NF16 – TP05

Compte-rendu

Arbres et Calcul formel

Auteurs : Akim Sadaoui et Antoine Hars

- **Introduction :**

Le but de ce tp est d'effectuer la représentation et la transformation d'expressions mathématiques en se limitant à l'addition et à la multiplication, contenant des constantes et des variables formelles.
Dans ce tp, nous utilisons la notation post-ordre, aussi appelée notation polonaise inversée, pour les expressions mathématiques.

- **Les fonctions :**

La fonction de saisie de l'expression de la forme :

- *Node * saisie_expression()*

Cette fonction de saisie de l'expression effectue une lecture de l'entrée stdin, correspondant au clavier, jusqu'à ce que le caractère entré soit différent d'un retour à la ligne pour créer un arbre représentant l'expression entrée.
Pour récupérer les éléments de l'expression lue au clavier, nous faisons appel à la fonction fgets qui nous permet de lire une chaîne de caractères entrée au clavier, puis avec la fonction strtok, nous récupérons un par un, les éléments de l'expression à mettre dans l'arbre (réel, variable alphabétique ou opérateur + ou *).
Nous créons dans cette fonction un arbre en se basant sur une pile qui y stocke les nœuds à insérer. Une fois la construction de l'arbre fini, nous vérifions que la pile ne contient bien qu'un seul élément, le premier nœud de l'arbre, auquel cas l'expression est valide.

Au niveau de la complexité de cet algorithme, la fonction ne dépend que de la taille de l'expression lue au clavier car toutes les fonctions de manipulation de la pile et d'un nœud de l'arbre n'ont pas une complexité supérieure O(1) donc la complexité est en O(n).

Les fonctions d'affichage en post-ordre, pre-ordre et in-ordre :

- *void pre_ordre(Node * node)*
- *void in_ordre(Node * node)*
- *void post_ordre(Node * node)*

Les fonctions d'affichage selon l'ordre demandé sont des fonctions récursives qui vont afficher les différents termes de de l'arbre représentant l'expression. L'affichage et brute et ne prend pas en compte la priorité des opérateurs.

Ces 3 fonctions d'affichage parcourent récursivement tout les éléments de l'arbre donc elle dépendent de sa taille et leur complexité est en O(n).

La fonction d'affichage de l'expression :

- *void affiche_expression(Node * node)*

Cette fonction d'affichage de l'expression nous permet de voir les priorités des opérations avec l'ajout dans l'affichage des parenthèses pour spécifier l'ordre des calculs mathématiques à faire.
Dans cette fonction, nous vérifions dans un premier lieu si nous sommes dans un sous-calcul, si c'est le cas et que le calcul à représenter est une addition, nous encadrons le morceau de l'expression par des parenthèses.

Cette fonction parcourt tous les éléments de l'arbre pour les afficher donc sa complexité dépend juste de la taille de l'arbre de l'expression. La complexité est en O(n).

La fonction de copie de l'expression :

- *Node * clone(Node * node)*

Cette fonction de clonage de l'expression va parcourir tous les éléments de l'arbre récursivement pour les copier et créer en même temps un deuxième arbre à partir de la copie.
Nous faisons appel à la fonction de clonage sur les fils gauche et droit du nœud puis nous copions finalement ce nœud une fois que tous les nœuds fils sont copiés.

L'appel récursif de la fonction pour les fils des nœuds de l'arbre nous montre que cette fonction dépend uniquement de la taille de l'arbre car nous faisons un simple parcours de l'arbre donc la complexité de la fonction est en O(n).

La fonction de calcul intermédiaire :

- *void calcul_intermediaire(Node * node)*

Dans cette fonction de calcul intermédiaire, le but rechercher est de simplifier l'expression et d'effectuer les calculs au feuilles de l'arbre.
Nous vérifions d'abord si le nœud étudié a pour fils des variables et/ou des nombres. Si c'est le cas, nous réalisons le calcul ou le simplifions et nous supprimons les fils gauche et droit du nœud avec la fonction destroy_all qui libère la mémoire du nœud avec tous ses fils (dans notre cas, nous n'utilisons destroy_all pour une feuille donc destroy_all a pour complexité O(1) ici). S'il y a une opération imbriquée, nous effectuons le calcul dans les 2 fils du nœud.

