Grands Réseaux d'Interaction
TP nº 4/4 : Calcul de modularité par algorithme décrémental

à rendre pour le **15 mars 23h59**
Les règles générales en TP restent valides, voir feuille du TP 1

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 extrrémitiés appartiennent à V_i). Vous noterez que $\sum_{i,j} m(i, j) = 2m$.

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

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

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

II) L'algorithme décrémental

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 est l'algorithme de Louvain (du nom de l'université de Louvain-la-Neuve où l'équipe de Blondel l'a développé) [1]. Donc, cet algorithme ne calcule pas forcément *la* modularité maximale, mais s'en approche. Dans le TP, on implémentera une variante (attention, ce n'est pas le même algorithme, la trace est dn particulier très différence). Notre 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.

III) Travail à faire

Votre programme doit être lancé soit sous la forme à *deux paramètres* :
`java TP5 graphe.txt nombre_d_arrêtes`
soit sous la forme à *trois paramètres* `java TP5 graphe.txt nombre_d_arrêtes fichier_trace.txt`
Le graphe en entrée est comme d'habitude au format de Stanford, et on donne en deuxième paramètre le nombre de lignes du fichier tel que `wc -l` le fournit. le troisième paramètre optionnel est le nom d'un fichier de sortie. Qu'il y ait deux ou trois paramètres, le programme devra afficher deux nombres, sur deux lignes, dans cet ordre :

- La modularité estimée du graphe, c'est à dire la meilleure modularité rencontrée lors de l'algorithme décrémental
- Le nombre de clusters de la partition où cette modularité est atteinte (la première fois, si elle est atteinte plusieurs fois)

Si un troisème argument est donné, alors le programme devra aussi écrire une **trace** : un fichier texte (nommé selon ce paramètre) où, à chaque étape, c'est-à-dire initialement et après chaque fusion de cluster, on écrit trois lignes :

1. Notez que la page Wikipedia `Méthode de Louvain` contient plusieurs erreurs, par exemple la complexité ne peut pas être baissée jusqu'à $O(n \log n)$

- d'abord numéro d'étape
- puis la modularité courante
- enfin, sur une ligne la partition courante. Attention, je dois pouvoir parser cette ligne, donc respectez le format d'affichage suivant : on n'utilise que des chiffres ascii (pour les numéros de sommets), espace et crochets `'[ ']'`. On peut mettre 0 ou 1 espace avant ou après les crochets. L'ordre des sommets ou des clusters est quelconque. Voyez les `*_trace.txt` donnés en exemple.

IV) Exemples

On trouvera sur Moodle les fichiers `exemple.txt` et les traces. Voilà des exemples de commandes. On remarquera que la quasi-totalité du temps est passée à écrire la trace avec un `PrintWriter`, c'est sûrement améliorable, mais **le temps qui compte est celui sans l'option trace**, la trace n'étant là que pour permettre de suivre l'execution (il est néanmoins obligatoire de l'implémenter). Notez que mon implémentation est déterministe et donc la lancer deux fois produit toujours le même résultat, cependant, il y a des choix arbitraires dans le glouton en cas d'égalité : vous pouvez donc trouver une modularité et une trace différentes. Si la modularité est beaucoup plus petite ou plus grande, vous vous êtes trompés. Si elle est légèrement plus grande, bravo !

```
$ /usr/bin/time -f "%e\n%M" java TP4 exemple.txt 19 exemple_trace.txt
0.4265927977839335
4
0.08
24628
$ /usr/bin/time -f "%e\n%M" java TP4 ../../data/email-Eu-core.txt 25571
0.34488629962532835
84
0.26
60656
$ /usr/bin/time -f "%e\n%M" java TP4 ../../data/facebook_combined.txt 88234
0.43358593034779874
549
0.89
178920
$ /usr/bin/time -f "%e\n%M" java TP4 ../../data/as20000102.txt 13899
0.6077556652618239
56
2.08
270076
$ /usr/bin/time -f "%e\n%M" java TP4 ../../data/wiki-Vote.txt  103693
0.15740636558235122
5027
2.90
549012
$ /usr/bin/time -f "%e\n%M" java TP4 ../../data/email-Enron.txt 367666
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
$ /usr/bin/time -f "%e\n%M" java  -Xmx6g TP4 ../../data/email-Enron.txt 367666
0.23756844173369654
11559
377.32
5908984
```

Comme le montre le dernier exemple, le facteur limitant pour le corrigé est autant le temps que la mémoire : en effet, il est programmé avec une matrice et non un `HashMap`... Ici, on triche en donnant 6Go à la JVM, qui normalement alloue 1Go de tas au maximum...

```
$ /usr/bin/time -f "%e\n%M" java TP4 ../../data/email-Eu-core.txt 25571 email-Eu-core_trace.txt
0.34488629962532835
84
0.61
462644
$ /usr/bin/time -f "%e\n%M" java TP4 ../../data/facebook_combined.txt 88234 facebook_combined_trace.t
0.43358593034779874
549
12.47
636844
$ /usr/bin/time -f "%e\n%M" java TP4 ../../data/as20000102.txt 13899 as20000102_trace.txt
0.6077556652618239
56
41.51
916244
$ /usr/bin/time -f "%e\n%M" java TP4 wiki-Vote.txt 103693 wiki-Vote_trace.txt
0.15740636558235122
5027
74.34
995352
```

V) 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 $O(n^3 m)$. 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 $somDeg(c) := somDeg(a) + somDeg(b)$.

Enfin, notés que si à chaque étape on incrémente la modularité de cette valeur, **des grosses erreurs d'arrondi** peuvent avoir lieu : dans mon implémentation initiale la modularité finale dépassait 1 ! Donc il est préférable, après chaque fusion, de refaire le calcul exact de l'équation (1).

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 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 $O(n^2)$.

La note tiendra compte de l'efficacité de l'algorithme : il faut non seulement calculer une bonne modularité, mais la calculer vite. Il est toutefois préférable de faire une implémentation en $O(n^3m)$ juste qu'une plus rapide et fausse ! L'usage mémoire comptera aussi. Pour rappel, c'est le temps et la mémoire sans écrire de trace qui sont évalués, mais il est obligatoire d'implémenter cette option.