**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)**

(шифр, прізвище, ім'я, по батькові)

*Піонтківський Віталій ІП-12*

**Перевірив**

(прізвище, ім'я, по батькові)

*Головченко М.Н.*

Київ 2022

Зміст

[1 Мета лабораторної роботи 3](#_Toc52291748)

[2 Завдання 4](#_Toc52291749)

[3 Виконання 10](#_Toc52291750)

[3.1 Покроковий алгоритм 10](#_Toc52291751)

[3.2 Програмна реалізація алгоритму 10](#_Toc52291752)

[3.2.1 Вихідний код 10](#_Toc52291753)

[3.2.2 Приклади роботи 10](#_Toc52291754)

[3.3 Тестування алгоритму 11](#_Toc52291755)

[Висновок 12](#_Toc52291756)

[Критерії оцінювання 13](#_Toc52291757)

# Мета лабораторної роботи

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

# 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** |
| 6 | **Задача про найкоротший шлях** (300 вершин, відстань між вершинами випадкова від 5 до 150, степінь вершини не більше 10, але не менше 1) - задача пошуку найкоротшого шляху (ланцюга) між двома точками (вершинами) на графі, в якій мінімізується сума ваг ребер, що складають шлях.  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. |

Таблиця 2.2 – Варіанти алгоритмів і досліджувані параметри

|  |  |
| --- | --- |
| **№** | **Алгоритми і досліджувані параметри** |
| 1 | **Генетичний алгоритм:**   * оператор схрещування (мінімум 3); * мутація (мінімум 2); * оператор локального покращення (мінімум 2). |

Таблиця 2.3 – Варіанти задач і алгоритмів

|  |  |
| --- | --- |
| **№** | **Задачі і алгоритми** |
| 20 | Задача про найкоротший шлях + Генетичний алгоритм |

# Виконання

## Покроковий алгоритм

//створення перших особин

Child[] children = doFirstGeneration();

Child record = findMinDistance(children);

Random rand = new Random();

//основний цикл

for (int i = 0; i < GENERATIONS; i++) {

//вибір батьків

Child firstParent = record;

Child secondParent = children[rand.nextInt(children.length)];

while (secondParent == firstParent) {

secondParent = children[rand.nextInt(children.length)];

}

//створення нащадка (вибраної варіації)

Child child = new Child(new ArrayList<>());

if (crossingVariation == 1){

child = crossingV1(firstParent, secondParent);

} else if (crossingVariation == 2){

child = crossingV2(firstParent, secondParent);

} else if (crossingVariation == 3){

child = crossingV3(firstParent, secondParent);

}

//мутація (вибраної варіації)

Child copy = child;

if (mutationVersion == 1) {

child = mutationV1(child);

} else if (mutationVersion == 2) {

child = mutationV2(child);

}

//локальне покращення

copy = child;

if (rand.nextDouble() > 0.4) {

if (improvementVariation == 1) {

child = localImprovementV1(child);

} else if (improvementVariation == 2) {

child = localImprovementV2(child);

}

}

//видалення особини з найдовшим шляхом

children[findMaxIndex(children)] = child;

//change record

record = findMinDistance(children);

}

//повертаю найкращий шлях

return findMinDistance(children).countDistance(distancesMatrix);

}

## Програмна реалізація алгоритму

### Вихідний код

public class Main {  
 public static void main(String[] args) {  
  
 Graph world = new Graph(300);  
 world.printDistancesMatrix();  
 System.*out*.println("Is in each city no more than "+ Graph.*MAX\_ROADS\_IN\_CITY* + " roads and at least 1: " + world.canExist());  
  
 GeneticAlgorithm solve= new GeneticAlgorithm(world.distancesMatrix, 0, 299);  
  
