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 № 5 з дисципліни
«Проектування алгоритмів»

„Проектування і аналіз алгоритмів для вирішення NP-складних задач ч.2”

Виконав(ла) *ІП-13 Кисельов Микита* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Сопов Олексій Олександрович* _____
(прізвище, ім'я, по батькові)

Київ 2022

ЗМІСТ

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи – вивчити основні підходи розробки метаевристичних алгоритмів для типових прикладних задач. Опрацювати методологію підбору прийнятних параметрів алгоритму.

2 ЗАВДАННЯ

Згідно варіанту, формалізувати алгоритм вирішення задачі відповідно загальної методології.

Записати розроблений алгоритм у покроковому вигляді. З достатнім степенем деталізації.

Виконати його програмну реалізацію на будь-якій мові програмування.

Перелік задач наведено у таблиці 2.1.

Перелік алгоритмів і досліджуваних параметрів у таблиці 2.2.

Задача і алгоритм наведені в таблиці 2.3.

Змінюючи параметри алгоритму, визначити кращі вхідні параметри алгоритму. Для цього необхідно:

– обрати критерій зупинки алгоритму (кількість ітерацій або значення ЦФ);

– зафіксувати усі параметри крім одного і змінювати цей параметр, поки не буде досягнуто пікової ефективності;

– після цього параметр фіксується і змінюються інші параметри;

– далі повторюємо процедуру спочатку, з першого зафіксованого параметру;

– зупиняємось коли будуть знайдені оптимальні параметри для даної задачі або встановлена залежність одних параметрів від інших.

Зробити узагальнений висновок в якому обов’язково описати залежність якості розв’язку від вхідних параметрів.

Таблиця 2.1 – Прикладні задачі

№	Задача
2	**Задача комівояжера** (300 вершин, відстань між вершинами випадкова від 5 до 150) полягає у знаходженні найвигіднішого маршруту, що проходить через вказані міста хоча б по одному разу. В умовах завдання вказуються критерій вигідності маршруту (найкоротший, найдешевший, сукупний критерій тощо) і відповідні матриці відстаней, вартості тощо. Зазвичай задано, що маршрут повинен проходити через кожне місто тільки один раз, в такому випадку розв'язок знаходиться серед гамільтонових циклів. **Розглядається симетричний, асиметричний та змішаний варіанти.** В загальному випадку, асиметрична задача комівояжера відрізняється тим, що ребра між вершинами можуть мати різну вагу в залежності від напряму, тобто, задача моделюється орієнтованим графом. Таким чином, окрім ваги ребер графа, слід також зважати і на те, в якому напрямку знаходяться ребра. У випадку симетричної задачі всі пари ребер між одними й тими самими вершинами мають однакову вагу. У випадку реальних міст може бути як симетричною, так і асиметричною в залежності від тривалості або довжини маршрутів і напряму руху. Застосування: – доставка товарів (в цьому випадку може бути більш доречна постановка транспортної задачі - доставка в кілька магазинів з декількох складів); – доставка води; – моніторинг об'єктів; – поповнення банкоматів готівкою; – збір співробітників для доставки вахтовим методом.

Таблиця 2.2 – Варіанти алгоритмів і досліджувані параметри

№	**Алгоритми і досліджувані параметри**
3	**Бджолиний алгоритм:** – кількість ділянок; – кількість бджіл (фуражирів і розвідників).

Таблиця 2.3 – Варіанти задач і алгоритмів

№	**Задачі і алгоритми**
12	Задача комівояжера (симетрична мережа) + Бджолиний алгоритм

3 ВИКОНАННЯ

3.1 Покроковий алгоритм

1. ПОЧАТОК
2. СТВОРИТИ ПОЧАТКОВУ ПОПУЛЯЦІЮ ІЗ ВИПАДКОВИМИ РОЗВ’ЯЗКАМИ (ДІЛЯНКАМИ)
3. ПОВТОРИРИ ПОКИ НЕ ДОСЯГНУТИЙ КРИТЕРІЙ ОСТАНОВКИ
   - 3.1.БДЖОЛИ-РОЗВІДНИКИ ТАНЦЮЮТЬ ПРО ТЕ, ЯКІ ЗНАЙШЛИ РОЗВ’ЯЗКИ
   - 3.2.ІНШІ ОЖИДАЮЧИ БДЖОЛИ В ЗАЛЕЖНОСТІ ВІД РІВНЯ НЕКТАРУ ЗМІНЮЮТЬ НАПРЯМ ДО РОЗВ’ЯЗКІВ ТА СТАЮТЬ ФУРАЖИРАМИ
   - 3.3.ПОВТОРИТИ ДЛЯ КОЖНОЇ БДЖІЛИ- ФУРАЖИРА
     - 3.3.1. ЯКЩО БДЖІЛА НЕ ПОМИЛИЛАСЬ ЗА ПЕВНОЮ ЙМОВІРНІСТЮ ТО
       - 3.3.1.1. ПРИСВОЇТИ ДЛЯ НЕЇ НАЙКРАЩИМ РОЗВ’ЯЗКОМ АКТУАЛЬНИЙ ШЛЯХ
     - 3.3.2. ІНАКШЕ
       - 3.3.2.1. БДЖОЛА ПОВИШАЄ КІЛЬКІСТЬ ПОМИЛОК НА 1
     - 3.3.3. ВСЕ ЯКЩО
     - 3.3.4. ЯКЩО БДЖОЛА НЕВДАЛА ВЖЕ ДЕКІЛЬКА РАЗІВ ТО
       - 3.3.4.1. ПОСТАВИТИ СТАТУС ОЖИДАЮЧОЇ, ОБНУЛИТИ ПОМИЛКИ ТА ДОДАТИ ДО ОЖИДАЮЧИХ
     - 3.3.5. ВСЕ ЯКЩО
   - 3.4.ВСЕ ПОВТОРИТИ
   - 3.5.ОБРАТИ НАЙКРАЩИЙ ШЛЯХ ІЗ ПОЛУЧЕНИХ КОЖНОЮ БДЖОЛОЮ-ФУРАЖИРОМ
4. ВСЕ ПОВТОРИТИ
5. КІНЕЦЬ

