TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

Trường Công nghệ thông tin và truyền thông

BÁO CÁO PROJECT

MÔN THỰC HÀNH HƯỚNG ĐỐI TƯỢNG

Topic: 9

Giảng viên hướng dẫn : Cô Bùi Thị Mai Anh

Nhóm thực hiện : Nhóm 11

Lương Nguyễn Hoàng Anh – 20194471

Vũ Tài Công – 20194496

Nguyễn Anh Tú – 20194699

Hà Nội 2022

Mục Lục

Phân công thành viên trong nhóm

Thành viên	Vai trò	Nhiệm vụ
Lương Nguyễn Hoàng Anh	Leader	- Thiết kế giải thuật Dynamic Programming - Đưa ra ý tưởng về các class trong project - Thiết kế giao diện - Viết báo cáo
Vũ Tài Công	Member	- Thiết kế giải thuật Dijkstra - Bổ sung thêm trong thiết kế class trong project - Vẽ Class Diagram
Nguyễn Anh Tú	Member	- Thiết kế giải thuật Bellman Ford - Bổ sung thêm trong thiết kế class trong project

I. Phân tích yêu cầu bài toán

1.1. Yêu cầu bài toán:

Cho đồ thị G, tìm đường đi ngắn nhất từ một đỉnh tới tất cả các đỉnh khác trong đồ thị có hướng có trọng số. Yêu cầu sử dụng giải thuật:

- Dynamic Programming
- Bellman Ford
- Dijkstra

1.2. Mô tả giải thuật

a. Dynamic Programming:

Dynamic Programming (hay Quy hoạch động) được sử dụng để giải các bài toán có tính chất gối nhau hay các bài toán tối ưu. Trong phạm vi topic 9 thì thuật toán này được áp dụng để giải bài toán tối ưu tìm đường đi ngắn nhất.

Ý tưởng thuật toán:

Xét đồ thị G với N đỉnh, ta xây dựng ma trận trọng số W với N cột N hàng, biểu diễn đường đi ngắn nhất giữa hai đỉnh bất kì, trong đó ô (i, j) với trọng số $W_{i,j}$ thể hiện khoảng cách ngắn nhất giữa hai đỉnh.

Nếu coi đường đi ngắn nhất giữa 2 đỉnh i, j là d[i][j] và d[i][j] = d[i][k] + d[k][j] với k là điểm nằm giữa i và j, thì chắc chắn d[i][k] và d[k][j] là đường đi ngắn nhất của i,k và k,j. Như vậy nếu tính được sẵn d[i][k] và d[k][j] và lưu nó vào ma trận W thì khi tính d[i][j] rất dễ dàng.

b. Bellman Ford:

Ý tưởng thuật toán:

Ta thực hiện duyệt n lần, n là số đỉnh của đồ thị. Với mỗi lần duyệt, ta tìm tất cả các cạnh mà đường đi đi qua cạnh đó sẽ rút ngắn đường đi từ đỉnh gốc tới một đỉnh khác. Ở lần duyệt thứ n, nếu còn bất kì cạnh nào có thể rút ngắn đường đi, điều đó chứng tỏ đồ thị có chu trình âm, và ta kết thúc thuật toán.

c. Dijkstra:

Dijkstra là một trong những thuật toán cổ điển để giải quyết bài toán tìm đường đi ngắn nhất từ một điểm cho trước tới các điểm còn lại trong đồ thị trọng số.