 String summary = "";  
 for (int j = 1; j < 4; j++) {  
 int iterations =1;  
 int averageDistanceFirst = 0;  
 for (int i = 0; i < iterations; i++) {  
 averageDistanceFirst += solve.solve(j,1, 1);  
 }  
 averageDistanceFirst /=iterations;  
  
 int averageDistanceSecond = 0;  
 for (int i = 0; i < iterations; i++) {  
 averageDistanceSecond += solve.solve(j,1, 2);  
 }  
 averageDistanceSecond /=iterations;  
  
 int averageDistanceThird = 0;  
 for (int i = 0; i < iterations; i++) {  
 averageDistanceThird += solve.solve(j,2, 1);  
 }  
 averageDistanceThird /=iterations;  
  
 int averageDistanceFourth = 0;  
 for (int i = 0; i < iterations; i++) {  
 averageDistanceFourth += solve.solve(j,2, 2);  
 }  
 averageDistanceFourth /=iterations;  
  
 System.*out*.println("Average distance in first variation("+j+",1,1): "+ averageDistanceFirst);  
 summary+="Average distance in first variation("+j+",1,1): "+ averageDistanceFirst+"\n";  
 System.*out*.println("Average distance in second variation("+j+",1,2): "+ averageDistanceSecond);  
 summary+="Average distance in second variation("+j+",1,2): "+ averageDistanceSecond+"\n";  
 System.*out*.println("Average distance in third variation("+j+",2,1): "+ averageDistanceThird);  
 summary+="Average distance in third variation("+j+",2,1): "+ averageDistanceThird+"\n";  
 System.*out*.println("Average distance in fourth variation("+j+",2,2): "+ averageDistanceFourth);  
 summary+="Average distance in fourth variation("+j+",2,2): "+ averageDistanceFourth+"\n";  
 }  
 System.*out*.println();  
 System.*out*.println(summary);  
  
  
  
  
  
 }  
}

import java.util.ArrayList;  
import java.util.Random;  
  
public class GeneticAlgorithm {  
 static final int *NUMBER\_OF\_CHILDREN* = 100;  
 static final int *GENERATIONS* = 10000;  
  
 int[][] distancesMatrix;  
 int firstGen;  
 int lastGen;  
  
 int mutations;  
 int crossings;  
 int improvements;  
  
 public GeneticAlgorithm(int[][] distancesMatrix, int firstGen, int lastGen) {  
 this.distancesMatrix = distancesMatrix;  
 this.firstGen = firstGen;  
 this.lastGen = lastGen;  
 mutations = 0;  
 }  
  
 public Child[] doChildren() {  
 Child[] children = new Child[*NUMBER\_OF\_CHILDREN*];  
 for (int i = 0; i < *NUMBER\_OF\_CHILDREN*; i++) {  
 children[i] = doChild();  
 }  
 return children;  
 }  
  
 private Child doChild() {  
 ArrayList<Integer> firstGenes = new ArrayList<>();  
 firstGenes.add(firstGen);  
 ArrayList<Integer> secondGenes = new ArrayList<>();  
 secondGenes.add(lastGen);  
 Random rand = new Random();  
 //do two lists with random genes,  
 //first starts with first gen, second starts with second gen  
 for (int j = 0; j < distancesMatrix.length \* 0.7; j++) {  
 int nextGenForFirst = rand.nextInt(distancesMatrix.length);  
 int nextGenForSecond = rand.nextInt(distancesMatrix.length);  
 while (distancesMatrix[firstGenes.get(firstGenes.size() - 1)][nextGenForFirst] == 0) {  
 nextGenForFirst = rand.nextInt(distancesMatrix.length);  
 }  
 firstGenes.add(nextGenForFirst);  
 while (distancesMatrix[secondGenes.get(secondGenes.size() - 1)][nextGenForSecond] == 0) {  
 nextGenForSecond = rand.nextInt(distancesMatrix.length);  
 }  
 secondGenes.add(nextGenForSecond);  
 }  
 //stick up first and second lists  
 ArrayList<Integer> child = new ArrayList<>();  
 boolean isAdded = false;  
 for (int j = 0; j < firstGenes.size(); j++) {  
 for (int k = 0; k < secondGenes.size(); k++) {  
 if (distancesMatrix[firstGenes.get(firstGenes.size() - 1 - j)][secondGenes.get(secondGenes.size() - 1 - k)] != 0 && !isAdded) {  
 for (int l = 0; l < firstGenes.size() - j; l++) {  
 child.add(firstGenes.get(l));  
 }  
 for (int l = 0; l < secondGenes.size() - k; l++) {  
 child.add(secondGenes.get(secondGenes.size() - k - l - 1));  
 }  
 isAdded = true;  
 }  
 }  
 }  
 if (child.isEmpty()) return doChild();  
 else return new Child(child);  
  
