Sorbonne Université

Faculté des Sciences et Ingénierie

Parcours Science et Technologie du Logiciel
(STL)

Rapport de projet : Génération de diagrammes de décision binaires

Auteurs

Abdelkader Boumessaoud
Zaky Abdellaoui

Encadrant

Antoine Genitrini

Table des matières

1 Présentation

Le but de ce projet est de générer des diagrammes de décision binaires, réduits et ordonnés avec une approche analogue a celle présentée dans l'article de Newton et Verna, "A Theoretical and Numerical Analysis of the Worst-Case Size of Reduced Ordered Binary Decision Diagrams", et une comparaison de nos résultats à leurs expérimentations.

2 Échauffement

2.1 Question 1.1

Java permet la manipulation de nombres de taille arbitraire, donc il y à possibilité d'aller au-delà des limites des types numériques primitifs, sans perte de précision (ou bien une perte contrôlé).

Le paquet `java.math` contient deux classes pour représenter deux types de nombres de taille arbitraire :

— La classe `BigInteger` représente un nombre entier de taille arbitraire

— La classe `BigDecimal` représente un nombre réel de taille arbitraire

Malgré cela, des contraintes système sont toujours présentes :

— Contrainte d'espace : même si avec les machines modernes la taille limite d'un nombre en mémoire peut sembler haute, il ne faut pas oublier de multiplier par le nombre d'instances utilisées à un instant donné pour se rendre compte de la quantité de mémoire utilisée.

— Contrainte de temps : surtout dans le cas de nombres à très grande précision. L'addition et la soustraction s'effectuent en temps linéaire, mais pour la multiplication, la division, le calcul de racine ou de puissance nécessitent de confronter chaque chiffre du nombre à chaque chiffre de l'autre. Le temps de calcul est alors quadratique ou exponentiel.

2.2 Question 1.2

Grâce à la classe `java.lang.Math` nous avons à notre disposition la méthode `toString(int n)`, qui permet de convertir un `BigInteger` en une chaîne de caractères correspondant à sa représentation en base n (le cas échéant, $n = 2$). On convertit alors bit par bit la valeur retournée en valeurs booléennes et on les retourne sous forme de liste.

Sans utiliser cette classe, on aurait pu également passer par une division euclidienne.

Figure 1 – Test de la décomposition

```java
boolean[] binary_flags_test = echauffement.Fonctions_echauffement.decomposition(BigInteger.valueOf(38));
boolean[] binary_flags_expected = {false, true, true, false, false, true};
if(Arrays.equals(binary_flags_test, binary_flags_expected))
    System.out.println("fonction [decomposition] --> \033[32mWORKING\u001B[0m");
else
        System.out.println("fonction [decomposition] --> \033[31mNOT-WORKING");
```

Figure 2 – Vérification de la décomposition

```
fonction [decomposition] --> WORKING
```

2.3 Question 1.3

En prenant en entrée la liste résultante de la fonction précédente, ainsi qu'une taille donnée, on rogne la liste précédente pour en générer une nouvelle :

— Si la taille donnée est plus petite, on supprime les valeurs en trop
— Sinon on complète avec des `false`

FIGURE 3 – Test de la complétion

```java
boolean[] t = {false, true, true, false, false, true};
boolean[] completed_list_test1 = echauffement.Fonctions_echauffement.completion(t, 4);
boolean[] completed_list_test2 = echauffement.Fonctions_echauffement.completion(t, 8);
boolean[] completed_list_expected1 = {false, true, true, false};
boolean[] completed_list_expected2 = {false, true, true, false, false, true, false, false};
if((Arrays.equals(completed_list_test1, completed_list_expected1)) && (Arrays.equals(completed_list_test2, completed_list_expected2)))
    System.out.println("fonction [completion] --> \033[32mWORKING\u001B[0m");
else
        System.out.println("fonction [completion] --> \033[31mNOT-WORKING\u001B[0m");
```

FIGURE 4 – Vérification de la complétion

```
fonction [completion] --> WORKING
```

2.4 Question 1.4

Pour finir, on génère la table de vérité en combinant les deux fonctions précédentes.