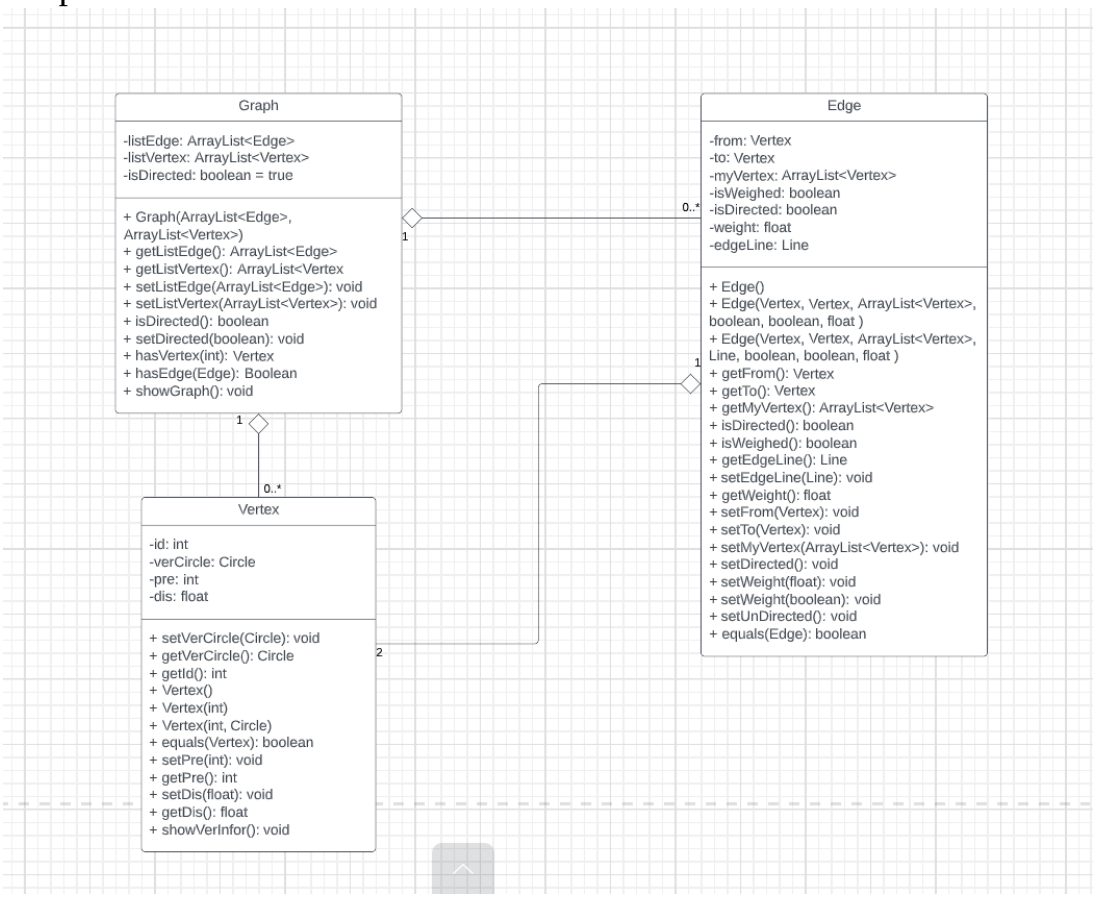
Ý tưởng thuật toán :

Duyệt từ đỉnh bắt đầu duyệt qua các đỉnh kề của đỉnh đang thăm, cập nhật đường đi ngắn dựa vào trọng số và đường đi ngắn nhất từ đỉnh bắt đầu đến các đỉnh hàng xóm. Sau đó chọn đỉnh có khoảng cách ngắn nhất trong tập các đỉnh đang xét, cập nhật thành đỉnh thăm và lặp lại vòng lặp cho đến hết tập đỉnh.

