Rapport :

Jeandidier Clément S2A

Hary Alexandre S2A

un graphe est constitué de nœuds (sommets ou point) et d'arc.

Les arêtes relient les nœuds entre eux, ce qui constitue des graphes

Représentation d'un graphe :

classe Nœud :

Nous allons faire la classe Nœud.

Question 1 :

Dans cette question, nous avons créé la classe `"Nœud"` qui représente les nœuds d'un graphe.

Création de la classe Nœud avec un constructeur :

On a créé la classe dans laquelle on a mit les attributs "String nom" et " adj" en private.

Dans le constructeur on a mit un “String n” (qui représente le nom qu'on veut donner au sommet) en paramètre.

Ce constructeur initialise "nom" en lui donnant la valeur de n et met une `"ArrayList"` vide dans adj

```
import java.util.ArrayList;

public class Noeud {

    private String nom;

    private ArrayList<Arc> adj;

    public Noeud(String n){
        this.nom = n;
        this.adj = new ArrayList<Arc>();
    }
}
```

Question 2 :

Dans cette question, nous avons redéfini la méthode `"boolean equals(Object o)"` qui nous permet de savoir si deux nœuds sont égaux en fonction de leurs nom.

Dans cette fonction on entre un `"String o"` (qui représente le nom d'un Nœud) en paramètre, on va ensuite comparer "o" avec le nom du Nœud `"this"`.

```
public boolean equals(String o){
        return this.nom == o;
    }
```

Question 3 :

Dans cette question, nous avons écrit la méthode publique voidAjouterArc(String destination, double cout) qui ajoute un arc entre deux noeuds.

Cette méthode a pour but d'ajouter un arc avec un coût "cout" (mit en paramètre) entre le Nœud "this" et le nœud destination (mit en paramètre)

```
public void ajouterArc(String destination, double cout) {
    this.adj.add(new Arc(destination, cout));
}
```

classe Arc :

Nous allons faire la classe Arc.

Question 4 :

Dans cette question, nous avons créé la classe `"Arc"` qui représente les arcs du graphe

Cette classe `"Arc"` a pour attributs un `"String dest"` (qui représente la destination de l'arc) et un `"double cout"` (qui représente le coût de l'arc).

Le constructeur de la classe `"Arc"` prend en paramètre un `"String d"` (qui représente la destination) et un `"double d"` (qui représente le coût de l'arc).

Si le coût est inférieur ou égal à 0, on n'initialise pas l'arc et une exception se produit disant que le coût doit être positif, sinon on initialise `"dest"` en lui donnant la valeur de `"d"` (mise en paramètre) puis on initialise `"cout"` en lui donnant la valeur `"c"` (mise en paramètre).

```java
class Arc {
    private String dest;
    private double cout;

    public Arc(String d, double c) {
        if(c <= 0){
            throw new IllegalArgumentException("Le cout doit etre positif");
        }
        this.dest = d;
        this.cout = c;
    }
}
```

Interface graphe :

Nous allons faire l'interface graphe.

Question 5 :

Dans cette question, nous allons faire l'interface "Graphe" qui contient la méthodes `"public ArrayList<String> listeNoeud()"` (qui retourne tout les nœuds du graphe) et la méthode `"public ArrayList<Arc> suivants(String n)"` (qui retourne la liste des arcs qui partent du "Nœud n" passé en paramètre).

Dans cet interface, nous avons créé les deux méthodes "listeNoeuds()" et "suivants(String n)"

```java
import java.util.ArrayList;

public interface Graphe {

    public ArrayList<String> listeNoeuds();

    public ArrayList<Arc> suivants(String n);
}
```

Classe GrapheListe :

implémentation de l'interface :

nous allons utiliser l'interface Graphe en l'implémentant dans une classe GrapheListe

Question 6 :

Dans cette question, nous allons créer la classe GrapheListe

Dans cette classe, il y a un attribut `"ArrayList<String> ensNom"` (qui permet de stocker les arcs), un autre attribut `"ArrayList<Noeuds> ensNoeuds"` (qui permet de stocker les arcs), un constructeur sans paramètre, une méthode `"public void ajouterArc(String depart , String destination , double cout )"` (cette méthode permet d'ajouter des nœuds et des arcs à un objet `GrapheListe`)

Le constructeur initialise "ensNom" et "ensNoeuds" en leurs donnant pour valeur une nouvelle ArrayList vide.

la méthode "public void `ajouterArc(String depart (qui représente le noeud de départ)`, `String destination` (qui représente le noeud d'arrivée), `double cout` (qui représente le coût de l'arc))"
dans cette méthode, on fait d'abord une condition qui ajoute le noeud de départ si il n'existe pas dans ensNom, on fait ensuite une deuxième condition qui ajoute le noeud de destination si il n'existe pas)
Ensuite, on recherche le Nœud de départ dans ensNoeuds puis on lui ajoute un arc avec une destination et un coût.

```
import java.io.*;
import java.util.ArrayList;


public class GrapheListe implements Graphe {
    private ArrayList<String> ensNom;
    private ArrayList<Noeud> ensNoeuds;

    public GrapheListe() {
        this.ensNom = new ArrayList<>();
        this.ensNoeuds = new ArrayList<>();
    }

    public void ajouterArc(String depart, String destination, double cout) {
        if (!ensNom.contains(depart)) {
            ensNom.add(depart);
            ensNoeuds.add(new Noeud(depart));
        }
        if (!ensNom.contains(destination)) {
            ensNom.add(destination);
            ensNoeuds.add(new Noeud(destination));
        }
        for (int i = 0; i < ensNoeuds.size(); i++) {
            if (ensNoeuds.get(i).getNom().equals(depart)) {
                ensNoeuds.get(i).ajouterArc(destination, cout);
            }
        }
    }
}
```

Question 7 :

Dans cette question, nous allons créer un main qui va créer le graphe de la figure 1.

Dans ce Main, on a créé un `"GrapheListe gr"` dans lequel on a ajouté les nœuds et arcs (en utilisant la méthode `gr.ajouterArc(...))` de la figure 1.

```
import java.io.IOException;
import java.util.ArrayList;
```

```
public class Main {
    public static void main(String[] args) throws IOException {
        GrapheListe gr = new GrapheListe();
        gr.ajouterArc("A", "B", 12);
        gr.ajouterArc("A", "D", 87);
        gr.ajouterArc("B", "E", 11);
        gr.ajouterArc("C", "A", 19);
        gr.ajouterArc("D", "B", 23);
        gr.ajouterArc("D", "C", 10);
        gr.ajouterArc("E", "D", 43);
    }
}
```

Affichage d'un graphe :

Question 8 :

Dans cette question, nous avons créé une méthode toString.