FIGURE 5 – Test de la table de vérité

```java
boolean[] table_de_verite_test = echauffement.Fonctions_echauffement.table(BigInteger.valueOf(38),8);
boolean[] table_de_verite_expected = {false, true, true, false, false, true, false, false};
if((Arrays.equals(table_de_verite_test, table_de_verite_expected)))
        System.out.println("fonction [table] --> \033[32mWORKING\u001B[0m");
else
        System.out.println("fonction [table] --> \033[31mNOT-WORKING\u001B[0m");
```

FIGURE 6 – Vérification de la table de vérité

```
fonction [table] --> WORKING
```

3 Arbre de décision et compression

3.1 Question 2.5

La structure d'un nœud de l'arbre est composé de 3 champs : 2 fils (gauche/droit) qui référencent un nœud et le contenu du nœud, de type `String`. La structure de l'arbre de décision est composée d'un nœud racine, qui est le point de départ de l'arbre. Enfin, on considère les feuilles de l'arbre comme de simple nœuds internes avec des fils de valeur `NULL`.

Figure 7 – Structures de données

```java
public static class Node
{
    String content;
    Node left;
    Node right;
}

public static class BinaryDecisionTree
{
    Node root;
}
```

3.2 Question 2.6

Pour ce qui est de la construction de l'arbre, on a comme entrée une table de vérité de taille 2^h, avec h la hauteur de l'arbre.
On récupère cette dernière avec la fonction `getExposantOfPowBaseTwo(tableDeVerite.length)`.

On boucle sur la valeur h en fractionnant la table de vérité pour la distribuer aux deux fils à chaque itération jusqu'à arriver aux feuilles, c'est à dire arriver à $h = 0$.

Figure 8 – Test de la construction de l'arbre

```java
boolean[] table_de_verite = echauffement.Fonctions_echauffement.table(BigInteger.
    valueOf(38),8);
Fonctions_arbre_de_decision_et_compression.BinaryDecisionTree BDT =
    Fonctions_arbre_de_decision_et_compression.cons_arbre(table_de_verite);
System.out.println("fonction [cons_arbre] --> \033[93mCONFIRMATION-VISUELLE\u001B
    [0m :");
treePrint("", BDT.root, false);
```

Figure 9 – Vérification de la construction de l'arbre

```
fonction [cons_arbre] --> CONFIRMATION-VISUELLE :
                                                        |-- false
                                        |-- x1
                                                        |-- false
                        |-- x2
                                                        |-- true
                                        |-- x1
                                                        |-- false
|-- x3
                                                        |-- false
                                        |-- x1
                                                        |-- true
                        |-- x2
                                                        |-- true
                                        |-- x1
                                                        |-- false
```

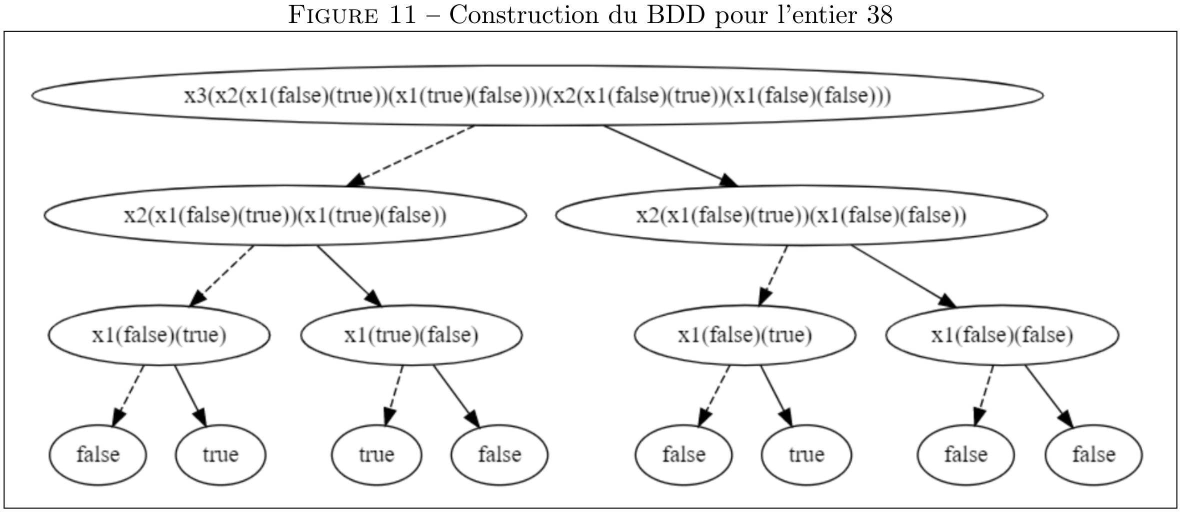
3.3 Question 2.7

Pour l'association de chaque nœud a son mot de Lukasiewicz, nous avons choisi l'implémentation la plus simple possible : explorer l'arbre de la racine aux feuilles en insérant dans les champs contenus de manière récursive.

