Додаток А

«Графічні матеріали атестаційної роботи»

ГЮИК.504310.002

(позначення документу)

ЗАТВЕРДЖЕНО
ГЮИК.504310.002

ДОСЛІДЖЕННЯ МЕТОДІВ ОБРОБКИ ДАНИХ У ВИСОКОНАВАНТАЖЕНИХ СИСТЕМАХ

Графічні матеріали атестаційної роботи

ГЮИК.504310.002

Аркушів 16

2019

Харківський Національний Університет Радіоелектроніки

«ЗАТВЕРДЖУЮ»
Керівник атестаційної роботи,
проф. *Рябченко І.М.*

РОЗРОБКА ТА ДОСЛІДЖЕННЯ ПІДСИСТЕМИ ПЛАНУВАННЯ ПЕРЕВЕЗЕНЬ КОМП'ЮТЕРИЗОВАНОЇ СИСТЕМИ УПРАВЛІННЯ ПІДПРИЄМСТВА

Графічні матеріали атестаційної роботи

ЛИСТ ЗАТВЕРДЖЕННЯ

ГЮИК.504310.002

УЗГОДЖЕНО:

РОЗРОБИЛА:
ст. гр. ІТПм-18-1
Ілюніна М.С.

2019

Додаток Б

«Текст програми»

ГЮИК.504310.002-01 12 01

Аркушів 8

ЗАТВЕРДЖЕНО
ГЮИК.504310.002-01 12 01

РОЗРОБКА ТА ДОСЛІДЖЕННЯ ПІДСИСТЕМИ ПЛАНУВАННЯ ПЕРЕВЕЗЕНЬ КОМП'ЮТЕРИЗОВАНОЇ СИСТЕМИ УПРАВЛІННЯ ПІДПРИЄМСТВА

Текст програми

ГЮИК.504310.002-01 12 01

Аркушів 17

Харківський Національний Університет Радіоелектроніки

«ЗАТВЕРДЖУЮ»
Керівник атестаційної роботи,
проф. *Рябченко І.М*

РОЗРОБКА ТА ДОСЛІДЖЕННЯ ПІДСИСТЕМИ ПЛАНУВАННЯ ПЕРЕВЕЗЕНЬ КОМП'ЮТЕРИЗОВАНОЇ СИСТЕМИ УПРАВЛІННЯ ПІДПРИЄМСТВА

Текст програми

ЛИСТ ЗАТВЕРДЖЕННЯ

ГЮИК.504310.002-01 12 01

УЗГОДЖЕНО:

РОЗРОБИЛА:
ст. гр. ІТПм-18-1
Ілюніна М.С.

2019

Додаток В

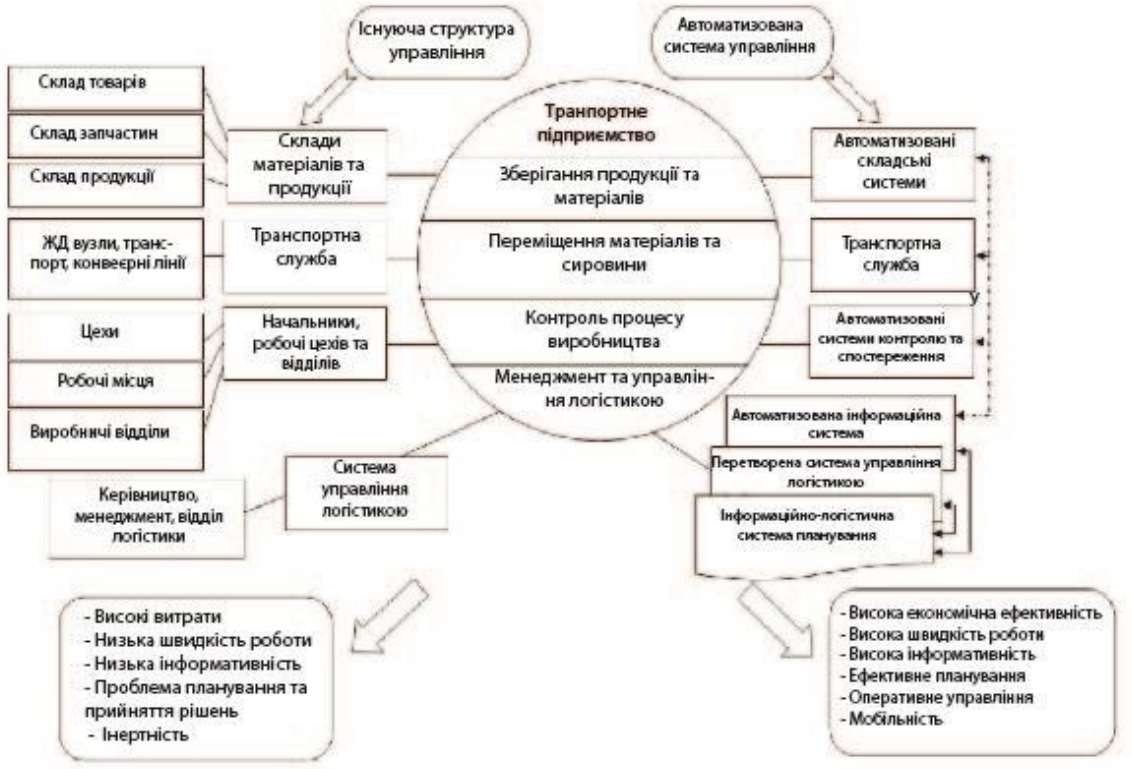
«Відомість атестаційної роботи»
ГЮИК.504310.002 ДЗ
(позначення документу)

2019

№	Позначення	Найменування	Дод. відомості
		Текстові документи	
1.	ГЮИК.504310.002 ПЗ	Пояснювальна записка	76 стор.
2.	ГЮИК.504310.002-01 12 01	Текст програми	8 стор.
		Графічні документи	
5.	ГЮИК.504310.002 С10	Структура системи управління підприємством	1 аркуш
6.	ГЮИК.504310.002 С10	Асинхронна модель однопоточного прийому та делегації обробки запитів	1 аркуш
7.	ГЮИК.504310.002 С10	Залежність часу розрахунку від числа вершин графа і числа	1 аркуш
8.	ГЮИК.504310.002 С10	Графічна інтерпретація алгоритму Флойда	1 аркуш
9.	ГЮИК.504310.002 С10	Графічна інтерпретація алгоритму Дейкстри	1 аркуш
10.	ГЮИК.504310.002 С10	Графічна інтерпретація алгоритму А*	1 аркуш
11.	ГЮИК.504310.002 С10	Задання кількості вершин у додатку	1 аркуш
12.	ГЮИК.504310.002 С10	Задання ваг шляхів для кожної вершини	1 аркуш
13.	ГЮИК.504310.002 С10	Результат розрахунку найкоротшого шляху	1 аркуш
14.	ГЮИК.504310.002 С10	Кількість виконаних операцій	1 аркуш
15.	ГЮИК.504310.002 С10	Задання вагів для 7 вершин	1 аркуш
16.	ГЮИК.504310.002 С10	Результат виконання алгоритму для 7 вершин	1 аркуш
17.	ГЮИК.504310.002 С10	Результат роботи алгоритму Дейкстри	1 аркуш
17.	ГЮИК.504310.002 С10	Результат роботи алгоритму А*	1 аркуш
18.	ГЮИК.504310.002 С10	Результат роботи мобільного додатку моделями	1 аркуш

Змін.	Арк.	№ докум	Підп.	Дата	ГЮИК.504310.002 ДЗ			
Розроб.		*Ілюніна М.С*			*Розробка та дослідження підсистеми планування перевезень комп'ютеризованої системи управління підприємства*		Акруш	Аркушів
Перев.		*Рябченко І.М*					*1*	*1*
Н. Контр.		*Рябченко І.М*				*ХНУРЕ Кафедра СТ*		
Затв.		*Гребеннік І.В.*						

