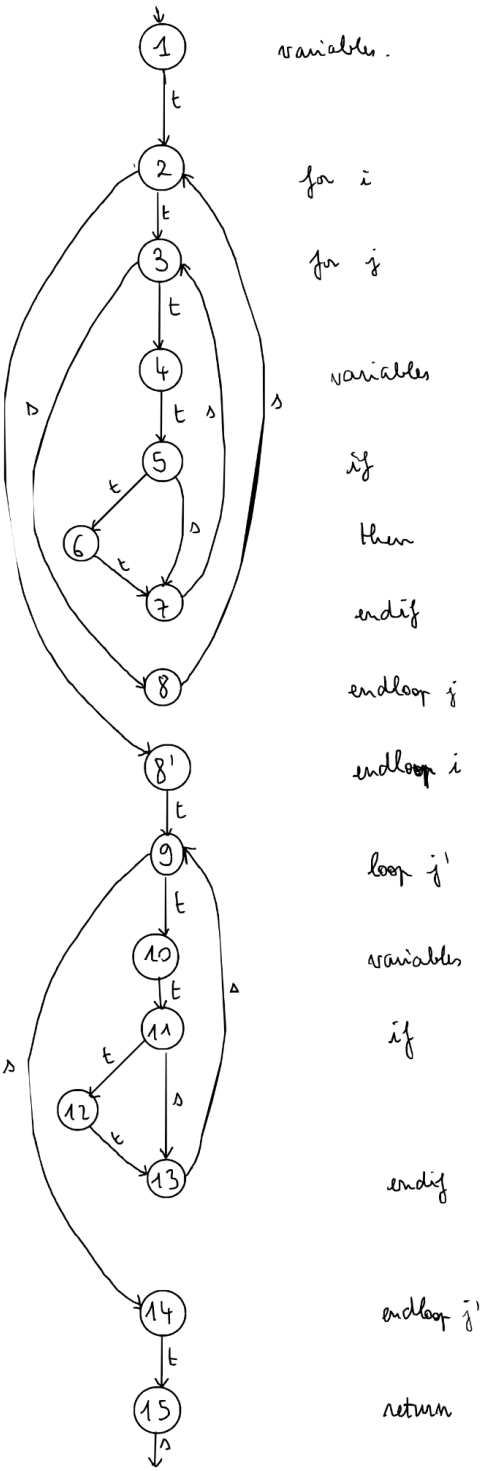
Lagier Hadrien
Vintrou Mathéo

# Rapport TP de ILU4 partie test

## Graphe de Contrôle de l'algorithme Bellman-Ford :



Pour ce TP nous avons décidé de partir sur un script de test en Java, dans un premier temps nous avons implémenté cet algorithme de Bellman Ford, en modifiant un peu son code afin d'obtenir des lettres majuscules en nom de noeud d'un graphe plutôt que des chiffres. Dans un second temps, nous avons implémenté une fonction static `runTestCase` qui prend en argument le nom d'un fichier texte, construit le graphe à partir des données du fichier texte puis test si le poid du plus court chemin entre deux point est bien celui renseigné dans le fichier. Et enfin, nous avons implémenté un `main` afin d'exécuter les différents tests de ce TP.

Voici l'affichage du script de test, il fournit un rapport d'analyse qui précise le nom du fichier de la série de tests, les tests qui ont été exécutés et le résultat de leur exécution (pass/fail), ainsi que le pourcentage des tests qui sont passés.

```
Début du jeu de test du fichier : question3.txt
----------------------------
Test 1 : pass
----------------------------
Test 2 : Graph contains negative weight cycle
fail

Nombre de tests réussis : 50.0 %
```

# Question 2

```
32      // Bellman-Ford Algorithm to find shortest paths from source to all vertices
33      int BellmanFordAlgo(BellmanFord graph, char source, char arrivee) {
34          int V = graph.V, E = graph.E;
35          int dist[] = new int[V];
36          // Step 1: Initialize distances from source to all other vertices as INFINITE
37          Arrays.fill(dist, Integer.MAX_VALUE);
38          dist[source-65] = 0;
39          // Step 2: Relax all edges |V| - 1 times.
40          for (int i = 1; i < V; ++i) {
41              for (int j = 0; j < E; ++j) {
42                  char u = graph.edge[j].source;
43                  char v = graph.edge[j].destination;
44                  int weight = graph.edge[j].weight;
45                  if (dist[u-65] != Integer.MAX_VALUE && dist[u-65] + weight < dist[v-65])
46                      dist[v-65] = dist[u-65] + weight;
47              }
48          }
49          // Step 3: Check for negative-weight cycles
50          for (int j = 0; j < E; ++j) {
51              char u = graph.edge[j].source;
52              char v = graph.edge[j].destination;
53              int weight = graph.edge[j].weight;
54              if (dist[u-65] != Integer.MAX_VALUE && dist[u-65] + weight < dist[v-65]) {
55                  System.out.println("Graph contains negative weight cycle");
56                  return 1;
57              }
58          }
59          return dist[arrivee-65];
60      }
```

Comme le montre l'image ci-dessus, toutes les instructions, hormis celles des lignes 55 et 56, permettent de trouver le plus court chemin dans le graphe de la question 2. La condition de la ligne 54 permet de savoir si le graphe contient une boucle négative, il n'y a en pas dans ce graphe, donc la condition n'est pas vérifiée.

