LSINF1252 - Projet de seconde session

Août 2018

1 Énoncé

Dans le cadre d'un logiciel de manipulation de matrices, vous devez construire une librairie permettant de stocker et de calculer efficacement avec des matrices creuses. Une matrice creuse (sparse matrix) est une matrice dont la majorité des éléments sont nuls. On les rencontre notamment en théorie des graphes, ou encore en analyse numérique, pour la résolution d'équations aux dérivées partielles.

$$\begin{pmatrix} 1 & 8 & 0 & 0 & 0 & 42 \\ 0 & 2 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 \end{pmatrix}$$

FIGURE 1 – Matrice creuse.

Traditionnellement, une matrice est representée en C sous la forme de tableau :

```
double elems[lines][columns];
```

La structure `matrix` visible à la Figure 2 peut ainsi être utilisée pour représenter une matrice, où `nlines` correspond au nombre de lignes et `ncols` au nombre de colonnes. Cette représentation naïve stocke un nombre réel en mémoire pour chaque élément de la matrice. Très facile à implémenter, elle a l'énorme désavantage d'avoir une occupation de la mémoire proportionnelle à la taille de la matrice. Cela peut rapidement devenir problématique lorsqu'il faut traiter de très grandes matrices.

```
struct matrix {
    double **elems;
    unsigned int nlines;
    unsigned int ncols;
};
```

FIGURE 2 – Structure nécessaire à l'implémentation d'une matrice sous forme de tableau

Afin de minimiser l'occupation de la mémoire, il est possible d'exploiter une propriété inhérente aux matrices creuses : le grand nombre d'éléments nuls. Il suffit de stocker en mémoire uniquement les éléments non-nuls. Ainsi, on peut déduire que tout élément qui n'est pas présent en mémoire est un élément nul. L'occupation en mémoire de la matrice ne dépend alors que du nombre d'éléments non-nuls.

Une méthode classique de représentation des matrices creuses est l'utilisation d'une liste de listes (chaînées). La première liste représente les lignes de la matrice. Chaque noeud de la liste est une ligne. Si un noeud n'est pas présent, cela signifie que la ligne correspondante n'est composée que de zéros. Si le noeud est présent, alors au moins un élément de la ligne est non-nul. Chaque noeud de cette liste contient à son tour sa propre liste chaînée. Celle-ci représente le contenu de la ligne correspondante. Chaque noeud de cette sous-liste représente un élément de la ligne, et donc de la matrice. Si un noeud n'est pas présent dans cette sous-liste, cela signifie qu'il vaut zéro. Si un élément est présent, alors il contient une valeur non-nulle.

1.1 Implémentation

Dans ce projet, vous devez implémenter une librairie de gestion de matrices creuses, **dans un premier temps**, grâce à la représentation naïve en tableau, **et, dans un second temps**, grâce à des listes simplement chaînées.

Par convention, une matrice d'une taille $m \times n$ est constituée de m lignes et n colonnes. Chaque élément $a_{i,j}$ de la matrice est situé à la ligne i et à la colonne j. Dans ce projet, on considère que les indices partent de 0. Ainsi, l'élément situé en haut à gauche d'une matrice est l'élément $a_{0,0}$ et l'élément en bas à droite est l'élément $a_{m-1,n-1}$.

Pour réaliser le projet, vous disposez des trois structures nécessaires à l'implémentation de la librairie ainsi qu'une description de l'API. Les trois structures sont visibles à la Figure 3. Le champ `precision` est la valeur absolue en dessous de laquelle un élément de la matrice doit être considéré comme nul (zéro). Une description précise de ces structures est disponible dans le fichier matrix.h qui vous est fourni, notamment en ce qui concerne les pré- et post-conditions. La Figure 4 illustre l'utilisation de ces différentes structures au travers de l'implémentation de la matrice d'exemple de la Figure 1. Notez que comme tous les éléments de la troisième ligne sont nuls, il n'existe pas de `struct line` tel que i = 2.

```
struct sp_matrix {
    struct line *lines;
    double precision;
    unsigned int nlines;
    unsigned int ncols;
};
```

(a) Structure représentant une matrice creuse.

```
struct line {
    struct line *next;
    struct elem *elems;
    unsigned int i;
};
```

(b) Structure représentant une ligne d'une matrice creuse.

```
struct elem {
    struct elem *next;
    unsigned int j;
    double value;
};
```

(c) Structure représentant un élément d'une matrice creuse.

