Grands Réseaux d'Interaction
TP n° 2 : paramètres de grands graphes

à rendre pour le 28 octobre

I) Rappel des règles générales en TP

Les règles générales restent valides en particulier celle-ci que beaucoup d'étudiants n'ont pas respecté au TP 1 (d'où 1 ou 2 points enlevé à la note) : chaque rendu doit être constitué d'une seule archive au **format tar**, donc non-compressée, portant les nom et prénom du ou des deux auteurs. Par exemple les étudiants Emmett Brown et Marty McFly rendront une archive créée par
`tar cf BrownEmmet_McFlyMarty.tar BrownEmmett_McFlyMarty/*.java`
J'insiste : votre archive ne doit pas se nommer `TP2.tar`, se décompresser sans créer de sous-répertoire, ni avoir de répertoire `src`. De plus votre classe principale doit se nommer `TP2`

II) Objectifs du TP

Le but de ce TP est de calculer divers paramètres de graphes concernant la distribution des degrés, la cardialité, le diamètre et le coefficient de clustering.

On saura quel calcul faire grâce à la ligne de commande. La classe principale de votre programme doit s'appeler `TP2` de sorte que l'on puisse lancer votre programme par la ligne de commande suivante :
`java TP2 graphe.txt commande <option>`
Commande est l'un des mot-clefs suivants : `distri`, `bascule`, `coeur`, `diametre`, `cluG` ou `cluL`. L'option est facultative et dépend de la commande. Ainsi on lancera par exemple :
`java TP2 web-BerkStan.txt distri oriente`
Le programme va charger le graphe, effectuer le calcul selon la commande, et afficher le résultat sous forme concise (ou un message d'erreur le cas échéant). On détaillera plus loin chacune des six commande et l'option éventuelle.

a) Affichage de la mémoire allouée

Il est obligatoire de placer, à la fin du chargement (avant le calcul) et à la fin du programme, un code affichant la mémoire allouée de la JVM, que l'on obtient par :

```
System.out.println("Mémoire allouée : " +
     (Runtime.getRuntime().totalMemory()-Runtime.getRuntime().freeMemory()) + "octets");
```

b) le graphe peut être orienté !

S'il n'y a pas d'option alors le graphe sera considéré comme non-orienté. En revanche si l'option vaut `oriente`, alors le graphe sera chargé comme un graphe orienté. Et cela même si des commentaires dans le fichier texte disent le contraire. Donc, si le graphe est orienté, alors chaque ligne est interprétée comme un arc (origine puis destination) et les boucles deviennent autorisées.

III) Implémentation des six commandes

La note dépendra principalement de la correction de vos calculs pour chacun des six commandes. Pour avoir la note maximale il faut implémenter l'option `oriente` pour la distribution des degrés et la cardialité. Cependant on pourra avoir une bonne note sans avoir tout implémenté.

a) Degrés

Si la commande est `distri` alors on donne la distribution des degrés. Si le degré maximum est $dmax$ alors on affiche $dmax + 1$ lignes de texte. La ligne i est formée de deux champs : i espace le nombre de sommets de degré i.
Si l'option `oriente` est donnée en plus de `distri` alors le graphe chargé est orienté et on affichera pour tout i trois champs : i espace le nombre de sommets de degré sortant i espace le nombre de sommets de degré entrant i. On va jusqu'au maximum des degrés sortant et entrant. Exemple :

```
$ java TP2 roadNet-CA.txt distri
Mémoire allouée : 314170064 octets
0 0
1 321027
2 204754
3 971276
4 454208
5 11847
6 1917
7 143
8 30
9 1
10 2
11 0
12 1
Mémoire allouée : 314170064 octets

$ java TP2 exemple.txt distri oriente
Mémoire allouée : 492840 octets
0 1 2
1 2 1
2 1 2
3 3 1
4 2 2
5 0 1
6 1 1
Mémoire allouée : 985688 octets
```

b) Bascule

La bascule est une mesure de la répartition des inégalités. En économie par exemple, on se posera la question : est-ce que 50% des gens possèdent 50% des richesses (société parfaitement égalitaire), ou est-ce que les 20% des gens les plus riches possèdent 80% des richesses (société inégalitaire), ou est-ce que 1% des gens possèdent 99% des richesses (société très inégalitaire) ? La **mesure d'équirépartition** est donc un pourcentage p tel que les p% de gens les plus riches possèdent $100 - p$% de la fortune totale. Plus cette valeur est basse moins les richesses sont équitablement réparties, mais concentrée chez les plus riches.

On va appliquer cela aux sommets d'un graphe. Considérons que l'on a classé les sommets du graphe par degré décroissant : x_1 est le sommet de plus fort degré, puis x_2 jusqu'à x_n (en cas d'égalité de degré on classe comme on veut). On note

$$S_j = \sum_{i=1}^{i=j} deg(x_i)$$

la somme partielle des j premiers degrés. On a donc $S_n = 2m$ (double du nombre d'arêtes). La **bascule** est le numéro b de sommet tel que

$$\frac{b}{n} \leq 1 - \frac{S_b}{2m} \quad \text{et} \quad \frac{b+1}{n} > 1 - \frac{S_{b+1}}{2m}$$

Notez que quand j va de 0 à n la fonction $\frac{j}{n}$ croît de 0 à 1 tandis que la fonction $1 - \frac{S_j}{2m}$ décroît de 1 à 0 donc b est bien défini et unique.