 }  
  
 public int solve(int crossingVariation, int mutationVersion, int improvementVariation) {  
 mutations = 0;  
 improvements = 0;  
 crossings = 0;  
 Child[] children = doChildren();  
 Child record = findMinDistance(children);  
 Random rand = new Random();  
 for (int i = 0; i < *GENERATIONS*; i++) {  
 //choose parents  
 //===============================================================  
 Child firstParent = record;  
 Child secondParent = children[rand.nextInt(children.length)];  
 while (secondParent == firstParent) {  
 secondParent = children[rand.nextInt(children.length)];  
 }  
  
 //do child(three variations)  
 //=================================================================  
 Child child = new Child(new ArrayList<>());  
 if (crossingVariation == 1){  
 child = crossingV1(firstParent, secondParent);  
 } else if (crossingVariation == 2){  
 child = crossingV2(firstParent, secondParent);  
 } else if (crossingVariation == 3){  
 child = crossingV3(firstParent, secondParent);  
 }  
 if (child.getGenes().get(child.getGenes().size() - 1) != lastGen) {  
 System.*out*.println("Crossing error");  
 System.*out*.println(firstParent.getGenes());  
 System.*out*.println(secondParent.getGenes());  
 System.*out*.println(child.getGenes());  
 throw new IllegalArgumentException();  
 }  
 //mutation  
 //===============================================================  
 Child copy = child;  
 if (mutationVersion == 1) {  
 child = mutationV1(child);  
 } else if (mutationVersion == 2) {  
 child = mutationV2(child);  
 }  
 if (child.getGenes().get(child.getGenes().size() - 1) != lastGen || !child.canExist(distancesMatrix)) {  
 System.*out*.println("Mutation error");  
 System.*out*.println(copy.getGenes());  
 throw new IllegalArgumentException();  
 }  
 //local improvement  
 //================================================================  
 copy = child;  
 if (rand.nextDouble() > 0.4) {  
 if (improvementVariation == 1) {  
 child = localImprovementV1(child);  
 } else if (improvementVariation == 2) {  
 child = localImprovementV2(child);  
 }  
  
 }  
 if (child.getGenes().get(child.getGenes().size() - 1) != lastGen) {  
 System.*out*.println("Improvement error");  
 System.*out*.println(copy.getGenes());  
 throw new IllegalArgumentException();  
 }  
 //rewrite the longest path with new child  
 //================================================================  
 children[findMaxIndex(children)] = child;  
 //change record  
 record = findMinDistance(children);  
 }  
// System.out.println(findMinDistance(children).getGenes());  
// System.out.println(findMinDistance(children).countDistance(distancesMatrix));  
// System.out.println("Crossings: " + crossings);  
// System.out.println("Mutations: " + mutations);  
// System.out.println("Improvements: " + improvements);  
 System.*out*.print(".");  
 return findMinDistance(children).countDistance(distancesMatrix);  
 }  
  
 private Child localImprovementV1(Child child) {  
 ArrayList<Integer> genes = new ArrayList<>(child.getGenes());  
 for (int i = 0; i < genes.size() - 1; i++) {  
 for (int j = i + 1; j < genes.size() - 1; j++) {  
 if (genes.get(i).intValue() == genes.get(j).intValue()) {  
 if (j > i) {  
 genes.subList(i, j).clear();  
 if (genes.get(genes.size() - 1) == lastGen && genes.get(0) == firstGen) {  
 improvements++;  
 return new Child(genes);  
 }  
 return new Child(new ArrayList<>(child.getGenes()));  
 }  
 }  
 }  
 }  
 return new Child(new ArrayList<>(child.getGenes()));  
 }  
  