FIGURE 3 – Structures nécessaires à l'implémentation d'une matrice creuse.

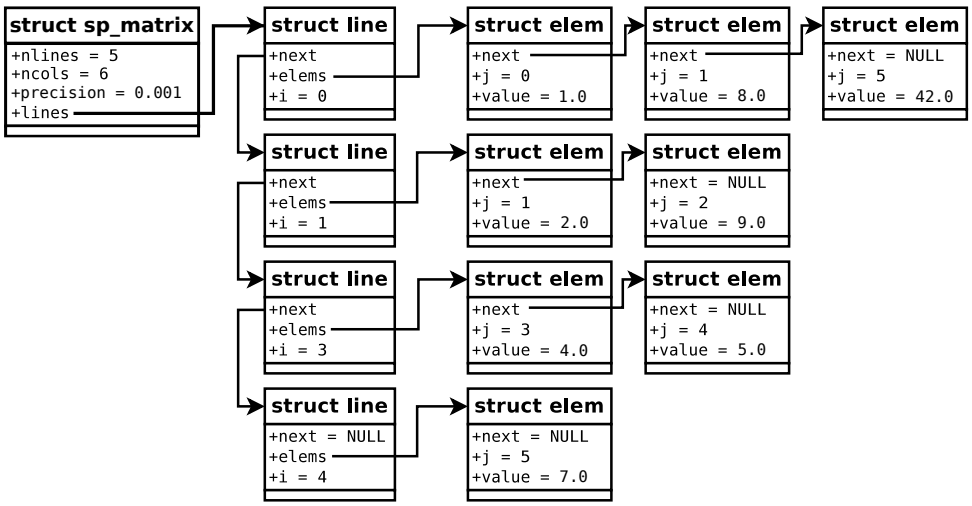

FIGURE 4 – Matrice de la Figure 1 représentée sous forme de listes.

On vous demande d'implémenter plusieurs fonctions permettant de manipuler ces structures. Ces fonctions sont les suivantes.

1.1.1 Initialisation

```
struct matrix *matrix_init(unsigned int nlines, unsigned int ncols);
struct sp_matrix *sp_matrix_init(double precision, unsigned int nlines,
                                 unsigned int ncols);
```

Cette fonction crée une matrice creuse en allouant de la mémoire pour celle-ci et en l'initialisant. Elle retourne un pointeur vers la matrice ou NULL si une erreur s'est produite. Tous les éléments de la nouvelle matrice ont la valeur 0 (zéro).

1.1.2 Destruction

```
void matrix_free(struct matrix *matrix);
void sp_matrix_free(struct sp_matrix *matrix);
```

Cette fonction libère l'entièreté de la mémoire qui a été allouée pour une matrice. Après appel, le contenu de la matrice est indéfini.

1.1.3 Définition de la valeur d'un élément d'une matrice

```
int matrix_set(struct matrix *matrix, unsigned int i, unsigned int j, double val);
int sp_matrix_set(struct sp_matrix *sp_matrix, unsigned int i, unsigned int j,
                  double val);
```

Cette fonction définit la valeur d'un élément d'une matrice. La fonction retourne -1 en cas d'erreur, 0 sinon.

Il s'agit de la fonction la plus **importante** de l'implémentation des matrices creuses. Si elle est correctement réalisée, les fonctions d'addition, de transposée et de conversion de tableau devraient être relativement triviales à implémenter.

Réfléchissez notamment aux cas suivants (liste **non exhaustive**) :

— `val` $\neq 0$ et la ligne `i` ne contient que des zéros

— `val` $\neq 0$ et l'élément (`i,j`) est nul

— `val` $= 0$ et la ligne ne contient que l'élément (`i,j`) comme élément non-nul

1.1.4 Récupération de la valeur d'un élément d'une matrice

```
double matrix_get(const struct matrix *matrix, unsigned int i, unsigned int j);
double sp_matrix_get(const struct sp_matrix *matrix, unsigned int i,
                     unsigned int j);
```

Cette fonction retourne la valeur associée à un élément de la matrice.