3.2 Програмна реалізація алгоритму

3.2.1 Вихідний код

constants.java:

```
package abc;

// найкращі параметри для розв'язку
public interface constants {
    int     MAX_VERTEX      = 300,
            MIN_W           = 5,
            MAX_W           = 150,
            TOTAL_BEES      = 60,
            SCOUT_BEES      = 15,
            ITERATIONS      = 1000,
            FREQ_TO_REPORT  = 20,
            pathStart       = 0,
            pathEnd         = MAX_VERTEX - 1,
            MIN_V_DEGREE    = 1,
            MAX_V_DEGREE    = 30,
            UNL_LIM         = 10;
}
```

Bee.java:

```
package abc;

public class Bee implements constants {
    public enum Status { ONLOOKER, EMPLOYED,
SCOUT }
    private int[] currentPath;
    private Status currentStatus;
    private int currentPathDistance,
unluckyIterateCount;

    public Bee(Status currentStatus) {
        this.currentStatus = currentStatus;
        currentPath = null;
        currentPathDistance = 0;
        unluckyIterateCount = 0;
    }

    public Status getCurrentStatus() {
        return currentStatus;
    }

    public void setCurrentStatus(Status
currentStatus) {
        this.currentStatus = currentStatus;
    }

    public int[] getCurrentPath() {
        return currentPath.clone();
    }

    public int getCurrentPathDistance() {
        return currentPathDistance;
    }

    public void setUnluckyIterateCount(int
unluckyIterateCount) {
        this.unluckyIterateCount =
unluckyIterateCount;
    }

    public void changePath(int[] path, int
pathDistance) {
        currentPath = path;
        currentPathDistance = pathDistance;
        unluckyIterateCount = 0;
    }

    public boolean isUnluckyOverLimit() {
        return unluckyIterateCount > UNL_LIM;
    }

    public void stayIdle() {
        ++unluckyIterateCount;
    }
}
```

