TME 6 : Représentation des fonctions Boolèennes : ROBDD

Objectif

L'objectif principal de ce TME est de vous familiariser avec la représentation des fonctions Booléennes sous forme de ROBDD. 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.

Créez un répertoire `tme6`, et recopiez dans ce répertoire les fichiers qui se trouvent dans le répertoire `/users/enseig/jpc/M1-CAO/TME/4.public`

A) Structures de données et fonctions de base

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

```
class Bdd {
  public:
    class Key {
      private:
                unsigned int _index;
                unsigned int _highIdent;
                unsigned int _lowIdent;
      public:
        inline              Key       ( unsigned int index, unsigned int high, unsigned int low
        friend bool         operator< ( const Key& lhs, const Key& rhs );
    };
  private:
    static unsigned int       _maxIdent;
    static unsigned int       _maxStamp;
    static std::map<Key,Bdd*> _bdds;
  private:
            unsigned int       _ident;    // Unicity identifier.
            unsigned int       _index;    // Decomposition variable index.
            Bdd*               _high;     // Low cofactor pointer.
            Bdd*               _low;      // High cofactor pointer.
            unsigned           _stamp;    // Used to flag already reached nodes in the DAG
```

```
                                        // recursive walktrhough.
  public:
    static Bdd*               ConstHigh;
    static Bdd*               ConstLow;
  private:
                              Bdd          ( unsigned index, Bdd* high, Bdd* low );
                             ~Bdd          ();
  public :
    static Bdd*               create       ( unsigned index, Bdd* high, Bdd* low );
    static Bdd*               apply        ( OperatorType, Bdd*, Bdd* );
    static Bdd*               Not          ( Bdd* );
    static Bdd*               And          ( Bdd*, Bdd* );
    static Bdd*               And          ( Bdd*, Bdd*, Bdd* );
    static Bdd*               And          ( Bdd*, Bdd*, Bdd*, Bdd* );
    static Bdd*               Xor          ( Bdd*, Bdd* );
    static Bdd*               Xor          ( Bdd*, Bdd*, Bdd* );
    static Bdd*               Xor          ( Bdd*, Bdd*, Bdd*, Bdd* );
    static Bdd*               Or           ( Bdd*, Bdd* );
    static Bdd*               Or           ( Bdd*, Bdd*, Bdd* );
    static Bdd*               Or           ( Bdd*, Bdd*, Bdd*, Bdd* );
  public:
    inline unsigned           getIdent     ();
    inline unsigned           getIndex     ();
    inline Bdd*               getHigh      ();
    inline Bdd*               getLow       ();
           Bdd*               neg          ();
           unsigned int       depth        ();
           Bdd*               satisfy      ();
           void               display      ( std::ostream& );
           ValueType          eval         ();
           Ebm*               toEbm        ();
    static Bdd*               fromEbm      ( Ebm* );
  private:
    inline Key                _getKey      ();
           void               _display     ( std::ostream& );
    friend std::ostream&      operator<<   ( std::ostream&, const Bdd* );
};
```

- Le constructeur `Bdd::Bdd()`, alloue un nouveau noeud BDD et l'ajoute dans le dictionnaire des noeuds BDD.

- La méthode **Bdd::create()** 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 attributs `_index`, `_high` et `_low`. La méthode Bdd::create() 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, elle le crée en appelant le constructeur de BDD.

- La méthode **Bdd::apply()** prend pour arguments un entier définissant un opérateur Booléen (les valeurs possibles sont OR, AND, et XOR), deux pointeurs `bdd1` et `bdd2` 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 méthode `Bdd::apply()` construit le graphe ROBDD représentant la fonction `F = F1 oper F2`, et renvoie un pointeur sur le noeud ROBDD racine de ce graphe.

- La méthode **Bdd::neg()** renvoie un pointeur sur un noeud BDD représentant la négation du noeud courant `NOT(bdd)`.

