction();
            this->calculateClusterCenters();
            this->step();
            double new_J = this->calculateObjectiveFunction();
            if (std::abs(this->J - new_J) < accuracy)
                break;
        } while (maxIterations > i);
        return i;
    }
};
```

Б.7 Розрахунок матриці відстаней

```
double** matrix;
matrix = new double*[pointsAmount + 1];
for (size_t i = 0; i < pointsAmount + 1; ++i) { matrix[i] = new
double[pointsAmount + 1]; }
```

```
for (size_t i = 0; i < pointsAmount + 1; ++i) { for (size_t j = 0;
j < pointsAmount + 1; ++j) { if (i == j) { matrix[i][j] = 0; }
else { matrix[i][j] = -1; } } }
int A_idx, B_idx;
double A_x, A_y, B_x, B_y, distance;
for (auto road : roads_array) {
    A_idx = getPointIndexByID(road.first);
    A_x = points_array[A_idx][1];
    A_y = points_array[A_idx][2];
    B_idx = getPointIndexByID(road.second);
    B_x = points_array[B_idx][1];
    B_y = points_array[B_idx][2];
    distance = std::sqrt(std::pow(B_x - A_x, 2) + std::pow(B_y -
A_y, 2));
    matrix[road.first][road.second] = distance;
    matrix[road.second][road.first] = distance;
}
size_t getPointIndexByID(const int id) { for (size_t i = 0; i <
points_array.size(); ++i) { if (points_array[i][0] == id) { return
i; } } }
```

Б.8 Підготовка кластерів до пошуку маршрутів

```
for (size_t i = 0; i < pointsAmount; ++i) {
    cluster_out.open(output_dir +
std::to_string(points_clusters[i]) + "-cluster.txt", std::ios::out
| std::ios::app);
    cluster_out << points_array[i][0] << " " << points_array[i][1]
<< " " << points_array[i][2] << "\n";
    cluster_out.close();
}
```

Б.9 Підготовка матриці відстаней до пошуку маршрутів

```
for (size_t i = 0; i < clustersAmount; ++i) {
    point_clusters_in.open(output_dir + std::to_string(i + 1) + "-
cluster.txt");
```

```
    if (!point_clusters_in.is_open()) { std::cerr << "Error:
Failed to open file." << std::endl << output_dir +
std::to_string(i + 1) + "-cluster.txt" << std::endl; return 1; }
    points_array.push_back(warehouse);
    while (getline(point_clusters_in, line)) {
        std::stringstream ss;
        std::vector<int> array;
        ss << line;
        while (!ss.eof()) {
            ss >> token;
            array.push_back(std::stoi(token));
        }
        points_array.push_back(array);
    }
    point_clusters_in.close();
    points_ids_relation_out.open(output_dir + std::to_string(i +
1) + "-points_id_in_cluster_and_real_relations.txt");
    for (size_t j = 0; j < points_array.size(); j++) {
points_ids_relation_out << points_array[j][0] << " " << j << "\n";
}
    points_ids_relation_out.close();
    double** cluster_matrix = new double*[points_array.size()];
    for (size_t kk = 0; kk < points_array.size(); ++kk) {
cluster_matrix[kk] = new double[points_array.size()]; }
    for (size_t _i = 0; _i < points_array.size(); ++_i) { for
(size_t _j = 0; _j < points_array.size(); ++_j) {
cluster_matrix[_i][_j] =
matrix[points_array[_i][0]][points_array[_j][0]]; } }
    computed_matrix_out.open(output_dir + std::to_string(i + 1) +
"-distance_matrix.txt");
    for (size_t _i = 0; _i < points_array.size(); ++_i) { for
(size_t _j = 0; _j < points_array.size(); ++_j) {
computed_matrix_out << cluster_matrix[_i][_j] << " "; }
computed_matrix_out << "\n"; }
    computed_matrix_out.close();
    for (size_t k = 0; k < points_array.size(); ++k) { delete[]
cluster_matrix[k]; }
    delete[] cluster_matrix;
    points_array.clear();
}
```

Б.10 Пошук маршрутів мурашиним алгоритмом

```
Map map(points_array);
double** matrix = new double*[points_array.size()];
for (size_t kk = 0; kk < points_array.size(); ++kk) {
    matrix[kk] = new double[points_array.size()];
}
distance_matrix_in.open(output_dir + std::to_string(i + 1) + "-
distance_matrix.txt");
size_t ii = 0, jj = 0;
if (distance_matrix_in.is_open()) {
    while (getline(distance_matrix_in, line)) {
        std::stringstream ss << line;
        while (!ss.eof()) {
            ss >> token;
            matrix[ii][jj] = std::stof(token);
            jj++;
        }
        jj = 0;
        ii++;
    }
}
distance_matrix_in.close();
map.setDistancesMatrix(matrix);
std::vector<size_t> solve;
AntColony algorithm(alpha, beta, rho, Q, numAnts, maxTime);
solve = algorithm.Solution(map);
pathes_out << "ACO-" << i + 1 << ": ";
if (!solve.empty()) { for (auto p : solve) { pathes_out <<
map.getRealIDbyID(p) << " "; } }
pathes_out << "\n";

class Point {
private:
    double x;
    double y;
    size_t id;
    size_t real_id;
public:
    Point() {}
```