Graph.java:

```
package abc;

import org.jetbrains.annotations.Contract;
import org.jetbrains.annotations.NotNull;

import java.util.*;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class Graph {
    public record Edge(int from, int to, int
weight) {}
    private final Random rand = new
Random(2022);

    private final int verticesCount;
    private final ArrayList<Edge> edges;

    public Graph(int verticesCount, int
numEdges) {
        this.edges = new
ArrayList<>(numEdges);
        this.verticesCount = verticesCount;
    }

    public int getEdgeCount() {
        return edges.size();
    }

    public Edge getEdgeOrNullIfNotFound(int
from, int to) {
        for (Edge e: edges)
            if ((e.from() == from && e.to() ==
to) || (e.from() == to && e.to() == from))
                return e;
        return null;
    }

    public boolean noEdge(int from, int to) {
        return getEdgeOrNullIfNotFound(from,
to) == null;
    }

    public void addEdge(@NotNull Edge edge) {
        if (noEdge(edge.from(), edge.to()))
edges.add(edge);
```

```
    }

    public int getWeight(int from, int to) {
        Edge edge =
getEdgeOrNullIfNotFound(from, to);
        return (getEdgeOrNullIfNotFound(from,
to) == null) ? -1 : edge.weight();
    }

    public int[] getNeighbours(int vertex) {
        HashSet<Integer> neighbours = new
HashSet<>();
        for (Edge e : edges) {
            if (e.from() == vertex)
neighbours.add(e.to());
            if (e.to() == vertex)
neighbours.add(e.from());
        }
        return
neighbours.stream().mapToInt(Integer::intValue
).toArray();
    }

    public int measureDistance(int @NotNull []
path) {
        int result = 0;
        for (int i = 0; i < path.length - 1;
++i) {
            int weight = getWeight(path[i],
path[i + 1]);
            if (weight < 0) return -1;
            result += weight;
        }
        return result;
    }

    public int[]
randomPathNullIfNotAvailable(int start, int
dest, int[] visitedVertices) {
        HashSet<Integer> visited =
(visitedVertices == null)
                ? new HashSet<>()
                : new
HashSet<>(IntStream.of(visitedVertices).boxed(
).collect(Collectors.toSet()));
        visited.add(start);
        ArrayList<Integer> path = new
ArrayList<>(List.of(start));
        int currentVertex =
path.get(path.size() - 1);
        while (currentVertex != dest) {
            int[] neighs =
except(getNeighbours(currentVertex),
visited.stream().mapToInt(Integer::intValue).t
oArray());
            if (neighs.length == 0)
                if (path.size() == 1) return
null;
                else path.remove(path.size() -
1);
            else
path.add(neighs[rand.nextInt(neighs.length)]);
            currentVertex =
path.get(path.size() - 1);
            visited.add(currentVertex);
        }
        return
path.stream().mapToInt(Integer::valueOf).toArr
ay();
    }

    public int[] modifyRandomPath(int @NotNull
[] path) {
        int[] unchangedPathPart = null,
changedPathPart = null;
        if (path.length > 2) while
(changedPathPart == null) {
            int changeIndex = rand.nextInt(1,
path.length - 1);
            System.arraycopy(path , 0 ,
unchangedPathPart = new int[changeIndex], 0,
changeIndex);
            changedPathPart =
randomPathNullIfNotAvailable(path[changeIndex]
, path[path.length - 1], unchangedPathPart);
        }
        return
concatTwoIntArrays(unchangedPathPart,
changedPathPart);
    }

    public boolean isValidPath(int @NotNull []
path) {
        for (int i = 0; i < path.length - 1;
++i)
            if (noEdge(path[i], path[i + 1]))
return false;
        return true;
    }

    @Contract(pure = true)
    private int @NotNull []
concatTwoIntArrays(int[] first, int[] second)
{
        int fLen = first == null ? 0 :
first.length, sLen = second == null ? 0 :
second.length, k = -1;
        int[] concat = new int[fLen + sLen];
        for (int i = 0; i < fLen; ++i)
concat[++k] = first[i];
        for (int i = 0; i < sLen; ++i)
concat[++k] = second[i];
        return concat;
    }

    private int[] except(int @NotNull []
first, int[] second) {
        ArrayList<Integer> res = new
ArrayList<>();
        for (int c : first) if
(Arrays.stream(second).noneMatch(n -> n == c))
res.add(c);
        return
res.stream().mapToInt(Integer::valueOf).toArra
y();
    }

    @Override public String toString() {
        StringBuilder sb = new
StringBuilder(verticesCount + " vertices, " +
getEdgeCount() + " edges:\n");
        int k = 0;
        for (Edge e: edges) {
            if (++k % 12 == 0)
sb.append('\n');
            sb.append(e.from()).append('-
').append(e.to()).append(":w").append(e.weight
()).append(", ");
        }
        return sb.replace(sb.length() - 2,
sb.length(), ";").toString();
    }
}
```

