Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"
Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт

з лабораторної роботи № 4 з дисципліни
«Проектування алгоритмів»

„Проектування і аналіз алгоритмів для вирішення NP-складних задач ч.1”

Виконав(ла) *Присяжний А. О.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Ахаладзе І. Е.* _____
(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи – вивчити основні підходи формалізації метаевристичних алгоритмів і вирішення типових задач з їхньою допомогою.

2 ЗАВДАННЯ

Згідно варіанту, розробити алгоритм вирішення задачі і виконати його програмну реалізацію на будь-якій мові програмування.

Задача, алгоритм і його параметри наведені в таблиці 2.1.

Зафіксувати якість отриманого розв'язку (значення цільової функції) після кожних 5 ітерацій до 100 і побудувати графік залежності якості розв'язку від числа ітерацій.

Зробити узагальнений висновок.

Таблиця 2.1 – Варіанти алгоритмів

№	**Задача і алгоритм**
1	Задача про рюкзак (місткість P=250, 100 предметів, цінність предметів від 2 до 20 (випадкова), вага від 1 до 10 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування одноточковий по 50 генів, мутація з ймовірністю 5% змінюємо тільки 1 випадковий ген). Розробити власний оператор локального покращення.
2	Задача комівояжера (100 вершин, відстань між вершинами випадкова від 5 до 50), мурашиний алгоритм ($\alpha = 2$, $\beta = 4$, $\rho = 0{,}4$, Lmin знайти жадібним алгоритмом, кількість мурах M = 30, починають маршрут в різних випадкових вершинах).
3	Задача розфарбовування графу (200 вершин, степінь вершини не більше 20, але не менше 1), бджолиний алгоритм ABC (число бджіл 30 із них 2 розвідники).
4	Задача про рюкзак (місткість P=200, 100 предметів, цінність предметів від 2 до 20 (випадкова), вага від 1 до 10 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування двоточковий порівну генів, мутація з ймовірністю 10% змінюємо тільки 1 випадковий ген). Розробити

	власний оператор локального покращення.
5	Задача комівояжера (150 вершин, відстань між вершинами випадкова від 5 до 50), мурашиний алгоритм (α = 2, β = 3, ρ = 0,4, Lmin знайти жадібним алгоритмом, кількість мурах M = 35, починають маршрут в різних випадкових вершинах).
6	Задача розфарбовування графу (250 вершин, степінь вершини не більше 25, але не менше 2), бджолиний алгоритм ABC (число бджіл 35 із них 3 розвідники).
7	Задача про рюкзак (місткість P=150, 100 предметів, цінність предметів від 2 до 10 (випадкова), вага від 1 до 5 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування рівномірний, мутація з ймовірністю 5% два випадкові гени міняються місцями). Розробити власний оператор локального покращення.
8	Задача комівояжера (200 вершин, відстань між вершинами випадкова від 0(перехід заборонено) до 50), мурашиний алгоритм (α = 3, β = 2, ρ = 0,3, Lmin знайти жадібним алгоритмом, кількість мурах M = 45, починають маршрут в різних випадкових вершинах).
9	Задача розфарбовування графу (150 вершин, степінь вершини не більше 30, але не менше 1), бджолиний алгоритм ABC (число бджіл 25 із них 3 розвідники).
10	Задача про рюкзак (місткість P=150, 100 предметів, цінність предметів від 2 до 10 (випадкова), вага від 1 до 5 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування рівномірний, мутація з ймовірністю 10% два випадкові гени міняються місцями). Розробити власний оператор локального покращення.
11	Задача комівояжера (250 вершин, відстань між вершинами випадкова від 0(перехід заборонено) до 50), мурашиний алгоритм (α = 2, β = 4, ρ =

	0,6, Lmin знайти жадібним алгоритмом, кількість мурах M = 45, починають маршрут в різних випадкових вершинах).
12	Задача розфарбовування графу (300 вершин, степінь вершини не більше 30, але не менше 1), бджолиний алгоритм ABC (число бджіл 60 із них 5 розвідники).
13	Задача про рюкзак (місткість P=250, 100 предметів, цінність предметів від 2 до 30 (випадкова), вага від 1 до 25 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування одноточковий 30% і 70%, мутація з ймовірністю 5% два випадкові гени міняються місцями). Розробити власний оператор локального покращення.
14	Задача комівояжера (250 вершин, відстань між вершинами випадкова від 1 до 40), мурашиний алгоритм (α = 4, β = 2, ρ = 0,3, Lmin знайти жадібним алгоритмом, кількість мурах M = 45 (10 з них дикі, обирають випадкові напрямки), починають маршрут в різних випадкових вершинах).
15	Задача розфарбовування графу (100 вершин, степінь вершини не більше 20, але не менше 1), класичний бджолиний алгоритм (число бджіл 30 із них 3 розвідники).
16	Задача про рюкзак (місткість P=250, 100 предметів, цінність предметів від 2 до 30 (випадкова), вага від 1 до 25 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування двоточковий 30%, 40% і 30%, мутація з ймовірністю 10% два випадкові гени міняються місцями). Розробити власний оператор локального покращення.
17	Задача комівояжера (200 вершин, відстань між вершинами випадкова від 1 до 40), мурашиний алгоритм (α = 2, β = 4, ρ = 0,7, Lmin знайти жадібним алгоритмом, кількість мурах M = 45 (15 з них дикі, обирають випадкові напрямки), починають маршрут в різних випадкових

