

Projet : Les bons comptes font les bons amis !

[Cours : INFO-F-203]

JACOBS Alexandre & INNOCENT Antoine
BA2 Informatique

Décembre 2016

Table des matières

1	Introduction	2
2	Structure	2
3	Algorithmes	2
3.1	Simplification des dettes	2
3.2	Identification des communautés	5
3.3	Identification du plus grand groupe d'amis	6
4	Test	8
5	Conclusion	8
6	Annexe	9
6.1	Exemple d'exécution en ligne de commande : Simplification de dettes	9
6.2	Exemple d'exécution en ligne de commande : Recherche des communautés	9
6.3	Exemple d'exécution en ligne de commande : Recherche du plus grand groupe d'amis	9
7	Bibliographie	10

1 Introduction

Dans le cadre de notre projet d'algorithmique, nous devons implémenter une succession d'algorithmes testant les connaissances et concepts vus en cours ainsi que notre capacité à travailler de manière autonome.

En effet, ce projet nous projette dans les bureaux d'une jeune start-up (Radino) qui a pour but de faciliter les dépenses et dettes entre amis via une application. Cet objectif est atteint à travers l'implémentation de trois algorithmes codés intégralement en Python3.

En effet, nous devons implémenter des solutions d'algorithmes pour les deux problèmes suivants et aussi devons en plus implémenter au moins un autre algorithme au choix répondant à d'autres problèmes :

- Simplification de dettes entre amis (obligatoire).
- Identification des communautés (obligatoire).
- Identification des hubs sociaux (facultatif).
- Identification du plus grands groupes d'amis (facultatif).
- Simplification supplémentaire (facultatif).

Nous avons choisi d'implémenter l'algorithme supplémentaire : « Identification du plus grands groupes d'amis ».

2 Structure

La structure principale présentée par le sujet est un graphe dirigé. Nous avons donc procédé à son implémentation en créant une structure de 3 classes distinctes :

- DirectedGraph : représentation d'un graphe dirigé.
- UndirectedGraph qui illustre un graphe non dirigé. Ce qui va notamment servir, par la suite, à l'identification des différentes communautés ainsi que la détection des plus grands groupes d'amis.
- Graph : qui reprend la structure de base du graphe et qui joue le rôle de classe parente des classes ci dessus pour des méthodes communes aux deux classes filles.

Afin de mieux manipuler notre graphe, nous avons opté pour un dictionnaire contenant, lui même, un autre dictionnaire. Ce qui nous permet d'y stocker les différents noeuds ainsi que le poid de leurs arêtes (les dettes). Quant aux noeuds, nous avons choisi de les stocker dans un ensemble.

Par ailleurs, nous avons également implémenté un fichier « main.py » permettant de faire des exécutions des différents algorithmes avec divers fichiers texte représentant des graphes. Aussi, en plus de cela nous avons aussi implémenté une fichier contenant « test_graph.py », ce dernier fera l'objet d'un paragraphe plus loin.

3 Algorithmes

3.1 Simplification des dettes

Le graphe donné laisse place à une simplification consistant à réajuster les dettes dans un cycle. Effectivement, afin d'implémenter cet algorithme, on a du faire une détection de tous les cycles du graphe et par la suite une simplification des dettes au sein du cycle.

La détection de cycle consiste en une recherche à partir d'un noeud en particulier. D'itérer sur ses successeurs. Dès le moment où nous revenons sur un noeud déjà visité nous détectons un cycle. Nous devons dès lors sélectionner tous les membres qui composent le cycle, trouver l'indice du noeud qui le démarre et l'ajouter en partant de celui-ci. Pour ce faire nous utilisons une méthode de backtracking.

