TP3 - INF303 19-20 octobre 2016

Parcours eulériens

Rappels :

- Les TPs de tout le semestre seront réalisés en `C++`. Le but n'est pas d'utiliser toute la puissance du `C++`, mais simplement quelques outils qui nous faciliteront la tâche, en particulier la fonction `cout` à la place de `printf`, et la structure de données `vector` pour faire des tableaux dynamiques.
- Les fichiers à télécharger sont disponibles à http://pagesperso.g-scop.fr/~pastorl/ (ou bien, tapez *Lucas Pastor* dans Google, trouvez sa page web, et descendez jusqu'à *Teaching*), et également sur Moodle → INF303.
- Les questions marquées d'une ou plusieurs étoiles sont plus compliquées. **Néanmoins, dans ce TP, il faut traiter les questions dans l'ordre.**
- **Toutes vos fonctions doivent être testées au fur et à mesure.**

Le but du TP est d'écrire un algorithme qui calcule les chemins ou cycles eulériens d'un graphe, lorsqu'il en existe.

1 Manipulation des graphes

On représente les graphes par leur matrices d'adjacence. Le type `graphe` est défini dans le fichier `graphe.cpp` par la ligne suivante :

```
typedef vector<vector<int> > graphe;
```

Cela signifie qu'un objet de `graphe` est un tableau de tableaux d'entiers, c'est à dire une matrice. Si `g` est de type `graphe`, on accède à l'élément de coordonnée (u, v) de la matrice d'adjacence par `g[u][v]`. On a `g[u][v] == 1` si uv est une arête et `g[u][v] == 0` sinon. Une matrice d'adjacence de graphe doit toujours vérifier `g[u][v] == g[v][u]` et `g[u][u] == 0`.

Comme un graphe est un tableau (de tableaux), on accède à sa taille par `g.size ()`.

Lorsqu'un donne un graphe g comme argument d'une fonction, on écrit & devant g pour indiquer que seul l'adresse mémoire du graphe est donnée à la fonction et que l'objet n'est pas recopié, comme dans `int degre(&g graphe, int v)`. Cela ne change rien à la syntaxe du reste du programme : `g` est un objet de type `graphe`.

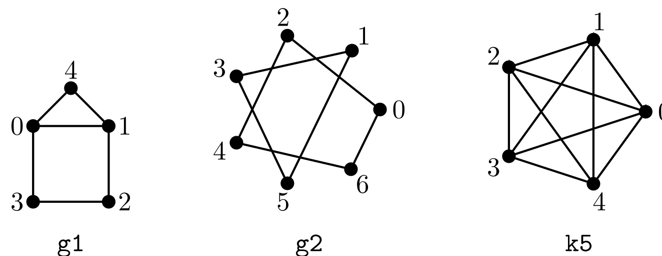
Question 1. Télécharger les fichiers `graphe.cpp`, `graphe.h` et `eulerien.cpp`. Compiler à l'aide de la commande `g++ graphe.cpp eulerien.cpp -o eulerien` puis tester avec `./eulerien`

Question 2. Écrire une fonction `void ajoute_arete(graphe &g, int u, int v)` qui ajoute l'arête uv au graphe `g`.

Question 3. Écrire une fonction `graphe graphe_complet(int n)` qui créé et renvoie le graphe complet à n sommets.

Question 4. Ecrire une fonction `graphe complementaire(graphe & g)` qui créé et renvoie le graphe complémentaire $\bar{g}$ de g. *Attention : la diagonale ne doit contenir que des 0.*

Question 5. Écire des fonctions `g1`, `g2` et `k5` qui renvoient les graphes suivants :



Ces graphes serviront à tester les fonctions ultérieures.

Question 6. Écrire une fonction `int nb_aretes(graphe &g)` qui renvoie le nombre d'arêtes du graphe.

Question 7. Écrire une fonction `int degre(graphe &g, int v)` qui prend en entrée un graphe g et un sommet v et qui renvoie le degré de v dans g.

Question 8. Écrire une fonction `int delta(graphe &g)` qui renvoie le degré maximal de g.

Question 9. (*) Écrire une fonction `void parcours_largeur(graphe &g, int depart)` qui parcourt le graphe en largeur en partant du sommet `depart`, et qui affiche les sommets au fur et à mesure.

Note : Vous aurez besoin d'une file qui existe sous le nom de `queue` de la librairie standard (consultez la documentation en ligne pour savoir comment l'utiliser). De plus, pour que l'algorithme ne boucle pas, on pourra maintenir un tableau `deja_vu` de type `vector<bool>` *indexé par les sommets, où `deja_vu[u]` indique si le sommet `u` a déjà été détecté.*

Question 10. (*) De même, écrire une fonction `void parcours_profondeur(graphe &g, int depart)` pour le parcours en profondeur en commençant du sommet `depart`.

