

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-22 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	3
2	ЗАВДАННЯ	4
3	ВИКОНАННЯ.....	11
3.1	ПОКРОКОВИЙ АЛГОРИТМ	11
3.2	ПРОГРАМНА РЕАЛІЗАЦІЯ АЛГОРИТМУ	13
3.2.1	<i>Вихідний код.....</i>	<i>13</i>
3.2.2	<i>Приклади роботи</i>	<i>37</i>
3.3	ТЕСТУВАННЯ АЛГОРИТМУ	38
	ВИСНОВОК	41
	КРИТЕРІЇ ОЦІНЮВАННЯ	42

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 ступенем 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.

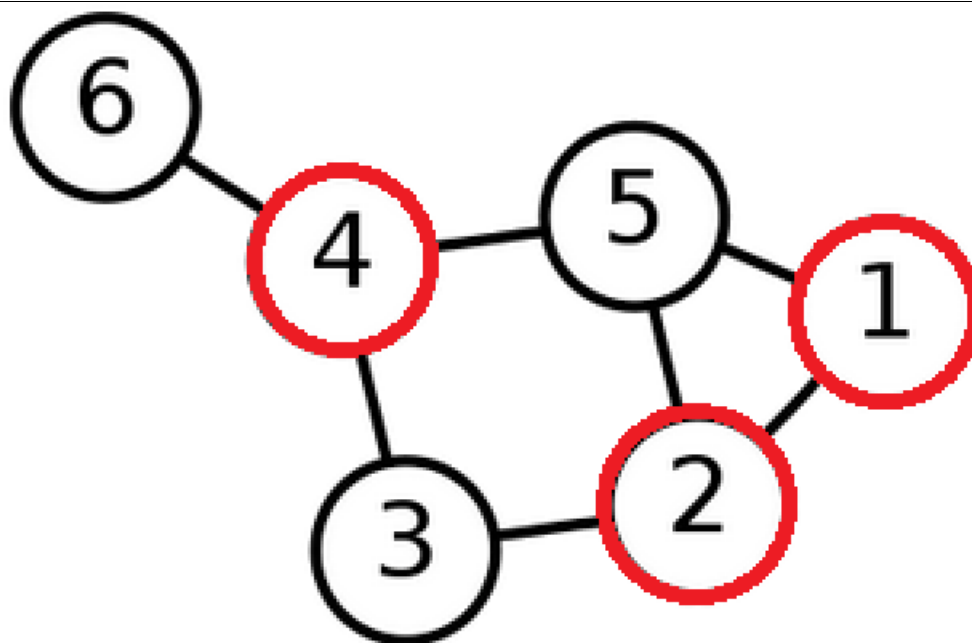
Зробити узагальнений висновок в якому обов'язково описати залежність якості розв'язку від вхідних параметрів.

Таблиця 2.1 – Прикладні задачі

№	Задача
1	Задача про рюкзак (місткість $P=500$, 100 предметів, цінність предметів від 2 до 30 (випадкова), вага від 1 до 20 (випадкова)). Для заданої множини предметів, кожен з яких має вагу і цінність, визначити яку кількість кожного з предметів слід взяти, так, щоб сумарна вага не

	<p>перевищувала задану, а сумарна цінність була максимальною.</p> <p>Задача часто виникає при розподілі ресурсів, коли наявні фінансові обмеження, і вивчається в таких областях, як комбінаторика, інформатика, теорія складності, криптографія, прикладна математика.</p>
2	<p>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.</p> <p>Розглядається симетричний, асиметричний та змішаний варіанти.</p> <p>В загальному випадку, асиметрична задача комівояжера відрізняється тим, що ребра між вершинами можуть мати різну вагу в залежності від напрямку, тобто, задача моделюється орієнтованим графом. Таким чином, окрім ваги ребер графа, слід також зважати і на те, в якому напрямку знаходяться ребра.</p> <p>У випадку симетричної задачі всі пари ребер між одними й тими самими вершинами мають однакову вагу.</p> <p>У випадку реальних міст може бути як симетричною, так і асиметричною в залежності від тривалості або довжини маршрутів і напрямку руху.</p> <p>Застосування:</p> <ul style="list-style-type: none"> – доставка товарів (в цьому випадку може бути більш доречна постановка транспортної задачі - доставка в кілька магазинів з декількох складів); – доставка води; – моніторинг об'єктів;

