

Durée prévue : 2 séances d'1h30.

Les objectifs de ce sujet de TP sont :

Révisions de C listes chaînées ;

Algorithmique représentation des graphes par listes de successeurs.

Les fonctions à implémenter (ou qui le sont déjà) sont documentées à l'adresse
<https://www.math.univ-paris13.fr/~rousselin/graphe-4/>

IMPORTANT : Ce sujet est plus ou moins facile selon votre habitude des listes chaînées. Mais il est très important car il introduit la bibliothèque graphe dans sa version finale, celle que vous utiliserez jusqu'à la fin du module. Il est recommandé de LIRE ATTENTIVEMENT L'ÉNONCÉ et de prendre son temps pour bien comprendre la représentation par liste de successeurs. Certaines questions ne nécessitent pas que vous codiez, elles sont aussi importantes que les autres. Ne vous jetez pas sur vos claviers ! Sortez plutôt des feuilles de brouillon et de quoi noter !

1 Mise en place

Créer un répertoire dédié à ce sujet de TP, télécharger l'archive correspondante sur l'ENT (`tp3_fichiers.tar.gz`) et l'extraire dans ce répertoire. Vous devriez voir les fichiers suivants :

<code>graphe-4.h</code>	nouvelle version de l'interface de la bibliothèque graphe
<code>graphe-4.c</code>	fichier source correspondant
<code>test-1.c</code>	première fonction <code>main</code> de test
<code>test-2.c</code>	deuxième fonction <code>main</code> de test
<code>test-3.c</code>	troisième fonction <code>main</code> de test
<code>test-4.c</code>	quatrième fonction <code>main</code> de test
<code>Makefile</code>	pour compiler ce petit monde sans effort

Taper la commande `make`, pour vérifier qu'il n'y a pas d'erreur de compilation (bien que les programmes produits ne soient pas intéressants).

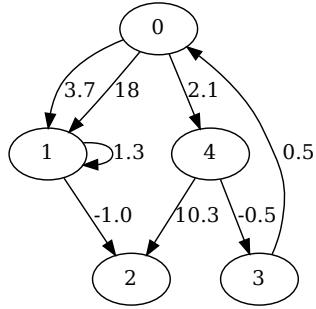
2 Listes de successeurs

2.1 Principe de la représentation

La plupart des algorithmes sur les graphes sont plus efficaces lorsque ceux-ci sont représentés avec des listes de successeurs plutôt qu'avec une matrice d'adjacence.

Pour chaque sommet, on dresse la liste de ses successeurs (éventuellement répétés en cas d'arcs/arêtes multiples), avec, dans le cas d'un graphe arc/arête-valué, la valeur de l'arête.

Prenons l'exemple du graphe orienté et arête-valué G suivant :



sommet v	successeurs, avec valeur de l'arc associée
0	(1, 3.7); (1, 18); (4, 2.1)
1	(2, -1); (1, 1.3)
2	
3	
4	(3, -0.5); (2, 10.3)

Remarquer que, par rapport à la représentation par matrice d'adjacence :

1. l'occupation en mémoire est proportionnelle au nombre d'arêtes (et non au carré du nombre de sommets), ce qui peut être un réel avantage pour les graphes peu denses (c'est-à-dire qui ont beaucoup moins que n^2 arêtes);
2. on peut considérer des arcs ou arêtes multiples de valeurs différentes.

Autre remarque : nous n'avons mis aucune contrainte sur l'ordre de la liste des successeurs.

Dans le cas de graphes non orientés, un successeur du sommet v est simplement un *voisin* du sommet v .

Question 1:

1. Donner les listes de successeurs du graphe non orienté complet à 4 sommets K_4 (où les valeurs associées aux arêtes sont arbitrairement fixées à 1.0).
2. Dans un graphe non orienté, combien de fois chaque arête est-elle présente dans les listes de successeurs ? Que dire des boucles ?

--- * ---

Pour implémenter ces listes de successeurs en C, nous allons utiliser un tableau de listes chaînées.

Par soucis de généralité, la bibliothèque **graphe-4** considère que tous les graphes sont arc/arête-valués. Lorsqu'on traite des graphes qui ne le sont pas, il suffit d'ignorer les valeurs des arcs/arêtes et de les fixer toutes à une constante arbitraire (1.0 semble un bon choix).

