NGUYEN Van Triet et PHAM Nu Huyen Trang
Université de Technologie de Compiègne
23 avril 2019

Rapport TP3 - NF16

Liste chaînée

A. La liste des structures et des fonctions supplémentaires - Explication du choix

1. Structures:

Dans ce TP, on a été demandé de créer les structures Élément, Matrice_creuse qui contient les valeurs et un pointeur vers l'élément suivant de la liste.

```
struct element {
  int ind;
  int val;
  Element* next;
};

struct _matrice_creuse {
   liste_ligne* ligne;
   int Nlignes;
   int Ncolonnes;
}
```

Dans le structure matrice_creuse, il faut créer un type supplémentaire liste_ligne qui est un pointeur de type Element*, cette variable pointe au premier élément de chaque ligne de matrice creusée. Cela permet de diviser la matrice en Nlignes petites listes chaînées et chaque ligne est une liste chaînée simple.

2. Fonctions supplémentaires:

Les fonctions demandées dans le sujet:

void constructMat(matrice_creuse *m, int t[N][M], int Nlig, int Ncol); //construire une matrice creusée à partir d'un tableau donné.

void lireFichier(matrice_creuse *m, char nomFichier[MAX]); //construire une matrice creusée à partir d'un tableau donné dans un fichier txt.

void afficherMat(matrice_creuse m); //afficher la matrice creusée déjà construite dans le procédure constructMat

void addMat(matrice_creuse m1, matrice_creuse m2); //calculer la somme des 2 matrices creusées

int nombreOctetsGagnes(matrice_creuse m1); //retourner le nombre d'octets gagnés de la représentation de matrice creusée par rapport à celle de matrice normale avec valeur 0

int getValue(matrice_creuse m, int i, int j); //récupérer une valeur dans une position donnée de la matrice creusée m

void putValue(matrice_creuse m, int i, int j); //mettre une valeur dans une position donnée de la matrice creusée m

En outre des fonctions qui sont demandées dans le sujet, nous avons créé aussi les fonctions supplémentaires pour simplifier le problème: ce sont les fonctions: **Element *creeEle(int col, int val)** qui permet de créer un nouveau élément pour la liste chaînée et **void delete(liste_ligne *ligne, int col)** qui permet d'effacer l'élément non valide (valeur 0) après ou pendant le calcul de somme des 2 matrices creusées et **void remplace(liste_ligne* ligne, int col, int val)** pour remplacer les éléments avec les situations peuvent se passer pendant l'addition de 2 matrices creusées; ces fonctions permettent de simplifier la fonction addMat. Sans ces fonctions, la fonction addMat a beaucoup de difficultés, on doit parcourir 2 matrices et de distinguer beaucoup de situations pour faire l'addition cela rend la complexité plus grande. De plus, nous avons ajouté aussi la fonction **void deleteMat(matrice_creuse *m)** afin de libérer la mémoire avant de terminer le programme; les fonctions **int nombreLignes(FILE *fichier)** pour calculer le nombre de lignes d'un fichier et **int nombreColonnes(FILE *fichier)** pour calculer le nombre de colonnes d'un fichier.

B. Complexité

1. Fonction constructMat() :

Vu que dans cette fonction il y a un boucle for avec un autre boucle for dedans, la complexité est simplement la multiplication du nombre d'itérations de ces deux boucles. On trouve donc la complexité égale O(Nlig*Ncol)

2. Fonction lireFichier():

Dans cette fonction, il y a 3 boucles (2 boucles for et boucle do..while avec la condition c != ';' && c != '\n'), alors, la complexité est la produit des fonctions f(n), g(n) et h(n) représentant ces boucles car dans les instructions dans les boucles et aussi dans la fonction sont toujours des instructions simples. Pour la boucle do..while, notre objectif est de récupérer seulement un par un élément de la matrice, on a pensé de la situation si cet élément a plus de 2 chiffres (par exemple 11,999, etc) donc le boucle a N itérations avec N>0, N est le nombre de chiffres. On a donc la complexité est O(lig*col*N) car le boucle for 1 répète lig fois et col fois pour la boucle for 2.