	вершинах).
18	Задача розфарбовування графу (300 вершин, степінь вершини не більше 50, але не менше 1), класичний бджолиний алгоритм (число бджіл 60 із них 5 розвідники).
19	Задача про рюкзак (місткість P=250, 100 предметів, цінність предметів від 2 до 30 (випадкова), вага від 1 до 25 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування триточковий 25%, мутація з ймовірністю 5% два випадкові гени міняються місцями). Розробити власний оператор локального покращення.
20	Задача комівояжера (200 вершин, відстань між вершинами випадкова від 1 до 40), мурашиний алгоритм ($\alpha = 3$, $\beta = 2$, $\rho = 0,7$, Lmin знайти жадібним алгоритмом, кількість мурах M = 45 (10 з них елітні, подвійний феромон), починають маршрут в різних випадкових вершинах).
21	Задача розфарбовування графу (200 вершин, степінь вершини не більше 30, але не менше 1), класичний бджолиний алгоритм (число бджіл 40 із них 2 розвідники).
22	Задача про рюкзак (місткість P=250, 100 предметів, цінність предметів від 2 до 30 (випадкова), вага від 1 до 25 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування триточковий 25%, мутація з ймовірністю 5% змінюємо тільки 1 випадковий ген). Розробити власний оператор локального покращення.
23	Задача комівояжера (300 вершин, відстань між вершинами випадкова від 1 до 60), мурашиний алгоритм ($\alpha = 3$, $\beta = 2$, $\rho = 0,6$, Lmin знайти жадібним алгоритмом, кількість мурах M = 45 (15 з них елітні, подвійний феромон), починають маршрут в різних випадкових вершинах).

24	Задача розфарбовування графу (400 вершин, степінь вершини не більше 50, але не менше 1), класичний бджолиний алгоритм (число бджіл 70 із них 10 розвідники).
25	Задача про рюкзак (місткість P=250, 100 предметів, цінність предметів від 2 до 20 (випадкова), вага від 1 до 10 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування одноточковий по 50 генів, мутація з ймовірністю 5% змінюємо тільки 1 випадковий ген). Розробити власний оператор локального покращення.
26	Задача комівояжера (100 вершин, відстань між вершинами випадкова від 5 до 50), мурашиний алгоритм ($\alpha = 2$, $\beta = 4$, $\rho = 0{,}4$, Lmin знайти жадібним алгоритмом, кількість мурах M = 30, починають маршрут в різних випадкових вершинах).
27	Задача розфарбовування графу (200 вершин, степінь вершини не більше 20, але не менше 1), бджолиний алгоритм ABC (число бджіл 30 із них 2 розвідники).
28	Задача про рюкзак (місткість P=200, 100 предметів, цінність предметів від 2 до 20 (випадкова), вага від 1 до 10 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування двоточковий порівну генів, мутація з ймовірністю 10% змінюємо тільки 1 випадковий ген). Розробити власний оператор локального покращення.
29	Задача комівояжера (150 вершин, відстань між вершинами випадкова від 5 до 50), мурашиний алгоритм ($\alpha = 2$, $\beta = 3$, $\rho = 0{,}4$, Lmin знайти жадібним алгоритмом, кількість мурах M = 35, починають маршрут в різних випадкових вершинах).
30	Задача розфарбовування графу (250 вершин, степінь вершини не більше 25, але не менше 2), бджолиний алгоритм ABC (число бджіл 35 із них 3 розвідники).

31	Задача про рюкзак (місткість P=250, 100 предметів, цінність предметів від 2 до 20 (випадкова), вага від 1 до 10 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування одноточковий по 50 генів, мутація з ймовірністю 5% змінюємо тільки 1 випадковий ген). Розробити власний оператор локального покращення.
32	Задача комівояжера (100 вершин, відстань між вершинами випадкова від 5 до 50), мурашиний алгоритм ($\alpha = 2$, $\beta = 4$, $\rho = 0,4$, Lmin знайти жадібним алгоритмом, кількість мурах M = 30, починають маршрут в різних випадкових вершинах).
33	Задача розфарбовування графу (200 вершин, степінь вершини не більше 20, але не менше 1), бджолиний алгоритм ABC (число бджіл 30 із них 2 розвідники).
34	Задача про рюкзак (місткість P=200, 100 предметів, цінність предметів від 2 до 20 (випадкова), вага від 1 до 10 (випадкова)), генетичний алгоритм (початкова популяція 100 осіб кожна по 1 різному предмету, оператор схрещування двоточковий порівну генів, мутація з ймовірністю 10% змінюємо тільки 1 випадковий ген). Розробити власний оператор локального покращення.
35	Задача комівояжера (150 вершин, відстань між вершинами випадкова від 5 до 50), мурашиний алгоритм ($\alpha = 2$, $\beta = 3$, $\rho = 0,4$, Lmin знайти жадібним алгоритмом, кількість мурах M = 35, починають маршрут в різних випадкових вершинах).

3 ВИКОНАННЯ

3.1 Програмна реалізація алгоритму

3.1.1 Вихідний код

ACO_TSP_Solver.h

```
#pragma once

#include "Ant.h"
#include "FullGraph.h"
#include "TSP_Greedy_Solver.h"
#include <algorithm>


#define AMOUNT_OF_ANTS 45
#define AMOUNT_OF_ELITE_ANTS 10
#define PHEROMONE_DEGREE 3
#define VISIBILITY_DEGREE 2
#define AMOUNT_OF_ITERATIONS 5


class ACO_TSP_Solver {
    FullGraph* graphToSolve;
    int amountOfAnts;
    int amountOfEliteAnts;
    int optimalCycleLength;
    int amountOfIterations;
    vector<Ant*> ants;

    vector<Ant*> currentEliteAnts;
```

```
    Path currentBestCycle;

    void calculateOptimalCycleLength();
    void placeAnts();
    void buildCyclesForAnts();
    void findCurrentEliteAnts();
    void renewPheromone();
    void clearAntsPathes();
public:
    ACO_TSP_Solver(FullGraph*& graph);

    Path solve();

    int getOptimalCycleLength() const;

    ~ACO_TSP_Solver();
};
```

Ant.h

```
#pragma once

#include "Vertex_Edge_Path.h"
#include "Random_generators.h"
#include <math.h>


class Ant {
    int optimalCycleLength;
    Vertex* startVertex;
    Vertex* currentVertex;
```

```
    Path path;
    int pheromoneDegree;
    int visibilityDegree;

    Edge* selectNextEdgeToMove();
    double calculateSelectionValue(Edge* edge);
    vector<Edge*> getPossibleNextEdges(float &generalSelectionValue);
    Edge* getLastPossibleEdge();
public:
    Ant(Vertex* startVertex, int optimalCycleLength, int pheromoneDegree, int visibilityDegree);

    void move();
    void extractPheromone();

    Path getPath();
    void clearPath();
};
```

