Travaux Pratiques de Structures de Données n°8

Cours d'Informatique de Seconde Année

— Licence 2.2 —

Le sujet de ce TP porte sur les graphes orientés. Un graphe orienté est un ensemble de sommet et un ensemble d'arc orienté. Un arc orienté est un lien entre deux sommet dirigé vers l'un de ces deux sommet. La direction des arc est importante, un arc allant d'un sommet i à un sommet j est différent de l'arc allant du sommet j au sommet i. Si il existe un arc allant de i à j, j est appelé *voisin sortant* de i.

Un graphe orienté peut être implanté de deux façons différentes. Par matrice d'adjacence, où chaque arc allant d'un sommet i à un sommet j sera représenté par un 1 dans la case de ligne i et de colonne j, ou par listes d'adjacences où chaque sommet i est associé à une liste chainée d'entier qui représente les voisins sortant du sommet i.

Dans la suite, on suppose que le nombre de sommet d'un graphe ne dépasse pas `MAX_SOMMET`.

```
MAX_SOMMET 100
```

Dans les premiers exercices on travaille avec les listes d'adjacences, donc pour représenter un graphe on utilise la structure suivante.

```
typedef struct _cellule_
{
   struct _cellule_* suivant;
   int valeur;
}Cellule, *Liste; // liste chainée

typedef struct
{
   Liste* sommets;// tableau de liste chainée de taille nbSommet
   int nbSommet;
} GrapheL;
```

Une liste de voisin d'un sommet est triée.

▸ **Exercice 1.**

- *Écrire une fonction* `void AjouterArc(GrapheL A, int source, int cible)` *qui ajoute au graphe* `A` *un arc allant du sommet* `source` *au sommet* `cible`. *Si la valeur* `source` *ou* `cible` *est supérieur ou égale au nombre de sommet du graphe la fonction ne fait rien.*
- *Écrire une fonction* `void LireGraphe(FILE *f,GrapheL* A)` *qui fabrique un graphe à partir d'un fichier ouvert en lecture. La première ligne du fichier doit contenir le nombre de sommet du graphe* `A` *que l'on veut construire et les (ou la) secondes lignes doivent décrire les arcs sous la forme* `source -> cible`.

- *Écrire une fonction* `void SauvegarderGraphe(FILE *f,GrapheL A)` *qui fabrique un fichier à partir d'un graphe. Le contenu du fichier doit pouvoir être lue par la fonction* `LireGraphe`. *Chaque arc de* `A` *doit donner lieu à une ligne* `source -> cible\n`.
- *Écrire une fonction* `void LibererGraphe(GrapheL A)` *qui libère la place occupée par le graphe* `A`.

Dans l'exercice suivant on utilise la représentation de graphe par matrice d'adjacence. On définit deux nouveaux types.

```
typedef int Matrice[MAX_SOMMET][MAX_SOMMET];

typedef struct
{
   int nbSommet;
   Matrice adj;
} GrapheM;
```

▸ **Exercice 2.**
- *Écrire une fonction* `void RemplirMatrice(GrapheL original, GrapheM* copie)` *qui fabrique le graphe* `copie` *à partir du graphe* `original`.
- *Écrire une fonction* `void Multiplier(Matrice Resultat,Matrice Gauche, Matrice Droite, int n)` *qui effectue le produit de la matrice* `Gauche` *avec* `Droite`. *Le resultat du calcul est stocké dans la matrice* `Resultat`. *Les matrices sont supposées carrées de taille* `n`.
- *Écrire une fonction* `void ChercherChemin(GrapheM* graphe, Matrice m)` *qui remplit la matrice* `m`. *Après appel de la fonction la case* `m[i][j]` *doit contenir la longueur du plus petit chemin allant de* i *à* j *ou* -1 *s'il n'existe pas de chemin.*

▸ **Exercice 3.** *Écrire une fonction* `int TrierTopologi(GrapheL A,int tri[MAX_SOMMET])` *qui effectue un tri topologique des sommets du graphe et remplit le tableau* `tri` *selon le tri effectué. La fonction doit renvoyer* 1 *si le graphe n'est pas acyclique,* 0 *si le tri a pu être effectué.*

▸ **Exercice 4.** *Écrire une fonction* `void RechercherCircuit(GrapheL A)` *qui affiche tout les circuits du graphe (sans utiliser la matrice d'ajdacence). On peut écrire et utiliser une fonction récursive* `void RechercherCircuitR(GrapheL A,int sommet, int longueurPile, int* pile, int* sommetsVue)`. *On souhaite seulement afficher les circuits élémentaires (ne passant pas deux fois par le même sommet) et on considère que deux circuits qui passent tout les deux par les mêmes points dans le même ordre mais avec deux points de départ différents sont égaux.*

▸ **Exercice 5.**
- *Écrire une fonction* `void Retourner(GrapheL* transpose,GrapheL original)` *qui fabrique un graphe* `transpose` *à partir de* `original` *en retournant les arc de* `original`.
- *Écrire une fonction* `void RechercherCFC(GrapheL graphe,int resultat[])` *qui remplit le tableau* `resultat` *tel que après appel de la fonction on ait : deux sommets* i *et* j *appartiennent à une même composante fortement connexe de* `graphe` *si et seulement si les valeurs des cases* `resultat[i]` *et* `resultat[j]` *sont égales. La fonction doit utiliser la fonction* `Retourner` *et faire un ou plusieurs parcours en profondeur.*

▸ **Exercice 6.** *Écrire une fonction* `void Dijkstra(GrapheM* G,int origine,int T[MAX_SOMMET])` *qui remplit les cases du tableau T par les distances des sommets du graphe au sommet* `origine`. *Dans le cas ou la distance est infini (le sommet est inacessible à partir de* `origine`*) la case doit contenir la valeur -1. Vous devez utiliser l'algorithme de Dijkstra. On suppose que la matrice d'adjacence du graphe* `*G` *contient les longueurs des arcs : si la case* `G->adj[0][1]` *est non nulle, alors la valeur qu'elle contient est la longueur de l'arc qui va du sommet* `0` *vers le sommet* `1`.