Devoir de Programmation
Algorithmique Avancée

Université Paris 6: M1 Semestre 1

Etienne PENAULT & Kiara GIGACZ

Sommaire

1. Présentation

Dans ce devoir de programmation, nous allons générer des diagrammes de décision binaires, réduits et ordonnés. Pour se faire, nous avons décidé d'utiliser le **C++**. C'est un choix qui nous a paru logique étant donné que nous serons amenés au cours du devoir à manipuler des pointeurs (comme par exemple lors de la compression où nous allons assigner un même pointeur à plusieurs nœuds d'un arbre). De plus, dans un cadre d'algorithmique avancée, il est toujours plaisant d'utiliser un langage tourné vers le coté "performance". Pour utiliser notre programme, il faut le compiler avec *make*, et le lancer avec *./ALGAV* ou *./ALGAV 42* si l'on souhaite utiliser le nombre 42.

Notre programme requiert l'utilisation:

- D'une libraire essentielle pour compiler:
  - ◇ **GMP**, la librairie de GNU qui nous permet de manipuler des grands nombres.
- De deux programmes Linux à proprement parlé qui seront appelés par un appel système:
  - ◇ **Graphviz**, qui nous permettra de représenter nos arbres de manière visuelle sous forme d'image.
  - ◇ **Gnuplot**, qui nous permettra de représenter nos statistiques par rapport à nos arbres de manière visuelle sous forme d'image.

Remarque: Toutes les questions de cette sections seront implémentées dans la classe *TableOfTruth* de notre code.

1.a. Question 1.1

Comme précisé au dessus, nous utilisons **GMP** afin de manipuler des grands entiers. Pour se faire dans notre programme, il faudra faire appel à la classe de la librairie *mpz_class* pour construire et manipuler nos nombres. Voici un exemple pour construire un nombre (en base 10) du type de la classe *mpz_class* à partir d'une chaîne de caractères:

```
1 mpz_class number("42", 10);
```

1.b. Question 1.2

Pour créer la méthode *Decomposition*, nous avions commencé par utiliser la méthode des divisions euclidiennes succesives par 2 afin d'obtenir le nombre binaire associé au nombre en base 10 traité. Cependant, notre librairie permettant d'utiliser des grands nombres nous propose déjà une méthode pour nous retourner un nombre sous une autre base. Voulant passer d'un nombre en base 10 en base 2, il nous suffit de faire:

```
number.get_str(2);
```

Remarque: Notons que cette méthode nous retourne le nombre binaire sous forme de chaîne de caractères suivant la convention *Bit de poids fort à gauche*. Or nous utilisons ici la convention *Bit de poids faible à gauche* lors du projet. Il faut donc retourner cette chaîne de caractères.

1.c. Question 1.3

La methode *Completion* a pour but d'adapter la représentation de notre nombre binaire selon une taille donnée. Il suffit de rogner notre nombre binaire représenté sous forme de chaîne de caractères par rapport à la taille donnée si la taille donnée et plus petite que la taille de notre nombre binaire. Sinon, nous augmentons la taille de notre chaîne de caractères représentant notre nombre binaire en insérant le nombre de "0" adéquat (donc la taille du nombre moins la taille voulu). Notre méthode ressemble donc à ça:

```
void TableOfTruth::Completion(int finalSize) {
    int nSize = _number.size();

    if (finalSize <= nSize) {
        _number = {_number.begin(), _number.end() + (finalSize - nSize)};
    } else {
        _number.insert(_number.end(), finalSize - nSize, '0');
    }
}
```

1.d. Question 1.4

La méthode *Table* consiste uniquement à l'assemblage des méthodes *Decomposition* et *Completion*:

```
std::string TableOfTruth::Table(const mpz_class &x, int n) {
    Decomposition(x);
    Completion(n);
    return _number;
}
```

2. Arbre de décision et compression

Notre devoir à la particularité de comporter deux implémentations différentes (donc deux structures de données différentes) pour compresser les ROBDDs et plus particulièrement les nœuds associés aux mots de *Lukasiewicz*.

2.a. Question 2.5

Afin de stocker nos arbres, nous avons décidé de définir une structure de données *Node*, et une classe *Tree*. Regardons ce qui les compose.

2.1.1. Représentation d'un nœud

Comme dit précédemment, nous avons défini une structure de données *Node* afin de l'utiliser dans la représentation de nos arbres. Nous définissons cette structure et ses fonctions associées dans *Node.h* et *Node.cpp*.

Cette structure est composée de 3 champs:

- Une valeur (afin de stocker le mot de *Lukasiewicz* associé) qui est une chaîne de caractères.
- Un fils gauche qui est pointeur de nœud.
- Un fils droit qui est pointeur de nœud.

```
1 typedef struct node Node;
2 struct node {
3     std::string _value;
4     std::shared_ptr<Node> _leftChild;
5     std::shared_ptr<Node> _rightChild;
6 };
```

Remarque: Nous avons utilisé les *shared_ptr* de la bibliothèque standard de C++ et non des pointeurs classiques afin de ne pas gérer la mémoire à la main. L'utilisation d'*unique_ptr* est à proscrire ici car nous avons pour but d'assigner plusieurs valeurs à un même pointeur.

De plus nous avons également fait des fonctions associées à l'utilisation de cette structure afin de faciliter son utilisation. Elles sont triviales et possèdent des noms explicites, nous ne les détaillerons pas ici mais voici leurs prototypes:

```
1 std::shared_ptr<Node> newNode(std::shared_ptr<Node> left,
2                                std::shared_ptr<Node> right,
3                                std::string value);
4 std::shared_ptr<Node> insert(std::shared_ptr<Node>& initTree,
5                               std::shared_ptr<Node> treeToAdd);
6 int getTreeHeightFromWidth(int width);
```

2.1.2. Représentation d'un Arbre

La représentation de nos arbres est basée sur la classe *Tree*. Cette classe possède deux éléments privés. Nous allons laisser le deuxième de côté pour le moment car il nous est pas encore utile. Nous y reviendrons lors de la partie sur l'étiquetage des nœuds de l'arbre par les mots de *Lukasiewicz*.

Pour le moment, notre classe sera composée uniquement d'un pointeur de *Node* qui nous permettera de stocker la racine de l'arbre, ce qui nous permettra de parcourir l'arbre à notre guise:

```
1 private:
2   std::shared_ptr<Node> _root;
```

2.b. Question 2.6

La méthode *ConsArbre* prend en argument une référence de chaîne de caractères. *ConsArbre* se contente d'appeler *createTreeFromTable*, une méthode auxiliaire récursive de notre classe.

Remarque: Nous utiliserons cette manière de faire avec les méthodes auxiliaires lorsque nous avons besoin d'utiliser la récursion.

createTreeFromTable est donc basée sur une récursion. Nous allons construire un arbre selon une hauteur passée en paramètre:

- Nous crééons un fils gauche et un fils droit au nœud courant.
- Lorsque la méthode est rappelée, nous décrémentons cette hauteur avant de la passer en argument.
- Nous rappelons la méthode sur le fils gauche créé, ainsi que le fils droit créé.
- Quand cette hauteur est à 0:
  - ◇ Nous savons que nous sommes sur les feuilles, c'est notre condition d'arrêt.
  - ◇ Nous en profitons pour étiqueter les feuilles par leurs valeurs depuis la table de vérité passée en argument (on retire l'élément que l'on assigne de la table de vérité, qui est toujours le premier élément de la table de vérité).

2.c. Question 2.7

2.3.1. Comment stocker les nœuds associés aux mots de Lukasiewicz?

Comme dit précédemment, nous avons implémenté deux manières pour stocker les nœuds vis-à-vis des mots de *Lukasiewicz*.