FullGraph.h

```
#pragma once

#include "Vertex_Edge_Path.h"
#include "Random_generators.h"
#include <iostream>
#include <iomanip>

#define EVAPORATION_COEFFICIENT 0.7
#define MAX_WEIGHT 40
```

```
class FullGraph {
    int amountOfVertexes;

    vector<Vertex*> vertexes;
    vector<Edge*> edges;

    void generateVertexes();
    void generateEdges();
    void displayVertexesInTop(int width, int amountOfDisplayedVertexes);
    void displayEdgeLength(int vertexIndex, int from, int to, int width);
    void displayFirstAndLastEdges(int vertexIndex, int amount, int width);
public:
    FullGraph(int amountOfVertexes);

    int getAmountOfVertexes() const;
    vector<Vertex*> getVertexes();
    vector<Edge*> getEdges();

    void display();
    Edge* getEdgeWithVertexes(Vertex* vertex1, Vertex* vertex2);

    ~FullGraph();
};
```

Input_Validators.h

```
#pragma once

#include <string>
#include <iostream>
using namespace std;
```

```
int inputPositiveNumberInRange(int low, int top);
bool isNumber(const string& input);
```

Random_generators.h

```
#pragma once

#include <stdlib.h>
using namespace std;

int generateNumberInRange(int lower_bound, int upperBound);

double generateDoubleNumberInRange(double lower_bound, double upperBound);
```

TSP_Greedy_Solver.h

```
#pragma once

#include "FullGraph.h"

class TSPGreedySolver {
    FullGraph* graphToSolve;

    Edge* findTheShortestEdge(Vertex* vertexes, Path& passedPath);
    void selectNextVertex(Vertex*& currentVertex, Edge* edge);
    Edge* findLastEdge(Vertex* startVertex, Vertex* currentVertex);
public:
    TSPGreedySolver(FullGraph*& graph);

    Path solve();
```

```
};
```

Vertex_Edge_Path.h

```
#pragma once

#include <vector>
#include <iostream>
using namespace std;

class Vertex;
class Edge;


class Vertex {
    vector<Edge*> incidentEdges;
    int number;
public:
    Vertex(int number);

    void addIncidentEdge(Edge* edgeToAdd);
    vector<Edge*> getIncidentEdges();
    int getNumber();

    bool operator==(const Vertex& obj);
};


class Edge {
    vector<Vertex*> connectedVertexes;
    int length;
```

```
    float amountOfPheromone;
    float evaporationCoefficient;
public:
    Edge(Vertex* vertex1, Vertex* vertex2, int length, float amountOfPheromone, float evaporationCoefficient);

    void evaporatePheromone();
    void addPheromone(float additionalPheromone);

    int getLength() const;
    float getAmountOfPheromone() const;
    bool contains(Vertex* vertex) const;

    vector<Vertex*> getConnectedVertexes();
    bool operator==(const Edge& obj);
};

class Path {
    vector<Edge*> edges;
    int amountOfEdges;
    int length;
public:
    Path();

    void addEdge(Edge* edge);
    void clear();

    Edge*& operator[](int index);
    int getAmountOfEdges() const;
    int getLength() const;
```

```
    bool containsVertexes(vector<Vertex*> vertexes) const;
    bool operator<(const Path& obj);

    void display();
};
```

ACO_TSP_Solver.cpp

```
#include "ACO_TSP_Solver.h"

ACO_TSP_Solver::ACO_TSP_Solver(FullGraph *&graph) {
    graphToSolve = graph;
    amountOfAnts = AMOUNT_OF_ANTS;
    amountOfEliteAnts = AMOUNT_OF_ELITE_ANTS;
    amountOfIterations = AMOUNT_OF_ITERATIONS;
}

Path ACO_TSP_Solver::solve() {
    calculateOptimalCycleLength();

    placeAnts();

    for(int i = 0; i < amountOfIterations; ++i) {
        buildCyclesForAnts();
        findCurrentEliteAnts();
        currentBestCycle = ants.at(0)->getPath();
        renewPheromone();
        clearAntsPathes();
    }
```

```
    return currentBestCycle;
}
void ACO_TSP_Solver::calculateOptimalCycleLength() {
    TSPGreedySolver solver(graphToSolve);
    Path greedyResult = solver.solve();
    optimalCycleLength = greedyResult.getLength();
}
void ACO_TSP_Solver::placeAnts() {
    vector<Vertex*> graphVertexes = graphToSolve->getVertexes();

    for(int i = 0; i < amountOfAnts; ++i) {
        int vertexIndex = generateNumberInRange(0, graphVertexes.size()-1);
        Ant* ant = new Ant(graphVertexes.at(vertexIndex), optimalCycleLength,
PHEROMONE_DEGREE, VISIBILITY_DEGREE);
        ants.push_back(ant);
        graphVertexes.erase(graphVertexes.begin() + vertexIndex);
    }
}
void ACO_TSP_Solver::buildCyclesForAnts() {
    for(auto ant : ants) {
        for(int i = 0; i < graphToSolve->getAmountOfVertexes(); ++i)
            ant->move();
    }
}
void ACO_TSP_Solver::findCurrentEliteAnts() {
    sort(ants.begin(), ants.end(), [](Ant* a, Ant* b) {return a->getPath() < b-
>getPath();});

    currentEliteAnts.clear();
    for(int i = 0; i < amountOfEliteAnts; ++i)
```

```
        currentEliteAnts.push_back(ants.at(i));
}
void ACO_TSP_Solver::renewPheromone() {
    for(auto edge : graphToSolve->getEdges())
        edge->evaporatePheromone();

    for(auto eliteAnt : currentEliteAnts)
        eliteAnt->extractPheromone();

    for(auto ant : ants)
        ant->extractPheromone();
}
void ACO_TSP_Solver::clearAntsPathes() {
    for(auto ant : ants)
        ant->clearPath();
}

int ACO_TSP_Solver::getOptimalCycleLength() const {
    return optimalCycleLength;
}

ACO_TSP_Solver::~ACO_TSP_Solver() {
    for(auto ant : ants)
        delete ant;
}
```