	<ul style="list-style-type: none"> – поповнення банкоматів готівкою; – збір співробітників для доставки вахтовим методом.
3	<p>Розфарбовування графа (300 вершин, степінь вершини не більше 30, але не менше 2) – називають таке приписування кольорів (або натуральних чисел) його вершинам, що ніякі дві суміжні вершини не набувають однакового кольору. Найменшу можливу кількість кольорів у розфарбуванні називають хроматичне число.</p> <p>Застосування:</p> <ul style="list-style-type: none"> – розкладу для освітніх установ; – розкладу в спорті; – планування зустрічей, зборів, інтерв'ю; – розклади транспорту, в тому числі - авіатранспорту; – розкладу для комунальних служб;
4	<p>Задача вершинного покриття (300 вершин, степінь вершини не більше 30, але не менше 2). Вершинне покриття для неорієнтованого графа $G = (V, E)$ - це множина його вершин S, така, що, у кожного ребра графа хоча б один з кінців входить в вершину з S.</p> <p>Задача вершинного покриття полягає в пошуку вершинного покриття найменшого розміру для заданого графа (цей розмір називається числом вершинного покриття графа).</p> <p>На вході: Граф $G = (V, E)$.</p> <p>Результат: множина $C \subseteq V$ - найменше вершинне покриття графа G.</p>



Застосування:

- розміщення пунктів обслуговування;
- призначення екіпажів на транспорт;
- проектування інтегральних схем і конвеєрних ліній.

5	<p>Задача про кліку (300 вершин, степінь вершини не більше 30, але не менше 2). Клікою в неорієнтованому графі називається підмножина вершин, кожні дві з яких з'єднані ребром графа. Іншими словами, це повний підграф первісного графа. Розмір кліки визначається як число вершин в ній.</p> <p>Задача про кліку існує у двох варіантах: у задачі розпізнавання потрібно визначити, чи існує в заданому графі G кліка розміру k, тоді як в обчислювальному варіанті потрібно знайти в заданому графі G кліку максимального розміру або всі максимальні кліки (такі, що не можна збільшити).</p> <p>Застосування:</p> <ul style="list-style-type: none"> – біоінформатика; – електротехніка;
6	<p>Задача про найкоротший шлях (300 вершин, відстань між вершинами випадкова від 5 до 150, степінь вершини не більше 10, але не менше 1) -</p>

	<p>задача пошуку найкоротшого шляху (ланцюга) між двома точками (вершинами) на графі, в якій мінімізується сума ваг ребер, що складають шлях.</p> <p>Важливість задачі визначається її різними практичними застосуваннями. Наприклад, в GPS-навігаторах здійснюється пошук найкоротшого шляху між точкою відправлення і точкою призначення. Як вершин виступають перехрестя, а дороги є ребрами, які лежать між ними. Якщо сума довжин доріг між перехрестями мінімальна, тоді знайдений шлях найкоротший.</p>
--	--

Таблиця 2.2 – Варіанти алгоритмів і досліджувані параметри

№	Алгоритми і досліджувані параметри
1	<p>Генетичний алгоритм:</p> <ul style="list-style-type: none"> - оператор схрещування (мінімум 3); - мутація (мінімум 2); - оператор локального покращення (мінімум 2).
2	<p>Мурашиний алгоритм:</p> <ul style="list-style-type: none"> - α; - β; - ρ; - L_{min}; - кількість мурах M і їх типи (елітні, тощо...); - маршрути з однієї чи різних вершин.
3	<p>Бджолиний алгоритм:</p> <ul style="list-style-type: none"> - кількість ділянок; - кількість бджіл (фуражирів і розвідників).