```
    Point(double x, double y, size_t id, size_t real_id) {
        this->x = x;
        this->y = y;
        this->id = id;
        this->real_id = real_id;
    }
    void setX(double value) { this->x = value; }
    void setY(double value) { this->y = value; }
    void setID(size_t value) { this->id = value; }
    void setRealID(size_t value) { this->real_id = value; }
    double getX() { return this->x; }
    double getY() { return this->y; }
    size_t getID() { return this->id; }
    size_t getRealID() { return this->real_id; }
    double getDistance(Point &other) {return
std::sqrt(std::pow(this->x - other.x, 2) + std::pow(this->y -
other.y, 2)); }
    friend std::ostream& operator<<(std::ostream& os, const Point&
p) { return os << p.real_id << " " << p.id << "\t| " << p.x << ";"
<< p.y << std::endl; }
};

class Map {
private:
    size_t pointsAmount;
    double** distanceMatrix;
    std::vector<Point> points;
public:
    Map() {}
    Map(std::vector<Point> &points) {
        this->points = points;
        this->pointsAmount = points.size();
    }
    void initDistanceMatrix() {
        this->distanceMatrix = new double*[this->pointsAmount];
        for (size_t i = 0; i < this->pointsAmount; ++i) {
            this->distanceMatrix[i] = new double[this-
>pointsAmount];
        }
    }
    size_t getPointsAmount() { return this->pointsAmount; }
```

```
    Point getPoint(size_t point_id) { return this-
>points[point_id]; }
    std::vector<Point> getPoints() { return this->points;  }
    double getDistance(size_t point_id_1, size_t point_id_2) {
return this->distanceMatrix[point_id_1][point_id_2];  }
    void setDistancesMatrix(double** matrix) { this-
>distanceMatrix = matrix;  }
    double** getDistancesMatrix() { return this->distanceMatrix;
}
    size_t getRealIDbyID(size_t id) { for (auto p : this->points)
{ if (p.getID() == id) { return p.getRealID(); } } return -1; }
    double getTotalDistance(std::vector<size_t> &cities) {
        if (cities.size() != this->pointsAmount) { return 0; }
        double result = 0;
        int actual = 0;
        int next = 0;
        for (size_t i = 0; i < this->pointsAmount - 1; ++i) {
            actual = cities[i];
            next = cities[i + 1];
            double distance = getDistance(actual, next);
            result += distance;
        }
        result += getDistance(next, cities[0]);
        return result;
    }
};

class AntColony {
private:
    double totalDistance;
    int alpha;
    int beta;
    double rho;
    double Q;
    size_t numAnts;
    int maxTime;
    Map map;
public:
    AntColony(int alpha, int beta, double rho, double Q, size_t
numAnts, int maxTime) {
        this->alpha = alpha;
```

```
        this->beta = beta;
        this->rho = rho;
        this->Q = Q;
        this->numAnts = numAnts;
        this->maxTime = maxTime;
    }
    std::vector<size_t> Solution(Map map) {
        this->map = map;
        std::vector<std::vector<size_t>> ants = initAnts(numAnts,
this->map.getPointsAmount());
        std::vector<size_t> bestTrail = getBestTrail(ants);
        double bestLength = getLength(bestTrail);
        std::vector<std::vector<double>> pheromones =
initPheromones(this->map.getPointsAmount());
        int time = 0;
        while (time < maxTime) {
            updateAnts(ants, pheromones);
            updatePheromones(pheromones, ants);
            std::vector<size_t> currBestTrail =
getBestTrail(ants);
            double currBestLength = getLength(currBestTrail);
            if (currBestLength < bestLength) {
                bestLength = currBestLength;
                bestTrail = currBestTrail;
            }
            ++time;
        }
        this->totalDistance = this-
>map.getTotalDistance(bestTrail);
        return bestTrail;
    }
    std::vector<std::vector<size_t>> initAnts(size_t numAnts,
size_t pointsAmount) {
        std::srand(unsigned(std::time(0)));
        std::vector<std::vector<size_t>> ants;
        ants.resize(numAnts);
        for (size_t k = 0; k < numAnts; ++k) {
            size_t start = std::rand() % (pointsAmount - 1);
            ants[k] = getRandomTrail(start, pointsAmount);
        }
        return ants;
```

```
    }
    std::vector<size_t> getRandomTrail(size_t start, size_t
pointsAmount) {
        std::srand(unsigned(std::time(0)));
        std::vector<size_t> trail;
        for (size_t i = 0; i < pointsAmount; ++i) {
trail.push_back(i); }
        for (size_t i = 0; i < pointsAmount; ++i) {
            size_t r = i + std::rand() % (pointsAmount - i);
            std::swap(trail[r], trail[i]);
        }
        size_t idx = getIndexOfTarget(trail, start);
        std::swap(trail[0], trail[idx]);
        return trail;
    }
    size_t getIndexOfTarget(std::vector<size_t> trail, size_t
target) { for (size_t i = 0; i < trail.size(); ++i) { if (trail[i]
== target) return i; } std::cerr << "Target not found in
IndexOfTarget" << std::endl; }
    double getLength(std::vector<size_t> trail) { double result =
0.0; for (size_t i = 0; i < trail.size() - 1; ++i) { result +=
this->map.getDistance(trail[i], trail[i + 1]); } return result; }
    std::vector<size_t>
getBestTrail(std::vector<std::vector<size_t>> ants) {
        double bestLength = getLength(ants[0]);
        int idxBestLength = 0;
        for (size_t k = 1; k < ants.size(); ++k) {
            double len = getLength(ants[k]);
            if (len < bestLength) {
                bestLength = len;
                idxBestLength = k;
            }
        }
        size_t pointsAmount = ants[0].size();
        std::vector<size_t> bestTrail(ants[idxBestLength]);
        return bestTrail;
    }
    std::vector<std::vector<double>> initPheromones(size_t
pointsAmount) {
        std::vector<std::vector<double>> pheromones;
        for (size_t i = 0; i < pointsAmount; ++i) {
```