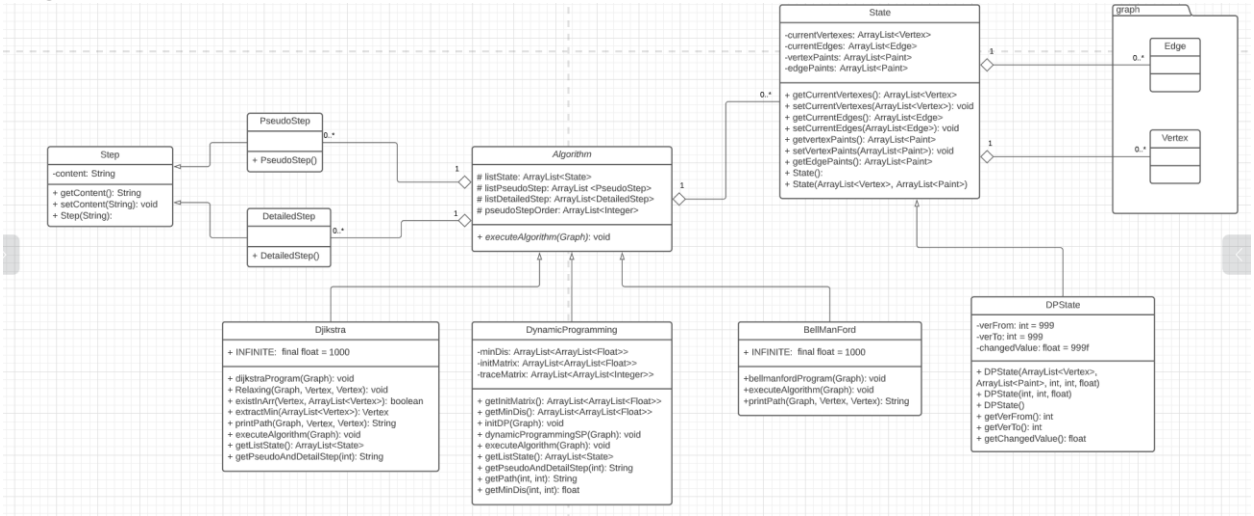
II. Phân tích thiết kế

1.1. Biểu đồ Class Diagram

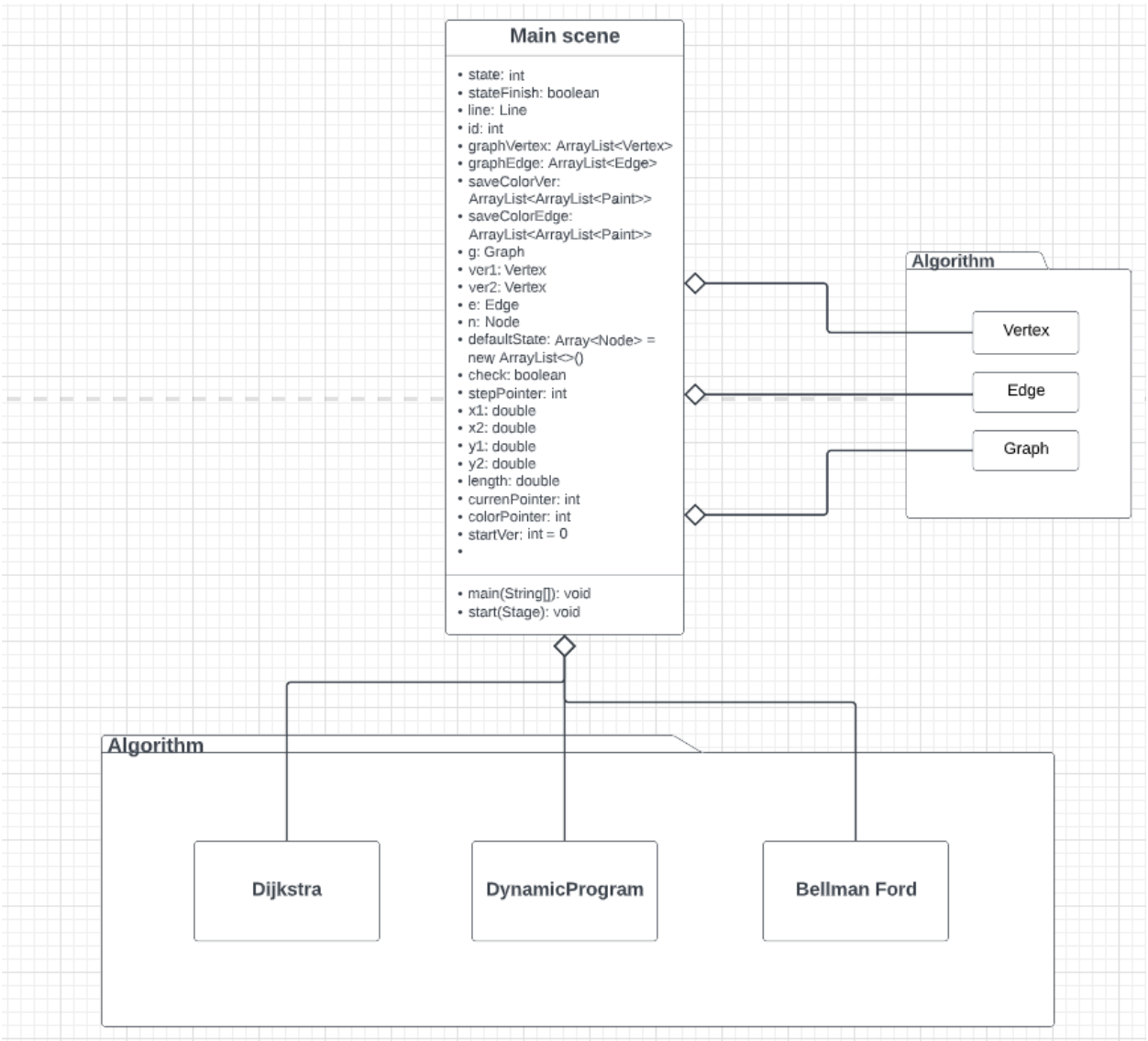
- Graph



- Algorithm



- MainScene


Main scene
• state: int
• stateFinish: boolean
• line: Line
• id: int
• graphVertex: ArrayList<Vertex>
• graphEdge: ArrayList<Edge>
• saveColorVer: ArrayList<ArrayList<Paint>>
• saveColorEdge: ArrayList<ArrayList<Paint>>
• g: Graph
• ver1: Vertex
• ver2: Vertex
• e: Edge
• n: Node
• defaultState: Array<Node> = new ArrayList<>()
• check: boolean
• stepPointer: int
• x1: double
• x2: double
• y1: double
• y2: double
• length: double
• currenPointer: int
• colorPointer: int
• startVer: int = 0
•
• main(String[]): void
• start(Stage): void
Algorithm
Vertex
Edge
Graph
Algorithm
Dijkstra
DynamicProgram
Bellman Ford

1.2. Thiết kế hướng đối tượng trong Project

a. Tính trừu tượng hóa (Abstraction)

Tính trừu tượng là một cơ chế cho phép biểu diễn một tình huống phức tạp trong thế giới thực bằng một mô hình được đơn giản hóa. Nó bao gồm việc tập trung vào các tính chất quan trọng của một đối tượng khi phải làm việc với lượng lớn thông tin.