Структура системи управління підприємством

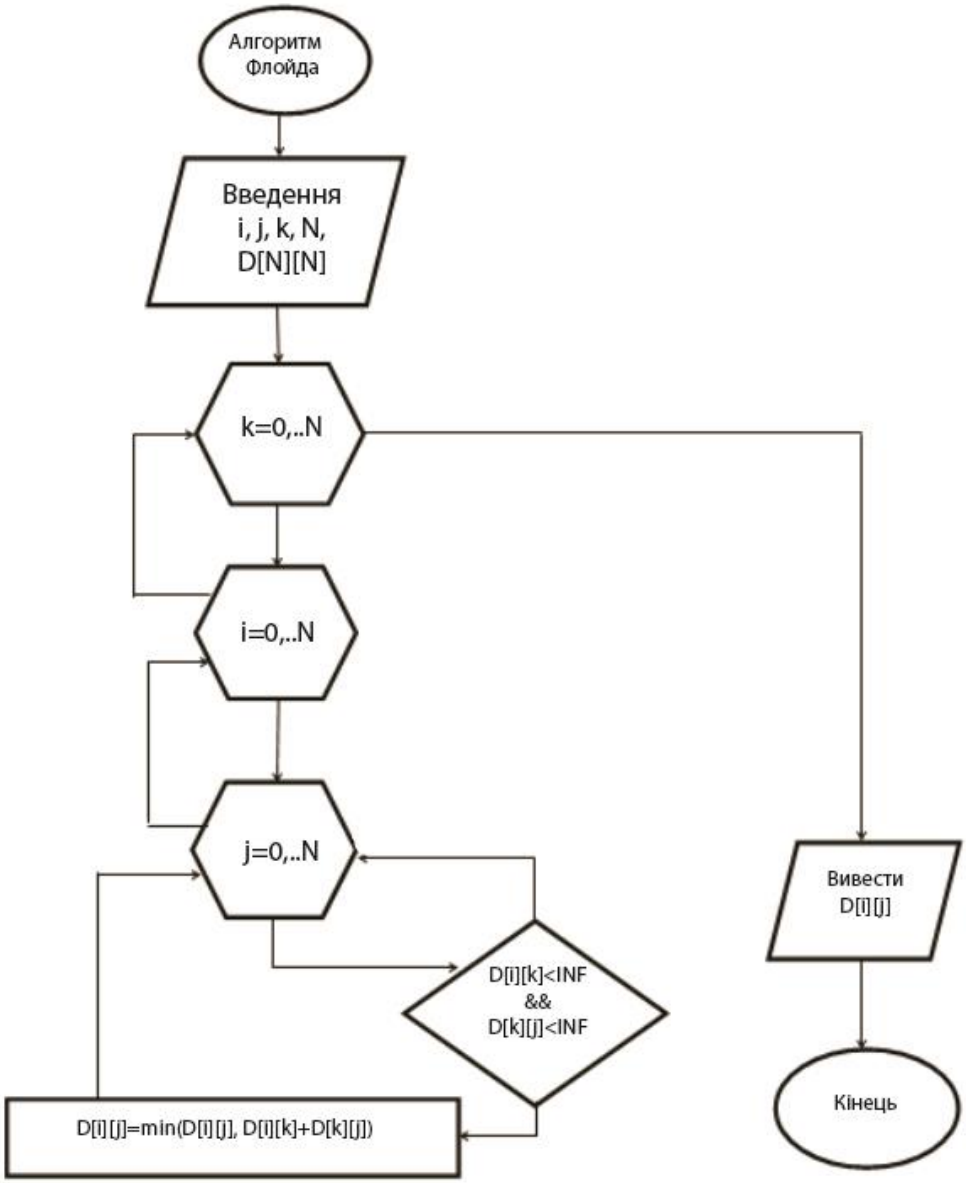


Розроб.	Ілюніна М.С.		12.12	Структура системи управління підприємством	
Перевір.	Гребеннік І.В.		12.12		
Н. Контр.	Гребеннік І.В.		12.12		
				ІТП-18-1	Лист 1
Затверд.	Гребеннік І.В.		12.12	СТ	Листів 2

Залежність часу розрахунку від числа вершин графа і числа процесорів

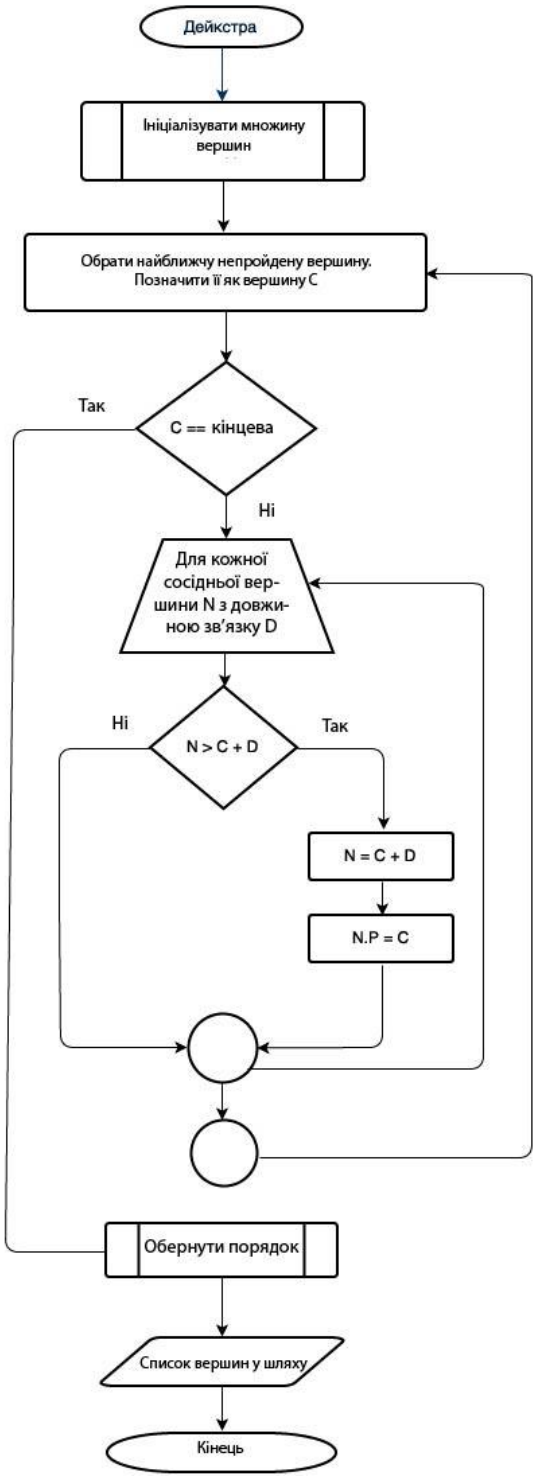
n	Время работы алгоритма					
	q=10	S_q	q=50	S_q	q=100	S_q
500	0,44	8,5	0,26	14,4	0,32	12,5
1000	6,34	8,6	2,78	19,7	1,89	28,9
3000	268,61	9,6	59,06	43,8	32,25	80,2

Графічна інтерпретація алгоритму Флойда



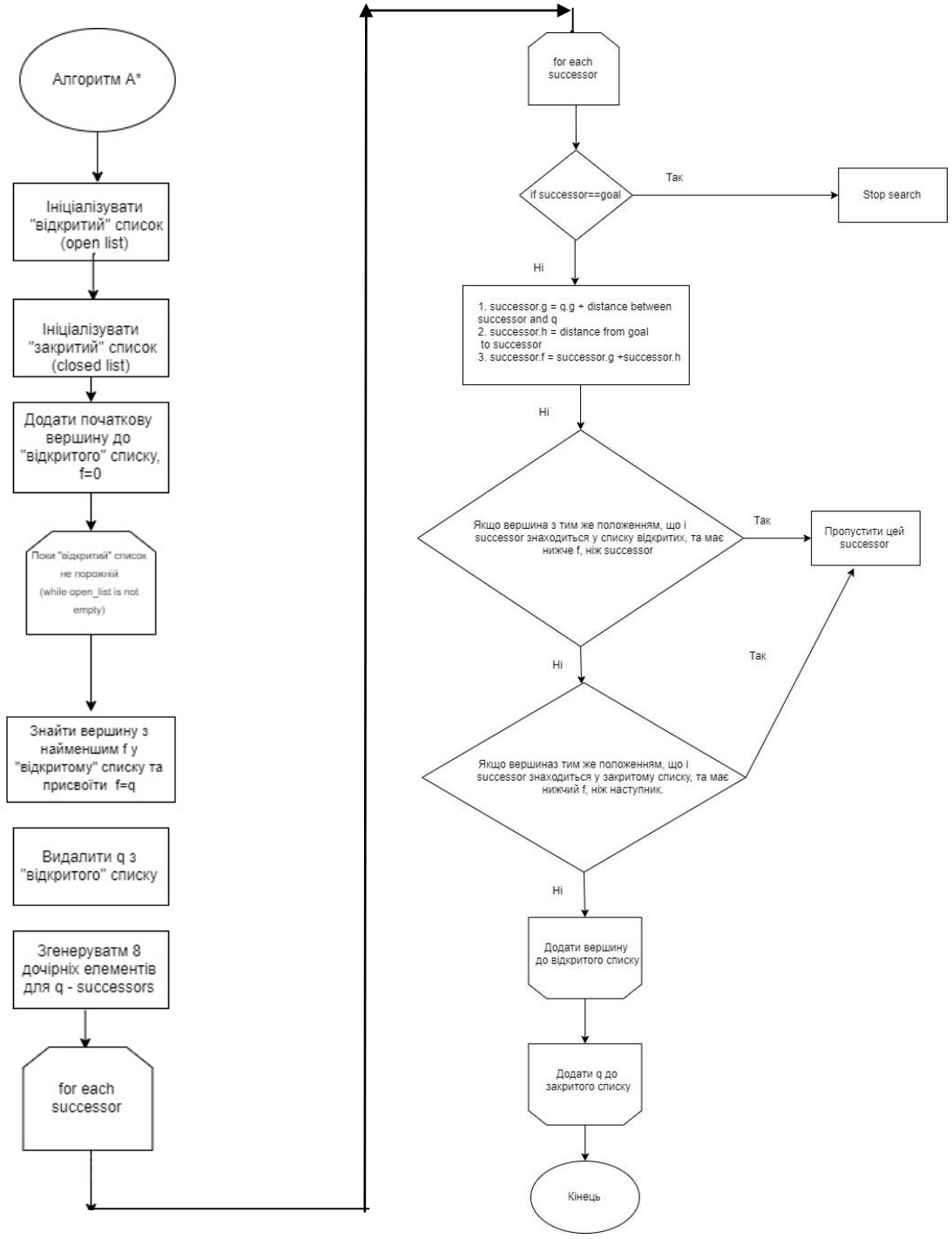
Розроб.	*Ілюніна М.С.*		12.12	*Графічна інтерпретація алгоритму Флойда*	
Перевір.	*Гребеннік І.В.*		12.12		
Н. Контр.	*Гребеннік І.В.*		12.12		
				ІТП-18-1	
Затверд.	*Гребеннік І.В.*		12.12	*СТ*	*Гребеннік І.В.*