Par:

- Une Table de Hachage (L'*unordered_map* de la bibliothèque standard de C++).
- Un Trie binaire (que nous avons implémenté nous-mêmes dans la classe nommée *BTrie*).

Chacune de ces manières a des avantages et des inconvénients dont nous discuterons plus tard, mais ont le même intérêt.

Le principe de ces structures est de stocker un nœud selon un mot de *Lukasiewicz* que nous utiliserons en tant que clef. Autrement dit, dès que nous voulons obtenir un nœud associé à un mot de *Lukasiewicz*, il nous suffit de questionner la structure de données en question afin de le récupérer. Cette structure de données est le second élément de notre classe *Tree* mentionné plus tôt, qui est nommé dans le code *_words*.

2.3.2. Par Table de Hachage

En utilisant une Table de Hachage, le second élement privé de notre classe serait donc une *unordered_map* ayant pour clef une chaîne de caractères, et comme valeur des pointeurs de nœuds. En termes de code nous aurons donc:

```
1 private:
2  std::unordered_map<std::string, std::shared_ptr<Node>> _words;
```

2.3.3. Par Trie Binaire

Avant de préciser à quoi ressemble *_words* dans la classe *Tree* en utilisant un Trie Binaire, regardons à quoi ressemble la classe *BTrie* qui sera utilisée. Voici dans un premier temps ses éléments privés:

```
1 private:
2  BTrie *_number[2];
3  std::shared_ptr<Node> _ptr;
```

Notre Trie possède donc deux champs:

- Un pointeur de *BTrie* qui correspond à ses fils (qui est ni plus ni moins qu'un tableau des caractères possibles, dans notre cas '0' ou '1', donc de taille 2)
- Un pointeur de *Node* qui sera assigné s'il existe un nœud correspondant à la clef (qui est un mot de *Lukasiewicz*) nous amenant à cet endroit au sein du Trie.

Ensuite, penchons nous sur ses méthodes:

```
1 void CheckAndInsert(const std::shared_ptr<de> &n);
```

Celle-ci gère les insertions dans notre Trie:

- Selon notre pointeur de nœud de type *Node* passé en argument (dont la clef est le mot de *Lukasiewicz* du nœud stocké dans son membre *_value*), nous parcourons le Trie vis-à-vis du caractère courant (soit un '0', soit un '1' donc on convertie ces caractères en index, qui sont des nombres entiers afin de les utiliser dans le membre *_number* du Trie courant).
- Si nous essayons d'accéder à un successeur du Trie qui n'est pas défini (donc notre Trie ne possède aucun mot de *Lukasiewicz* commençant par les mêmes caractères) par le membre *_number*, nous en créons un.
- Sinon on se contente de parcourir le Trie.
- Une fois notre Trie parcouru et avec les bons successeurs créés si besoin, il nous reste plus qu'à assigner le pointeur du nœud passé en argument au pointeur *_ptr* du successeur du Trie où nous nous situons (du coup à l'endroit correspondant à la clef dans l'arborescence du Trie).

Ensuite, nous avons cette fonction membre:

```
1 std::shared_ptr<Node> Find(std::string key);
```

qui nous retourne le pointeur du nœud associé à une clef dans notre Trie (qui est en fait un mot de *Lukasiewicz*). Cette dernière est basée sur un parcours du même type que la méthode *CheckAndInsert*. Dès que nous sommes au bon endroit en parcourant notre Trie selon la clef, nous retournons le nœud *Node* associé au Trie courant (*_ptr* dans notre structure Trie).

Pour finir nous avons en plus trois autres méthodes publiques pour notre classe *BTrie*:

```
1 void Display();
2 void Remove(std::string &key);
3 void RemoveAll();
```

Remove est basée sur le même parcours que les deux méthodes présentées précédemment, tandis que *Display* et *RemoveAll* sont basées sur des récursions parcourant l'entiereté du Trie.

2.3.4. La méthode *Luka*

Maintenant que nous avons défini comment stocker des pointeurs de nœuds vis-à-vis des mots de *Lukasiewicz* en tant que clef, nous pouvons établir l'étiquetage de notre arbre avec les mots. Pour se faire, nous allons procéder comme suit dans notre méthode *Luka*:

- Cette méthode appelle la méthode auxiliaire récursive *lukaAux* qui prend en argument un pointeur d'un nœud.

- On rappelle la méthode sur le fils gauche du nœud.
- On rappelle la méthode sur le fils droit du nœud.
- On définit ensuite la valeur du mot de *Lukasiewicz* (donc stocké dans l'attribut *_value* du nœud) comme étant la concaténation du mot de son fils droit et du mot de son fils gauche.
- On insére le nœud dans notre structure de données *_words* si un nœud n'est pas déjà présent pour cette valeur de clef (qui est le mot de *Lukasiewicz*):
  - ◇ Pour la méthode avec une Hashmap, nous devons vérifier au préalable si il existe déjà un élément avec une clef identique, cela ressemble à cela:

```
1 if (_words.find(n->_value) == _words.end()) {
2     // not found
3     _words[n->_value] = n;
4 }
```

  - ◇ Pour le BTrie, la méthode *CheckAndInsert* vérifie en même temps que l'insertion si il existe déjà un nœud assigné pour la valeur de la clef, il suffit juste de faire:

```
1 _words->CheckAndInsert(n);
```

- Enfin la condition d'arrêt est quand un des fils du nœud courant n'est pas défini (*nullptr*), où nous prenons soin d'également ajouter le nœud de la feuille actuelle dans *_words* si il n'y a aucun nœud avec ce mot dans la structure de données (naturellement cette étape est en début de méthode).

Voici un exemple d'un arbre étiqueté après l'appel de la méthode *Luka*:

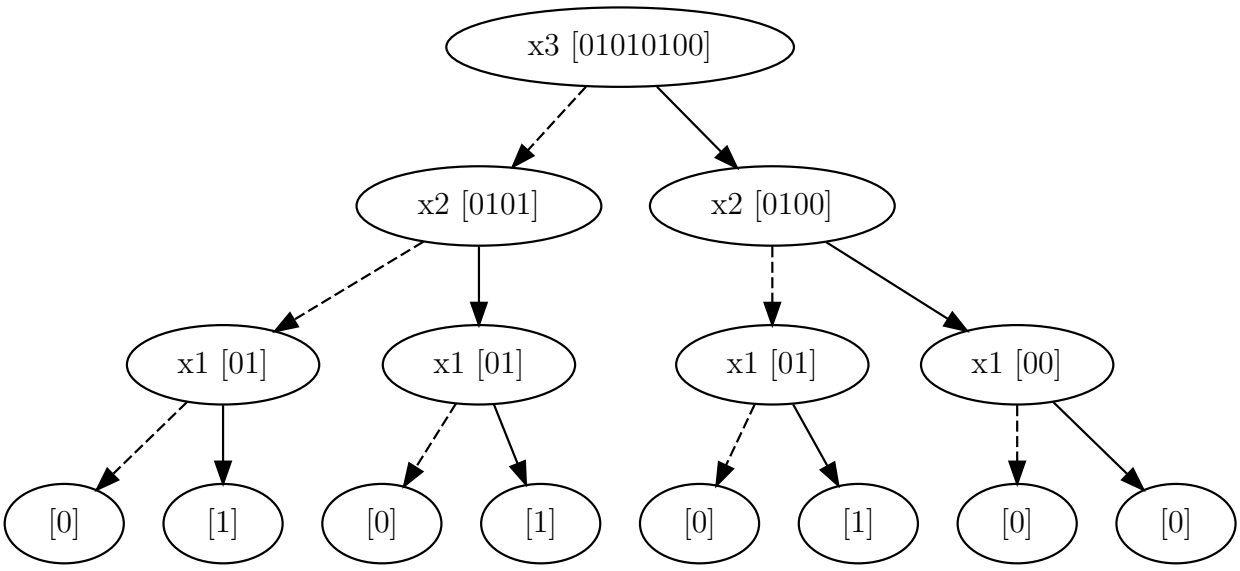

Figure 1: Arbre du nombre 42

2.d. Question 2.8

La méthode *Compress* appelle une méthode *compressAux* qui est récursive. *compressAux* prend en argument un pointeur de nœud (nous appellerons cette fonction sur la racine de l'arbre). Cette méthode repose sur un parcours préfixe de l'arbre:

- Notre condition d'arrêt est invoquée lorsque le pointeur d'un nœud n'est pas défini (*nullptr*).
- On assigne au nœud courant le nœud correspondant au mot de *Lukasiewicz* actuel en le récupérant dans notre structure de données *_words* si le nœud est différent:
  - ◇ Pour la méthode avec une Hashmap:

```
if (n != _words[n->_value]) {
        n = _words[n->_value];
    }
```

  - ◇ Pour le BTrie:

```
std::shared_ptr<Node> toAssign = _words->Find(n->_value);
if (toAssign != n) {
    n = toAssign;
}
```

- On rappelle la méthode sur le fils gauche.
- On rappelle la méthode sur le fils droit.

Voici un exemple d'un arbre après l'appel de la méthode *Compress*:

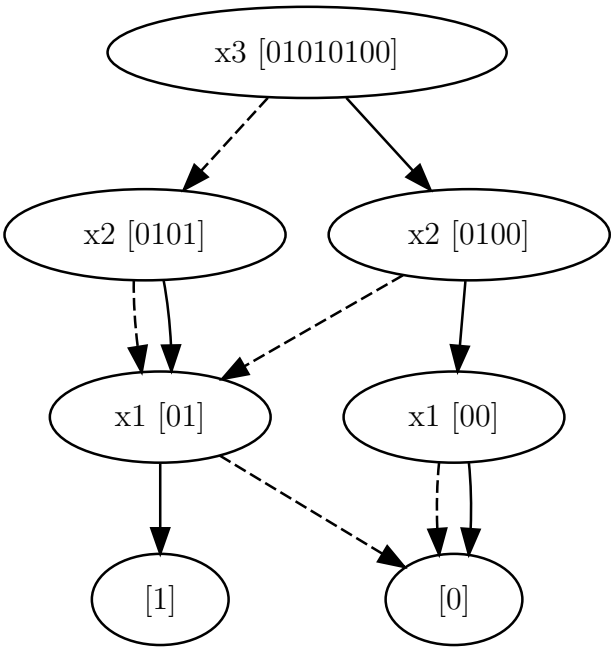

Figure 2: Arbre du nombre 42 compressé

2.e. Question 2.9

La méthode *Dot* construit un fichier *.dot* selon le nom que l'on lui passe en argument. Nous n'allons détailler cette méthode que brièvement car elle ne sert qu'à l'affichage de notre arbre.

Dans un premier temps nous remplissons le fichier *.dot* par les méthodes:

- *defineInDot* qui nous sert à définir tous les nœuds de notre arbre selon leurs adresses que nous écrirons dans le fichier. De plus, par la taille des mots de *Lukasiewicz*, nous labélisons les nœuds dans le fichier par *x1, x2, x3....* Si besoin, nous ajoutons en plus au nom d'un nœud son mot de *Lukasiewicz*.
- *linkInDot* relie les nœuds selon l'arborescence de l'arbre à afficher.

Ensuite, nous faisons un appel système à la comande Linux *dot* afin de générer l'image à partir du fichier *.dot* que l'on vient de créer. Les fichiers finaux se trouveront dans le dossier "tree" à la racine du projet.

Remarque: Nous pouvons passer un argument en plus à la méthode *Dot* afin d'indiquer si l'on veut labeliser notre arbre avec les mots de *Lukasiewicz* ou non dans notre arbre (il est utile pour les grands nombre de désactiver cet affichage, cependant nous conservons les *x1, x2, x3...*).

3. Arbre de décision et ROBDD

3.a. Question 3.10

La méthode *CompressionBDD* appelle une méthode *compressAux* qui est récursive. *compressAux* prend en paramètre un pointeur vers le nœud de l'arbre que l'on souhaite compresser, un pointeur vers son parent, et un flag qui indique s'il est:

- La racine
- Le fils gauche de son parent
- Le fils droit de son parent

Dans cette méthode, nous appliquons les trois règles de compression citées dans l'article de Newton et Verna : la *terminal rule*, la *merging rule* et la *deletion rule* (alors que *Compress* n'appliquait uniquement la *terminal rule* et la *merging rule*).

Nous avons codé cet algorithme avec un parcours suffixe:

- Notre condition d'arrêt est invoquée lorsque le pointeur du nœud passé en paramètre n'est pas défini (*nullptr*).
- On appelle la méthode sur le fils gauche.
- On appelle la méthode sur le fils droit.
- On applique la deletion rule sur le nœud courant, en vérifiant si son fils gauche et son fils droit représentent le même mot. Si cela est le cas, nous faisons pointer le parent du nœud courant vers le fils du nœud courant (tout en supprimant le nœud courant). Pour la version avec une HashMap, celle reviendrait à ça:

```
if (n->_leftChild != nullptr && n->_rightChild != nullptr &&
    n->_leftChild->_value == n->_rightChild->_value) {
    // update lukas map
    _words.erase(n->_value);
    auto cpy = n;
    if (from == NOTHING) {
        n = n->_leftChild;
    } else if (from == LEFT) {
        parent->_leftChild = n->_leftChild;
    } else if (from == RIGHT) {
        parent->_rightChild = n->_rightChild;
    }
}
```

- Pour terminer, nous appliquons les règles *merging rule* et *terminal rule* en remplaçant le pointeur du nœud courant par le pointeur de l'unique nœud que l'on veut conserver avec ce mot (celui stocké dans notre structure de données *_words*).

Remarque 1: Dans le cas particulier où le nœud courant est la racine, nous avons décidé de remplacer la racine du ROBDD par son fils. Ce choix a été fait pour suivre la même convention que celle utilisé dans l'article de Newton et Verna, en particulier pour pouvoir avoir une étude expérimentale comparable à la leur.

Remarque 2: Nous avons essayé de réfléchir à une façon de faire la compression par un parcours préfixe pour davantage améliorer la complexité. Malheureusement il nous semble impossible de le faire avec un parcours en profondeur comme nous l'avons implémenté. Nous pensons que cela pourrait peut-être marcher si l'algorithme était plutôt un parcours en largeur.

Voici un le ROBDD après l'appel de la méthode *CompressionBDD*:

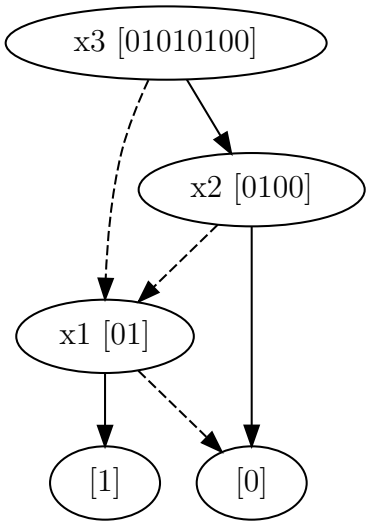

Figure 3: ROBDD du nombre 42

3.b. Question 3.11

Nous allons montrer dans cette partie que la longueur du mot de *Lukasiewicz* de la racine d'un arbre de hauteur $1 \leq h \leq 9$ est majorée par $l_h = 11 \cdot 2^h - 6$.

