Grands Réseaux d'Interaction
TP n° 3 et 4 : Modularité

- Ce TP est à rendre sur Moodle en deux parties, pour le **19 novembre** (partie I) et le **3 décembre** (partie II).
- Le plagiat conduit à avoir 0
- Ce TP est à faire avec le groupe (de 1 ou 2 étudiants) auquel vous êtes affecté. Tout changement de groupe doit être demandé par mail et motivé.

I) Problématique

a) Clustering

Un *clustering* d'un graphe est une partition de ses sommets en ensembles (appelés **communautés** ou **clusters**) qui vérifient la propriété (informelle) suivante :
- Chaque cluster a peu de liens externes (vers les autres)
- Chaque cluster a beaucoup de liens internes

Il existe toute une familles d'*algorithmes de clustering*. Ils se distinguent par
- la définition de la quantité à optimiser
- la méthode choisie : séparatif (couper le graphe en deux et recommencer récursivement), aggrégatifs (cf plus bas) ou autres
- le temps de calcul

Nous travaillerons en TP sur un algorithme optimisant la *modularité de Newman*, aggrégatif et de complexité moyenne. Ce choix est un compromis simplicité à programmer / efficacité des résultats.

b) La modularité de Newman

Soit $G = (V, E)$ un graphe non orienté et $P = V_1..V_k$ une *partition* des sommets de G en k ensembles, appellés clusters (chaque sommet appartient à un et un seul ensemble, aucun ensemble n'est vide).

La **modularité** de la partition est, informellement, la proportion des *liens internes* de la partition moins la proportion de liens internes de la même partition mais sur le graphe rebranché aléatoirement.

Plus formellement, soient V_i et V_j deux clusters. On note $m(i, j)$ le nombre d'arêtes ayant une extrémité dans i et l'autre dans j. On a en particulier $m(i, i)$ est le nombre d'arêtes **internes** au cluster V_i (dont les deux extrémitiés appartiennent à V_i). Vous noterez que $\sum_{i,j} m(i, j) = m$.

La proportion d'arêtes entre V_i et V_j (pourcentage du total des arêtes du graphe) est

$$e_{ij} = \frac{m(i, j)}{m}$$

Si on recombinait les arêtes du graphe au hasard en respectant les degrés (ce que l'on appelle un *null model*), alors la probabilité qu'il y ait une arête du sommet u au sommet v serait

$$\frac{deg(u)}{2m} \times \frac{deg(v)}{2m}$$

On définit a_{ij} la proportion d'arêtes entre les clusters V_i et V_j dans le *null model* :

$$a_{ij} = \sum_{u \in V_i, v \in V_j} \frac{deg(u) \times deg(v)}{4m^2}$$

On peut remarquer que si $i = j$:

$$a_{ii} = \frac{\left(\sum_{v \in V_i} deg(v)\right)^2}{4m^2}$$

Finalement la modularité de Newman de la partition P, notée $Q(P)$ est :

$$Q(P) = \sum_{i=1}^{i=k} e_{ii} - a_{ii}$$

et en développant

$$\boxed{Q(P) = \sum_{i=1}^{i=k} \left(\frac{m(i,i)}{m} - \frac{\left(\sum_{u \in V_i} deg(u)\right)^2}{4m^2} \right)} \quad (1)$$

La modularité est un nombre entre -1 et 1. Un nombre proche de 1 indique un bon clustering.

II) L'algorithme de Louvain

Calculer la partition en clusters de modularité maximale est un problème NP-complet. Donc on propose des **heuristiques** pour la calculer. L'une des plus connues, celle qui s'agit d'implémenter dans ce TP, est l'algorithme de Louvain (du nom de l'université de Louvain-la-Neuve où l'équipe de Blondel l'a développé). Donc, cet algorithme ne calcule pas forcément *la* modularité maximale, mais s'en approche.

Cet algorithme est *décrémental* et *aggrégatif* : on part d'une partition triviale P^1 où chaque cluster est fait d'un unique sommet (il y a donc $k = n$ clusters).

Ensuite à chaque étape t (t de 1 à n) on **fusionne deux clusters**. C'est-à-dire qu'on passe de $P^t = V_1^t, ...V_k^t$ à $P^{t+1} = V_1^{t+1}, ...V_{k-1}^{t+1}$ qui a les même clusters, sauf deux : les clusters V_a^t et V_b^t de P^t sont remplacés par un unique $V_c^{t+1} = V_a^t \cup V_b^t$ dans P^{t+1}.

Quels clusters a et b prendre ? Ceux, parmis les $O(k^2)$ choix possibles, **les deux qui vont faire le plus augmenter la modularité**. Notez que la modularité peut diminuer d'une étape à l'autre (il se peut qu'aucune fusion n'améliore la modularité : dans ce cas on fait la fusion la moins pire).