 private Child localImprovementV2(Child child) {  
 ArrayList<Integer> genes = new ArrayList<>(child.getGenes());  
 for (int i = 0; i < genes.size() - 1; i++) {  
 for (int j = i + 2; j < genes.size() - 1; j++) {  
 if (distancesMatrix[genes.get(i)][genes.get(j)] != 0) {  
 if (j > i) {  
 genes.subList(i + 1, j).clear();  
 if (genes.get(genes.size() - 1) == lastGen && genes.get(0) == firstGen) {  
 improvements++;  
 return new Child(genes);  
 }  
 return new Child(new ArrayList<>(child.getGenes()));  
 }  
 }  
 }  
 }  
 return new Child(new ArrayList<>(child.getGenes()));  
 }  
  
 private Child mutationV1(Child child) {  
 if (child.getGenes().size() <= 2) {  
 return child;  
 } else {  
 Random rand = new Random();  
 ArrayList<Integer> genes = new ArrayList<>(child.getGenes());  
 int genForMutation = rand.nextInt(genes.size() - 1);  
 if (genForMutation == 0) {  
 genForMutation++;  
 }  
 int newGen = rand.nextInt(distancesMatrix.length - 2) + 1;  
 genes.set(genForMutation, newGen);  
 Child mutant = new Child(genes);  
 if (mutant.canExist(distancesMatrix)) {  
 mutations++;  
 return mutant;  
 } else return child;  
 }  
 }  
  
 private Child mutationV2(Child child) {  
 if (child.getGenes().size() <= 2) {  
 return child;  
 } else {  
 ArrayList<Integer> genes = new ArrayList<>(child.getGenes());  
 Random rand = new Random();  
 for (int i = rand.nextInt(genes.size() - 2); i < genes.size() - 2; i++) {  
 for (int j = 0; j < distancesMatrix.length; j++) {  
 if (distancesMatrix[j][genes.get(i)] != 0 && distancesMatrix[j][genes.get(i + 2)] != 0) {  
 if (j != genes.get(i + 1)) {  
 genes.set(i + 1, j);  
 mutations++;  
 return new Child(genes);  
 }  
 }  
 }  
 }  
 return new Child(genes);  
 }  
 }  
  
  
 private int findMaxIndex(Child[] children) {  
 int maxIndex = 0;  
 int max = Integer.*MIN\_VALUE*;  
 for (int i = 0; i < children.length; i++) {  
 int currentDistance = children[i].countDistance(distancesMatrix);  
 if (max < currentDistance) {  
 max = currentDistance;  
 maxIndex = i;  
 }  
 }  
 return maxIndex;  
 }  
  
 private Child crossingV1(Child firstParent, Child secondParent) {  
 Random rand = new Random();  
 if (firstParent.getGenes().size() > secondParent.getGenes().size()) {  
 Child curr = firstParent;  
 firstParent = secondParent;  
 secondParent = curr;  
 }  
 int cutLine = rand.nextInt(firstParent.getGenes().size() / 2);  
 ArrayList<Integer> genes = new ArrayList<>();  
 for (int i = 0; i < secondParent.getGenes().size(); i++) {  
 if (i < cutLine) {  
 genes.add(firstParent.getGenes().get(i));  
 } else genes.add(secondParent.getGenes().get(i));  
 }  
  
 Child child = new Child(genes);  
 if (child.canExist(distancesMatrix)) {  
 crossings++;  
 return child;  
 } else return new Child(new ArrayList<>(firstParent.getGenes()));  
 }  
  
  
  
 private Child crossingV2(Child firstParent, Child secondParent) {  
 Random rand = new Random();  
 if (firstParent.getGenes().size() > secondParent.getGenes().size()) {  
 Child curr = firstParent;  
 firstParent = secondParent;  
 secondParent = curr;  
 }  
 int firstCutLine = rand.nextInt(firstParent.getGenes().size() / 2);  
 int secondCutLine = rand.nextInt(firstParent.getGenes().size() -firstParent.getGenes().size() / 2)+firstParent.getGenes().size() / 2;  
 ArrayList<Integer> genes = new ArrayList<>();  
 for (int i = 0; i < firstParent.getGenes().size(); i++) {  
 if (i < firstCutLine || i > secondCutLine-1 ) {  
 genes.add(firstParent.getGenes().get(i));  
 } else genes.add(secondParent.getGenes().get(i));  
 }  
  