On remarque grâce au coverage Emma que certaines instructions dans la condition et (&&) de notre algorithme n'était pas "couvert". Après modification du jeu de test en y ajoutant des tests au cas limites le coverage était parfait sur l'algorithme.

# Question 3

```
 1 6
 2 9
 3 A B 6
 4 A C 4
 5 A D 5
 6 C B -2
 7 D C -2
 8 B E -1
 9 C E 3
10 E F 3
11 D F -1
12 A B
13 1
14 4
15 4
16 A B 1
17 B C 1
18 C B -2
19 B D 2
20 A D
21 3
```

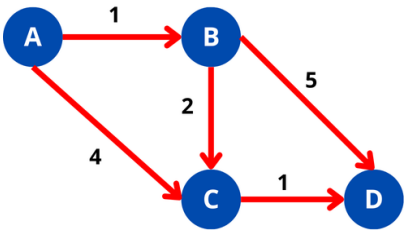
Il est possible de trouver une suite de tests qui couvre toutes les instructions.
Celle-ci contient deux graphe, celui de la question 2 et un graphe contenant un cycle négatif.
Explication des fichiers test :

- ligne 1 : nombre de noeud
- ligne 2 : nombre d'arêtes
- lignes 3 à 11 : les arêtes
- lignes 12 : les noeud entre lesquels on cherche le plus court chemin
- ligne 13 : le poid de ce plus court chemin (résultat attendu)
- ect..

# Question 4

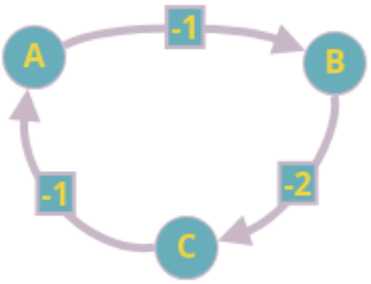
Pour la méthode des partitions et des catégories nous avons opté pour 7 types de graphes :

- un graphe contenant uniquement des arêtes à poids positif : (Test #1)

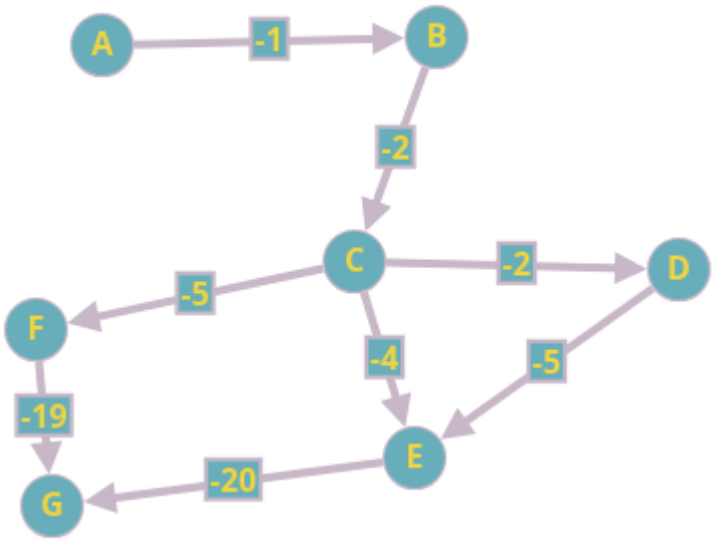



- un graphe contenant une arête à poids négatif uniquement :

  Avec cycle : (Test #2)

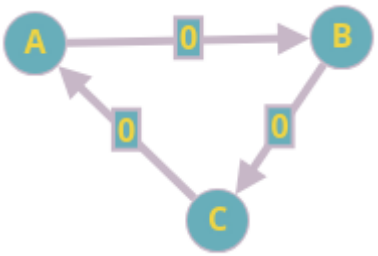


Sans cycle: (Test #3)

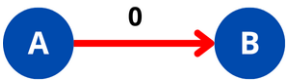



- un graphe avec poid égal à 0 :

  Avec cycle : (Test #4)

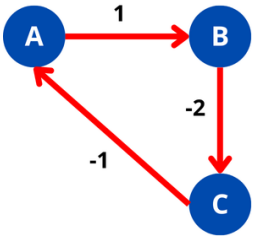



  Sans cycle : (Test #5)

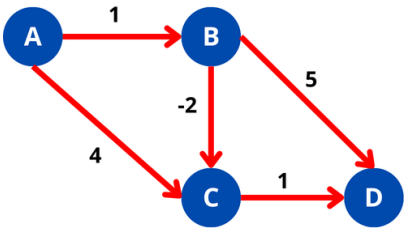



- un graphe avec poids positifs et négatifs :

  Avec cycle : (Test #6)