Nhóm đã áp dụng tính chất này của thiết kế hướng đối tượng khi thiết kế các giải thuật, bằng việc trừu tượng hóa class Algorithm và chỉ thể hiện những thuộc tính và phương thức đặc trưng của một giải thuật như phương thức thực thi,…

```
public abstract class Algorithm {

    protected ArrayList<State> listState;
    protected ArrayList<PseudoStep>listPseudoStep;
    protected ArrayList<Integer>pseudoStepOrder;
    protected ArrayList<DetailedStep>listDetailedStep;

    //protected int stepPointer;

    public abstract void executeAlgorithm(Graph graph);

    //public abstract void run();
}
```

b. Tính đóng gói (Encapsulation)

Tính đóng gói là kỹ thuật ẩn giấu thông tin không liên quan và hiển thị thông tin không liên quan. Thông qua các phương thức truy cập như Getter hoặc Setter, tính đóng gói sẽ đảm bảo việc che giấu thông tin (data hiding) – ngăn chặn các lớp bên ngoài truy cập, thay đổi thuộc tính và phương thức.

Nhóm đã áp dụng tính chất này bằng việc đặt access modifier của các thuộc tính của các lớp về *private*. Sau đó thiết lập các phương thức getter và setter để có thể truy cập dữ liệu trong các lớp.

```
private ArrayList<Edge> listEdge;
private ArrayList<Vertex> listVertex;

private boolean isDirected = true;

public ArrayList<Edge> getListEdge() { return listEdge; }

public ArrayList<Vertex> getListVertex() { return listVertex; }

public void setListEdge(ArrayList<Edge> listEdge) { this.listEdge = listEdge; }

public void setListVertex(ArrayList<Vertex> listVertex) { this.listVertex = listVertex; }
```

```
private int id;

private int pre = -1;

private Circle verCircle;

public void setVerCircle(Circle verCircle) { this.verCircle = verCircle; }

public Circle getVerCircle() { return verCircle; }

public int getPre() { return pre; }

public void setPre(int pre) { this.pre = pre; }

public int getId() { return id; }

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

c. Tính kế thừa (Inheritance)

Tính kế thừa là quan hệ mang tính phân cấp mà trong đó các thành viên của một lớp được kế thừa bởi các lớp được dẫn xuất trực tiếp hoặc gián tiếp từ lớp đó.

Các lớp PseudoStep, DetailedStep được kế thừa từ lớp cha là Step, hoặc các lớp giải thuật được kế thừa lớp cha Algorithm, hoặc lớp DPState được kế thừa từ lớp cha là State trong thiết kế project là cách mà nhóm đã áp dụng tính chất này của hướng đối tượng.

```
public class DynamicProgramming extends Algorithm {

    private ArrayList<ArrayList<Float>> minDis = new ArrayList<>();
    private ArrayList<ArrayList<Float>> initMatrix = new ArrayList<>();
```

```
public class BellmanFord extends Algorithm {
    public static final int INFINITE = 1000;
    public void bellmanfordProgram(Graph g, int startID) {...}
    public String printPath(Graph g, Vertex v, Vertex start){...}
```

```
public class Dijkstra extends Algorithm {
    public static final float INFINITE = 1000;