Pour cela, nous allons commencer par montrer qu'il y a $2^h - 1$ nœuds internes, 2^h feuilles dans un arbre. Par convention, nous considérerons que la hauteur d'un arbre qui contient un nœud est 0.

3.2.1. Nœuds internes

Base:

h = 0, nombre de nœuds internes = 0.
$(2^h - 1) = 1 - 1 = 0$

La propriété est vraie pour h = 0.

Induction:

Supposons que la propriété est vraie pour une hauteur h. Montrons qu'elle est vraie pour une hauteur h+1. Pour construire un arbre de hauteur $h+1$, nous relions deux arbres de hauteur h par un nouveau nœud qui sera la racine du nouvel arbre. Par hypothèse d'induction, nous avons donc $2^h - 1$ nœuds internes pour chaque arbre de hauteur h, plus le nouveau nœud. Cela nous donne donc: $2(2^h - 1) + 1 = 2^{h+1} - 2 + 1 = 2^{h+1} - 1$ nœuds internes pour un arbre de hauteur $h + 1$.

3.2.2. Feuilles

Base:

h = 0, nombre de feuilles = 1.
$2^h = 2^0 = 1$
La propriété est vraie pour h = 0.

Induction:

Supposons que la propriété est vraie pour une hauteur h. Montrons qu'elle est vraie pour une hauteur h+1.

Pour construire un arbre de hauteur $h+1$, nous relions deux arbres de hauteur h par un nouveau nœud qui sera la racine du nouvel arbre. Par hypothèse d'induction, nous avons donc 2^h feuilles pour chaque arbre de hauteur h. Cela nous donne donc: $2(2^h) = 2^{h+1}$ feuilles pour un arbre de hauteur $h + 1$.

3.2.3. Nombre total de nœuds d'un arbre

= Nombre de nœuds internes + Nombre de feuilles
$= (2^h - 1) + (2^h)$
$= 2(2^h) - 1$

Un mot de *Lukasiewicz* est composé de trois éléments :

- Les parenthèses ouvrantes et fermantes
- Les chaînes représentant les noms de variables, de la forme "x_" où _ peut être un chiffre de 1 à 9
- Les chaînes représentant les valeurs des feuilles

La longueur d'un mot de *Lukasiewicz* est donc la somme des caractères de ces trois types d'éléments.

Parenthèses:

Il y a une parenthèse ouvrante et une parenthèse fermante pour chaque nœud de l'arbre, sauf la racine. Cela nous donne donc : $2 \times (nb\ total\ de\ nœuds - 1) = 2(2(2^h) - 2) = 4(2^h) - 4$

caractères.

Noms de variables:

Le nombre de variables est égal à la hauteur de l'arbre. Les noms de variables sont composé d'un x et d'un nombre représentant leur étage dans l'arbre. Étant donné que la hauteur est au plus de 9, les étages sont représentés par des chiffres (c'est-à-dire que la longueur du nombre est de 1). Chaque nœud interne de l'arbre est donc étiqueté par un nom de variable qui est de longueur 2. Cela nous donne donc $2 \times nb\ n\text{œuds}\ internes = 2(2^h - 1) = 2(2^h) - 2$ caractères.

Valeurs des feuilles:

La valeur d'une feuille peut être "True" (4 caractères) ou "False" (5 caractères). Comme nous souhaitons majorer la longueur du mot, nous considérerons le pire cas, c'est-à-dire quand toutes les feuilles sont étiquetées par "False". Cela nous donne donc $5 \times nb\ feuilles = 5(2^h)$ caractères.
La longueur d'un mot de *Lukasiewicz* à la racine d'un arbre est donc majorée par

$$4(2^h) - 4 + 2(2^h) - 2 + 5(2^h)$$

$$= 11(2^h) - 6$$

3.c. Question 3.12

Dans l'algorithme de compression, une partie importante en termes de complexité en comparaison de caractères est la recherche d'autres nœuds ayant le même mot de *Lukasiewicz* pour n'en garder qu'un seul. Nous allons dans un premier temps considérer une méthode naïve de compression.

Pour cela, à chaque étage r de l'arbre, nous comparons chaque nœud à ses frères (deux nœuds ne peuvent avoir le même mot de *Lukasiewicz* que s'ils sont sur le même étage car tous les nœuds sur un même étage ont la même longueur). Pour chaque nœud de l'étage, nous faisons donc au maximum 2^r comparaisons de nœuds.

Pour davantage affiner ce calcul, nous considérons que dans la réalité, seul le premier nœud d'un étage aura besoin de faire 2^r comparaisons. Les nœuds suivants auront besoin d'une comparaison de moins que leur prédécesseur immédiat.

Pour chacune de ces comparaisons entre deux nœuds, nous faisons dans le pire cas un nombre de comparaisons de caractères équivalent à la longueur des mots à cet étage, c'est-à-dire l_{h-r}.

Cela nous amène à une complexité en nombre de comparaison de caractères de :

$$\sum_{r=0}^{h}\sum_{i=1}^{2^r}(i-1)2^r \cdot l_{h-r}$$

$$= \sum_{r=0}^{h}\sum_{i=1}^{2^r}(i-1)2^r \cdot (11 \cdot 2^{h-r} - 6)$$

Les constantes étant négligeables, nous considérons

$$11\sum_{r=0}^{h}\sum_{i=1}^{2^r}(i)2^r \cdot 2^{h-r}$$

$$= 11\sum_{r=0}^{h}\sum_{i=1}^{2^r}(i)2^h$$

$$= 11 \cdot 2^h \sum_{r=0}^{h}\sum_{i=1}^{2^r} i$$

$$= 11 \cdot 2^h \sum_{r=0}^{h} 2^{r-1}(2^r + 1)$$

$$\simeq 11 \cdot 2^h \sum_{r=0}^{h} 2^r \cdot 2^r$$

$$= 11 \cdot 2^h \sum_{r=0}^{h} 2^{2r}$$

$$= 11 \cdot 2^h \cdot (2^{2h+1} - 1)$$

$$\simeq 11 \cdot 2^{3h+1}$$

Maintenant considérons la stratégie de compression améliorée du cours. Cette stratégie nous permet de compresser l'arbre avec l_r comparaisons de caractères pour chaque nœud de l'arbre, r étant l'étage du nœud dans l'arbre.

La complexité au pire cas en fonction de n serait donc

$$\sum_{r=0}^{h} 2^r(11 \cdot 2^r - 6)$$

Les constantes étant négligeables, nous considérons

$$11h \sum_{r=0}^{h} 2^r \cdot 2^r$$

$$11h \sum_{r=0}^{h} 2^{2r}$$

$$= 11h(2^{2h+1} - 1)$$

$$\simeq 11h \cdot 2^{2h}$$

Nous trouvons donc une complexité au pire cas de $\mathcal{O}(11\text{h} \cdot 2^{2h})$.

3.d. Question 3.13

Nous avons vu plus haut que le nombre total de nœuds, n, d'un arbre de décision de hauteur h est $2(2^h) - 1$. Nous pouvons aussi exprimer la hauteur de l'arbre en fonction de n :

$$h = log_2(\frac{n+1}{2})$$

La complexité au pire cas en fonction de n serait donc

$$11(log_2(\frac{n+1}{2})) \cdot 2^{2log_2(\frac{n+1}{2})}$$

$$11(log_2(\frac{n+1}{2})) \cdot (\frac{n+1}{2})$$

Les constantes étant négligeables, nous avons donc une complexité au pire cas en $\mathcal{O}$(n log n).

