# Grands Réseaux d'Interaction
# TP n° 1 : manipulation de grands graphes

## Règles générales en TP

- Les feuilles de TD et TP se trouveront sur Moodle, cours *Grands Réseaux d'Interaction*. La clef d'inscription est `biggergraphs`. Les rendus de TP sont à faire sur Moodle également.
- Les TPS sont notés. Les rendus sont **individuels**. Il est autorisé de s'entre-aider mais **le plagiat donnera une note de 0**. Il est bien sûr possible que la même courte portion de code se retrouve chez plusieurs personnes mais en cas d'excès, c'est 0.
- Les seuls langages autorisés sont **Java** ou **C**.
- Le code doit être affiace, clair et concis. Si vous utilisez des structures de données lourdes de l'API Java vous devez pouvoir justifier votre choix
- Les programmes doivent pouvoir être testés par un script. En conséquence, ils doivent être non-interactifs et avoir un affichage en mode texte, sauf avis contraire. Si une réponse brève est demandée, le fait de sortir des pages de texte inutile sera pénalisé, les "messages de debuggage" doivent donc être soigneusement éliminés du rendu final.
- de même le format d'entrée doit être respecté : la (ou les) commandes doivent avoir le nom et les paramètres spécifiés dans chaque exercice. À défaut une commande `-help` doit être implémentée, ou un fichier `readme.txt` décrivant la syntaxe du programme.
- Chaque rendu doit être constitué d'une seule archive au **format tar**, donc non-compressée (pas de zip, tar.gz, rar...) Cette archive doit se décompresser en créant un répertoire portant votre nom et prénom. Elle ne doit pas contenir les grands graphes qui vous seront fournis, ni des `.class`, executables (des scripts sont autorisés), logs ou autres choses inutiles. Par exemple l'étudiant John Smith rendra une archive créée par `tar cf SmithJohn.tar SmithJohn/`
- Ce répertoire doit contenir un `Makefile` de sorte que lancer la commande `make` dans ce répertoire compile tous les programmes nécessaires (vous avez le droit de copier le makefile du voisin), ou bien `javac *.java` doit suffire.
- Les critères pour l'évaluation sont la correction du résultat, mais aussi le temps d'éxecution, la mémoire allouée et la lisibilité du code.

## Le format dot

Le format `dot` est un format texte standard pour stocker des graphes. Vous trouverez des infos sur ce format sur le Web, par exemple sur
`http://www.graphviz.org/doc/info/lang.html`

Un (très) petit sous-ensemble du langage, suffisant pour commencer, est le suivant. La première ligne est `digraph nom {` pour un graphe orienté et `graph nom {` pour un graphe non-orienté appelé `nom`. Ensuite chaque arc ou arête s'écrit sur une ligne, avec les deux extrémités séparées par `--` si non-orienté et `->` si orienté. Les lignes se terminent par un point-virgule et le fichier par une accolade fermante (voir exemples page suivante). Noter qu'un sommet isolé est possible, c'est le cas du sommet 4 sur le graphe `Exemple` (alors qu'il n'y a pas de sommet 6). On utilisera pour l'instant les règles suivantes :

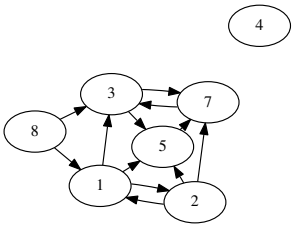
- Comme nom de sommet, on se limite à des entiers positifs en base 10 pour un parsing facile.
- Les arcs ou arêtes du fichier `.dot` d'entrée sont triées par premier sommet croissant
- On ignore tout le reste du format dot

Le package Linux `graphviz` offre plusieurs programmes de manipulation de fichier `*.dot` : `dot, neato, twopi, circo, fdp,` ou encore `sfdp` (voir `man dot`). Leur syntaxe est la même : par exemple
`neato -Tpng -O Exemple.dot`
va créer l'image `Exemple.dot.png` page suivante (car le programme `dot` donne un résultat moins joli). Il existe aussi le programme (plus ou moins) interactif `dotty`.

Vous trouverez sur les machines de l'UFR quelques exemples de tels grands graphes dans le répertoire `/info/master2/Public/GRI`

```
digraph Exemple {
      1 -> 2;
      1 -> 3;
      1 -> 5;
      2 -> 1;
      2 -> 5;
      2 -> 7;
      3 -> 5;
      3 -> 7;
      4;
      5 -> 7;
      7 -> 3;
      8 -> 1;
      8 -> 3;
}
```

```
graph AutreExemple {
      1 -- 3;
      2 -- 1;
      3 -- 2;
      4 -- 3;
}
```




# Travail demandé

Nous manipulerons des graphes stockés sous forme de *liste d'adjacence*. Si un sommet donné possède $k$ voisins sortants, alors une boucle faisant $k$ itérations, chacune en temps constant, doit pouvoir parcourir son voisinage. On sera sensible la mémoire occupée, chaque arête devant occuper un espace mémoire raisonable. Les graphes que l'on manipule seront toujours *statiques* au sens qu'il n'y a pas d'ajout/suppression de sommet ni d'arêtes après chargement. Donc, contrairement à ce que le nom semble indiquer, un **tableau** de $k$ cases est la façon la plus simple de stocker les $k$ voisins d'un sommet donné !

Si le graphe est orienté, sa liste d'adjacence contient ses voisins sortants. S'il est non-orienté, il y a tous ses voisins. **Pour ce TP 1 on se limite aux graphes orientés**.

**Exercice 1 : chargement**

Faire une fonction ou méthode qui, étant donné le nom d'un fichier `*.dot`, charge le graphe en mémoire, c'est-à-dire alloue et remplit correctement le tableau des sommets, les listes d'adjacence et autres structures.

**Exercice 2 : statistiques**

Faire un programme qui, étant donné un graphe (fichier `*.dot` passé en paramètre de la ligne de commande), affiche cinq nombres (sur une ligne, séparés par des espaces) :

- Le nombre $n$ de sommets
- Le nombre $m$ d'arcs
- Le degré sortant maximum d'un sommet
- Le degré entrant maximum d'un sommet
- Le plus grand numéro de sommet

L'executable doit etre nommé `statsDeBase`. Ainsi la commande

```
./statsDeBase Exemple.dot
```

doit fournir l'affichage

```
7 12 3 3 8
```

NB : les commandes seront écrites pour des éxécutables poduits en C ; si vous êtes en Java ce sera bien sûr `java StatsDeBase Exemple.dot`. Cela ne sera plus reprécisé.

**Exercice 3 : parcours**

À titre d'échauffement pour la suite on implémentera le parcours en largeur ou en profondeur. Faire un programme nommé `parcours` qui, étant donné un graphe (paramètre de la ligne de commande) et un numéro de sommet, affiche le nombre de sommets accessibles depuis le sommet donné (y compris lui-même). Ainsi la commande

```
./parcours Exemple.dot 1
```

doit fournir l'affichage

```
5
```