Ant.cpp

```
#include "Ant.h"
```

```
Ant::Ant(Vertex *startVertex, int optimalCycleLength, int pheromoneDegree,
int visibilityDegree) : optimalCycleLength(optimalCycleLength) {
    this->startVertex = startVertex;
    this->currentVertex = startVertex;
    this->pheromoneDegree = pheromoneDegree;
    this->visibilityDegree = visibilityDegree;
}

void Ant::move() {
    Edge* nextEdge;
    nextEdge = selectNextEdgeToMove();
    path.addEdge(nextEdge);

    vector<Vertex*> nextEdgeConnectedVertexes;
    nextEdgeConnectedVertexes = nextEdge->getConnectedVertexes();

    if(*currentVertex == *nextEdgeConnectedVertexes[0])
        currentVertex = nextEdgeConnectedVertexes[1];
    else
        currentVertex = nextEdgeConnectedVertexes[0];
}

Edge *Ant::selectNextEdgeToMove() {
    double nextEdgeSelector = generateDoubleNumberInRange(0, 1);

    float generalSelectionValue = 0;
    vector<Edge*>                possibleNextEdges                =
getPossibleNextEdges(generalSelectionValue);

    if(possibleNextEdges.empty()) {
```

```
            possibleNextEdges.push_back(getLastPossibleEdge());
            generalSelectionValue = calculateSelectionValue(possibleNextEdges[0]);
        }

        for(auto edge : possibleNextEdges) {
            double selectionPossibility = calculateSelectionValue(edge) / generalSelectionValue;
            nextEdgeSelector -= selectionPossibility;

            if(nextEdgeSelector <= 0)
                return edge;
        }

        return possibleNextEdges.back();
    }
    double Ant::calculateSelectionValue(Edge *edge) {
        double pheromoneContribution = pow(edge->getAmountOfPheromone(), pheromoneDegree);
        double visibilityContribution = pow((1.0 / edge->getLength()), visibilityDegree);
        return pheromoneContribution*visibilityContribution;
    }
    vector<Edge *> Ant::getPossibleNextEdges(float &generalSelectionValue) {
        vector<Edge*> possibleNextEdges;

        for(auto edge : currentVertex->getIncidentEdges()) {
            if(path.containsVertexes(edge->getConnectedVertexes()))
                continue;
            possibleNextEdges.push_back(edge);
            generalSelectionValue += calculateSelectionValue(edge);
```

```
        }

        return possibleNextEdges;
    }
    Edge *Ant::getLastPossibleEdge() {
        for(auto edge : currentVertex->getIncidentEdges()) {
            if(edge->contains(startVertex))
                return edge;
        }
    }


    void Ant::extractPheromone() {
        double additionalAmountOfPheromone =
(double)optimalCycleLength/path.getLength();
        for(int i = 0; i < path.getAmountOfEdges(); ++i)
            path[i]->addPheromone(additionalAmountOfPheromone);
    }

    Path Ant::getPath() {
        return path;
    }

    void Ant::clearPath() {
        path.clear();
    }
```

FullGraph.cpp

```
    #include "FullGraph.h"
```

```
FullGraph::FullGraph(int amountOfVertexes) {
    this->amountOfVertexes = amountOfVertexes;
    generateVertexes();
    generateEdges();
}
void FullGraph::generateVertexes() {
    for(int i = 0; i < amountOfVertexes; ++i) {
        Vertex* vertex = new Vertex(i+1);
        vertexes.push_back(vertex);
    }
}
void FullGraph::generateEdges() {
    for(int i = 0; i < amountOfVertexes; ++i) {
        for(int j = i+1; j < amountOfVertexes; ++j) {
            int length = generateNumberInRange(1, MAX_WEIGHT);
            double amountOfPheromone = generateDoubleNumberInRange(0, 1);

            Edge* edge = new Edge(vertexes.at(i), vertexes.at(j), length, amountOfPheromone, EVAPORATION_COEFFICIENT);
            edges.push_back(edge);

            vertexes.at(i)->addIncidentEdge(edge);
            vertexes.at(j)->addIncidentEdge(edge);
        }
    }
}

int FullGraph::getAmountOfVertexes() const {
    return  amountOfVertexes;
}
```

```
vector<Vertex *> FullGraph::getVertexes() {
    return vertexes;
}
vector<Edge *> FullGraph::getEdges() {
    return edges;
}

Edge *FullGraph::getEdgeWithVertexes(Vertex *vertex1, Vertex *vertex2) {
    for(auto &edge : edges) {
        if(edge->contains(vertex1) && edge->contains(vertex2))
            return edge;
    }

    return nullptr;
}

void FullGraph::display() {
    const int amountOfDisplayedVertexes = 5;
    const int width = 3;

    displayVertexesInTop(width, amountOfDisplayedVertexes);

    for (int i = 0; i < amountOfDisplayedVertexes; ++i)
        displayFirstAndLastEdges(i, amountOfDisplayedVertexes, width);

    for (int i = 0; i < amountOfDisplayedVertexes * 2 + 2; ++i)
        cout << "... ";
    cout << endl;
```

```
        for (int i = amountOfVertexes - amountOfDisplayedVertexes; i < amountOfVertexes; ++i)
            displayFirstAndLastEdges(i, amountOfDisplayedVertexes, width);
    }

    void FullGraph::displayFirstAndLastEdges(int vertexIndex, int amount, int width) {
        cout << setw(width) << vertexes.at(vertexIndex)->getNumber() << " ";
        displayEdgeLength(vertexIndex, 0, amount, width);
        cout << "... ";
        displayEdgeLength(vertexIndex, amountOfVertexes - amount, amountOfVertexes, width);
        cout << endl;
    }
    void FullGraph::displayVertexesInTop(int width, int amountOfDisplayedVertexes) {
        cout << setw(width) << "" << " ";
        for (int i = 0; i < amountOfDisplayedVertexes; ++i)
            cout << setw(width) << vertexes.at(i)->getNumber() << " ";
        cout << "... ";
        for (int i = amountOfVertexes - amountOfDisplayedVertexes; i < amountOfVertexes; ++i)
            cout << setw(width) << vertexes.at(i)->getNumber() << " ";
        cout << endl;
    }
    void FullGraph::displayEdgeLength(int vertexIndex, int from, int to, int width) {
        for (int j = from; j < to; ++j) {
            if (vertexIndex == j) {
                cout << setw(width) << "0" << " ";
```

```
            continue;
        }
        Edge* edge = getEdgeWithVertexes(vertexes.at(vertexIndex),
vertexes.at(j));
        cout << setw(width) << edge->getLength() << " ";
    }
}

FullGraph::~FullGraph() {
    for(auto vertex : vertexes)
        delete vertex;
    for(auto edge : edges)
        delete edge;
}
```

