Rapport INFO-F-203

Algorithmique II

Les bons comptes font les bons amis !

Par

Mourad Akandouch

Mounir Hafif

19 décembre 2016

Table des matières

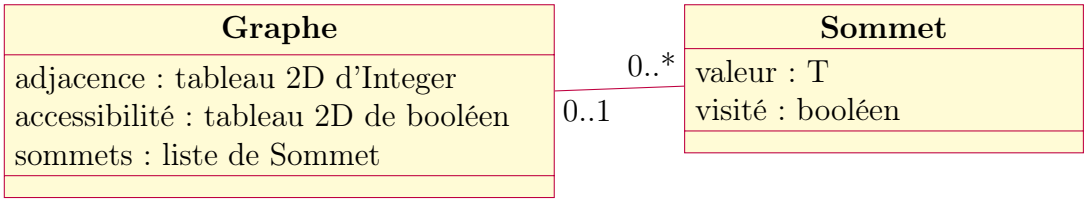
1 Introduction

Dans le but de concrétiser nos connaissances en algorithmique dans le domaine des graphes, il nous a été demandé de réaliser un petit projet se basant sur un réseau de dettes entre amis. Plusieurs exercices nous ont été proposés et nous avons dû en choisir au moins trois parmi ceux-ci (dont deux nous sont imposés). Le choix du langage nous a été laissé au choix, nous avons donc opté pour Java.

Pour bien comprendre comment nos algorithmes fonctionnent, il est essentiel d'exposer la structure de notre projet :

Pour des raisons d'optimisation, nous avons opté pour une représentation du graphe sous forme d'une classe `Graphe` contenant la matrice d'adjacence et d'accessibilité de celui-ci.

Plus concrètement, notre structure est définie ainsi :



L'absence d'arc menant d'un sommet i à un sommet j est indiquée par un `Integer` de valeur `null` dans la matrice d'adjacence à l'indice ij. Si l'élément à l'indice ij est différent de `null`, il s'agit donc d'une présence d'arc indiquant le montant de la dette.

Par exemple, soit la matrice d'accessibilité M où les sommets sont identifiés par les indices des lignes/colonnes :

$$M = \begin{pmatrix} \texttt{null} & 42 \\ 0 & \texttt{null} \end{pmatrix} \tag{1}$$

M nous indique qu'il n'y a pas d'arc menant du sommet d'indice 0 à lui-même. Pareil pour le sommet d'indice 1, M nous indique également que le sommet 0 a une dette de 42 envers celui dont l'indice est 1 et enfin, le sommet d'indice 1 *avait* une dette envers celui d'indice 0. Nous avons opté pour un `Integer` et non un `int` car nous considérons également les dettes de valeur négatives (qui, en soi, n'est qu'une dette mais de sens inverse).

La matrice d'accessibilité (générée dans le constructeur de `Graphe` tel que proposé par l'algorithme de *Roy-Warshall*) nous est utile dans plusieurs cas. Par exemple, pour identifier tous les nœuds possédant un cycle, il suffit d'itérer sur la diagonale de cette matrice et repérer quelles éléments sont à `true`. D'autres structures de données sont utilisées mais elles seront expliquées ci-dessous aux sections adéquates.

Enfin, il est également à noter que la liste sommets qui est montré dans le pseudo-diagramme de classe ci-dessus (de la classe `Graphe`) référence l'ensemble des sommets dans l'ordre dans lequel ils sont apparus dans le fichier texte à parser.

Ainsi, l'indice des matrices correspondent exactement à l'indice du sommet qu'on considère. Par exemple, $M_{2,3}$ indique s'il y a un arc menant du sommet d'indice 2 à celui d'indice 3 (de la liste `sommets`).

Concernant l'exécution de notre programme, un makefile est mis à disposition afin de compiler le code source. Nous avons conscience qu'il est des mauvaises pratique que de ne pas créer de package java mais ce projet était minimaliste et ne nécessitait donc pas d'en faire autant (à mon sens).

Aussi, après la compilation, deux modes d'exécutions vous sont fournis :

Soit vous exécutez sans argument la commande :

```
java Main
```

et un petit programme interactif vous demandera de choisir quel graphe voulez-vous tester. Leur numéro sont affectés en accord avec l'annexe de ce rapport.

Soit vous passez un argument à l'application :

```
java Main <path_to_file>
```

et elle se chargera de lire le fichier passé en argument et d'afficher les résultats sur la sortie standard (en accord avec ce qui est demandé dans le PDF).

Cette brève introduction étant faite, passons maintenant au vif du sujet.