Таблиця 2.3 – Варіанти задач і алгоритмів

№	Задачі і алгоритми
1	Задача про рюкзак + Генетичний алгоритм
2	Задача про рюкзак + Бджолиний алгоритм
3	Задача комівояжера (асиметрична мережа) + Генетичний алгоритм
4	Задача комівояжера (симетрична мережа) + Генетичний алгоритм
5	Задача комівояжера (змішана мережа) + Генетичний алгоритм
6	Задача комівояжера (асиметрична мережа) + Мурашиний алгоритм
7	Задача комівояжера (симетрична мережа) + Мурашиний алгоритм
8	Задача комівояжера (змішана мережа) + Мурашиний алгоритм
9	Задача вершинного покриття + Генетичний алгоритм
10	Задача вершинного покриття + Бджолиний алгоритм
11	Задача комівояжера (асиметрична мережа) + Бджолиний алгоритм
12	Задача комівояжера (симетрична мережа) + Бджолиний алгоритм
13	Задача комівояжера (змішана мережа) + Бджолиний алгоритм
14	Розфарбовування графа + Генетичний алгоритм
15	Розфарбовування графа + Бджолиний алгоритм
16	Задача про кліку (задача розпізнавання) + Генетичний алгоритм
17	Задача про кліку (задача розпізнавання) + Бджолиний алгоритм
18	Задача про кліку (обчислювальна задача) + Генетичний алгоритм
19	Задача про кліку (обчислювальна задача) + Бджолиний алгоритм
20	Задача про найкоротший шлях + Генетичний алгоритм
21	Задача про найкоротший шлях + Мурашиний алгоритм
22	Задача про найкоротший шлях + Бджолиний алгоритм
23	Задача про рюкзак + Генетичний алгоритм
24	Задача про рюкзак + Бджолиний алгоритм
25	Задача комівояжера (асиметрична мережа) + Генетичний алгоритм
26	Задача комівояжера (симетрична мережа) + Генетичний алгоритм
27	Задача комівояжера (змішана мережа) + Генетичний алгоритм

28	Задача комівояжера (асиметрична мережа) + Мурашиний алгоритм
29	Задача комівояжера (симетрична мережа) + Мурашиний алгоритм
30	Задача комівояжера (змішана мережа) + Мурашиний алгоритм

3 ВИКОНАННЯ

3.1 Покроковий алгоритм

int amountOfIterations=100;

int mutationChance = 75;

Функція Chromosome* GASolver::solve():

Для і від 0 до amountOfIterations:

Chromosome* randomChromosomeForCrossover =
getRandomChromosomeForCrossover()

Chromosome* childChromosome = crossover(currentBestChromosome,
randomChromosomeForCrossover)

int mutationValue = випадкове число від 1 до 100

Якщо mutationValue <= mutationChance **то**

mutate(childChromosome)

Додати childChromosome до population

Якщо довжина childChromosome <= довжина currentBestChromosome
то

currentBestChromosome = childChromosome

Видалити TheWorstChromosome()

Повернути currentBestChromosome

Функція Chromosome* GASolver::crossover(Chromosome* parent1,
Chromosome* parent2):

Chromosome* child = створити новий об'єкт Chromosome

Список parent1Vertexes = отримати вершини parent1

Список parent2Vertexes = отримати вершини parent2

Список positionsOfIntersection = знайти точки перетину parent1 і parent2

Якщо positionsOfIntersection порожній **то**:

Для кожної вершини у parent2Vertexes:

Додати вершину до child

Повернути child

int pos1 = positionsOfIntersection[0]

int pos2 = positionsOfIntersection[1]

Для і від 0 до pos1:

Додати parent1Vertexes[i] до child

Для і від pos2 до parent2Vertexes.size():

Додати parent2Vertexes[i] до child

Повернути child

Функція Chromosome* GASolver::getRandomChromosomeForCrossover():

Повернути population[випадкове число від 0 до population.size() - 1]

Процедура GASolver::mutate(Chromosome* chromosome):

Константа maxMutationLength = 3

Список chromosomeVertexes = отримати вершини chromosome

int beginIndex = випадкове число від 0 до chromosomeVertexes.size() - maxMutationLength - 1

Якщо beginIndex < 0 **то** beginIndex = 0

```
int endIndex = beginIndex + випадкове число від 1 до maxMutationLength
Якщо endIndex > chromosomeVertexes.size() - 1 то endIndex =
chromosomeVertexes.size() - 1
```

```
Chromosome* subChromosome = створити новий об'єкт Chromosome
Chromosome* deadEndVertexes = створити новий об'єкт Chromosome
```

```
getRandomPath(chromosomeVertexes[beginIndex],
chromosomeVertexes[endIndex], subChromosome, deadEndVertexes)
chromosome->replace(beginIndex, endIndex, subChromosome)
```

```
Видалити deadEndVertexes
Видалити subChromosome
```

