

Les ROBDD (Reduced Ordered Binary Decision Diagram) sont utilisés pour représenter de façon compacte une ou plusieurs fonctions Booléennes, partageant le même support (c'est à dire dépendant d'un même ensemble de variables Booléennes). Pour un ordre donné des variables constituant le support, cette représentation est canonique : A chaque fonction Booléenne est associé un unique graphe orienté acyclique (DAG). Le graphe étant acyclique, chaque noeud BDD définit un sous-graphe dont il est la racine. Par conséquent, chaque noeud BDD correspond à une fonction Booléenne particulière. On utilise le fait que les variables Booléennes constituant le support sont ordonnées pour identifier ces variables par leur index.

La structure C permettant de représenter un noeud du graphe ROBDD est définie de la façon suivante :

```
typedef struct bdd_t {  
  
    unsigned int INDEX;  
  
    struct bdd_t *HIGH;  
  
    struct bdd_t *LOW;  
  
} bdd_t
```

Pour la gestion des variables, vous utiliserez les mêmes fonctions que celles vues lors du TME5. Vous trouverez ces fonctions dans les fichiers var.h et var.c.

```
typedef struct var_t{  
  
    char *NAME; unsigned INDEX; unsigned VALUE;  
  
} var_t;
```

- extern var_t *cons_var (char *name, unsigned index, unsigned value);
 - ◆ extern var_t *get_var_index (unsigned index);
- extern var_t *get_var_name (char *name);

Pour la gestion de la structure bdd, vous trouverez la déclaration des types et des fonctions dans les fichiers bdd.h et bdd.c. En particulier :

- extern bdd_t *create_bdd(unsigned int index, bdd_t *high, bdd_t *low)

Cette fonction gère un dictionnaire de tous les noeuds BDD déjà créés et fournit la garantie qu'il n'existera jamais deux noeuds BDD possédant les mêmes valeurs pour les champs INDEX, HIGH et LOW. La fonction create_bdd() recherche le noeud BDD possédant les valeurs définies par les arguments, et renvoie un pointeur sur le noeud BDD correspondant. Si ce noeud n'existe pas, cette fonction crée un nouveau noeud dans le dictionnaire, en affectant la valeur index au champs INDEX, la valeur high au champs HIGH, et la valeur low au champs LOW.

- extern bdd_t *apply_bdd(unsigned oper, bdd_t *p1, bdd_t *p2)

Cette fonction prend pour arguments un entier définissant un opérateur Booléen (les valeurs possibles sont OR, AND, et XOR), deux pointeurs p1 et p2 sur deux noeuds BDD représentant deux fonctions Booléennes F1 et F2 (notons que les deux fonctions F1 et F2 ne dépendent pas forcément de toutes les variables Booléennes définies dans le support global). La fonction apply_bdd() construit le graphe ROBDD représentant la fonction $F = F1 \text{ oper } F2$, et renvoie un pointeur sur le noeud ROBDD racine de ce graphe.

- extern bdd_t *not_bdd(bdd_t *p)

La fonction `not_bdd()` prend pour unique argument un pointeur `p` sur un noeud BDD représentant une fonction Booléenne `F`, et renvoie un pointeur sur le noeud BDD représentant la fonction Booléenne `NOT(F)`.

- `extern void display_bdd(bdd_t *p)`

Cette fonction prend pour unique argument un pointeur `p` sur un noeud BDD représentant une fonction Booléenne `F`, et affiche sur le terminal standard l'ensemble des noeuds BDD qui interviennent dans le graphe ROBDD représentant `F`. Il y a un noeud BDD par ligne, et chaque noeud BDD est décrit par quatre entiers :

- `N` représente un numéro identifiant le noeud BDD
- `H` représente le numéro du noeud BDD représentant le cofacteur `HIGH`.
- `L` représente le numéro du noeud BDD représentant le cofacteur `LOW`.
- `X` représente l'INDEX de la variable associée au noeud BDD

A/ Représentation Graphique

Soit la fonction Booléenne $F(a,b,c,d,e)$, définie par l'expression suivante

$E0 = (a ? . (b + c + d ?) . e) + c$ La notation $a ?$ signifie $NOT(a)$.