On a décidé d'utiliser un `StringBuilder` qu'on a appelé sb.
On a ensuite fait une boucle for qui parcours tout les noeuds.
dans cette boucle, on a ajouté le nom du noeud à sb puis on a mit une autre boucle for pour parcourir tout les noeuds adjacents au permier noeud.
On a ensuite ajouté les noeuds adjacents avec le coût à sb.
Ensuite, on a fait un retour à la ligne puis pour finir, on return sb.

```
public String toString() {
    StringBuilder sb = new StringBuilder();
    for (Noeud noeud : ensNoeuds) {
        sb.append(noeud.getNom()).append(" -> ");
        for (Arc arc : noeud.getAdj()) {
            sb.append(arc.getDest()).append("
("").append((int)arc.getCout()).append(") ");
        }
        sb.append("\n");
    }
    return sb.toString();
}
```

Question 9 :

Dans cette question, nous avons créé une méthode `toGraphviz` qui a pour but de retourner un String qui permettra de construire un graphe sur `GraphvizOnline`.

Pour faire cette méthode, nous avons repris la méthode toString de la question 8 et nous l'avons modifié.
Dans cette méthode, nous avons créé un `Stringbuilder` nommé "sb" qui commencera par "digraph G {".
Ensuite, nous avons fait 2 boucles for imbriquées, la première parcours tout les noeuds du graphes et la seconde boucle parcours tout les arcs du noeuds.
Dans cette deuxième boucle, nous avons ajouté les nom du noeud, une flèche, le nom du noeud

de destination, puis "[label = ", puis nous avons ajouté le coût, et enfin, nous avons ajouté "]". Nous sommes sortis de la boucle for et nous avons fini le `StringBuilder` et lui ajoutant "}" et nous avons retourné sb.

```
public String toGraphviz() {
    StringBuilder sb = new StringBuilder("digraph G {\n");
    for (Noeud noeud : ensNoeuds) {

        for (Arc arc : noeud.getAdj()){
            sb.append(noeud.getNom()).append(" -> ");
            sb.append(arc.getDest()).append(" [label =
").append((int)arc.getCout()).append("]\n");
        }
    }
    sb.append("}");
    return sb.toString();
}
```

Question 10 :

Dans cette question, nous avons testé si ce que la méthode toGraphVis est correct.

Pour se faire, nous avons utilisé le main créé à la question 7 et nous avons ajouté "System.out.println(gr.toGraphviz());" après la création du graphe.

```
import java.io.IOException;
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) throws IOException {
        GrapheListe gr = new GrapheListe();
        gr.ajouterArc("A", "B", 12);
        gr.ajouterArc("A", "D", 87);
        gr.ajouterArc("B", "E", 11);
        gr.ajouterArc("C", "A", 19);
        gr.ajouterArc("D", "B", 23);
        gr.ajouterArc("D", "C", 10);
        gr.ajouterArc("E", "D", 43);

        System.out.println(gr.toGraphviz());
    }
}
```

On obtient un affichage comme ceci :

```
digraph G {
A -> B [label = 12]
A -> D [label = 87]
B -> E [label = 11]
D -> B [label = 23]
D -> C [label = 10]
E -> D [label = 43]
C -> A [label = 19]
}
```

Ensuite, nous avons copié l'affichage et nous l'avons collé sur le site "https://dreampuf.github.io/GraphvizOnline/".

Nous avons ensuite constaté que c'était le même graphe que sur la figure 2.

Validation du graphe :

Question 11 :

Dans cette question, nous avons créé des tests pour vérifier que le graphe se fasse correctement.

Nous avons créé 3 tests :

Un premier qui ajoute un arc dans un graphe vide, dans ce test, nous avons créé un nouveau `GrapheListe` vide dans lequel on a ajouté un arc de coût 10 entre les nœuds A et B. Enfin, nous avons vérifié que les nœuds A et B se sont bien ajoutés au graphe.

Le second test permet de vérifier si on arrive correctement a avoir accès au noeud suivant, nous avons créé un nouveau `grapheListe vide` auquel on a ajouté un arc de coût 15 entre A et B. Nous avons créé une `Arraylist "arcs"` qui récupère tout les arcs dont "A" est le début. On vérifie ensuite qu'il y a bien qu'un seul arc dans "arcs", on a ensuite vérifié si la destination était bien "B" et enfin que le coût était bien égal à 15.

Le troisième test nous permet de vérifier si on a correctement accès à la liste des nœuds. Pour faire ce test, nous avons créé un nouveau `GrapheListe` dans lequel on a ajouté trois arcs, il doit donc avoir les nœuds A, B, C et D dans le graphe, nous avons ensuite récupéré la liste des nœuds dans une `ArrayList "nœud"`, nous avons vérifié ensuite si "nœud" était de longueur 4 et enfin si il y avait bien les nœuds A, B, C et D dans la liste.

```
import org.junit.jupiter.api.Test;
import java.util.ArrayList;
import static org.junit.jupiter.api.Assertions.*;

public class GrapheTest {

    @Test
    public void testAjouterArc() {
        GrapheListe gr = new GrapheListe();
        gr.ajouterArc("A", "B", 10);
        assertTrue(gr.listeNoeuds().contains("A"));
        assertTrue(gr.listeNoeuds().contains("B"));
    }

    @Test
    public void testSuivants() {
        GrapheListe gr = new GrapheListe();
```

```
        gr.ajouterArc("A", "B", 15);
        ArrayList<Arc> arcs = gr.suivants("A");
        assertEquals(1, arcs.size());
        assertEquals("B", arcs.get(0).getDest());
        assertEquals(15, arcs.get(0).getCout());
    }

    @Test
    public void testListeNoeuds() {
        GrapheListe gr = new GrapheListe();
        gr.ajouterArc("A", "B", 12);
        gr.ajouterArc("B", "C", 23);
        gr.ajouterArc("C", "D", 34);
        ArrayList<String> noeud = gr.listeNoeuds();
        assertEquals(4, noeud.size());
        assertTrue(noeud.contains("A"));
        assertTrue(noeud.contains("B"));
        assertTrue(noeud.contains("C"));
        assertTrue(noeud.contains("D"));
    }
}
```

Chargement de Graphe :

Question 12 :

Dans cette question, nous avons créé un deuxième constructeur (qui prend un nom de fichier en paramètre) dans la classe GrapheListe

Ce deuxième constructeur (avec un "String" en paramètre) de la classe `Grapheliste` prends un `"String nomFichier"` (qui représente le nom d'un fichier qui contient du texte représentant un graphe sous forme de liste d'arcs) en paramètre.
On commence par initialiser `"ensNom"` et `"ensNoeuds"` en leurs donnant pour valeur une nouvelle `ArrayList` vide.
On parcours le fichier mit en paramètre ligne par ligne en utilisant un readline. nous avons ensuite mit le départ, la destination et le coût de la ligne dans un tableau de String en séparant le début, la destination et la coût.