Figure 10 – Calcul des mots de Lukasiewicz

```java
public static void lukaOnNode(Node root)
{
    if (root.left != null)
    {
    lukaOnNode(root.left);
        lukaOnNode(root.right);
        root.setContent(root.getContent()+"("+root.left.getContent()+")("+root.
            right.getContent()+")");
    }
}
public static void luka(BinaryDecisionTree tree)
{
    lukaOnNode(tree.root);
}
```

Figure 11 – Construction du BDD pour l'entier 38



3.4 Question 2.8

Pour générer l'arbre compressé, on crée une liste de références (pointeurs) vers des mots uniques pour ensuite ré-explorer l'arbre et remplacer les références multiples de mots uniques par celles de notre liste, obtenant de ce fait un arbre compressé.

FIGURE 12 – Création de la liste de références

```java
    public static Node getPointersFromPointersList (List<Node> theList, String
        content)
    {
        for (int i=0; i<theList.size(); i++)
            if (theList.get(i).getContent().equals(content)) return theList.get(i)
                ;
        return null;
    }

    public static boolean existInPointersList(List<Node> theList, Node root)
    {
        boolean result = false;
        for (int i=0; i<theList.size(); i++) {
            if (theList.get(i).getContent().equals(root.getContent())) {result =
                true;}}
        return result;
    }

    public static void createListOfPointers(Node root, List<Node> theList)
    {
        if (root != null)
        {
            createListOfPointers(root.left, theList);
            if (!existInPointersList(theList, root))
                theList.add(root);
            createListOfPointers(root.right, theList);
            if (!existInPointersList(theList, root))
                theList.add(root);
        }
    }
```

FIGURE 13 – Vérification des remplacements

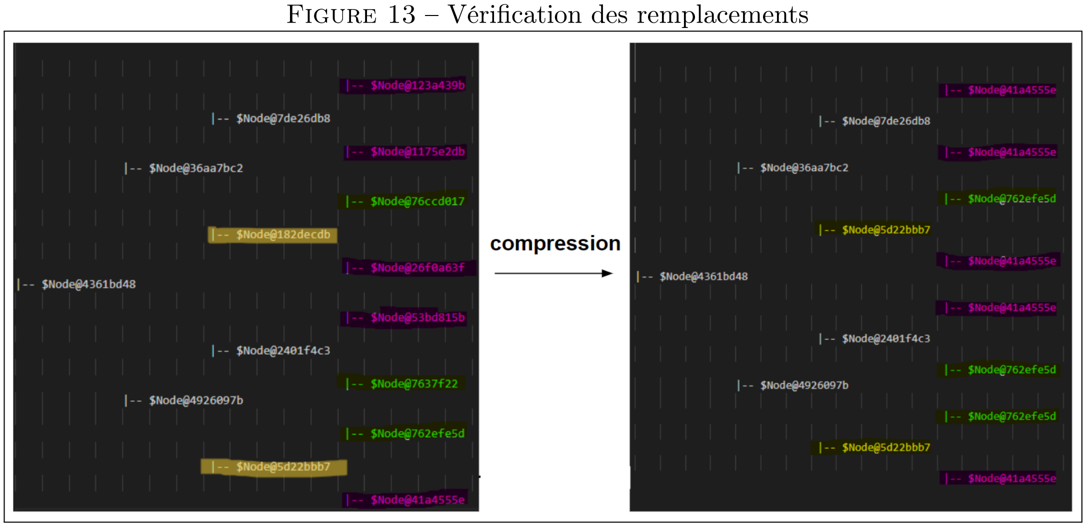
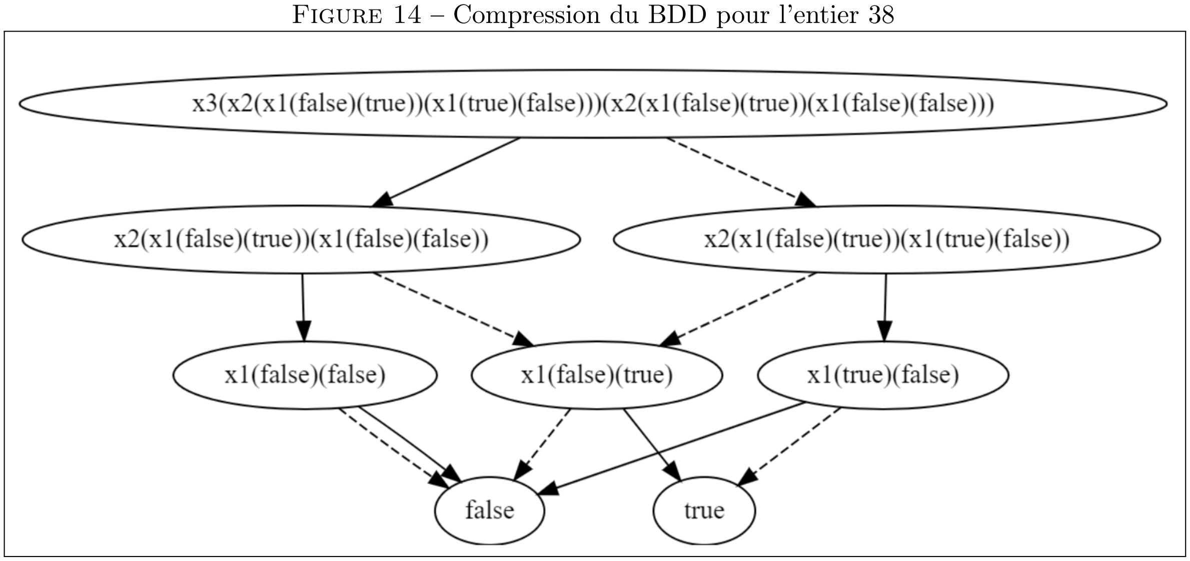