    public void dijkstraProgram(Graph g, int startID){
        int count;
```

d. Tính đa hình (Polymorphism)

Tính đa hình có thể hiểu đơn giản là các đối tượng, phương thức giống nhau có thể có các hành vi khác nhau tùy thuộc tình huống hiện tại.

Trong thiết kế của project của nhóm, có thể lấy ví dụ về tính đa hình như sau: phương thức excuteAlgorithm() được ghi đè lại bởi các giải thuật lớp con (Dijkstra, Bellman,…) và tùy thuộc vào việc người dùng chạy giải thuật nào mà phương thức này sẽ được thực thi tương ứng với giải thuật đó.

Ngoài ra, các phương thức constructor trong các lớp của project cũng được nạp chồng lên nhau (cùng tên khác chữ kí), từ đó có thể khởi tạo các đối tượng thuộc cùng lớp nhưng có thuộc tính ban đầu khác nhau.

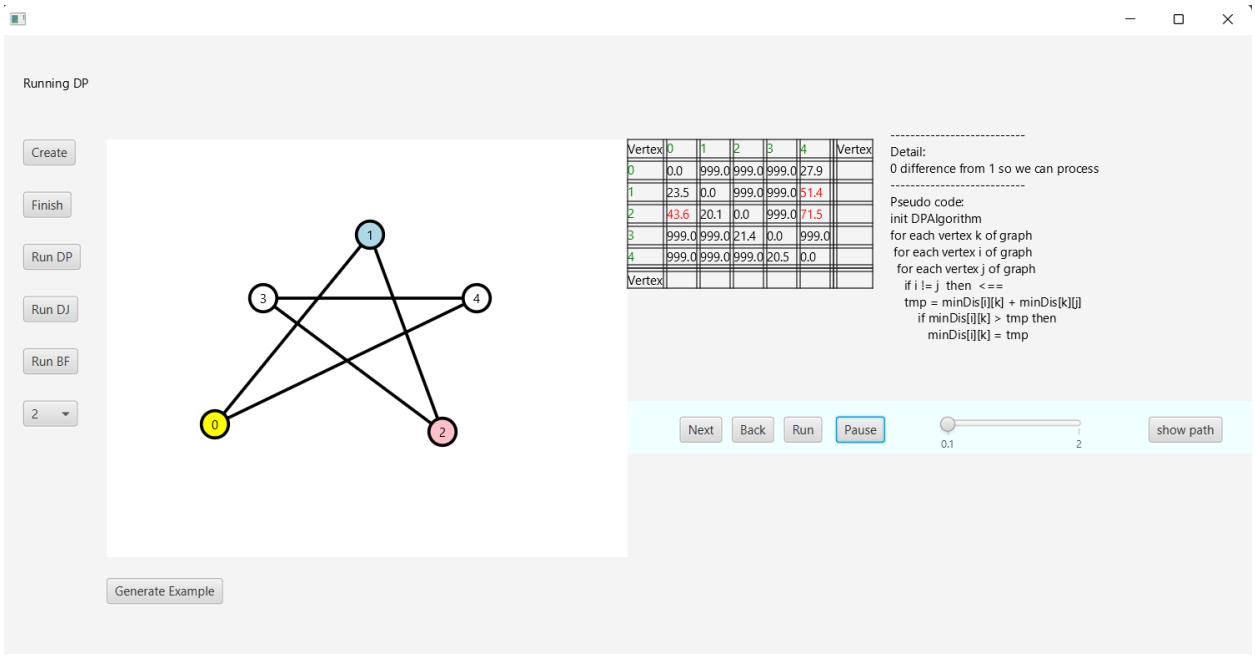
```java
public class BellmanFord extends Algorithm {
    public static final int INFINITE = 1000;
    public void bellmanfordProgram(Graph g, int startID) {...}
    public String printPath(Graph g, Vertex v, Vertex start){...}