On vérifie ensuite si la destination et le cout sont déjà dans le graphe, si il n'y sont pas, on les ajoute à ensNoeud.

Enfin on ajoute l'arc en utilisant la méthode `ajouterArc()`.

```
public GrapheListe(String nomFichier) {
    this.ensNom = new ArrayList<>();
    this.ensNoeuds = new ArrayList<>();

    try (BufferedReader br = new BufferedReader(new FileReader(nomFichier))) {
        String ligne = br.readLine();
        while (ligne != null) {
            String[] partie = ligne.split("\t");
            String noeud1 = partie[0];
            String noeud2 = partie[1];
```

```
            double cost = Double.parseDouble(partie[2]);

            if (!this.ensNom.contains(noeud1)) {
                this.ensNom.add(noeud1);
                this.ensNoeuds.add(new Noeud(noeud1));
            }
            if (!this.ensNom.contains(noeud2)) {
                this.ensNom.add(noeud2);
                this.ensNoeuds.add(new Noeud(noeud2));
            }
            int index = this.ensNom.indexOf(noeud1);
            this.ensNoeuds.get(index).ajouterArc(noeud2, cost);
            ligne = br.readLine();
        }
    } catch(FileNotFoundException e) {
        System.out.println("Fichier introuvable.");
    } catch (IOException e) {
        System.out.println("Erreur dans la lecture du fichier.");
    }

}
```

Question 13 :

Dans cette question, nous avons créé une méthode `fichierMatrice` qui prends en paramètre un `"String matriceFichier"` (qui représente le fichier contenant le descriptif d'un graphe sous forme de matrice d'adjacence) et un `"String arcFichier"` (qui représente le fichier où on va écrire la liste d'arcs).

Pour faire cette méthode, nous avons fait un `"BufferedReader"` qui va lire la matrice et un `"BufferedWriter"` qui va écrire la liste d'arcs.
On utilise un tableau de String qui va contenir le nœud de départ, le nœud de destination et le coût.
Enfin on utilise le buffered writer pour écrire l'arc.

```
public static void fichierMatrice(String matriceFichier, String arcFichier) {
    try (BufferedReader br = new BufferedReader(new FileReader(matriceFichier));
         BufferedWriter bw = new BufferedWriter(new FileWriter(arcFichier))) {

        // Lire la première ligne du fichier qui contient les noms des noeuds.
        String[] noeuds = br.readLine().split("\\s+");

        String ligne;
        while ((ligne = br.readLine()) != null) {
            // diviser la ligne en un tableau de chaînes.
            String[] partie = ligne.split("\\s+");

            // L'indice du noeud source dans le tableau noeuds.
            int index = -1;
            for (int i = 0; i < noeuds.length; i++) {
                if (noeuds[i].equals(partie[0])) {
                    index = i;
```

```
                    break;
                }
            }
            // Écrire les arcs dans le fichier de sortie.
            for (int i = 1; i < partie.length; i++) {
                // Si le coût n'est pas 0, écrire l'arc dans le fichier de
sortie.
                double cout = Double.parseDouble(partie[i]);
                if (cout != 0) {
                    bw.write(noeuds[index] + " " + noeuds[i] + " " + cout +
"\n");
                }
            }
        }
    } catch (FileNotFoundException e) {
        System.out.println("Fichier non trouvé");
    } catch (IOException e) {
        System.out.println("Erreur lors de la lecture/écriture du fichier: ");
    }
}
```

Calcul du plus court chemin par point fixe :

Algorithme du point fixe :

Question 14 :

Dans cette question, nous avons fait un algorithme qui trouve le chemin le plus court dans un graphe passé en paramètre en partant du noeud passé en paramètre.

Nous avons d'abord parcouru tout les noeuds en initialisant la valeur du noeud de départ à `0` et la valeur des autres noeuds à `+∞`.
Ensuite, il parcours chaque noeud non visité avec la distance minimale tant qu'il y en a et met à jour la distance, si une distance plus courte est trouvée, l'algorithme marque ce noeud comme visité.
Les valeurs finales "distance" et "parent" de chaque noeud correspond au chemin le plus court.

```
Fonction pointFixe(Graphe g InOut, Noeud depart)
    // Initialisation
    Pour chaque noeud v dans g.listeNoeuds() faire
        Si v == depart alors
            v.distance ← 0
        Sinon
            v.distance ← +∞
        Fin Si
        v.parent ← null
    Fin Pour

    Tant que il y a des nœuds non visités faire
        // Trouver le noeud non visité avec la distance minimale
        Noeud min ← null
        Pour chaque nœud v dans g.listeNoeuds() faire
            Si v non visité et (min est null ou v.distance < min.distance) alors
                min ← v
```

```
            Fin Si
        Fin Pour

        Si min est null alors
            Sortir de la boucle
        Fin Si

        // Mettre à jour les distances des noeud adjacents
        Pour chaque arc a dans g.suivants(mine) faire
            Noeud voisin ← a.dest
            Double nouveauDistance ← min.distance + a.cout
            Si nouveauDistance < voisin.distance alors
                voisin.distance ← nouveauDistance
                voisin.parent ← min
            Fin Si
        Fin Pour
        min.visité ← vrai
    Fin Tant que
Fin
```

Classe Valeur :

Programmation de l'algorithme du point fixe :

Question 15 :

Dans cette question, nous allons faire une classe Bellman-Ford dans laquelle on va écrire une méthode `"Valeur resoudre( Graphe g` (qui représente le graphe qu'on va utiliser dans le méthode)`, String depart(qui représente le nœud de départ))"`.

Dans cette méthode, on initialise la distance de tout les noeuds à +∞ sauf pour le noeud de départ qu'on initialise à 0.
Ensuite, tant qu'il y a des changements, on parcours tous les noeuds et leurs arcs puis on met à jour leurs distances et les parents si une distance plus courte est trouvée.
Si il n'y a eu aucun changement alors les distances et les parents sont bons.
Pour finir on retourne la valeur.

```java
public Valeur resoudre(Graphe g, String depart) {

    //Initialisation des valeurs
    Valeur val = new Valeur();
    for (Noeud n : g.getNoeuds()) {
        val.setValeur(n.getNom(), Double.POSITIVE_INFINITY);
    }
    val.setValeur(depart, 0.0);

    boolean changer = true;
    while (changer) {
        changer = false;
        for (Noeud n : g.getNoeuds()) {
            for (Arc arc : n.getAdj()) {
                String destination = arc.getDest();
                double cout = arc.getCout();
```

```
                if (val.getValeur(n.getNom()) + cout <
val.getValeur(destination)) {
                    val.setValeur(destination, val.getValeur(n.getNom()) +
cout);
                    val.setParent(destination, n.getNom());
                    changer = true;
                }
            }
        }
    }
    return val;
}
```

Test de l'algorithme du point fixe :

Question 16 :

Dans cette question, nous allons créer un Main qu'on appellera `MainBellmanFord`. Dans ce main, nous allons afficher chaque valeur de chaque noeud pour vérifier qu'elles correspondent aux valeurs calculées

Dans cette classe, nous commençons par créer un `"GrapheListe"`.
Ensuite on crée une instance `"BellmanFord"`.

