La bibliothèque `graph_mat-1`

(Pierre Fouilhoux, Pierre Rousselin, Antoine Rozenknop)

Le but de ce sujet de TP est d'implémenter une bibliothèque de manipulation de graphes représentés par leurs matrices d'adjacence.

1 Quelques rappels sur `malloc`

La fonction `void *malloc(size_t size)` déclarée dans `stdlib.h` demande au système d'exploitation d'allouer `size` **octets** et renvoie un pointeur vers la mémoire nouvellement allouée. Si la valeur de retour est `NULL`, c'est que cette allocation a échoué[1]. Le type `size_t` est un type **entier non signé** qui dépend de l'implémentation (souvent `unsigned long`).

La fonction `void free(void *ptr)` libère une zone mémoire pointée par `ptr`, **où `ptr` a été obtenu par un appel antérieur à `malloc`** (ou `calloc` ou `realloc`). Si `ptr` vaut `NULL`, `free` n'a simplement aucun effet.

On considère le programme suivant :

```c
#include <stdlib.h> /* malloc, free et exit */
#include <stdio.h>
#include <string.h> /* pour memset */
#define TAILLE 50000000UL
#define NUM 100 /* puis 10 avec valgrind */
int main()
{
        int *tab, i;
        for (i = 0; i < NUM; ++i) {
                tab = malloc(TAILLE * sizeof(int));
                if (tab == NULL) {
                        perror("malloc");
                        exit(-1);
                }
                /* remplir tous les octets de tab avec l'octet i */
                memset(tab, i, TAILLE * sizeof(int));
                printf("Itération numéro %d ok\n", i);
                /* free(tab); */
        }
        printf("Succès ! %d mallocs de taille %zu\n", NUM, TAILLE);
        return 0;
}
```

Ce programme dont le fichier source, appelé `trop_de_malloc.c` est fourni, est à lire attentivement. En résumé, il cherche à créer `NUM` tableaux contenant `TAILLE` entiers de type de `int`. Chacun de ces tableaux étant initialisé à l'aide de l'indice de boucle[2].

1. Malheureusement, la réciproque est fausse...
2. Là aussi, il y a des subtilités qui dépendent de la machine et il vaut mieux pour ce test « utiliser » la mémoire allouée

Exercice 1 :

1. Compiler ce programme avec la commande
   `$ gcc -Wall -g trop_de_malloc.c -o trop_de_malloc`
   (sous un macOS récent, remplacer `gcc` par `clang`) et l'exécuter (commande `./trop_de_malloc`). Que se passe-t-il ?[3]
2. Recommencer après avoir décommenté la ligne qui contient l'appel `free(tab)`. Qu'est-ce qui a changé ?
3. Le programme `valgrind` permet d'analyser l'exécution d'un programme (en particulier sa gestion de la mémoire). **Son utilisation est vivement recommandée.** Recommenter l'instruction `free(tab)`, puis passer la valeur de `NUM` à `10`, puis lancer la commande
   `$ valgrind ./trop_de_malloc`
   Lire attentivement les messages imprimés sur le terminal.
4. Recommencer la question précédente en décommentant l'instruction `free(tab)`.

--- * ---

2 Un prototype de notre bibliothèque

Notre but est d'écrire une bibliothèque de graphes représentés par matrices d'adjacences, où ces matrices sont des tableaux alloués dynamiquement.

Nous avons commencé à écrire un premier prototype dans le programme suivant nommé `prototype_graph_mat.c` (et disponible sur l'ENT).

```c
#include <stdio.h>
#include <stdlib.h>

struct graphe {
        unsigned n; /* ordre du graphe */
        unsigned *adj; /* matrice d'adjacence */
};

void afficher(struct graphe *g);

int main()
{
        unsigned n = 4;
        struct graphe *g = malloc(sizeof(struct graphe));

        g->n = n;
        g->adj = calloc(n * n, sizeof(unsigned));

        g->adj[2 * n + 1] = g->adj[1 * n + 2] = 1;

        afficher(g);
```

3. Si tout semble se passer normalement, augmenter `NUM` et/ou `TAILLE`.

```
        return 0;
}

void afficher(struct graphe *g)
{
        unsigned v, w;
        for (v = 0; v < g->n; ++v)
                for (w = 0; w < g->n; ++w) {
                        printf("%3u", g->adj[v * g->n + w]);
                        if (w != g->n - 1)
                                printf(" ");
                        else
                                printf("\n");
                }
}
```

Exercice 2 :

