M23 Graphes - épisode 1

Dans le cadre de ce module, nous allons voir comment on peut représenter un graphe en mémoire et écrire des algorithmes pour construire, parcourir, déterminer des propriétés, colorier, calculer des chemins, etc. sur ces graphes.

Le langage de programmation que nous allons utiliser est le C.

---

1 - Compilation / Exécution

Dans le dossier `m23` fourni, vous trouvez des fichiers source (`.h` et `.c`) ainsi qu'un `Makefile`.

Le code fourni compile en l'état. À l'exécution de la commande `make`, deux dossiers sont créés :

- `build` : contient les fichiers `.o` produits par la compilation de chaque fichier `.c`
- `bin` : contient l'exécutable `m23`

Pour exécuter le programme, vous pouvez lancer la commande `./bin/m23`.

> ***Compiler et exécuter le programme.***

Bien entendu, vous devez modifier la fonction `main` au fil des exercices afin de tester l'ensemble des fonctionnalités développées.

---

2 - Structure de graphe

On commence tout d'abord par se mettre d'accord sur quelques types de base, déclarés dans le fichier `graphe.h`.

Un sommet est représenté par un entier :

```
typedef size_t sommet;
```

Les sommets d'un graphe sont nommés par les entiers compris entre 0 et l'ordre du graphe - 1.

Une arête est définie par ses 2 sommets incidents, `s1` et `s2`.

> ***Compléter la déclaration de la `struct arete`.***

La structure de données de graphe que nous allons mettre en oeuvre est une *liste d'arêtes*.

Pour stocker la liste d'arêtes, on utilise un tableau d'arêtes, c'est-à-dire une zone mémoire dont la taille est un multiple de la taille d'une arête. Cette zone mémoire va devoir grandir au fur et à mesure de l'ajout d'arêtes dans le graphe. Comme il serait inefficace de réallouer la zone mémoire à chaque ajout d'arête, on va mettre en place un mécanisme d'allocation anticipée : la zone mémoire est initialisée avec une taille donnée, puis à chaque ajout d'arête, s'il reste de la place, on stocke la nouvelle arête, s'il n'y a plus de place, on commence par allouer une nouvelle zone plus grande, on y copie le contenu de la zone initiale avant de la libérer.

Au sein de notre structure `graphe`, cela se traduit par plusieurs champs :

- `ordre` : l'ordre du graphe (de type `size_t`)
- `aretes` : un pointeur vers le tableau d'arêtes
- `aretes_capacite` : la taille du tableau d'arêtes (sa capacité)
- `nb_aretes` : le nombre d'arêtes effectives (nécessairement inférieur ou égal à la capacité du tableau)

Compléter la déclaration de la `struct graphe`.

---

3 - Fonctions sur les graphes

Dans le fichier `graphe.h`, des prototypes de fonctions sont déjà déclarés. Les corps de toutes ces fonctions sont déclarés (mais vides) dans le fichier `graphe.c`. Des commentaires sont donnés dans le code afin de préciser ce qui est attendu.

Écrire le corps de ces fonctions.

Une fois ces fonctions écrites, on pourra écrire par exemple :

```
graphe g; // déclaration
init_graphe(&g);
for (size_t i = 0; i < 5; ++i)
    ajouter_sommet(&g);
printf("ordre = %zu\n", ordre(&g));
free_graphe(&g);
```

---

4 - Premiers algorithmes

Dans le fichier `algos.h`, des prototypes de fonctions sont déjà déclarés.

Écrire le corps des fonctions suivantes :

- `size_t degre(graphe const *g, sommet s);`
- `bool est_regulier(graphe const *g);`
- `void afficher(graphe const *g);`
- `void generer_complet(graphe *g, size_t ordre);`

La fonction `afficher` devra générer une sortie sur le modèle suivant :

```
# sommets = 5
# aretes  = 5
--SOMMETS--
0 (degre: 2) <-> 2 3
1 (degre: 1) <-> 4
2 (degre: 2) <-> 0 3
3 (degre: 3) <-> 0 2 4
4 (degre: 2) <-> 1 3
--ARETES--
0 - 2
0 - 3
1 - 4
2 - 3
3 - 4
```

La fonction `generer_complet` devra `initialiser` le graphe `g`, y insérer un nombre de sommets correspondant à l'ordre souhaité, puis créer l'ensemble des arêtes.

Pour information, si on a la définition de structure suivante :

```c
typedef struct resultat
{
    int note;
    char* module;
} resultat;
```

alors on peut initialiser une variable de ce type de la manière suivante :

```c
resultat e1 = {12, "M23"}; // ordre des champs à respecter

resultat e2 = {
    .module = "M21", // initialisation "nommée"
    .note = 2        // ordre arbitraire
};

fonction((resultat){42, "M22"}); // définition d'une variable "à la volée"
                                 // lors du passage en paramètre
```

Une fois ces fonctions écrites, votre `main` pourra ressembler par exemple à cela :

```c
graphe g;
init_graphe(&g);

generer_complet(&g, 5);

printf("AFFICHE GRAPHE\n");
printf("==============\n");
afficher(&g);

if (est_regulier(&g))
    printf("Le graphe est régulier\n");
else
    printf("Le graphe n'est pas régulier\n");

free_graphe(&g);
```