Sorbonne Université

Rapport Final

16 Décembre 2022

Devoir de Programmation - Algorithmique Avancée

Auteurs :
Daniel Sima
Yukai Luo

Encadrants :
Antoine Genitrini

Table des matières

Chapitre 1

Présentation

Ce devoir de programmation nous confronte a différents diagrammes de décision binaire en passant par l'arbre de décision construit avec une table de vérité, a sa transformation aux noeuds avec les mots de Lukasiewicz et la compression suivant différentes règles présentés dans un article scientifique. En dernière place, une étude expérimentale similaire a celle de l'article en question est effectué pour comparer les différents résultats obtenus et la fusion des diagrammes de décisions binaire, plus exactement de deux ROBDD, est mise en place.

Question 1.1

Le langage choisit est Python, qui ne s'avère pas être le meilleur pour la récursion et est assez lent, mais ce dernier nous permet de manipuler directement des entiers de taille arbitraire a travers l'import de la bibliothèque **numpy**.

Question 1.2

Algorithm 1 Pseudo-code Decomposition

Entrée(s) entier x
  LR prend liste vide
  //En commençant par les bits de poids fort
  pour tout bit i de l'entier x en binaire **faire**
    si i vaut 1 **alors**
      ajouter $True$ au début de LR
    sinon
      ajouter $False$ au début de LR
    fin du si
  fin du pour
Sortie(s) LR

Voir le code de **decomposition(x)** dans le fichier *main.py*.
Pour tester, lancer dans le terminal *python main.py*, parmi les affichages de la partie I un nombre sera tiré aléatoirement et sa décomposition avec la fonction sera faite.

Question 1.3

Algorithm 2 Pseudo-code Completion

Entrée(s) liste L, entier n
  si $n < L$ **alors**
    tronquer L du début jusqu'à l'indice n
  sinon
    remplir a la fin de L la différence entre n et taille de L avec des $False$
  fin du si
Sortie(s) L

Voir le code de **completion(Liste, n)** dans le fichier *main.py*.
Pour tester, lancer dans le terminal *python main.py*, parmi les affichages de la partie I la complétion de la décomposition du nombre tiré aléatoirement sera faite.

Question 1.4

Il suffit comme mentionné d'appeler la fonction complétion avec comme arguments :
- decomposition(x) pour la Liste.
- n pour l'entier qui définit la taille.

Voir le code de **table(x, n)** dans le fichier *main.py*.
Pour tester, lancer dans le terminal *python main.py*, parmi les affichages de la partie I la table de vérité (appel a la fonction table) sera faite avec la complétion de la décomposition du nombre tiré aléatoirement et avec une taille n = 5.
Pour tester d'autres combinaisons de tests aller a la fin du fichier main.py et chercher dans la partie Tests, la partie I ou vous pourrez changer les valeurs de nombre et n et voir le résultant dans le terminal.

Chapitre 2

Arbre de décision et compression

Question 2.5

Nous avons choisit comme structure de donnée un arbre binaire *ArbreBinaire*, défini dans la classe *ArbresBinaires.py*.

```
def __init__(self, valeur, id) :
    self.valeur = valeur # Étiquette du noeud
    self.enfant_gauche = None # Enfant gauche du noeud
    self.enfant_droit = None # Enfant droit du noeud
    self.id = id # Identifiant pour un affichage sans doublons
```

FIGURE 2.1 – Champs de la structure de donnée utilisé

Question 2.6

Algorithm 3 Pseudo-code Construction Arbre de Décision

Entrée(s) table de vérité T
h prend log_2(taille T) // Hauteur de l'arbre
création de *root*, la racine de l'arbre de décision avec comme étiquette "x"+h
pos_tab prend 2^h, la position la plus a gauche d'une feuille d'un noeud
res prend $build(h, T, root, pos\_tab)$ // Création de l'arbre récursivement
Sortie(s) *res*

Algorithm 4 Pseudo-code Build

Entrée(s) hauteur h, T, arbre courant *tree*, position extrémité droite la feuille *pos_tab*
si $h = 1$ **alors**
l'arbre courant *tree* crée un fils gauche avec comme étiquette l'indice $(pos\_tab - 2)$ de T et un fils droit avec comme étiquette l'indice $(pos\_tab - 1)$ de T
sinon
l'arbre courant *tree* crée un fils gauche avec comme étiquette "x" + $(h - 1)$ et appelle récursivement Build($(h - 1)$, T, fils gauche de *tree*, $pos\_tab - 2^{h-1}$
l'arbre courant *tree* crée un fils droit avec comme étiquette "x" + $(h - 1)$ et appelle récursivement Build($(h - 1)$, T, fils droit de *tree*, *pos_tab*
fin du si
Sortie(s) *tree*