3.e. Question 3.14

Nous avons vu à la Question 3.11 que la hauteur bornée entre 1 et 9 nous permettait de déduire que la longueur de l'étiquette d'un nœud de l'arbre était de 2 caractères. Si la hauteur n'était plus bornée par 9, nous pouvons voir que la longueur de l'étiquette croîtra de 1 pour chaque puissance de dix.

4. Étude expérimentale

4.a. Question 4.15

Voici les courbes réalisées en se basant sur la distribution non exhaustive du nombre de fonctions par rapport au nombre de variables pour 1, 2, 3 et 4 variable(s). Nous voulions essayer de réaliser un test pour la distribution non exhaustive pour 5 variables, mais le temps estimé avoisinait les 6/7 heures, nous ne l'avons donc pas réalisé.

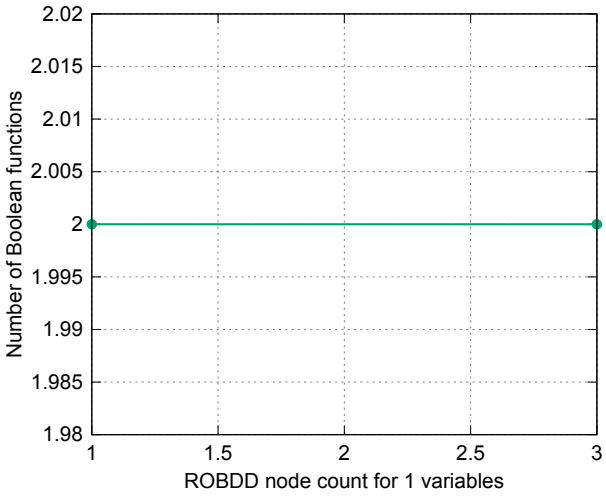

Figure 4: 1 variables

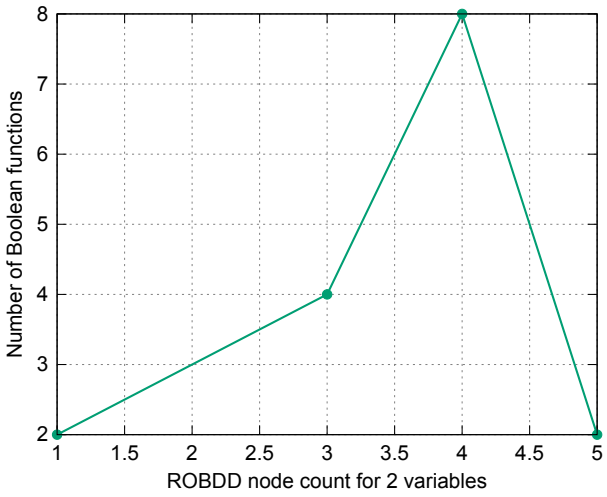

Figure 5: 2 variables

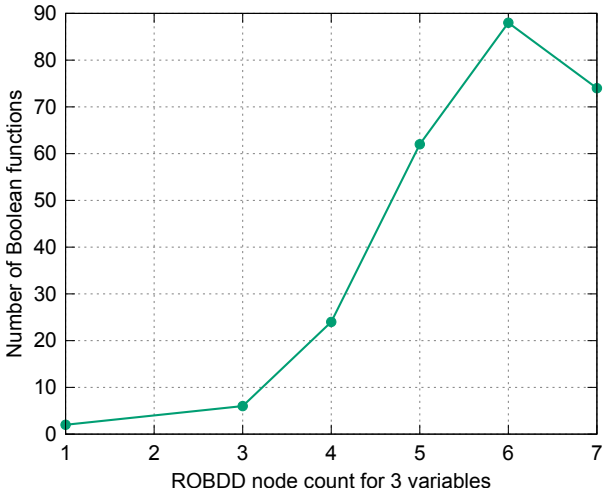

Figure 6: 3 variables

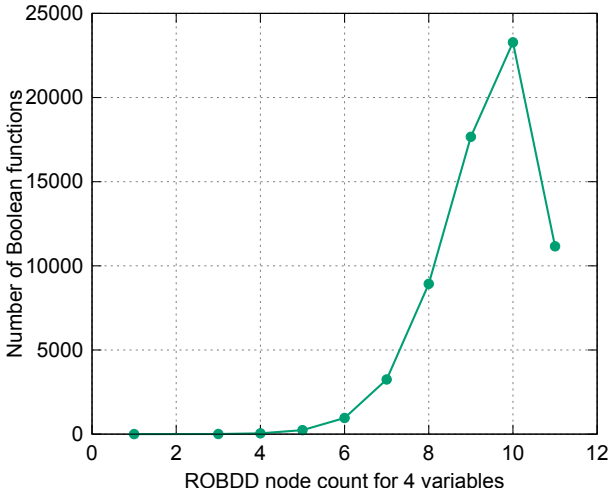

Figure 7: 4 variables

4.b. Question 4.16

Ces courbes ont été réalisées par échantillonage, donc d'une manière différente des courbes précédemment vu. Étant donné qu'il y avait beaucoup trop de ROBDD différents pour les nombres comportant plus de 4 variables, nous avons tiré aléatoirement des nombres (en

s'assurant de ne pas tirer deux nombres identiques à l'aide d'un *set*) et construit leurs ROBDD afin de réaliser ces statistiques. Il est intéressant de noter que la courbe devient au fur et à mesure plus "lisse" malgré que nous prenons un échantillonage de même cardinalité pour tout ces tests, peu importe le nombre de variables. On peut en conclure que plus nous montons en termes de variables, plus nous avons un nombre de nœuds différents par ROBDD. Ces graphiques on été générés par la fonction *plot* en tête de fichier dans le fichier main.