Ensuite, on utilise la méthode résoudre faite à le question précédente en mettant en paramètre le `"GrapheListe"` et un `String` de valeur "A".

Enfin, on affiche le nom, la distance et les parents de chaque noeud du graphe puis on peut vérifiersi les valeurs correspondent.

```
public class MainBellmanFord {
    public static void main(String[] args) {

        //Creation d'un graphe à partir d'un fichier
        GrapheListe g = new GrapheListe("Graphes/GrapheExemple.txt");

        // Créer une instance de BellmanFord
        BellmanFord bf = new BellmanFord();

        // Résoudre le graphe en partant du noeud 1
        Valeur resultat = bf.resoudre(g, "A");

        // Afficher les valeurs pour chaque noeud
        for (Noeud n : g.getNoeuds()) {
            System.out.println("Noeud: " + n.getNom());
            System.out.println("Distance: " + resultat.getValeur(n.getNom()));
            System.out.println("Parent: " + resultat.getParent(n.getNom()));
            System.out.println("----------------");
        }

    }
}
```

Voici un exemple d'affichage :

```
Noeud: D
Parent: E
Distance: 66.0
----------------
Noeud: C
Parent: D
Distance: 76.0
----------------
Noeud: A
Parent: null
Distance: 0.0
----------------
Noeud: B
Parent: A
Distance: 12.0
----------------
Noeud: E
Parent: B
Distance: 23.0
----------------
```

Question 17 :

Dans cette question, nous allons écrire un test unitaite pour vérifier que l'algorithme du point fixe est correct et que les parents sont bien calculés.

Dans ce main, nous commençons par créer un `GrapheListe gr`.

On ajoute ensuite des noeuds (1, 2, 3, 4) au graphe.

Ensuite, on crée l'instance `"BellmanFord bf"`.

on résouds le graphe en partant du noeud 1.

Enfin on fait des assertEquals pour vérifier la valeur et les parents de chaque noeud.

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class BellmanFordTest {
    @Test
    public void testResoudre() {
        // Créer un graphe
        GrapheListe gr = new GrapheListe();


        // Ajouter des noeuds et des arcs au graphe
        gr.ajouterArc("1","2",5);
        gr.ajouterArc("2","3",6);
        gr.ajouterArc("3","4",7);
        gr.ajouterArc("4","1",8);

        // Créer une instance de BellmanFord
        BellmanFord bf = new BellmanFord();

        // Résoudre le graphe en partant du noeud 1
```

```
        Valeur resultat = bf.resoudre(gr, "1");

        // Vérifier les valeurs pour chaque noeud
        assertEquals(0, resultat.getValeur("1"));
        assertEquals(5, resultat.getValeur("2"));
        assertEquals(11, resultat.getValeur("3"));
        assertEquals(18, resultat.getValeur("4"));

        // Vérifier les parents pour chaque noeud
        assertEquals(null, resultat.getParent("1"));
        assertEquals("1", resultat.getParent("2"));
        assertEquals("2", resultat.getParent("3"));
        assertEquals("3", resultat.getParent("4"));
    }
}
```

Calcul du meilleur chemin :

Question 18 :

Dans cette question, nous avons créé la méthode `"List<String> calculerChemin(String destination)"`.
Cette méthode doit renvoyer un chemin vers le noeud destination donné en paramètre.

Dans cette méthode, nous commençons par créer une "ArrayList chemin" qui sera vide.
Ensuite, on initialise un nouveau "String noeud" avec destination (donné en paramètre).
Ensuite, on entre dans une boucle while qui se fait tant que "noeud" n'est pas null.
On va ajouter le "String noeud" au début de "chemin" puis on prends le parent de "noeud".
une fois que "noeud" n'a plus de parent, on sort de la boucle.

Enfin on retourne l'ArrayList chemin.

```
public ArrayList<String> calculerChemin(String destination) {
    ArrayList<String> chemin = new ArrayList<>();

    // Commencez par la destination
    String noeud = destination;

    while (noeud != null) {
        // Ajoutez le noeud  au début du chemin
        chemin.add(0, noeud);

        // Allez au parent du noeud
        noeud = getParent(noeud);
    }

    // Le chemin est maintenant du noeud de départ à la destination
    return chemin;
}
```

Calcul du meilleur chemin par Dijkstra :

Principe de l’algorithme de Dijkstra :

Algorithme de Dijkstra :

Question 19 :

Dans cette question, nous avons recopié l'algorithme donné puis nous l'avons traduit en java afin d'écrire la méthode Valeur qui prends en paramètre un `"GrapheListe graphe"` le nom du nœud de départ `"String depart"`.

Dans cette méthode, on commence par initialiser une Valeur v (qui sera vide pour l'instant). Ensuite, on initialise La liste de nœuds à traiter `"ArrayList<Nœud> noeudsATraiter"` (qui aura pour valeur `ensNoeuds` du graphe mit en paramètre). On utilise aussi un `HashMap<>()` qui permet d'optimiser un maximum le temps de recherche d'un élément dans une liste..

Ensuite, on initialise la distance de chaque noeud à +∞ sauf pour son parent à qui on donne la valeur null et celle du noeud de départ à 0.

Ensuite, on entre dans une nouvelle boucle while qui se finit lors que noeudsATraiter est vide. Dans cette boucle, on va trouver le Nœud avec la distance minimale.
ensuite, on enlève le noeud de noeudsATraiter.

Ensuite, dans la boucle for, on vérifie si e noeud destination v est encore dans "noeudsATraiter". Si il est encore dedans, on calcule la nouvelle distance. Si elle est inférieure, on met la distance à jour.

On sort de la boucle for et while puis on retourne "v" qui contient les distances et les parents des noeuds.

