Institut National des Sciences Appliquées **2019 - 2020**

Licence 3 Informatique Graphes et Algorithmes

T.P. 1 : Représentation d'un graphe - structure de données

Modalités pour chaque séance de TP :

- Les programmes seront développés de préférence **en langage C**. L'utilisation d'un autre langage ne sera acceptée que si la mémoire **dynamique** est gérée par l'utilisateur (allocation et libération explicites dans le code).
- Vos programmes doivent pouvoir être compilés et exécutés sous Windows **et** Linux.
- **Attention :** ne seront récupérés que les fichiers sources (.h et .c en 'C'). Par conséquent, les codes sources pour lesquels un **Makefile** (pour Linux) et un **ReadMe** auront été fournis seront **les seuls évalués**. Les archives contenant des *fichiers projets* générés à partir d'un éditeur quelconque (Eclipse, Code Blocks, NetBeans, Visual Studio, ...) **ne seront pas acceptées**. Vous devez fournir les informations nécessaires à l'enseignant pour qu'il puisse générer l'exécutable sans avoir à utiliser un logiciel particulier.

Évaluation (lire attentivement)

Votre travail sera évalué de la façon suivante :

- **1 note sur 10 pour le travail réalisé durant les 2 dernières séances ($\Rightarrow$ 1 note sur 20).**
- Le TP2 et TP3 doivent obligatoirement être déposés sur moodle avant la fin de la séance **par chaque étudiant** (même si vous travaillez en groupe). Les traveaux envoyés par mail ne seront pas acceptés.
- **Si votre code ne s'exécute pas et si vous n'avez mis aucun commentaire dans le code, ceci entraînera un 0.**
- Vous pouvez travailler en groupe de max 2 étudiants. Mettre en commentaire dans chaque fichiers les noms de tous les étudiants du groupe.

Une absence à un TP entraînera un 0, sauf si cette absence est justifiée (certificat médical ou raison **validée par l'enseignant**).

But du TP1 :

Développer une bibliothèque de manipulation de graphes **orientés simples** en utilisant une représentation d'un graphe par **matrice d'adjacence** et par **liste d'adjacence**. Ces structures de données seront la base du TP2 et du TP3. Le TP1 n'est pas directement pris en compte dans la notation mais sans lui impossible de faire les TP2 et 3, il ne faut donc pas le laisser de côté.

Décomposition du code :

Pour ce TP1, on stockera le code dans les fichiers suivants :

- `graphe_matrice.h`, `graphe_matrice.c` : fichiers contenant les différentes fonctions de manipulation via une matrice d'adjacence.
- `graphe_liste.h`, `graphe_liste.c` : fichiers contenant les différentes fonctions de manipulation via une liste d'adjacence.
- `main.c` : programme principal.

Pour simplifier la manipulation et le développement, on suppose **dans toute la suite** que les graphes disposent de n sommets numérotés de 0 à n-1. De plus le graphe sera noté avec $G = (S, A)$, où S représente l'ensemble des sommets et A l'ensemble des arcs.

Tests et validation

Il s'agit ici de compléter le fichier `main.c`, en proposant un menu à l'utilisateur pour qu'il puisse choisir ce qu'il souhaite faire.

1 Définition des types

1.1 Matrice d'adjacence

Le fichier d'en-tête de la manipulation d'un graphe via une matrice d'adjacence est donné partiellement ci-après (fichier `graphe_matrice.h`). **Attention**, on rappelle que la mémoire doit être gérée de façon dynamique.

```
typedef int SID; /* Renommage du type entier pour nommer les sommets
*/
typedef struct {
 int n;           /* Nombre de sommets */
 int **M;         /* Matrice d'adjacence */
                  /* M[i][j] == 1 s'il y a un arc entre i et j, 0 sinon */
} MATRICE;
```

1.2 Liste d'adjacence

Le fichier d'en-tête de la manipulation d'un graphe via des listes d'adjacence est donné partiellement ci-dessous (fichier `graphe_liste.h`) :

```
typedef int SID;
typedef struct maillon {
 SID s;                     /* Sommet stocké dans le maillon */
 struct maillon *suivant;  /* Pointeur vers le maillon suivant */
} MAILLON;

typedef struct {
 int n;                  /* Nombre de sommets */
 MAILLON **listes;       /* Liste d'adjacence */
} LISTE;
/* listes[i] :  liste chaînée contenant les sommets adjacents au
sommet i*/
```

2 Fonctions de base : matrice d'adjacence

Écrire les fonctions spécifiées ci-dessous dans le fichier `graphe_matrice.c`. Pour chaque fonction, vous rappellerez dans un commentaire son nom, son rôle ainsi que celui de ces paramètres, et vous donnerez une **évaluation de sa complexité** temporelle dans le pire des cas dans les commentaires dans de code du fichier (**pour chaques fonction**).