```
            std::vector<double> v;
            v.resize(pointsAmount);
            pheromones.push_back(v);
        }
        for (size_t i = 0; i < pheromones.size(); ++i) { for
(size_t j = 0; j < pheromones[i].size(); ++j) { pheromones[i][j] =
0.01; } } return pheromones;
    }
    void updateAnts(std::vector<std::vector<size_t>> ants,
std::vector<std::vector<double>> pheromones) {
        std::srand(unsigned(std::time(0)));
        size_t pointsAmount = pheromones.size();
        for (size_t k = 0; k < ants.size(); ++k) {
            size_t start = std::rand() % (pointsAmount - 1);
            std::vector<size_t> newTrail = buildTrail(k, start,
pheromones);
            ants[k] = newTrail;
        }
    }
    std::vector<size_t> buildTrail(int k, size_t start,
std::vector<std::vector<double>> pheromones) {
        size_t pointsAmount = pheromones.size();
        std::vector<size_t> trail;
        std::vector<bool> visited;
        trail.resize(pointsAmount);
        visited.resize(pointsAmount);
        trail[0] = start;
        visited[start] = true;
        for (size_t i = 0; i < pointsAmount - 1; ++i) {
            size_t point_id_1 = trail[i];
            size_t next = getNextPoint(k, point_id_1, visited,
pheromones);
            trail[i + 1] = next;
            visited[next] = true;
        }
        return trail;
    }
    size_t getNextPoint(int k, size_t point_id_1,
std::vector<bool> visited, std::vector<std::vector<double>>
pheromones) {
        std::srand(unsigned(std::time(0)));
```

```
        std::vector<double> probs = moveProbs(k, point_id_1,
visited, pheromones);
        std::vector<double> cumul;
        cumul.resize(probs.size() + 1);
        for (size_t i = 0; i < probs.size(); ++i) { cumul[i + 1] =
cumul[i] + probs[i]; }

        double p = static_cast<double>(std::rand() % 1000) / 1000;
        for (size_t i = 0; i < cumul.size() - 1; ++i) { if (p >=
cumul[i] && p < cumul[i + 1]) { return i; } }
        std::cerr << "Failure to return valid city in
getNextPoint" << std::endl;
    }
    std::vector<double> moveProbs(int k, size_t point_id_1,
std::vector<bool> visited, std::vector<std::vector<double>>
pheromones) {
        size_t pointsAmount = pheromones.size();
        std::vector<double> taueta;
        taueta.resize(pointsAmount);
        double sum = 0.0;
        for (size_t i = 0; i < taueta.size(); ++i) { if (i ==
point_id_1) { taueta[i] = 0.0; } else if (visited[i] == true) {
taueta[i] = 0.0; } else { taueta[i] =
std::pow(pheromones[point_id_1][i], alpha) * std::pow((1.0 / this-
>map.getDistance(point_id_1, i)), beta); if (taueta[i] < 0.0001) {
taueta[i] = 0.0001; } else if (taueta[i] >
(std::numeric_limits<double>::max() / (pointsAmount * 100))) {
taueta[i] = std::numeric_limits<double>::max() / (pointsAmount *
100); } } sum += taueta[i]; }
        std::vector<double> probs;
        for (size_t i = 0; i < pointsAmount; ++i) {
probs.push_back(taueta[i] / sum); }
        return probs;
    }
    void updatePheromones(std::vector<std::vector<double>>
pheromones, std::vector<std::vector<size_t>> ants) {
        for (size_t i = 0; i < pheromones.size(); ++i) {
            for (size_t j = i + 1; j < pheromones[i].size(); ++j)
{
                for (size_t k = 0; k < ants.size(); ++k) {
                    double length = getLength(ants[k]);
```

```
                    double decrease = (1.0 - rho) *
pheromones[i][j];
                    double increase = 0.0;
                    if (EdgeInTrail(i, j, ants[k]) == true) {
increase = (Q / length); }
                    pheromones[i][j] = decrease + increase;
                    if (pheromones[i][j] < 0.0001) {
pheromones[i][j] = 0.0001; }
                    else if (pheromones[i][j] > 100000.0) {
pheromones[i][j] = 100000.0; }
                    pheromones[j][i] = pheromones[i][j];
                }
            }
        }
    }
    bool EdgeInTrail(size_t point_id_1, size_t point_id_2,
std::vector<size_t> trail) {
        size_t lastIndex = trail.size() - 1;
        size_t idx = getIndexOfTarget(trail, point_id_1);
        if (idx == 0 && trail[1] == point_id_2) return true;
        else if (idx == 0 && trail[lastIndex] == point_id_2)
return true;
        else if (idx == 0) return false;
        else if (idx == lastIndex && trail[lastIndex - 1] ==
point_id_2) return true;
        else if (idx == lastIndex && trail[0] == point_id_2)
return true;
        else if (idx == lastIndex) return false;
        else if (trail[idx - 1] == point_id_2) return true;
        else if (trail[idx + 1] == point_id_2) return true;
        else return false;
    }
    double getTotalDistance() { return this->totalDistance; }
};
```

Б.11 Пошук маршрутів генетичним алгоритмом