2 Simplification des dettes

2.1 L'algorithme

Sachant que tous les cycles dirigés peuvent être simplifiés (cf. énoncé du projet point 2.1), j'ai commencé par essayer de comprendre le principe de réduction de dettes. Grâce à ces différents essais, j'ai pu construire la succession des étapes à effectuer pour obtenir le bon résultat.

Voici une brève description des étapes de cet algorithme :

1. Rechercher tous les cycles qui se trouvent dans le graphe donné.
2. Effectuer un tri croissant de ces cycles trouvés sur base de leur ordre.
3. Réduire les dettes de chaque cycle dans l'ordre résultant du tri de l'étape 2.

> **Note :** Lors de l'étape de réduction de dette, un cycle contenant un arc dont la dette est de 0 est considéré comme irréductible.

Voici un pseudo-code décrivant la résolution de cet exercice :

```
Entrée : Graphe g
Sortie : -
    cheminCycle ← liste vide;
    mapListeCycle ← map vide;
    pour chaque sommet faire
    |   cheminCycle.ajouter(sommetCourant);
    |   pour chaque sommet atteignable depuis sommetCourant faire
    |   |   identifierCycles(sommetAtteignable, cheminCycle, mapListeCycle,
    |   |    g);
    |   fin
    |   cheminCycle.supprimer(sommetCourant);
    fin
    pour chaque cycle dans g.mapListeCycle faire
    |   reduireCycle (cycleCourant, g);
    fin
```

Algorithme 1 : réductionDette, algorithme initialisant la recherche de cycle récursive et réduisant les dettes des cycles trouvés

Note : la matrice d'accessibilité permet de lancer la recherche récursive de cycle uniquement sur les noeuds pertinents (possédant un cycle).

```
Entrée : Graphe g, noeudDépart, mapListeCycle
Sortie : -
    cheminCycle.ajouter(noeudDépart);
    g.marquerSommetVisité(noeudDépart);
    si noeudDépart == cheminCycle.obtenirElementPosition(0) alors
    |   mapListeCycle.ajouter(cheminCycle);
    fin
    sinon
    |   pour chaque sommet atteignable depuis noeudDépart faire
    |   |   si noeudDépart n'est pas marqué comme visité alors
    |   |   |   identifierCycles(noeudDépart, cheminCycle, mapListeCycle,
    |   |   |    g);
    |   |   fin
    |   fin
    fin
    g.marquerSommetNonVisité(noeudDépart);
    cheminCycle.supprimer(noeudDépart);
```

Algorithme 2 : identifierCycles, algorithme de recherche de cycle

2.2 Tests effectués

Le premier test a été effectué sur l'exemple donné dans le PDF du projet. Une fois la vérification du succès de ce test, je me suis concentré sur la vérification de la cohérence des cycles renvoyés par mon algorithme ainsi que l'ordre dans lequel ils sont triés.

Une fois cette vérification effectuée, j'ai choisi quelques graphes de test pour m'assurer du bon fonctionnement de cet algorithme. Ces dits graphes sont annexés à ce rapport et sont les suivants : graphe 1, graphe 2 et le graphe 5.

3 Identification des communautés

Cet exercice nous demandait d'identifier, sur base d'un graphe fourni en paramètre, l'ensemble des communautés de celui-ci. En termes de graphes, cela revenait à simplement identifier et retourner une liste contenant l'ensemble de ses sous-graphes. Ce

```
Entrée : Graphe g, cheminCycle
Sortie : -
    plusPetitDette : Entier;
    plusPetitDette ← valeur dette arc départ de cheminCycle ;
    pour tous les autres arcs de cheminCycle faire
    |  si detteArcCourante == 0 alors
    |  |  inutile de réduire, quitter la function;
    |  fin
    |  si detteArcCourante < plusPetitDette alors
    |  |  plusPetitDette ← detteArcCourante ;
    |  fin
    fin
    pour tous les arcs de cheminCycle faire
    |  detteArcCourante ← detteArcCourante- plusPetitDette ;
    fin
```

Algorithme 3 : réduireDetteCycle, algorithme réduisant la dette de chaque cycle

n'est pas un problème d'une grande difficulté dès lors qu'on choisit une approche matricielle du problème plutôt que l'approche classique consistant en des arcs pointant vers des noeuds, etc. et ce, récursivement.

Aussi, puisque nous devions recevoir un graphe en paramètre, nous avons décidé de rendre la méthode statique pour garder une certaine cohérence OO. En effet, il serait discutable de faire de cet exercice une méthode d'instance puisqu'elle ne se base que sur le graphe fourni et n'a rien à voir avec son instance. Donc, nous l'avons rendue statique.