    @Override
    public void executeAlgorithm(Graph g, int startID) {...}
```

III. Xây dựng chương trình JAVA

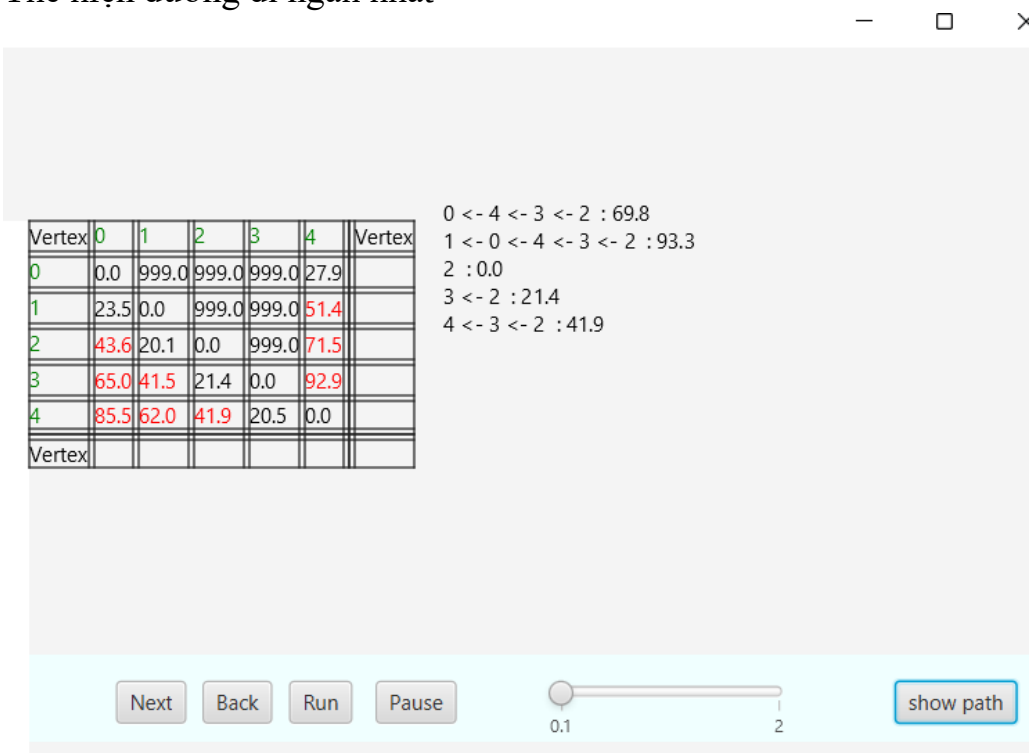
1.1. Hoàn thiện yêu cầu bài toán

a. Dynamic Programming

- Giải thuật được biểu diễn gồm mã giả, ma trận trọng số W và biểu diễn trên đồ thị

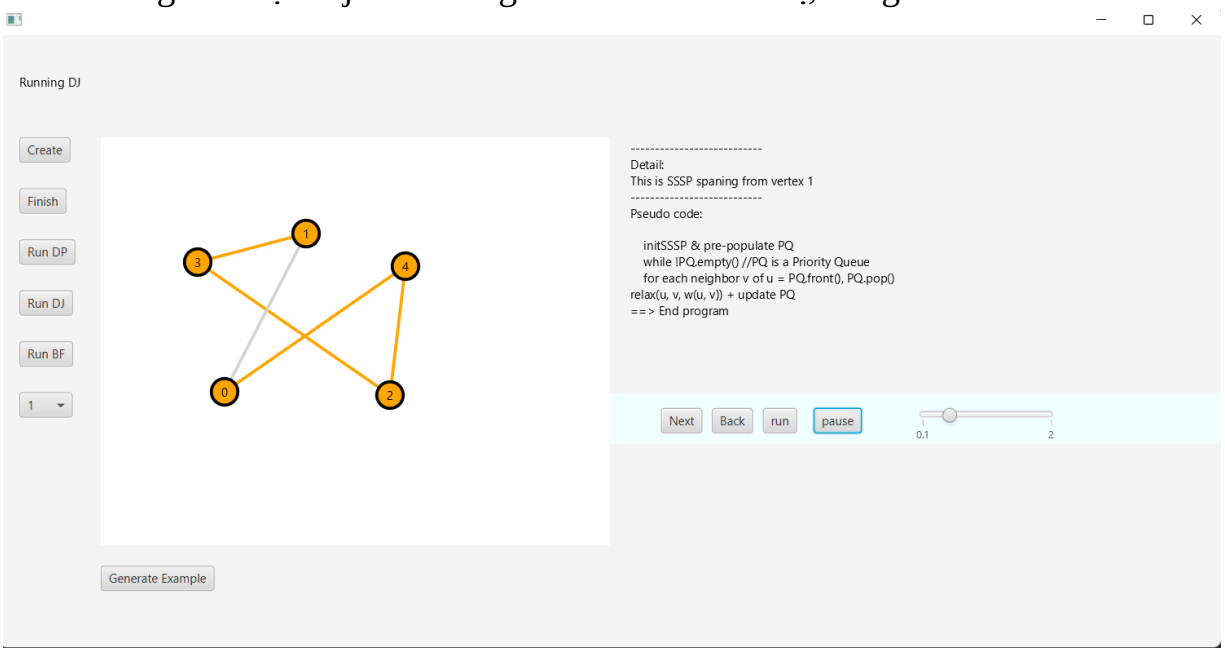


- Thể hiện đường đi ngắn nhất