Input_Validators.cpp

```
#include "Input_Validators.h"

int inputPositiveNumberInRange(int low, int top) {
    int number;
    string input;
    bool repeat;
    do {
        repeat = false;
        getline(cin, input);
        if (!isNumber(input) || (input[0] == '0' && input.length() != 1)) {
            cout << "Invalid data, input positive integer number, please." << endl;
            repeat = true;
            continue;
        }
```

```
        number = stoi(input);
        if (number < low || number > top) {
            cout << "Number must be from " << low << " to " << top << endl;
            repeat = true;
        }
    } while (repeat);
    return number;
}
bool isNumber(const string& input) {
    for (char ch : input) {
        if (!isdigit(ch))
            return false;
    }
    return true;
}
```

main.cpp

```
#include "FullGraph.h"
#include "ACO_TSP_Solver.h"
#include "Input_Validators.h"

#define AMOUNT_OF_VERTEXES 200

void displayWelcomeInfo();

int main() {
    srand(time(nullptr));

    displayWelcomeInfo();
```

```
        int run;
        do {
            FullGraph graph(AMOUNT_OF_VERTEXES);
            FullGraph* graphPointer = &graph;

            cout << "Graph:" << endl;
            graph.display();

            ACO_TSP_Solver solver(graphPointer);
            Path result = solver.solve();
            cout << "Greedy algorithm length: " << solver.getOptimalCycleLength()
<< endl;
            cout << "Ant colony optimization solution:" << endl;
            result.display();

            cout << "Enter 0 to stop the program of 1 to run it again" << endl;
            run = inputPositiveNumberInRange(0, 1);
        } while (run);
        return 0;
    }

    void displayWelcomeInfo() {
        cout << "This program implements Ant Colony Optimization algorithm for
solving Traveling Salesman Problem" << endl;
        cout << "Amount of vertexes in graph: " << AMOUNT_OF_VERTEXES
<< endl;
        cout << "a (pheromone degree): " << PHEROMONE_DEGREE << endl;
        cout << "b (visibility degree): " << VISIBILITY_DEGREE << endl;
        cout << "p (evaporation coefficient): " <<
EVAPORATION_COEFFICIENT << endl;
```

```
    cout << "Amount of ants: " << AMOUNT_OF_ANTS << endl;
    cout << "Amount of elite ants: " << AMOUNT_OF_ELITE_ANTS << endl;
    cout << "Amount of iterations: " << AMOUNT_OF_ITERATIONS << endl;
}
```

Random_generators.cpp

```
#include "Random_generators.h"

int generateNumberInRange(int lowerBound, int upperBound) {
    return lowerBound + rand() % (upperBound - lowerBound + 1);
}

double generateDoubleNumberInRange(double lowerBound, double upperBound) {
    double f = (double)rand() / RAND_MAX;
    return lowerBound + f * (upperBound - lowerBound);
}
```

TSP_Greedy_Solver.cpp

```
#include "TSP_Greedy_Solver.h"

TSPGreedySolver::TSPGreedySolver(FullGraph *&graph) {
    graphToSolve = graph;
}

Path TSPGreedySolver::solve() {
    Path result;

    vector<Vertex*> vertexes = graphToSolve->getVertexes();
    Vertex* startVertex = vertexes.at(0);
```

```
        Vertex* currentVertex = vertexes.at(0);

        for(int i = 0; i < vertexes.size(); ++i) {
            Edge* nextEdge = findTheShortestEdge(currentVertex, result);

            if(!nextEdge)
                nextEdge = findLastEdge(startVertex, currentVertex);

            result.addEdge(nextEdge);
            selectNextVertex(currentVertex, nextEdge);
        }

        return result;
    }

    Edge *TSPGreedySolver::findTheShortestEdge(Vertex *vertex, Path &passedPath) {
        int minimalLength = INT32_MAX;
        Edge *resultEdge = nullptr;
        for(auto edge : vertex->getIncidentEdges()) {
            if(edge->getLength() < minimalLength && !passedPath.containsVertexes(edge->getConnectedVertexes())) {
                minimalLength = edge->getLength();
                resultEdge = edge;
            }
        }
        return resultEdge;
    }
```

```
void TSPGreedySolver::selectNextVertex(Vertex *&currentVertex, Edge *edge) {
    vector<Vertex*> connectedVertexes = edge->getConnectedVertexes();
    if(currentVertex == connectedVertexes.at(0))
        currentVertex = connectedVertexes.at(1);
    else
        currentVertex = connectedVertexes.at(0);
}

Edge *TSPGreedySolver::findLastEdge(Vertex *startVertex, Vertex *currentVertex) {
    for(auto edge : currentVertex->getIncidentEdges()) {
        if(edge->contains(startVertex))
            return edge;
    }
}
```