Графічна інтерпретація алгоритму Дейкстри



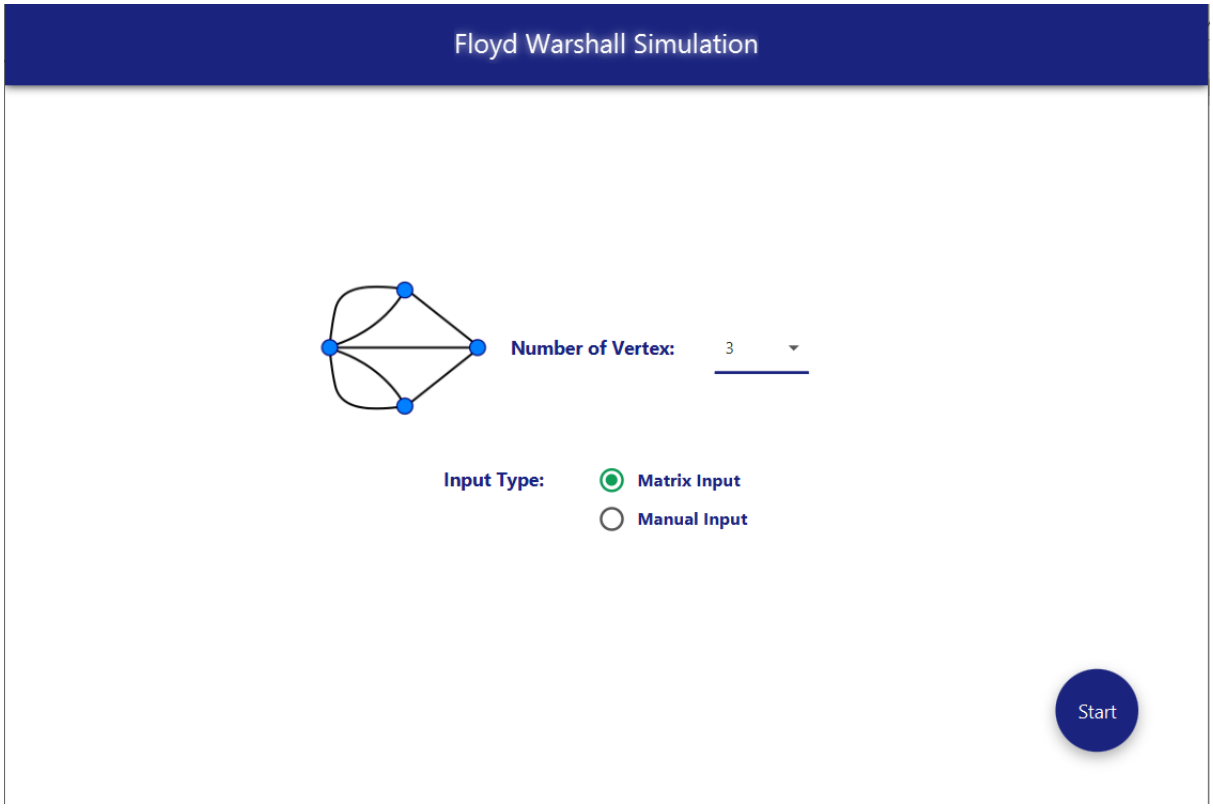
Розроб.	Ілюніна М.С.		12.12	Графічна інтерпретація алгоритму Дейкстри	
Перевір.	Гребеннік І.В.		12.12		
Н. Контр.	Гребеннік І.В.		12.12		
				ІТП-18-1	
Затверд.	Гребеннік І.В.		12.12	СТ	Гребеннік І.В.

Графічна інтерпретація алгоритму A*



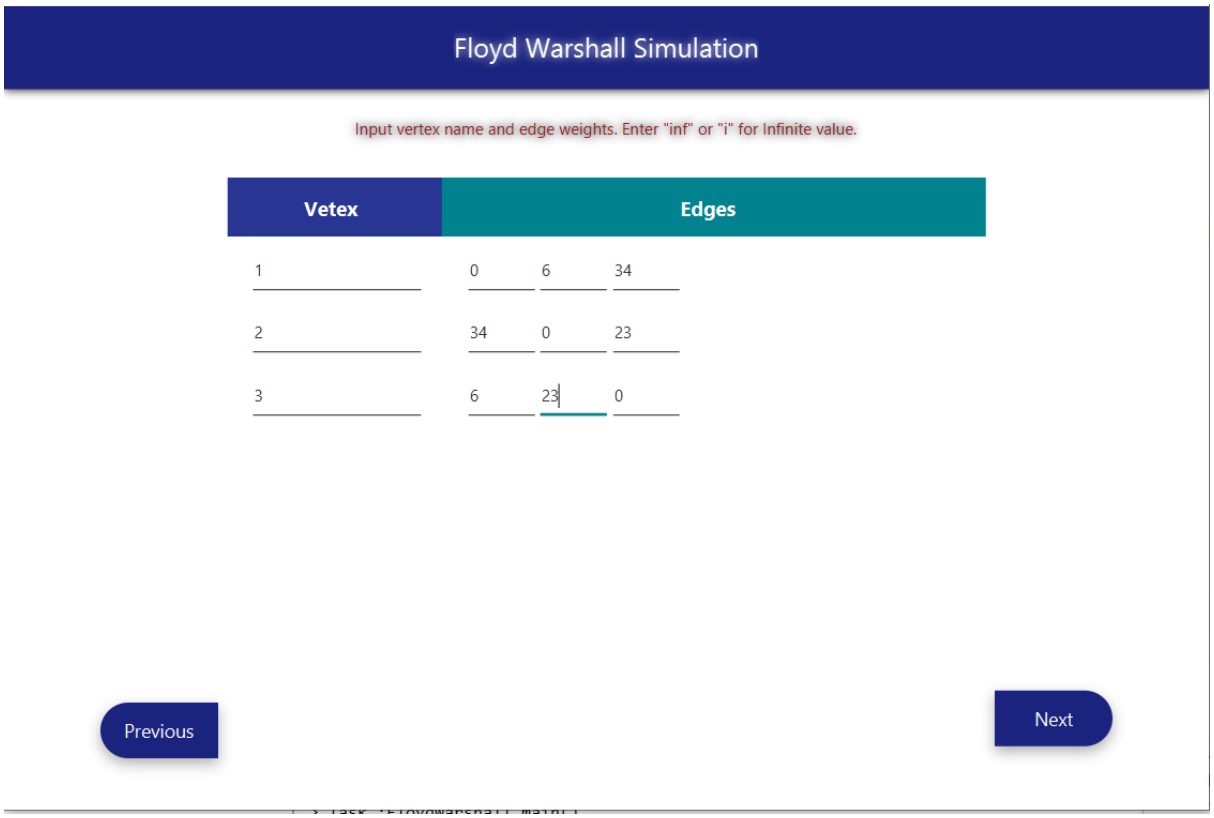
Розроб.	Ілюніна М.С.		12.12	Графічна інтерпретація алгоритму A*	
Перевір.	Гребеннік І.В.		12.12		
Н. Контр.	Гребеннік І.В.		12.12		
				ІТП-18-1	
Затверд.	Гребеннік І.В.		12.12	СТ	Гребеннік І.В.