Figure 14 – Compression du BDD pour l'entier 38



3.5 Question 2.9

La génération du graphe au format `.dot` se fait en deux étapes : d'abord, on crée les nœuds avec la méthode `graph.addNode`, puis on crée des liens entre les nœuds avec la méthode `graph.link`, qui a pour arguments le nœud père et le nœud fils, ainsi qu'un booléen précisant s'il s'agit d'un fils gauche ou pas (utile lors de la création des liens en pointillés).

FIGURE 15 – Génération du fichier `.dot`

```
        if(tree.left !=null) {
        if (h != 1) {
            if (!graph.exists(tree.left.toString().substring(81)))
                graph.addNode("" + tree.left.toString().substring(81), /*("x"+(h-1)) +*/ tree.left.content);
            graph.link("" + tree.toString().substring(81), "" + tree.left.toString().substring(81), true);
            if (!graph.exists(tree.right.toString().substring(81)))
                graph.addNode("" + tree.right.toString().substring(81), /*("x"+(h-1)) +*/ tree.right.content);
            graph.link("" + tree.toString().substring(81), "" + tree.right.toString().substring(81), false);
            dot(tree.left, h - 1, graph);
            dot(tree.right, h - 1, graph);
        } else {
            if (!graph.exists(tree.left.toString().substring(81)))
                graph.addNode("" + tree.left.toString().substring(81), tree.left.content);
            graph.link("" + tree.toString().substring(81), "" + tree.left.toString().substring(81), true);
            if (!graph.exists(tree.right.toString().substring(81)))
                graph.addNode("" + tree.right.toString().substring(81), tree.right.content);
            graph.link("" + tree.toString().substring(81), "" + tree.right.toString().substring(81), false);
        }
    }
```