AlgorithmImplementation.java:

```
package abc;

import org.jetbrains.annotations.Contract;
import org.jetbrains.annotations.NotNull;

import java.util.*;

class AlgorithmImplementation implements
constants {
    private final Random rand = new Random();
    // ймовірність бездіяльної бджоли
відгукнутися на танець розвідника
    private final double
persuasionProbability, mistakeProbability;
    // за одну ітерацію всі агенти здійснюють
по одній дії
    private final int workerCount, scoutCount,
pathStart, pathEnd, maxCycles, reportEvery;
    private int bestDistance;
    private int[] bestPath;
    Graph graph;
    ArrayList<Bee> scouts, onlookers,
employed;
    HashMap<int[], Integer> scoutedPaths; //
рішення, знайдені розвідниками

    public AlgorithmImplementation(int
workerCount, int scoutCount, Graph graph, int
pathStart, int pathEnd,
                                   int
maxCycles, int reportEvery) {
        this.maxCycles = maxCycles;
        this.reportEvery = reportEvery;
        this.graph = graph;
        this.pathStart = pathStart;
        this.pathEnd = pathEnd;
        this.scoutedPaths = new HashMap<>();
        persuasionProbability = 0.95;
        mistakeProbability = 0.01;
        this.workerCount = workerCount;
        this.scoutCount = scoutCount;
    }

    // створює початкову популяцію: розвідники
та фуражири, які очікують у вулику
    private void produceInitialPopulation() {
        employed = new ArrayList<>();
        onlookers = new ArrayList<>();
        scouts = new ArrayList<>(scoutCount);
        for (int i = 0; i < workerCount; ++i)
            onlookers.add(new
Bee(Bee.Status.ONLOOKER));
        for (int i = 0; i < scoutCount; ++i)
            scouts.add(new
Bee(Bee.Status.SCOUT));
    }

    public void solve() {
        produceInitialPopulation();
        long start =
System.currentTimeMillis();
        for (int i = 0; i < maxCycles;) {
            for (int k = 0; k < reportEvery;
++k) {
                scoutPhase();
                onlookerPhase();
                employedPhase();
                keepBestPath();
            }
            System.out.printf("Iteration
#%d\n%s\n", i += reportEvery, this);
        }
        System.out.printf("Solution time - %8d
seconds\n", (System.currentTimeMillis() -
start) / 1000);
        System.out.printf("Best path: %s\n" +
                "Is valid? - %b.",
Arrays.toString(bestPath),
graph.isValidPath(bestPath));
    }

    // якщо робітникам вдалося знайти краще
рішення, запам'ятовуємо його
    private void keepBestPath() {
        for (Bee employ: employed)
            if (bestPath == null ||
employ.getCurrentPathDistance() <
bestDistance) {
                bestDistance =
employ.getCurrentPathDistance();
                bestPath =
employ.getCurrentPath();
            }
    }

    private void employedPhase() {
        for (Bee employ: employed)
processEmployedBee(employ);
        employed.removeIf(employ ->
employ.getCurrentStatus() !=
Bee.Status.EMPLOYED);
    }

    private void processEmployedBee(@NotNull
Bee employedBee) {
        int[] neighborSolution =
graph.modifyRandomPath(employedBee.getCurrentP
ath());
        int neighborDistance =
graph.measureDistance(neighborSolution);
        boolean isMistaken = rand.nextDouble()
< mistakeProbability,
                foundNewSolution =
neighborDistance <
employedBee.getCurrentPathDistance();
        if (foundNewSolution ^ isMistaken)
employedBee.changePath(neighborSolution,
neighborDistance);
        else employedBee.stayIdle();
        // бджола-невдаха припиняє спроби
поліпшити шлях
        if (employedBee.isUnluckyOverLimit())
{

employedBee.setCurrentStatus(Bee.Status.ONLOOK
ER);

employedBee.setUnluckyIterateCount(0);
            onlookers.add(employedBee);
        }
    }

    private void onlookerPhase() {
        HashMap<Double, int[]> rollingWheel =
createScoutedPathsRollingWheel();
        for (Bee onlooker: onlookers)
            processOnlookerBee(onlooker,
rollingWheel);
        onlookers.removeIf(onlooker ->
onlooker.getCurrentStatus() !=
Bee.Status.ONLOOKER);
    }

    // будує рулетку рішень розвідників =
проєктує кожне рішення в відрізок усередині
[0..1]
    // довжини відрізків пропорційні
добротності значень цільової функції
    private @NotNull HashMap<Double, int[]>
createScoutedPathsRollingWheel() {
        int distanceSum = 0;
        for (int[] path:
scoutedPaths.keySet())
            distanceSum +=
```

```
scoutedPaths.get(path);
        HashMap<Double, int[]> res = new
HashMap<>();
        double prevProb = 0f;
        for (int[] path:
scoutedPaths.keySet()) {
            double prob = 1f -
scoutedPaths.get(path) / (double)distanceSum;
            res.put(prevProb + prob, path);
            prevProb += prob;
        }
        return res;
    }

    private void scoutPhase() {
        scoutedPaths.clear();
        for (Bee scout: scouts)
processScoutBee(scout);
    }

    private void processScoutBee(Bee bee) {
        if (onlookers.size() == 0) return; //
якщо у вулику немає вільних робітників, немає
сенсу шукати рішення
        int[] randomSolution =
graph.randomPathNullIfNotAvailable(pathStart,
pathEnd, null);
        bee.changePath(randomSolution,
graph.measureDistance(randomSolution));
        scoutedPaths.put(bee.getCurrentPath(),
bee.getCurrentPathDistance()); // бджола
танцює до вулика про те, який шлях знайшла
    }

    private void processOnlookerBee(Bee bee,
HashMap<Double, int[]> rollingWheel) {
        if (rand.nextDouble() <
persuasionProbability) {
            int[] path =
getPathFromWheel(rand.nextDouble(),
rollingWheel);
            bee.changePath(path,
scoutedPaths.get(path));

bee.setCurrentStatus(Bee.Status.EMPLOYED);
            employed.add(bee);
        }
    }
    // обчислює потрапляння точки в відрізок
на рулетці й отримує звідти відповідне рішення
    private int[] getPathFromWheel(double
randomDouble, @NotNull HashMap<Double, int[]>
rollingWheel) {
        int[] res = null;
        double[] wheelRange =
concatTwoDoubleArrays(new Double[]{.0},
rollingWheel.keySet().toArray(new Double[0]));
        Arrays.sort(wheelRange);
        for (int i = 0; i < wheelRange.length
- 1; ++i)
            if (randomDouble >= wheelRange[i]
&& randomDouble < wheelRange[i + 1])
                res =
rollingWheel.get(wheelRange[i + 1]);
        if (res == null)
            res =
rollingWheel.values().iterator().next();
        return res;
    }

    @Contract(pure = true)
    private double @NotNull []
concatTwoDoubleArrays(Double[] first, Double[]
second) {
        int fLen = first == null ? 0 :
first.length, sLen = second == null ? 0 :
second.length, k = 0;
        double[] concat = new double[fLen +
sLen];
        for (int i = 0; i < fLen; ++i)
concat[k++] = first[i];
        for (int i = 0; i < sLen; ++i)
concat[k++] = second[i];
        return concat;
    }

    @Override public String toString() {
        StringBuilder s = new
StringBuilder(onlookers.size() + " onlookers,
");
        s.append(employed.size()).append("
employed, ").append(scouts.size()).append("
scouts, ")
        .append(scouts.size() +
onlookers.size() + employed.size()).append("
total\n").append("Best path found: ");
        if (bestPath != null) {
            for (int i = 0; i <
bestPath.length - 1; ++i)

s.append(bestPath[i]).append("-");
            s.append(bestPath[bestPath.length
- 1]).append("\n")
            .append("Path distance:
").append(bestDistance).append("\n");
        } else s.append("none");
        return s.toString();
    }
}
```