```
import java.util.*;

public class Dijkstra implements Algorithme {

    public Valeur resoudre(Graphe g, String depart) {
        Valeur v = new Valeur();
        ArrayList<Noeud> noeudsATraiter = new ArrayList<>(g.getNoeuds());
        Map<String, Noeud> noeuds = new HashMap<>();

        // Initialisation des valeurs et des parents pour chaque nœud
        for (Noeud noeud : noeudsATraiter) {
            v.setValeur(noeud.getNom(), Double.POSITIVE_INFINITY);
            v.setParent(noeud.getNom(), null);
            noeuds.put(noeud.getNom(), noeud);
        }

        // La distance du nœud de départ est initialisée à zéro
        v.setValeur(depart, 0);

        // Tri de la liste des nœuds à traiter en fonction de leur distance
        noeudsATraiter.sort(Comparator.comparing(n -> v.getValeur(n.getNom())));

        while (!noeudsATraiter.isEmpty()) {
            // Le nœud avec la plus petite distance est toujours en tête de la
liste après le tri
            Noeud u = noeudsATraiter.remove(0);
```

```
            // Parcours des arcs sortants du nœud u
            for (Arc arc : u.getAdj()) {
                if (noeuds.containsKey(arc.getDest())) {
                    // Calcul de la distance jusqu'au nœud de destination à
partir du nœud u
                    double d = v.getValeur(u.getNom()) + arc.getCout();
                    if (d < v.getValeur(arc.getDest())) {
                        // Mise à jour de la distance et du parent du nœud de
destination
                        v.setValeur(arc.getDest(), d);
                        v.setParent(arc.getDest(), u.getNom());

                        // Tri de la liste des nœuds à traiter après la mise à
jour de la distance
                        noeudsATraiter.sort(Comparator.comparing(n ->
v.getValeur(n.getNom())));
                    }
                }
            }

            // Suppression du nœud u de la liste des nœuds à traiter
            noeuds.remove(u.getNom());
        }

        return v;
    }
}
```

Validation Dijkstra :

Question 20 :

Dans cette question, nous allons écrire des tests pour vérifier que la méthode résoudre de la classe Dijkstra fonctionne bien.

Nous avons commencé par créer un "Grapheliste grapheTest".

Nous avons ensuite testé l'algorithme dijkstra en vérifiant que le chemin de A à B était le bon, on a fait la même chose pour le chemin de A à D et pour celui de E à B.

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

import java.util.ArrayList;

public class DijkstraTest {

    @Test
    public void testDijkstra() {

        // Création d'un graphe de test
        GrapheListe grapheTest = new GrapheListe("Graphes/GrapheExemple.txt");

        // Test de l'algorithme Dijkstra
```

```
        Dijkstra dijkstra = new Dijkstra();
        Valeur valeur = dijkstra.resoudre(grapheTest, "A");

        ArrayList<String> chemin = valeur.calculerChemin("B");

        // Vérification que le chemin le plus court de A à B est celui attendu
        ArrayList<String> cheminAttendu = new ArrayList<>();
        cheminAttendu.add("A");
        cheminAttendu.add("B");
        assertEquals(cheminAttendu, chemin);

        // Teste le chemin le plus court de A à D.
        chemin = valeur.calculerChemin("D");

        // Vérification que le chemin le plus court de A à D est celui attendu
        cheminAttendu = new ArrayList<>();
        cheminAttendu.add("A");
        cheminAttendu.add("B");
        cheminAttendu.add("E");
        cheminAttendu.add("D");
        assertEquals(cheminAttendu, chemin);

        // Teste le chemin le plus court de E à B.
        valeur = dijkstra.resoudre(grapheTest, "E");
        chemin = valeur.calculerChemin("B");

        // Vérification que le chemin le plus court de E à B est celui attendu
        cheminAttendu = new ArrayList<>();
        cheminAttendu.add("E");
        cheminAttendu.add("D");
        cheminAttendu.add("B");
        assertEquals(cheminAttendu, chemin);
    }
}
```

Programme principal :

Question 21 :

Dans cette question, nous allons créer un Main qu'on va appeler `MainDjikstra` qui permet de lire les graphes à partir de fichiers texte, faire le calcul des chemins les plus court pour des noeuds donnés et enfin l'affichage des chemins pour des noeuds donnés.

Dans ce main, nous commençons par initialiser le scanner, on demande ensuite à l'utilisateur d'entrer un nom de fichier puis un noeud de départ grâce au scanner.

Ensuite, on calcule les chemins les plus courts en utilisant la méthode résoudre de la classe Dijkstra en mettant le graphe et le noeud de départ donné par l'utilisateur en paramètre.

Ensuite, grâce au scanner on demande à l'utilisateur d'entrer un noeud de destination.

On calcule le chemin le plus court en utilisant la méthode calculerChemin.

Enfin on affiche la valeur du chemin le plus court du départ jusqu'à la destination puis on ferme le scanner.

```java
import java.util.ArrayList;
import java.util.Scanner;

public class MainDijkstra {

    public static void main(String[] args) {

        // Initialisation du scanner pour lire l'input utilisateur
        Scanner scanner = new Scanner(System.in);

        // Demande à l'utilisateur le nom du fichier
        System.out.print("Veuillez entrer le nom du fichier: ");
        String nomFichier = scanner.nextLine();

        // Création du graphe à partir du fichier
        GrapheListe graphe = new GrapheListe(nomFichier);

        // Demande à l'utilisateur le noeud de départ
        System.out.print("Veuillez entrer le nom du noeud de départ: ");
        String depart = scanner.nextLine();

        // Calcul des chemins les plus courts à partir du noeud de départ
        Dijkstra dijkstra = new Dijkstra();
        Valeur valeur = dijkstra.resoudre(graphe, depart);

        // Demande à l'utilisateur le noeud de destination
        System.out.print("Veuillez entrer le nom du noeud de destination: ");
        String destination = scanner.nextLine();

        // Calcul et affichage du chemin le plus court
        ArrayList<String> chemin = valeur.calculerChemin(destination);
        System.out.println("Le chemin le plus court de " + depart + " à " +
destination + " est : " + chemin);

        // Fermeture du scanner
        scanner.close();
    }
}
```

Validation et expérimentation :

Comportement qualitatif des algorithmes :

Question 22 et 23:

On remarque l'algorithme de Dijkstra fonctionne en construisant un ensemble de nœuds dont le plus court chemin depuis le nœud de départ a déjà été déterminé alors que l'algorithme de Bellman-Ford fonctionne en relaxant progressivement toutes les arêtes du graphe et en mettant à jour les chemins les plus courts.

De plus l'algorithme de Dijkstra pour chaque itération, il sélectionne le nœud non visité ayant la distance minimale, calcule la distance de ce nœud à tous ses voisins non visités et met à jour les chemins si nécessaire. Il continue jusqu'à ce que tous les nœuds soient visités. Tandis que l'algorithme de Bellman-Ford pour chaque itération de l'algorithme, il traverse toutes les arêtes

du graphe et tente de mettre à jour le chemin le plus court entre les paires de nœuds. Il le fait pour chaque nœud, N fois, où N est le nombre de nœuds.

