

- TP 1. La composante géante. -

Le but de ce TP est d'observer le phénomène suivant : si l'on tire au hasard un graphe G ayant n sommets et m arêtes, avec m proche de n , on s'aperçoit que les composantes de G sont de tailles très inégales. Plus précisément, une *composante géante* apparaît presque sûrement, alors que les autres composantes sont soit des points isolés, soit très petites.

Langage. Programme en C++. Votre programme pourra contenir :

```
#include <cstdlib>
#include <iostream>
#include <vector>

using namespace std;

int main(){
    int n;      // Nombre de sommets.
    int m;      // Nombre d'arêtes.
    cout << "Entrer le nombre de sommets:";
    cin >> n;
    cout << "Entrer le nombre d'arêtes:";
    cin >> m;
    int edge[m][2]; // Tableau des arêtes.
    int comp[n];    // comp[i] est le numero de la composante contenant i.
    ...
    ...
    return 0;
}
```

Ce début de code est récupérable là : <http://www.lirmm.fr/~bessy/GLIN501/TP/tp1.cc>

!!!! Pensez à tester chaque code produit sur de petits exemples!!!!

- Exercice 1 - Création d'un graphe aléatoire G à n sommets et m arêtes.

L'ensemble des sommets de G est codé par $\{0, \dots, n-1\}$. L'ensemble des m arêtes de G est stocké dans un tableau **edge** de taille $m \times 2$ et dont les entrées appartiennent à $\{0, \dots, n-1\}$. Ainsi, si xy est l'arête de G d'indice k , on aura **edge**[k][0] = x et **edge**[k][1] = y . Par exemple, le graphe sur l'ensemble de sommets $\{0, 1, 2, 3\}$ ayant pour arêtes 01, 02, 03, 12, 23 est codé par le tableau **edge**[5][2] = $\{\{0,1\}, \{0,2\}, \{0,3\}, \{1,2\}, \{2,3\}\}$.

Ecrire une fonction *void grapherandom(int n, int m, int edge[][2])* qui engendre aléatoirement le tableau **edge** en tirant au hasard chacune de ses entrées. On permettra la création d'arêtes multiples ou de boucles.

On pourra utiliser les appels :

- *srand (time(NULL))* // Initialise la graine (seed) de la fonction rand sur l'horloge.
- *rand()%k* // Retourne un entier entre 0 et $k-1$.

- Exercice 2 - Calcul des composantes connexes.

Implémenter l'algorithme du cours *void composantes(int n, int m, int edge[][2], int comp[])* qui calcule les entrées du tableau **comp** de telle sorte que **comp**[i]=**comp**[j] si et seulement si i et j

appartiennent à la même composante connexe de G .

- Exercice 3 - Retourner les tailles des composantes connexes.

Ecrire un algorithme `void ecrituretailles(int n, int m, int comp[])` qui écrit :

- Le nombre de points isolés de G (i.e. les composantes de taille 1).
- Les nombre de composantes des autres tailles, dans l'ordre croissant.

Par exemple, le résultat sera de la forme :

```
Il y a 464 points isoles.
Il y a 41 composantes de taille 2.
Il y a 12 composantes de taille 3.
Il y a 5 composantes de taille 4.
Il y a 1 composante de taille 4398.
```

Essayer d'avoir un algorithme linéaire en n pour la fonction `ecrituretailles`.

Faire varier n et m pour constater l'apparition de la composante géante.

- Exercice 4 - Optimisation de l'algorithme.

Améliorer les performances de votre algorithme de telle sorte que :

1. Lors de la lecture d'une arête ij , si `comp[i] != comp[j]`, seuls les sommets k vérifiant `comp[k] = comp[j]` sont relus et réaffectés en `comp[i]`.
2. Entre `comp[i]` et `comp[j]`, choisir en priorité de réaffecter la composante de taille minimum.

On pourra à cet effet, notamment pour 1), utiliser la structure de donnée `vector`, voir ci-dessous.

Essayer d'augmenter n et m le plus possible, avec la version non-optimisée et avec la version optimisée. La différence est-elle sensible? Etablir la complexité de chacune des versions.

- Exercice 5 - Pour aller plus loin.

Lorsque $n = 10000$ et $m = 5000$, quelle est environ la proportion de points isolés? Faites plusieurs tests et relevez la moyenne obtenue.

De même avec $n = 10000$ et $m = 10000$.

Modélisez mathématiquement le problème et trouvez les moyennes théoriques.

Mémento vector (à garder pas loin pour les tps suivants)

- `vector<int> vect` // Déclare la variable vect comme vector d'entiers.
- `vect[i]` // Accède à la $i^{\text{ème}}$ entrée de `vect`.
- `vect.size()` // Renvoie la taille de `vect`.
- `vect.push_back(i)` // Empile la valeur i sur `vect`.
- `vect.pop_back()` // Dépile `vect`.
- `vect.back()` // Retourne la valeur en haut de `vect`.
- `vect.empty()` // Retourne VRAI lorsque `vect` est vide.
- `vect.erase(vect.begin())` // Supprime la première case de `vect`.

Remarques :

- Si vous préférez, vous pouvez utiliser les structures dédiées de la STL pour gérer les piles et les files (`stack` et `queue`), ou les réimplémenter vous-même...

- Pour des simplifications d'écriture (l'important dans ces tps, c'est l'algorithmique, un peu moins la programmation...), tous les tableaux de taille variable sont déclarés dans la pile, ce qui est permis depuis C99. Toutefois, on peut vite remplir l'espace autorisé. La déclaration dans le tas fait gagner un facteur 10 sur la taille des graphes traitables (d'après mes petits tests...). Si vous le souhaitez, faites des allocations dynamiques, c'est plus performant.