2.2 Les maillons de successeurs

Un maillon de successeur contient trois données :

sommet de type **int**, le successeur du sommet courant ;

valeur de type **double**, la valeur (suivant le contexte, le coût, la capacité, ...) de l'arc/arête allant du sommet courant à son successeur ;

suivant un pointeur vers le prochain maillon de successeur du sommet courant ou bien le pointeur nul (**NULL**) si c'est le dernier successeur de la liste.

Ceci est mis en œuvre dans le type **msuc** (pour « maillon de successeur ») suivant :

```

typedef struct msuc {
    int sommet;
    double valeur;
    struct msuc *suivant;
} msuc;
```



```

msuc *msuc_suivant(msuc *m);
int msuc_sommet(msuc *m);
double msuc_valeur(msuc *m);
```

Les fonctions `msuc_suivant`, `msuc_sommet` et `msuc_valeur` sont simplement des accesseurs pour les différents champs de la structure.

La figure 1 donne une représentation schématique de la liste des successeurs du sommet 0 dans le graphe G ci-dessus (les adresses sont fantaisistes).



FIGURE 1 – Schéma d'une liste de successeurs en mémoire

Question 2: Donner, en fonction de la taille des `int`, des `double` et des pointeurs, la taille d'un maillon de successeur, puis de toute la liste des successeurs d'un sommet.

--- * ---

2.3 Le tableau de pointeurs vers des maillons de successeurs

On stocke dans un tableau `tab_sucs` à n éléments, pour chaque sommet i du graphe, l'adresse `tab_sucs[i]` de son premier maillon de successeur (ou le pointeur `NULL` si la liste de ses successeurs est vide).

La figure 2 représente ce tableau ainsi que les listes de tous les successeurs du graphe G considéré en début d'énoncé. Le type graphe est maintenant défini par la structure suivante :

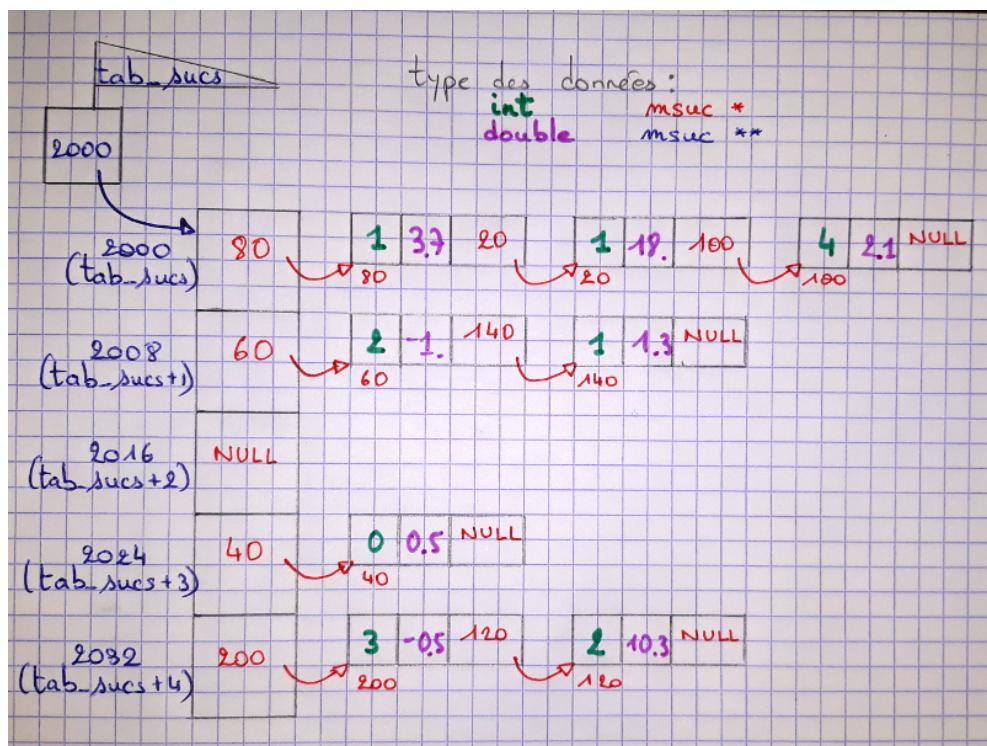


FIGURE 2 – Schéma d'une liste de successeurs en mémoire

```
struct s_graphe {
    int n; /* nombre de sommets du graphe (doit être >=0) */
    int m; /* nombre d'arêtes du graphe */
    msuc **tab_sucs; /* tableau de pointeurs vers les listes de successeurs du graphe */
    int est_or; /* 1 si le graphe est orienté, 0 sinon */
};

typedef struct s_graphe graphe;
```