Vertex_Edge_Path.cpp

```
#include "Vertex_Edge_Path.h"

Vertex::Vertex(int number) {
    this->number = number;
}

bool Vertex::operator==(const Vertex &obj) {
    return this->number == obj.number;
}
void Vertex::addIncidentEdge(Edge *edgeToAdd) {
    incidentEdges.push_back(edgeToAdd);
}
```

```
vector<Edge *> Vertex::getIncidentEdges() {
    return incidentEdges;
}
int Vertex::getNumber() {
    return number;
}


Edge::Edge(Vertex* vertex1, Vertex* vertex2, int length, float amountOfPheromone, float evaporationCoefficient) {
    connectedVertexes.push_back(vertex1);
    connectedVertexes.push_back(vertex2);
    this->length = length;
    this->amountOfPheromone = amountOfPheromone;
    this->evaporationCoefficient = evaporationCoefficient;
}

void Edge::addPheromone(float additionalPheromone) {
    amountOfPheromone += additionalPheromone;
}
void Edge::evaporatePheromone() {
    amountOfPheromone = (1-evaporationCoefficient)*amountOfPheromone;
}

int Edge::getLength() const {
    return length;
}
float Edge::getAmountOfPheromone() const {
    return amountOfPheromone;
```

```
}
vector<Vertex *> Edge::getConnectedVertexes() {
   return connectedVertexes;
}
bool Edge::operator==(const Edge &obj) {
   return (this->connectedVertexes[0] == obj.connectedVertexes[0] &&
           this->connectedVertexes[1] == obj.connectedVertexes[1]) ||
           (this->connectedVertexes[0] == obj.connectedVertexes[1] &&
            this->connectedVertexes[1] == obj.connectedVertexes[0]);
}

bool Edge::contains(Vertex *vertex) const {
   return (*connectedVertexes[0] == *vertex) ||
           (*connectedVertexes[1] == *vertex);
}

Path::Path() {
   amountOfEdges = 0;
   length = 0;
}
void Path::addEdge(Edge *edge) {
   ++amountOfEdges;
   length += edge->getLength();
   edges.push_back(edge);
}

void Path::clear() {
   amountOfEdges = 0;
   length = 0;
   edges.clear();
```

```
}

int Path::getLength() const {
    return length;
}

int Path::getAmountOfEdges() const {
    return amountOfEdges;
}

Edge *&Path::operator[](int index) {
    return edges.at(index);
}

bool Path::containsVertexes(vector<Vertex *> vertexes) const {
    for(auto vertex : vertexes) {
        bool contains = false;

        for(auto edge : edges) {
            if(edge->contains(vertex)) {
                contains = true;
                break;
            }
        }

        if(!contains)
            return false;
    }

    return true;
```

```
}

bool Path::operator<(const Path &obj) {
   return length < obj.length;
}

void Path::display() {
   Vertex *vertex1, *vertex2;
   vector<Vertex*> connectedVertexes1 = edges.at(0)->getConnectedVertexes();
   vector<Vertex*> connectedVertexes2 = edges.at(1)->getConnectedVertexes();
   if((connectedVertexes1.at(0) == connectedVertexes2.at(0)) || (connectedVertexes1.at(0) == connectedVertexes2.at(1))) {
      vertex1 = connectedVertexes1.at(1);
      vertex2 = connectedVertexes1.at(0);
   } else {
      vertex1 = connectedVertexes1.at(0);
      vertex2 = connectedVertexes1.at(1);
   }
   cout<<vertex1->getNumber()<<" -> "<<vertex2->getNumber()<<", ";
   for(int i = 1; i < edges.size(); ++i) {
      connectedVertexes1 = edges.at(i)->getConnectedVertexes();

      if(connectedVertexes1.at(0) == vertex2) {
         vertex1 = connectedVertexes1.at(0);
         vertex2 = connectedVertexes1.at(1);
      } else {
         vertex1 = connectedVertexes1.at(1);
         vertex2 = connectedVertexes1.at(0);
```

```
            }
            cout<<vertex1->getNumber()<<" -> "<<vertex2->getNumber()<<", ";
        }
        cout<<endl<<"Length: "<<length<<endl;
    }
```