4 Arbre de décision et ROBDD

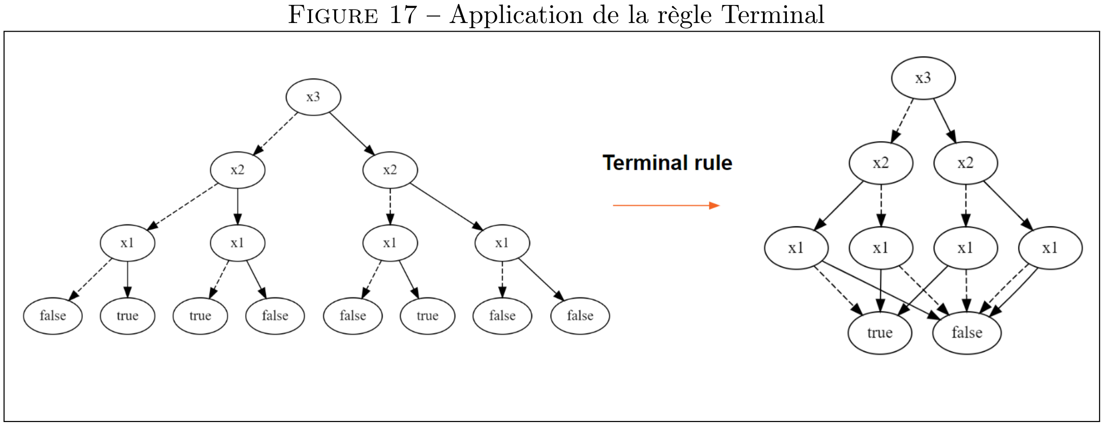
4.1 Question 3.10

4.1.1 Question 3.10.1 : Règle Terminal

FIGURE 16 – Code de la règle Terminal

```
        if (node != null) {
            Fonctions_arbre_de_decision_et_compression.Node unique_true =
                Fonctions_arbre_de_decision_et_compression.
                getPointersFromPointersList(theList, "true");
            Fonctions_arbre_de_decision_et_compression.Node unique_false =
                Fonctions_arbre_de_decision_et_compression.
                getPointersFromPointersList(theList, "false");
            if (node.getLeft().getContent().equals("true")) {
                node.setLeft(unique_true);
            } else if (node.getLeft().getContent().equals("false")) {
                node.setLeft(unique_false);
            } else {
                terminal(node.getLeft(), theList);
            }
            if (node.getRight().getContent().equals("true")) {
                node.setRight(unique_true);
            } else if (node.getRight().getContent().equals("false")) {
                node.setRight(unique_false);
            } else {
                terminal(node.getRight(), theList);
            }
        }
```

FIGURE 17 – Application de la règle Terminal



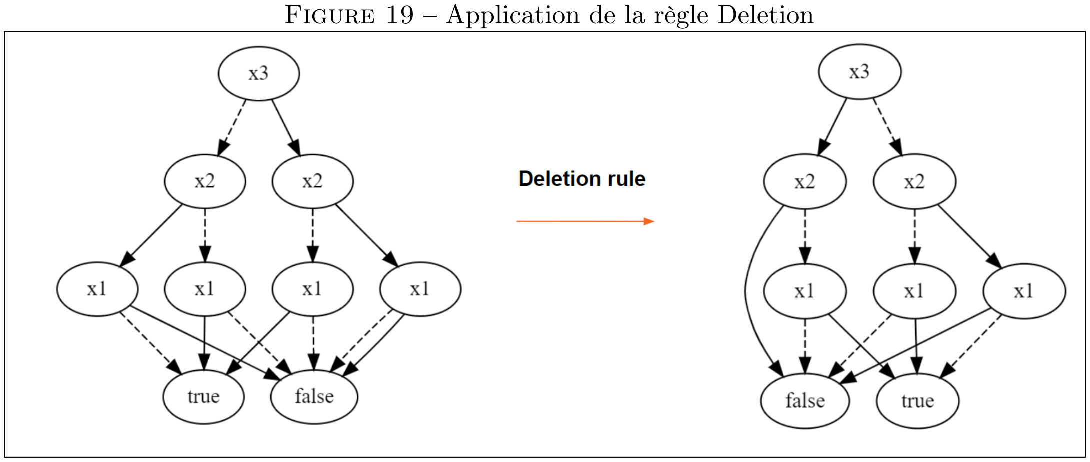
4.1.2 Question 3.10.1 : Règle Deletion

FIGURE 18 – Code de la règle Deletion

```
static void deletion_node(Fonctions_arbre_de_decision_et_compression.
    BinaryDecisionTree tree, Fonctions_arbre_de_decision_et_compression.Node
    node) {
    Fonctions_arbre_de_decision_et_compression.Node treeRoot = tree.getRoot();
    Fonctions_arbre_de_decision_et_compression.Node currNode = node;
    if (treeRoot.equals(currNode)) {
        if (treeRoot.getLeft() != null) {
            if (treeRoot.getLeft().equals(treeRoot.getRight())) {
                currNode = treeRoot.getLeft();
                tree.setRoot(treeRoot.getLeft());
            }
            //deletion_node(tree,currNode);
        }
    }
    if (currNode != null && currNode.getLeft() != null) {
        deletion_node(tree, currNode.getLeft());
        if (currNode.getLeft().getLeft() != null && currNode.getLeft().getLeft
            ().equals(currNode.getLeft().getRight())) {
            currNode.setLeft(currNode.getLeft().getLeft());
        }
        deletion_node(tree, currNode.getRight());
        if (currNode.getRight().getLeft() != null && currNode.getRight().
            getLeft().equals(currNode.getRight().getRight())) {
            currNode.setRight(currNode.getRight().getLeft());
        }
    }
}

static void deletion(Fonctions_arbre_de_decision_et_compression.
    BinaryDecisionTree tree) {
    Fonctions_arbre_de_decision_et_compression.Node treeRoot = tree.getRoot();
    deletion_node(tree, treeRoot);
}
```