Задання кількості вершин у додатку



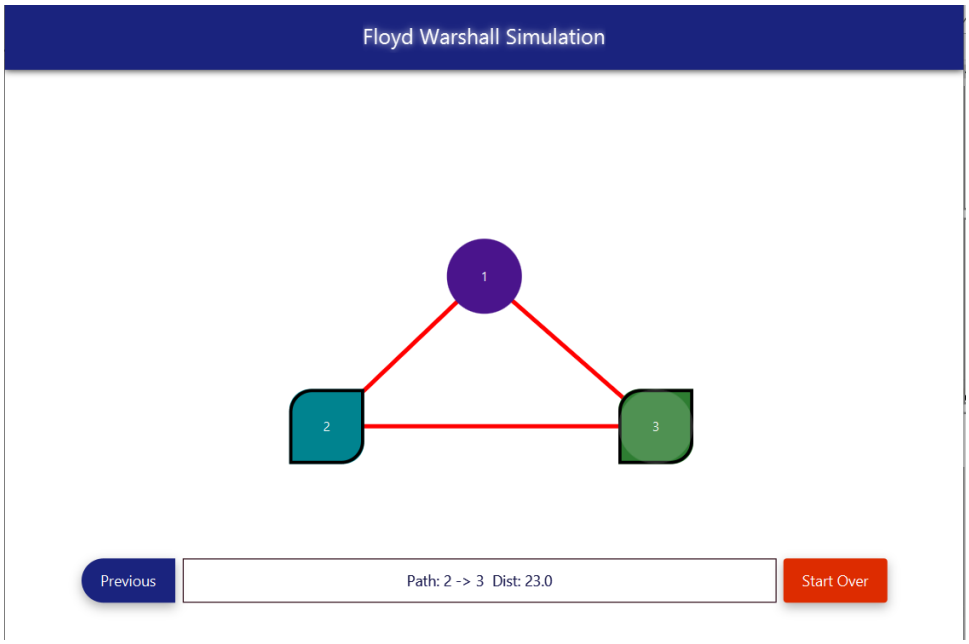
Розроб.	Ілюніна М.С.		12.12	Задання кількості вершин у додатку	
Перевір.	Гребеннік І.В.		12.12		
Н. Контр.	Гребеннік І.В.		12.12		
				ІТП-18-1	
Затверд.	Гребеннік І.В.		12.12	СТ	Гребеннік І.В.

Задання ваг шляхів для кожної вершини



Розроб.	Ілюніна М.С.		12.12	Задання ваг шляхів для кожної вершини	
Перевір.	Гребеннік І.В.		12.12		
Н. Контр.	Гребеннік І.В.		12.12		
				ІТП-18-1	
Затверд.	Гребеннік І.В.		12.12	СТ	Гребеннік І.В.

Результат розрахунку найкоротшого шляху



Розроб.	Ілюніна М.С.		12.12	Результат розрахунку найкоротшого шляху	
Перевір.	Гребеннік І.В.		12.12		
Н. Контр.	Гребеннік І.В.		12.12		
				ІТП-18-1	
Затверд.	Гребеннік І.В.		12.12	СТ	Гребеннік І.В.

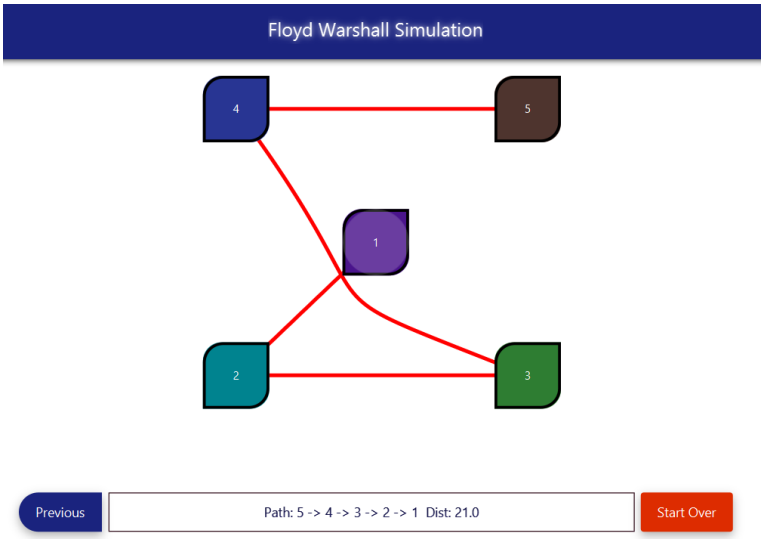
Кількість виконаних операцій

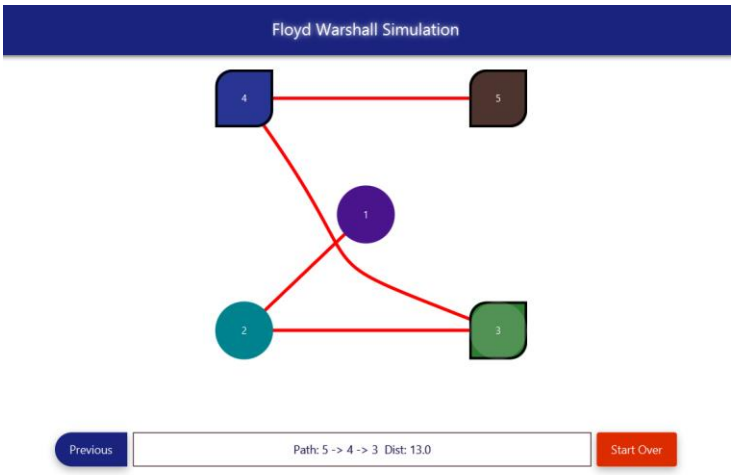
```
Calculations count27

BUILD SUCCESSFUL in 3m 29s
```

Задання ваг шляхів для кожної вершини

Розроб.	Ілюніна М.С.		12.12	Задання ваг шляхів для кожної вершини	
Перевір.	Гребеннік І.В.		12.12		
Н. Контр.	Гребеннік І.В.		12.12		
				ІТП-18-1	
Затверд.	Гребеннік І.В.		12.12	СТ	Гребеннік І.В.