Algorithm 1 find_all_cycles

```
for node in nodes do
  if node is visited then
    Cycle found
  else
    visited  $\leftarrow$  node
    for Connected_node in Connected_nodes do
      find_all_cycles(Connected_node)
    end for
    visited  $\leftarrow$  node
  end if
end for = 0
```

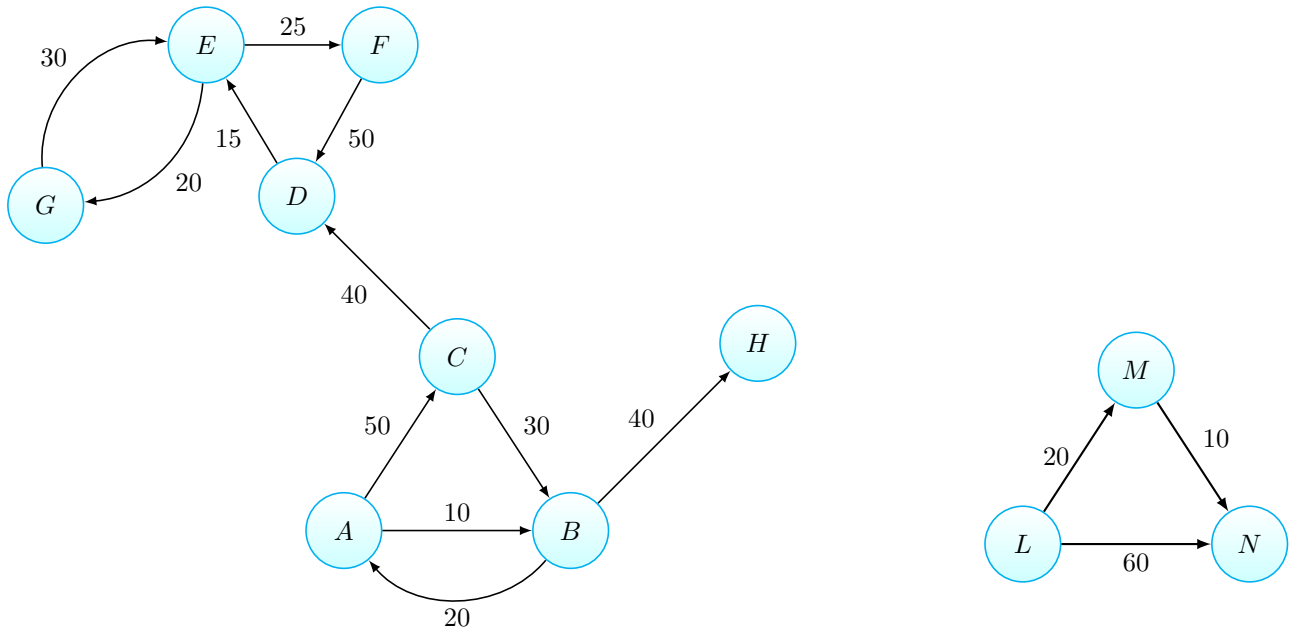
Une fois les cycles trouvés, nous avons du faire la simplification des dettes au sein de chaque cycle, il a fallu itérer sur chacun d'entre eux (en partant des plus petits cycles) de manière à soustraire la plus petite dette aux différentes branches de celui-ci. Après la réalisation de cette manipulation, il ne restait plus qu'à remplacer les anciennes dettes de chaque branche par leurs nouvelles valeurs. Voici en pseudo-code, suivi d'un exemple d'exécution pour aider à la compréhension :

Algorithm 2 _resolve_cycle

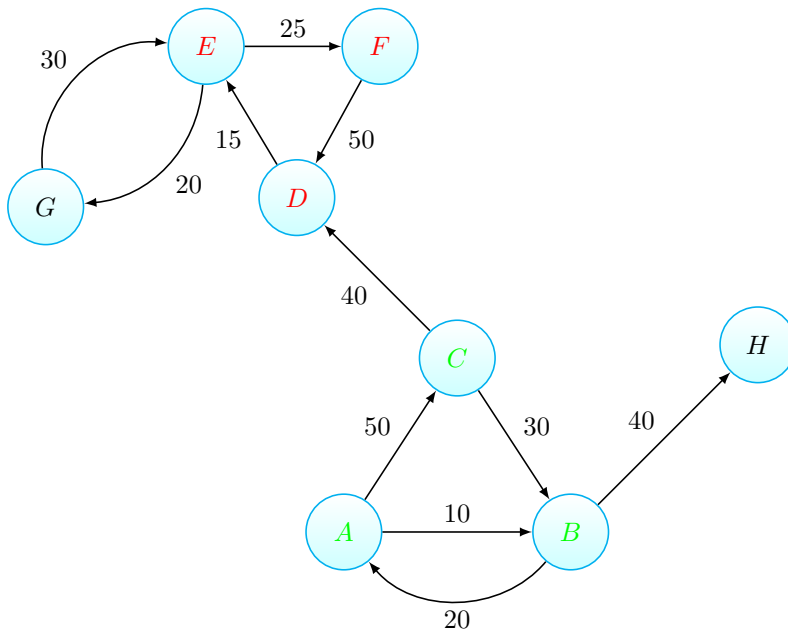
```
for index,node in cycles do
  if index is not lastcycle then
    tempWeight  $\leftarrow$  weight(node, nextnodeofcycle)
  else
    tempWeight  $\leftarrow$  weight(node, firstnodeofcycle)
  end if
  Weights  $\leftarrow$  tempweight
end for
Min_Weight  $\leftarrow$  minimum weight in Weights
for Weight in Weights do
  Weights  $\leftarrow$  weight - Min_Weight
end for
for index,node in cycles do
  if index is not lastcycle then
    Weight of node and next node = index in Weights
  else
    Weight of node and first node = index in Weights
  end if
end for = 0
```