 Child child = new Child(genes);  
 if (child.canExist(distancesMatrix)) {  
 crossings++;  
 return child;  
 } else return new Child(new ArrayList<>(firstParent.getGenes()));  
 }  
  
  
 private Child crossingV3(Child firstParent, Child secondParent) {  
 Random rand = new Random();  
 int indexOfCutGenInFirst = rand.nextInt(firstParent.getGenes().size());  
 int cutGenInFirst = firstParent.getGenes().get(indexOfCutGenInFirst);  
 int indexOfCutGenInSecond = secondParent.getGenes().indexOf(cutGenInFirst);  
 if (indexOfCutGenInSecond == -1) return new Child(new ArrayList<>(firstParent.getGenes()));  
 ArrayList<Integer> genes = new ArrayList<>();  
 for (int i = 0; i < indexOfCutGenInFirst; i++) {  
 genes.add(firstParent.getGenes().get(i));  
 }  
 for (int i = indexOfCutGenInSecond; i < secondParent.getGenes().size(); i++) {  
 genes.add(secondParent.getGenes().get(i));  
 }  
 Child child = new Child(genes);  
 if (child.canExist(distancesMatrix)) {  
 crossings++;  
 return child;  
 } else return new Child(new ArrayList<>(firstParent.getGenes()));  
 }  
  
 public Child findMinDistance(Child[] children) {  
 int minIndex = 0;  
 int min = Integer.*MAX\_VALUE*;  
 for (int i = 0; i < children.length; i++) {  
 int currentDistance = children[i].countDistance(distancesMatrix);  
 if (min > currentDistance) {  
 min = currentDistance;  
 minIndex = i;  
 }  
 }  
 return children[minIndex];  
 }  
}

import java.util.ArrayList;  
  
public class Child {  
 private ArrayList<Integer> genes;  
  
 public Child(ArrayList<Integer> genes) {  
 this.genes = genes;  
 }  
  
 public ArrayList<Integer> getGenes() {  
 return genes;  
 }  
  
 public boolean canExist(int[][] matrix){  
 for (int i = 0; i < genes.size()-1; i++) {  
 if (matrix[genes.get(i)][genes.get(i+1)]==0){  
 return false;  
 }  
 }  
 return true;  
 }  
  
 public int countDistance(int[][] matrix){  
 int distance = 0;  
 for (int i = 0; i < genes.size()-1; i++) {  
 distance+=matrix[genes.get(i)][genes.get(i+1)];  
 }  
 return distance;  
 }  
}

import java.util.Arrays;  
import java.util.Random;  
  
public class Graph {  
 int[][] distancesMatrix;  
  
 static final int *MAX\_ROADS\_IN\_CITY* = 10;  
 static final int *MAX\_DISTANCE* = 150;  
 static final int *MIN\_DISTANCE* = 5;  
  
 public Graph(int size) {  
 this.distancesMatrix = new int[size][size];  
 fillMatrix(size);  
  
 }  
  
 private void fillMatrix(int size) {  
  
 //put at least one road to each city  
 putOneToRow();  
 while(!canExist()){  
 this.distancesMatrix = new int[size][size];  
 putOneToRow();  
 }  
  
 Random rand = new Random();  
 for (int i = 0; i < distancesMatrix.length; i++) {  
 for (int j = i + 1; j < distancesMatrix.length; j++) {  
 if (rand.nextDouble() < (double)*MAX\_ROADS\_IN\_CITY*/size && canPut(i, j)) {  
 int distance = rand.nextInt(*MAX\_DISTANCE*-*MIN\_DISTANCE*+1)+*MIN\_DISTANCE*;  
 distancesMatrix[i][j] = distance;  
 distancesMatrix[j][i] = distance;  
 }  
 }  
 }  
 }  
  
 public boolean canExist() {  
 for (int i = 0; i < distancesMatrix.length; i++) {  
 int roads =0;  
 for (int j = 0; j < distancesMatrix.length; j++) {  
 if (distancesMatrix[i][j]!=0){  
 roads++;  
 }  
 }  
 if (roads >*MAX\_ROADS\_IN\_CITY* || roads == 0){  
 return false;  
 }  
 }  
 return true;  
 }  
  