Graphes de tailles différentes :

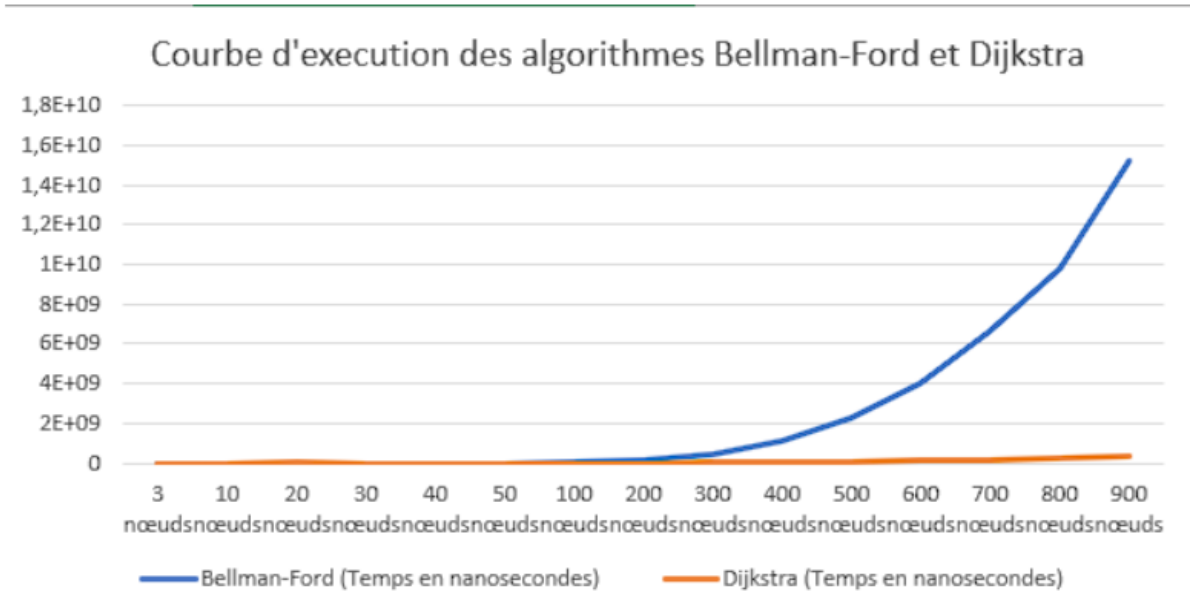
Question24 :

Pour comparer les deux algorithmes nous avons mise en place des "chronomètres" qui vont calculer le temps exécution d'un algorithme en nanosecondes. Nous avons réalisé ces tests dans une classe `MainComparaisonAlgorithme` sur un panel large de graphes plus ou moins grand avec les graphes qui été déjà fournie. Nous avons pu tester le temps grâce à la méthode System.nanoTime().

```
import java.util.ArrayList;

public class MainComparaisonAlgorithme {
    public static void main(String[] args) {
    // Création des graphes avec un fichier
        // Création d'un graphe avec 3 noeuds
        GrapheListe graphe1 = new GrapheListe("Graphes/Graphe5.txt");

        // Création d'un graphe avec 10 noeuds
        GrapheListe graphe2 = new GrapheListe("Graphes/Graphe15.txt");

        // Création d'un graphe avec 20 noeuds
        GrapheListe graphe3 = new GrapheListe("Graphes/Graphe25.txt");

        // Création d'un graphe avec 30 noeuds
        GrapheListe graphe4 = new GrapheListe("Graphes/Graphe35.txt");

        // Création d'un graphe avec 40 noeuds
        GrapheListe graphe5 = new GrapheListe("Graphes/Graphe45.txt");

        // Création d'un graphe avec 50 noeuds
        GrapheListe graphe6 = new GrapheListe("Graphes/Graphe55.txt");

        // Création d'un graphe avec 100 noeuds
        GrapheListe graphe7 = new GrapheListe("Graphes/Graphe105.txt");

        // Création d'un graphe avec 200 noeuds
        GrapheListe graphe8 = new GrapheListe("Graphes/Graphe205.txt");

        // Création d'un graphe avec 300 noeuds
        GrapheListe graphe9 = new GrapheListe("Graphes/Graphe305.txt");

        // Création d'un graphe avec 400 noeuds
        GrapheListe graphe10 = new GrapheListe("Graphes/Graphe405.txt");

        // Création d'un graphe avec 500 noeuds
        GrapheListe graphe11 = new GrapheListe("Graphes/Graphe505.txt");

        // Création d'un graphe avec 600 noeuds
        GrapheListe graphe12 = new GrapheListe("Graphes/Graphe605.txt");

        // Création d'un graphe avec 700 noeuds
```

```
        GrapheListe graphe13 = new GrapheListe("Graphes/Graphe705.txt");

        // Création d'un graphe avec 800 noeuds
        GrapheListe graphe14 = new GrapheListe("Graphes/Graphe805.txt");

        // Création d'un graphe avec 900 noeuds
        GrapheListe graphe15 = new GrapheListe("Graphes/Graphe905.txt");

        //Initialisation de la valeur de départ
        String depart = "1";

        //Mise en place des graphes dans un tableau
        GrapheListe[] tabGraphe =
{graphe1,graphe2,graphe3,graphe4,graphe5,graphe6,graphe7,graphe8,graphe9,graphe1
0,graphe11,graphe12,graphe13,graphe14,graphe15};
        String[] tabNbNoeud =
{"3","10","20","30","40","50","100","200","300","400","500","600","700","800","9
00"};
        //Mise en place des variables de temps
        long debut;
        long fin;
        //Boucle d'affichage des temps d'execution
        for (int i = 0; i < tabGraphe.length; i++) {
            System.out.println("<<< Graphe de " + tabNbNoeud[i] + " noeuds
>>>");
            BellmanFord bf = new BellmanFord();
            //Mesure du temps d'execution de l'algorithme de BellmanFord sur les
différents graphes
            debut = System.nanoTime();
            Valeur valeurBellmanFord = bf.resoudre(tabGraphe[i], depart);
            fin = System.nanoTime();
            long tempsBellmanFord = fin - debut;
            System.out.println("Temps d'exécution (BellmanFord) : " +
tempsBellmanFord + " ns");

            Dijkstra dj = new Dijkstra();
            //Mesure du temps d'execution de l'algorithme de Dijkstra sur les
différents graphes
            debut = System.nanoTime();
            Valeur valeurDijkstra = dj.resoudre(tabGraphe[i], depart);
            fin = System.nanoTime();
            long tempsDijkstra = fin - debut;
            System.out.println("Temps d'exécution (Dijkstra) : " + tempsDijkstra
+ " ns");
            System.out.println(' ');
        }
    }
}
```

Graphe	Bellman-Ford (Temps en nanosecondes)	Dijkstra (Temps en nanosecondes)
3 nœuds	500400	1400300
10 nœuds	292700	233900
20 nœuds	763800	20565100
30 nœuds	1534900	673700
40 nœuds	3071900	1157400
50 nœuds	5573900	1307300
100 nœuds	19055700	3685900
200 nœuds	144588800	14700500
300 nœuds	480150000	30508200
400 nœuds	1162604500	60806500
500 nœuds	2283109500	80638100
600 nœuds	4034476900	138384700
700 nœuds	6591945500	147623200
800 nœuds	9814742000	213024000
900 nœuds	15241131000	335768200



On remarque que sur la courbe exécution, le temps d'exécution de l'algorithme de Bellman-Ford augmente drastiquement en comparaison avec l'algorithme de Dijkstra. Cependant on peut remarque sur le tableaux, que parfois pour des plus petit graphes, le temps exécution de Bellman-Ford et plus rapide que celui de Dijkstra. On peut en conclure que dans l'ensemble, l'utilisation de l'algorithme de Dijkstra et à privilégié pour des graphes de moyennes et grandes tailles car il est extrêmement plus rapide. Cependant l'algorithme de Bellman-Ford reste tout de même efficace mais uniquement pour des petits graphes.