b. Dijkstra

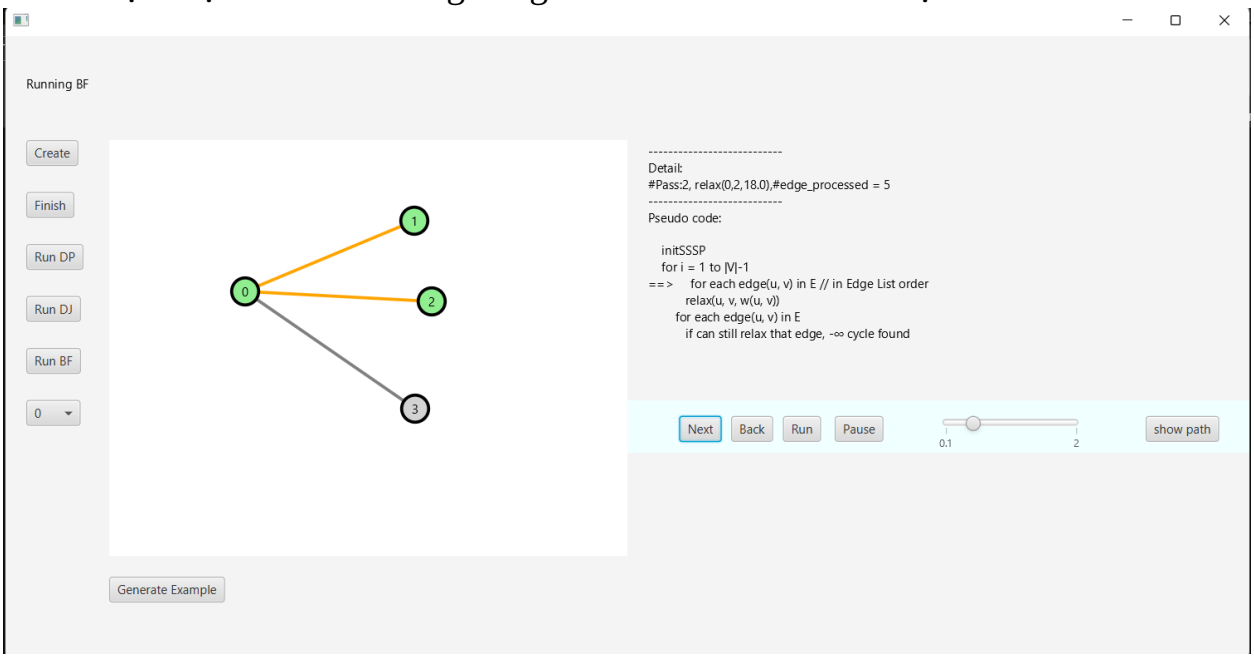
- Biểu diễn giải thuật Dijkstra bao gồm biểu diễn đồ thị, mã giả



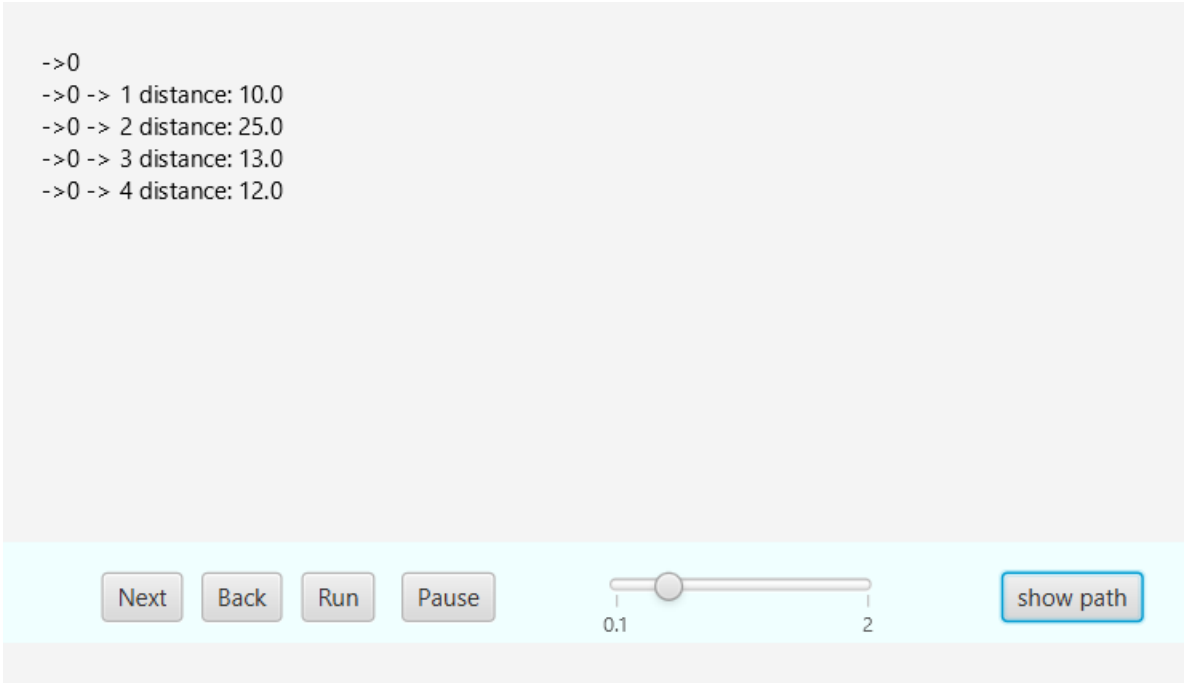
- Thể hiện đường đi ngắn nhất bao gồm biểu diễn trên đồ thị và độ dài đường đi

c. Bellman Ford

- Giải thuật được biểu diễn bằng mã giả và biểu diễn trên đồ thị

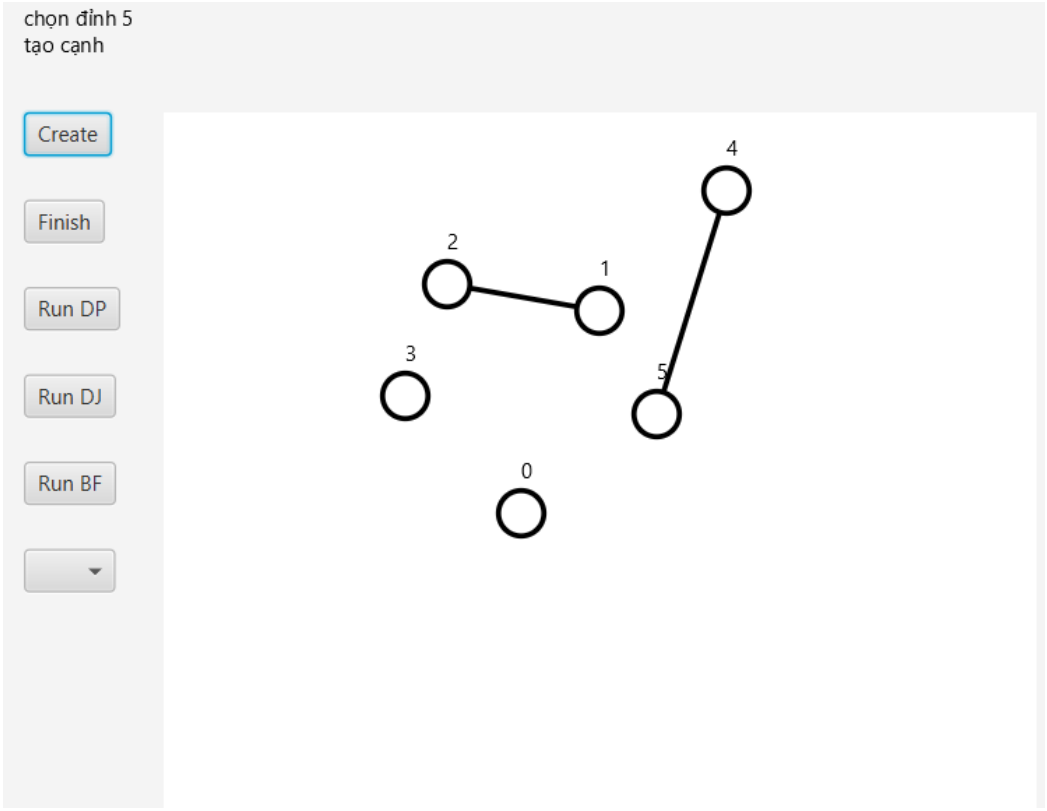