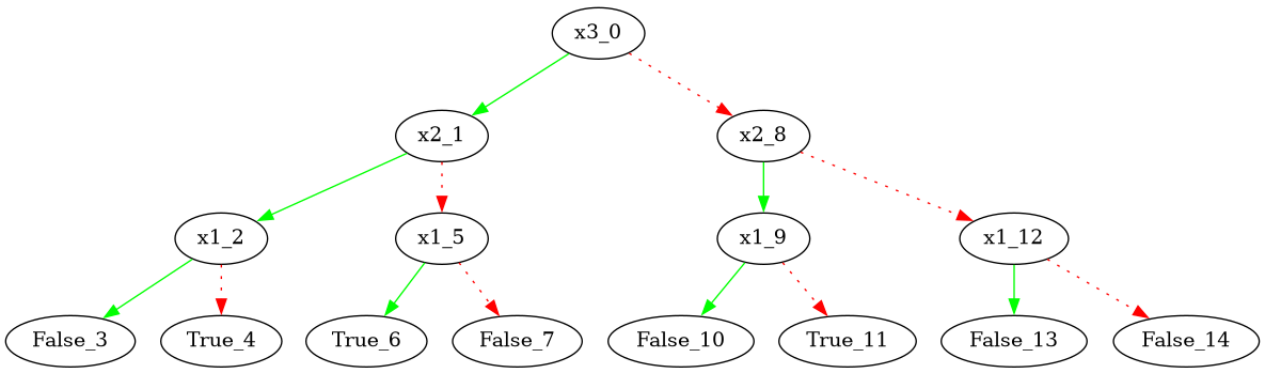


FIGURE 2.2 – Arbre de décision issu de la table de vérité de taille 8 construite sur l'entier 38

Voir le code de **cons_arbre(table)** et **build(h, table, tree, pos_tab)** dans le fichier *main.py*. Pour tester, lancer dans le terminal *python main.py*, parmi les affichages de la partie II il est indiqué que l'arbre de décision de la table de vérité de taille 8 avec l'entier 38 à été construit dans le dossier *./Graphes/graphe.png*.
Pour tester d'autres valeurs, aller a la fin du fichier *main.py* et chercher dans la partie Tests, la partie II ou vous pourrez changer les valeurs de *nombre* et *taille* et voir le résultant dans le même dossier que précédemment.

Question 2.7

Algorithm 5 Pseudo-code Luka

Entrée(s) arbre courant *tree*, le mot Lukasiewicz *text* // *text* au début vide
  si *tree* a un enfant gauche **alors**
    former le mot Lukasiewicz de l'enfant gauche en appelant $luka(enfant\_gauche, text)$
    text prend la concaténation de l'étiquette de l'arbre *tree* avec le mot *text* de l'enfant gauche
    si *tree* a un enfant droit **alors**
      former le mot Lukasiewicz de l'enfant droit en appelant $luka(enfant\_droit, text)$
      text prend la concaténation du *text* courant avec le mot *text* de l'enfant droit
      mettre le mot *text* courant dans l'étiquette de l'arbre *tree* courant
    fin du si
  sinon
    text prend l'étiquette de l'arbre courant *tree*
  fin du si
Sortie(s) *text*