UsualBeeColonyAlgorithmImplementation.java:

```
package abc;

import org.jetbrains.annotations.NotNull;

import java.util.Random;

public class UsualBeeColonyAlgorithmImplementation implements constants {
    public static void main(String[] args) {
        Graph graph = loadGraph();
        System.out.println("Loaded graph:\n" + graph);
        AlgorithmImplementation ai = new AlgorithmImplementation(TOTAL_BEES - SCOUT_BEES,
SCOUT_BEES, graph,
                pathStart, pathEnd, ITERATIONS, FREQ_TO_REPORT);
        ai.solve();
    }

    private static @NotNull Graph loadGraph() {
        Random rand = new Random(2022);
        Graph result = new Graph(MAX_VERTEX, MAX_VERTEX * (MAX_V_DEGREE + MIN_V_DEGREE - 1));
```

```
        for (int i = 0; i < MAX_VERTEX * (MAX_V_DEGREE + MIN_V_DEGREE - 1) >> 1; ++i) {
            int from = rand.nextInt(0, MAX_VERTEX), to, weight = rand.nextInt(MIN_W, MAX_W);
            do to = rand.nextInt(0, MAX_VERTEX); while (to == from);
            // реалізація симетричної мережі
            result.addEdge(new Graph.Edge(from, to, weight));
            result.addEdge(new Graph.Edge(to, from, weight));
        }
        return result;
    }
}
```