1. Compiler ce programme et l'exécuter.
2. Que signifie `g->n` ? Pourquoi utiliser la flèche `->` et non un point (`.`) ?
3. La fonction `void *calloc(size_t nmemb, size_t size)` retourne un pointeur vers une zone mémoire de `nmemb` cases, chacune de taille `size`, **initialisées à zéro.** Pourquoi demandons-nous `n * n` cases ? Pourquoi sont-elles initialisées à 0 ? À quoi cela correspond-il pour notre graphe ?
4. Malheureusement, il n'y a pas vraiment de syntaxe agréable pour traiter notre tableau `adj` alloué dynamiquement comme un tableau à deux dimensions. On fait donc « comme si » en disant que la première ligne de la matrice d'adjacence correspond aux cases numéro 0, 1, ..., $n-1$ du tableau `g->adj`, la deuxième ligne aux cases n, $n+1$, ..., $2n-1$, etc.
   Que signifie, en terme de graphe, l'instruction
   `g->adj[2 * n + 1] = g->adj[1 * n + 2] = 1;`
   ? Ajoutez une arête entre les sommets 0 et 3 et une boucle autour de 2 (exécuter le programme).
5. Comment accéder au nombre d'arêtes entre les sommets v et w dans le graphe ?
6. Le programme ne teste pas si les allocations mémoire retournent `NULL`. Le faire, avec une action appropriée dans ce cas.
7. Exécuter le programme avec `valgrind`. Que manque-t-il ? Dans quel ordre faut-il le faire ? (En cas de doute, tester les différentes possibilités avec `valgrind`).

--- * ---

3 La bibliothèque `graph_mat-1`

Les fichiers `graph_mat-1.h` et `graph_mat-1.c` sont fournis avec l'énoncé. Le fichier `graph_mat-1.c` est à compléter.

Exercice 3 :
Écrire dans le fichier `graph_mat-1.c` les définitions des fonctions `gm_init`, `gm_free`, `gm_n`, `gm_m`, `gm_add_edge` et `gm_mult_edge` en respectant les spécifications données dans le fichier `graph_mat-1.h`.

Penser à compiler régulièrement (syntaxe, warnings, etc) avec la commande

```
$ gcc -Wall -g -c graph_mat-1.c
```

Enfin, utiliser le programme de test `test-1.c`. Pour compiler, produire un exécutable et l'exécuter (par exemple sous valgrind) :

```
$ gcc -Wall -g -c graph_mat-1.c
$ gcc -Wall -g graph_mat-1.o test-1.c -o test-1
$ valgrind ./test-1
```

Un fichier nommé `test-1.dot` a été créé. Visualiser le graphe obtenu avec une commande `dot` (vous devriez obtenir le premier graphe du sujet de TP 1).

--- * ---

Exercice 4 :
Écrire la définition de la fonction `gm_rm_edge`. La tester en utilisant le fichier `test-2.c` avec les mêmes consignes que précédemment.

--- * ---

Exercice 5 :
Écrire la définition de la fonction `gm_degree`. Modifier le fichier `test-1.c` de façon à écrire sur le terminal le degré de chaque sommet et tester.

Quelle est la complexité temporelle de la fonction `gm_degree` ? Comment pourrait-on la réduire ?

--- * ---

Exercice 6 :
Écrire la définition de la fonction `gm_sum`. Tester cette fonction en modifiant à nouveau `test-1.c`.

--- * ---

Exercice 7 :
Écrire la définition de la fonction `gm_prod`. Tester cette fonction en modifiant à nouveau `test-1.c`.

--- * ---

Exercice 8 :
Dans le programme `chemins.c` est créé un tableau de n graphes d'ordre n. Les cases 0 à $n-1$ sont initialisées par des graphes sans arête, sauf la case 1 qui est initialisée par un graphe aléatoire.

Les différents graphes sont affichés sur le terminal sous forme de matrices d'adjacence et un fichier *dot* est enregistré pour chacun (les fichiers sont nommés *chemins-01.dot, chemins-02.dot,* ...).

1. remplacer la ligne d'initialisation de `g[k]`, de telle sorte que dans le graphe `g[k]`, la multiplicité d'une arête (i, j) soit égale au *nombre de chemins de longueur k reliant les sommets i et j* dans `g[1]`. Tester.

2. calculer un nouveau graphe, dans lequel la multiplicité d'une arête (i, j) doit être égale au *nombre de chemins de longueur inférieure ou égale à n reliant les sommets i et j* dans `g[1]`. Tester.
3. à partir du graphe de la question précédente, créer un graphe contenant la clôture transitive de `g[k]`. Tester.
4. faire afficher au programme, pour chaque couple de sommets, leur distance dans `g[1]`
5. faire afficher au programme le diamètre de `g[1]`

`--- * ---`