 private void putOneToRow() {  
 Random rand = new Random();  
 for (int i = 0; i < distancesMatrix.length;i++ ) {  
 int column = rand.nextInt(distancesMatrix.length);  
 while(column==i){  
 column = rand.nextInt(distancesMatrix.length);  
 }  
 int distance = rand.nextInt(*MAX\_DISTANCE*-*MIN\_DISTANCE*+1)+*MIN\_DISTANCE*;  
 distancesMatrix[i][column] = distance;  
 distancesMatrix[column][i] = distance;  
 }  
 }  
  
 private boolean canPut(int row, int column) {  
 if (row == column) return false;  
 int roadsInColumn = 0;  
 int roadsInRow = 0;  
 for (int i = 0; i < distancesMatrix.length; i++) {  
 if (distancesMatrix[row][i] != 0) {  
 roadsInRow++;  
 }  
 if (distancesMatrix[i][column] != 0) {  
 roadsInColumn++;  
 }  
 }  
 return roadsInRow < Graph.*MAX\_ROADS\_IN\_CITY* && roadsInColumn < Graph.*MAX\_ROADS\_IN\_CITY*;  
 }  
  
 public void printDistancesMatrix(){  
 for (var i:distancesMatrix) {  
 System.*out*.println(Arrays.*toString*(i));  
 }  
 }  
}

### Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми.