3.2.2 Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

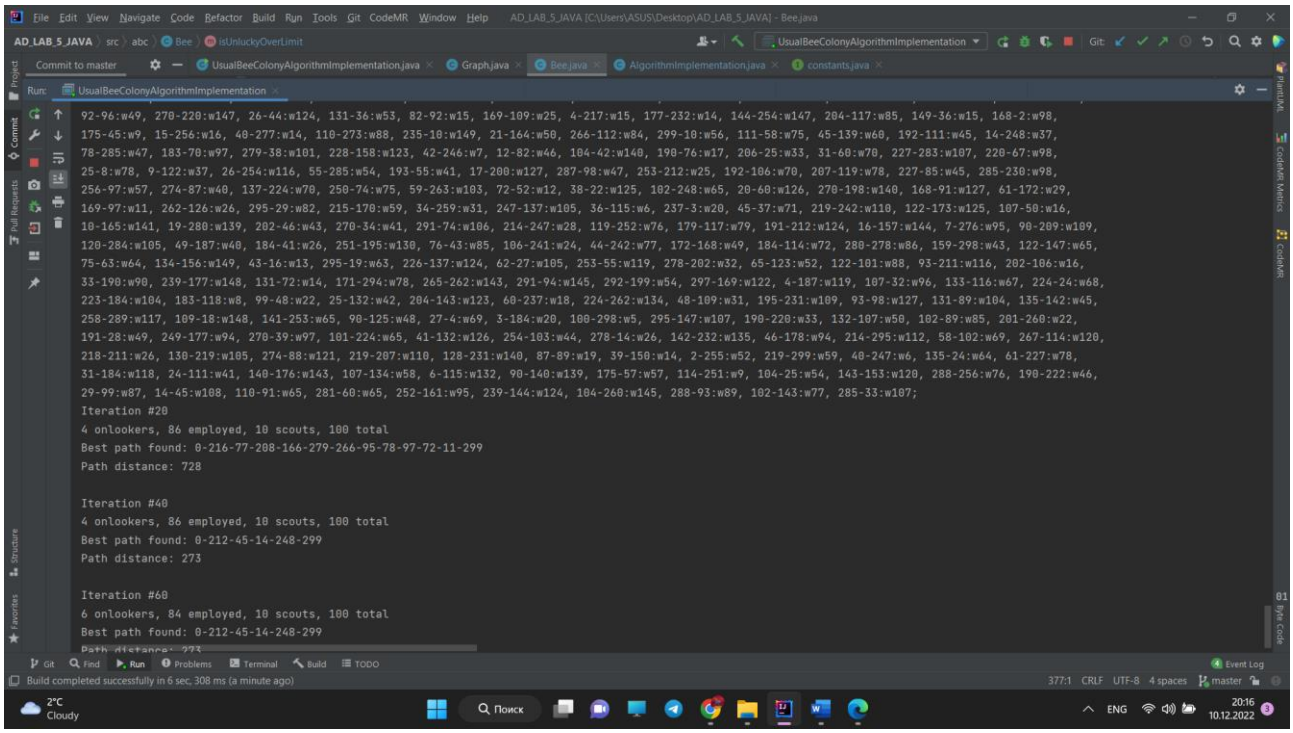

Рисунок 3.1 – початок роботи алгоритму, на якому вже видно покращення рішення

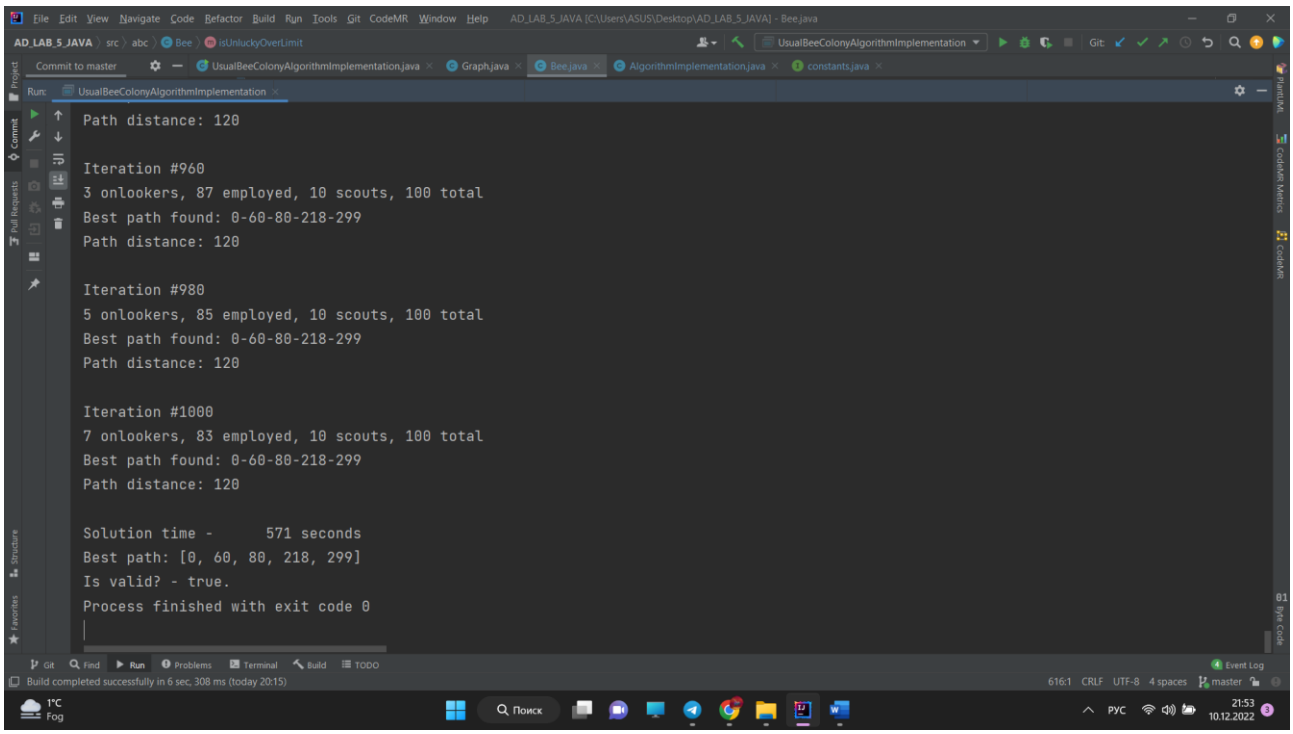

Рисунок 3.2 – результат роботи алгоритму, який показує, що начальне рішення було на 700 дорожче

3.3 Тестування алгоритму

Почнемо тестування з дослідження параметру “кількість ділянок”. Відповідає за кількість ділянок, які розв’язуються до початку основного алгоритму жадібним методом, які потім ми будемо покращувати. Значення цього параметру будемо змінювати від 5 до 30 з кроком 5. Кількість бджіл візьмемо максимальну. Для кожного значення запустимо алгоритм 5 разів та запишемо середні значення. Зафіксуємо всі результати в таблиці 3.3.1 нижче та середні значення відобразимо на рисунку 3.3.2.

Таблиця 3.3.1 - залежність часу від кількості ділянок.

Кількість ділянок	**Номер дослідження**	**Час розв'язку у секундах**
5	1	635
	2	597
	3	615
	4	610
	5	684
	Середнє	**628,2**
10	1	534
	2	419
	3	555
	4	481
	5	643
	Середнє	**526,4**
15	1	491
	2	514
	3	505
	4	517
	5	538
	Середнє	**513**
20	1	532
	2	503
	3	510
	4	578
	5	548
	Середнє	**534,2**
25	1	541
	2	526
	3	523
	4	589
	5	534
	Середнє	**542,6**

Продовження таблиці 3.3.1.

