

ILU4 - Rapport des Travaux Pratiques de test

Question 0 - Spécification du programme Bellman-Ford en Java

Note : Le programme se trouve dans le fichier `BellmanFord.java`, dans la fonction `BellmanFordAlgo`.

Spécification du programme :

Le programme renvoie une liste de distance du chemin le plus court entre le sommet `source` et tous les autres sommets du graphe s'il n'y a pas de cycle à poids négatif, une liste vide sinon. Si un chemin entre `source` et un sommet n'existe pas, la distance est égal à la taille maximale du type `Integer`.

Détails de la classe `BellmanFord` :

Cette classe modélise un graphe avec comme attributs le nombre de sommets du graphe `nb_sommets`, le nombre d'arêtes du graphe `nb_aretes` et le tableau `aretes` contenant les arêtes du graphe de type `Arete` définies par un sommet de départ `source`, un sommet d'arrivée `destination` et un poids `poids`.

Elle contient la méthode `BellmanFordAlgo` qui exécute l'algorithme en 3 étapes :

1. Initialisation des distances (∞ sauf pour le sommet source).
2. Détermination des plus petites distances en `nb_sommets - 1` tours.
3. Détection de cycles de poids négatif.

Question 1

Voir fichier `BellmanFordTestJSON.java` et `JeuDeTestEtOracleFormat.java`

Nous avons décidé d'utiliser du JSON pour se familiariser avec d'autres données que des fichiers texte, de plus le JSON est utilisé dans beaucoup d'infrastructures logiciels.

Question 2

Étude de notre programme sur le graphe suivant :

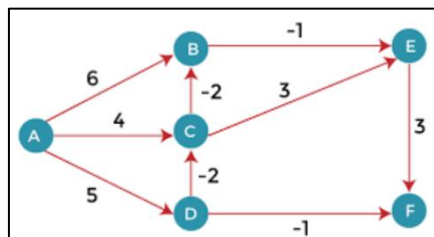


Figure 1 : Graphe étudié à la question 2

Après la construction du graphe (voir `Question2.java`) via le constructeur `BellmanFord` et les méthodes `addAretes`, nous obtenons le résultat suivant :

```

Integer[] bellmanFordAlgo(BellmanFord graph, int source) {
    int nb_sommets = graph.nb_sommets; nb_aretes = graph.nb_aretes;
    Integer dist[] = new Integer[nb_sommets];

    // Etape 1: Initialise la distance du sommet source à tous les autres
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[source] = 0;

    // Etape 2: Parcours de toutes les arêtes |nb_sommets| - 1 fois.
    for (int i = 1; i < nb_sommets; ++i) {
        for (int j = 0; j < nb_aretes; ++j) {
            int u = graph.aretes[j].source;
            int v = graph.aretes[j].destination;
            int poids = graph.aretes[j].poids;
            if (dist[u] != Integer.MAX_VALUE && dist[u] + poids < dist[v])
                dist[v] = dist[u] + poids;
        }
    }

    // Etape 3: Verification de la non existence d'un cycle négatif
    for (int j = 0; j < nb_aretes; ++j) {
        int u = graph.aretes[j].source;
        int v = graph.aretes[j].destination;
        int poids = graph.aretes[j].poids;
        if (dist[u] != Integer.MAX_VALUE && dist[u] + poids < dist[v])
            return new Integer[0];
    }

    return dist;
}

```

Figure 2 : Résultat de la couverture de code sur le graphe donné en énoncé

Les lignes surlignées en vert correspondent aux cas dans lesquels l'algorithme est passé dans le cas de notre graphe, les lignes en rouge l'inverse. Les lignes en jaune correspondent aux conditions dans lesquelles seulement certains prédicats passent.

Exemple : dans le second `if` en jaune, on a, en plus du surlignage jaune, un affichage « 2 of 4 branches missed », en approfondissant, on comprend que l'on est entré dans les branches où le premier prédicat vaut true (branche 1), le second vaut false (branche 2).

On sait d'ailleurs d'avance que le cas (true, true) n'est pas passé : la ligne dans le bloc du `if` est en rouge.