1.1.5 Addition de deux matrices

```
struct matrix *matrix_add(const struct matrix *m1, const struct matrix *m2);
struct sp_matrix *sp_matrix_add(const struct sp_matrix *m1,
                                const struct sp_matrix *m2);
```

Cette fonction retourne une **nouvelle** matrice résultant de l'addition des deux matrices données en paramètre. En cas d'erreur, NULL est retourné. Les deux matrices `m1` et `m2` ne sont pas modifiées.

1.1.6 Transposée d'une matrice

```
struct matrix *matrix_transpose(const struct matrix *matrix);
struct sp_matrix *sp_matrix_transpose(const struct sp_matrix *matrix);
```

Cette fonction calcule la transposée de la matrice passée en paramètre. Cette transposée est une **nouvelle** matrice et est retournée par la fonction. En cas d'erreur, NULL est retourné. La matrice `matrix` passée en paramètre n'est pas modifiée.

1.1.7 Multiplication de deux matrices

```
struct matrix *matrix_mult(const struct matrix *m1, const struct matrix *m2);
struct sp_matrix *sp_matrix_mult(const struct sp_matrix *m1,
                                 const struct sp_matrix *m2);
```

Cette fonction calcule la **nouvelle** matrice, résultant du produit des deux matrices données en paramètre. Cette transposée est une **nouvelle** matrice et est retournée par la fonction. En cas d'erreur, NULL est retourné. Les deux matrices `m1` et `m2` ne sont pas modifiées.

1.1.8 Conversion d'une matrice tableau vers une matrice creuse

```
struct sp_matrix *matrix_to_sp_matrix(const struct matrix *matrix,
                                      double precision);
```

Cette fonction crée une matrice creuse dont les éléments correspondent à la matrice tableau `matrix`. Autrement dit, la valeur de l'élément (`i,j`) de la nouvelle matrice est égal à `matrix->elems[i][j]`.

1.1.9 Conversion d'une matrice creuse vers une matrice tableau

```
struct matrix *sp_matrix_to_matrix(const struct sp_matrix *matrix);
```

Cette fonction crée une matrice tableau dont les éléments correspondent à la matrice cruse `matrix`. Autrement dit, la valeur de `elems[i][j]` de la nouvelle matrice est égal à l'élément (`i,j`).

1.1.10 Sauvegarde d'une matrice dans un fichier

```
int matrix_save(const struct matrix *matrix, char* path);
int sp_matrix_save(const struct sp_matrix *matrix, char* path);
```

Cette fonction sauvegarde une matrice `matrix` dans un fichier dont le chemin est donné par l'argument `path`. Si le fichier existe, son contenu est remplacé. Elle retourne 0 en cas de succès, -1 en cas d'erreur.

Afin de mener à bien cette opération, vous devez choisir un format de fichier qui vous semble le plus approprié. Ce choix devra être justifié dans un rapport annexe.

1.1.11 Chargement d'une matrice à partir d'un fichier

```
struct matrix *matrix_load(char* path);
struct sp_matrix *sp_matrix_load(char* path);
```

Cette fonction charge et retourne une matrice depuis un fichier dont le chemin est donné par l'argument `path`. En cas d'erreur, elle retourne -1.

Une matrice respectivement sauvegardée à l'aide des fonctions `matrix_save` et `sp_matrix_save` doit pouvoir être chargée à l'aide des fonctions `matrix_load` et `sp_matrix_load`.

1.2 Invariants

On vous demande de respecter un certain nombre d'invariants qui assurent la cohérence interne des différentes structures. Un invariant est une propriété qui doit rester vraie avant et après chaque appel d'une fonction de l'API. Il se peut bien entendu qu'une fonction viole temporairement un invariant au cours de son exécution, mais celui-ci doit être restauré avant que la fonction ne retourne. Ces invariants sont les suivants.