3.1 L'algorithme

L'idée qui m'est venue en tête pour identifier toutes les communautés (autrement dit, tous les sous-graphes du graphe) était de simplement parcourir ma liste des sommets et de démarrer un parcours de graphe sur chacun d'eux. Après chaque parcours, je ne remets **pas** à `false` le fait d'avoir marqué un sommet comme visité.

Aussi, je parcours le graphe de manière non-dirigée. Les arcs deviennent des arêtes dans cet algorithme. Plus concrètement, cela se traduit par le fait que pour aller du sommet i au sommet j, il suffit simplement que M_{ij} **ou** M_{ji} soit différent de `null`.

L'algorithme se divise en deux fonctions. Une me permettant d'initialiser la récursivité et une autre pour parcourir le graphe. Le parcours celui-ci ne fait rien d'autre qu'ajouter le sommet courant à la communauté s'il n'a pas déjà été visité, puis relance le parcours pour chacun des voisins du sommet courant. Un parcours classique en somme.

```
Entrée : Graphe g
Sortie  : liste de Communauté (une Communauté est une liste de String)
    listeCommunauté ← liste vide;
    communauté ← liste vide;
    pour chaque sommet faire
    |   parcours(g, communauté, sommetCourant);
    |   si communauté n'est pas vide alors
    |   |   listeCommunauté.ajouter(copie de communauté);
    |   fin
    |   vider communauté ;
    fin
    reset visite de tous les sommets
```

Algorithme 4 : Algorithme initialisant la récursivité et ajoutant les communautés à la liste des communauté

```
Entrée : Graphe g, communauté, sommetCourant
Sortie  : -
    si sommetCourant non visité alors
    |   marquer sommetCourant comme visité;
    |   communauté.ajouter(sommetCourant);
    |   pour chaque sommet atteignable depuis sommetCourant faire
    |   |   parcours(g, communauté, sommet suivant);
    |   fin
    fin
```

Algorithme 5 : Algorithme récursif rajoutant dans la communauté tous les sommets atteignables depuis sommetCourant

3.2 Tests effectués

Les différents tests étaient assez faciles à réaliser pour cet exercice car il suffisait de créer un graphe à l'aide d'un programme fait pour, d'en réaliser le fichier texte cor-

respondant et voir si le nombre de communauté retourné par ma fonction concordait avec le nombre de sous-graphe que je voyais.

Par exemple, il est clair que le graphe 4 (voir annexe) ne comporte qu'une seule communauté (très soudée), pareil pour le 5.

4 Identification des hubs sociaux

Le but de cet exercice fut d'identifier les hubs sociaux d'un graphe fourni en paramètre avec une contrainte sur le nombre d'individus si celui-ci venait à disparaître. En d'autres termes, nous devions chercher les points d'articulations du graphe fourni dont la suppression entraîne la création de N sous-graphes d'ordre K au moins. ($N \in \mathbb{N}$, K est fourni en paramètre)

Encore une fois, étant donné que le graphe nous est fourni en paramètre et pour les mêmes raisons que mentionnées en section 3, la méthode a été déclarée statique.

4.1 L'algorithme

Concernant la résolution de cet exercice, l'algorithme est relativement simple : je commence par compter le nombre de communautés (cf. section 3) d'au moins K individus [1] grâce à la fonction qui identifie les communautés et je stocke ce nombre dans une variable (appelons-la σ).

Ensuite, j'itère sur chacun de mes sommets, pour chacun :

1. Je supprime le noeud courant (Concrètement, je le marque simplement comme déjà visité, il sera ainsi inexistant aux yeux de ma fonction qui identifie les communautés).
2. J'appelle la fonction qui identifie les communautés, je compte le nombre de communautés d'au moins K individus et si ce nombre est *strictement supérieur* à σ, alors ce noeud est ajouté dans une liste contenant les hubs sociaux.

   Note : Il n'est pas nécessaire de remettre le noeud courant à l'état non-visité après l'étape 2 car la fonction qui identifie les communautés s'en charge déjà (cf. dernière ligne de l'algorithme 4)

Le pseudo-code de l'algorithme plus bas décrit la résolution de cet exercice.