À la fin P^n ne contient plus qu'un cluster. Ce n'est pas elle qui nous intéresse ! Mais on renvoie $Q(P^{t_{max}})$, la meilleure partition vue dans au cours du calcul (la partition qui maximise $Q(P^t)$, pour t entre 1 et n). En effet on peut calculer que la modularité initiale est légèrement négative :

$$Q(P^1) = -\frac{\sum_v deg(v)^2}{4m^2} \quad (2)$$

tandis que la modularité finale est $Q(P^n) = 0$. Entre les deux on espère s'être approché de 1.

Notez que la page Wikipedia `Méthode de Louvain` contient plusieurs erreurs, par exemple la complexité ne peut pas être baissée jusqu'à $O(n \log n)$ (ce qui est plus petit que la taille du graphe dans certains cas ! !), et que la description de l'algorithme diffère. Il faut donc prendre la version donnée par ce TP.

III) Travail demandé

Votre programme doit être lancé par
`java TP34 option graphe.txt fichier.clu`
Il y a donc trois paramètre en ligne de commande :

- la commande qui peut être soit le mot `modu`, soit `paire`, soit `louvain`
- le nom d'un fichier texte stockant un graphe au format de Stanford
- le nom d'un fichier texte stockant une partition au format `.clu`, voir ci-dessous. Ce fichier sera **lu** si la commande est `modu` ou `fusion` et **écrit** si c'est `louvain`

Si la commande est `modu`, le programme doit calculer la modularité de la partition donnée du graphe donné, et afficher sa valeur. On affiche donc seulement un nombre entre -1 et 1 (ou bien un message d'erreur)

Si la commande est `paire`, le programme doit calculer la paire de clusters de la partition dont la fusion augmenterait le plus la modularité. Il s'agit donc d'une étape de l'algorithme de Louvain. L'affichage consiste dans ce cas en trois lignes : les deux premières lignes représentant des deux cluster (chaque ligne est la liste des sommets du cluster, comme dans un fichier `.clu`). Si plusieurs paires de cluster augmentent la modularité de la même (meilleure) valeur exactement vous pouvez afficher la paire que vous voulez (mais une seule paire). La dernière ligne est de combien la modularité augmente quand ces deux clusters-là sont fusionné (différence entre la modularité avant la fusion et après la fusion).

Enfin si la commande est `louvain` alors le programme effectue tout l'algorithme de Louvain. Il doit écrire dans le fichier `.clu` dont le nom est passé en paramètre, la partition atteignant la modularité maximale. Le programme fait un affichage unique : la modularité de la partition du `.clu` qui vient d'être écrit (la meilleure modularité rencontrée.) On affiche donc uniquement ce nombre, qui est l'approximation de la modularité du graphe, et rien d'autre, sauf en cas d'erreur.

a) Le format `.clu`

Pour représenter une partition d'un ensemble on utilise un format de fichier très simple. C'est un fichier texte. Il y a une ligne par cluster. Chaque ligne est constituée de numéros de sommets du cluster, chaque numéro étant suivi par un espace, ainsi chaque ligne (même la dernière) se termine par un `" \n"` (espace puis newline). Il n'y a pas de ligne vide ni de commentaire. Chaque sommet qui apparaît dans le fichier d'entrée `.txt` au format de Stanford doit apparaître une et une seule fois dans le fichier `.clu`. Cela implique que au lieu de considérer les sommets qui n'apparaissent pas dans le graphe comme ayant degré 0, ont considère maintenant qu'ils n'existent pas, et donc ils n'apparaissent pas non plus dans le `.clu`. Par exemple si un graphe a comme ensemble de sommets $V = \{1, 2, 3, 5, 7, 8\}$ et une partition $P = \{\{1, 3, 5\}, \{2\}, \{7, 8\}\}$ alors le fichier `.clu` correspondant est

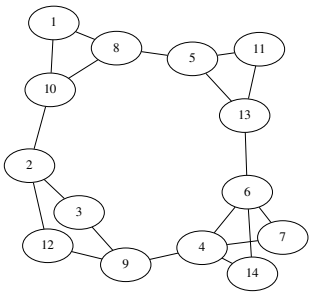
```
1 3 5 \n
2 \n
7 8 \n
```

L'ordre des clusters est quelconque (permuter les lignes d'un `.clu` représente la même partition) et l'ordre des sommets aussi (permuter les mots d'une même ligne d'un `.clu` représente la même partition). La même partition P est donc représentée validement par le fichier `.clu` :

```
8 7 \n
1 5 3 \n
2 \n
```

b) Exemples

On trouvera sur Moodle le fichier `exemple.txt` correspondant à ce graphe :