FIGURE 19 – Application de la règle Deletion

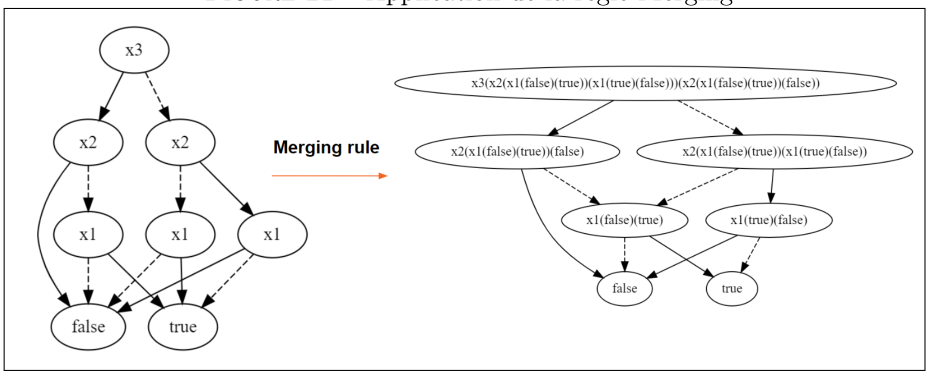


4.1.3 Question 3.10.1 : Règle Merging

Figure 20 – Code de la règle Merging

```
        Fonctions_arbre_de_decision_et_compression.compressionOnNode(node.getLeft
            (), node, theList);
        Fonctions_arbre_de_decision_et_compression.compressionOnNode(node.getRight
            (), node, theList);
```

Figure 21 – Application de la règle Merging



4.2 Question 3.11

4.2.1 Question 3.11.1 : Nœuds Internes

Initialisation :
$2^0 - 1 = 1 - 1 = 0$
$h = 0$, nombre de nœuds internes $= 0$.

Hérédité :
La propriété est vraie pour une h. Montrons qu'elle est vraie pour $h + 1$.

Un arbre de hauteur $h + 1$ est obtenu en créant une nouvelle racine a laquelle on insérer en fils gauche et droit des arbres de hauteur h. On obtient alors $2^h - 1$ nœuds pour chacun des deux arbres fils de hauteur h, plus la nouvelle racine.

Cela donne finalement : $2(2^h - 1) + 1 = 2^{h+1} - 2 + 1 = 2^{h+1} - 1$ nœuds internes.

4.2.2 Question 3.11.2 : Feuilles

Initialisation :
$2^0 = 1$
h $= 0$, nombre de feuilles $= 1$.

Hérédité :
La propriété est vraie pour h. Montrons qu'elle est vraie pour h $+ 1$.

Un arbre de hauteur $h + 1$ est obtenu en créant une nouvelle racine à laquelle on insère en fils gauche et droit des arbres de hauteur h, ce qui donne 2^h feuilles pour un arbre de hauteur h. On a donc : $2\left(2^h\right) = 2^{h+1}$ feuilles pour un arbre de hauteur $h + 1$.

4.2.3 Question 3.11.3 : Total

$$\begin{aligned}
&= \text{Nombre de nœuds internes} + \text{Nombre de feuilles} \\
&= \left(2^h - 1\right) + \left(2^h\right) \\
&= 2(2^h) - 1 && = 2^{h+1} - 1
\end{aligned}$$

Pour un mot de Lukasiewicz composé de trois éléments :
Parenthèses :
Ouvrante et fermante pour chaque nœud, ce qui donne :
$2 \times (\text{nombre de nœuds} - \text{la racine}) = 2\left(2(2^h) - 2\right) = 4(2^h) - 4$ caractères.