1. J'applique simplement un filtre grâce aux streams Java8

Entrée : Graphe g, K : le nombre d'individus minimum
Sortie : liste de String (nom des sommets étant hub sociaux)

```
listeHub ← liste vide;
listeCommunautés ← Graphe.identifierCommunauté(g);
KIndividus ← nombre de communautés d'au moins K individus dans
 listeCommunautés ;
pour chaque sommets faire
    marquer sommetCourant comme visité;
    listeCommunautés ← Graphe.identifierCommunauté(g);
    si KIndividus < nombre de communautés d'au moins K individus dans
     listeCommunautés alors
        listeHub.ajouter(sommetCourant);
    fin
fin
```

Algorithme 6 : Algorithme récursif retournant l'ensemble des hubs sociaux

4.2 Tests effectués

Pareil que pour la section 3, nos tests se sont surtout tournés vers le visuel : c'est-à-dire que nous avons créé des graphes (voir annexes) et avons testé les hubs sociaux pour $K = 1...5$ individus. Si l'algorithme concorde avec ce que nous voyons, alors c'est correct.

5 Identification du plus grand groupe d'amis

5.1 L'algorithme

Étant donné que je suis des cours de Master (dont le cours de Computability and complexity), je me suis rapidement rendu compte que cet exercice référence un problème bien connu qui est celui de trouver la 'clique maximum' d'un graphe. Une clique d'un graphe est, en théorie des graphes, un sous-ensemble des sommets de ce graphe dont le sous-graphe induit est complet. Une 'clique maximum' d'un graphe est une clique dont le cardinal est le plus grand (c'est-à-dire qu'elle possède le plus grand nombre de sommets).

En effectuant des recherches, j'ai trouvé plusieurs algorithmes qui résolvent cet exercice ; j'ai utilisé celui de Bron Kerbosch.

Ci-dessous (Algorithme 7), un pseudo-code décrivant la résolution de cet exercice :

Entrée : List R, P, X : Sommets
Sortie : L'une des cliques maximum

si *P et X sont tous les deux vides* **alors**
  Nous avons une nouvelle clique;
  si *nbSommets dans cette clique > ancienne clique sauvée* **alors**
    Sauver cette nouvelle clique comme la plus grande rencontrés;
  fin
fin
pour *chaque sommet $s \in P$* **faire**
  `BronKerbosch(`$R \cup s$, $P \cap N(s)$, $X \cap N(s)$`)`;
  P $\leftarrow P \setminus s$;
  X $\leftarrow X \cup s$;
fin

Algorithme 7 : BronKerbosch, algorithme récursif permettant de trouver tous les cliques d'un graphe

Lors du premier appel R et X sont vides, et P contient tous les sommets du graphe. R est un résultat temporaire, P l'ensemble des candidats possibles et X l'ensemble exclu. $N(s)$ indique les voisins du sommet s.

L'algorithme peut être interprété comme suit : Choisir un sommet s de P à développer. Ajouter s à R et enlever ses non-voisins de P et X. Puis choisir un autre sommet du nouvel ensemble et répéter le processus. Continuer jusqu'à ce que P soit vide. Une fois que P est vide, si X est vide, signaler le contenu de R comme une nouvelle clique maximale (si ce n'est pas le cas, alors R contient un sous-ensemble d'une clique déjà trouvée). Maintenant, revenir au dernier sommet choisi et restaurer P, R et X comme avant le choix, retirer le sommet s de P et l'ajouter à X, puis développer le sommet suivant.

Dans notre cas, si P et X sont vides, on vérifie également que le nombre de sommets composants la clique trouvée est plus grande que l'ancienne. si la réponse s'avère positive, on sauve la nouvelle clique (groupe d'amis), sinon on continue.

5.2 Tests effectués

Comme pour la section 1, j'ai d'abord testé l'algorithme sur l'exemple du graphe donné dans le PDF. Une fois cette vérification effectuée, les tests suivants se sont déroulés sur certains des graphes que nous avons crée (voir annexes). les résultats de ces tests ont été vérifiés visuellement.

6 Conclusion

Le projet ne fut pas d'une si grande difficulté, nous avions déjà vu l'entièreté de la matière durant notre bachelier. Ainsi, nous n'avons pas rencontré de problèmes lors du projet.

Concernant la répartition des tâches, je (Mourad) me suis chargé de l'identification des communautés (exercice 2.2) ainsi que de l'identification des hubs sociaux (exercice 2.3). Mounir s'est chargé des deux restants cités plus haut (exercices 2.1 et 2.4). Nous nous sommes donc réparti le projet en 50% chacun.

7 Annexes

Ci-dessous, vous retrouverez l'ensemble des graphes sur lesquels nous avons testés nos algorithmes. Vous constaterez qu'ils sont tous assez différent et proposent des cas limites pour nos algorithmes.