- La fonction **Bdd::display()** affichera sur la sortie standard le graphe de noeuds BDD ayant le noeud courant pour racine. L'affichage sera récursif et indenté:

```
Bdd( [10] idx:4 high:8 low:7 ) - x1
```

```
  High of [10] x1
    [...]
  Low of [10] x1
    [...]
```

Ou `[X]`, `high:Y` et `low:Z` sont respectivement les identificateurs du neud courant (X), du BDD high cofacteur (Y) et du BDD low cofacteur (Z). `idx:I` est l'index de la variable suivant laquelle on décompose et `x1` son nom. Un exemple complet donnera:

```
Bdd( [10] idx:4 high:8 low:7 ) - x1
  High of [10] x1
    Bdd( [8] idx:3 high:1 low:2 ) - x2
      High of [8] x2
        Bdd( One )
      Low of [8] x2
        Bdd( [2] idx:2 high:1 low:0 ) - x3
          High of [2] x3
            Bdd( One )
          Low of [2] x3
            Bdd( Zero )
  Low of [10] x1
    Bdd( [7] idx:3 high:2 low:0 ) - x2
      High of [7] x2
        Bdd( [2] idx:2 high:1 low:0 ), already displayed.
      Low of [7] x2
        Bdd( Zero )
```

Pour la modélisation des variables, vous réutiliserez l'objet `EbmVar` vu au TME précédent. Pour ne pas bloquer sur des erreurs que contiendrais éventuellement votre implantation, des *headers* `Ebm.h`, `EbmVar.h`, `EbmExpr.h` ainsi qu'une librairie compilée `libebm.a` vous sont fournis

B) Techniques de Programmation

B.1) Constructeur pré-conditionné

Avant de créer un nouveau BDD, il est nécessaire de vérifier s'il n'existe pas déja. Cela ne peut pas être fait *dans* le constructeur puisque si nous sommes dans cette fonction, c'est que l'objet est *déjà* en cours de construction.

Pour résoudre ce problème on adopte l'architecture suivante:

- Le constructeur est rendu *privé*. Ce qui inderdit son utilisation, sauf par des fonctions membre de la classe.
- On ajoute une fonction membre statique `Bdd::create()` qui va jouer le rôle du constructeur. Comme elle est membre de la classe, elle a le droit d'appeler le constructeur. Et comme elle est `static` elle n'est pas attachée à un objet de la classe. Dans cette fonction on peut donc effectuer les vérifications (est-ce que le BDD existe déjà?) puis créer ou non un nouvel objet.

B.2) Dictionnaire de BDD

Comme il a été montré en cours, une expression booléenne sous la forme canonique de Shannon est associée à un et un seul ROBDD. Celui-çi étant entièrement caractérisé par son triplet `(_index,_high,_low)`. Lorsque la méthode `Bdd::create()` est appelée il faut qu'elle puisse déterminer rapidement si un ROBDD existe ou non.

Le conteneur approprié pour ce type d'opération est la `stl::map<>`. La `map<>` associe une valeur (le pointeur sur le ROBDD) à une clé (le triplet). Elle à les propriétés suivantes:

- Une clé d'une valeur donnée sera présente en un unique exemplaire dans la `map<>`, et sera associée à une unique valeur.

- On peut retrouver très rapidement une valeur en fonction d'une clé.

- N'importe quel objet (classe) peut servir de clé *sous réserve* de disposer d'une fonction d'ordre. C'est à dire de disposer d'une surcharge de l'operateur *strictement inférieur*.

La définition de la classe `Bdd` comporte donc une classe imbriquée `Key` représentant le triplet et proposant une fonction d'ordre. La relation d'ordre sera simplement l'ordre lexicographique du triplet. Soit:

```
if ( lhs->_index != rhs->_index ) return lhs->_index < rhs->_index;

if ( lhs->_highIndex != rhs->_highIndex )
  return lhs->_highIndex < rhs->_highIndex;

return lhs->_lowIndex < rhs->_lowIndex;
```

C) 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)

C.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.

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

D) Méthodes Bdd::neg() et Bdd::apply()

Les méthodes `Bdd::apply()` et `bdd::neg()` ont été présentées en cours. Vous trouverez le code de ces fonctions dans le fichier `Bdd.cpp`.

D.1 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 . F1H + x? . F1L
- F2 = x . F2H + x? . F2L

La relation de recurrence (F1 op F2) = x . (F1H op F2H) + x?. (F1L 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 cas particuliers où l?une des deux fonctions F1 ou F2 est égale à une des deux fonctions constantes 0 ou 1.

