Sorbonne Université - Master STL

MU4IN511 - Ouverture

Année 2023-2024

Projet : Génération des ZDD

Alex XU & Jean-Marc ZHUANG

SOMMAIRE

1. Echauffement.....2
2. Arbre de décision.....3
3. Compression de l'arbre de décision et ZDD.....4
4. Compression avec historique stocké dans une structure arborescente.....4
5. Analyse de complexité.....4
6. Etude expérimentale.....4

1. Echauffement

Question 1.1

Ici, nous avons créer une structure de donnée nommée *ListInt64* qui est une référence vers *'a list*, dans cette structure nous avons plusieurs primitives:

create() qui créer une réference vers une liste vide
insertHead(x, list) qui ajoute en tête de liste
insertTail(x, list) qui ajoute en queue
removeHead(list) qui supprime la tête et la renvoie
length(list) qui renvoie la taille de la liste en utilisant *List.length*

Question 1.2

La fonction *decomposition* prend en entrée un grand entier *l* et le décompose en une liste de bits, dont le bit de poids faible se trouve à gauche. Pour ce faire, elle utilise deux fonctions auxiliaires : *loop* et *loop2*.

Dans le sujet, *0L* représente 2^{64}, on a donc implémenté la fonction *bit64* qui renvoie une liste contenant uniquement des *false*, puisque 2^{64} s'écrit avec 64 bits à 0 et le bit le plus fort à 1.

La fonction *loop* prend en argument un grand entier *x* et une liste *list*. Elle décompose *x* en bits en utilisant une division par 2 et un modulo qui sont tous deux des opérations unsigned, afin de traiter les entrées comme des entiers non signés.

Si x est égal à 0, elle renvoie la liste.

Sinon, elle vérifie si le reste de la division de *x* par 2 est 1 (ce qui signifie que son bit de poids faible est 1). Si c'est le cas, elle ajoute *true* à la liste, sinon elle ajoute *false*.

Ensuite, elle appelle récursivement *loop* avec *x* divisé par 2 et la nouvelle liste.

La fonction *loop2* prend en argument le grand entier *l* et l'accumulateur *list*. Elle parcourt la liste *l* et pour chaque élément, si l'élément est égal à 0, elle ajoute à la liste une liste de 64 bits. Sinon, elle appelle loop avec l'élément et la liste.

Enfin, *decomposition* appelle *loop2* avec le grand entier *l* et l'accumulateur vide.

Question 1.3

La fonction *completion* permet de tronquer la liste décomposée si *n* est inférieur à la taille de la liste, sinon elle "complète" la liste en ajoutant des false aux bits les plus forts pour changer la taille de la représentation sans changer sa valeur. Pour cela nous avons 2 primitives

- *ajoutFalse(list, n, accu)* qui ajoute *n* false dans *accu*, et dès que n est nul on concatène *list et accu*
- *suppElem(list, n, accu)* qui tronque list pour que la taille de la liste soit égale à n. Donc à la sortie de la récursivité, on renvoie un couple de *(accu, list)* où accu est la liste tronquée, et list le reste sans la liste tronquée car nous en avons encore besoin dans la fonction package (Question 1.4).

Ainsi pour la fonction *completion* si la taille de la liste est supérieur à n, on fait appel à *suppElem* sinon on fait appel à *ajoutFalse*

Question 1.4
La fonction *composition* passe de la représentation binaire à sa représentation en base 10, c'est-à-dire un grand entier. Chaque élément du grand entier est sur 64 bits, donc nous avons créé une fonction *package(list)* qui va créer plusieurs paquets de 64 bits du grand entier si la taille de la liste est supérieur à 64. Ainsi, package renvoie une matrice de boolean, la matrice représente le grand entier, et les sous tableaux représentent les éléments.
Visuellement [0L, 1L] est la matrice, 0L et 1L sont les éléments.
LoopExt parcours les sous tableaux et *LoopInt* créer les éléments du grand entier. Pour créer l'élément, on fait un décalage à gauche de l'accumulateur puis on fait un OR logique entre l'accumulateur et le boolean se trouvant dans la matrice.