Donc, le pourcentage p qui nous intéresse, c'est-à-dire la mesure d'équirépartition du graphe, est alors $p = 100\frac{b}{n}$. Si notre programme est lancé avec la commande `bascule` alors on affichera trois choses : la valeur d'équirépartition p (exprimée en pourcentage) ; la valeur de bascule b (qui est le nombre des sommets "riches" possédant $100-p\%$ des degrés du graphe) ; et le degré du sommet x_b (degré de bascule). La commande `bascule` ne prend aucune option. Exemple :

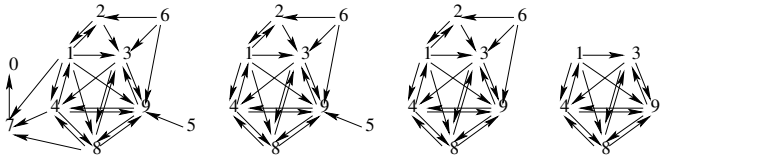
```
$ java TP2 roadNet-CA.txt bascule
Mémoire allouée : 310356320 octets
valeur d'équirépartition : 44.17603040088418%
bascule : 868150 sur 1965206 sommets
degre de bascule : 3
Mémoire allouée : 318217160 octets

$ java TP2 web-BerkStan.txt bascule
Mémoire allouée : 324129416 octets
valeur d'équirépartition : 24.104461275776018%
bascule : 165171 sur 685230 sommets
degre de bascule : 17
Mémoire allouée : 326870352 octets
```

c) Calculs de cardialité

Étant donné un graphe G et un entier k, le *k-cœur* de G est le sous graphe (unique) de G consistant à enlever les sommets de degré inférieur (strictement) à k de G, itérativement, jusqu'à stabilisation. C'est-à-dire que, partant de $G_0 = G$, tant que G_i et G_{i+1} sont différents, G_{i+1} est le sous-graphe induit par les sommets de degré $\geq k$ de G_i, en gardant les arcs (ou arêtes) entre eux. Par exemple, le 2-cœur d'un arbre est vide, car on enlève les feuilles (qui ont degré 0) étape par étape, jusqu'à avoir tout enlevé.

Notre programme, lancé avec la commande `coeur`, affichera un résultat où la ligne i donne le nombre de sommets du i-cœur. i commence à 0. On s'arrêtera quand le i-coeur est vide (0 sommets).

Cette commande prend une option : `oriente`. Si elle est fournie en ligne de commande, on travaille avec des graphes orientés. Le degré à prendre en compte est dans ce cas le degré `sortant` : pour le 1-cœur il ne reste plus de sommet de degré sortant 0, etc. Voilà un exemple orienté :



Exemple :

```
$ java TP2 roadNet-CA.txt coeur
Mémoire allouée : 315591000 octets
0-coeur restent 1965206 sommets
1-coeur restent 1965206 sommets
2-coeur restent 1591795 sommets
3-coeur restent 4568 sommets
4-coeur restent 0 sommets
Mémoire allouée : 315591000 octets

java TP2 exemple.txt coeur oriente
Mémoire allouée : 985688 octets
0-coeur restent 10 sommets
1-coeur restent 8 sommets
2-coeur restent 7 sommets
```

```
3-coeur restent 5 sommets
4-coeur restent 0 sommets
Mémoire allouée : 985688 octets

$ java TP2 exemple.txt coeur
Mémoire allouée : 985688 octets
0-coeur restent 10 sommets
1-coeur restent 10 sommets
2-coeur restent 8 sommets
3-coeur restent 8 sommets
4-coeur restent 5 sommets
5-coeur restent 0 sommets
Mémoire allouée : 985688 octets

$ java TP2 web-BerkStan.txt coeur
Mémoire allouée : 323048992 octets
0-coeur restent 685230 sommets
1-coeur restent 685230 sommets
2-coeur restent 629459 sommets
3-coeur restent 546350 sommets
4-coeur restent 478440 sommets
5-coeur restent 433892 sommets
6-coeur restent 365164 sommets
7-coeur restent 329179 sommets
8-coeur restent 290249 sommets
... // NB : j'ai enlevé 143 lignes
152-coeur restent 618 sommets
153-coeur restent 392 sommets
... // tous les coeurs de 153 à 201 ont la même taille
201-coeur restent 392 sommets
202-coeur restent 0 sommets
Mémoire allouée : 323048992 octets
```