Génération de graphes :

Question 25 :

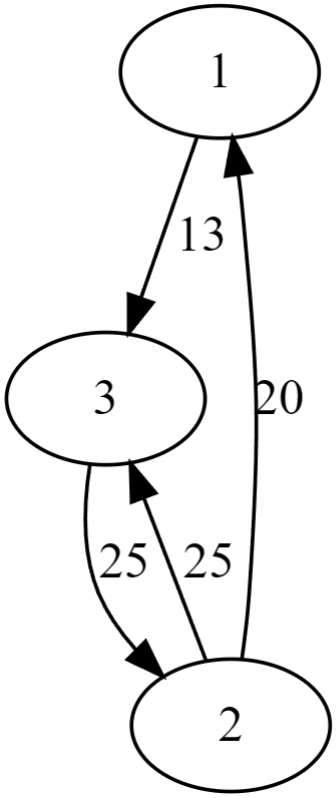
La méthode `genererGraphe` génère un graphe aléatoire avec une taille spécifiée, en utilisant un chemin du nœud de départ connecté à tous les autres nœuds. Voici ce que le programme fait étape par étape :

```java
public static GrapheListe genererGraphe(int taille, String depart, String
arrivee) {
        // Création d'un graphe vide
        GrapheListe graphe = new GrapheListe();

        // Création d'une variable aléatoire
        Random random = new Random();

        // Création de l'arc entre le noeud de départ et celui d'arrivée avec un
coût aléatoire
        double cout = 1 + random.nextInt(25);
        graphe.ajouterArc(depart, arrivee, cout);
        if (random.nextBoolean()) {
            graphe.ajouterArc(arrivee, depart, cout); // Arc dans la direction
opposée
        }

        // Ajout des nœuds restants
        for (int i = 2; i < taille; i++) {
            String nouveauNoeud = String.valueOf(i);

            // Assurer qu'au moins un noeud existant pointe vers le nouveau
noeud
            String noeudExistant =
graphe.getNoeuds().get(random.nextInt(graphe.getNoeuds().size())).getNom();
            cout = 1 + random.nextInt(25);
            graphe.ajouterArc(noeudExistant, nouveauNoeud, cout);
            if (random.nextBoolean()) {
                graphe.ajouterArc(nouveauNoeud, noeudExistant, cout); // Arc
dans la direction opposée
            }

            // Assurer qu'au moins le nouveau noeud pointe vers un noeud
existant
            noeudExistant =
graphe.getNoeuds().get(random.nextInt(graphe.getNoeuds().size())).getNom();
            while (noeudExistant.equals(nouveauNoeud)) {
                noeudExistant =
graphe.getNoeuds().get(random.nextInt(graphe.getNoeuds().size())).getNom();
            }
            cout = 1 + random.nextInt(25);
            graphe.ajouterArc(nouveauNoeud, noeudExistant, cout);
            if (random.nextBoolean()) {
                graphe.ajouterArc(noeudExistant, nouveauNoeud, cout); // Arc
dans la direction opposée
            }
        }

        return graphe;
    }
```

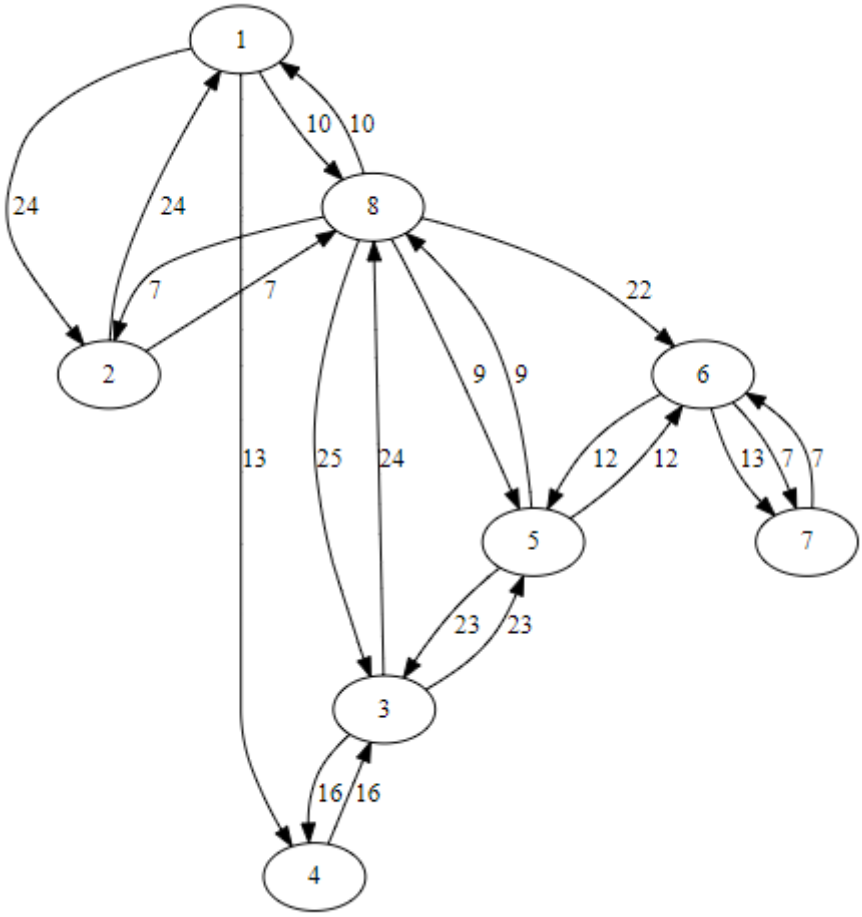
Question 26 :

```
public class Main {
    public static void main(String[] args) throws IOException {


        // Génération d'un graphe aléatoire avec 3 nœuds
        System.out.println(GrapheListe.genererGraphe(3, "1", "3").toGraphviz());
        System.out.println("--------------------");

        // Génération d'un graphe aléatoire avec 8 nœuds
        System.out.println(GrapheListe.genererGraphe(8, "1", "8").toGraphviz());
        System.out.println("--------------------");

        // Génération d'un graphe aléatoire avec 15 nœuds
        System.out.println(GrapheListe.genererGraphe(15, "1",
"15").toGraphviz());


    }
}
```