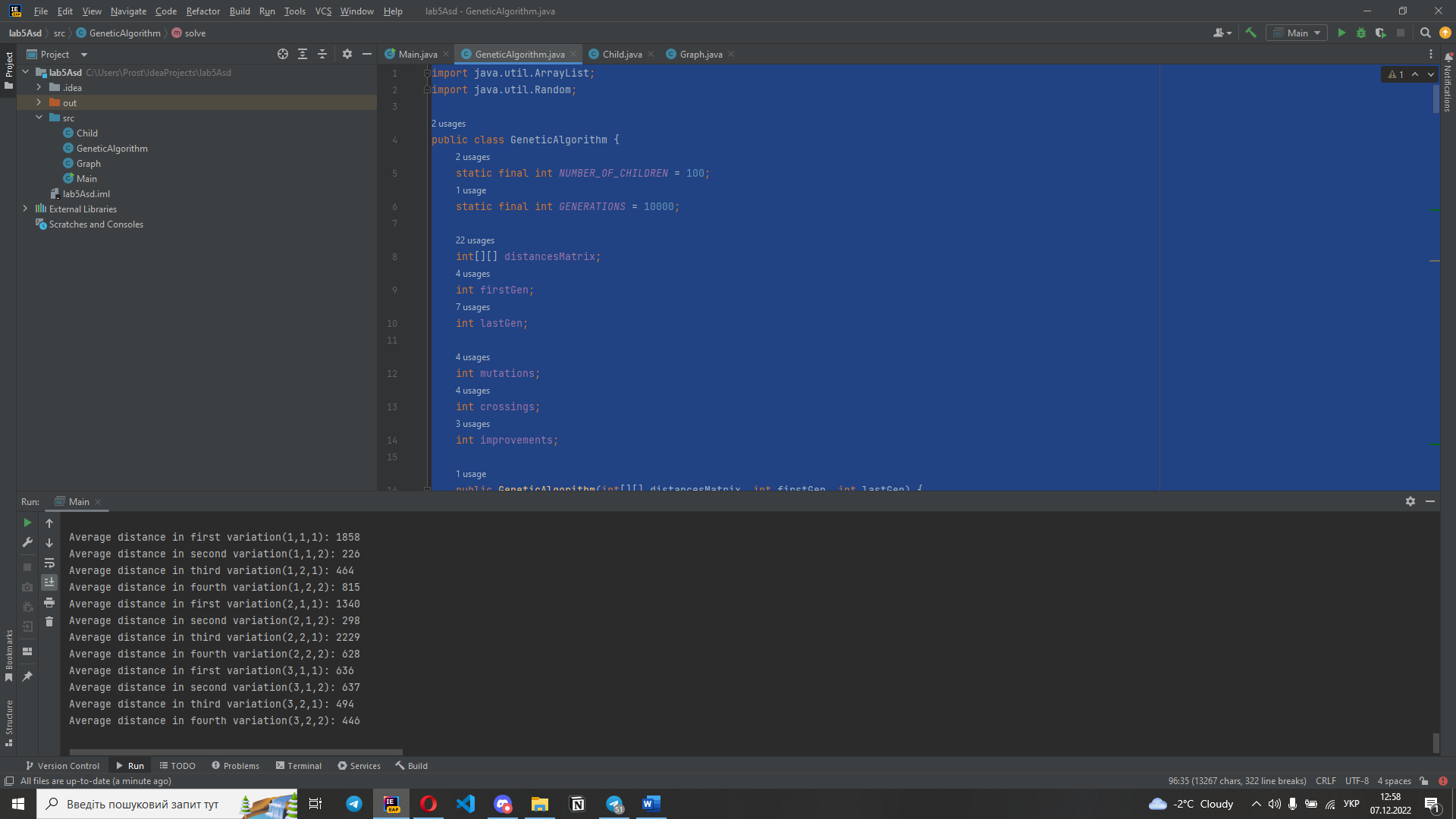


Рисунок 3.1 –

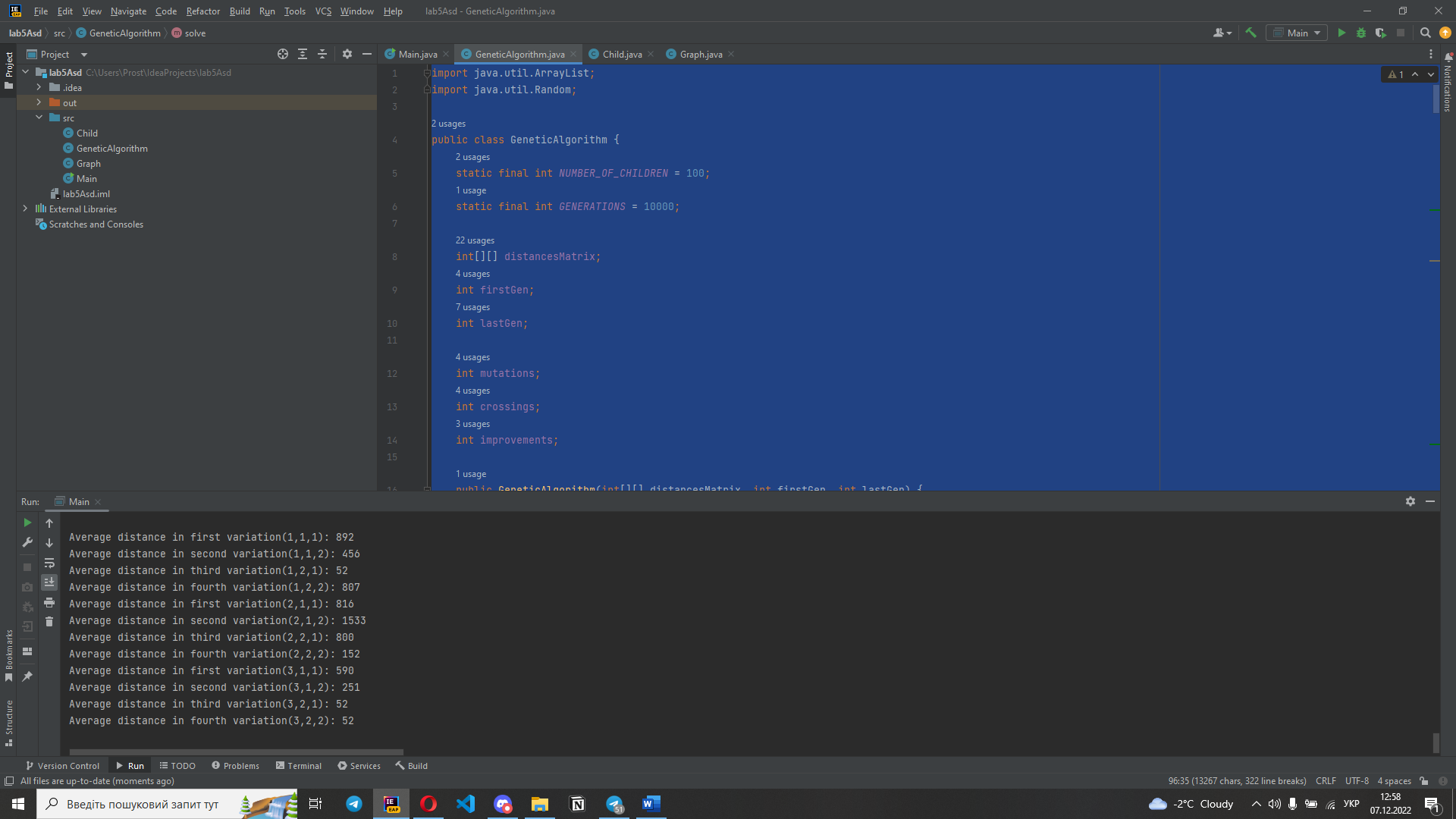
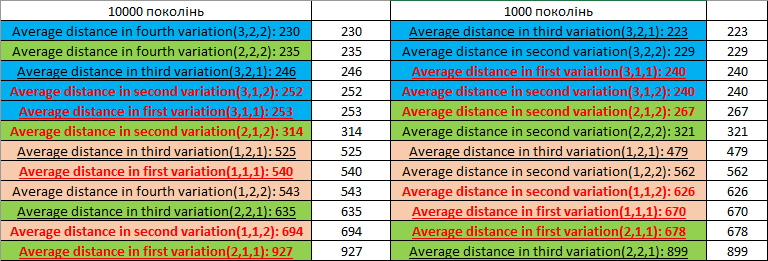
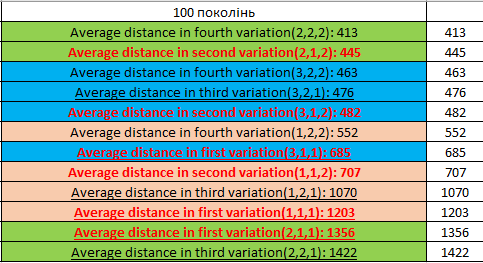
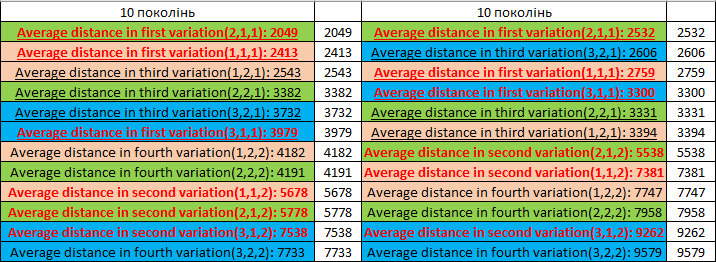