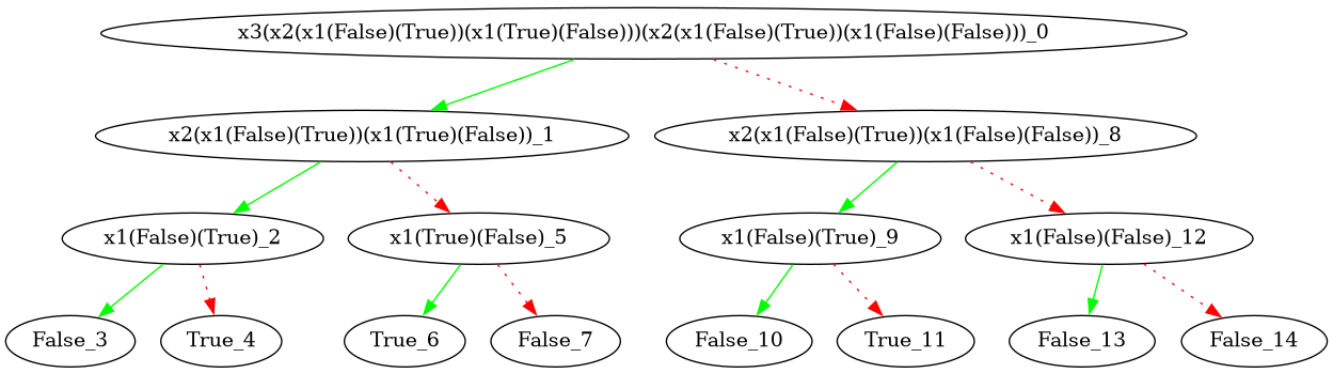


FIGURE 2.3 – Arbre de décision avec les mot de Lukasiewicz mis pour chaque noeud de la figure précédente

Voir le code de **luka(tree, text)** dans le fichier *main.py*.
Pour tester, lancer dans le terminal *python main.py*, parmi les affichages de la partie II il est indiqué que l'arbre de décision de la Figure 2.2 avec le mot de Lukasiewicz mis pour chaque noeud a été construit dans le dossier *./Graphes/graphe2.png*.
Cet arbre a les valeurs (nombre et taille) précédentes, pour d'autres combinaisons de tests, changer ces valeurs.

Question 2.8

Pour cette question, notre algorithme de *compression* a appliqué les règles Terminal Rule et Merging Rule mentionné dans l'article. Pour stocker des noeud visités, on implémente une dictionnaire *dict* dont clé est le étiquette du noeud et la valeur est le noeud lui-même.
Le dictionnaire étant une méthode de hachage, au vu de l'avancement du cours, nous aurions du implémenter un simple tableau .

$$\{\textit{étiquette du noeud} : \textit{noeud}\}$$

FIGURE 2.4 – Dictionnaire implémenté dans l'algorithme de compression

Algorithm 6 Pseudo-code Compression

Entrée(s) arbre courant *tree*, le dictionnaire *dict*
  // enfant gauche et/ou droit
  si $enfant != null$ **alors**
    si !$enfant$ n'est pas dans *dict* **alors**
      $dict[\textit{étiquette de l'}enfant] \leftarrow enfant$
      $compression(enfant, dict)$
    sinon
      $enfant \leftarrow dict[\textit{étiquette de l'}enfant]$
    fin du si
  fin du si

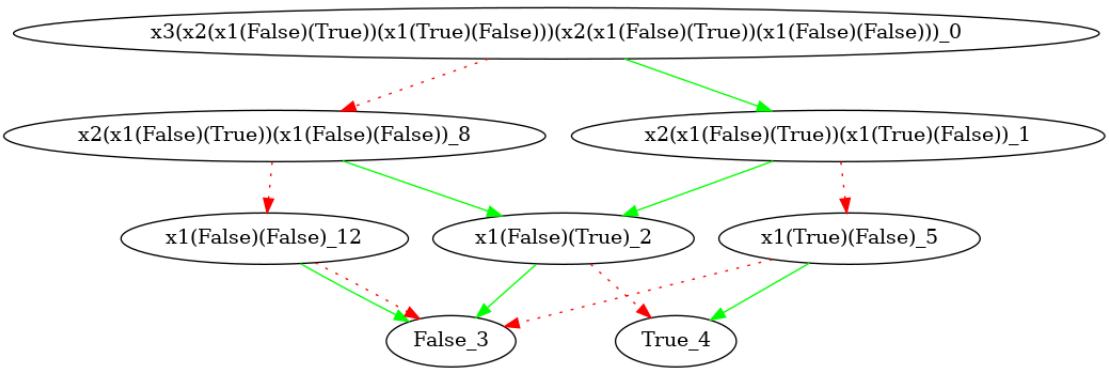


FIGURE 2.5 – Arbre de décision compressé de la figure précédente

Voir le code de **luka_compresse(tree, text)** dans le fichier *main.py*.
Pour tester, lancer dans le terminal *python main.py*, parmi les affichages de la partie II il est indiqué que l'arbre de décision de la Figure 2.3 a été compressé et construit dans le dossier *./Graphes/graphe3.png*.
Cet arbre a les valeurs (nombre et taille) précédents, pour d'autres combinaisons de tests, changer ces valeurs.

