

Il existe différentes manières de représenter les graphes. Dans un premier temps, on choisit de les représenter par leur matrice d’adjacence.

1 Structure de données (voir fichier graphe-1.h)

Dans ce TP introductif, un graphe est caractérisé par :

- son nombre n de sommets : **son ensemble V de sommets est alors $\{0, 1, \dots, n - 1\}$** ;
- sa matrice `adj` d’adjacence ;
- son nombre m d’arêtes.

De plus, l’ordre des graphes représentés est borné par une constante `GRAPHE_ORDRE_MAX`.

Ceci est défini dans le fichier `graphe-1.h` par les lignes suivantes :

```
#define GRAPHE_ORDRE_MAX 9 /* nombre maximal de sommets dans un graphe */
struct s_graphe {
    int n; /* nombre de sommets du graphes
           doit être compris entre 1 et GRAPHE_ORDRE_MAX
           V = {0 ,..., n - 1} */
    int m; /* nombre d'arêtes du graphes (entier naturel) */
    int adj[GRAPHE_ORDRE_MAX][GRAPHE_ORDRE_MAX];
    /* matrice d'adjacence du graphe (doit être cohérente avec m ) :
       seules les lignes et colonnes d'indice 0 à n - 1 sont à lire */
};

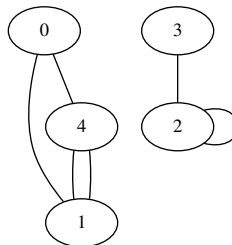
typedef struct s_graphe graphe;
```

2 Fonctions de manipulation (voir fichier graphe-1.h)

Pour cette structure, on adoptera les règles de gestion suivantes :

- un graphe peut être initialisé (pour l’instant, au graphe stable), pour un nombre de sommets donné ;
- on peut ensuite ajouter ou supprimer des arêtes du graphe ;
- en revanche l’ordre du graphe ne peut être modifié sans ré-initialiser totalement le graphe.

Question 1: Donner les valeurs des champs de la structure, si la variable `g` de type `graphe` représente le graphe ci-dessous.



Combien (en nombre d’`int`) de place en mémoire prend une variable de type `graphe` ?

--- * ---

2.1 Initialisation

Étant donné un graphe et un ordre `n` à valeur dans $\{0, 1, \dots, \text{GRAPHE_ORDRE_MAX}\}$, on souhaite pouvoir initialiser le graphe en un stable, c'est à dire un *graphe sans arête* d'ordre `n`.

Question 2: Implémenter dans le fichier `graphe-1.c` la fonction `int graphe_stable(graphe* g, int n)` qui permet d'initialiser le graphe pointé par `g` en un graphe stable et retourne 0 en cas de succès, -1 en cas d'échec (valeur de `n` trop grande ou négative).

--- * ---

2.2 Ajout / suppression d'arête

Étant donné un graphe, on souhaite pouvoir :

- ajouter une arête $\{v, w\}$ spécifique ;
- supprimer une arête $\{v, w\}$ spécifique.

Question 3: Implémenter les fonctions `void graphe_ajouter_arete(graphe* g, int v, int w)` et `int graphe_supprimer_arete(graphe* g, int v, int w)` correspondantes.

La valeur de retour de `graphe_supprimer_arete` est 0 en cas de succès, -1 en cas d'échec.

--- * ---

2.3 Accesseurs en lecture

Étant donné un graphe, on souhaite pouvoir connaître ou savoir :

- son nombre de sommets ;
- son nombre d'arêtes ;
- le nombre de fois que deux sommets donnés v et w sont adjacents ;
- le degré d'un sommet v donné ;
- s'il est ou non simple (c'est-à-dire sans boucle ni arête multiple).

Question 4: Implémenter les fonctions correspondantes dans le fichier `graphe-1.c`.

--- * ---

2.4 Entrées / Sorties

La fonction `void graphe_afficher(graphe* g)` affiche sur la console :

- la valeur des champs `n` et `m` de ce graphe, et
- sa matrice d'adjacence, avec en plus les degrés de chaque sommet.

Cette fonction, un peu longue à écrire, est déjà implémentée pour vous. N'hésitez toutefois pas à parcourir son code et si nécessaire, à poser des questions à votre chargé de TP.

3 Test des premières fonctions

À présent que les fonctions sont écrites, il s'agit d'en vérifier le bon fonctionnement. Le fichier `main.c` comporte déjà les appels de fonctions permettant d'initialiser le graphe `g` au graphe représenté dans la première question.

1. Compiler et exécuter un programme qui affiche ce graphe (il suffit de lancer la commande `make`, puis le programme `./graphe-test`).
2. Tester la fonction `graphe_supprimer_arete` (cas d'usage : l'arête existe en 0, 1 ou plusieurs exemplaires).
3. Tester la fonction `graphe_stable` (cas d'usage : `n` dans et hors bornes).
4. Tester la fonction `graphe_est_simple` sur le graphe déjà écrit d'abord, puis en le modifiant de manière à ce qu'il devienne simple.

4 Fonctions supplémentaires

On vous demande maintenant d'écrire de nouvelles fonctions permettant d'initialiser un graphe ou d'en étudier certaines caractéristiques.