On y trouvera aussi les fichiers `exemple_1.clu`, `exemple_2.clu` ... `exemple_14.clu` correspondant aux 14 étapes d'une exécution de l'algorithme de Louvain sur ce graphe. `exemple_11.clu` atteint la meilleure modularité et est identique au `exemple_best.clu` généré ci-dessous, avec 4 clusters et une modularité de 0.5318... Les autres `.clu` correspondent aux partitions de meilleure modularité trouvée par mon programme. Leur nom correspond à une graphe de Stanford. Les temps sont sur la machine servant à corriger. Il est possible que vous trouviez des partitions de modularité encore meilleure !

```
$ java TP34 modu exemple.txt exemple_1.clu
-0.07617728531855956
$ java TP34 paire exemple.txt exemple_1.clu
1
8
incrément de modularité 0.04432132963988919
$ java TP34 modu exemple.txt exemple_2.clu
-0.031855955678670375
java TP34 paire exemple.txt exemple_2.clu
10
1 8
incrément de modularité 0.08448753462603878
$ java TP34 louvain exemple.txt exemple_best.clu
Meilleure modularité : 0.5318559556786704
java TP34 modu exemple.txt exemple_best.clu
0.5318559556786704
$ time java TP34 louvain facebook_combined.txt facebook_combined.clu
Meilleure modularité : 0.4336539313451467

real 0m37,628s

$ time java TP34 louvain email-Eu-core.txt email-Eu-core.clu
Meilleure modularité : 0.3471325650460906

real 0m1,570s

$ time java TP34 louvain Wiki-Vote.txt Wiki-Vote.clu
Meilleure modularité : 0.157603779087831

real 2m55,280s

$ time java TP34 louvain as20000102.txt as20000102.clu
Meilleure modularité : 0.6130186846639589

real 3m1,939s

$ java TP34 modu exemple.txt facebook_combined.clu
Exception in thread "main" java.lang.Error: Sommet 15 dans facebook_combined.clu mais pas dans graphe
at Partition.<init>(TP34.java:237)
at TP34.main(TP34.java:481)
$ java TP34 modu facebook_combined.txt exemple_best.clu
Exception in thread "main" java.lang.Error: Erreur taille partition 14 et taille graphe 4039
at Partition.<init>(TP34.java:267)
at TP34.main(TP34.java:481)
```

c) Découpage en deux rendus

Pour le 19 novembre, il faut rendre une implémentation de la commande `modu`. Il faut donc impérativement :

- Parser la ligne de commande en affichant la réponse, ou un message `commande non implémentée`, ou `commande inexistante`, selon le cas
- Parser les fichiers au format de Stanford et au format `.clu` en détectant les erreurs éventuelles
- Avoir une structure de donnée pour représenter les graphes (comme aux deux TPs précédents) avec les classes `Graphe` et `Sommet`, mais aussi les partitions (classe `Partition`) et les clusters (classe `Cluster`).
- Pouvoir calculer la modularité par une méthode `Partition.Q()`

Pour le 3 décembre, il faut tout rendre (donc implémenter les deux autres commandes)

Les critères d'évaluation communs sont la correction du résultat, ce qui inclut la valeur de modularité trouvée pour l'algorithme de Louvain (qui doit être le plus élevé possible pour une partition valide en clusters), le temps d'éxecution (un programme ne donnant aucun résultat au bout de 20 fois le temps du corrigé sera tué) et la qualité du code. **La quantité de mémoire utilisée comptera pour le TP3 mais pas pour le TP4.** En effet le calcul de modularité peut se faire en temps et espace linéaire, mais pas l'algorithme de Louvain. Pour le TP 4, on privilégiera au maximum le temps d'exécution au détriment éventuel de la mémoire. Vous aurez deux notes diffénrentes et comptant autant l'une que l'autre, donc une absence de rendu du TP3 et un rendu parfait du TP4 compteront autant que deux 10/20 dans la note finale. Inversement, si vous avez tout terminé au 19 novembre, rendez quand même un TP4 (cela peut être le même code exactement que le TP3...) Notez que la classe principale s'appellera toujours `TP34`.