Note : Pour un parcours en profondeur, on pourra utiliser une fonction récursive `explore` avec un tableau `deja_visite` de type `vector<bool>` indexé par les sommets. Il est également possible de le faire avec une pile, mais attention aux détails !

2 Cycle eulérien

Question 11. Rappeler à quelle condition nécessaire et suffisante un graphe contient un cycle eulérien. Faire de même pour un chemin eulérien.

Question 12. Écrire une fonction `int nb_impairs(graphe &g)` qui renvoie le nombre de sommets de degré impair du graphe g.

Pour appliquer le critère, il reste à détecter si le graphe est connexe. Pour savoir si un graphe est connexe, on le parcourt (par exemple par un parcours en profondeur) et on vérifie si tous les sommets ont été parcourus.

Question 13. Écrire une fonction `bool est_connexe(graphe &g)` qui renvoie 1 si le graphe est connexe et 0 sinon. *On pourra s'inspirer du code des fonctions de parcours en largeur ou en profondeur.*

Question 14. Déduire une fonction `bool a_cycle_eulerien(graphe &g)` qui indique si le graphe possède un cycle eulérien.

Question 15. Écrire de même une fonction `bool a_chemin_eulerien(graphe &g)` qui indique si le graphe possède un chemin eulérien.

On va maintenant construire le cycle ou chemin eulérien, s'il existe.

Question 16. Écrire une procédure `void supprime_arete(graphe &g, int u, int v)` qui enlève l'arête uv du graphe `g`. *Attention : il faut penser à modifier `g[u][v]`* **et** *`g[v][u]`.*

Pour éviter de modifier la matrice de manière incohérente, on utilisera toujours cette fonction pour supprimer des arêtes.

Question 17. Écrire une procédure `int voisin(graphe &g, int u)` qui renvoie le premier voisin de u. Cette fonction renverra la valeur -1 si u n'a pas de voisin.

Question 18. (*) On construit un chemin partant de u de la manière suivante :
À chaque étape,

- On prend le premier voisin v de u. Si il n'y en a pas, on s'arrête.
- On enlève l'arête uv du graphe.
- On recommence avec $u := v$

Écrire une fonction récursive `void promenade(graphe &g, int u, vector<int> &t)` qui calcule un chemin de la manière décrite ci-dessus, et écrit la liste des sommets parcourus (dans l'ordre) dans t. Pour que la suite fonctionne bien, **on n'écrira pas le point de départ dans `t`**. Votre fonction peut être au choix impérative ou récursive. *Note : On ajoute un élément x à la fin d'un tableau `t` avec la fonction `t.push_back(x)`.*

Question 19. Que peut-on dire sur le début et la fin d'une promenade comme ci-dessus dans le cas où le graphe possède un cycle eulérien ?

Question 20. Tester la fonction `promenade` sur `k5`. Avez-vous bien parcouru chacune des 10 arêtes du graphe une et une seule fois ? Ré-afficher `k5` après l'appel à `promenade` et commenter le résultat.

L'algorithme de calcul de cycle eulérien est donné ci-dessous :

Algorithm 1 CycleEulérien(G, v)

Afficher v ;
Appel de `promenade(g,v,t)` qui, d'une part, stocke dans `t` les sommets (sauf le point de départ `v`) dans l'ordre défini par un chemin maximal dans G (chemin qui doit commencer par v et qui ne doit pas passer deux fois par la même arête), et qui, d'autre part, supprime les arêtes parcourues.
for w dans L **do**
  CycleEulérien(G, w)
end for

Question 21. Exécuter l'algorithme à la main sur $k5$. Quel est le cycle eulérien parcouru ?
Question 22. (*) Implémenter une fonction `void cycle_eulerien(graphe &g)` qui affiche un cycle eulérien d'un graphe selon l'algorithme ci-dessus.
Question 23. (*) De même, implémenter un algorithme qui affiche un chemin eulérien d'un graphe lorsqu'il existe.
Question 24. Implémenter une variante de la fonction `cycle_eulerien` qui prend en argument supplémentaire un `vector<int> ordre` supposé vide au début, et qui stocke dans `ordre` les sommets dans l'ordre parcouru par le cycle eulérien (au lieu de seulement les afficher).
Question 25. (**) Quelle est la complexité de votre algorithme en fonction du nombre n de sommets du graphe ?
Question 26. (**) La fonction `voisin` renvoie toujours la bonne réponse, mais de manière peu efficace lorsqu'elle est appelée plusieurs fois. En stockant en mémoire le dernier voisin trouvé pour chaque sommet, implémenter une version de l'algorithme précédent de complexité $O(n^2)$.