Question 2.9

Algorithm 7 Pseudo-code Dot

Entrée(s) étiquette de l'arbre *pere*, étiquette de l'arbre $fils$, entier de direction *orentation*, dictionnaire pour stocker les liens *existence*
ouverture du fichier
si *orentation* est positive **alors**
#c'est un fils gauche
si ce couple de père et fils n'est pas dans *existence* **alors**
écrire le lien dans le fichier "graphe.dot"
stocker ce couple de père et fils dans *existence*
fin du si
sinon
#c'est un fils droit
si ce couple de père et fils n'est pas dans *existence* **alors**
écrire le lien dans le fichier "graphe.dot"
stocker ce couple de père et fils dans *existence*
fin du si
fin du si
fermeture du fichier

Algorithm 8 Pseudo-code Parcours_dot

Entrée(s) arbre courant *tree*, dictionnaire des liens stockés *existence*

si *tree* existe **alors**
  si *tree* a un $enfant\_gauche$ **alors**
    écrire les liens dans *graphe.dot* en appelant *dot*(étiquette_id, (étiquette d'enfant_gauche)_(id d'enfant_gauche), 1, existence)
    parcours_dot($enfant\_gauche$, existence)
  fin du si
  si *tree* a un $enfant\_droit$ **alors**
    écrire les liens dans *graphe.dot* en appelant *dot*(étiquette_id, (étiquette d'enfant_droite)_(id d'enfant_droite), -1, existence)
    parcours_dot($enfant\_droite$, existence)
  fin du si
fin du si
si *tree* existe et n'a pas d'enfant et *existence* est vide **alors**
  # cas particulier ou il n'y a qu'un seul noeud dans l'arbre
  ouvrir le fichier *graphe.dot*
  écrire l'étiquette de *tree*
  fermer le fichier
fin du si

Voir code de **dot(pere, fils, orientation, existence)** et **parcours_dot(tree, existence)** dans *main.py*.
Le bon fonctionnement de ces fonctions est justifié a travers les affichages précédents se trouvant dans le dossier *./Graphes/* ainsi que les affichages générés pour vos combinaisons de tests.

Chapitre 3

Arbre de décision et ROBDD

Question 3.10

Notre algorithme de *compression_bdd* applique la *Deletion Rule* de l'article.

Algorithm 9 Pseudo-code compression_bdd

Compression_bdd(*arbre*) :
si *arbre* ! = *null* **alors**
  si (*enfant_gauche* ! = *null*) *and* (*enfant_droite* ! = *null*) **alors**
    si *étiquette_enfant_gauche* = *étiquette_enfant_droite* **alors**
      arbre ←enfant_gauche
      compression_bdd(*arbre*)
    sinon
      compression_bdd(*enfant_gauche*)
      compression_bdd(*enfant_droite*)
    fin du si
  fin du si
fin du si

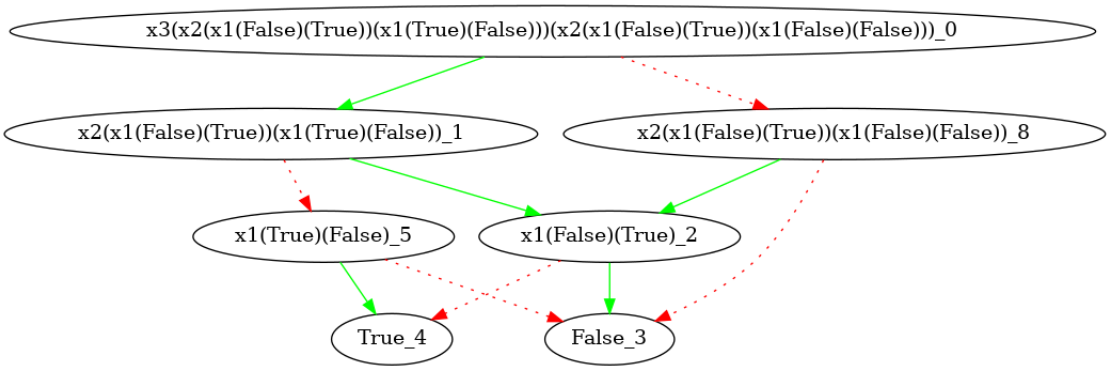


FIGURE 3.1 – ROBDD obtenu par compression de l'arbre en appliquant les 3 règles

Voir le code de **compression_bdd2(tree) :** dans le fichier *main.py*.
Pour tester, lancer dans le terminal *python main.py*, parmi les affichages de la partie III il est indiqué que l'ROBDD obtenu par compression de l'arbre associé à la figure 2.2 en appliquant les 3 règle a été construit dans le dossier *./Graphes/graphe4.png*.

