Grands Réseaux d'Interaction
TP 1 Structure de données pour parcourir un graphe

Fabien de Montgolfier
fm@irif.fr

21 janvier 2022

à rendre pour le 28 janvier 23h59

1 Règles générales en TP

— Les feuilles de TP se trouveront sur Moodle, cours *IFECY070 Grands Réseaux d'Interaction*. La clef d'inscription est `GRI2022`.

— Les TPS sont notés. Chaque groupe doit déposer un fichier sur Moodle. Les rendus sont à faire par groupe (un dépôt par groupe). soit individuellement, soit en groupe de **2 étudiants maximum**. Il est autorisé de s'entre-aider entre groupes mais **le plagiat donnera une note de 0**. Il est bien sûr possible que la même courte portion de code se retrouve dans plusieurs programmes mais en cas d'excès, c'est 0.

— Les TPs se font en **Java** exclusivement, en Java 11 et sans API externe (seule celle par défaut est autorisée)

— Vos programmes doivent impérativement prendre leurs paramètre en ligne de commande comme indiqué dans l'énoncée. Il est exclus de les prendre interactivement ou par des constantes en dur dans le code (pas de nom de fichier dans le code !). Le non-respect des consignes peut perturber la correction automatique qui sera effectuée ensuite avec un fort impact négatif sur la note.

— Chaque dépôt doit être constitué d'une seule archive au **format zip** (pas `.tar` ni autre). Elle ne doit pas contenir les grands graphes qui vous seront fournis, ni des `.class`, ni des `.jar`, logs ou autres choses inutiles. Cette archive doit se décompresser en créant un répertoire (et non en mettant les fichiers dans la racine du répertoire courant, ni dans un répertoire `src`) portant le nom `GroupeXY` par exemple `Groupe01`.

— La compilation doit se faire avec `javac TP1.java` sans aucune autre option, et l'éxécution par `java TP1 [arguments]`

— Les critères pour l'évaluation sont la correction du résultat, le temps d'exécution, la mémoire allouée et la beauté du code.

2 La base de données de Stanford

Le site `https://snap.stanford.edu/data/` est une vraie mine d'or pour le traitement de grands graphes réels. Les graphes sont en général fournis sous forme d'un fichier texte, où chaque ligne contient deux nombres (en décimal ASCII) : origine puis extrémité d'un arc. Si le graphe est non-orienté, alors il s'agit des deux extrémités d'une arête (la base précise "Undirected" ou "Directed" selon le cas). On trouve toutes les tailles, du millier au demi-millard d'arcs. Les lignes commençant par un `#` sont des commentaires à ignorer. On ne s'intéressera qu'aux graphes de la base qui suivent cette syntaxe simple.

La pluspart des paramètres qui seront à calculer en TP sont déjà donnés dans la base, ce qui permet de vérifier vos calculs. Remarquez que parfois les graphes sont triés (les arcs arrivent dans l'ordre lexicographique), parfois non, et qu'il y a parfois des "trous" (numéros de sommets non attribués). On travaille en fait sur des multi-graphes : si un même arc $x \to y$ apparaît deux fois, alors y sera deux fois voisin de x, et compte pour 2 dans son degré. Par contre, pour les graphes non-orientés, une arête est (en géréral, mais pas toujours) stockée une seule fois (on stocke xy mais pas yx). Dans ce premier TP, on ignorera cela.

3 Représentation d'un graphe en mémoire

Le but de ce TP est avant tout d'avoir une réprésentation en mémoire, qui resservira ensuite. Vous **devez** respecter les caractéristiques suivantes :

- Le chargement du graphe doit être une étape distincte des calculs : le fichier d'entrée doit être lu (par exemple par un constructeur) avant les calculs
- Dans ce premier TP (mais pas les suivants), on considérera uniquement des **graphes orientés**.
- Chaque sommet du graphe sera codé par un `int` (ce qui nous laisse travailler avec des graphes jusqu'à deux milliards de sommets environ).
- Le meilleur moyen, chaudement recommandé, de stocker des listes d'adjacence est... un tableau `int[] tab`. Vous pouvez utiliser des structures dynamiques (`ArrayList`,...) à vos risques et périls (malus possible pour la consommation mémoire et la laideur du code) mais je le déconseille.
- Il ne faut pas garder en mémoire du texte provenant du fichier. Le type `String` et ses variantes ne peuvent être utilisés pour coder l'adjacence du graphe : ça ne passera pas pour de grands graphes.
- Il doit y avoir une classe `Graphe` (ou `Graph` si vous préférez...)
- Deux options s'offrent principalement à vous :
  - stocker chaque $N(u)$ dans un tableau de type `int[]` et l'ensemble des listes d'adjacence sera stocké dans un tableau de tableaux (plus simple),
  - ou stocker dans un seul tableau tous les $N(u)$ pour $u = 0 \dots n-1$ bout à bout et utiliser un autre tableau pour stocker l'index où commence chaque $N(u)$ (plus compact).
- Il est demandé que votre implémentation utilise un espace mémoire linéaire (en le nombre de sommets plus d'arcs). Le plus petit sera le mieux. Mais en Java, la consommation mémoire exacte ne peut être obtenue aussi précisément que dans d'autres langages.
- Le parcours d'une liste d'adjacence doit être en temps linéaire en sa taille.