Dans cette fonction la complexité est dans le meilleur des cas en Ω(1) et dans le pire des cas, équivaut à la taille de l'arbre, soit en O(n).

La fonction de calcul de l'expression entière :

- *void calcul(Node * node)*

Cette fonction calcule l'expression entière ou la simplifie au maximum si elle contient des variables. Elle effectue dans un premier temps les calculs prioritaires, donc ceux tout en bas de l'arbre, nous parcourons donc tout l'arbre pour commencer puis nous effectuons tous les calculs mathématiques et les simplifications possibles en commençant par les opérations prioritaires pour obtenir à la fin une expression simplifiée au maximum.
À chaque calcul ou simplification effectué, nous supprimons le nœud obsolète au moyen de la fonction destroy_all, qui va détruire dans notre cas une feuille de l'arbre, ce qui nous permet de n'avoir à la fin que le strict minimum de nœud pour représenter l'expression.

Dans cette fonction, vu que nous parcourons tous les éléments de l'arbre pour faire les calculs et les simplifications possibles, la complexité dépend avant tout de la taille de l'arbre car dans la fonction l'appel de la destruction d'un nœud est en O(1) dans tous les cas (nous supprimons à chaque fois une feuille de l'arbre et non un nœud). La complexité de l'algorithme est donc en O(n) dans le pire des cas et en Ω(1) dans le meilleur des cas ou il n'y a qu'un seul élément dans l'arbre.

La fonction de développement de l'expression :

- *void developpement(Node * node)*

Dans cette fonction, le but recherché est de développer tous les additions imbriquées dans une multiplication. Pour cela, nous commençons par les feuilles de l'arbre puis nous remontons tout l'arbre en essayant de développer chaque nœud. Lorsque l'expression est développable (une somme imbriquée dans une multiplication), nous créons et insérons les nœuds qui vont contenir les éléments résultants du développement (nous faisons appel à la fonction clone et à create_node pour ajouter des nœuds à l'arbre).

La complexité de l'arbre dans le meilleur des cas est en Ω(1) pour un arbre d'un élément et dans le pire des cas, en O(n) car nous parcourons tout l'arbre pour pouvoir développer chaque opération.

La fonction de dérivation de l'expression :

- *void derivation(Node * node, char v)*

Dans la fonction de dérivation, nous avons 3 cas :

- Si le nœud est une constante ou une variable, le nœud prend la valeur 0 ou 1 (pour la variable).
- Dans le cas d'une addition, la dérivé correspond à la somme des dérivation, donc nous appelons la fonction derivation sur les fils gauche et droit.
- Dans le cas d'une multiplication, nous créons un clone pour le fils gauche et pour le fils droit et appelons la fonction derivation sur ces 2 clones et nous organisons les résultats obtenus selon la formule de la dérivation d'une multiplication : (fg)' = f' * g + f * g'.

Dans le meilleur des cas où l'arbre ne contient qu'un nœud (une variable ou un

constante), la complexité de la fonction est en Ω(1) mais dans le cas d'un arbre composé de plusieurs nœuds, la complexité est en O(n) car nous effectuons un parcours de l'arbre en entier pour pouvoir dériver tous les termes de l'expression.

- **Conclusion :**

La réalisation de ce tp nous a permis d'approfondir nos connaissances dans la manipulation de structures imbriquées (les éléments de la pile contenant l'arbre) avec la création de l'arbre.
Nous avons pu coder la réalisation d'une insertion grâce notamment à la fonction developpement, utiliser la récursivité dans les différents parcours de la l'arbre et apprendre la procédure pour détruire des nœuds d'un arbre.

Au cours de ce tp, nous avons pu observer les problèmes que cause parfois la fonction scanf qui utilisée pour une variable et ensuite pour un caractère entraîne une absence de la lecture du deuxième caractère du fait qu'il récupère le caractère '\n' engendré par l'appel du scanf précédent et donc de privilégier l'utilisation de la fonction getchar dans ce genre de cas.