Результат розрахунку найкоротшого шляху



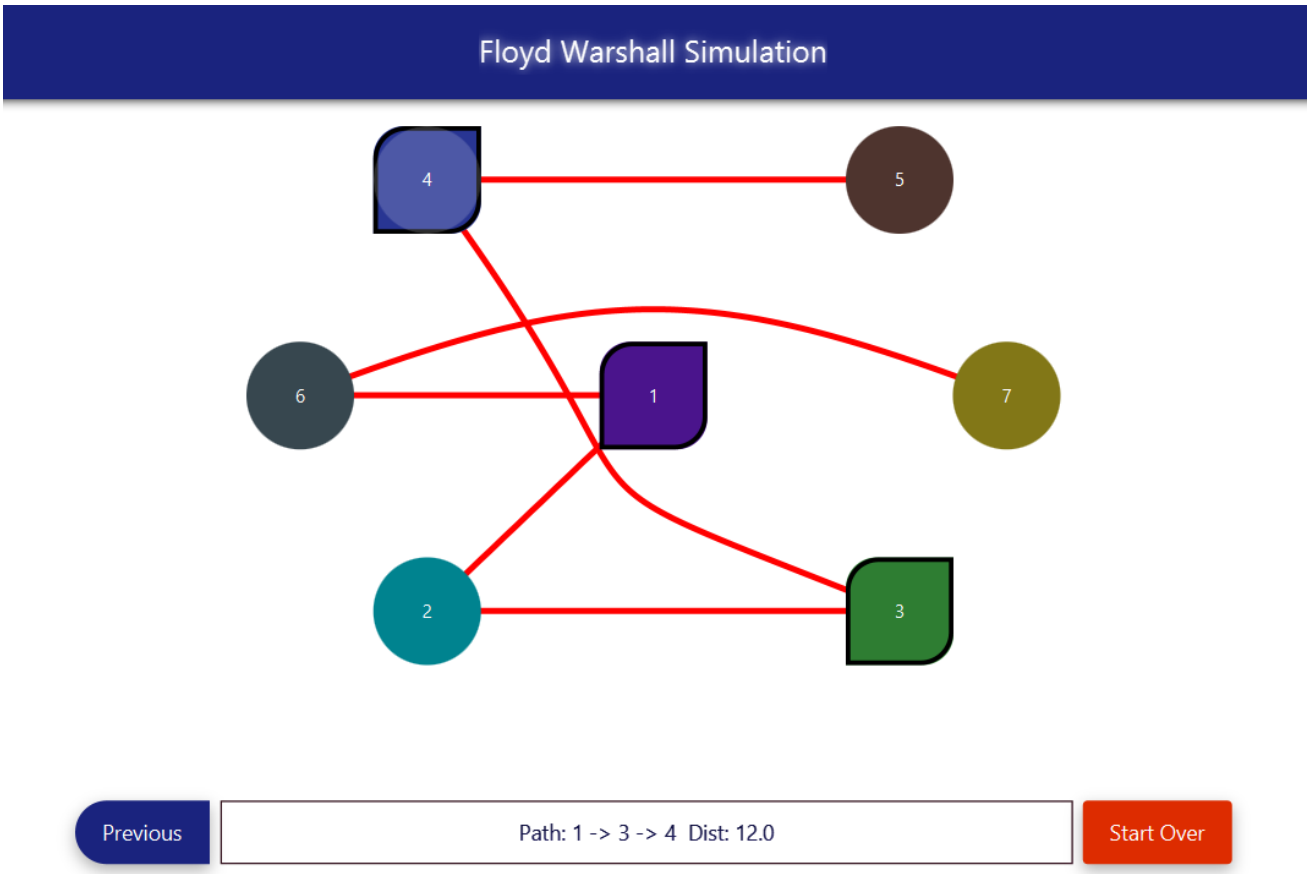


Розроб.	Ілюніна М.С.		12.12	Результат розрахунку найкоротшого шляху	
Перевір.	Гребеннік І.В.		12.12		
Н. Контр.	Гребеннік І.В.		12.12		
				ІТП-18-1	
Затверд.	Гребеннік І.В.		12.12	СТ	Гребеннік І.В.

Задання вагів для 7 вершин

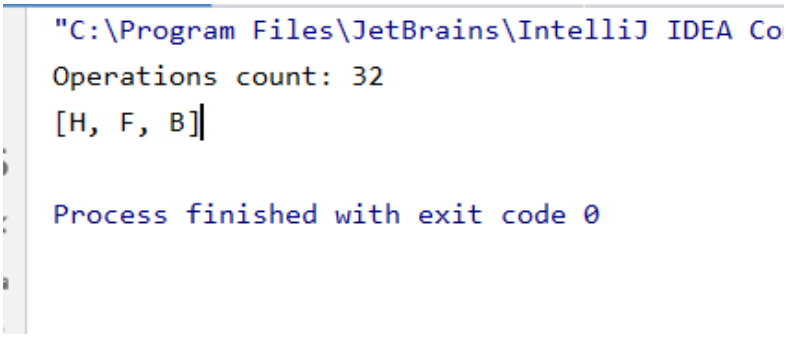
Розроб.	Ілюніна М.С.		12.12	Задання вагів для 7 вершин	
Перевір.	Гребеннік І.В.		12.12		
Н. Контр.	Гребеннік І.В.		12.12		
				ІТП-18-1	
Затверд.	Гребеннік І.В.		12.12	СТ	Гребеннік І.В.

– Результат виконання алгоритму для 7 вершин

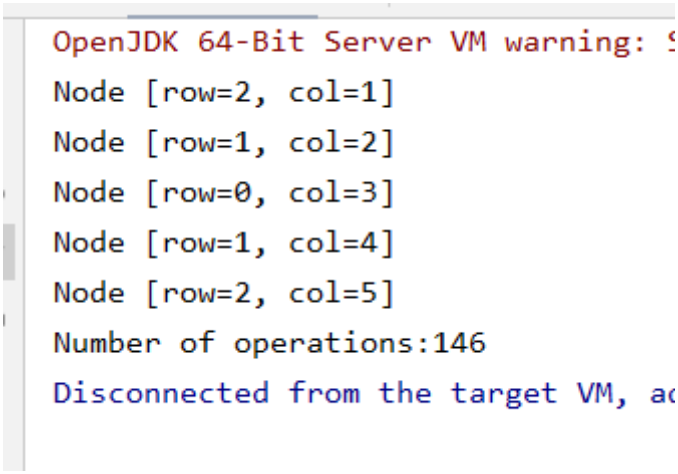


Розроб.	Ілюніна М.С.		12.12	– Результат виконання алгоритму для 7 вершин	
Перевір.	Гребеннік І.В.		12.12		
Н. Контр.	Гребеннік І.В.		12.12		
				ІТП-18-1	
Затверд.	Гребеннік І.В.		12.12	СТ	Гребеннік І.В.