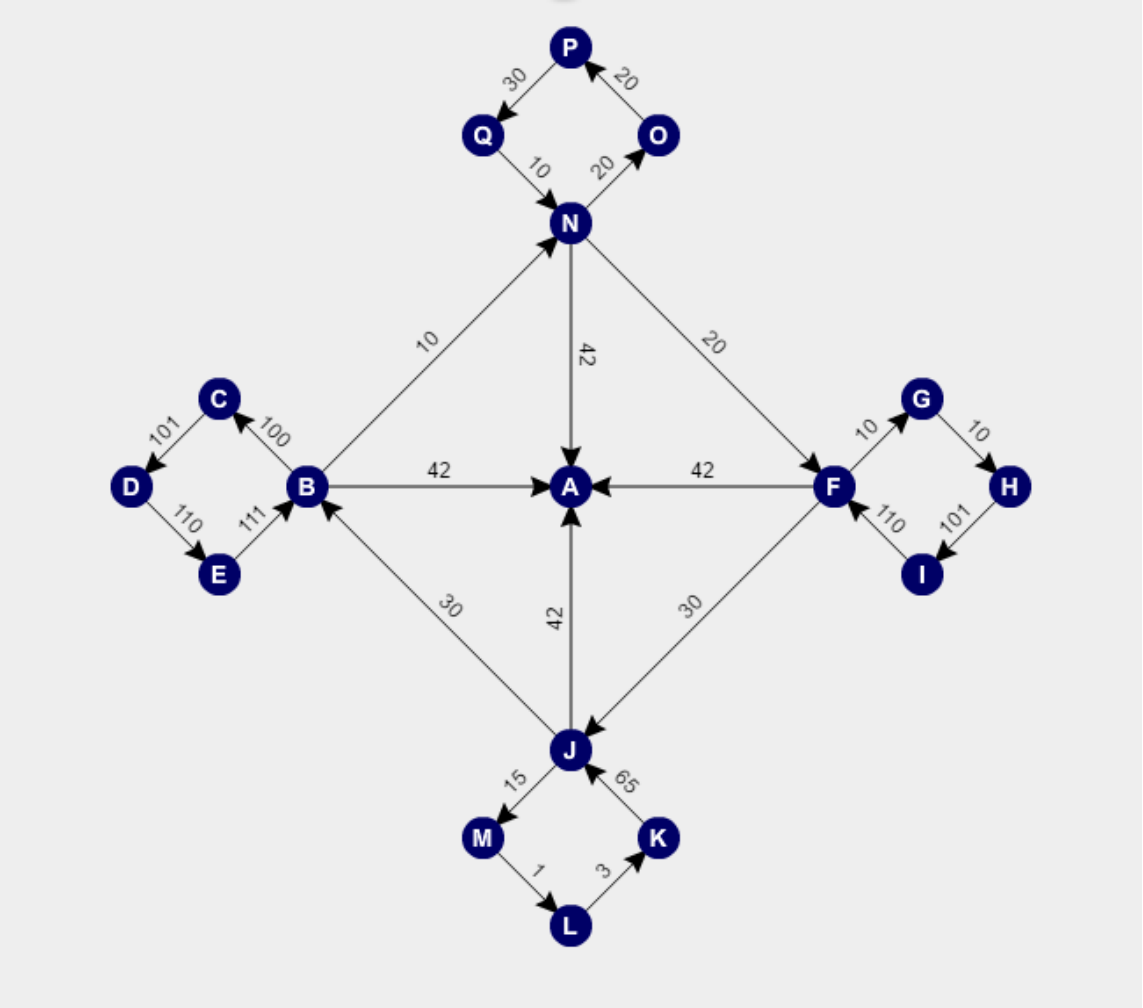


FIGURE 1 – Graphe représentant le test numéro 1 dans notre programme

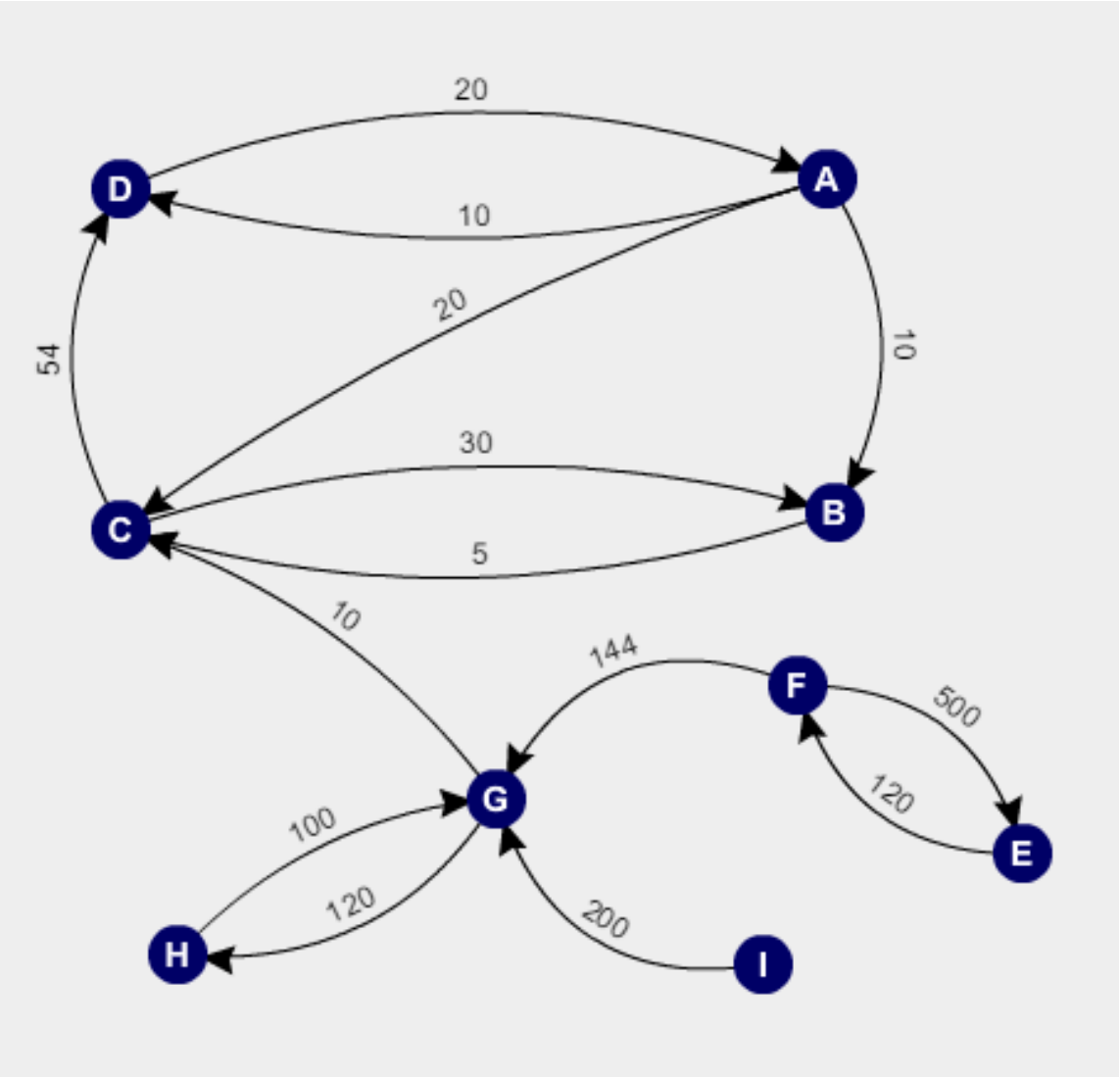