Explication : Soit les prédicats (p_1, p_2) joint par un ET logique : $(p_1 \ \&\& \ p_2)$ (4 branches), (true, false) couvre 2 branches (**p_1 =true, p_2 =false**), (false, false) couvre 1 branche (**p_1 =false, p_2 non évalué**) (équivalent à (false, true)), (true, true) couvre 1 branche (**p_1 =true, p_2 =true**).

Ainsi, pour le graphe de la Figure 1, notre programme passe par les instructions en vert de la Figure 2 ainsi que dans certains prédicats en jaunes. Il ne passe pas par la ligne en rouge qui est réservée aux cycles de poids négatifs, ce graphe n'en a pas.

Note : le reste de la classe ne nous intéresse pas pour la Figure 1, on l'exécute forcément pour construire notre graphe (addArete(...), constructeur...), c'est forcément en vert.

Question 3

Voir EmmaMain.java

Protocole

Dans notre algorithme, on peut couvrir toutes les instructions en 2 tests :

- Cas standard : Plusieurs sommets, pas de cycle négatifs
- Cas cycle de poids négatif : Un cycle de poids négatif dans le graphe

```

public class EmmaMain {
    public static void main(String[] args) {
        testCasStandard();
        testCasCycleNegatif();
    }

    // Cas classique
    private static void testCasStandard() {
        System.out.println("Test cas standard :");
        BellmanFord graphe = new BellmanFord(6,9);
        graphe.addArete(0, 1, 6);
        graphe.addArete(0, 2, 4);
        graphe.addArete(0, 3, 5);
        graphe.addArete(1, 4, -1);
        graphe.addArete(2, 1, -2);
        graphe.addArete(2, 4, 3);
        graphe.addArete(3, 2, -2);
        graphe.addArete(3, 5, -1);
        graphe.addArete(4, 5, 3);
        System.out.println(Arrays.toString(graphe.bellmanFordAlgo(graphe, 2)));
    }

    // Cas cycle négatif
    private static void testCasCycleNegatif() {
        System.out.println("Test cas cycle négatif :");
        BellmanFord graphe = new BellmanFord(3, 3);
        graphe.addArete(0, 1, 1);
        graphe.addArete(1, 2, -2);
        graphe.addArete(2, 0, -1);
        System.out.println(Arrays.toString(graphe.bellmanFordAlgo(graphe, 0)));
    }
}

```

Figure 3: Test de couverture globale

Pour le cas standard, nous avons décidé d'utiliser le même graphe que la question 2 mais pas le même sommet de départ.

Nous avons choisi un sommet initiale différent de 0 car dans la fonction on parcourt les sommets dans l'ordre croissant et donc forcément la distance du sommet initiale 0 au sommet 0 est égale à 0 et on ne rencontre pas de valeur infinie. Tandis que si on choisit un sommet initiale différent de 0, lors du parcours de la boucle la distance du sommet initiale au sommet 0 sera infinie.

Résultat du cas standard

```
Integer[] bellmanFordAlgo(BellmanFord graph, int source) {
    int nb_sommets = graph.nb_sommets, nb_aretes = graph.nb_aretes;
    Integer dist[] = new Integer[nb_sommets];

    // Etape 1: Initialise la distance du sommet source à tous les autres
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[source] = 0;

    // Etape 2: Parcours de toutes les arêtes |nb_sommets| - 1 fois.
    for (int i = 1; i < nb_sommets; ++i) {
        for (int j = 0; j < nb_aretes; ++j) {
            int u = graph.aretes[j].source;
            int v = graph.aretes[j].destination;
            int poids = graph.aretes[j].poids;
            if (dist[u] != Integer.MAX_VALUE && dist[u] + poids < dist[v])
                dist[v] = dist[u] + poids;
        }
    }

    // Etape 3: Verification de la non existence d'un cycle négatif
    for (int j = 0; j < nb_aretes; ++j) {
        int u = graph.aretes[j].source;
        int v = graph.aretes[j].destination;
        int poids = graph.aretes[j].poids;
        if (dist[u] != Integer.MAX_VALUE && dist[u] + poids < dist[v])
            return new Integer[0];
    }

    return dist;
}
```

Figure 4 : Résultat de la couverture globale 1/2

La ligne en rouge n'est pas atteinte, car il n'y a pas de cycle de poids négatif dans ce cas-là.