Кількість ділянок	Номер дослідження	Час розв'язку у секундах
30	1	614
	2	578
	3	604
	4	631
	5	597
	Середнє	**604,8**

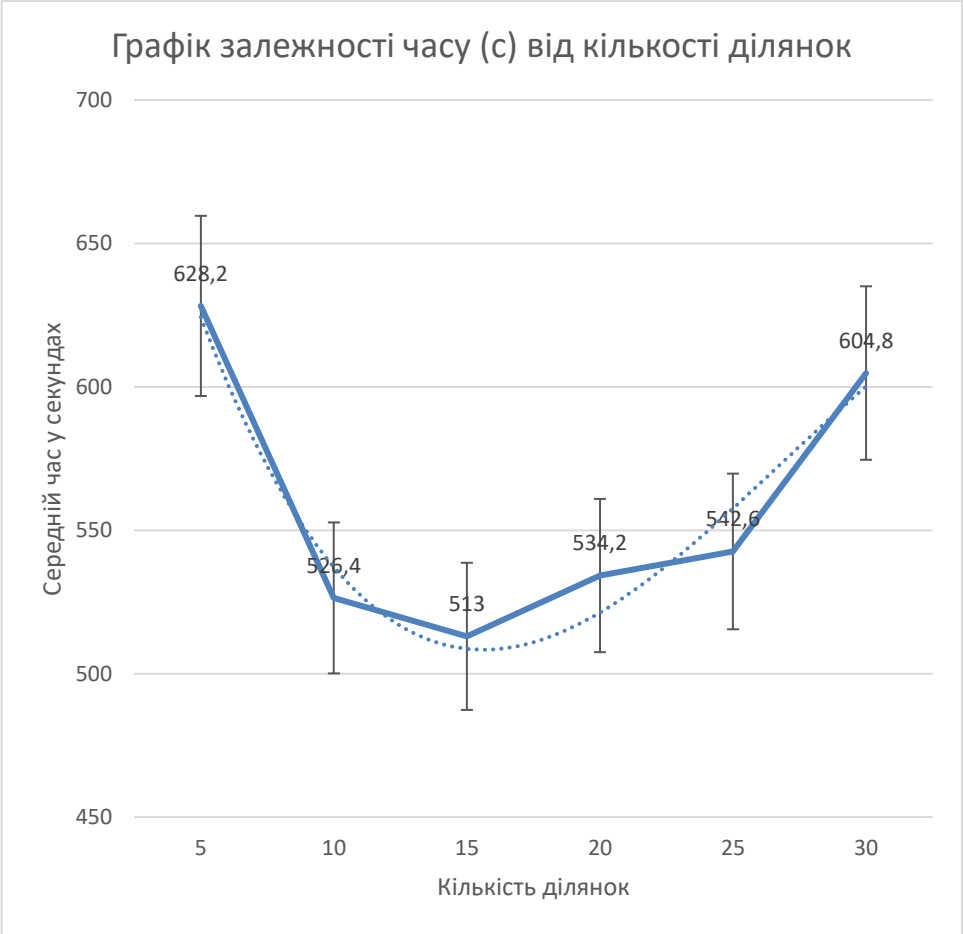

Рисунок 3.3.2 – залежність часу від кількості ділянок.

Таким чином Я отримав оптимальне значення для кількості ділянок – 15 при максимальної (більше 1/3 від вершин немає сенсу) кількості бджіл. При 5 алгоритм стає майже жадібним а при більших за 15 значеннях алгоритм стає неоптимальним.

Далі проведемо тестування з дослідження параметру “кількість бджіл” із значенням кількості ділянок, для якого час був мінімальним, тобто 15. Відповідає за кількість бджіл (фуражирів та розвідників), які будуть виконувати певні дії для покращенні розв’язку. Значення цього параметру будемо змінювати від 50 до 150 з кроком 10 бо менше або більше немає сенсу змінювати через основи роботи алгоритму або значно сповільнює роботу. Для кожного значення запустимо алгоритм 5 разів та запишемо середні значення. Зафіксуємо всі результати в таблиці 3.3.3 нижче та середні значення відобразимо на рисунку 3.3.4.

Оцінку будемо рахувати за наступною формулою:

$$\text{Оцінка} = \frac{\frac{\text{Вартість} + \text{час}}{2} * \text{Вартість}}{30000}$$

Таблиця 3.3.3 - залежність часу від кількості ділянок.

Кількість бджіл	Номер дослідження	Оцінка
10	1	0,829733333
	2	0,8428
	3	0,836266667
	4	0,849333333
	5	0,826466667
	Середнє	0,83692
20	1	0,7742
	2	0,8232
	3	0,790533333
	4	0,777466667
	5	0,806866667
	Середнє	0,794453333
30	1	0,550333333
	2	0,639166667
	3	0,533
	4	0,5915
	5	0,528666667
	Середнє	0,568533333
40	1	0,743166667
	2	0,836333333
	3	0,754
	4	0,806
	5	0,741
	Середнє	0,7761

Продовження Таблиці 3.3.3.