1. Réservation de l'espace mémoire nécessaire pour représenter un graphe comportant n sommets. Cette fonction initialisera également le graphe de sorte à ce qu'il soit initialement sans arc :
   `void reservation_en_memoire(int n, MATRICE *g);`

2. Libération de la mémoire occupée par le graphe g :
   `void liberation_memoire(MATRICE *g);`

3. Ajout d'un arc entre les sommets i et j dans le graphe g **si il n'existe pas déjà** :
   `void ajouter_arc(MATRICE *g, SID i, SID j);`

4. Retrait de l'arc entre les sommets i et j **si il existe** dans le graphe g :
   `void retirer_arc(MATRICE *g, SID i, SID j);`

5. Fonction qui retourne 1 si j est successeur de i dans le graphe g, 0 sinon :
   `int est_successeur(MATRICE *g, SID i, SID j);`

6. Génération de deux tableaux, *dplus* et *dmoins*, contenant pour chaque sommet i son degré sortant ($dplus[i]$) et son degré entrant ($dmoins[i]$) :
   `void calcul_degres(MATRICE *g, SID I, int *dplus, int *dmoins);`

7. Copie du graphe $g1$ dans le graphe $g2$:
   `void copie_graphe(MATRICE *g1, MATRICE *g2);`

Pour les fonctions suivantes de lecture et d'écriture dans un fichier, on utilisera un format unique : le premier entier dans le fichier est le nombre de sommets, puis viennent successivement (1 par ligne) des couples d'entiers représentant les sommets reliés par les arcs (sommet origine puis sommet destination).

8. Lecture et rangement dans g du graphe contenu dans le fichier de nom physique *nom* :
   `void lire_graphe(char *nom, MATRICE *g);`

9. Écriture du graphe g dans le fichier de nom physique *nom* :
   `void ecrit_graphe(MATRICE *g, char *nom);`

3 Fonctions de base : liste d'adjacence

Écrire les fonctions spécifiées ci-dessous dans le fichier `graphe_liste.c`. Pour chaque fonction, vous rappellerez dans un commentaire son nom, son

rôle ainsi que celui de ces paramètres, et vous donnerez une évaluation de sa complexité temporelle dans le pire des cas.

1. Réservation de l'espace mémoire nécessaire pour représenter un graphe comportant n sommets. Cette fonction initialisera également le graphe de sorte à ce qu'il soit initialement sans arête :
   `void reservation_en_memoire(int n, LISTE *g);`

2. Libération de la mémoire occupée par le graphe g :
   `void liberation_memoire(LISTE *g);`

3. Ajout d'une arête entre les sommets i et j dans le graphe g **si elle n'existe pas déjà** :
   `void ajouter_arete(LISTE *g, SID i, SID j);`

4. Retrait de l'arête entre les sommets i et j **si elle existe** dans le graphe g :
   `void retirer_arete(LISTE *g, SID i, SID j);`

5. Affichage d'un graphe sous forme textuelle (une ligne par sommet avec le sommet en début de ligne et la liste de ses sommets adjacents ensuite) :
   `void affichage(LISTE *g);`

6. Fonction qui retourne 1 si j est **adjacent** à i dans le graphe g, 0 sinon :
   `int est_adjacent(LISTE *g, SID i, SID j);`

Pour les fonctions de lecture et d'écriture dans un fichier, on utilisera le format suivant : le premier entier dans le fichier est le nombre de sommets, puis viennent successivement (1 par ligne) des couples d'entiers correspondant aux extrémités d'une arête du graphe.

7. Lecture et rangement dans g du graphe contenu dans le fichier de nom physique *nom* :
   `void lire_graphe(char *nom, LISTE *g);`

8. Écriture du graphe g dans le fichier de nom physique *nom* :
   `void ecrit_graphe(LISTE *g, char *nom);`