Процедура GASolver::deleteTheWorstChromosome():

```
int index = 0
Для i від 1 до population.size():
    Якщо population[i]->getLength() > population[index]->getLength(), то
index = i
```

```
Видалити population[index]
```

3.2 Програмна реалізація алгоритму

3.2.1 Вихідний код

chromosome.h

```
#pragma once
```

```
#include "vertex_edge.h"
```

```
#include <iostream>
```

```

using namespace std;

class Chromosome {
    vector<Vertex*> vertexes;

public:
    void addVertex(Vertex* vertex);

    bool contains(Vertex* vertex);
    void deleteLastVertex();

    int getLength() const;
    vector<Vertex*> getVertexes();
    vector<int> findPositionsOfIntersection(Chromosome* obj) const;

    void replace(int begin, int end, Chromosome* newPath);

    void display() const;

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

```

GASolver.h

```

#pragma once

#include "graph.h"
#include "chromosome.h"
#include <algorithm>

```

```

#define AMOUNT_OF_ITERATIONS 120
#define POPULATION_SIZE 130
#define MUTATION_CHANCE 75

class GASolver {
    int populationSize;
    int mutationChance;
    int amountOfIterations;

    Graph* graphToSolve;

    Vertex *source, *destination;

    vector<Chromosome*> population;
    Chromosome* currentBestChromosome;

    bool getRandomPath(Vertex* startVertex, Vertex* endVertex,
Chromosome *&path, Chromosome *&deadEndVertexes);
    vector<Vertex*> getPossibleNextVertexes(Vertex* currentVertex,
Vertex* endVertex, Chromosome* visitedVertexes, Chromosome*
deadEndVertexes);

    void createInitialPopulation();
    void findCurrentBestChromosome();

    Chromosome* crossover(Chromosome* parent1, Chromosome*
parent2);
    Chromosome* getRandomChromosomeForCrossover();

```

```

        void mutate(Chromosome* chromosome);
        void deleteTheWorstChromosome();
public:
        GASolver(Graph* graph, Vertex* source, Vertex* destination);

        Chromosome* solve();

        ~GASolver();
};

```

graph.h

```

#pragma once

#include "vertex_edge.h"
#include "random_generators.h"
#include <iostream>
#include <iomanip>

#define MIN_DEGREE 1
#define MAX_DEGREE 10
#define MIN_LENGTH 5
#define MAX_LENGTH 150

class Graph {
    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:
    Graph(int amountOfVertexes);

    int getAmountOfVertexes() const;
    vector<Vertex*> getVertexes();
    Vertex* getVertexWithNumber(int number);

    vector<Edge*> getEdges();

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

    ~Graph();
};

```

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);
```

vertex_edge.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();
```

```
    Edge* getConnectingEdge(Vertex* connectedVertex);
```

```
    int getNumber();
```

```

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

class Edge {
    vector<Vertex*> connectedVertexes;
    int length;
public:
    Edge(Vertex* vertex1, Vertex* vertex2, int length);

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

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

```

chromosome.cpp

```

#include "chromosome.h"

void Chromosome::addVertex(Vertex* vertex) {
    vertexes.push_back(vertex);
}

int Chromosome::getLength() const {
    int length = 0;

    for (int i = 0; i < vertexes.size() - 1; ++i) {
        auto edge = vertexes.at(i)->getConnectingEdge(vertexes.at(i + 1));
        length += edge->getLength();
    }
}

```

```

    }

    return length;
}

vector<Vertex*> Chromosome::getVertexes() {
    return vertexes;
}

bool Chromosome::contains(Vertex* vertex) {
    for (auto vert : vertexes) {
        if (vert == vertex)
            return true;
    }

    return false;
}

vector<int> Chromosome::findPositionsOfIntersection(Chromosome* obj)
const {
    vector<int> positionsOfIntersection;

    for (int i = 1; i < vertexes.size() - 1; ++i) {
        for (int j = 1; j < obj->vertexes.size() - 1; ++j) {
            if (vertexes.at(i) == obj->vertexes.at(j)) {
                positionsOfIntersection.push_back(i);
                positionsOfIntersection.push_back(j);
                return positionsOfIntersection;
            }
        }
    }
}