Résultat du cas standard et du cas cycle de poids négatifs

```
Integer[] bellmanFordAlgo(BellmanFord graph, int source) {
    int nb_sommets = graph.nb_sommets, nb_aretes = graph.nb_aretes;
    Integer dist[] = new Integer[nb_sommets];

    // Etape 1: Initialise la distance du sommet source à tous les autres
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[source] = 0;

    // Etape 2: Parcours de toutes les arêtes |nb_sommets| - 1 fois.
    for (int i = 1; i < nb_sommets; ++i) {
        for (int j = 0; j < nb_aretes; ++j) {
            int u = graph.aretes[j].source;
            int v = graph.aretes[j].destination;
            int poids = graph.aretes[j].poids;
            if (dist[u] != Integer.MAX_VALUE && dist[u] + poids < dist[v])
                dist[v] = dist[u] + poids;
        }
    }

    // Etape 3: Verification de la non existence d'un cycle négatif
    for (int j = 0; j < nb_aretes; ++j) {
        int u = graph.aretes[j].source;
        int v = graph.aretes[j].destination;
        int poids = graph.aretes[j].poids;
        if (dist[u] != Integer.MAX_VALUE && dist[u] + poids < dist[v])
            return new Integer[0];
    }

    return dist;
}
```

Figure 5 : Résultat de la couverture globale 2/2

Question 4

Nous allons utiliser les partitions dites « à partir des interfaces » d'après le cours

« ILU4.TestFonctionnel » : on va considérer que les valeurs d'entrées sont bien typés et du bon nombre, en Java, ces erreurs sont détectés à la compilation et non à l'exécution, exemple :

```
new Integer(32, 32) // constructeur Integer avec 2 arguments non-existant, erreur
détecté dès la compilation, pas testable

new Integer(3.4f) // constructeur avec ce type non-existant, erreur détecté à la
compilation
```

Nous testons donc uniquement les classes valides dans le cas du nombre d'argument et type (C1 à C6).

Tableau des exigences :

Exigence	Classe valide	Classe invalide
C1 Nombre d'entrées BellmanFordAlgo	2	
C2 Sommet de départ entier	source entier	
C3 Nombre de sommets entier	nb_sommets entier	
C4 Nombre d'arêtes entier	nb_aretes entier	
C5 Poids des arêtes entier	Tous les poids entier	
C5 Source des arêtes entier	Toutes les sources entier	
C6 Destination des arêtes entier	Toute les destination entier	
C7 Sommet de départ valide	$0 \leq \text{source} < \text{nb_sommets}$	source < 0 source \geq nb_sommets
C8 Nombre de sommets positif	nb_sommets > 0	nb_sommets \leq 0
C9 Nombre d'arêtes positif ou nul	nb_aretes \geq 0	nb_aretes < 0
C10 Source des arêtes valide	$\forall \text{arêtes}, 0 \leq \text{source} < \text{nb_sommets}$	\exists arête, source < 0 \exists arête, source \geq nb_sommets
C11 Destination des arêtes valide	$\forall \text{arêtes}, 0 \leq \text{destination} < \text{nb_sommets}$	\exists arête, destination < 0 \exists arête, destination \geq nb_sommets
C12 Liste d'arêtes correspondant au nombre d'arête	Longueur liste arêtes \leq nb_aretes	Longueur liste arêtes > nb_aretes
C13 BellmanFord renvoie un tableau de distance entre la source et les points quand le graphe n'a pas de cycle de poids négatifs	Le graphe ne contient aucun cycle de poids négatifs	
C14 BellmanFord renvoie un tableau vide quand le graphe a un cycle de poids négatifs	Le graphe contient un cycle de poids négatifs	
C15 BellmanFord renvoie un tableau de distance normale quand un sommet a un poids négatif mais que le graphe n'a pas de cycle de poids négatifs	Le graphe contient un poids négatif mais pas de cycle de poids négatifs	
C16 BellmanFord renvoie un tableau de distance normale quand un sommet est isolé, la distance dans le tableau pour ce sommet est MAX_INT	Au moins un sommet n'est pas connecté aux autres	