- Biểu diễn đường đi ngắn nhất

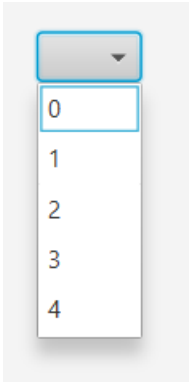


1.2. Các chức năng khác

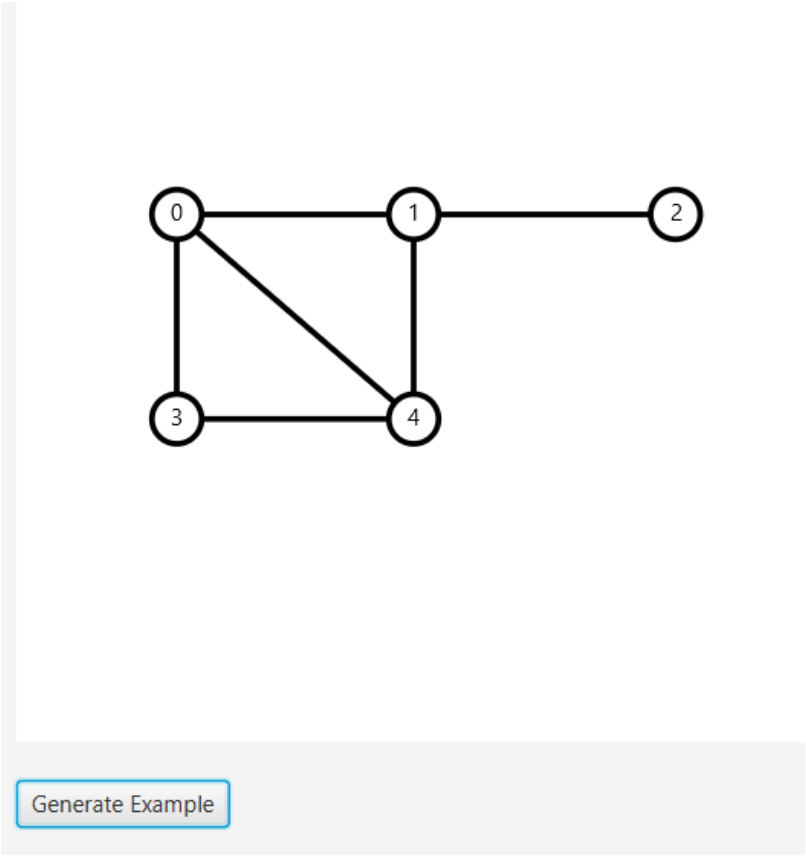
- Tạo đồ thị



- Chọn đỉnh



- Tạo tự động ví dụ



1.3. Một số lỗi tồn đọng

- Sau khi chạy xong thuật toán, hiện trạng đồ thị hiện tại giữ nguyên, không trở về ban đầu
- Chưa hiển thị được hướng cạnh