FIGURE 2 – Graphe représentant le test numéro 2 dans notre programme

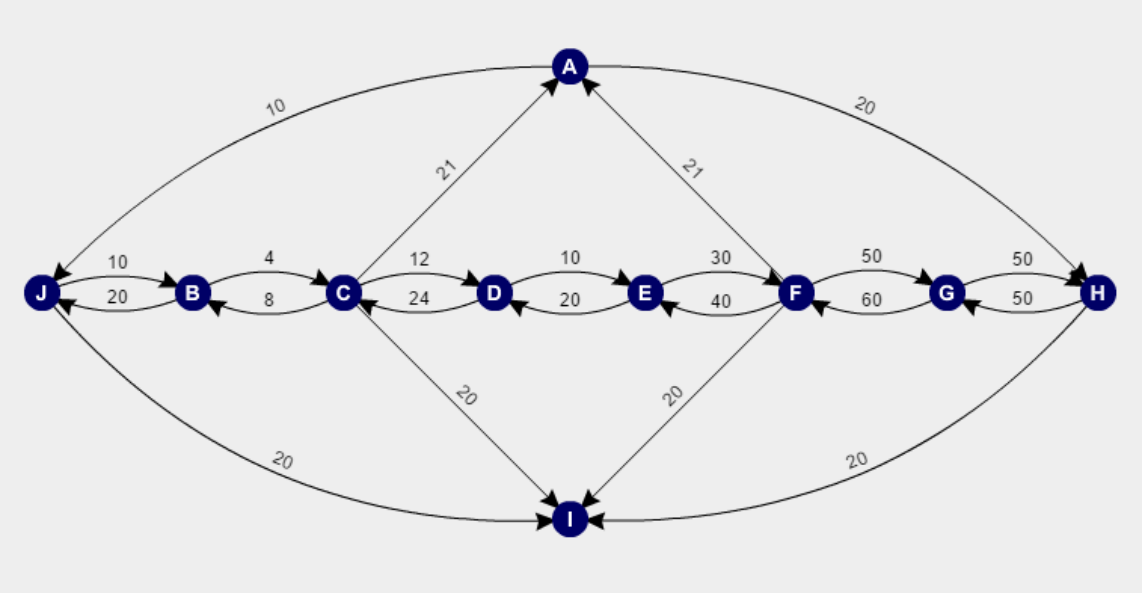


FIGURE 3 – Graphe représentant le test numéro 3 dans notre programme