Cet arbre a les valeurs (nombre et taille) précédents, pour d'autres combinaisons de tests, changer ces valeurs.

Question 3.11

Nous allons d'abord montrer qu'il y a 2^h feuilles et $2^h - 1$ noeuds internes dans un arbre par récurrence.

Cas de Base

Pour h = 0, nous avons 1 seule feuille et pas de noeud interne.

Hypothèse de Récurrence

Supposons pour h = n, il y a 2^n feuilles et $2^n - 1$ noeuds internes dans un arbre.

Pour h=n+1, chaque feuille pour le cas h=n va générer 2 nouvelles feuilles. Et les feuilles pour le cas h=n devient les noeuds internes. Donc il y a $2^n - 1 + 2^n = 2*2^n - 1 = 2^{n+1} - 1$ noeuds internes et $2*2^n = 2^{n+1}$ feuilles.

On a montré qu'il y a 2^h feuilles et $2^h - 1$ noeuds internes dans un arbre.

Chaque feuille a au plus 5 caractères (*False*). Chaque noeud interne a $1+c_h+4$ caractères ($xc_h()()$). Donc il y a $5*2^h + (1+c_h+4)*(2^h - 1) = (10 + c_h)*2^h$ - $(5 + c_h)$ caractères dans un arbre.

Question 3.12

Nous cherchons la complexité du passage de l'arbre de décision au ROBDD associé, pour cela partons de l'arbre de décision ci-dessous. Cet arbre a pour table de vérité $table(int('01100011', 2), 8)$, et se porte bien pour montrer le pire des cas.

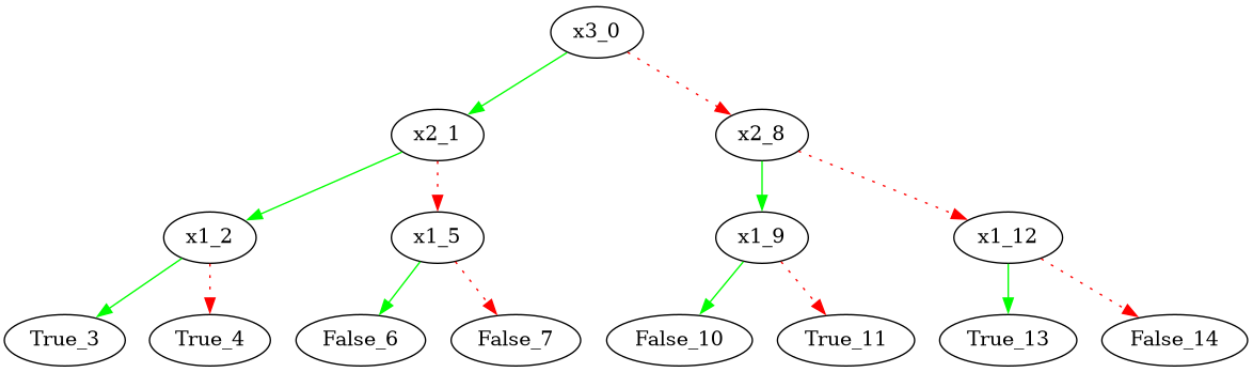


FIGURE 3.2 – Arbre de décision issu de table(int('01100011', 2) ,8)

La prochaine étape et de transformer chaque noeud en son mot de Lukasiewicz associé. Pour cela un parcours suffixe s'impose, et ce dernier possede une complexité en $O(nombre\ noeuds) = O(nombre\ de\ noeuds\ internes\ +\ nombre\ de\ feuilles) = O(2^h - 1 + 2^h) = O(2^{h+1} - 1)$.