Question 1.5
La fonction *table(x, n)* crée la table de vérité de l'entier *x*, on applique donc *decomposition* sur *x* pour finir on applique *completion* sur la représentation binaire de x.

Question 1.6
La fonction *gen_alea*, prend en entrée une valeur *n* et génère un grand entier aléatoire de n bits au maximum.
Dans cette fonction nous implémentons une fonction auxiliaire loop qui prend en argument le grand entier aléatoire de n bits à renvoyer *acc* ainsi que le nombre de bits restant *n*.
On construit notre acc en insérant en tête tous les entiers aléatoire de 64 bits, tant que le nombre de bit restant est supérieur ou égale à 64, puis le dernier entier aléatoire de $n-\ell\times64$.
Enfin, lorsqu'il ne reste aucun bit, on renvoie List.rev du grand entier de n bits, car on souhaite que l'entier de $n-\ell\times64$ soit en queue.

2. Arbre de décision

Question 2.7
```
type btree =
 | Leaf of bool //true ou false
 | Node of int * btree * btree //profondeur * enfant_gauche * enfant_droit
```

Question 2.8
La fonction *cons_arbre*, prend en entrée une table de vérité *table* et construit l'arbre de décision associé à la table de vérité.
On utilisera une fonction auxiliaire *construction* afin de construire notre arbre à partir d'une profondeur 0 qu'on incrémente à chaque appel, il prend en deuxième paramètre une table de vérité.
Si la table est vide on renverra une exception *"Table de vérité vide"*.
S'il n'y a qu'un élément, on renvoie la feuille contenant le booléen.
Sinon on construit notre arbre en appelant récursivement *construction,* en incrémentant la profondeur pour l'indice et divisant la table de vérité en deux à l'aide de la fonction *diviser_liste* qui prend en argument la table de vérité à couper en deux.

Notre fonction *diviser_liste,* possède aussi une fonction auxiliaire, qui permet de décrémenter la taille de la première liste au fur et à mesure que l'on construit celle-ci. On retourne un couple de liste, dont la première de taille n/2 et la deuxième est la table restante.

Question 2.9
La fonction *liste_feuilles*, prend en entrée un nœud *arbre* et construit la liste des étiquettes des feuilles du sous-arbre enraciné en N. On construit notre liste en parcourant notre arbre de manière préfixe, ça permettra d'avoir une liste des feuilles gauches à droites.

3. Compression de l'arbre de décision et ZDD

Question 3.10

```
type dejaVus = int64 list * btree ref;;
type listeDejaVus = dejaVus list;;
```

Question 3.11
Pour créer la fonction compressionParListe, nous avons au préalable créé 3 primitives:

- *findSeen* qui va rechercher si l'élément n (composition d'une liste de boolean) à été vu, si en parcourant toute la liste, la fonction ne trouve pas n alors on renvoie None, sinon on renvoie le pointeur vers le noeud
- *onlyFalse renvoie vrai si le tableau contient uniquement des false sinon renvoie faux*
- *regleM* prend un arbre, le tableau des int64 vu et la représentation binaire de l'entier comme argument. Tout d'abord on applique composition à la représentation binaire pour renvoyer un int64 qui sera maintenant appelé n, on va ensuite appliquer *findSeen sur n. Si la fonction renvoie node* alors tree sera remplacé par node sinon on touche pas à l'arbre et on ajoute un couple *(n, tree),* tree est le noeud où on se trouve actuellement.

compressionParListe à un arbre *(tree)* et une liste de déjà vu *(seen)* comme argument, si *tree* est une feuille alors on applique la règle M, sinon si c'est un noeud, on va appliquer *onlyFalse* au fils droit de *tree*.
Si le fils droit contient que des false alors on fait une récursion que sur le fils gauche et ensuite, *tree* est remplacé par le fils gauche de *tree (règle Z)*
Sinon on fait un parcours suffixe et appliquer la règle M

4. Compression avec historique stocké dans une structure arborescente
5. Analyse de complexité
6. Etude expérimentale