Question 3:

1. Pour un graphe orienté, donner en fonction de n , de m et des tailles des `int`, des `double` et des pointeurs, l'occupation totale en mémoire d'un graphe avec cette représentation.
2. Modifier votre formule pour un graphe non orienté.

--- * ---

3 Enfin, on code !

3.1 Initialisation et ajout d'arc pour un graphe orienté

Question 4: Définir dans le fichier `graphe-4.c` la fonction

```
int graphe_stable(graphe* g, int n, int est_or);
```

qui initialise le graphe d'adresse `g` au graphe à `n` sommets, **sans arc/arête**, non orienté si `est_or` vaut 0 et orienté sinon.

Environ 10 lignes de code.

--- * ---

Question 5: Définir la fonction

```
int graphe_ajouter_arc(graphe *g, int v, int w, double val);
```

qui prend comme arguments l'adresse `g` d'un graphe, qu'on suppose *sans le vérifier* orienté, deux sommets `v` et `w` ainsi qu'une valeur `val` et ajoute l'arc (v, w) , de valeur `val` au graphe, c'est-à-dire ajoute un maillon successeur à la liste des successeurs du sommet `v`.

Ce nouveau maillon apparaîtra, pour un soucis d'efficacité, en début de liste.

La valeur de retour est `-1` si l'allocation mémoire pour le nouveau maillon a échoué, 0 sinon. Penser à mettre à jour le nombre d'arcs du graphe.

Environ 10 lignes de code.

--- * ---

Question 6: Tester ces deux premières fonctions en compilant et en exécutant le programme `test-1`. Vous devriez voir apparaître à l'écran le graphe orienté donné en début de sujet.

--- * ---

Question 7: Définir la fonction

```
void graphe_detruire(graphe *g);
```

qui libère la mémoire allouée pour tous les maillons de successeurs ainsi que pour le tableau `tab_sucs`. Dans le fichier `test-1.c`, décommenter la ligne contenant l'appel à cette fonction, compiler et exécuter. Si l'exécution révèle une erreur de segmentation, c'est qu'il y a une erreur à corriger dans cette fonction !

Une douzaine de lignes de code.

--- * ---

3.2 Initialisation et ajout d'arête pour un graphe non orienté

Question 8: Écrire la définition de la fonction

```
int graphe_ajouter_arete(graphe *g, int v, int w, double val);
```

qui ajoute l'arête $\{v, w\}$ de valeur `val` au graphe, supposé non orienté, d'adresse `g` et retourne -1 en cas d'échec d'allocation mémoire pour le ou les nouveaux maillon(s) et 0 sinon. On pourra utiliser la fonction `graphe_ajouter_arc` en corrigeant si besoin le champ `m` (nombre d'arêtes) du graphe.

Une douzaine de lignes de code.

--- * ---

Question 9: Tester, en compilant et en exécutant le programme `test-2` votre fonction d'ajout. Vous devriez voir apparaître à l'écran le graphe complet non orienté K_4 .

--- * ---

3.3 Accesseurs en lecture

Les accesseurs en lecture de la bibliothèque `graphe-4` sont les suivants :

```
int graphe_est_or(graphe *g); /* déjà définie */
int graphe_get_n(graphe* g); /* déjà définie */
int graphe_get_m(graphe* g); /* déjà définie */
msuc *graphe_get_prem_msuc(graphe *g, int v); /* à faire ! */
int graphe_get_multiplicite_arc(graphe* g, int v, int w); /* à faire ! */
int graphe_get_multiplicite_arete(graphe* g, int v, int w); /* à faire ! */
int graphe_get_degre_sortant(graphe* g, int v); /* à faire ! */
int graphe_get_degre_entrant(graphe* g, int v) /* à faire ! */
int graphe_get_degre(graphe* g, int v); /* à faire ! */
```

Comme indiqué ci-dessus, les trois premiers, évidents, sont déjà écrits.