Jeux de tests :

Exemple : C7-Invalide-2 correspond à C7 dans le deuxième cas invalide : « source \geq nb_sommets »

C8-Valide correspond à C8 dans le (seule) cas valide : « nb_sommets > 0 »

C1 correspond à C1 dans le cas valide (aucun cas invalide)

	Nom du test (dans le code)	Cas testé	Paramètres
J1	testSansPoidsNegatifSansSommetIsolé	C1, C2, C3, C4, C5, C6, C5, C6, C7-Valide, C8-Valide, C9-Valide, C10-Valide, C11-Valide, C12-Valide, C13-Valide	nb_sommets = 3 nb_aretes = 3 Arêtes ajoutées : Format (source, destination, poids) - (0, 1, 4) - (1, 2, 3) - (2, 0, 10) sommet_depart = 0

J2	testSommetDeDepartNegatif	C7-Invalide-1	nb_sommets = 3 nb_aretes = 3 Aucune arêtes ajoutées sommet_depart = -1
J3	testSommetDeDepartSupOuEgalANbSommets	C7-Invalide-2	nb_sommets = 3 nb_aretes = 3 Aucune arêtes ajoutées sommet_depart = 3
J4	testNbSommetsInfOuEgalA0	C8-Invalide	2 cas (différents traitement selon si nbSommet = 0 ou < 0) : <i>J4.1</i> nb_sommets = -2 nb_aretes = 3 Aucune arêtes ajoutées sommet_depart = 0 <i>J4.2</i> nb_sommets = 0 nb_aretes = 3 Aucune arêtes ajoutées sommet_depart = 0
J5	testNbAretesNegatif	C9-Invalide	nb_sommets = 3 nb_aretes = -1 Aucune arêtes ajoutées
J6	testUneAreteAvecSourceNegatif	C10-Invalide-1	nb_sommets = 3 nb_aretes = 1 Arêtes ajoutées : - (-2, 0, 10) sommet_depart = 0
J7	testUneAreteAvecSourceSupOuEgalANbSommets	C10-Invalide-2	nb_sommets = 3 nb_aretes = 1 Arêtes ajoutées : - (5, 0, 10) sommet_depart = 0
J8	testUneAreteAvecDestinationNegatif	C11-Invalide-1	nb_sommets = 3 nb_aretes = 1 Arêtes ajoutées : - (0, -2, 10) sommet_depart = 0
J9	testUneAreteAvecDestinationSupOuEgalANbSommets	C11-Invalide-2	nb_sommets = 3 nb_aretes = 1 Arêtes ajoutées : - (0, 5, 10) sommet_depart = 0
J10	testLongueurListeAreteSupANbArete	C12-Invalide	nb_sommets = 2 nb_aretes = 1 Arêtes ajoutées : - (0, 1, 6) - (1, 0, 3)
J11	testCycleDePoidsNegatif	C14	nb_sommets = 3 nb_aretes = 3 Arêtes ajoutées : - (0, 1, -4) - (1, 2, -3) - (2, 0, 1) sommet_depart = 0
J12	testPoidsNegatifSansCycleDePoidsNegatif	C15	nb_sommets = 3 nb_aretes = 3 Arêtes ajoutées : - (0, 1, 4) - (1, 2, -3) - (2, 0, 10) sommet_depart = 0
J13	testSommetIsoleAvecGrapheValide	C16	nb_sommets = 3 nb_aretes = 1 Arêtes ajoutées : - (0, 1, 4) sommet_depart = 0

Les tests par partition codés, avec JUnit 5, se trouvent ainsi dans TestPartition.java.