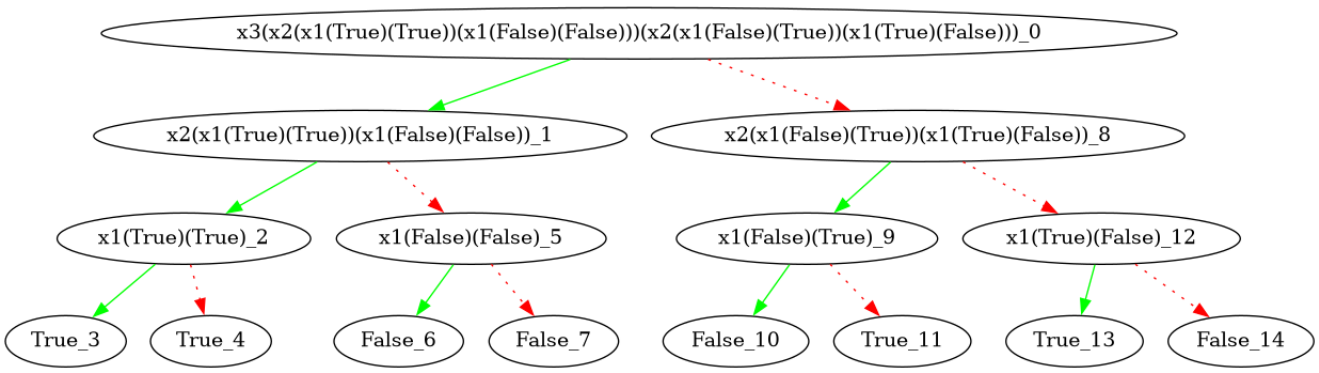


FIGURE 3.3 – Arbre de décision qui à chaque nœud associe le mot de Lukasiewicz enrichi

Supposons que nous utilisons un tableau (même si dans le code, nous utilisons un dictionnaire) qui va nous permettre de réaliser les deux règles, Terminal et Deletion Rule.
Au pire des cas tous les noeuds internes sont différents, ce qui donne $O(2^h - 1)$ éléments dans le tableau + 2 éléments venant des feuilles (True + False).
Au pire des cas nous voulons compresser l'une des feuilles qui se trouver dans après ces éléments, les feuilles en encadré orange ci-dessous :

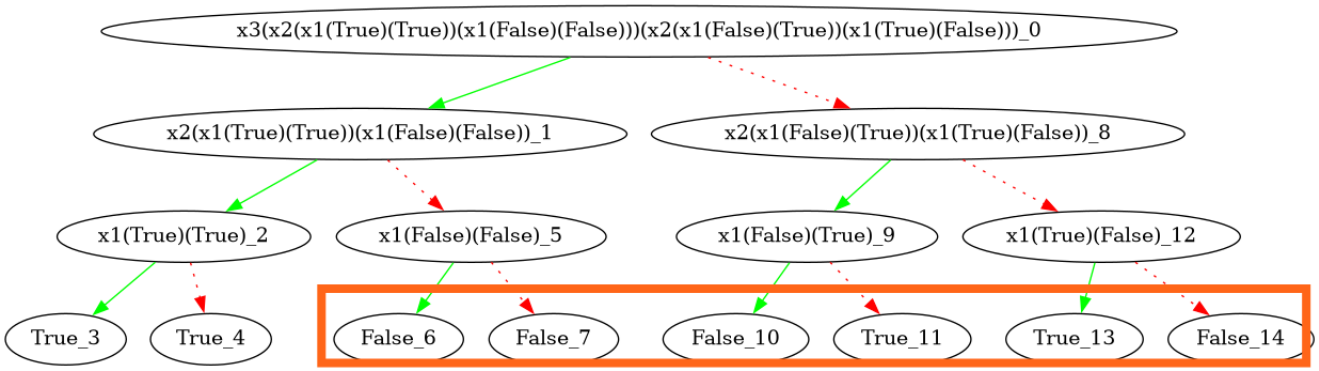


FIGURE 3.4 – Arbre de décision qui à chaque nœud associe le mot de Lukasiewicz enrichi

Pour cela nous comparons son mot de Lukasiewicz a tous les autres mots de Lukasiewicz dans le tableau. Nous avons donc au pire des cas $O((2^h - 1) + 2)$ comparaisons a faire, or ces comparaisons sont naïves, nous devons donc faire pour une comparaison entre deux éléments du tableau $O(l_h) = O(2^h)$ comparaisons caractère par caractère.
Finalement, nous trouvons une complexité en nombre de comparaisons en $O(((2^h - 1) + 2) * 2^h) = O(2^h + 2^h) = O(2^{2h})$