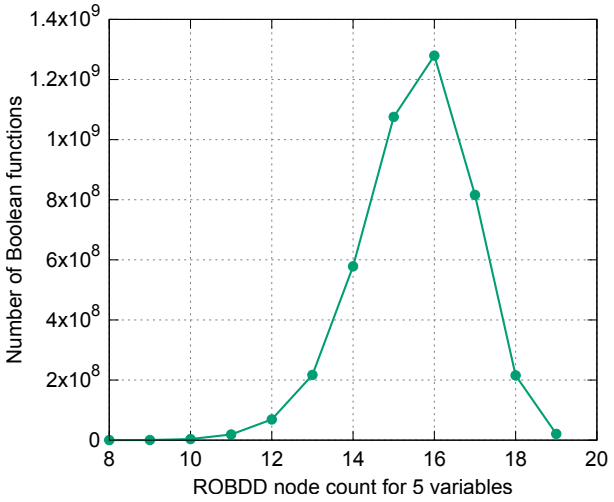

Figure 8: 5 variables

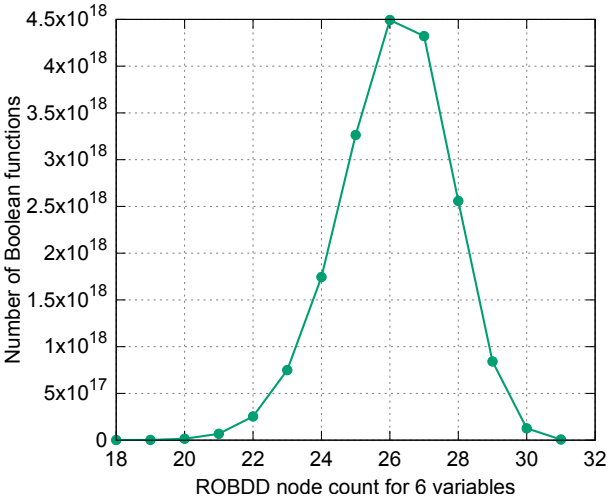

Figure 9: 6 variables

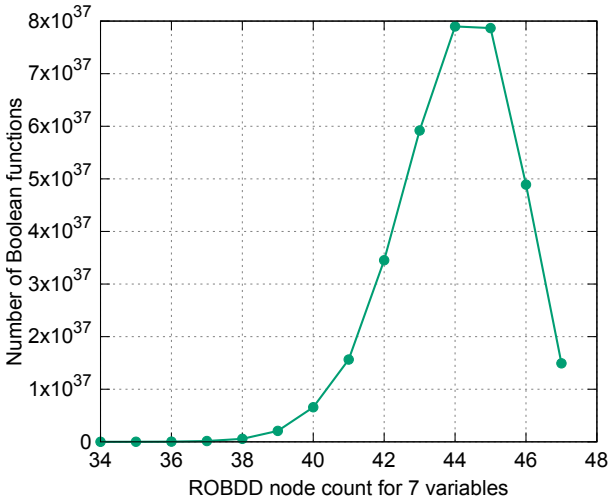

Figure 10: 7 variables

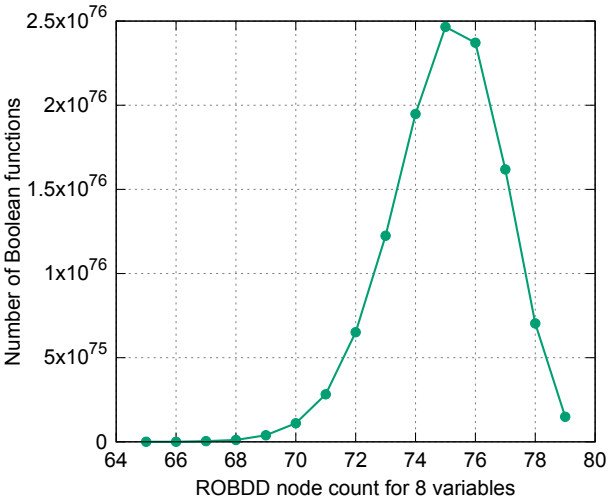

Figure 11: 8 variables

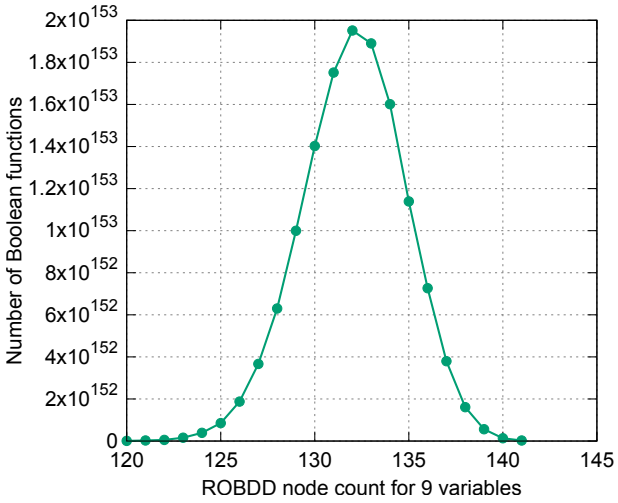

Figure 12: 9 variables

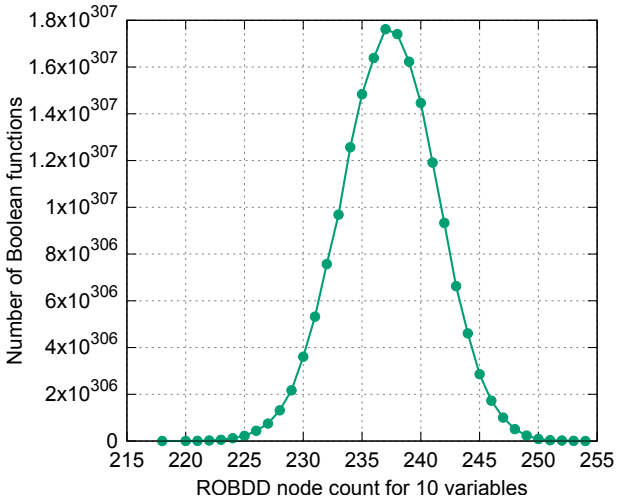

Figure 13: 10 variables

4.c. Question 4.17

Nous avons utilisé deux structures de données pour stocker nos pointeurs de nœuds vis-à-vis des mots de *Lukasiewicz*. Nous avons donc fait deux tableaux et allons les comparer:

Nombre de Variables	Nombre d'Échantillons	Nombre de Tailles Uniques	Temps d'Éxecution (en s)	Temps par ROBDD (en µs)
5	100000	13	0.721746	5.25439
6	100000	16	1.38005	11.0973
7	100000	13	2.71223	22.672
8	100000	16	5.3939	46.1506
9	100000	25	13.4248	103.087
10	100000	35	33.1934	246.32

Figure 14: Données par échantillonnage (Hashmap en $\mathcal{O}(1)$)

Nombre de Variables	Nombre d'Échantillons	Nombre de Tailles Uniques	Temps d'Éxecution (en s)	Temps par ROBDD (en µs)
5	100000	14	0.805422	4.22426
6	100000	14	1.71721	10.6015
7	100000	13	3.71506	24.0502
8	100000	15	8.05614	53.6603
9	100000	23	28.2274	158.154
10	100000	33	71.2974	443.935

Figure 15: Données par échantillonnage (BTrie en $\mathcal{O}(2^h)$)

On se rend compte que l'utilisation du BTrie est plus rapide que l'utilisation d'Hashmap lorsque nous travaillons sur des arbres n'ayant pas beaucoup de variables. Cependant, nous nous rendons compte que sur un nombre plus conséquent de variables, la Hashmap est plus performante. Cela peut s'expliquer par le fait que la taille du BTrie est proportionelle à la longueur des mots de *Lukasiewicz* insérés dedans ($\mathcal{O}(2^h)$) tandis qu'il n'y a pas de relation entre la longueur des mots et la rapidité d'accés à un élément dans la Hashmap ($\mathcal{O}(1)$).

Nous pourrions gagner en temps d'exécution avec l'utilisation d'un Trie qui ne serait pas binaire. Actuellement, dans notre BTrie, nous naviguons et insérons des éléments selon les caractères '0' et '1'. Cependant, une manière plus astucieuse pour implémenter le Trie serait de rassembler tous les '0' et '1' ensemble lorsqu'ils n'aboutissent pas directement à un chemin vers successeur possédant un nœud *Node* (en quelque sorte les paquer) afin de rajouter cela en un seul chemin dans le Trie. Cela éviterait les étapes intermediaires lors des parcours au sein du Trie. Pour illustrer cette idée, voici un exemple:

Compression des mots 11110100, 0100, 01, 0, 1 (les numéros soulignés comportent le nœud du mot associé):

Avec la méthode naïve:

$$\nearrow \quad \underline{1} \to 1 \to 1 \to 1 \to 0 \to 1 \to 0 \to \underline{0}$$

$$\searrow \quad \underline{0} \to \underline{1} \to 0 \to \underline{0}$$

Avec la méthode astucieuse:

$$\nearrow \quad \underline{1} \to \underline{1110100}$$

$$\searrow \quad \underline{0} \to \underline{1} \to \underline{00}$$

5. Pour aller plus loin...

5.a. Question 5.18

La méthode *Meld* appelle une méthode *meldAux* qui est récursive. *meldAux* utilise l'algorithme proposé par Knuth dans le volume 4 du livre *The Art of Computer Programming* pour fusionner deux ROBDD. La méthode prend en paramètre un pointeur vers un nœud A du premier arbre, un pointeur vers un nœud B du deuxième arbre, une chaîne de caractères représentant la table de vérité de l'opération que l'on souhaite appliquer durant la fusion, ainsi que le nouvel arbre *Tree* à remplir. La chaîne de caractère pour la table de vérité peut être vide dans la version avec HashMap si nous souhaitons avoir un arbre fusionné sans l'application d'une opération mais doit avoir obligatoirement une chaîne de caractères passé pour la version avec un BTrie. En effet, notre Trie gére uniquement les caractères '0' et '1', et non les caractères ' ' et '|' qui sont utilisés pour stocker les mots de *Lukasiewicz* fusionnés.