d) Approximation du diamètre

Si la commande `diametre` est donnée, alors on lancera l'algorithme de Habib (double-double-BFS) pour calculer le diamètre du graphe. Se reporter au cours pour en avoir la description. L'option est le numéro du premier sommet r. Si aucune option n'est donnée on part du sommet de plus petit numéro existant dans le graphe. En plus d'afficher le diamètre ainsi calculé, on affichera les sommets r, a_1, b_1, m_1, a_2, b_2 et les distances $dist(a_1, b_1)$ et $dist(a_2, b_2)$. Exemple :

```
 $ java TP2 roadNet-CA.txt diametre 42
Mémoire allouée : 313195584 octets
Warning : non connexe, on a parcouru seulement 1957027 sommets sur 1965206
Warning : non connexe, on a parcouru seulement 1957027 sommets sur 1965206
Warning : non connexe, on a parcouru seulement 1957027 sommets sur 1965206
Warning : non connexe, on a parcouru seulement 1957027 sommets sur 1965206
Heuristique de Habib r=42 a1=1221069 b1=1706539 a2=1411928 b2=807041
dist(1221069,1706539)=861 dist(1411928,807041)=799
Mémoire allouée : 435766928 octets
```

e) Coefficient de clustering global

Le *coefficient de clustering global* $cluG(G)$ d'un graphe G est la statistique disant si deux sommets ayant un voisin commun sont relié. Il modélise donc l'adage *les amis de mes amis sont mes amis* : si x et y ont un ami z en commun, seront-ils reliés ? Parmi toutes les paires de sommets d'un graphe G ayant un voisin en commun, la proportion de ceux qui sont reliés est notée $cluG(G)$. Cela va de 1 pour un graphe complet à 0 pour un graphe sans triangle (par exemple, un graphe biparti). Plus précisément, on pose :

- un triangle dans un graphe est un ensemble de trois sommets tous reliés entre eux
- $tri(G)$ est le nombre de triangles de G
- un $\mathbb{V}$ dans un graphe est formé de deux arêtes incidentes, c'est-à-dire ayant un sommet en commun. Un $\mathbb{V}$ xyz correspond donc à une arête xy et une arête yz, peu importe que l'arête yz existe ou pas. En particulier, un triangle correspond à trois $\mathbb{V}$ différents.
- $nv(G)$ est le nombre de $\mathbb{V}$ de G. On peut remarquer que

$$nv(G) = \sum_{\substack{x \text{ sommet de } G \\ \text{avec } deg(x)>0}} \frac{deg(x)(deg(x)-1)}{2} \quad \text{et on a} \quad cluG(G) = \frac{3 * tri(G)}{nv(G)}$$

Donc, si notre programme est lancé avec la commande `cluG` alors on affichera le coefficient de clustering global. Cette commande ne prend aucune option (on ne travaille que sur des graphes non-orientés). Exemple :

```
$ java TP2 roadNet-CA.txt cluG
Mémoire allouée : 338612848 octets
nombre de triangles 120676 nombre de V 5995090
clustering global 0.06038741703627468
Mémoire allouée : 338612848 octets

$ java TP2 web-BerkStan.txt cluG
Mémoire allouée : 320003992 octets
nombre de triangles 64690980 nombre de V 21540536336
clustering global 0.009009661457484334
Mémoire allouée : 320003992 octets
```

f) Coefficient de clustering local moyen

Ce coefficent est une variante du précédent.

- Le **voisinage** d'un sommet x est le graphe en obtenu en enlevant tous les sommets qui ne sont pas voisins de x, ainsi que x lui-même.
- La **densité** d'un graphe non-orienté avec n sommets et m arêtes est son nombre d'arêtes divisé par le nombre d'arête maximal qu'il pourrait avoir. Si le graphe a moins de deux sommet alors sa densité est 0. Sinon elle vaut
$$\frac{2m}{n(n-1)}$$
- Le *coefficient de clustering local* d'un sommet x est la densité du voisinage de x.

Pour le dire autrement, en posant $cluL(x)$ le clustering local d'un sommet x, $tri(x)$ le nombre de triangles auxquel appartient le sommet x, et $deg(x)$ son degré, on a (quand $deg(x) \geq 2$) :

$$cluL(x) = \frac{2 * tri(x)}{deg(x)(deg(x)-1)}$$

Et $cluL(x) = 0$ si x est isolé ou a un seul voisin.
Le *clustering local moyen* $cluL(G)$ du graphe est la moyenne des clustering locaux de ses sommets, soit

$$cluL(x) = \frac{1}{n} \sum_{x \in G} cluL(x)$$

Ainsi, quand le programme est lancé avec la commande `cluL` alors on affichera le coefficient de clustering local moyen. Cette commande ne prend aucune option (on ne travaille toujours que sur des graphes non-orientés). Exemples :

```
$ java TP2 roadNet-CA.txt cluL
Mémoire allouée : 310586824 octets
clustering local moyen 0.04637027007475569
Mémoire allouée : 310586824 octets
```

```
$ java TP2 web-BerkStan.txt cluL
Mémoire allouée : 330041600 octets
clustering local moyen 0.5966969491965554
Mémoire allouée : 330041600 octets
```