Рисунок 3.2 –

## Тестування алгоритму

В рамках тестування алгоритму було створено програмне забезпечення, що дозволяло проводити довільну кількість дослідів кожного варіанта роботи алгоритму. Було проведено по 25 дослідів для різної кількості поколінь з однаковим графом. Результати тестування наведені в таблиці 3.1. Фони різних кольорів відрізняють кросинговер, колір тексту – варіант мутації, підкресленість тексту – варіант локального оптимуму.





Таблиця 3.1 –

З результатів тестування видно, що при малій кількості поколінь найкоротший шлях найкраще шукає варіація з 2ого варіанта кросинговеру і перших варіантів мутації та локального оптимуму, але при збільшенні кількості ітерацій – найкраще працюють варіації з третьою варіацією кросинговеру (з’єднання за лівої частини першого батька і правої частини другого батька, відносно однакового гена) та другою варіацією мутацій (заміна одного гена на випадковий).

Висновок

В рамках даної лабораторної роботи було створено програмне забезпечення для реалізації і тестування генетичного алгоритму, де присутні три варіанта кросинговеру, два варіанта мутації та два варіанта локального покращення. В результату тестування програми виявилось, що при великій кількості поколінь локальний пошук фактично не впливає на розв’язок, а найкращий результат дають абсолютно випадкові мутації та кросинговер, що може утворити шлях з більшою кількістю генів у хромосомі (таким чином алгоритм стає менш «жадібним» і прораховує більшу кількість хороших варіантів розв’язку, що базуються на локальних оптимумах).

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%.