```

```

    }

    return positionsOfIntersection;
}

void Chromosome::replace(int begin, int end, Chromosome* newPath) {
    vertexes.erase(vertexes.begin() + begin, vertexes.begin() + end + 1);
    vertexes.insert(vertexes.begin() + begin, newPath->vertexes.begin(),
newPath->vertexes.end());
}

void Chromosome::deleteLastVertex() {
    vertexes.pop_back();
}

void Chromosome::display() const {
    for (int i = 0; i < vertexes.size()-1; ++i)
        cout << vertexes.at(i)->getNumber() << " -> ";
    cout << vertexes.back()->getNumber() << endl;
}

bool Chromosome::operator==(const Chromosome& obj) {
    if (vertexes.size() != obj.vertexes.size())
        return false;

    for (int i = 0; i < vertexes.size(); ++i) {
        if (vertexes.at(i) != obj.vertexes.at(i))
            return false;
    }
}

```

```
        return true;
    }
```

GASolver.cpp

```
#include "GASolver.h"
```

```
GASolver::GASolver(Graph* graph, Vertex* source, Vertex* destination) {
    this->graphToSolve = graph;
    this->source = source;
    this->destination = destination;
    this->populationSize = POPULATION_SIZE;
    this->mutationChance = MUTATION_CHANCE;
    this->amountOfIterations = AMOUNT_OF_ITERATIONS;

    createInitialPopulation();
    findCurrentBestChromosome();
}
```

```
void GASolver::createInitialPopulation() {
    Chromosome* deadEndVertexes = new Chromosome;
    for (int i = 0; i < populationSize; ++i) {
        Chromosome* path = new Chromosome;
        Vertex* startVertex = source;
        Vertex* endVertex = destination;
        getRandomPath(source, destination, path, deadEndVertexes);
        population.push_back(path);
    }
    delete deadEndVertexes;
}
```

```

void GASolver::findCurrentBestChromosome() {
    currentBestChromosome = population.at(0);

    for (int i = 1; i < population.size(); ++i) {
        if (population.at(i)->getLength() < currentBestChromosome-
>getLength())
            currentBestChromosome = population.at(i);
    }
}

bool GASolver::getRandomPath(Vertex* startVertex, Vertex* endVertex,
Chromosome*& path, Chromosome*& deadEndVertexes) {
    if (startVertex == endVertex) {
        path->addVertex(startVertex);
        return true;
    }

    path->addVertex(startVertex);

    vector<Vertex*> possibleNextVertexes =
getPossibleNextVertexes(startVertex, endVertex, path, deadEndVertexes);

    if (possibleNextVertexes.empty()) {
        deadEndVertexes->addVertex(startVertex);
        path->deleteLastVertex();
        return false;
    }

    random_shuffle(possibleNextVertexes.begin(),
possibleNextVertexes.end());

```

```

    for (auto vertex : possibleNextVertexes) {
        if (getRandomPath(vertex, endVertex, path, deadEndVertexes))
            return true;
    }

    path->deleteLastVertex();
    deadEndVertexes->addVertex(startVertex);
    return false;
}

vector<Vertex*> GASolver::getPossibleNextVertexes(Vertex* currentVertex,
Vertex* endVertex, Chromosome* visitedVertexes, Chromosome*
deadEndVertexes) {
    vector<Vertex*> possibleNextVertexes;

    for (auto edge : currentVertex->getIncidentEdges()) {
        Vertex* nextVertex;
        vector<Vertex*> connectedVertexes = edge-
>getConnectedVertexes();

        if (currentVertex == connectedVertexes.at(0))
            nextVertex = connectedVertexes.at(1);
        else
            nextVertex = connectedVertexes.at(0);

        if (!visitedVertexes->contains(nextVertex) &&
!deadEndVertexes->contains(nextVertex))
            possibleNextVertexes.push_back(nextVertex);
    }
}

```



```

        return possibleNextVertexes;
    }

    Chromosome* GASolver::solve() {
        for (int i = 0; i < amountOfIterations; ++i) {
            Chromosome* randomChromosomeForCrossover =
getRandomChromosomeForCrossover();
            Chromosome* childChromosome =
crossover(currentBestChromosome, randomChromosomeForCrossover);

            int mutationValue = generateNumberInRange(1, 100);
            if(mutationValue <= mutationChance)
                mutate(childChromosome);

            population.push_back(childChromosome);

            if (childChromosome->getLength() <= currentBestChromosome-
>getLength())
                currentBestChromosome = childChromosome;

            deleteTheWorstChromosome();
        }

        return currentBestChromosome;
    }

    Chromosome* GASolver::crossover(Chromosome* parent1, Chromosome*
parent2) {
        Chromosome* child = new Chromosome;