Question 10: Définir la fonction

```
msuc *graphe_get_prem_msuc(graphe *g, int v);
```

qui retourne un pointeur vers le premier maillon de la liste des successeurs du sommet `v` dans le graphe d'adresse `g` (ou le pointeur nul si cette liste est vide). **1 ligne de code.**

--- * ---

Question 11:

1. Définir la fonction

```
int graphe_get_multiplicite_arc(graphe *g, int v, int w);
```

qui retourne le nombre d'arcs d'extrémité initiale `v` et d'extrémité terminale `w` dans le graphe d'adresse `g`, supposé orienté. **Environ 5 lignes de code, utiliser la fonction de la question précédente et les fonctions `msuc_sommet` et `msuc_suivant`.**

2. En utilisant cette fonction, définir, en 1 ligne,

```
int graphe_get_multiplicite_arete(graphe *g, int v, int w);
```

3. Que dire de la complexité de votre algorithme ? Comparer avec celui utilisé lorsque le graphe est représenté par une matrice d'adjacence (comme dans `graphe-2` par exemple).

--- * ---

Question 12:

1. Définir la fonction

```
int graphe_get_degre_sortant(graphe* g, int v);
```

en utilisant la liste des successeurs de v . Pour un graphe non orienté, le résultat est simplement le degré du sommet v .

Environ 5 lignes de code.

2. Que dire de la complexité de cet algorithme ? Comparer avec le cas où le graphe est représenté par sa matrice d'adjacence.
3. Mêmes questions pour la fonction

```
int graphe_get_degre_entrant(graphe* g, int v);
```

Que pourrions-nous faire pour en améliorer la complexité dans le cas orienté ?

4. Définir la fonction

```
int graphe_get_degre(graphe* g, int v);
```

en utilisant les fonctions précédentes. Pour un graphe non orienté, on fera attention à ce que la complexité de l'algorithme reste convenable.

Environ 5 lignes de code.

--- * ---

Question 13:

Tester vos fonctions à l'aide des programmes `test-3` (cas orienté) et `test-4` (cas non orienté). Dans les deux cas, le graphe est celui représenté en début d'énoncé (en oubliant l'orientation des arcs pour le cas non orienté).

--- * ---

3.4 Parcours des arcs ou des arêtes

Question 14:

On fait le bilan :

1. Comment parcourir les arcs ou arêtes issus du sommet v ?
2. Comment parcourir tous les arcs d'un graphe orienté ? Toutes les arêtes (une fois chaque arête) d'un graphe non orienté ?
3. Comparer avec le cas d'un graphe représenté par sa matrice d'adjacence.

--- * ---

3.5 Suppressions d'arcs ou d'arêtes

Question 15:

1. Définir la fonction

```
int graphe_supprimer_arc(graphe *g, int v, int w, double val);
```

qui supprime, s'il en existe, un arc d'extrémité initiale v , d'extrémité terminale w et de valeur val dans le graphe d'adresse g . En cas de succès (un tel arc existe), la fonction retourne 0. Si un tel arc n'existe pas, la fonction ne modifie pas le graphe et retourne -1.

On veillera à libérer la mémoire désormais inutile.

Environ 15 lignes de code.

2. Définir la fonction

```
int graphe_supprimer_arc(graphe *g, int v, int w, double val);
```

qui traite le cas d'un graphe non orienté, en utilisant la fonction précédente.

Environ 10 lignes de code.

--- * ---

4 Parcours

Enfin, on exploite les fonctions de la bibliothèque pour faire tourner des algorithmes de graphe.

Question 16: Déclarer, définir et tester une fonction `graphe_est_biparti` qui prend en entrée un graphe et renvoie 1 si le graphe est biparti, 0 sinon.

--- * ---

Question 17: Déclarer, définir et tester une fonction `graphe_distance` qui prend en entrée un graphe et un sommet de ce graphe, et renvoie un tableau associant à chaque sommet du graphe sa distance au sommet passé en paramètre (NULL en cas de problème d'allocation mémoire).

--- * ---

Question 18: Déclarer, définir et tester une fonction `graphe_get_excentricite` qui prend en entrée un graphe et renvoie son excentricité.

--- * ---

Question 19: Déclarer, définir et tester une fonction `graphe_articulation` qui prend en entrée un graphe et renvoie la liste de ses points d'articulation.

--- * ---