```
Map map(points_array);
double** matrix = new double*[points_array.size()];
```

```
for (size_t kk = 0; kk < points_array.size(); ++kk) { matrix[kk] =
new double[points_array.size()]; }
distance_matrix_in.open(output_dir + std::to_string(i + 1) + "-
distance_matrix.txt");
size_t ii = 0, jj = 0;
if (distance_matrix_in.is_open()) {
    while (getline(distance_matrix_in, line)) {
        std::stringstream ss << line;
        while (!ss.eof()) {
            ss >> token;
            matrix[ii][jj] = std::stof(token);
            jj++;
        }
        jj = 0;
        ii++;
    }
}
distance_matrix_in.close();
map.setDistancesMatrix(matrix);
std::vector<Point> solve;
GeneticAlgorithm algorithm(populationCount, maxTime);
solve = algorithm.Solution(map);
if (!solve.empty()) {
    for (auto p : solve) {
        pathes_out << p.getRealID() << " ";
    }
}
pathes_out << "\n";

class Point {
private:
    int pointId, clusterId;
    int dimensions;
    std::vector<double> values;
    std::vector<double> lineToVec(std::string &line) {
        std::vector<double> values;
        std::string tmp = "";
        for (int i = 0; i < static_cast<int>(line.length()); i++)
{
```

```
            if ((48 <= int(line[i]) && int(line[i]) <= 57) ||
line[i] == '.' || line[i] == '+' || line[i] == '-' || line[i] ==
'e') {
                tmp += line[i];
            }
            else if (tmp.length() > 0) {
                values.push_back(stod(tmp));
                tmp = "";
            }
        }
        if (tmp.length() > 0) {
            values.push_back(stod(tmp));
            tmp = "";
        }
        return values;
    }
public:
    Point(int id, std::string line) {
        pointId = id;
        values = lineToVec(line);
        dimensions = values.size();
        clusterId = 0; // Initially not assigned to any cluster
    }
    int getDimensions() { return dimensions;}
    int getCluster() { return clusterId; }
    int getID() { return pointId; }
    void setCluster(int val) { clusterId = val; }
    double getVal(int pos) { return values[pos]; }
    static double getTotalDistance(Point startLocation,
std::vector<Point> points) {
        if (points.empty()) std::cerr << "The points array must
have at least one element." << std::endl;
        double result = startLocation.getDistance(points[0]);
        for (size_t i = 0; i < points.size() - 1; i++) {
            auto actual = points[i];
            auto next = points[i + 1];
            auto distance = actual.getDistance(next);
            result += distance;
        }
        result += points[points.size() -
1].getDistance(startLocation);
```

```
        return result;
    }
    static void swapLocations(std::vector<Point> points, size_t
index1, size_t index2) {
        if (index1 < 0 || index1 >= points.size()) std::cerr <<
"ArgumentOutOfRangeException index1" << std::endl;

        if (index2 < 0 || index2 >= points.size()) std::cerr <<
"ArgumentOutOfRangeException index2" << std::endl;
        std::swap(points[index1], points[index2]);
    }
    static void moveLocations(std::vector<Point> points, size_t
fromIndex, size_t toIndex) {
        if (fromIndex < 0 || fromIndex >= points.size()) std::cerr
<< "ArgumentOutOfRangeException fromIndex" << std::endl;
        if (toIndex < 0 || toIndex >= points.size()) std::cerr <<
"ArgumentOutOfRangeException toIndex" << std::endl;
        auto temp = points[fromIndex];
        if (fromIndex < toIndex) { for (size_t i = fromIndex + 1;
i <= toIndex; i++) points[i - 1] = points[i]; }
        else { for (size_t i = fromIndex; i > toIndex; i--)
points[i] = points[i - 1]; }
        points[toIndex] = temp;
    }
    static void reverseRange(std::vector<Point> points, size_t
startIndex, size_t endIndex) {
        if (startIndex < 0 || startIndex >= points.size())
std::cerr << "ArgumentOutOfRangeException startIndex" <<
std::endl;
        if (endIndex < 0 || endIndex >= points.size()) std::cerr
<< "ArgumentOutOfRangeException endIndex" << std::endl;
        if (endIndex < startIndex) { std::swap(endIndex,
startIndex); }
        while (startIndex < endIndex) {
            std::swap(points[endIndex], points[startIndex]);
            startIndex++;
            endIndex--;
        }
    }
};
template <>
```