Question 5

Tableau des exigences :

Exigence	Classe valide	Classe invalide
C7 Sommet de départ valide	source = 0 source = nb_sommets - 2 source = nb_sommets - 1	source = -1 source = nb_sommets
C8 Nombre de sommets positif	nb_sommets = 1 nb_sommets = 2 nb_sommets = max int nb_sommets = max int - 1	nb_sommets = 0 nb_sommets = max int + 1
C9 Nombre d'arêtes positif ou nul	nb_aretes = 0 nb_aretes = 1 nb_aretes = max int nb_aretes = max int - 1	nb_aretes = -1 nb_aretes = max int + 1
C10 Source des arêtes valide	source = 0 source = 1 source = nb_sommets - 2 source = nb_sommets - 1	source = -1 source = nb_sommets
C11 Destination des arêtes valide	destination = 0 destination = 1 destination = nb_sommets - 2 destination = nb_sommets - 1	destination = -1 destination = nb_sommets
C5 Poids des arêtes entier	poids = min int poids = min int + 1 poids = max int poids = max int - 1	poids = min int - 1 poids = max int + 1
C12 Liste d'arêtes correspondant au nombre d'arête	Nombre d'appels à addArete = 0 Nombre d'appels à addArete < nb_aretes Nombre d'appels à addArete = nb_aretes	Nombre d'appels à addArete >= nb_aretes

Jeux de tests :

	Nom du test	Cas testé	Paramètres
J1	testSommetSeulSansArete	C7-Valide-1, C8-Valide-1, C9-Valide-1, C12-Valide-1	nb_sommets = 1 nb_aretes = 0 Aucune arêtes ajoutées sommet_depart = 0
J2	testGrapheClassique	C7-Valide-3, C8-Valide-2, C10-Valide, C11-Valide, C5-Valide, C12-Valide-3	nb_sommets = 2 nb_aretes = 4 Arêtes ajoutées : (0,0,min int) (0,1,max int) (1,0,min int + 1) (1,0,max int - 1) sommet_depart = 1
J3	testGrapheUnSeuleArete	C12-Valide-2, C9-Valide-2	nb_sommets = 1 nb_aretes = 1 Aucune arêtes ajoutées sommet_depart = 0
J4	testSommetDepartNbSommetsMoinsDeux	C7-Valide-2	nb_sommets = 4 nb_aretes = 4 Arêtes ajoutées : (2,1,2) (2,3,2) (2,0,2) sommet_depart = 2
J5	testZeroSommets	C8-Invalide-1	nb_sommets = 0 nb_aretes = 0 Aucune arêtes ajoutées sommet_depart = 0
J6	testMaxIntPlusUnSommets	C8-Invalide-2	nb_sommets = max int + 1

			nb_arettes = 0 Aucune arêtes ajoutées sommet_depart = 0
J7	testNbAretesNegatif	C9-Invalide-1	nb_sommets = 1 nb_arettes = -1
J8	testMaxIntPlusUnAretes	C9-Invalide-2	nb_sommets = 1 nb_arettes = max int + 1
J9	testSommetDepartNegatif	C7-Invalide-1	nb_sommets = 1 nb_arettes = 0 Aucune arêtes ajoutées sommet_depart = -1
J10	testSommetDepartInconnu	C7-Invalide-2	nb_sommets = 1 nb_arettes = 0 Aucune arêtes ajoutées sommet_depart = 1
J11	testSourceAretelInvalideInferieur	C10-Invalide-1	nb_sommets = 2 nb_arettes = 2 Arêtes ajoutées : (-1,1,6) (1,1,6) sommet_depart = 1
J12	testDestinationAretelInvalideInferieur	C11-invalid-1	nb_sommets = 2 nb_arettes = 2 Arêtes ajoutées : (1,-1,6) (1,1,6) sommet_depart = 1
J13	testSourceAretelInvalideSuperieur	C10-Invalide-2	nb_sommets = 2 nb_arettes = 2 Arêtes ajoutées : (2,1,6) (1,1,6) sommet_depart = 1
J14	testDestinationAretelInvalideSuperieur	C11-Invalide-2	nb_sommets = 2 nb_arettes = 2 Arêtes ajoutées : (1,2,6) (1,1,6) sommet_depart = 1
J15	testNbAreteAjouteInvalide	C12-Invalide	nb_sommets = 2 nb_arettes = 1 Arêtes ajoutées : (0,1,6) (1,0,3)
J16	testPoidsDepassementSuperieur	C5-Invalide-1	nb_sommets = 2 nb_arettes = 1 Arêtes ajoutées : (0,1,max int + 1) sommet_depart = 0
J17	testPoidsDepassementInferieur	C5-Invalide-2	nb_sommets = 2 nb_arettes = 1 Arêtes ajoutées : (0,1,min int - 1) sommet_depart = 0