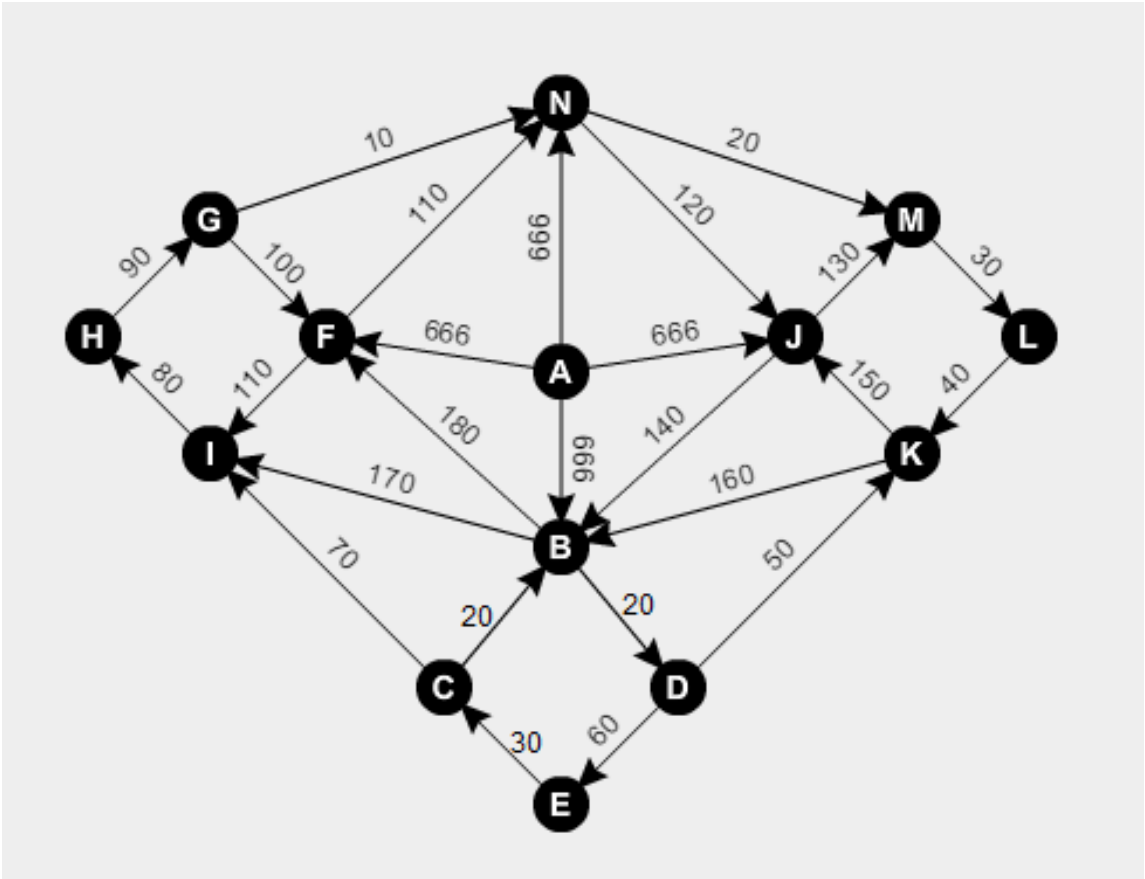


FIGURE 4 – Graphe représentant le test numéro 4 dans notre programme

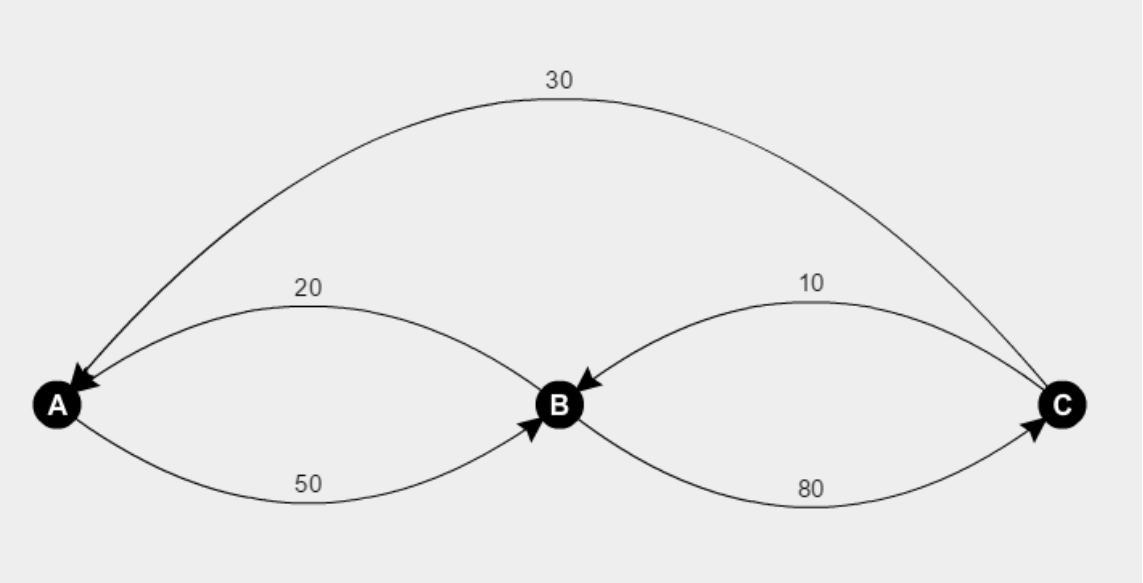