```
struct std::hash<Point> {
    std::size_t operator()(const Point& p) const {
        std::size_t h1 = std::hash<size_t>{}(p.id);
        std::size_t h2 = std::hash<size_t>{}(p.real_id);
        return h1 ^ (h2 << 1);
    }
};
class Map {
private:
    size_t pointsAmount;
    double** distanceMatrix;
    std::vector<Point> points;
public:
    Map() {}
    Map(std::vector<Point> &points) {
        this->points = points;
        this->pointsAmount = points.size();
    }
    size_t getPointsAmount() { return this->pointsAmount; }
    Point getPoint(size_t point_id) { return this-
>points[point_id]; }
    std::vector<Point> getPoints() { return this->points; }
    double getDistance(size_t point_id_1, size_t point_id_2) {
return this->distanceMatrix[point_id_1][point_id_2]; }
    void setDistancesMatrix(double** matrix) { this-
>distanceMatrix = matrix; }
    double** getDistancesMatrix() { return this->distanceMatrix; }
    size_t getRealIDbyID(size_t id) { for (auto p : this->points)
{ if (p.getID() == id) { return p.getRealID(); } } return -1; }
    double getTotalDistance(std::vector<size_t> &cities) {
        if (cities.size() != this->pointsAmount) { return 0; }
        double result = 0;
        int actual = 0;
        int next = 0;
        for (size_t i = 0; i < this->pointsAmount - 1; ++i) {
            actual = cities[i];
            next = cities[i + 1];
            double distance = getDistance(actual, next);
            result += distance;
        }
        result += getDistance(next, cities[0]);
```

```
        return result;
    }
};

class RandomProvider {
public:
    static size_t getRandomValue(size_t limit) {
        std::srand(unsigned(std::time(0)));
        if (limit == 0) limit = 1;
        return std::rand() % limit;
    }
    static void mutateRandomLocations(std::vector<Point> points) {
        if (points.size() < 2) std::cerr << "ArgumentException The
points array must have at least two items." << std::endl;
        size_t mutationCount = getRandomValue(points.size() / 10)
+ 1;
        for (size_t mutation_idx = 0; mutation_idx <
mutationCount; mutation_idx++) {
            size_t index_1 = getRandomValue(points.size());
            size_t index_2 = getRandomValue(points.size() - 1);
            if (index_2 >= index_1) index_2++;
            switch (getRandomValue(3)) {
                case 0: Point::swapLocations(points, index_1,
index_2); break;
                case 1: Point::moveLocations(points, index_1,
index_2); break;
                case 2: Point::reverseRange(points, index_1,
index_2); break;
                default: std::cerr << "InvalidOperationException"
<< std::endl; break;
            }
        }
    }
    static void fullyRandomizeLocations(std::vector<Point> points)
{ for (size_t i = points.size() - 1; i > 0; i--) { size_t value =
getRandomValue(i + 1); if (value != i)
Point::swapLocations(points, i, value); } }
    static void _CrossOver(std::vector<Point> points_1,
std::vector<Point> points_2, bool mutateFailedCrossovers) {
        std::unordered_set<Point> availableLocations;
```

```
        for (const Point &i : points_1) {
availableLocations.insert(i); }
        size_t startPosition = getRandomValue(points_1.size());
        size_t crossOverCount = getRandomValue(points_1.size() -
startPosition);
        if (mutateFailedCrossovers) {
            bool useMutation = true;
            int pastEndPosition = startPosition + crossOverCount;
            for (int i = startPosition; i < pastEndPosition; i++)
{ if (points_1[i].getID() != points_2[i].getID()) { useMutation =
false; break; } }
            if (useMutation) { mutateRandomLocations(points_1);
return; }
        }
        for (size_t i = startPosition; i <= crossOverCount; ++i) {
points_1[i] = points_2[i]; }
        std::vector<size_t> toReplaceIndexes;
        size_t index = 0;
        for (auto value : points_1) { if
(!availableLocations.erase(value))
toReplaceIndexes.push_back(index); index++; }
        if (!toReplaceIndexes.empty()) {
            auto itA = toReplaceIndexes.begin();
            auto itB = availableLocations.begin();
            while (itA != toReplaceIndexes.end() || itB !=
availableLocations.end()) {
                if (itA != toReplaceIndexes.end()) { ++itA; }
                if (itB != availableLocations.end()) { ++itB; }
                points_1[(*itA)] = (*itB);
            }
        }
    }
};

class GeneticAlgorithm {
private:
    double totalDistance;
    Point startPoint;
    std::vector<std::pair<std::vector<Point>, double>>
populationWithDistances;
    size_t populationCount;
```

```
    int maxTime;
    void startGeneticAlgorithm(Point startLocation,
std::vector<Point> destinations) {
        if (this->populationCount < 2) std::cerr <<
"ArgumentOutOfRangeException populationCount" << std::endl;
        if (this->populationCount % 2 != 0) std::cerr <<
"ArgumentException The populationCount parameter must be an even
value." << std::endl;
        this->startPoint = startLocation;
        this->populationWithDistances.resize(this-
>populationCount);
        for (size_t solutionIndex = 0; solutionIndex < this-
>populationCount; solutionIndex++) {
            std::vector<Point>
newPossibleDestinations(destinations);
            for (size_t random_idx = 0; random_idx <
newPossibleDestinations.size(); random_idx++)
RandomProvider::fullyRandomizeLocations(newPossibleDestinations);
            auto distance = Point::getTotalDistance(startLocation,
newPossibleDestinations);
            std::pair<std::vector<Point>, double> pair;
            pair.first = newPossibleDestinations;
            pair.second = distance;
            this->populationWithDistances[solutionIndex] = pair;
        }
        sortPopulationWithDistances();
    }
    std::vector<Point> _GetFakeShortest(std::vector<Point>
destinations) {
        std::vector<Point> result;
        result.resize(destinations.size());
        auto currentPoint = this->startPoint;
        for (size_t fillingIndex = 0; fillingIndex <
destinations.size(); fillingIndex++) {
            int bestIndex = -1;
            double bestDistance =
std::numeric_limits<double>::max();
            for (size_t evaluatingIndex = 0; evaluatingIndex <
destinations.size(); evaluatingIndex++) {
                auto evaluatingItem =
destinations[evaluatingIndex];
```