Lab4_tests.cpp

```
#include "pch.h"
#include "CppUnitTest.h"
#include "../Lab4/FullGraph.h"
#include "../Lab4/Vertex_Edge_Path.h"
using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace Lab4tests
{
    TEST_CLASS(FullGraphTests)
    {
    public:

        TEST_METHOD(getAmountOfVertexesTest) {
            int amountOfVertexes = 10;
            FullGraph graph(amountOfVertexes);
            Assert::AreEqual(amountOfVertexes,
graph.getAmountOfVertexes());
        }
    };

    TEST_CLASS(VertexTests)
    {
    public:
```

```
        TEST_METHOD(getNumberTest) {
            int number = 1;
            Vertex vertex(number);
            Assert::AreEqual(number, vertex.getNumber());
        }

        TEST_METHOD(equalVertexesComparrisonTest) {
            int number = 1;
            Vertex vertex1(number);
            Vertex vertex2(number);
            Assert::IsTrue(vertex1 == vertex2);
        }

        TEST_METHOD(nonEqualVertexesComparrisonTest) {
            int number1 = 1;
            int number2 = 2;
            Vertex vertex1(number1);
            Vertex vertex2(number2);
            Assert::IsFalse(vertex1 == vertex2);
        }
};

TEST_CLASS(EdgeTests)
{
private:
        Edge* edge;
        Vertex* vertex1;
        Vertex* vertex2;
        int length;
```

```
        float amountOfPheromone;
        float evaporationCoefficient;
    public:
        EdgeTests() {
            int number1 = 1;
            int number2 = 2;
            vertex1 = new Vertex(number1);
            vertex2 = new Vertex(number2);
            length = 10;
            amountOfPheromone = 10;
            evaporationCoefficient = 0.5;
            edge = new Edge(vertex1, vertex2, length, amountOfPheromone, evaporationCoefficient);
        }

        TEST_METHOD(getLengthTest) {
            Assert::AreEqual(length, edge->getLength());
        }
        TEST_METHOD(getAmountOfPheromoneTest) {
            Assert::AreEqual(amountOfPheromone, edge->getAmountOfPheromone());
        }
        TEST_METHOD(evaporatePheromoneTest) {
            edge->evaporatePheromone();

    Assert::AreEqual(amountOfPheromone*evaporationCoefficient, edge->getAmountOfPheromone());
        }
        TEST_METHOD(addPheromoneTest) {
            float additionalPheromone = 5;
```

```
            edge->addPheromone(5);

        Assert::AreEqual(amountOfPheromone+additionalPheromone, edge->getAmountOfPheromone());
        }

        TEST_METHOD(containsExistingTest) {
            Assert::IsTrue(edge->contains(vertex1));
        }
        TEST_METHOD(containsNonExistingTest) {
            Vertex vertex(5);
            Assert::IsFalse(edge->contains(&vertex));
        }
        TEST_METHOD(getConnectedVertexesTest) {
            vector<Vertex*> connectedVertexes = edge->getConnectedVertexes();
            Assert::IsTrue(vertex1 == connectedVertexes.at(0));
            Assert::IsTrue(vertex2 == connectedVertexes.at(1));
        }

        TEST_METHOD(equalEdgesComparrisonTest) {
            Edge testEdge(vertex1, vertex2, length, amountOfPheromone, evaporationCoefficient);
            Assert::IsTrue(*edge == testEdge);
        }

        TEST_METHOD(nonEqualEdgesComparrisonTest) {
            int number1 = 5;
            int number2 = 6;
            Vertex testVertex1(number1);
```

```
            Vertex testVertex2(number2);
            Edge testEdge(&testVertex1, &testVertex2, length,
amountOfPheromone, evaporationCoefficient);
            Assert::IsFalse(*edge == testEdge);
        }
        ~EdgeTests() {
            delete edge;
            delete vertex1;
            delete vertex2;
        }
    };

    TEST_CLASS(PathTests) {
        Path* path;
        Vertex* vertex1, * vertex2, * vertex3;
        int length, amountOfEdges;
        Edge* edge1, * edge2;
    public:
        PathTests() {
            path = new Path();
            int number1 = 1, number2 = 2, number3 = 3;
            vertex1 = new Vertex(number1);
            vertex2 = new Vertex(number2);
            vertex3 = new Vertex(number3);

            int length1 = 10, length2 = 15;
            length = length1 + length2;

            edge1 = new Edge(vertex1, vertex2, length1, 0, 0);
            edge2 = new Edge(vertex2, vertex3, length2, 0, 0);
```

```
            path->addEdge(edge1);
            path->addEdge(edge2);

            amountOfEdges = 2;
        }

        TEST_METHOD(getAmountOfEdgesTest) {
            Assert::AreEqual(amountOfEdges, path->getAmountOfEdges());
        }
        TEST_METHOD(getLengthTest) {
            Assert::AreEqual(length, path->getLength());
        }
        TEST_METHOD(clearAmountOfEdgesTest) {
            path->clear();
            Assert::AreEqual(0, path->getAmountOfEdges());
        }
        TEST_METHOD(clearLengthTest) {
            path->clear();
            Assert::AreEqual(0, path->getLength());
        }
        TEST_METHOD(containsExistingVertexesTest) {
            vector<Vertex*> vertexes;
            vertexes.push_back(vertex1);
            vertexes.push_back(vertex2);

            Assert::IsTrue(path->containsVertexes(vertexes));
        }
        TEST_METHOD(containsNonExistingVertexesTest) {
```

```
            vector<Vertex*> vertexes;
            Vertex testVertex(7);
            vertexes.push_back(vertex1);
            vertexes.push_back(&testVertex);
            Assert::IsFalse(path->containsVertexes(vertexes));
        }
        TEST_METHOD(comparisonWithLongerTest) {
            Edge testEdge(vertex1, vertex2, length + 10, 0, 0);
            Path testPath;
            testPath.addEdge(&testEdge);

            Assert::IsTrue(*path < testPath);
        }
        TEST_METHOD(comparisonWithShorterTest) {
            Edge testEdge(vertex1, vertex2, length - 10, 0, 0);
            Path testPath;
            testPath.addEdge(&testEdge);

            Assert::IsFalse(*path < testPath);
        }
        ~PathTests() {
            delete path;
            delete vertex1;
            delete vertex2;
            delete vertex3;
            delete edge1;
            delete edge2;
        }
    };
}
```

3.1.2 Приклади роботи

На рисунку 3.1 показано приклад роботи програми.