FIGURE 5 – Graphe représentant le test numéro 5 dans notre programme

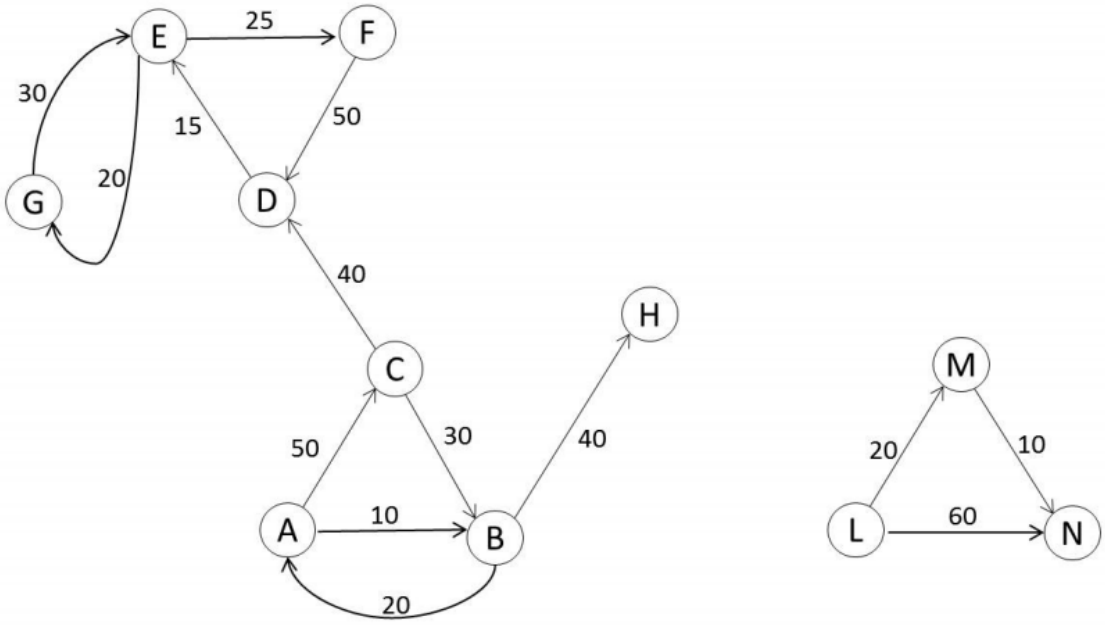


FIGURE 6 – Graphe test numéro 6 dans notre programme, celui du projet