```
                double distance =
currentPoint.getDistance(evaluatingItem);
                if (distance < bestDistance) {
                    bestDistance = distance;
                    bestIndex = evaluatingIndex;
                }
            }
            result[fillingIndex] = destinations[bestIndex];
            currentPoint = destinations[bestIndex];
            destinations.erase(destinations.begin() + bestIndex);
        }

        return result;
    }
    std::vector<Point> getBestSolutionSoFar() { return this-
>populationWithDistances[0].first; }
    std::vector<Point> _Reproduce(std::vector<Point> parent) {
        std::vector<Point> result(parent);
        if (!this->MustDoCrossovers) {
            RandomProvider::mutateRandomLocations(result);
            return result;
        }
        int otherIndex = RandomProvider::getRandomValue(this-
>populationWithDistances.size() / 2);
        auto other = this-
>populationWithDistances[otherIndex].first;
        RandomProvider::_CrossOver(result, other, this-
>MustMutateFailedCrossovers);
        if (!this->MustMutateFailedCrossovers) if
(RandomProvider::getRandomValue(10) == 0)
RandomProvider::mutateRandomLocations(result);
        return result;
    }
public:
    bool MustMutateFailedCrossovers;
    bool MustDoCrossovers;
    GeneticAlgorithm(size_t populationCount, int maxTime) {
        this->populationCount = populationCount;
        this->maxTime = maxTime;
    }
    std::vector<Point> Solution(Map map) {
```

```
        std::vector<Point> randomPoints;
        randomPoints.resize(map.getPointsAmount());
        randomPoints = map.getPoints();
        startGeneticAlgorithm(randomPoints[0], randomPoints);
        int i = this->maxTime;
        std::vector<Point> _bestSolutionSoFar =
getBestSolutionSoFar();
        bool _mutateFailedCrossovers = false;
        bool _mutateDuplicates = false;
        bool _mustDoCrossovers = true;
        while (i-- > 0) {
            this->MustMutateFailedCrossovers =
_mutateFailedCrossovers;
            this->MustDoCrossovers = _mustDoCrossovers;
            Reproduce();
            if (_mutateDuplicates) mutateDuplicates();
            auto newSolution = getBestSolutionSoFar();
            if (newSolution != _bestSolutionSoFar) {
_bestSolutionSoFar = newSolution; }
        }
        this->totalDistance =
Point::getTotalDistance(randomPoints[0], _bestSolutionSoFar);
        return _bestSolutionSoFar;
    }
    void Reproduce() {
        auto bestSoFar = this->populationWithDistances[0];
        size_t halfCount = this->populationWithDistances.size() /
2;
        for (size_t i = 0; i < halfCount; i++) {
            auto parent = this->populationWithDistances[i].first;
            auto child1 = _Reproduce(parent);
            auto child2 = _Reproduce(parent);
            std::pair<std::vector<Point>, double> pair1;
            std::pair<std::vector<Point>, double> pair2;
            pair1.first = child1;
            pair1.second = Point::getTotalDistance(this-
>startPoint, child1);
            pair2.first = child2;
            pair2.second = Point::getTotalDistance(this-
>startPoint, child2);
            this->populationWithDistances[i * 2] = pair1;
```

```
            this->populationWithDistances[i * 2 + 1] = pair2;
        }
        this->populationWithDistances[this-
>populationWithDistances.size() - 1] = bestSoFar;
        sortPopulationWithDistances();
    }
    void mutateDuplicates() {
        bool needToSortAgain = false;
        int countDuplicates = 0;
        auto previous = this->populationWithDistances[0];
        for (size_t i = 1; i < this-
>populationWithDistances.size(); i++) {
            auto current = this->populationWithDistances[i];
            if (previous.first != current.first) {
                previous = current;
                continue;
            }
            countDuplicates++;
            needToSortAgain = true;
            RandomProvider::mutateRandomLocations(current.first);
            std::pair<std::vector<Point>, double> new_pair;
            new_pair.first = current.first;
            new_pair.second = Point::getTotalDistance(this-
>startPoint, current.first);
            this->populationWithDistances[i] = new_pair;
        }
        if (needToSortAgain) { sortPopulationWithDistances(); }
    }
    void sortPopulationWithDistances() {
        this->populationWithDistances;
        for (size_t i = 0; i < this-
>populationWithDistances.size(); ++i) { for (size_t j = (i % 2) ?
0 : 1; j + 1 < this->populationWithDistances.size(); j += 2) { if
(this->populationWithDistances[j].second > this-
>populationWithDistances[j + 1].second) { std::swap(this-
>populationWithDistances[j], this->populationWithDistances[j +
1]); } } }
    }
    double getTotalDistance() { return this->totalDistance; }
};
```

Б.12 Оцінка маршрутів