Exemples d'exécution :

Pour cet exemple, nous partirons du graphe de base de l'énoncer.



Première étape chercher les différents cycles :

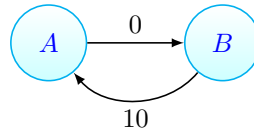
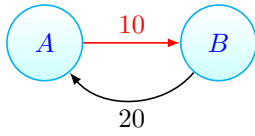


Ainsi que E,G et A,B :

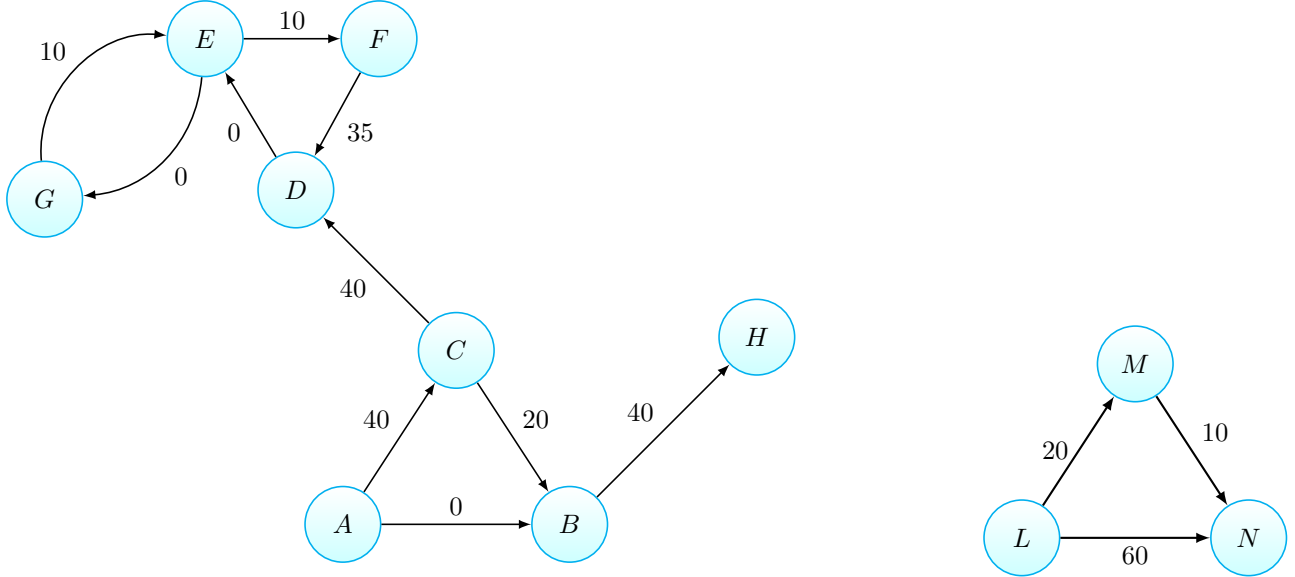


Une fois que nous avons les cycles, nous allons chercher la plus petite dette de chaque cycle (en commençant par le plus petit), et nous la soustrayons à toutes les dettes du cycle.

Pour illustrer ce concept prenons le simple cycle A,B :



Une fois toutes les simplifications achevées, nous obtenons ce résultat final :



3.2 Identification des communautés

Une communauté est définie comme étant un ensemble de personnes connectées par leurs dettes actuelles et passées, cela correspond ici à un sous-graphe du graphe principal, plus concrètement appelé composant connexe.