Variables :
Les variables sont sous la forme : $x +$ l'indice de leur hauteur dans l'arbre. Le nombre de variables est la hauteur de l'arbre, en la majorant cette hauteur par 9 (soit $\cdot ch = 1$), on peut en conclure que la longueur du nombre est 1. Cela donne :
$2 \times$ nombre de nœuds internes $= 2(2^h - 1) = 2(2^h) - 2$ caractères.

Feuilles :
Elles peuvent valoir `true` ou `false`, donc 4 ou 5 caractères. En prenant le pire cas de 5 caractères, on obtient :
$5 \times$ nombre de feuilles $= 5(2^h)$ caracteres.
La longueur d'un mot de Lukasiewicz à la racine d'un arbre de hauteur h est donc majorée par

$$\begin{aligned} &4(2^h) - 4 + 2(2^h) - 2 + 5(2^h) \\ &= 11(2^h) - 6 \end{aligned}$$

4.3 Question 3.12

Considérons une méthode naïve de compression : chercher dans l'arbre d'autres nœuds avec le même mot de Lukasiewicz.

À chaque étage de l'arbre, nous comparons chaque nœud à ses frères vu que c'est le seul cas ou ils peuvent avoir la même longueur et donc le même mot. Pour chaque nœud de l'étage, on fera donc 2^h comparaisons. Puis, pour ce qui est de la taille en caractère des deux nœuds a comparer, on prendra la valeur précédemment calculée : $11 \cdot 2^h - 6$.

Longueur du mot a l'étage h actuel comme nombre de comparaison (pire des cas).

$$\begin{aligned} &= \sum_{h=0}^{H} 2^h \left(11 \cdot 2^h - 6\right) \\ &\approx 11H \sum_{h=0}^{H} 2^h \cdot 2^h \\ &= 11H \sum_{h=0}^{H} 2^{2h} \\ &= 11H \left(2^{2H+1} - 1\right) \\ &\approx 11H \cdot 2^{2H} \end{aligned}$$

Donc la complexité au pire cas de l'algorithme de compression est en $\mathcal{O}(2^{2\mathrm{h}})$.

4.4 Question 3.13

Il a été calculé précédemment que le nombre total de nœuds d'un arbre de décision de hauteur h est $2\left(2^h\right) - 1$. en calculant s fonction inversé pour trouver h on obtient :

$$\begin{aligned} h &= \frac{\ln(n+1)}{\ln(2)} - 1 \\ &\approx \log_2\left(\frac{n+1}{2}\right) \end{aligned}$$

ce qui donne une complexité au pire cas de :

$$
\begin{gathered}
11\left(\log_2 \frac{n+1}{2}\right) \cdot 2^{2\log_2\left(\frac{n+1}{2}\right)} \\
11\left(\log_2 \frac{n+1}{2}\right) \cdot \left(\frac{n+1}{2}\right)
\end{gathered}
$$

Et donc en $\mathcal{O}(\mathrm{n}\log\mathrm{n})$.

5 Étude expérimentale

5.1 Question 4.14

On reproduit les courbes de l'article en créant tous les ROBDD possibles pour un certain nombre de variables booléennes et en mesurant leur taille (en nombre de noeuds) de ces ROBDD. On crée ensuite une courbe histogramme qui montre la distribution des ROBDD en fonction de leur taille.

La reproduction des courbes pour $1 \leq n \leq 4$ se fait avec facilité, et les calculs sont effectués dans un temps raisonnable. Lorsqu'on tente d'utiliser le même code pour calculer la distribution des tailles de ROBDD pour $n = 5$, le programme semble mettre un temps très long.

On se rend compte en analysant le code que non seulement le temps d'exécution pour un unique ROBDD est en complexité exponentielle à cause des parcours intégraux des arbres binaires ($\mathcal{O}(2^n)$ le cas échéant), mais que la quantité de données à traiter, à savoir le nombre de BDD pour chaque valeur de n, est égale à 2^{2^n}.

En se basant sur des données expérimentales pour le temps de calcul d'un seul ROBDD à 5 variables, l'histogramme pour $n = 5$ nécessiterait un temps de calcul d'environ 2 jours, 11 heures et 40 minutes, ce qui est très conséquent. Pour $n = 6$, on aurait un temps de l'ordre de 91 millions d'années, et pour $n = 7$, on dépasserait le cube de l'âge de l'univers (!!!) Cette complexité impraticable justifie, comme dans l'article, l'extrapolation en utilisant des échantillons aléatoires pour $n \geq 5$.