```
for (size_t i = 0; i < clustersAmount; ++i) {
    points_in.open(output_dir + std::to_string(i + 1) + "-
cluster.txt");
    while (getline(points_in, line)) {
        std::stringstream ss << line;
        points_id_array.push_back(std::stoul(line));
    }
    points_in.close();
    int sum = 0;
    for (auto id : points_id_array) { for (auto amount : plan[id])
{ sum += amount; } }
    trucks_capacity_out << "C-" << i + 1 << ": " << sum << "\n";
    points_id_array.clear();
}
```

Б.13 Візуалізація

```
#define _USE_MATH_DEFINES
const int coeff = 20;
const std::vector<std::string> saturated_colors = { "#0351c1",
"#b40a1b", "#00cf91", "#2e3f7f", "#ff2970", "#026e04", "black" };
const std::vector<std::string> pale_colors = { "#7eb3ff",
"#ff756b", "#b5fbdd", "#5569b5", "#ff9ca1", "#1cc91f", "#808080"
};
double arrowX, arrowY, cos, b, c, xx, yy;
int arrowR = 0, addR = 0;
map.open("../output/k-means__map.svg");
if (map.is_open()) {
    map << "<svg xmlns=\"http://www.w3.org/2000/svg\" height=\""
<< 50 * coeff << "\" width=\"" << 50 * coeff << "\">\n";
    map << "<symbol id=\"arrow\" viewBox=\"0 0 42 70\"
width=\"15\" height=\"25\"><polygon points =
\"42,4.243751525849802 37.47187805175781,0 0,34.99999856945942
37.47187805175781,70.00000238415669 42,65.77812433239887
9.078125,34.99999856945942 \" stroke=\"black\" /></symbol>\n";
    for (std::vector<int> road : roads_array) { map << "<line
id=\"" << road[0] << "\" x1=\"" << road[1] * coeff << "\" y1=\""
```

```
<< road[2] * coeff << "\" x2=\"" << road[3] * coeff << "\" y2=\""
<< road[4] * coeff << "\" stroke-width=\"3\" stroke=\"#606060\"
/>\n"; }
    for (auto path : pathes_array) {
        for (size_t i = 0; i < path.size() - 2; ++i) {
            double x_1 = warehouse[1];
            double y_1 = warehouse[2];
            double x_2 = warehouse[1];
            double y_2 = warehouse[2];
            for (size_t j = 0; j < points_array.size(); ++j) {
                if (points_array[j][0] == path[i]) {
                    x_1 = static_cast<double>(points_array[j][1]);
                    y_1 = static_cast<double>(points_array[j][2]);
                }
                if (points_array[j][0] == path[i + 1]) {
                    x_2 = static_cast<double>(points_array[j][1]);
                    y_2 = static_cast<double>(points_array[j][2]);
                }
            }
            arrowX = std::abs((x_1 + x_2) / 2.0);
            arrowY = std::abs((y_1 + y_2) / 2.0);
            arrowX *= coeff;
            arrowY *= coeff;
            b = std::sqrt(std::pow(x_2 - x_1, 2) + std::pow(y_1 -
y_1, 2));
            c = std::sqrt(std::pow(x_2 - x_1, 2) + std::pow(y_2 -
y_1, 2));
            cos = b / c;
            xx = x_2 - x_1;
            yy = y_2 - y_1;
            if (xx < yy) { if (yy > 0) { addR = 90; } else { addR
= 270; } } else { if (xx > 0) { addR = 180; }  else {addR = 0; } }
            arrowR = static_cast<int>((cos * 180) / M_PI) + addR;
            std::cout << "Cos: " << cos << "\t\tGrad: " << arrowR
<< std::endl;
            map << "<use href=\"#arrow\" from=\"" << path[i] <<
"\" to=\"" << path[i + 1] << "\" x=\"" << arrowX - 15.0 / 2.0 <<
"\" y=\"" << arrowY - 25.0 / 2.0 << "\" transform=\"rotate(" <<
arrowR << " " << arrowX << " " << arrowY << ")\"></use>\n";
        }
    }
```

```
    for (size_t i = 0; i < points_array.size(); ++i) { map <<
"<circle id=\"" << points_array[i][0] << "\" cx=\"" <<
points_array[i][1] * coeff << "\" cy=\"" << points_array[i][2] *
coeff << "\" r=\"5\" fill=\"" <<
saturated_colors[points_clusters[i] - 1] << "\" />\n"; }
    for (size_t i = 0; i < points_array.size(); ++i) { map <<
"<text x=\"" << points_array[i][1] * coeff + 10 << "\" y=\"" <<
points_array[i][2] * coeff + 10 << "\">" << points_array[i][0] <<
"</text>\n"; }
    map << "<circle id=\"" << warehouse[0] << "\" cx=\"" <<
warehouse[1] * coeff << "\" cy=\"" << warehouse[2] * coeff << "\"
r=\"10\" fill=\"black\" />\n";
    for (size_t i = 0; i < cluster_centroids.size(); ++i) { map <<
"<circle id=\"" << i + 1 << "\" cx=\"" <<
cluster_centroids[i].first * coeff << "\" cy=\"" <<
cluster_centroids[i].second * coeff << "\" r=\"5\" stroke=\"" <<
pale_colors[i] << "\" stroke-width=\"4\" fill=\"transparent\"
/>\n"; }
    map << "</svg>";
}
map.close();
map.open("../output/c-means__map.svg");
if (map.is_open()) {
    map << "<svg xmlns=\"http://www.w3.org/2000/svg\" height=\""
<< 50 * coeff << "\" width=\"" << 50 * coeff << "\">\n";
    map << "<symbol id=\"arrow\" viewBox=\"0 0 42 70\"
width=\"15\" height=\"25\"><polygon points =
\"42,4.243751525849802 37.47187805175781,0 0,34.99999856945942
37.47187805175781,70.00000238415669 42,65.77812433239887
9.078125,34.99999856945942 \" stroke=\"black\" /></symbol>\n";
    for (std::vector<int> road : roads_array) { map << "<line
id=\"" << road[0] << "\" x1=\"" << road[1] * coeff << "\" y1=\""
<< road[2] * coeff << "\" x2=\"" << road[3] * coeff << "\" y2=\""
<< road[4] * coeff << "\" stroke-width=\"3\" stroke=\"#606060\"
/>\n"; }
    for (auto path : pathes_array) {
        for (size_t i = 0; i < path.size() - 2; ++i) {
            double x_1 = warehouse[1];
            double y_1 = warehouse[2];
            double x_2 = warehouse[1];
            double y_2 = warehouse[2];
```