```

```

vector<Vertex*> parent1Vertexes = parent1->getVertexes();
vector<Vertex*> parent2Vertexes = parent2->getVertexes();

vector<int> positionsOfIntersection = parent1-
>findPositionsOfIntersection(parent2);

if (positionsOfIntersection.empty()) {
    for (int i = 0; i < parent2Vertexes.size(); ++i)
        child->addVertex(parent2Vertexes.at(i));
    return child;
}

int pos1 = positionsOfIntersection.at(0);
int pos2 = positionsOfIntersection.at(1);

for (int i = 0; i < pos1; ++i)
    child->addVertex(parent1Vertexes.at(i));
for (int i = pos2; i < parent2Vertexes.size(); ++i)
    child->addVertex(parent2Vertexes.at(i));

return child;
}

Chromosome* GASolver::getRandomChromosomeForCrossover() {
    return population.at(generateNumberInRange(0, population.size() - 1));
}

void GASolver::mutate(Chromosome* chromosome) {
    const int maxMutationLength = 3;

    vector<Vertex*> chromosomeVertexes = chromosome->getVertexes();

```

```

        int        beginIndex        =        generateNumberInRange(0,
chromosomeVertexes.size() - maxMutationLength - 1);
        if (beginIndex < 0)
            beginIndex = 0;

        int    endIndex    =    beginIndex    +    generateNumberInRange(1,
maxMutationLength);
        if (endIndex > chromosomeVertexes.size() - 1)
            endIndex = chromosomeVertexes.size() - 1;

        Chromosome* subChromosome = new Chromosome;
        Chromosome* deadEndVertexes = new Chromosome;

        getRandomPath(chromosomeVertexes.at(beginIndex),
chromosomeVertexes.at(endIndex), subChromosome, deadEndVertexes);
        chromosome->replace(beginIndex, endIndex, subChromosome);

        delete deadEndVertexes;
        delete subChromosome;
    }

    void GASolver::deleteTheWorstChromosome() {
        int index = 0;
        for (int i = 1; i < population.size(); ++i) {
            if    (population.at(i)->getLength()    >    population.at(index)-
>getLength())
                index = i;
        }
    }

```

```

        delete population.at(index);
        population.erase(population.begin() + index);
    }

```

```

GASolver::~~GASolver() {
    for (auto chromosome : population)
        delete chromosome;
}

```

graph.cpp

```

#include "graph.h"

```

```

Graph::Graph(int amountOfVertexes) {
    this->amountOfVertexes = amountOfVertexes;
    generateVertexes();
    generateEdges();
}

```

```

void Graph::generateVertexes() {
    for (int i = 0; i < amountOfVertexes; ++i) {
        Vertex* vertex = new Vertex(i + 1);
        vertexes.push_back(vertex);
    }
}

```

```

void Graph::generateEdges() {
    for (int i = 0; i < amountOfVertexes; ++i) {
        int    vertexDegree    =    generateNumberInRange(MIN_DEGREE,
MAX_DEGREE/2);

```

```

        for (int j = i + 1; j < i + 1 + vertexDegree && j < amountOfVertexes; ++j)
        {
            int length = generateNumberInRange(MIN_LENGTH,
MAX_LENGTH);

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

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

```

```

int Graph::getAmountOfVertexes() const {
    return amountOfVertexes;
}

```

```

vector<Vertex*> Graph::getVertexes() {
    return vertexes;
}

```

```

vector<Edge*> Graph::getEdges() {
    return edges;
}

```

```

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

```

```

    }

    return nullptr;
}

Vertex* Graph::getVertexWithNumber(int number) {
    for (auto vertex : vertexes) {
        if (vertex->getNumber() == number)
            return vertex;
    }

    return nullptr;
}

void Graph::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 Graph::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 Graph::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 Graph::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));

```

```

        if (!edge)
            cout << setw(width) << "-" << " ";
        else
            cout << setw(width) << edge->getLength() << " ";
    }
}