Pour cette partie, l'implémentation d'un graphe dirigé ne nous permettait pas d'atteindre notre objectif de trouver les différentes communautés du graphe. C'est pour cela que nous avons implémenté une classe UndirectedGraph. Nous y avons remédié en transformant le graphe dirigé de base en un graphe non dirigé, gardant la même structure mais en ajoutant simplement les arêtes retour pour chaque noeud avec ses successeurs.

Pour détecter les différentes composantes connexes du graphe de base, nous avons dû implémenter un algorithme de parcours en profondeur généralisé (Depth First Search) « find_communities ». Celui-ci faisant appel à un simple DFS normal, à partir d'un sommet du graphe non marqué, est chargé de marquer tout les sommets rencontrés et recommencer jusqu'à ce qu'il ne reste plus aucun sommets non marqués.

Dès lors après que tous les noeuds aient été marqué, nous aurons obtenu les différents composants connexes du graphe. Voici en pseudo-code l'algorithme que nous avons implémenté.

Algorithm 3 find_communities

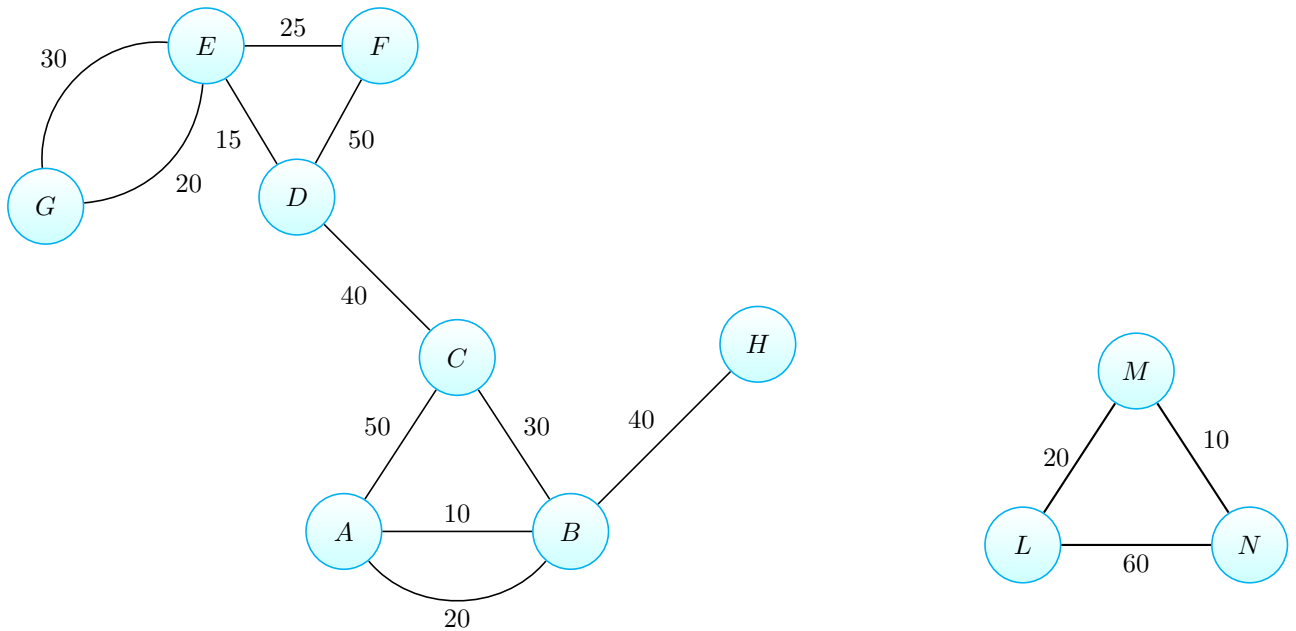
```

for node n in nodes do
  if node not visited then
    communities ← dfs(n)
  end if
end for

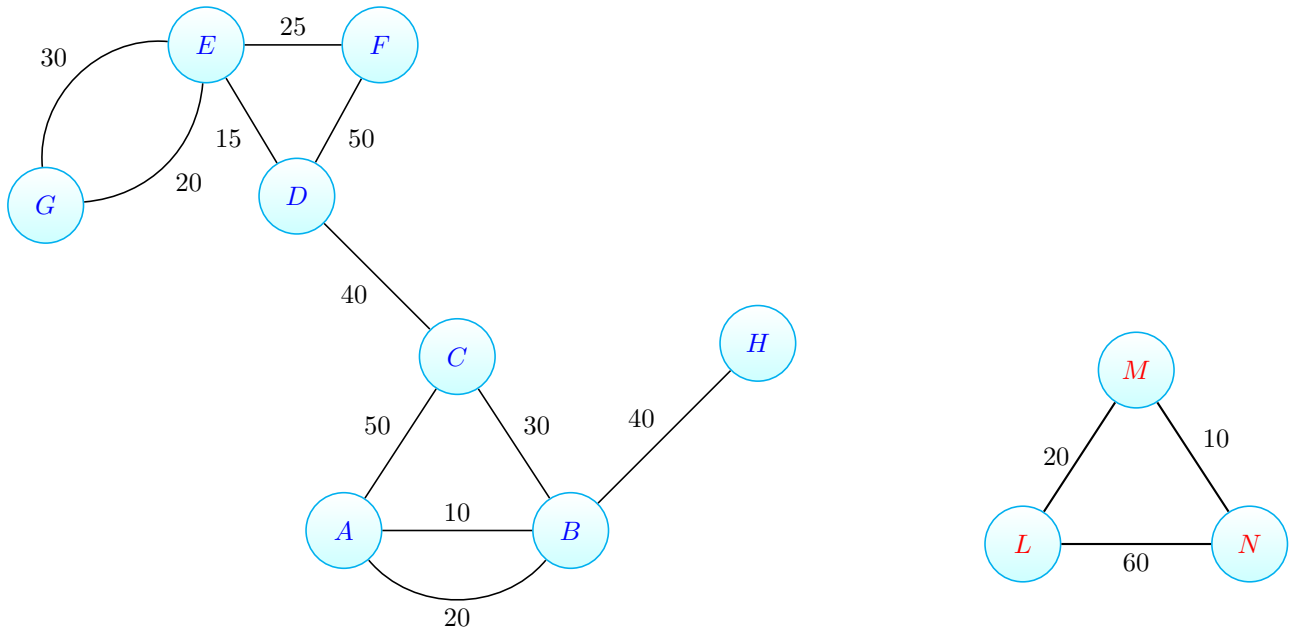
```