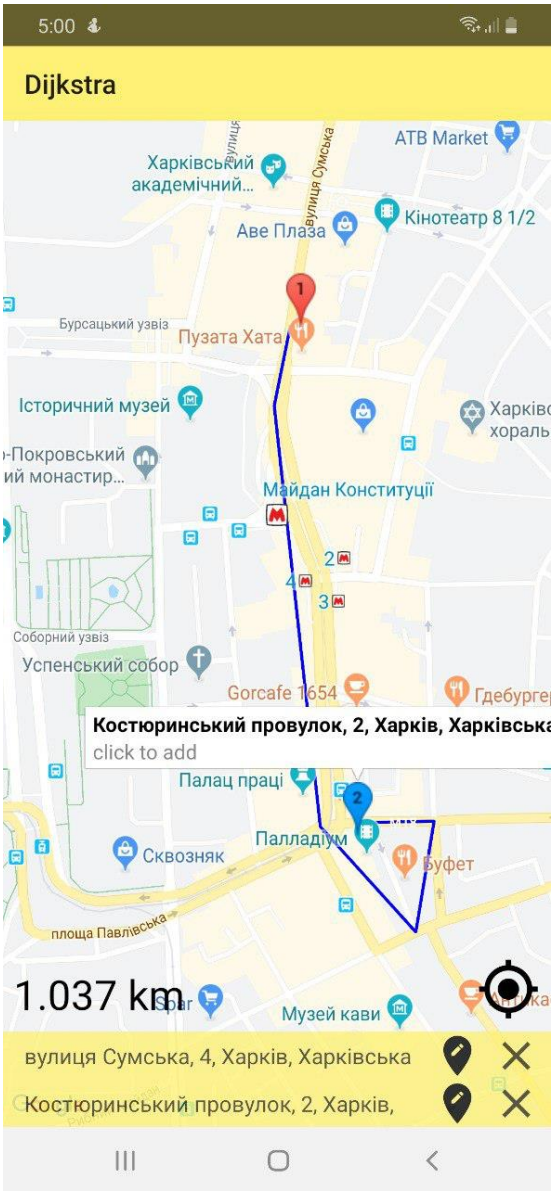
Результат роботи алгоритму Дейкстри



Результат роботи алгоритму A*



Результат роботи мобільного додатку



Розроб.	Ілюніна М.С.		12.12	Результат роботи мобільного додатку	
Перевір.	Гребеннік І.В.		12.12		
Н. Контр.	Гребеннік І.В.		12.12		
				ІТП-18-1	
Затверд.	Гребеннік І.В.		12.12	СТ	Гребеннік І.В.

```
package algo;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;

public class Dijkstras {

   public static void main(String[] args) {
      Graph g = new Graph();
      g.addVertex('A', Arrays.asList(new Vertex('B', 7), new Vertex('C',
8)));
      g.addVertex('B', Arrays.asList(new Vertex('A', 7), new Vertex('F',
2)));
      g.addVertex('C', Arrays.asList(new Vertex('A', 8), new Vertex('F', 6),
new Vertex('G', 4)));
      g.addVertex('D', Arrays.asList(new Vertex('F', 8)));
      g.addVertex('E', Arrays.asList(new Vertex('H', 1)));
      g.addVertex('F', Arrays.asList(new Vertex('B', 2), new Vertex('C', 6),
new Vertex('D', 8), new Vertex('G', 9), new Vertex('H', 3)));
      g.addVertex('G', Arrays.asList(new Vertex('C', 4), new Vertex('F',
9)));
      g.addVertex('H', Arrays.asList(new Vertex('E', 1), new Vertex('F',
3)));
      System.out.println(g.getShortestPath('A', 'H'));
   }

}

class Vertex implements Comparable<Vertex> {

   private Character id;
   private Integer distance;

   public Vertex(Character id, Integer distance) {
      super();
      this.id = id;
      this.distance = distance;
   }

   public Character getId() {
      return id;
   }

   public Integer getDistance() {
      return distance;
   }

   public void setId(Character id) {
      this.id = id;
```

```
}

public void setDistance(Integer distance) {
   this.distance = distance;
}

@Override
public int hashCode() {
   final int prime = 31;
   int result = 1;
   result = prime * result
         + ((distance == null) ? 0 : distance.hashCode());
   result = prime * result + ((id == null) ? 0 : id.hashCode());
   return result;
}

@Override
public boolean equals(Object obj) {
   if (this == obj)
      return true;
   if (obj == null)
      return false;
   if (getClass() != obj.getClass())
      return false;
   Vertex other = (Vertex) obj;
   if (distance == null) {
      if (other.distance != null)
         return false;
   } else if (!distance.equals(other.distance))
      return false;
   if (id == null) {
      if (other.id != null)
         return false;
   } else if (!id.equals(other.id))
      return false;
   return true;
}

@Override
public String toString() {
   return "Vertex [id=" + id + ", distance=" + distance + "]";
}

@Override
public int compareTo(Vertex o) {
   if (this.distance < o.distance)
      return -1;
   else if (this.distance > o.distance)
      return 1;
   else
      return this.getId().compareTo(o.getId());
}
```

```
}

class Graph {

   private final Map<Character, List<Vertex>> vertices;

   public Graph() {
      this.vertices = new HashMap<Character, List<Vertex>>();
   }

   public void addVertex(Character character, List<Vertex> vertex) {
      this.vertices.put(character, vertex);
   }

   public List<Character> getShortestPath(Character start, Character finish)
{
      int operationsCount = 0;
      final Map<Character, Integer> distances = new HashMap<Character,
Integer>();
      final Map<Character, Vertex> previous = new HashMap<Character,
Vertex>();
      PriorityQueue<Vertex> nodes = new PriorityQueue<Vertex>();

      for(Character vertex : vertices.keySet()) {
         if (vertex == start) {
            distances.put(vertex, 0);
            nodes.add(new Vertex(vertex, 0));
         } else {
            distances.put(vertex, Integer.MAX_VALUE);
            nodes.add(new Vertex(vertex, Integer.MAX_VALUE));
         }
         previous.put(vertex, null);
         operationsCount++;
      }

      while (!nodes.isEmpty()) {
         Vertex smallest = nodes.poll();
         if (smallest.getId() == finish) {
            final List<Character> path = new ArrayList<Character>();
            while (previous.get(smallest.getId()) != null) {
               operationsCount++;
               path.add(smallest.getId());
               smallest = previous.get(smallest.getId());
            }
            System.out.println("Operations count: " + operationsCount);
            return path;
         }

         if (distances.get(smallest.getId()) == Integer.MAX_VALUE) {
            break;
         }

         for (Vertex neighbor : vertices.get(smallest.getId())) {
```

```
            Integer alt = distances.get(smallest.getId()) +
neighbor.getDistance();
            if (alt < distances.get(neighbor.getId())) {
               distances.put(neighbor.getId(), alt);
               previous.put(neighbor.getId(), smallest);

               forloop:
               for(Vertex n : nodes) {
                  operationsCount++;
                  if (n.getId() == neighbor.getId()) {
                     nodes.remove(n);
                     n.setDistance(alt);
                     nodes.add(n);
                     break forloop;
                  }
               }
            }
         }
         operationsCount++;
      }
      System.out.println("Operations count: " + operationsCount);
      return new ArrayList<Character>(distances.keySet());
   }

}


package floyd.warshall;

/**
 *
 *
 */
public class FloydWarshallAlgorithm {
    //private static final int MAX = 8;

    private double[][][] res;
    private double[][][] sequence;

    public FloydWarshallAlgorithm(int value, Double[][] edges) {

        res = new double[value + 1][value][value];
        sequence = new double[value + 1][value][value];

        for (int i = 0; i < value; i++) {
            for (int j = 0; j < value; j++) {
                res[0][i][j] = edges[i][j];

                if(i == j)
                    sequence[0][i][j] = 0;
                else
                    sequence[0][i][j] = j + 1;
            }
        }

        System.out.println("FloydWarshall initializaation step: " + value*value);
```

```
        int x = 1;

        for (int k = 0; k < value; k++) {
            for (int i = 0; i < value; i++) {
                for (int j = 0; j < value; j++) {

                    sequence[x][i][j] = sequence[x - 1][i][j];

                    if (edges[i][j] > edges[i][k] + edges[k][j]) {
                        edges[i][j] = edges[i][k] + edges[k][j];
                        sequence[x][i][j] = x;
                    }

                    res[x][i][j] = edges[i][j];
                }
            }

            x++;
        }

        System.out.println("Calculations count"+value*value*value);
    }

    public double[][][] getPath()
    {
        return res;
    }

    public double[][][] getSequence()
    {
        return sequence;
    }
}

/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package floyd.warshall;

import com.jfoenix.controls.JFXButton;
import java.io.IOException;
import java.net.URL;
import java.util.ArrayList;
import java.util.ResourceBundle;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.fxml.Initializable;
import javafx.scene.control.Label;
import javafx.scene.layout.AnchorPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.CubicCurve;
import javafx.scene.shape.Line;

/**
 * FXML Controller class
 *
 * @author void
 */
```

```
public class GraphViewerController implements Initializable {

    public static String[] vertex;
    public static double[][][] shortestPath;
    public static double[][][] sequence;
    private double seqGen[][];
    public static int step;
    private JFXButton[] btn = new JFXButton[7];
    private static String[] colors = {"#4A148C", "#00838F", "#2E7D32", "#283593",
"#4E342E", "#37474F", "#827717"};
    public static boolean[] selection = new boolean[7];
    private boolean newInstance = false;

    @FXML
    private AnchorPane pane;

    @FXML
    private JFXButton startOverButton;

    @FXML
    private Label resultLabel;

    @FXML
    private JFXButton previousButton;

    @FXML
    void previousButtonAction(ActionEvent event) throws IOException {
        AnchorPane temp =
FXMLLoader.load(getClass().getResource("/CategoryChooser.fxml"));
        pane.getChildren().setAll(temp);
    }

    @FXML
    void startOverButtonAction(ActionEvent event) throws IOException {
        AnchorPane temp =
FXMLLoader.load(getClass().getResource("/FXMLDocument.fxml"));
        pane.getChildren().setAll(temp);
    }

    @Override
    public void initialize(URL url, ResourceBundle rb) {

        int[] layoutX = {415, 267, 576, 267, 576, 184, 646};
        int[] layoutY = {221, 362, 362, 80, 80, 221, 221};

        final int nodeRadius = 35;

        Line[][] newEdge = new Line[7][7];
        boolean[][] flag = new boolean[7][7];

        for (int i = 0; i < 7; i++) {
            btn[i] = new JFXButton();
        }

        //Generating Edges
        for (int i = 0; i < step; i++) {
            for (int j = 0; j < step; j++) {
                //Coordinate of starting point
                double x1 = layoutX[i] + nodeRadius;
                double y1 = layoutY[i] + nodeRadius;
```

```
                //Coordinate of ending point
                double x2 = layoutX[j] + nodeRadius;
                double y2 = layoutY[j] + nodeRadius;

                if (!flag[i][j]) {

                    newEdge[i][j] = new Line(x1, y1, x2, y2);

                    if ((i == 2 && j == 3) || (j == 2 && i == 3)) {
                        CubicCurve lineCurve = new CubicCurve();
                        lineCurve.setStartX(x1);
                        lineCurve.setStartY(y1);
                        lineCurve.setControlX1(layoutX[0] - 70);
                        lineCurve.setControlY1(layoutY[0] + 70);
                        lineCurve.setControlX2(layoutX[0] + 70);