```

```

Graph::~~Graph() {
    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 "graph.h"
#include "random_generators.h"
#include "GASolver.h"
#include "input_validators.h"

using namespace std;

int main() {
    srand(time(nullptr));
    const int amountOfVertexes = 300;

    Graph graph(amountOfVertexes);
    graph.display();
}

```

```

vector<Vertex*> vertexes = graph.getVertexes();
int run;
do {
    int start = generateNumberInRange(0, amountOfVertexes / 2);
    int finish = generateNumberInRange(start + 1, amountOfVertexes
- 1);

    GASolver solver(&graph, vertexes.at(start), vertexes.at(finish));
    Chromosome* solution = solver.solve();
    cout << "The shortest path between vertexes " << start+1 << " and
" << finish+1 << " is:" << endl;
    solution->display();
    cout << "Length: " << solution->getLength() << endl;

    cout << "Enter 1 to run the program again or 0 to exit:" << endl;
    run = inputPositiveNumberInRange(0, 1);
} while (run);
return 0;
}

```

random_generators.cpp

```

#include "Random_generators.h"

int generateNumberInRange(int lowerBound, int upperBound) {
    if (upperBound < lowerBound) {
        int temp = lowerBound;
        lowerBound = upperBound;
        upperBound = temp;
    }
}

```

```

    return lowerBound + rand() % (upperBound - lowerBound + 1);
}

```

vertex_edge.cpp

```

#include "vertex_edge.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);
}

```

```

Edge* Vertex::getConnectingEdge(Vertex* connectedVertex) {
    for (auto edge : incidentEdges) {
        if (edge->contains(connectedVertex))
            return edge;
    }

    return nullptr;
}

```

```

vector<Edge*> Vertex::getIncidentEdges() {
    return incidentEdges;
}

```

```
}
```

```
int Vertex::getNumber() {  
    return number;  
}
```

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

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

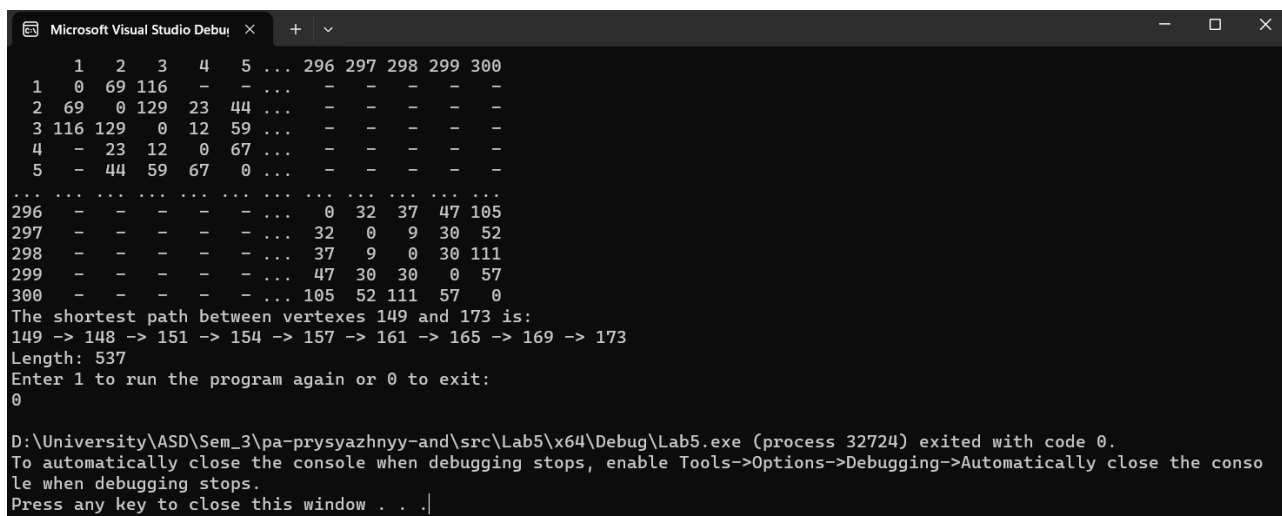
```
vector<Vertex*> Edge::getConnectedVertexes() {  
    return connectedVertexes;  
}
```

```
bool Edge::operator==(const Edge& obj) {  
    return (this->connectedVertexes.at(0) == obj.connectedVertexes.at(0) &&  
            this->connectedVertexes.at(1) == obj.connectedVertexes.at(1)) ||  
            (this->connectedVertexes.at(0) == obj.connectedVertexes.at(1) &&  
            this->connectedVertexes.at(1) == obj.connectedVertexes.at(0));  
}
```

```
bool Edge::contains(Vertex* vertex) const {  
    return (*connectedVertexes.at(0) == *vertex) ||  
            (*connectedVertexes.at(1) == *vertex);  
}
```