Exemples d'exécution :

Pour cet exemple, nous partirons du graphe de base de l'énoncer. Transformons celui ci en un graphe non dirigé.



Nous parcourons notre graphe à l'aide d'un parcours en profondeur généralisé expliqué ci dessus. Nous obtenons ainsi ces 2 communautés :



3.3 Identification du plus grand groupe d'amis

Un groupe d'amis est un ensemble de personnes tel que chacune de ces personnes a une dette passée ou présente avec chacun des membres du groupe (une dette dans un sens ou dans l'autre). En d'autres mots, un groupe d'amis est l'ensemble des noeuds d'un sous graphe non-dirigé complet du graphe des dettes. Notre algorithme, après un appel à celui-ci, renvoie le plus grand groupe d'amis (s'il en existe plusieurs de différentes tailles), ou n'importe lequel si tous les groupes d'amis ont exactement la même taille.

Toujours en passant par un graphe non dirigé, nous avons implémenté un problème déjà rencontré en cours : la détection de cycle dont tous les noeuds sont inter-connectés entre eux.

Par ailleurs après de plus amples recherches sur internet, nous avons remarque que cela constituait un problème

connu sous le nom de : « problème de la plus grande clique ». Une clique est un sous ensemble de sommets qui sont tous adjacents entre eux et forment donc un sous-graphe complet.