Question 5: Implémenter (et penser à déclarer) la fonction `int graphe_complet(graphe *g, int n)`, qui étant donné un nombre n de sommets initialise le graphe pointé par g au graphe complet à n sommets, c'est-à-dire au graphe dans lequel deux sommets distincts sont toujours reliés par une arête. La valeur de retour sera 0 en cas de succès et -1 en cas d'échec (mauvaise valeur de n). Penser à tester votre fonction.

--- * ---

Rappels : l'entête `stdlib.h` de la bibliothèque standard du C contient la déclaration des fonctions `int rand()` et `void srand(unsigned seed)` ainsi que la constante `RAND_MAX`. Les appels successifs à `rand` retournent des nombres entiers pseudo-aléatoires compris entre 0 et `RAND_MAX`.

La fonction `srand` initialise la graine du générateur de nombres aléatoires à son argument (la même graine donne lieu aux mêmes valeurs de retour successives de `rand`).

L'entête `time.h` de la bibliothèque standard du C contient la déclaration de la fonction `time_t time(time_t *t)` qui, appelé avec l'argument `NULL`, indique le nombre de seconde depuis le 1^{er} janvier 1970.

Tout cela mis ensemble permet d'écrire le bout de code suivant, qui affiche 10 résultats de pile ou face équilibré à la suite.

```
#include <stdio.h>
#include <stdlib.h> /* rand, srand, RAND_MAX */
#include <time.h> /* time pour initialiser la graine */

int main()
{
    int i;
    double U;
    /* Initialisation du générateur de nombre aléatoire :
     * la graine est donnée par le nombre de secondes depuis
     * le 1er janvier 1970, et sera donc différente à chaque exécution (ou
     * presque). */
    srand(time(NULL));
    /* 10 tirages de pile ou face (affichage sur le terminal) */
    for (i = 0; i < 10; ++i) {
        U = (double) rand() / RAND_MAX; /* U aléatoire entre 0 et 1 */
        if (U <= .5)
            puts("PILE !");
        else
            puts("FACE !");
    }
    return EXIT_SUCCESS;
}
```

Question 6: Implémenter la fonction `int graphe_aleatoire(graphe *g, int n, double p)`, qui doit initialiser le graphe pointé par g à un graphe aléatoire à n sommets, où pour tous sommets u et v distincts, l'arête $\{u, v\}$ est présente avec probabilité p (et absente avec probabilité $1 - p$), ceci de façon indépendante pour chaque couple de sommets. La valeur de retour est encore 0 en cas d'échec, -1 si n n'est pas adapté et -2 si p n'est pas compris entre 0 et 1. Tester votre fonction.

--- * ---

Question 7: Implémenter la fonction `int graphe_cycle(graphe *g, int n)` qui initialise le graphe pointé par g au graphe *cycle* de taille n , c'est-à-dire au graphe où les arêtes sont $\{0, 1\}$, $\{1, 2\}$, $\{2, 3\}$, ..., $\{n - 1, 0\}$.

--- * ---

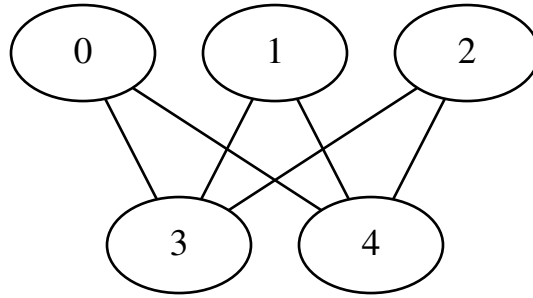


FIGURE 1 – Le graphe biparti $K_{3,2}$.

Un graphe biparti, est un graphe dont l'ensemble V des sommets peut être partitionné en deux parties W et Z telles que toute arête a exactement une extrémité dans W et l'autre dans Z .

Un graphe biparti complet est un graphe biparti simple où, pour tout sommet w de W et tout sommet z de Z , l'arête $\{w, z\}$ est présente dans le graphe. Ce graphe est noté $K_{m,p}$, où m est le cardinal de W et p celui de Z . La figure 1 représente $K_{3,2}$.

Question 8: Implémenter la fonction `int graphe_biparti(graphe *g, int m, int p)` qui initialise le graphe pointé par `g` au graphe biparti complet $K_{m,p}$. On supposera que la partie notée W ci-dessus est $\{0, 1, 2, \dots, m-1\}$ et la partie notée Z ci-dessus est $\{m, m+1, \dots, m+p-1\}$ (comme sur la figure 1). On retournera 0 en cas de succès, -1 si les valeurs de m et de p ne conviennent pas.

--- * ---

On rappelle qu'un graphe $H = (V, F)$ est un sous-graphe partiel d'un graphe $G = (V, E)$ si F est inclus dans E (remarquez que les ensembles de sommets sont les mêmes).

Question 9: Implémenter la fonction `int graphe_est_sousgraphe_partiel(graphe *g, graphe *h)` qui retourne 1 si le graphe pointé par `h` est un sous-graphe partiel du graphe pointé par `g` et 0 sinon.

--- * ---

Question 10: Implémenter la fonction `int graphe_contient_triangle(graphe *g)` qui retourne 1 si le graphe pointé contient un triangle (c'est-à-dire un cycle de longueur 3) et 0 sinon.

--- * ---

Question 11: Cette question est *pour l'instant* plus délicate et est en bonus.

Implémenter la fonction `int graphe_est_connexe(graphe *g)` qui retourne 1 si le graphe pointé par `g` est connexe, et 0 sinon.

--- * ---