3.2.2 Приклади роботи

На рисунку 3.1 показано приклад роботи програми.



```
Microsoft Visual Studio Debug Console
1 2 3 4 5 ... 296 297 298 299 300
1 0 69 116 - - ... - - - - -
2 69 0 129 23 44 ... - - - - -
3 116 129 0 12 59 ... - - - - -
4 - 23 12 0 67 ... - - - - -
5 - 44 59 67 0 ... - - - - -
... ..
296 - - - - - ... 0 32 37 47 105
297 - - - - - ... 32 0 9 30 52
298 - - - - - ... 37 9 0 30 111
299 - - - - - ... 47 30 30 0 57
300 - - - - - ... 105 52 111 57 0
The shortest path between vertexes 149 and 173 is:
149 -> 148 -> 151 -> 154 -> 157 -> 161 -> 165 -> 169 -> 173
Length: 537
Enter 1 to run the program again or 0 to exit:
0

D:\University\ASD\Sem_3\pa-prysyazhnyy-and\src\Lab5\x64\Debug\Lab5.exe (process 32724) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .|
```

Рисунок 3.1 – Приклад роботи програми

Тестування алгоритму

Таблиця 1 – залежність якості розв’язку від ймовірності мутацій, розмір популяції – 100, кількість ітерацій – 200

Ймовірність мутації (%)	Довжина знайденого шляху
0	9853
5	10010
10	9704
15	10096
20	9862
25	9728
30	10247
35	10051
40	9808
45	10147
50	10054
55	9646
60	10293
65	9673
70	9484
75	9343
80	9771
85	9696
90	10029
95	9348
100	9549

Отже, найкраща ймовірність мутації – 75%.

Таблиця 2 – залежність якості розв’язку від розміру популяції, ймовірність мутації – 75%, кількість ітерацій – 200.

Розмір популяції	Довжина знайденого шляху
10	10300
20	10496
30	9655
40	9558
50	9745
60	9205
70	9768
80	9666
90	9902
100	9775
110	9780
120	9309
130	8947
140	9796
150	10160
160	9826
170	9641
180	9369
190	9731
200	9307

Отже, найкращий розмір популяції – 130.

Таблиця 3 – залежність якості розв’язку від кількості ітерацій, ймовірність мутації – 75%, розмір популяції – 130.

Кількість ітерацій	Довжина знайденого шляху
10	10311
20	9859
30	9227
40	9977
50	9802
60	9465
70	9923
80	9651
90	9805
100	9791
110	9778
120	9252
130	9647
140	9657
150	9301
160	9310
170	9919
180	9734
190	9741
200	9693

Отже, найкраща кількість ітерацій – 120.

ВИСНОВОК

В рамках даної лабораторної роботи я реалізував генетичний алгоритм для задачі про пошук найкоротшого шляху між двома вершинами в графі за допомогою засобів мови програмування C++. Я провів дослідження вхідних параметрів алгоритму, а саме: ймовірність мутації, кількість ітерацій, розмір популяції. При виконанні роботи я переконався, що в середньому найбільш ефективними вхідними параметрами є: ймовірність мутації – 75%, розмір початкової популяції – 130, кількість ітерацій – 120.

КРИТЕРІЇ ОЦІНЮВАННЯ

При здачі лабораторної роботи до 24.12.2023 включно максимальний бал дорівнює – 5. Після 24.12.2023 максимальний бал дорівнює – 4,5.

Критерії оцінювання у відсотках від максимального балу:

- покроковий алгоритм – 10%;
- програмна реалізація алгоритму – 45%;
- робота з гіт – 20%;
- тестування алгоритму – 20%;
- висновок – 5%.

+1 додатковий бал можна отримати за виконання та захист роботи до 17.12.2023