Pour cela, nous sommes passés par deux fonctions, « find_all_cycles » qui nous ont permis de trouver tout les cycles du graphe et de les stocker dans un ensemble. Elle même faisant un appel à une autre fonction « visited » qui part du noeud reçu en argument et trouve le cycle dans lequel il appartient (si celui ci est existant) . Notre fonction de départ itère dans tous les noeuds du graphe pour nous permettre d’avoir la totalité des cycles. Voici son pseudo code :

Algorithm 4 find_all_cycles

```

for node n in nodes do
  cycle_detected ← visited(node)
end for
=0

```

La seconde fonction « find_highest_friend_group » est chargé d’itérer dans tout les cycles trouvés (trié par longueur décroissante) afin de déterminer si celui-ci constitue un sous graphe complet. La manière dont nous déterminons si le cycle est complet ou non, est fait grâce à une itération sur les différents noeuds du cycle, dont nous lui prenons ses successeurs et vérifions si le cycle (sans le noeud courant) est un sous ensemble de ses successeurs. Ensuite, afin de déterminer si cela est un sous ensemble, nous utilisons une méthode « built-in » des set qui est « issubset ». Dès que nous obtenons un sous graphe complet, on arrête la recherche et retournons celui-ci. Ci dessous, le pseudo code du « problème de la plus grande clique » :

Algorithm 5 find_highest_friend_group

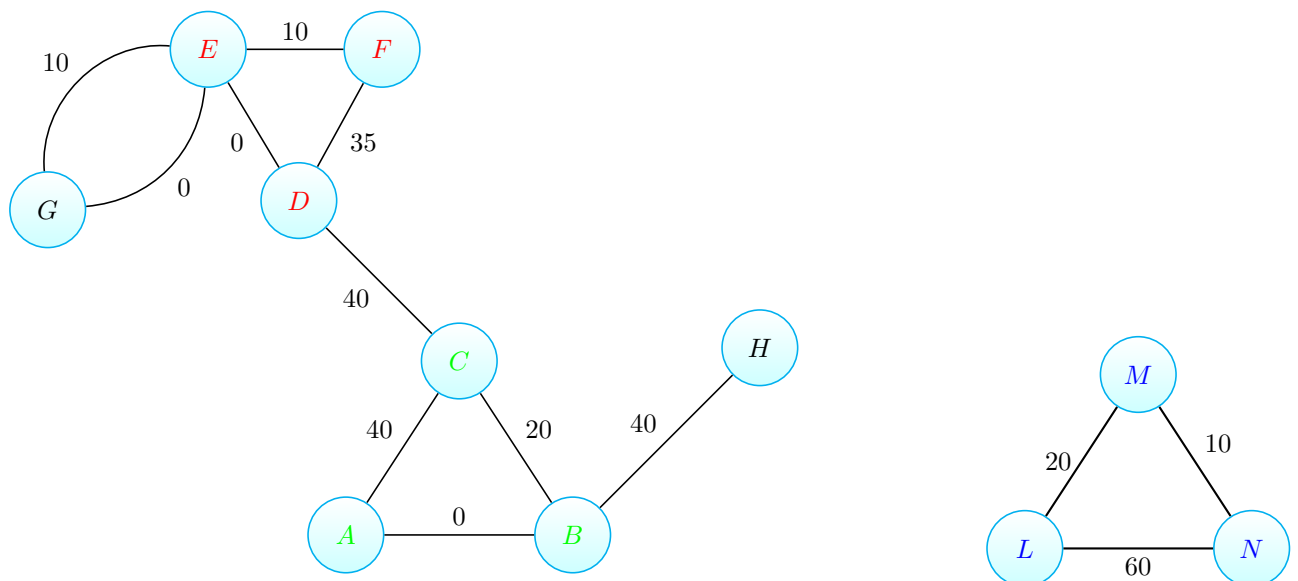
```

for cycle in all_sorted_cycles do
  for node in cycle do
    if neighboursnodes is a subset of successors node then
      return cycle
    end if
  end for
end for
=0

```

Exemples d’exécution :

Dans notre cas, nous avons 3 cycles contenant chacun 3 individus. La fonction en question va donc renvoyer un des cycles suivants (rouge, vert, bleu) :



4 Test

Afin d'assurer le bon fonctionnement des structures principales ainsi que des algorithmes précédents, nous avons implémenté un fichier « test_graph.py » faisant des tests unitaires sur le graphe dirigé et sur le graphe non dirigé. Ce fichier de test contient 2 classes test pour pouvoir faire les tests sur les graphes dirigés et non dirigés. Cela nous permettait au fur et à mesure de faire passer une batterie de tests à notre code en cours d'élaboration/modification, si celui ci correspondait et s'exécutait bien nos attentes, jusqu'à l'obtention du code final pour résoudre le problème de base.