Pour les tests J18, J19, J20, J21 qui concerne respectivement les cas de C8-Valide-3, C8-Valide-4, C9-Valide-3 et C9-Valide-4 nous n'avons pas pu les exécuter car Java nous retourne une erreur de dépassement de mémoire. En théorie, ces tests sont des classes valides mais en pratique Java ne nous permet pas de les réaliser.

Les tests aux limites codés, avec JUnit 5, se trouvent ainsi dans TestLimites.java.

Question 6

Voir *BellmanFordMutant1.java*, *BellmanFordMutant2.java*, *BellmanFordMutant3.java*.

Mutation 1:

```
// Etape 3: Verification de la non existence d'un cycle négatif
for (int j = 0; j < nb_aretes; ++j) {
    int u = graph.aretes[j].source;
    int v = graph.aretes[j].destination;
    int poids = graph.aretes[j].poids;
    if (dist[u] != Integer.MAX_VALUE && dist[u] + poids <= dist[v])
        return new Integer[0];
}
```

Figure 6 : extrait de *BellmanFordMutant1.java*

Nous réalisons la première mutation sur l'étape 3 en remplaçant < par <=.

Observations des résultats sur les tests de partitions et de limites :

```
testCycleDePoidsNegatif() (0,009 s)
testUneAreteAvecSourceSupOuEgalANbSommets() (0,007 s)
testUneAreteAvecSourceNegatif() (0,002 s)
testSommetsDeDepartSupOuEgalANbSommets() (0,003 s)
testSansPoidsNegatifSansSommetsIsole() (0,010 s)
testSommetsIsoleAvecGrapheValide() (0,002 s)
testNbSommetsInfOuEgalA0() (0,003 s)
testUneAreteAvecDestinationSupOuEgalANbSommets() (0,003 s)
testPoidsNegatifSansCycleDePoidsNegatif() (0,003 s)
testUneAreteAvecDestinationNegatif() (0,002 s)
testSommetsDeDepartNegatif() (0,002 s)
testNbAretesNegatif() (0,002 s)
```

Figure 7 : Résultat tests partition mutant 1

```
testDestinationAreteInvalidInferieur() (0,025 s)
testGrapheClassique() (0,019 s)
testSommetsDepartNbSommetsMoinsDeux() (0,002 s)
testSommetsSeulSansArete() (0,003 s)
testSourceAreteInvalidSuperieur() (0,002 s)
testGrapheUnSeuleArete() (0,004 s)
testMaxIntPlusUnSommets() (0,003 s)
testZeroSommets() (0,002 s)
testSourceAreteInvalidInferieur() (0,001 s)
testNbAreteAjouteInvalid() (0,002 s)
testDestinationAreteInvalidSuperieur() (0,002 s)
testSommetsDepartNegatif() (0,001 s)
testNbAretesNegatif() (0,001 s)
testSommetsDepartInconnu() (0,002 s)
testMaxIntPlusUnAretes() (0,001 s)
```

Figure 8 : Résultat tests limites mutant 1

Nous observons 6 tests qui ne passent pas. Ces tests concernent tous le cas classique (valide et sans cycle négatif). Prenons le test *testSansPoidsNegatifSansSommetsIsole* :