                        lineCurve.setControlY2(layoutY[0] + 140);
                        lineCurve.setEndX(x2);
                        lineCurve.setEndY(y2);
                        if (shortestPath[0][i][j] != 0 && shortestPath[0][i][j] !=
Double.POSITIVE_INFINITY
                                && shortestPath[0][j][i] != 0 &&
shortestPath[0][j][i] != Double.POSITIVE_INFINITY) {
                            lineCurve.setStroke(Color.RED);
                            lineCurve.setStrokeWidth(4);
                            lineCurve.setFill(Color.web("#ffffff", 0));
                            flag[j][i] = true;
                            pane.getChildren().add(lineCurve);

                        } else if (shortestPath[0][i][j] != 0 &&
shortestPath[0][i][j] != Double.POSITIVE_INFINITY) {
                            lineCurve.setStroke(Color.web(colors[i]));
                            lineCurve.setStrokeWidth(4);
                            lineCurve.setFill(Color.web("#ffffff", 0));
                            pane.getChildren().add(lineCurve);
                        }

                    } else if ((i == 1 && j == 4) || (j == 1 && i == 4)) {
                        CubicCurve lineCurve = new CubicCurve();
                        lineCurve.setStartX(x1);
                        lineCurve.setStartY(y1);
                        lineCurve.setControlX1(layoutX[0]);
                        lineCurve.setControlY1(layoutY[0] - 40);
                        lineCurve.setControlX2(layoutX[0]);
                        lineCurve.setControlY2(layoutY[0] - 40);
                        lineCurve.setEndX(x2);
                        lineCurve.setEndY(y2);
                        if (shortestPath[0][i][j] != 0 && shortestPath[0][i][j] !=
Double.POSITIVE_INFINITY
                                && shortestPath[0][j][i] != 0 &&
shortestPath[0][j][i] != Double.POSITIVE_INFINITY) {
                            lineCurve.setStroke(Color.RED);
                            lineCurve.setStrokeWidth(4);
                            lineCurve.setFill(Color.web("#ffffff", 0));
                            flag[j][i] = true;
                            pane.getChildren().add(lineCurve);

                        } else if (shortestPath[0][i][j] != 0 &&
shortestPath[0][i][j] != Double.POSITIVE_INFINITY) {
                            lineCurve.setStroke(Color.web(colors[i]));
                            lineCurve.setStrokeWidth(4);
                            lineCurve.setFill(Color.web("#ffffff", 0));
```

```
                            pane.getChildren().add(lineCurve);
                        }

                    } else if ((i == 5 && j == 6) || (j == 5 && i == 6)) {
                        CubicCurve lineCurve = new CubicCurve();
                        lineCurve.setStartX(x1);
                        lineCurve.setStartY(y1);
                        lineCurve.setControlX1(layoutX[0]);
                        lineCurve.setControlY1(layoutY[0] - 40);
                        lineCurve.setControlX2(layoutX[0] + 70);
                        lineCurve.setControlY2(layoutY[0] - 40);
                        lineCurve.setEndX(x2);
                        lineCurve.setEndY(y2);

                        if (shortestPath[0][i][j] != 0 && shortestPath[0][i][j] !=
Double.POSITIVE_INFINITY
                                && shortestPath[0][j][i] != 0 &&
shortestPath[0][j][i] != Double.POSITIVE_INFINITY) {
                            lineCurve.setStroke(Color.RED);
                            lineCurve.setStrokeWidth(4);
                            lineCurve.setFill(Color.web("#ffffff", 0));
                            flag[j][i] = true;
                            pane.getChildren().add(lineCurve);

                        } else if (shortestPath[0][i][j] != 0 &&
shortestPath[0][i][j] != Double.POSITIVE_INFINITY) {
                            lineCurve.setStroke(Color.web(colors[i]));
                            lineCurve.setStrokeWidth(4);
                            lineCurve.setFill(Color.web("#ffffff", 0));
                            pane.getChildren().add(lineCurve);
                        }

                    } else if (shortestPath[0][i][j] != 0 && shortestPath[0][i][j]
!= Double.POSITIVE_INFINITY
                            && shortestPath[0][j][i] != 0 && shortestPath[0][j][i]
!= Double.POSITIVE_INFINITY) {
                        newEdge[i][j].setStroke(Color.RED);
                        flag[j][i] = true;
                        newEdge[i][j].setStrokeWidth(4);

                        pane.getChildren().add(newEdge[i][j]);
                    } else if (shortestPath[0][i][j] != 0 && shortestPath[0][i][j]
!= Double.POSITIVE_INFINITY) {

                        newEdge[i][j].setStroke(Color.web(colors[i]));
                        newEdge[i][j].setStrokeWidth(4);

                        pane.getChildren().add(newEdge[i][j]);
                    }

                }

            }
        }

        //Generating Vertices
        for (int i = 0; i < step; i++) {
            btn[i].setLayoutX(layoutX[i]);
            btn[i].setLayoutY(layoutY[i]);
            btn[i].setPrefHeight(70);
            btn[i].setPrefWidth(70);
```

```
            btn[i].setTextFill(Color.WHITE);
            btn[i].setStyle("-fx-background-color: " + colors[i] + "; -fx-
background-radius: 35 35 35 35");
            pane.getChildren().add(btn[i]);

            switch (i + 1) {
                case 1:
                    btn[i].setText(vertex[i]);
                    break;
                case 2:
                    btn[i].setText(vertex[i]);
                    break;
                case 3:
                    btn[i].setText(vertex[i]);
                    break;
                case 4:
                    btn[i].setText(vertex[i]);
                    break;
                case 5:
                    btn[i].setText(vertex[i]);
                    break;
                case 6:
                    btn[i].setText(vertex[i]);
                    break;
                case 7:
                    btn[i].setText(vertex[i]);
                    break;
            }
        }

        seqGen = new double[step + 1][step + 1];

        for (int x = 0; x < step + 1; x++) {
            for (int j = 0; j < step + 1; j++) {
                if (x != 0 && j != 0) {
                    seqGen[x][j] = sequence[step][x - 1][j - 1];
                }
            }
        }

        btn[0].setOnAction((event) -> {
            KopaShamsuKopa(btn[0], 0);
        });

        btn[1].setOnAction((event) -> {
            KopaShamsuKopa(btn[1], 1);
        });

        btn[2].setOnAction((event) -> {
            KopaShamsuKopa(btn[2], 2);
        });

        btn[3].setOnAction((event) -> {
            KopaShamsuKopa(btn[3], 3);
        });

        btn[4].setOnAction((event) -> {
            KopaShamsuKopa(btn[4], 4);
        });

        btn[5].setOnAction((event) -> {
```

```
            KopaShamsuKopa(btn[5], 5);
        });

        btn[6].setOnAction((event) -> {
            KopaShamsuKopa(btn[6], 6);
        });

    }

    private static int selectionCtr = 0;
    private static int first;
    private static JFXButton temp;

    private void KopaShamsuKopa(JFXButton btnx, int i) {
        if (selection[i]) {
            btnx.setStyle("-fx-background-color:" + colors[i] + "; -fx-background-
radius: 35 35 35 35");
            selection[i] = false;
            --selectionCtr;
        } else {
            if (selectionCtr == 0) {
                if (newInstance) {
                    for (int t = 0; t < step; t++) {
                        newInstance = false;
                        btn[t].setStyle("-fx-background-color:" + colors[t] + "; -
fx-background-radius: 35 35 35 35");;
                    }
                }
                btnx.setStyle("-fx-background-color:" + colors[i] + "; -fx-border-
color: #00BFA5; -fx-border-width: 3");
                selection[i] = true;
                first = i;
                temp = btnx;

                ++selectionCtr;

            } else {
                double res = shortestPath[step][first][i];

                //TO DO FIND PATH SEQUENCE
                //Genearating sequence
                String result = "";

                if (res != Double.POSITIVE_INFINITY) {
                    ArrayList<Integer> seq = sequenceGen(first + 1, i + 1);

                    int flag = 0;
                    for (Integer trav : seq) {
                        if (flag != 0) {
                            result += " -> " + vertex[trav - 1];
                        } else {
                            result += vertex[trav - 1];
                            flag++;
                        }

                        btn[trav - 1].setStyle("-fx-border-color: #000000;"
                                + " -fx-background-color:" + colors[trav - 1]
                                + "; -fx-background-radius: 20 0 20 0"
                                + "; -fx-border-width: 3; -fx-border-radius: 20 0
20 0");
                    }
```

```
                } else {
                    result = "No available path!";
                    btn[first].setStyle("-fx-background-color:" + colors[first] +
"; -fx-background-radius: 35 35 35 35");;
                }

                resultLabel.setText("Path: " + result + "  Dist: " +
Double.toString(res));
                selection[first] = false;
                selection[i] = false;
                selectionCtr = 0;
                newInstance = true;
            }
        }
    }

    //This method will find the shortest path and return it as a string

    public ArrayList<Integer> sequenceGen(int x, int y) {

        ArrayList<Integer> res = new ArrayList<>();

        while ((int) seqGen[x][y] != y) {
            res.add(x);
            x = (int) seqGen[x][y];