3. Fonction afficherMat():

Dans cette fonction, on a le boucle while avec un autre boucle while dedans. Pour le premier boucle, on est sure qu'il fait Nlignes itérations vu la condition de ce boucle; de plus, pour le deuxième, on a seulement la condition l_aux !=NULL, donc ce boucle peut répéter m fois avec 0<m<=Ncolonnes pour chaque ligne. Au meilleur cas, la complexité est O(Nlignes*max(m)) (avec max(m) est le nombre d'itération maximum entre les lignes de la matrice creusée car ce nombre n'est pas le même pour chaque ligne et max(m)<Ncolonnes - On suppose dans ce cas-là il y a les éléments 0 dans la matrice originale). Au pire cas, la complexité est O(Nlignes*Ncolonnes) - On suppose dans ce cas-là il n'y a pas les éléments 0 dans la matrice originale

4. Fonction getValue():

Il y a un boucle while dont le nombre d'itérations est N avec N = min(nombre d'éléments dans la ligne i, nombre d'éléments dont l'indice est inférieur à j); 0<N<=Ncolonnes dans cette fonction. C'est pourquoi la complexité est O(N) au meilleur cas et O(Ncolonnes) au pire cas (dans la boucle il y a seulement des instructions simples)

5. Fonction putValue():

Dans cette fonction, on a seulement les instructions simples et aucun boucle mais on a de rappel des fonctions **creeEle()**, **delete()** et **remplace()**. Pour la fonction **creeEle()**, la complexité est O(1) car il n'y a que des instructions simples. Pour la fonction **delete()**, la complexité est O(1) pour le meilleur cas (le cas où col soit le premier élément de la ligne i) et O(N) pour le pire cas avec N est le minimum entre le nombre d'éléments de la ligne i et le

nombre d'éléments dont l'indice est plus petit que col; donc la complexité de cette fonction est O(N). Pour la fonction **remplace()**, la complexité est le maximum de complexité des instructions if-else. Tous les autres instructions sont simples (même si avec le rappel de fonction creeEle()) sauf le boucle while avec la condition est p!=NULL && k<col donc la complexité de la fonction **remplace()** est O(M) avec M est le minimum entre le nombre d'éléments de la ligne i et le nombre d'éléments dont l'indice est plus petit que col. Bref, la complexité de la fonction **putValue()** est O(max(N,M)) mais on a i,j sont même comme paramètres de deux fonctions **delete()** et **remplace()** d'où M=N alors la complexité est O(N).

6. Fonction addMat():

La complexité de cette fonction est le maximum entre les complexités de la condition if-else.

Dans le cas de else, la complexité est O(1)

Dans le cas de if, on trouve qu'il y a deux boucle for et l'appel des fonctions **putValue()** et **getValue()** qui ont même complexité O(N) donc la complexité de cette fonction est O(Nlignes*Ncolonnes*N)

7. Fonction nombreOctetsGagnes():

Même explication et complexité de la fonction **afficherMat()**

8. Fonction deleteMat():

Même complexité de la fonction **addMat()**

Note:

Dans le programme, nous avons corrigé nos fautes qui ont été indiqués par notre chargé TP. Avant, nous n'avons pas précisé les situations dans la fonction getValue() et putValue(); nous avons aussi le problème de ne pas lire la matrice à partir d'un fichier contenant plusieurs lignes. De plus, nous ne savons pas que nous devons libérer les mémoires avant de fermer le programme (choix 6) et aussi que nous calculons incorrectement le nombre d'octets gagnés.

Nous avons mis la fonction delete(), remplace(), nombreLignes(), nombreColonnes() et creeEle() dans notre nouveau programme.