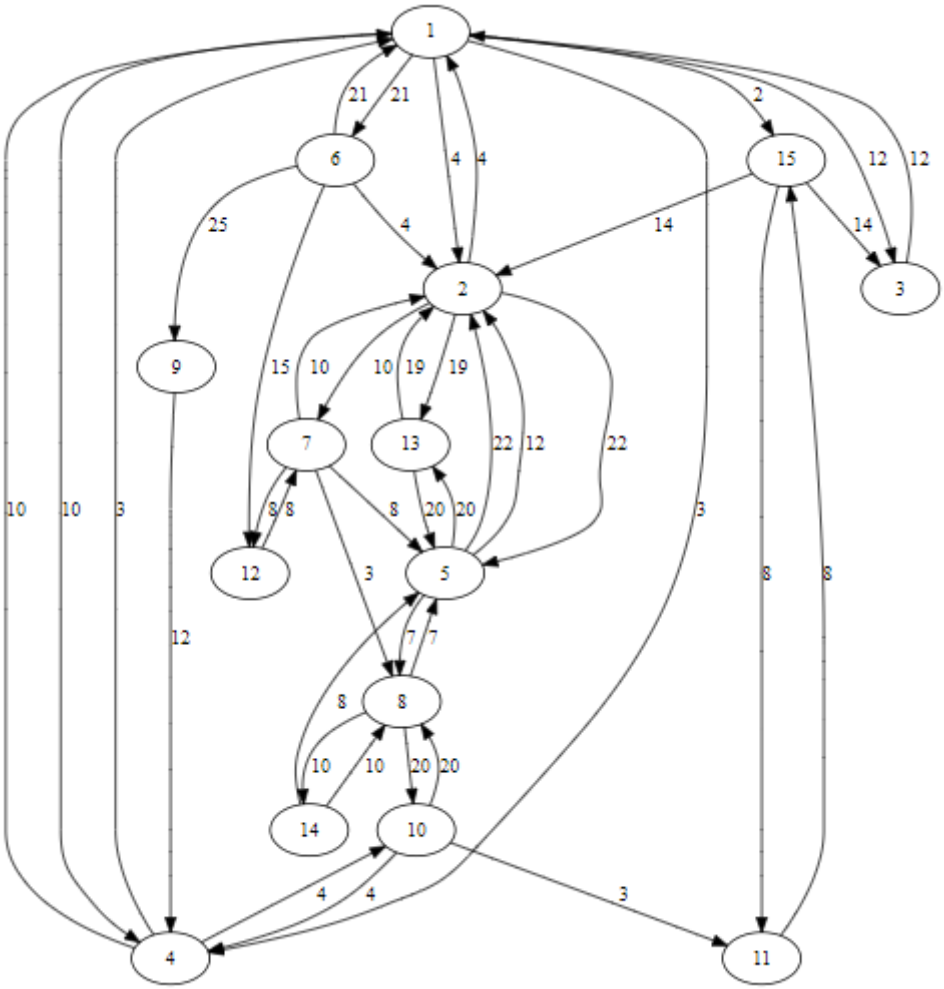
Graphe aléatoire de 3 nœuds:



Graphe aléatoire de 8 nœuds:



Graphe aléatoire de 15 noeuds:



Comparaison de résultats et conclusion :

Question 27 :

Nous avons réaliser 6 tests comme pour la question 24 mais cette fois avec des Graphes généré aléatoirement et dont le nombre de Nœuds est croissant.

```
<<< Graphe de 3 noeuds >>>
Temps d'exécution (BellmanFord) : 626600 ns
Temps d'exécution (Dijkstra) : 1257300 ns

<<< Graphe de 5 noeuds >>>
Temps d'exécution (BellmanFord) : 39500 ns
Temps d'exécution (Dijkstra) : 108500 ns

<<< Graphe de 7 noeuds >>>
Temps d'exécution (BellmanFord) : 110700 ns
Temps d'exécution (Dijkstra) : 118400 ns

<<< Graphe de 20 noeuds >>>
Temps d'exécution (BellmanFord) : 460900 ns
Temps d'exécution (Dijkstra) : 488700 ns

<<< Graphe de 100 noeuds >>>
Temps d'exécution (BellmanFord) : 6064700 ns
Temps d'exécution (Dijkstra) : 2210000 ns
```

On remarque que pour des petit graphes, L'algorithme de Bellman-Ford et plus efficace mais après Dijkstra sera toujours plus rapide.

Question 28 :

Le rapport de performance entre l'algorithme de Bellman-Ford et l'algorithme de Dijkstra dépend du nombre de nœuds et de la structure du graphe.

L'algorithme de Dijkstra a une meilleure performance en moyenne par rapport à l'algorithme de Bellman-Ford, car il utilise une approche basée sur la recherche de la plus courte distance à partir d'un nœud de départ vers tous les autres nœuds du graphe.

D'un autre côté, l'algorithme de Bellman-Ford peut gérer les graphes avec des arêtes de coût négatif, ce que l'algorithme de Dijkstra ne peut pas faire directement.

Donc, en général, l'algorithme de Dijkstra est plus efficace que l'algorithme de Bellman-Ford pour la recherche du plus court chemin dans un graphe, sauf dans les cas où les arêtes de coût négatif sont présentes. Cependant, il est important de noter que la performance exacte dépendra de la taille du graphe, de la densité des arêtes et de la présence d'arêtes de coût négatif. Le rapport de performance entre les deux algorithmes peut varier en fonction de ces facteurs.

Question 29 :

L'étude indique que l'algorithme de Dijkstra est généralement plus rapide que l'algorithme de Bellman-Ford pour trouver le plus court chemin dans un graphe, surtout lorsque les arêtes du graphe ont des coûts positifs. Cela est dû au fait que l'algorithme de Dijkstra est optimisé et prend moins de temps pour trouver la solution. En revanche, l'algorithme de Bellman-Ford peut gérer les graphes avec des coûts négatifs, ce que l'algorithme de Dijkstra ne peut pas faire directement. Cependant, l'algorithme de Bellman-Ford est généralement plus lent en raison de son fonctionnement qui nécessite plus d'opérations.

En conclusion, le choix de l'algorithme dépend des caractéristiques du graphe et de la présence ou non de coûts négatifs. Si le graphe a uniquement des coûts positifs, il est préférable d'utiliser l'algorithme de Dijkstra pour une meilleure performance. Si le graphe a des coûts négatifs, l'algorithme de Bellman-Ford est nécessaire, même s'il est généralement plus lent. Il est important de prendre en compte ces facteurs lors du choix de l'algorithme adapté à chaque situation.