Remarque: La chaîne de caractères représentant la table de vérité a pour convention (*str* étant notre chaîne, l'opérateur [ ] indiquant la position d'un caractère et $\diamond$ l'opération à appliquer):

Première valeur X	Deuxième valeur Y	Valeur final après X $\diamond$ Y
0	0	$str[0]$
0	1	$str[1]$
1	0	$str[2]$
1	1	$str[3]$

Figure 16: Table de vérité sous forme de chaîne de caractères

En utilisant cette convention, la table de vérité sous forme de chaîne de caractères pour l'opérateur binaire *AND* serait "0001".

Notre condition d'arrêt est invoquée lorsqu'un des pointeurs passé en paramètre n'est pas défini (*nullptr*). Pour appliquer l'algorithme de Knuth, nous comparons la longueur des mots de *Lukasiewicz* des nœuds A et B :

- Si la longueur des deux mots est équivalente, on créé un nouveau nœud où le fils gauche est le résultat de l'appel de la fonction *meldAux* sur les fils gauches de A et de B, et le fils droit le résultat de la fusion des fils droits de A et de B.
- Si la longueur du mot de A est supérieure à celle du mot de B, alors on créé un nouveau nœud dont le fils gauche est le résultat de la fusion du fils gauche de A avec B, et le fils droit est le résultat de la fusion du fils droit de A avec B.
- Sinon, la longueur du mot de B est supérieure à celle de A. On créé donc un nouveau nœud dont le fils gauche est le résultat de la fusion du fils gauche de B avec A, et le fils droit est le résultat de la fusion du fils droit de B avec A.

- Si une table de vérité est passé en arguments, alors on transforme les valeurs des mots de *Lukasiewicz* finaux selon cette table (en faisant du cas par cas) et la convention expliqué si dessus.

Remarque: On n'oublie pas d'insérer les mots de *Lukasiewicz* dans la structure de données *_words* lorsque l'on crée un nouveau nœud à l'arbre final.

Une fois la fusion faite, nous pouvons appliquer la méthode *compressionBDD* pour compresser l'arbre résultant.

5.b. Question 5.19

Pour vérifier que notre implémentation de la fusion est correcte, nous avons testé plusieurs cas.

Le premier exemple est celui du cours:

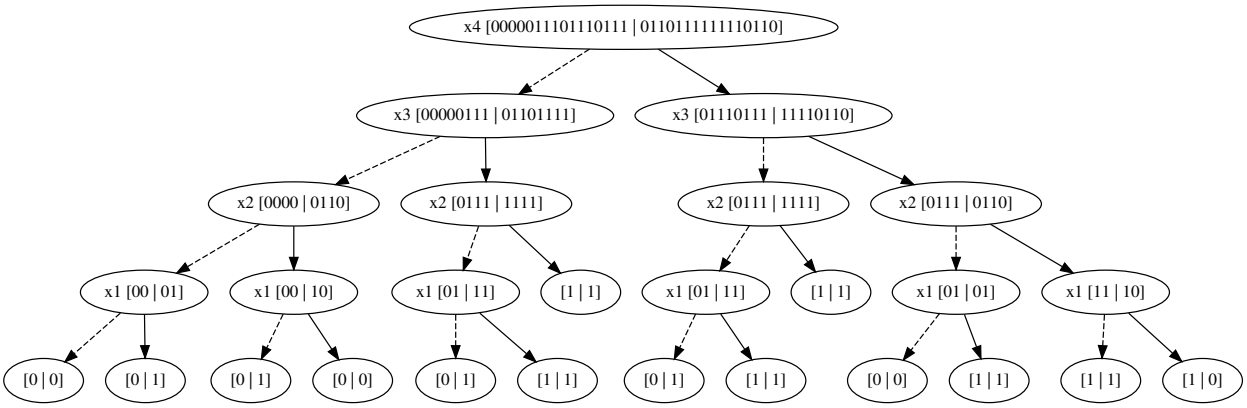

Figure 17: ROBDD résultant de la fusion des ROBDD des nombres 61152 et 28662

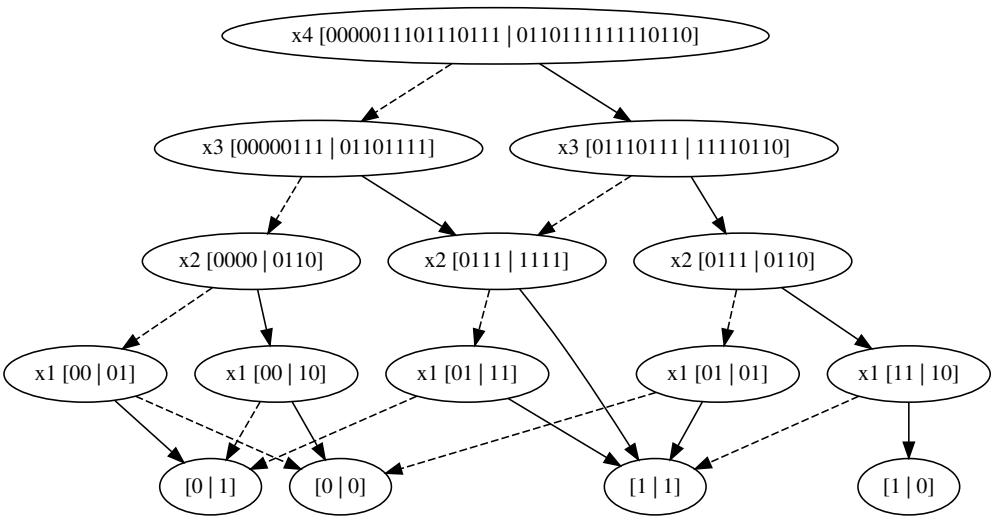

Figure 18: ROBDD de la figure ci-dessus compressé

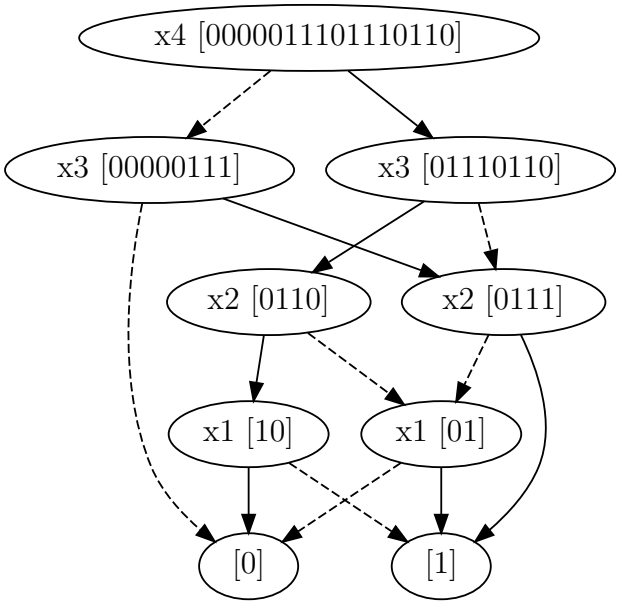

Figure 19: Fusion des ROBDD des nombres 61152 et 28662 avec la table de vérité de l'opérateur *AND* ("0001")