$E0$ peut se re-écrire sous forme préfixée de la façon suivante :

$E0 = OR(AND(NOT(a) OR(b c NOT(d) e) c)$

A.1) En utilisant la décomposition de Shannon, représentez graphiquement le graphe ROBDD associé à la fonction $F(a,b,c,d,e)$ pour l'ordre $a > b > c > d > e$, puis pour l'ordre $e > d > c > b > a$.

A.2) Précisez, pour chaque noeud du ROBDD ainsi construit, quelle est la fonction Booléenne représentée par ce noeud.

B/ Fonctions `notbdd()` et `applybdd()`

Les fonctions `apply_bdd()` et `not_bdd()` ont été présentées en cours. Vous trouverez le code de ces fonctions dans le fichier `bdd.c`.

B.1) Démontrer la relation de récurrence entre la fonction Booléenne $NOT(F)$ et les fonctions Booléennes $NOT(FH)$ et $NOT(FL)$, où les fonctions Booléennes FH et FL sont les cofacteurs de Shannon de la fonction F , décomposée suivant la variable x :

- $F = x . FH + x ? . FL$

B.2) Soit la fonction F , définie par l'expression $E1 = a . (b + c)$ Représentez graphiquement le ROBDD représentant la fonction F , pour l'ordre des variables $a > b > c$. On appelle `p0`, `p1`, `p2`, `p3`, `p4` les pointeurs sur les 5 noeuds BDD contenus dans ce graphe :

- `p0` représente la fonction $F0 = 0$
- `p1` représente la fonction $F1 = 1$
- `p2` représente la fonction $F2 = c$
- `p3` représente la fonction $F3 = b + c$
- `p4` représente la fonction $F4 = a . (b + c)$

Représentez graphiquement l'arbre d'appels des fonctions lorsqu'on appelle la fonction `not_bdd()` avec pour argument le pointeur `p4`. Quel est le nombre maximum d'appels de fonctions empilés sur la pile d'exécution? Combien de noeuds BDD vont être créés par la fonction `create_bdd()`, si on suppose que la structure de données ne

contient initialement que les 5 noeuds pointés par p0, p1, p2, p3, et p4 au moment de l'appel de la fonction not_bdd(p4).

B.3) Soient F1 et F2 deux fonctions Booléennes, et les fonctions F1H, F1L, F2H, F2L définies par la décomposition de Shannon suivant la variable x :

- $F1 = x \cdot F1H + x? \cdot F1L$
- $F2 = x \cdot F2H + x? \cdot F2L$

La relation de recurrence

- $(F1 \text{ op } F2) = x \cdot (F1H \text{ op } F2H) + x? \cdot (F1L \text{ op } F2L)$

a été démontrée en cours dans le cas des opérateurs OR et AND. Démontrez que cette relation est vraie dans le cas d'un opérateur XOR. Analysez le cas général, ainsi que les quatre cas particuliers où l'une des deux fonctions F1 ou F2 est égale à une des deux fonctions constantes 0 ou 1.

C/ Fonction abl2bdd()

On cherche à écrire la fonction abl2bdd(), qui construit automatiquement le graphe ROBDD représentant une fonction Booléenne F, à partir d'un arbre ABL représentant une expression Booléenne particulière de cette fonction F.

- bdd_t *abl2bdd(bip_t *p)

Cette fonction prend comme unique argument un pointeur sur la racine d'un arbre ABL, et renvoie un pointeur sur le noeud BDD représentant la fonction. Puisque les arbres ABL ne contiennent que des opérateurs OR, AND, XOR et NOT, et qu'on dispose des fonctions apply_bdd() et not_bdd(), il est possible de construire le graphe ROBDD par application récursive de ces deux fonctions.

C1) Décrire en français, l'algorithme récursif de cette fonction abl2bdd() dans le cas particulier où tous les opérandes AND, OR ou XOR présents dans l'arbre ABL n'ont que deux opérandes,

C.2) Ecrire en langage C la fonction abl2bdd(). Pour valider cette fonction, on écrira un programme main(), qui effectue successivement les opérations suivantes :

- déclaration et indexation de toutes les variables Booléennes (var.h).
- construction explicite de l'arbre ABL représentant l'expression E1, avec la fonction parse_abl() (parse_abl.h),
- affichage de cette expression Booléenne, avec la fonction display_abl(), pour vérifier que l'arbre ABL est correctement construit (bip.h),
- construction du graphe ROBDD représentant la fonction F, avec la fonction abl2bdd(),
- affichage du graphe ROBDD ainsi construit, en utilisant la fonction display_bdd().