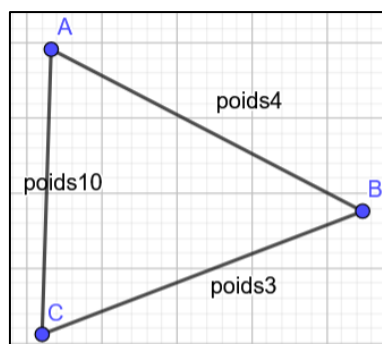


Figure 9 : Exemple de cas pour le premier mutant

L'algorithme effectue 2 itérations ($\text{nb_sommets} - 1$) et détecte au tour 1 et 2 la même distance minimale entre le point A (source) et B : 4. Donc (grâce au = dans <=) il détecte un cycle de poids négatif alors qu'il n'y en a pas. La mutation détecte des faux positifs : des cycles négatifs là où il devrait ne pas y en avoir.

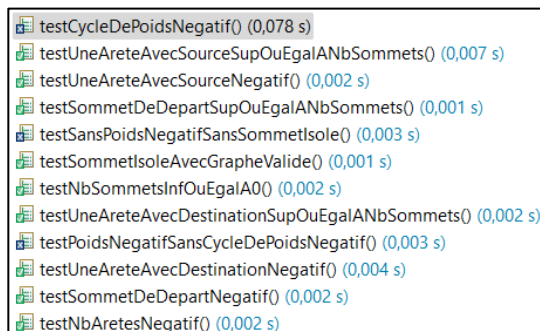
Mutation 2:

```
// Etape 3: Verification de la non existence d'un cycle négatif
for (int j = 0; j < nb_aretes; ++j) {
    int u = graph.aretes[j].source;
    int v = graph.aretes[j].destination;
    int poids = graph.aretes[j].poids;
    if (dist[u] != Integer.MAX_VALUE && dist[u] + poids > dist[v])
        return new Integer[0];
}
```

Figure 10 : extrait de BellmanFordMutant2.java

Nous réalisons la deuxième mutation sur l'étape 3 en remplaçant < par > cette fois.

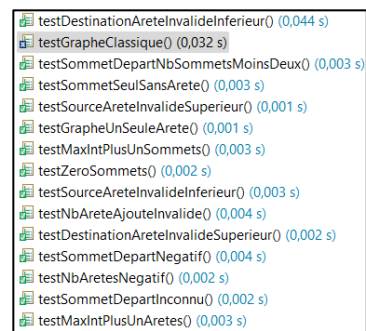
Observations des résultats sur les tests de partitions et de limites :



A screenshot of a test runner window showing a list of 13 test cases. Each test case is preceded by a small icon (a document with a checkmark) and followed by its execution time in seconds. The tests are:

- testCycleDePoidsNegatif() (0,078 s)
- testUneAreteAvecSourceSupOuEgalANbSommets() (0,007 s)
- testUneAreteAvecSourceNegatif() (0,002 s)
- testSommetDeDepartSupOuEgalANbSommets() (0,001 s)
- testSansPoidsNegatifSansSommetsIsole() (0,003 s)
- testSommetIsoleAvecGrapheValide() (0,001 s)
- testNbSommetsInfOuEgalA0() (0,002 s)
- testUneAreteAvecDestinationSupOuEgalANbSommets() (0,002 s)
- testPoidsNegatifSansCycleDePoidsNegatif() (0,003 s)
- testUneAreteAvecDestinationNegatif() (0,004 s)
- testSommetDeDepartNegatif() (0,002 s)
- testNbAretesNegatif() (0,002 s)

Figure 11 : Résultat tests partition 2



A screenshot of a test runner window showing a list of 15 test cases. Each test case is preceded by a small icon (a document with a checkmark) and followed by its execution time in seconds. The tests are:

- testDestinationAreteInvalidelInferieur() (0,044 s)
- testGrapheClassique() (0,032 s)
- testSommetDepartNbSommetsMoinsDeux() (0,003 s)
- testSommetSeulSansArete() (0,003 s)
- testSourceAreteInvalidesuperieur() (0,001 s)
- testGrapheUnSeuleArete() (0,001 s)
- testMaxIntPlusUnSommets() (0,003 s)
- testZeroSommets() (0,002 s)
- testSourceAreteInvalidelInferieur() (0,003 s)
- testNbAreteAjouteInvalides() (0,004 s)
- testDestinationAreteInvalidesuperieur() (0,002 s)
- testSommetDepartNegatif() (0,004 s)
- testNbAretesNegatif() (0,002 s)
- testSommetDepartInconnu() (0,002 s)
- testMaxIntPlusUnAretes() (0,003 s)

Figure 12 : Résultat tests limites mutant 2

On inverse toute la logique, dans certains cas, quand il y a un cycle de poids négatif, il n'est pas détecté, quand il n'y en a pas, il est détecté.

Exemple : testCycleDePoidsNegatif, le cycle n'est pas détecté, on fait 2 tours (nb_sommets - 1), et on obtient pour la distance minimale du sommet 0 au sommet 1 : -4 (tour 1), -6 (tour 2), et la condition se fait entre -6 et -4, dans le programme originale on fait -6 < -4 alors cycle négatif, ici on fait -6 > -4 donc le cycle négatif n'est jamais détecté.

Mutation 3 :

```
Integer[] bellmanFordAlgo(BellmanFord graph, int source) {  
    int nb_sommets = graph.nb_sommets, nb_aretes = graph.nb_aretes;  
    Integer dist[] = new Integer[nb_sommets + 1];  
}
```

Figure 13 : extrait de BellmanFordMutant3.java

Nous réalisons la troisième mutation à l'initialisation de l'algorithme en remplaçant la taille max du tableau des distances minimales `dist, nb_sommets` par `nb_sommets - 1`.

Observations des résultats sur les tests de partitions et de limites :

- testCycleDePoidsNegatif() (0,018 s)
- testUneAreteAvecSourceSupOuEgalANbSommets() (0,007 s)
- testUneAreteAvecSourceNegatif() (0,003 s)
- testSommetDeDepartSupOuEgalANbSommets() (0,012 s)
- testSansPoidsNegatifSansSommetIsole() (0,009 s)
- testSommetIsoleAvecGrapheValide() (0,002 s)
- testNbSommetsInfOuEgalA0() (0,004 s)
- testUneAreteAvecDestinationSupOuEgalANbSommets() (0,001 s)
- testPoidsNegatifSansCycleDePoidsNegatif() (0,002 s)
- testUneAreteAvecDestinationNegatif() (0,002 s)
- testSommetDeDepartNegatif() (0,003 s)
- testNbAretesNegatif() (0,003 s)

Figure 14 : Résultat tests partition mutant 3

- testPoidsDepassementInferieur() (0,023 s)
- testDestinationAreteInvalideInferieur() (0,006 s)
- testGrapheClassique() (0,002 s)
- testSommetDepartNbSommetsMoinsDeux() (0,003 s)
- testSommetSeulSansArete() (0,001 s)
- testSourceAreteInvalideSuperieur() (0,003 s)
- testGrapheUnSeuleArete() (0,002 s)
- testMaxIntPlusUnSommets() (0,001 s)
- testZeroSommets() (0,001 s)
- testSourceAreteInvalideInferieur() (0,001 s)
- testNbAreteAjouteInvalide() (0,001 s)
- testPoidsDepassementSuperieur() (0,001 s)
- testDestinationAreteInvalideSuperieur() (0,003 s)
- testSommetDepartNegatif() (0,001 s)
- testNbAretesNegatif() (0,001 s)
- testSommetDepartInconnu() (0,002 s)

Figure 15 : Résultat tests limites mutant 3

L'ajout d'un sommet possible provoque un comportement pertinent sur le test `testSommetDeDepartSupOuEgalANbSommets`, le test met le sommet de départ de l'algorithme à 3 (égal à `nb_sommet`), dans le cas normal c'est impossible, mais ici `dist[3]` va être possible et donc permettre à ce test de ne pas renvoyer d'exception, et donc de réussir.