```
            for (size_t j = 0; j < points_array.size(); ++j) {
                if (points_array[j][0] == path[i]) {
                    x_1 = static_cast<double>(points_array[j][1]);
                    y_1 = static_cast<double>(points_array[j][2]);
                }
                if (points_array[j][0] == path[i + 1]) {
                    x_2 = static_cast<double>(points_array[j][1]);
                    y_2 = static_cast<double>(points_array[j][2]);
                }
            }
            arrowX = std::abs((x_1 + x_2) / 2.0);
            arrowY = std::abs((y_1 + y_2) / 2.0);
            arrowX *= coeff;
            arrowY *= coeff;
            b = std::sqrt(std::pow(x_2 - x_1, 2) + std::pow(y_1 -
y_1, 2));
            c = std::sqrt(std::pow(x_2 - x_1, 2) + std::pow(y_2 -
y_1, 2));
            cos = b / c;
            xx = x_2 - x_1;
            yy = y_2 - y_1;
            if (xx < yy) { if (yy > 0) { addR = 90; } else { addR
= 270; } } else { if (xx > 0) { addR = 180; }  else {addR = 0; } }
            arrowR = static_cast<int>((cos * 180) / M_PI) + addR;
            std::cout << "Cos: " << cos << "\t\tGrad: " << arrowR
<< std::endl;
            map << "<use href=\"#arrow\" from=\"" << path[i] <<
"\" to=\"" << path[i + 1] << "\" x=\"" << arrowX - 15.0 / 2.0 <<
"\" y=\"" << arrowY - 25.0 / 2.0 << "\" transform=\"rotate(" <<
arrowR << " " << arrowX << " " << arrowY << ")\"></use>\n";
        }
    }
    for (size_t i = 0; i < points_array.size(); ++i) { map <<
"<circle id=\"" << points_array[i][0] << "\" cx=\"" <<
points_array[i][1] * coeff << "\" cy=\"" << points_array[i][2] *
coeff << "\" r=\"5\" fill=\"" <<
saturated_colors[points_clusters[i] - 1] << "\" />\n"; }
    for (size_t i = 0; i < points_array.size(); ++i) { map <<
"<text x=\"" << points_array[i][1] * coeff + 10 << "\" y=\"" <<
points_array[i][2] * coeff + 10 << "\">" << points_array[i][0] <<
"</text>\n"; }
```

```
    map << "<circle id=\"" << warehouse[0] << "\" cx=\"" <<
warehouse[1] * coeff << "\" cy=\"" << warehouse[2] * coeff << "\"
r=\"10\" fill=\"black\" />\n";
    for (size_t i = 0; i < cluster_centroids.size(); ++i) { map <<
"<circle id=\"" << i + 1 << "\" cx=\"" <<
cluster_centroids[i].first * coeff << "\" cy=\"" <<
cluster_centroids[i].second * coeff << "\" r=\"5\" stroke=\"" <<
pale_colors[i] << "\" stroke-width=\"4\" fill=\"transparent\"
/>\n";}
    map << "</svg>";
}
map.close();
```

ДОДАТОК В
ВІДОМІСТЬ КВАЛІФІКАЦІЙНОЇ РОБОТИ

№	Позначення	Найменування	Примітка
		Текстові документи	
1	ГЮИК.506120.001 ПЗ	Пояснювальна записка	83 с.
2	ГЮИК.506120.001 34 01 ІЗ	Текст програми	42 с.
		Графічні документи	2 с.
3		Блок-схема мурашиного алгоритму	1 стор.
4		Блок-схема генетичного алгоритму	1 стор.

					ГЮИК. 506120.001 ДЗ			
Змін.	*Арк*	*№ докум.*	*Підпис*	*Дата*				
Разраб.		*Бронза Є.С.*			Дослідження та розробка методів маршрутизації транспортних засобів в системі мережевого ритейлу	Літера	Арк.	Аркушів
Перевірив.		*Імангулова З.А.*				У	*1*	*1*
Н. Контр.		*Імангулова З.А.*				*ХНУРЭ Кафедра СТ*		
Затвер.		*Гребеннік І.В.*			Відомість атестаційної роботи			