IV) Implémentation efficace

L'algorithme fait $O(n^3)$ calculs de modularité (un entre tout couple de clusters à chaque étape t), chacun en temps $O(m)$, si on a programmé naïvement donc en tout on est en $O(n^3m)$. Il existe plusieurs façons d'accélérer le calcul.

a) Incrément de modularité

Tout d'abord, remarquons que quand on fusionne V_a et V_b en V_c :

$$Q(P^{t+1}) - Q(P^t) = (e_{cc} - a_{cc}) - (e_{aa} - a_{aa}) - (e_{bb} - a_{bb})$$

Cette quantité dit de combien la modularité augmente, c'est donc elle qu'il faut maximiser dans le calcul de la meilleure paire de clusters. Pour la calculer, on ne regarde que les clusters V_a et V_b (sans avoir à les fusionner effectivement en V_c car cette fusion ne sera rendue effective que si ce sont eux qui sont choisis). Si on note, pour chaque cluster V_i, la somme des degrés de ses sommets $somDeg(i) = \sum_{x \in V_i} deg(x)$, alors on a $e_{cc} - (e_{aa} + e_{bb}) = \frac{m(a,b)}{m}$ et donc

$$Q(P^{t+1}) - Q(P^t) = \frac{m(a,b)}{m} - \frac{(somDeg(a) + somDeg(b))^2}{4m^2} + \frac{(somDeg(a))^2}{4m^2} + \frac{(somDeg(b))^2}{4m^2} \quad (3)$$

Notez que les équations (2) et (3) font qu'il n'est même pas nécéssaire d'implémenter le calcul de modularité de l'équation (1). L'équation (3) doit par contre être calculée entre *toute paire* de clusters à l'étape t, pour savoir lesquels vont être effectivement fusionner pour passer à l'étape $t+1$: les deux qui ont le meilleur incrément de modularité.

Dans l'équation (3), le calcul de $somDeg(i)$ peut se faire en temps constant ! En effet initialement (dans la partition P^1) les clusters ont un seul sommet : pour tout i $V_i = \{x_i\}$ et on a simplement $somDeg(i) = deg(x_i)$. Ensuite, quand on fusionne V_a et V_b en V_c on a simplement $somDeg(c) := somDeg(a) + somDeg(b)$.

Reste à calculer $m(i,j)$ ce qui est plus compliqué.

b) Calcul de $m(i,j)$

Il existe plusieurs façons de calculer $m(i,j)$, le nombre d'arêtes entre les clusters V_i et V_j. Par ordre d'efficacité croissante :

1. avec la méthode naïve (compter les arêtes) on reste asymptotiquement en temps $O(n^3m)$, même si en pratique ce sera plus rapide que calculer l'incrément de modularité que la modularité complète.
2. Un unique parcours de toutes les arêtes (donc un parcours de chaque liste d'adjacance) à t'étape t permet de calcuter *tous* les $m(i,j)$ en $O(m)$ (on regarde, pour toute arête, entre quels clusters elle est). Le calcul de l'équation (3) est maintenant en temps constant, donc cette implémentation de l'algorithme de Louvain est en $O(n(m+n^2)) = O(n^3)$.

3. Stocker $m(i,j)$ dans une matrice $m[i][j]$. Cette solution consomme de la mémoire mais assure aussi un calcul en $O(n^3)$ en tout. En effet initialement cette matrice est la matrice d'adjacence du graphe ($m[i][j]$=1 si les sommets x_i et x_j, uniques sommets des clusters V_i et V_j de la partition initiale, sont reliés par une arête, et 0 sinon). Puis à chaque fusion de clusters V_a et V_b en V_c, on décide que V_c sera, disons, dans la même ligne et colonne de la matrice que V_a. Il suffit d'ajouter à cette ligne la ligne correspondant à V_b
4. la matrice en question est creuse (beaucoup de cases à 0). On peut utiliser diverses implémentations des matrices creuses en Java, le plus simple étant d'utiliser une `HashMap`. Ainsi l'espace mémoire occupé est en $O(m)$ au lieu d'être en $O(n^2)$ tandis que le temps de calcul de l'équation (3) est toujours constant.

c) Usage d'un tas

Enfin une implémentation encore meilleure est fournie grâce à un tas. En effet, à chaque étape t on fait beaucoup de calculs identiques (la pluspart des clusters n'ont pas changé entre P^t et P^{t+1}) donc on peut stocker dans un tas-max, pour toute paire de clusters, de combien leur fusion améliorerait la modularité. Ce sont donc des paires de clusters qui sont les éléments du tas, avec comme clé de combien leur fusion améliorerait la modularité (l'incrément de modularité). En Java on utilise une `PriorityQueue`. Pour savoir quels clusters fusionner, on n'a qu'à extraire le maximum de ce tas. Il faut réinjecter les $O(n)$ nouvelles valeurs dans le tas à chaque fusion (et gérer les suppressions, soit immédiatement, soit plutôt en ignorant les valeurs qui sortiront concernant des clusters supprimés antérieurement). À chaque fusion il faut remettre à jour les $m(i,j)$ et le tas, ce qui se fait en temps $O(n)$ si on a bien programmé. Au final on est donc en $O(n^2)$.