4 Travail demandé

Il est demandé de faire un programme qui, étant donnés quatre paramètres, dans cet ordre :

- un nom de fichier de graphe (fichier texte au format de Stanford)
- son nombre de ligne (cela permet de gagner du temps de lecture) On n'a pas à le vérifier : le comportement du programme n'est pas défini si ce n'est pas le nombre de lignes tel que donné par `wc -l` du premier paramètre.
- un numéro de sommet (nommons R ce sommet)
- une distance (un entier positif, appellons-le d)

charge le graphe en mémoire, puis affiche sur la sortie standard

- Soit un message d'erreur en une ligne contenant le mot "ERREUR" en majuscule, plus d'autres informations utiles
- Soit, s'il n'y a pas eu d'erreur, les 4 nombres suivants, un par ligne (donc 4 lignes en tout) :
  - Le plus grand numéro de sommet, n
  - le nombre exact m d'arcs
  - le degré maximum d'un sommet (nombre de voisins du sommets qui en a le plus)
  - le nombre de sommets à distance d du sommet R

5 Exemples

Si `f.txt` contient :

```
# Exemple
1       2
5       2
1       3
1       6
5       1
```

alors la commande `java TP1 f.txt 6 5 2` doit afficher :

```
6
5
3
2
```

car les numéros de sommets vont de 0 à 6 (première ligne), il y a 5 arcs (deuxième ligne), le sommet 1 a 3 voisins (2, 3 et 6) et il y a deux sommets (le 3 et le 6) à distance 2 de 5

La commande `/usr/bin/time -f"%e\n%M"` donne sur deux lignes le temps d'éxécution (en seconde) et la mémoire **maximale** (en Ko) allouée à un moment donnée par le processus (toute la JVM). Les deux dernières valeurs données sont les valeurs du corrigé sur ma propre machine. Vous pouvez faire mieux, soit que vous ayez plus de CPU, soit que vous programmiez plus efficacement que moi ! Par contre mon script tue les programmes au bout de 10 minutes, donc un temps supérieur à 10 minutes vous sera compté comme un échec.

```
$ wc -l ~/data/as20000102.txt
13899 ~/data/as20000102.txt
$ /usr/bin/time -f"%e\n%M" java TP1  ~/data/as20000102.txt 13899 1 3
6473
13895
1458
2992
0.20
42068
$ wc -l ~/data/ca-AstroPh.txt
396164 ~/data/ca-AstroPh.txt
$ /usr/bin/time -f "%e\n%M" java TP1 ~/data/ca-AstroPh.txt 396164 127393 5
133279
396160
504
1072
0.26
70640
$ wc -l ~/data/roadNet-CA.txt
5533218 ~/data/roadNet-CA.txt
$ /usr/bin/time -f "%e\n%M" java TP1 ~/data/roadNet-CA.txt 5533218 2022 42
1971280
5533214
12
637
0.97
274516
$ wc -l ~/data/web-BerkStan.txt
7600599 ~/data/web-BerkStan.txt
$ /usr/bin/time -f "%e\n%M" java TP1 ~/data/web-BerkStan.txt 7600599 1234 5
685230
7600595
249
17431
0.94
282920
$ wc -l ~/data/soc-pokec-relationships.txt
30622564 ~/data/soc-pokec-relationships.txt
$ /usr/bin/time -f "%e\n%M" java TP1 ~/data/soc-pokec-relationships.txt 30622564 1234 5
1632803
30622564
8763
795821
3.81
706140
```

6 Conseils

6.1 Affichage de la mémoire allouée

Pour contrôler la mémoire utilisée, vous pouvez afficher la mémoire allouée de la JVM par :

```
public static void mem() {
     Runtime rt = Runtime.getRuntime();
     rt.gc();
     System.err.println("Allocated memory : "
                        + (rt.totalMemory() - rt.freeMemory()) / 1000000
                        + " Mb");
     System.err.flush();
 }
```

Il est recommandé d'afficher la mémoire allouée après avoir chargé les arcs.

6.2 variables intermédiaires

On peut utiliser des tableaux pour stocker (sous forme de int) l'origine et la destination des arcs, avant de construire les listes d'adjacences.

6.3 Lecture du fichier

Deux choix sont possibles pour lire le ficher : utiliser `Scanner` (plus simple) ou `BufferedReader` (plus rapide).

6.4 le graphe est orienté !

Pour ce TP on considérera que tous les graphes donnés en entrée sont orientés. On ne se souciera pas de créer des doublons, le nombre d'arcs du graphe sera le nombre d'arcs lus, ou encore l somme des tailles des listes d'adjacence.

6.5 Parcours en largeur

Pour pouvoir calculer la distance et le nombre de sommets accessibles il est necessaire d'implémenter un parcours en largeur (anglais : *Breadth First Search*). Pour la file d'attente, on peut utiliser `ArrayDeque` ou `LinkedList`.