D.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 >

3. 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 méthode bdd::neg() sur l'objet pointé par 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 méthode bdd::neg()?

E) Méthode Bdd::fromEbm()

On cherche à écrire la méthode `Bdd::fromEbm()`, qui construit automatiquement le graphe ROBDD représentant une fonction Booléenne F, à partir d'une Expression Booléenne multiniveaux particulière de cette fonction F. Cette méthode prend comme unique argument un pointeur sur la racine d?une `Ebm`, et renvoie un pointeur sur le noeud BDD représentant la fonction. Puisque `Ebm` ne contiennent que des opérateurs OR, AND, XOR et NOT, et qu?on dispose des fonctions `Bdd::apply()` et {{{bdd::neg()}}, il est possible de construire le graphe ROBDD en utilisant uniquement ces deux fonctions.

E.1 Décrire en français, l?algorithme de cette méthode `Bdd::fromEbm()` dans le cas particulier où tous les opérandes AND, OR ou XOR présents dans l?arbre ABL n?ont que deux opérandes. On traite successivement les trois cas suivants:

- Le pointeur p désigne une variable.
- Le pointeur p désigne une expression Booléenne multi-niveaux, dont l'opérateur racine est l'opérateur NOT.
- Le pointeur p désigne une expression Booléenne multi-niveaux, dont l'opérateur racine est l'un des trois opérateurs OR, AND ou XOR.

E.2 Ecrire en langage C++ la méthode `Bdd::fromEbm()` dans le cas particulier de la question précédente. Pour valider cette fonction, on écrira un programme main(), qui effectue successivement les opérations suivantes :

- Construction de l'`Ebm` représentant l?expression E1, avec la fonction `Ebm::parse()`.
- Affichage de cette expression Booléenne, avec la fonction `Ebm::display()`, pour vérifier que l?`Ebm` est correctement construite.
- Construction du graphe ROBDD représentant la fonction F, avec la fonction Bdd::fromEbm().
- Affichage du graphe ROBDD ainsi construit, en utilisant la fonction Bdd::display().

E.3 Comment faut-il modifier la fonction `Bdd::fromEbm()`, 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.

F) Méthode Bdd::satisfy()

Soit une fonction Booléenne quelconque F(x1, x2, x3, ... xn), dépendant de n variables. Soit la fonction Booléenne G, définie comme un produit (opérateur AND) d?un nombre queconque de variables xi (directes ou complémentées). On dit que G ?satisfait? F si on a la relation G => 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). Notez que, pour une fonction F donnée, il existe plusieurs fonction G qui satisfont F...

Exemple : F = (a . b) + c

On peut avoir `G = c` ou `G = a . b` ou encore `G = a . b . c`?.

Dans la suite de cet exercice, on cherche à construire automatiquement le ROBDD représentant une fonction G satisfaisant une fronction F quelconque. Comme il existe plusieurs solutions, on choisira la systématiquement la solution qui minimise le nombre de variable `xi` dans le support de G. La méthode Bdd::satisfy() appliqué au noeud représentant F retourne un pointeur sur le noeud représentant G.

F.1 La décomposition de Shannon de la fonction F suivant la variable x d?index le plus élévé définit les cofacteurs FH et FL : F = x . FH + x? . FL Pour alléger les notations, on note sat(F) la fonction Booléenne construite par la méthode `Bdd::satisfy()` 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.

F.2 Ecrire, en langage C++, le code de la méthode `Bdd::satisfy()`, et appliquez-la sur la fonction Booléenne F définie dans la partie, en modifiant le programme main de la fonction précédente.

G) Méthode Bdd::toEbm()

On souhaite pour finir écrire la méthode Bdd::toEbm(), qui construit automatiquement une expression Booléenne multi-niveaux, à partir d'un ROBDD représentant une fonction Booléenne.

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).

G.1 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)

G.2 Comment cette expression Booléenne se simplifie-t-elle lorsque l?une des deux fonctions FH ou FL est égale à 0 ou à 1. Etudier un par un les 4 cas particuliers.

G.3 En utilisant les résultats des questions précédentes, proposez un algorithme.