Кількість бджіл	Номер дослідження	Оцінка
50	1	0,983666667
	2	0,918666667
	3	0,884
	4	0,953333333
	5	0,927333333
	Середнє	0,9334
60	1	0,1092
	2	0,105
	3	0,1215
	4	0,1089
	5	0,1176
	Середнє	0,11244
70	1	0,129
	2	0,1245
	3	0,1221
	4	0,1218
	5	0,126
	Середнє	0,12468
80	1	0,1515
	2	0,1302
	3	0,1266
	4	0,1245
	5	0,141
	Середнє	0,13476
90	1	0,1605
	2	0,1548
	3	0,2346
	4	0,2133
	5	0,2436
	Середнє	0,20136
100	1	0,1983
	2	0,1737
	3	0,2178
	4	0,249
	5	0,1695
	Середнє	0,20166

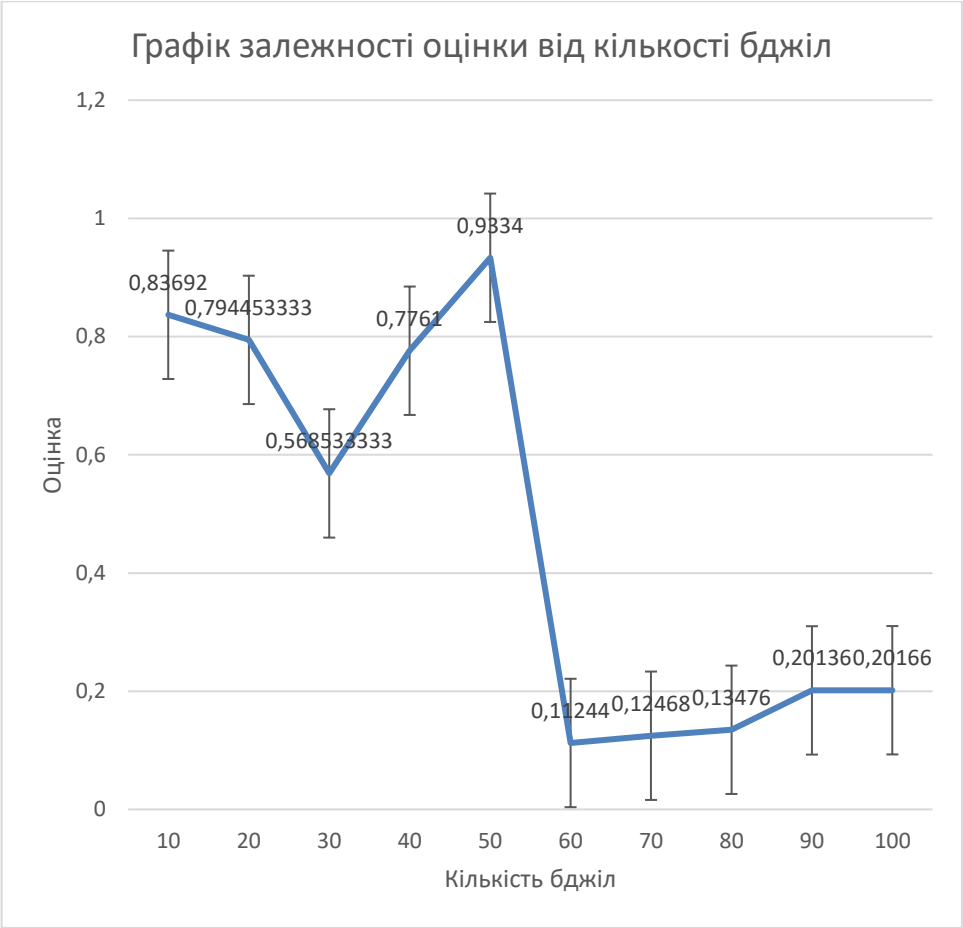

Рисунок 3.3.4 – залежність оцінки від кількості бджіл.

Таким чином Я отримав оптимальне значення для кількості бджіл – 60 при оптимальної кількості ділянок - 15. При значеннях, менших за 60 алгоритм знаходить шлях, але не найкращий, при більших значеннях алгоритм працює довше.

ВИСНОВОК

В рамках даної лабораторної роботи Я вирішив задачу Комівояжера на 300 вершин бджолиним алгоритмом (Bees algorithm).

Також провів дослід, при якому знайшов оптимальні параметри для вирішення цієї задачі. Досліджуваними параметрами були – кількість ділянок та кількість бджіл (фуражирів та розвідників).

Результатами цієї роботи стало те, що бджолиний алгоритм (Bees algorithm) на 300 вершин вирішує задачу Комівояжера (без від'ємних циклів) краще з 15 ділянками та 60 бджіл, інакше алгоритм або знаходить неоптимальний варіант, або шукає його доволі довго.

КРИТЕРІЇ ОЦІНЮВАННЯ

При здачі лабораторної роботи до 11.12.2022 включно максимальний бал дорівнює – 5. Після 11.12.2022 максимальний бал дорівнює – 1.

Критерії оцінювання у відсотках від максимального балу:

- покроковий алгоритм – 15%;
- програмна реалізація алгоритму – 50%;
- тестування алгоритму– 30%;
- висновок – 5%.