        }

        res.add(x);
        res.add(y);

        return res;

    }
}
package com.ai.astar;

import java.util.List;

public class AStarTest {

    public static void main(String[] args) {
        Node initialNode = new Node(2, 1);
        Node finalNode = new Node(2, 5);
        int rows = 6;
        int cols = 7;
        AStar aStar = new AStar(rows, cols, initialNode, finalNode);
        int[][] blocksArray = new int[][]{
                {1, 3},
                {2, 3},
                {3, 3}
        };
        aStar.setBlocks(blocksArray);
        List<Node> path = aStar.findPath();
        for (Node node : path) {
            System.out.println(node);
        }
        System.out.println("Number of operations:"+aStar.counter);
         -   -   -   -   -   -
    }
}
```

```
public class AStar {
    private static int DEFAULT_HV_COST = 10; // Horizontal - Vertical Cost
    private static int DEFAULT_DIAGONAL_COST = 14;
    private int hvCost;
    private int diagonalCost;
    private Node[][] searchArea;
    private PriorityQueue<Node> openList;
    private Set<Node> closedSet;
    private Node initialNode;
    private Node finalNode;
    public int counter;

    public AStar(int rows, int cols, Node initialNode, Node finalNode, int hvCost,
int diagonalCost) {
        this.hvCost = hvCost;
        this.diagonalCost = diagonalCost;
        setInitialNode(initialNode);
        setFinalNode(finalNode);
        this.searchArea = new Node[rows][cols];
        this.openList = new PriorityQueue<Node>(new Comparator<Node>() {
            @Override
            public int compare(Node node0, Node node1) {
                return Integer.compare(node0.getF(), node1.getF());
            }
        });
        setNodes();
        this.closedSet = new HashSet<>();
    }

    public AStar(int rows, int cols, Node initialNode, Node finalNode) {
        this(rows, cols, initialNode, finalNode, DEFAULT_HV_COST,
DEFAULT_DIAGONAL_COST);
    }

    private void setNodes() {
        for (int i = 0; i < searchArea.length; i++) {
            for (int j = 0; j < searchArea[0].length; j++) {
                Node node = new Node(i, j);
                node.calculateHeuristic(getFinalNode());
                this.searchArea[i][j] = node;
            }
        }
    }

    public void setBlocks(int[][] blocksArray) {
        for (int i = 0; i < blocksArray.length; i++) {
            int row = blocksArray[i][0];
            int col = blocksArray[i][1];
            setBlock(row, col);
        }
    }

    public List<Node> findPath() {
        openList.add(initialNode);
        while (!isEmpty(openList)) {
            Node currentNode = openList.poll();
            closedSet.add(currentNode);
            if (isFinalNode(currentNode)) {
                return getPath(currentNode);
            } else {
                counter++;
```

```
                addAdjacentNodes(currentNode);
            }
        }
        return new ArrayList<Node>();
    }

    private List<Node> getPath(Node currentNode) {
        List<Node> path = new ArrayList<Node>();
        path.add(currentNode);
        Node parent;
        while ((parent = currentNode.getParent()) != null) {
            counter++;
            path.add(0, parent);
            currentNode = parent;
        }
        return path;
    }

    private void addAdjacentNodes(Node currentNode) {
        addAdjacentUpperRow(currentNode);
        addAdjacentMiddleRow(currentNode);
        addAdjacentLowerRow(currentNode);
    }

    private void addAdjacentLowerRow(Node currentNode) {
        int row = currentNode.getRow();
        int col = currentNode.getCol();
        int lowerRow = row + 1;
        if (lowerRow < getSearchArea().length) {
            if (col - 1 >= 0) {
                checkNode(currentNode, col - 1, lowerRow, getDiagonalCost()); //
Comment this line if diagonal movements are not allowed
            }
            if (col + 1 < getSearchArea()[0].length) {
                checkNode(currentNode, col + 1, lowerRow, getDiagonalCost()); //
Comment this line if diagonal movements are not allowed
            }
            checkNode(currentNode, col, lowerRow, getHvCost());
        }
    }

    private void addAdjacentMiddleRow(Node currentNode) {
        int row = currentNode.getRow();
        int col = currentNode.getCol();
        int middleRow = row;
        if (col - 1 >= 0) {
            checkNode(currentNode, col - 1, middleRow, getHvCost());
        }
        if (col + 1 < getSearchArea()[0].length) {
            checkNode(currentNode, col + 1, middleRow, getHvCost());
        }
    }

    private void addAdjacentUpperRow(Node currentNode) {
        int row = currentNode.getRow();
        int col = currentNode.getCol();
        int upperRow = row - 1;
        if (upperRow >= 0) {
            if (col - 1 >= 0) {
                checkNode(currentNode, col - 1, upperRow, getDiagonalCost()); //
Comment this if diagonal movements are not allowed
```

```
            }
            if (col + 1 < getSearchArea()[0].length) {
                checkNode(currentNode, col + 1, upperRow, getDiagonalCost()); //
Comment this if diagonal movements are not allowed
            }
            checkNode(currentNode, col, upperRow, getHvCost());
        }
    }

    private void checkNode(Node currentNode, int col, int row, int cost) {
        Node adjacentNode = getSearchArea()[row][col];
        if (!adjacentNode.isBlock() && !getClosedSet().contains(adjacentNode)) {
            if (!getOpenList().contains(adjacentNode)) {
                counter++;
                adjacentNode.setNodeData(currentNode, cost);
                getOpenList().add(adjacentNode);
            } else {
                counter++;
                boolean changed = adjacentNode.checkBetterPath(currentNode, cost);
                if (changed) {
                    // Remove and Add the changed node, so that the PriorityQueue
can sort again its
                    // contents with the modified "finalCost" value of the modified
node
                    getOpenList().remove(adjacentNode);
                    getOpenList().add(adjacentNode);
                }
            }
        }
    }

    private boolean isFinalNode(Node currentNode) {
        return currentNode.equals(finalNode);
    }

    private boolean isEmpty(PriorityQueue<Node> openList) {
        return openList.size() == 0;
    }

    private void setBlock(int row, int col) {
        this.searchArea[row][col].setBlock(true);
    }

    public Node getInitialNode() {
        return initialNode;
    }

    public void setInitialNode(Node initialNode) {
        this.initialNode = initialNode;
    }

    public Node getFinalNode() {
        return finalNode;
    }

    public void setFinalNode(Node finalNode) {
        this.finalNode = finalNode;
    }

    public Node[][] getSearchArea() {
        return searchArea;
```

```
    }

    public void setSearchArea(Node[][] searchArea) {
        this.searchArea = searchArea;
    }

    public PriorityQueue<Node> getOpenList() {
        return openList;
    }

    public void setOpenList(PriorityQueue<Node> openList) {
        this.openList = openList;
    }

    public Set<Node> getClosedSet() {
        return closedSet;
    }

    public void setClosedSet(Set<Node> closedSet) {
        this.closedSet = closedSet;
    }

    public int getHvCost() {
        return hvCost;
    }

    public void setHvCost(int hvCost) {
        this.hvCost = hvCost;
    }

    private int getDiagonalCost() {
        return diagonalCost;
    }

    private void setDiagonalCost(int diagonalCost) {
        this.diagonalCost = diagonalCost;
    }
}



ublic class Node {

    private int g;
    private int f;
    private int h;
    private int row;
    private int col;
    private boolean isBlock;
    private Node parent;

    public Node(int row, int col) {
        super();
        this.row = row;
        this.col = col;
    }

    public void calculateHeuristic(Node finalNode) {
        this.h = Math.abs(finalNode.getRow() - getRow()) +
Math.abs(finalNode.getCol() - getCol());
    }
```

```
    public void setNodeData(Node currentNode, int cost) {
        int gCost = currentNode.getG() + cost;
        setParent(currentNode);
        setG(gCost);
        calculateFinalCost();
    }

    public boolean checkBetterPath(Node currentNode, int cost) {
        int gCost = currentNode.getG() + cost;
        if (gCost < getG()) {
            setNodeData(currentNode, cost);
            return true;
        }
        return false;
    }

    private void calculateFinalCost() {
        int finalCost = getG() + getH();
        setF(finalCost);
    }

    @Override
    public boolean equals(Object arg0) {
        Node other = (Node) arg0;
        return this.getRow() == other.getRow() && this.getCol() == other.getCol();
    }

    @Override
    public String toString() {
        return "Node [row=" + row + ", col=" + col + "]";
    }

    public int getH() {
        return h;
    }

    public void setH(int h) {
        this.h = h;
    }

    public int getG() {
        return g;
    }

    public void setG(int g) {
        this.g = g;
    }

    public int getF() {
        return f;
    }

    public void setF(int f) {
        this.f = f;
    }

    public Node getParent() {
        return parent;
    }
```

```
    public void setParent(Node parent) {
        this.parent = parent;
    }

    public boolean isBlock() {
        return isBlock;
    }

    public void setBlock(boolean isBlock) {
        this.isBlock = isBlock;
    }

    public int getRow() {
        return row;
    }

    public void setRow(int row) {
        this.row = row;
    }

    public int getCol() {
        return col;
    }

    public void setCol(int col) {
        this.col = col;
    }
}
```