Sans cycle : (Test #7)




Voici le rapport d'analyse des tests exécutés :

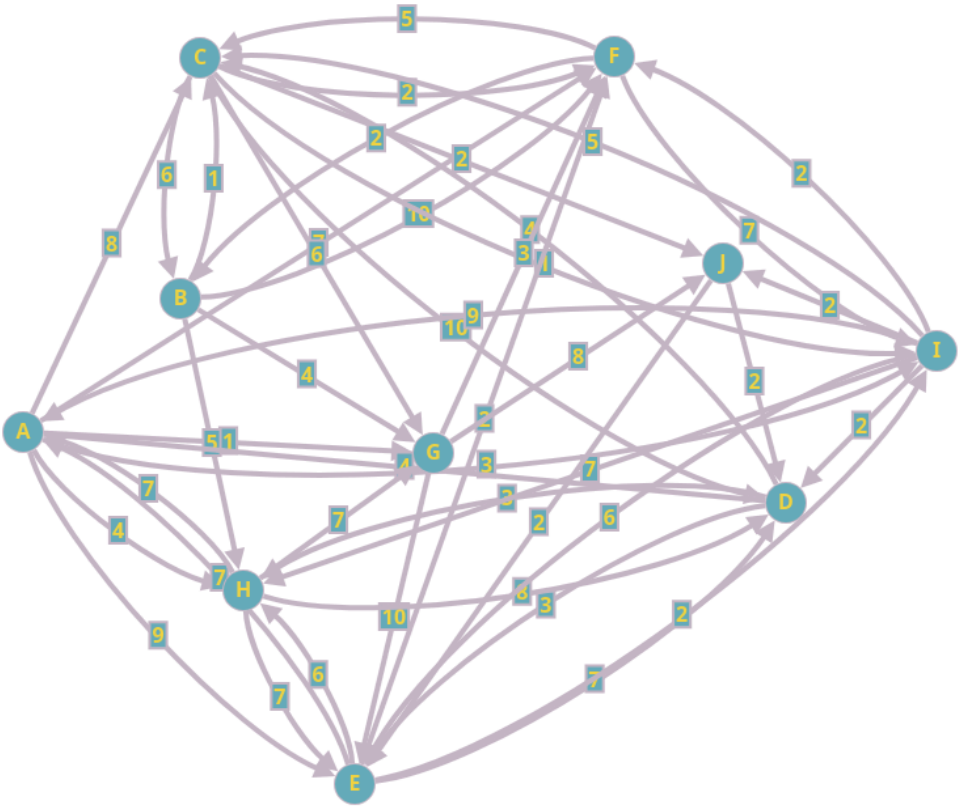
```
Début du jeu de test du fichier : question4.txt
----------------------------
Test 1 : pass
----------------------------
Test 2 : Graph contains negative weight cycle
fail
----------------------------
Test 3 : pass
----------------------------
Test 4 : pass
----------------------------
Test 5 : pass
----------------------------
Test 6 : Graph contains negative weight cycle
fail
----------------------------
Test 7 : pass

Nombre de tests réussis : 71.42857 %
```

# Question 5

Pour une suite de tests appliquées aux cas limites nous décidons

- un graphe vide :
- un graphe avec un seul noeud :


- un graphe fortement connexe (graphe dense) :




- un graphe avec un grand nombre de noeud :

# Question 6

Rappel des résultats des tests précédents :

- question 3 : 50%
- question 4 : 71.4%
- question 5 : 100%

Le **premier** critère de mutation (mutant1 sur notre fichier) est un changement d'opérateurs relationnels à la ligne 45 du fichier BellmanFord.java, remplaçant '<' par '<=' passant de la ligne

```
if (dist[u-65] != Integer.MAX_VALUE && dist[u-65] + weight < dist[v-65])
```

à la ligne `if (dist[u-65] != Integer.MAX_VALUE && dist[u-65] + weight <= dist[v-65])`

Résultats des tests avec ce mutant :

- question 3 : 50%
- question 4 : 71.4%
- question 5 : 100%

soit aucun changement sur le comportement, le mutant survit.

Le **second** critère de mutation est un changement d'opérateurs arithmétiques à la ligne 46 du fichier BellmanFord.java, remplaçant '+' par '-' passant de la ligne

```
dist[v-65] = dist[u-65] + weight;
```

à la ligne `dist[v-65] = dist[u-65] - weight;`

Résultats des tests avec ce mutant :

- question 3 : 50%
- question 4 : 28.6%
- question 5 : 33.3%

trop de changement de comportement, le mutant est tué.

Le **troisième** critère de mutation est un changement de variable à la ligne 46 du fichier BellmanFord.java, remplaçant 'v' par 'u' passant de la ligne

```
dist[v-65] = dist[u-65] + weight;
```

à la ligne `dist[u-65] = dist[u-65] + weight;`

Résultats des tests avec ce mutant :

- question 3 : 50%
- question 4 : 0%
- question 5 : 33.3%

trop de changement de comportement, le mutant est tué.