C.3) Comment faut-il modifier la fonction abl2bdd(), pour traiter le cas général, où les opérateurs OR, AND et XOR peuvent avoir une arité quelconque? Modifier le code de la fonction, et validez ces modifications sur l'exemple de l'expression Booléenne E0, ou sur des expressions encore plus complexes.

D/ Fonction satisfybdd()

Soit une fonction Booléenne quelconque $F(x_1, x_2, x_3, \dots, x_n)$, dépendant de n variables. Soit la fonction Booléenne G, définie comme un produit (opérateur AND) d'un nombre quelconque de variables x_i (directes ou complémentées). On dit que G "satisfait" F si $G \Rightarrow F$. (Autrement dit, si $G = 1$, alors $F = 1$). Remarquez que la condition $G = 1$ impose la valeur de toutes les variables appartenant au support de G (les variables directes doivent

prendre la valeur 1, et les variables complémentées doivent prendre la valeur 0). Pour une fonction F donnée, la fonction G n'est évidemment pas unique...

Exemple : $F = (a \cdot b) + c$

On peut avoir $G = c$ ou $G = a \cdot b$ ou encore $G = a \cdot b \cdot c$?

Dans la suite de cet exercice, on cherche à écrire une fonction C qui construit automatiquement le ROBDD représentant une fonction G satisfaisant une fonction F quelconque. Comme il existe plusieurs solutions, on choisira systématiquement la solution qui minimise le nombre de variable xi dans le produit.

- `bdd_t *satisfy_bdd(bdd_t *p)`

La fonction `satisfy_bdd()` prend pour argument un pointeur sur le noeud BDD représentant la fonction F et renvoie un pointeur sur le noeud BDD représentant la fonction G.

D.1) La décomposition de Shannon de la fonction F suivant la variable x d'index le plus élevé définit les cofacteurs FH et FL :

- $F = x \cdot FH + \bar{x} \cdot FL$

Pour alléger les notations, on note `sat(F)` la fonction Booléenne construite par la fonction `satisfy_bdd()` pour la fonction F. Donner la relation de récurrence entre les fonctions `sat(F)`, `sat(FH)`, et `sat(FL)`. On étudiera successivement le cas général où ni FH et FL ne sont constantes, et les quatre cas particuliers où l'une des deux fonctions FH ou FL sont égales à 0 ou 1.

D.2) Ecrire, en langage C, le code de la fonction `satisfy_bdd()`, et appliquez-la sur la fonction Booléenne F définie dans la partie A de l'énoncé, en modifiant le programme `main` de la fonction précédente.

E/ Fonction `bdd2abl()`

On souhaite pour finir écrire la fonction `bdd2abl()`, qui construit automatiquement l'arbre ABL représentant une expression Booléenne multi-niveaux, pour une fonction Booléenne représentée par un ROBDD.

- `bip_t *bdd2abl(bdd_t *p)`

Cette fonction prend comme unique argument un pointeur sur un noeud BDD, et renvoie un pointeur sur le bipointeur représentant la racine de l'arbre ABL représentant l'expression Booléenne.

E.1) Il existe évidemment un grand nombre d'expressions Booléennes équivalentes pour une même fonction Booléenne. Dans un premier temps, nous nous contenterons d'utiliser des opérateurs OR, et AND à deux opérandes, ainsi que l'opérateur NOT. Soit un noeud BDD représentant une fonction Booléenne F. Soient x la variable associée à ce noeud BDD, FH et FL les fonctions Booléennes associées aux noeuds BDD pointés par les pointeurs HIGH et LOW (cofacteurs de Shannon).

- Quelle expression Booléenne dépendant de x, FH et FL peut-on associer à la fonction F dans le cas général ou FH et FL sont quelconques ? (aucune des deux fonctions FH ou FL n'est égale à 0 ou à 1)
- Comment cette expression Booléenne se simplifie-t-elle lorsque l'une des deux fonctions FH ou FL est égale à 0 ou à 1. Etudiez un par un les différents cas.

E.2) En utilisant les résultats de la question E.1, proposez un algorithme de construction de l'arbre ABL

E.3) Ecrire en langage C la fonction `bdd2abl()`, et validez-la sur différents exemples, en utilisant la fonction `display_abl()`.