Pour l'élaboration des tests unitaires qui fut une grande première pour nous, nous avons dû faire quelques recherches sur internet pour voir comment les élaborer, aider à comprendre nos différentes erreurs rencontrées lors de la création de ceux ci. Nous testions tous nos algorithmes avec le fichier de base disponible sur l'UV (Université Virtuel de l'ULB). Voici un exemple de comment se passe une exécution des tests unitaires :

```
root@LAPTOP-8RUBB3RB:/mnt/c/Users/Alexandre/Dropbox/BA2 Informatiques/Projet-INFOF203# python3 test_graph.py -v
test_detect_and_resolve_all_cycles (__main__.TestGraph) ... ok
test_find_all_cycles (__main__.TestGraph) ... ok
test_find_communities (__main__.TestGraph) ... ok
test_find_highest_friend_group (__main__.TestGraph) ... ok
test_get_weight (__main__.TestGraph) ... ok
test_iter_edges (__main__.TestGraph) ... ok
test_load_from_filename (__main__.TestGraph) ... ok
test_add_edge (__main__.TestGraphUndirected) ... ok
test_create_from_directed_graph (__main__.TestGraphUndirected) ... ok
test_find_all_cycles (__main__.TestGraphUndirected) ... ok
test_find_highest_friend_group (__main__.TestGraphUndirected) ... ok
-----
Ran 11 tests in 0.013s
OK
```

5 Conclusion

Pour finir, nous avons été confronté à plusieurs défis. Effectivement, la première difficulté rencontrée était la simplification de dettes. Comme celle ci n'avait aucune référence dans le cours, nous avons dû dépasser le cadre de celui ci. L'exemple d'algorithme du cours, nous ayant pas trop servi nous avons dû faire plusieurs recherches et cela nous à demandé beaucoup d'essais et d'erreurs. La deuxième difficulté étant de faire de bon test vérifiant le bon fonctionnement des différents algorithmes. Enfin la dernière, et loin d'être une difficulté des plus sérieuse la réalisation de ce rapport reprenant les explications du projet. Par ailleurs quant à la répartition des tâches, pour la majeure partie du code, Alexandre avait le volant avec moi dans le siège passager et inversement pour la rédaction du rapport. Ainsi pour conclure, ce projet nous aura permis d'intégrer différents concepts algorithmiques vus en cours et de les implémenter par rapport à un problème précis.

6 Annexe

6.1 Exemple d'exécution en ligne de commande : Simplification de dettes

```
Voici le graphe avant simplification
A B 10
A C 50
F D 50
M N 10
C B 30
C D 40
D E 15
B A 20
B H 40
L N 60
L M 20
G E 30
E F 25
E G 20

Voici le graphe après simplification s'il y en a eu
A B 0
A C 40
F D 35
M N 10
C B 20
C D 40
D E 0
B A 0
B H 40
L N 60
L M 20
G E 10
E F 10
E G 0
```

6.2 Exemple d'exécution en ligne de commande : Recherche des communautés

```
Voici les coummunautés:
L,M,N
A,D,C,E,B,F,G,H
```

6.3 Exemple d'exécution en ligne de commande : Recherche du plus grand groupe d'amis

```
Voici le plus grand groupe d'amis
('L', 'M', 'N')
```

7 Bibliographie

Références

- [1] BERNARD FORTZ, OLIVIER MARKOWITCH, FRÉDÉRIC SERVAIS. " *Algorithmique 2 (INFO - F - 203)* ". ULB, 2016.
- [2] THOMAS CORMEN ET AL. . " *Introduction à l'algorithmique, Dunod* ". ULB, 2016.
- [3] ROBERT SEDGEWICK . " *Algorithms in C++, 2d edition, Addison Wesley* ". ULB, 2016.
- [4] ROBERT SEDGEWICK . " *Algorithms en langage C, Dunod* ". ULB, 2016.
- [5] GITHUB.COM/NETWORKX/NETWORKX . " *Site ayant permis de trouver une structure adéquate* ". ULB, 2016.
- [6] FR.WIKIPEDIA.ORG " *Théorie des graphes* ". ULB, 2016
- [7] DOCS.PYTHON.ORG/3.6/ " *Documentation Python* ". ULB, 2016
- [8] FR.WIKIPEDIA.ORG " *Clique (théorie des graphes)* ". ULB, 2016