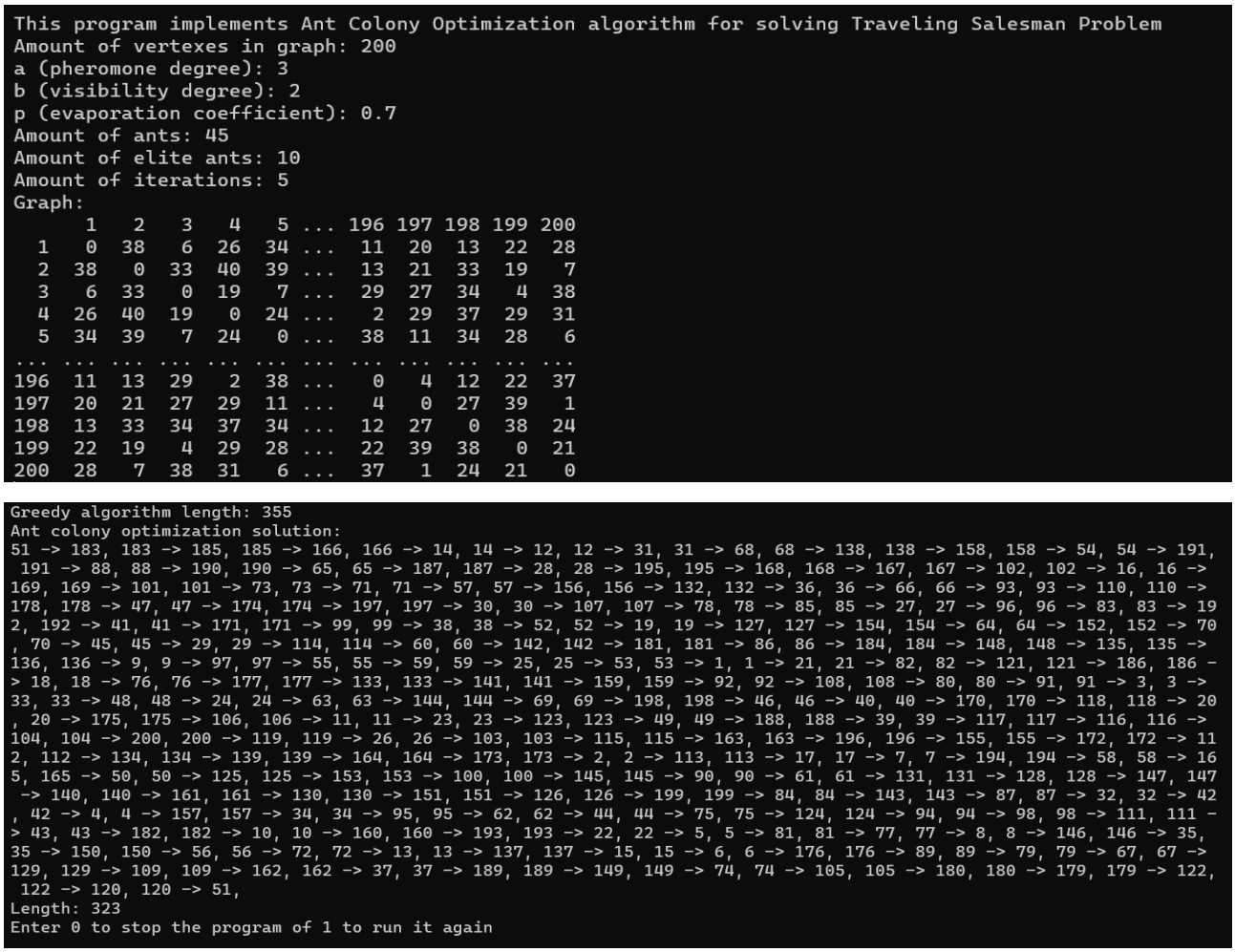

Рисунок 3.1 – Робота програми

Тестування алгоритму

3.1.3 Значення цільової функції зі збільшенням кількості ітерацій

У таблиці 3.1 наведено значення цільової функції зі збільшенням кількості ітерацій.

Номер ітерації	Значення цільової функції (довжина шляху)
1	642
6	297
11	272
16	270
21	270
26	267
31	264
36	258
41	258
46	258
51	258
56	258
61	264
66	261
71	258
76	258
81	258
86	264
91	258
96	259

3.1.4 Графіки залежності розв'язку від числа ітерацій

На рисунку 3.3 наведений графік, який показує якість отриманого розв'язку.

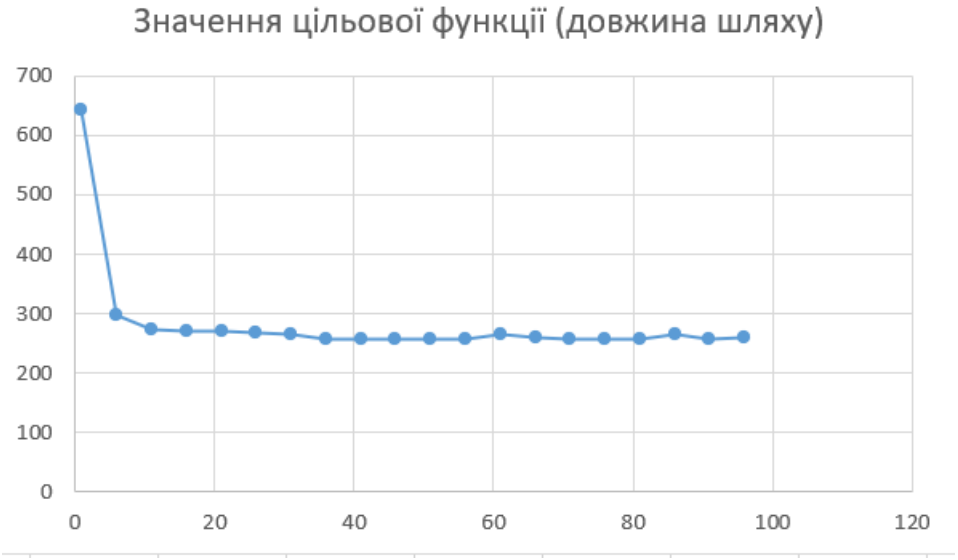

Рисунок 3.3 – Графік залежності розв'язку від числа ітерацій

ВИСНОВОК

В рамках даної лабораторної роботи я реалізував мурашиний алгоритм для розв’язку задачі комівояжера засобами мови програмування C++. Провівши випробування з 100 ітерацій алгоритму, я переконався, що вже через кілька ітерацій мурашиний алгоритм знаходить розв’язок оптимальніший, ніж розв’язок, знайдений за допомогою жадібного алгоритму. Отримані результати я заніс в таблицю та намалював на основі цих даних графік залежності значення цільової функції (у моєму випадку – довжина шляху) від кількості ітерацій алгоритму.

КРИТЕРІЇ ОЦІНЮВАННЯ

При здачі лабораторної роботи до 10.12.2023 включно максимальний бал дорівнює – 5. Після 10.12.2023 максимальний бал дорівнює – 4,5.

Критерії оцінювання у відсотках від максимального балу:

- програмна реалізація алгоритму – 55%;
- робота з гіт – 20%;
- тестування алгоритму– 20%;
- висновок – 5%.

+1 додатковий бал можна отримати за виконання роботи до 3.12.2023