5.2 Question 4.15

On utilise donc le même procédé, mais avec un nombre fini et "raisonnable" (en termes de temps de calcul) de ROBDD tirés uniformément et de manière aléatoire.

Figure 22 – Histogrammes estimés des distributions de taille des ROBDD pour $n = 5$ et $n = 6$

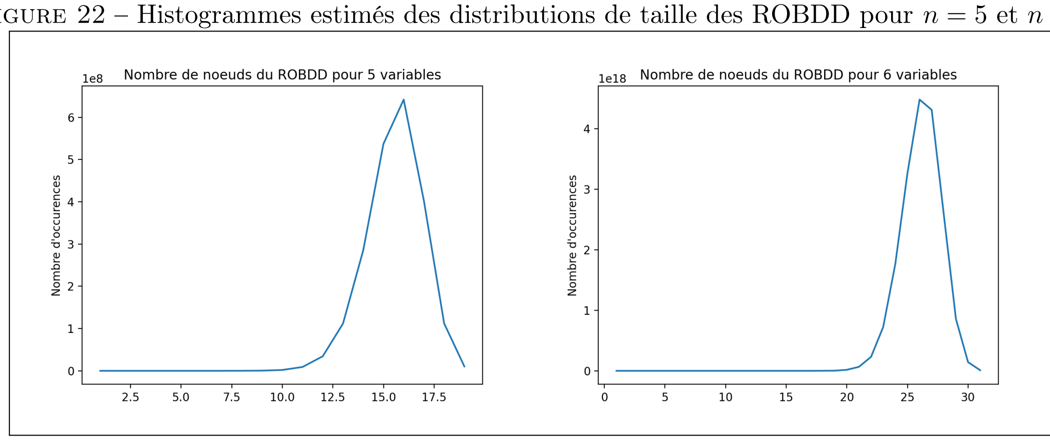
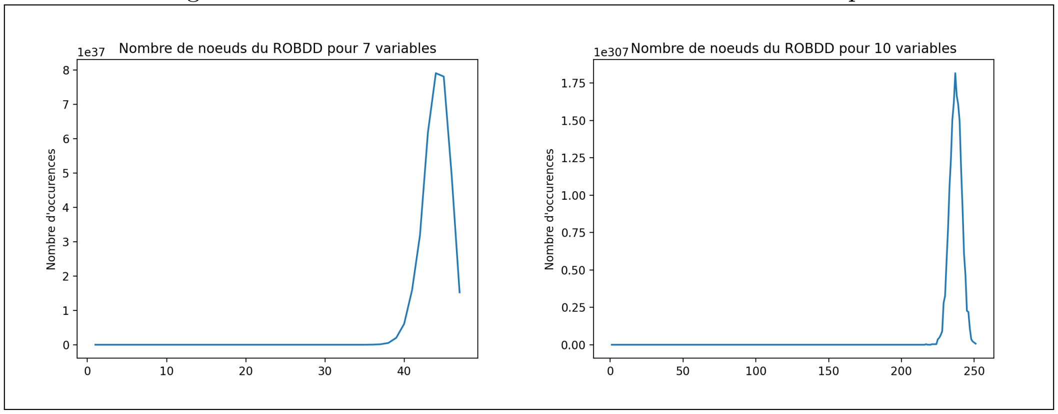


Figure 23 – Histogrammes estimés des distributions de taille des ROBDD pour $n = 7$ et $n = 10$



5.3 Question 4.16

Figure 24 – Nombre d'échantillons et temps de calcul pour générer les histogrammes

```
n = 5;  150000 samples; 12 unique sizes;    Total time = 8.7669419 s;    0.058446279333333344 ms per ROBDD.
n = 6;  100000 samples; 15 unique sizes;    Total time = 15.6411577 s;   0.15641157700000002 ms per ROBDD.
n = 7;  50000 samples;  13 unique sizes;    Total time = 25.2219832 s;   0.504439664 ms per ROBDD.
n = 8;  20000 samples;  14 unique sizes;    Total time = 33.618165 s;    1.68090825 ms per ROBDD.
n = 9;  5000 samples;   20 unique sizes;    Total time = 31.6274509 s;   6.32549018 ms per ROBDD.
n = 10; 3000 samples;   29 unique sizes;    Total time = 76.6204587 s;   25.540152900000002 ms per ROBDD.
```