1. Une `struct line` ne peut jamais avoir son champ `elems` à NULL.
2. Une `struct elem` ne peut jamais avoir sa valeur absolue de son champ `value` inférieur à celle de son champ `precision`.
3. Les noeuds des listes chaînées doivent toujours être en ordre croissant par rapport à leur index (`i` pour les `struct line` et `j` pour les `struct elem`).
4. Il ne peut jamais y avoir plus de noeuds dans la liste des `struct line` qu'indiqué par le champ `nlines` de la `struct sp_matrix`.
5. Il ne peut jamais y avoir plus de lignes dans le tableau `elems` qu'indiqué par le champ `nlines` de la `struct matrix`.
6. Il ne peut jamais y avoir plus de noeuds dans une liste de `struct elem` qu'indiqué par le champ `ncols` de la `struct sp_matrix`.
7. Il ne peut jamais y avoir plus de colonnes dans le tableau `elems` qu'indiqué par le champ `ncols` de la `struct matrix`.
8. Les champs `nlines` et `ncols` des `struct matrix` et `struct sp_matrix` ne peuvent jamais être nuls.

De ces invariants, on peut notamment en déduire que lorsque tous les éléments d'une matrice sont nuls, alors le champ `lines` de la `struct sp_matrix` correspondante vaut NULL.

1.3 Tests

Nous vous demandons d'écrire des tests unitaire permettant de vérifiant le bon fonctionnement de votre implémentation. Ces tests doivent être écrits dans un fichier `test.c`. Ce fichier doit contenir une fonction main, permettant d'exécuter les tests. Vous devez utiliser la librairie `CUnit` [1]. Un chapitre du cours est dédié à cette librairie [2].

1. http ://cunit.sourceforge.net/
2. http ://sites.uclouvain.be/SystInfo/notes/Outils/html/cunit.html

1.4 Makefile et compilation

Votre code **doit** être accompagné d'un Makefile permettant de compiler les tests via la commande make (sans arguments). Le résultat de cette commande doit être la création d'un binaire nommé test. Si cette opération ne fonctionne pas, votre projet ne sera pas corrigé.

Votre code **doit** pouvoir compiler avec les flags de gcc suivants : `-g -Wall -W -Werror -std=gnu99`. C'est **important** car les tests que nous utiliserons pour corriger votre projet utilisent ces flags. Si votre code ne compile pas de cette manière, il ne sera pas corrigé.

La compilation de votre code **doit** fonctionner sur les **machines de la salle Intel**.

1.5 Fuites de mémoire

Votre code ne peut pas avoir de fuite de mémoire. Pour le vérifier, vous devez utiliser l'outil Valgrind [3].

2 Délivrables

Vous devrez remettre votre code sur Moodle dans une archive de type .tar.gz nommée **NOMA.tar.gz**. Bien entendu, vous devez remplacer "NOMA" par votre NOMA. Cette archive doit contenir un répertoire nommé **NOMA**. Ce répertoire doit contenir les fichiers suivants :

- `matrix.h` : fichier donné avec l'énoncé. Il ne peut **pas** avoir été **modifié**.
- `matrix.c` : votre implémentation de l'API. Il ne peut **pas** contenir de fonction **main**. Le code doit être lisible et (raisonnablement) commenté.
- `test.c` : vos tests unitaires, réalisés avec CUnit.
- `report.pdf` : votre rapport décrivant le format des fichiers.
- `Makefile` : Makefile permettant de compiler votre projet avec la commande `make`.

Afin de créer l'archive, vous pouvez utiliser la commande suivante : `tar zcf NOMA.tar.gz NOMA` où `NOMA.tar.gz` est l'archive à créer et `NOMA` est le répertoire qui contient le code de votre projet. Par exemple, si votre NOMA est `99991500`, alors vous devez taper la commande suivante : `tar zcf 99991500.tar.gz 99991500`.

Une tâche INGInious, disponible à l'adresse https ://inginious.info.ucl.ac.be/course/LSINF1252/p3check, vous permet de vérifier le bon format de votre archive ainsi que la compilation correcte de votre code. Si votre projet ne passe pas ces tests de base, il ne sera pas corrigé.

La date de remise du projet est fixée au **15 août 2018**.

3. http ://sites.uclouvain.be/SystInfo/notes/Outils/html/valgrind.html