Notre résultat semble cohérent car:

$$0000011101110111 \ \& \ 0110111111110110 = 0000011101110110$$

Nous avons aussi testé la fusion avec des arbres de tailles différentes:

Par exemple avec les nombres 136 et 28662 :

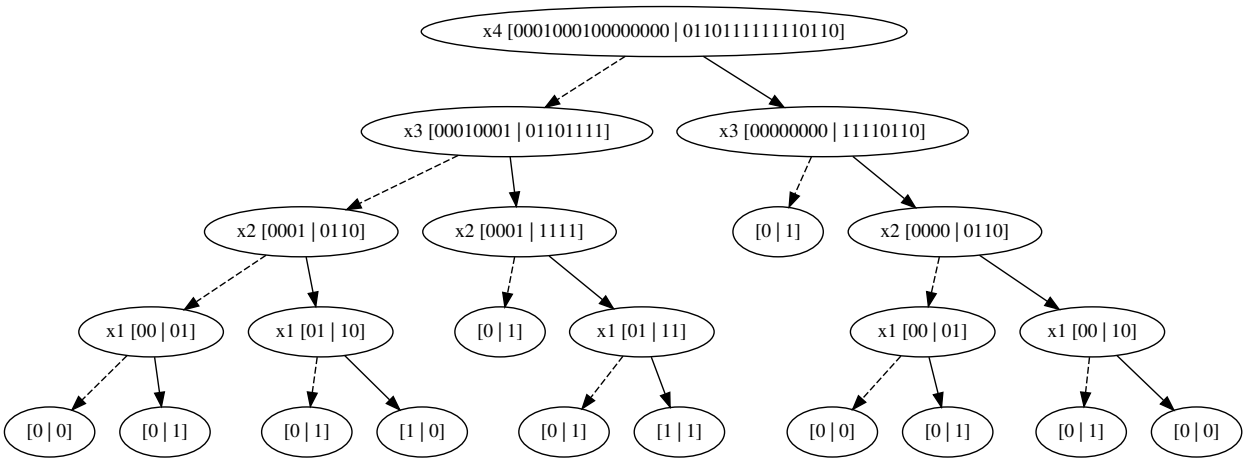

Figure 20: ROBDD résultant de la fusion des ROBDD des nombres 136 et 28662

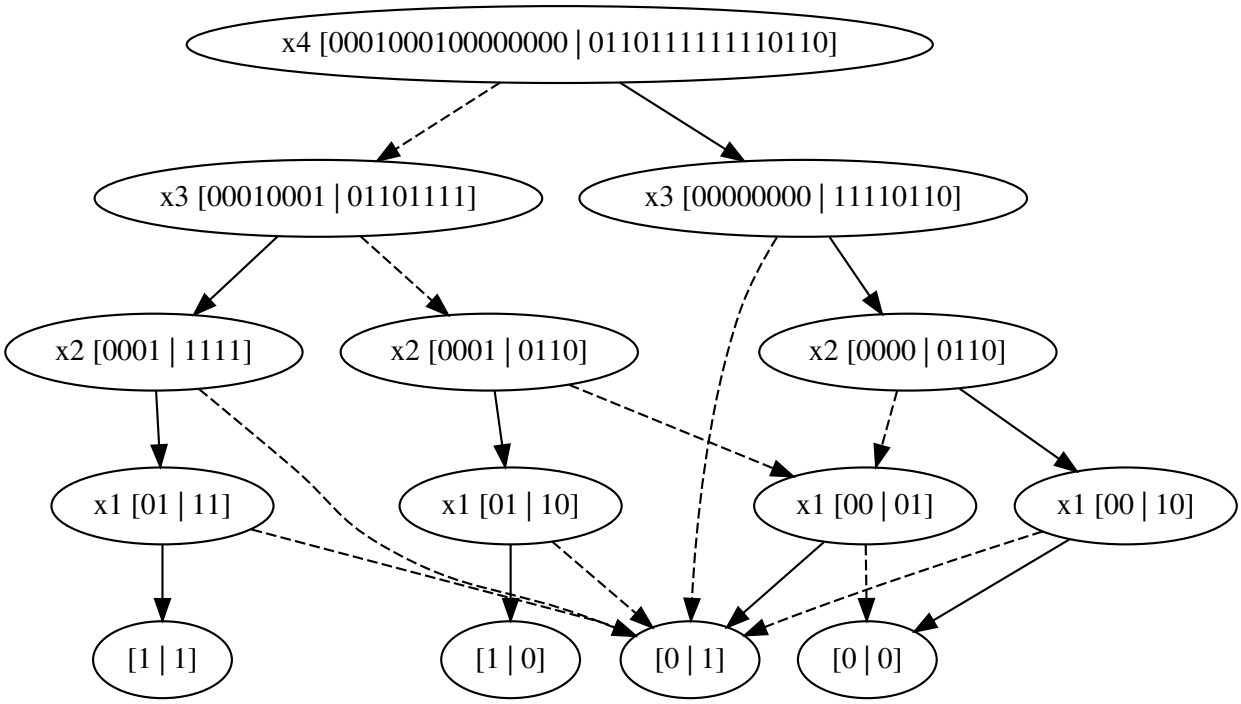

Figure 21: ROBDD de la figure ci-dessus compressé

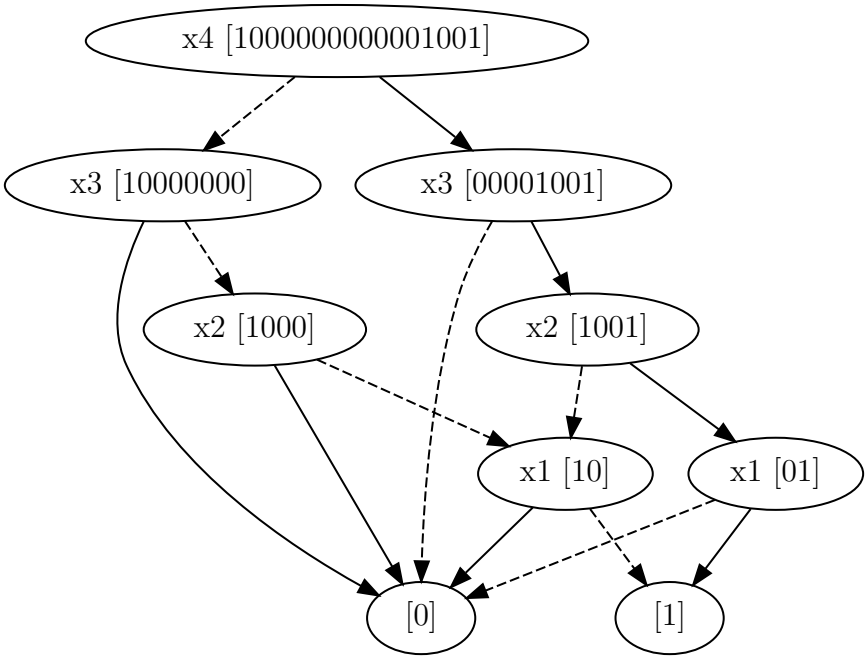

Figure 22: Fusion des ROBDD des nombres 136 et 28662 avec la table de vérité de l'opérateur *NOR* ("1000")

Notre résultat semble cohérent car:

$$00010001\ NOR\ 0110111111110110 = 1000000000001001$$