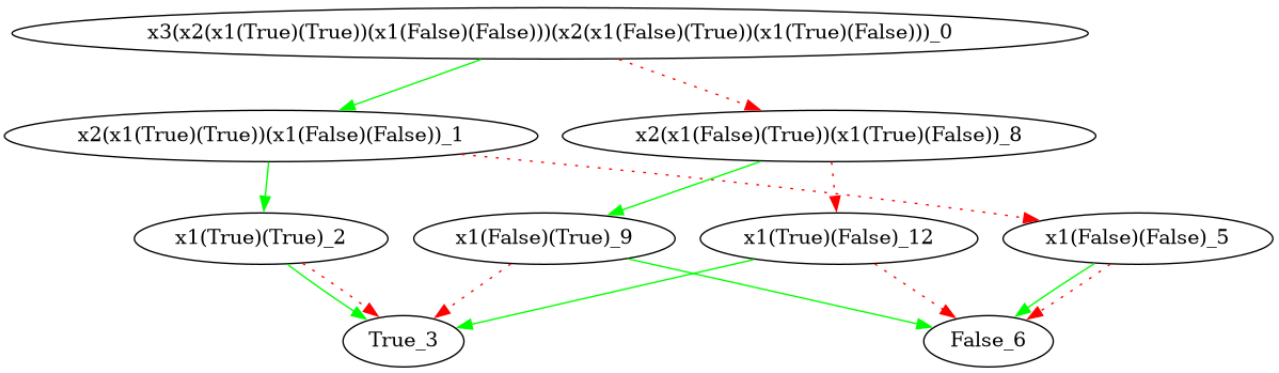


FIGURE 3.5 – DAG obtenu par compression de l'arbre précèdent avec Terminal et Merging Rule

Nous remarquons bien que la compression avec les deux règles ne compresse que les feuilles puisque tous les noeuds internes sont différents. Ce qui montre également que l'ordre d'application des règles est important seulement pour la Terminal Rule, celle-ci doit être avant les deux autres.
La dernière règle pour avoir le ROBDD associé est la Deletion Rule.
Au pire des cas nous avons donc le même nombre de noeuds internes comme précédemment, or pour chaque noeud nous devons comparer son fils gauche avec son fils droit. Nous avons donc le nombre de noeuds internes comparaisons a faire, i.e : $O(2^h - 1)$.
Nous comparons comme précédemment d'une manière naïve, nous avons donc une complexité en nombre de comparaisons en $O((2^h - 1) * l_h) = O((2^h - 1) * 2^h) = O(2^{2h} - 2^h) = O(2^{2h})$.

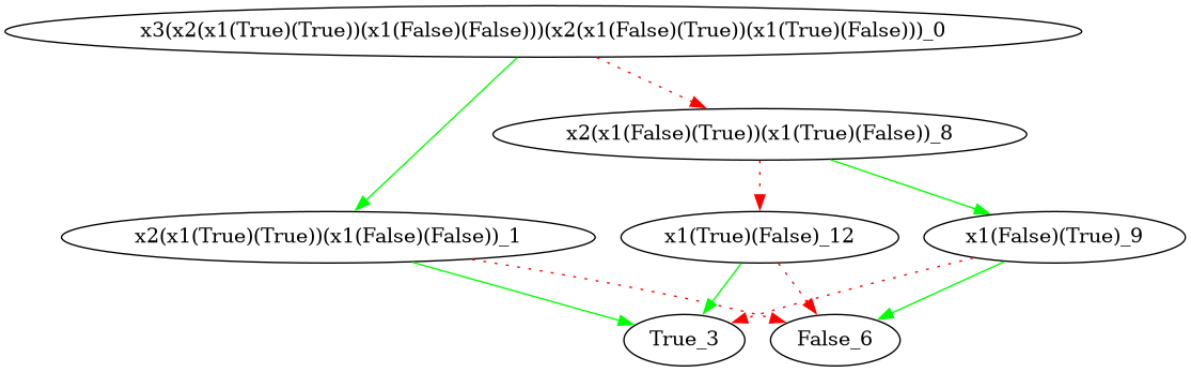


FIGURE 3.6 – ROBDD associée en appliquant Deletion Rule

Nous pouvons ajouter les complexités de toutes ces étapes pour trouver la complexité totale : $O((2^{h+1} - 1) + (2^{2h} + 2^{2h})) = O(2 * 2^{2h}) = O(2^{2h})$.

Question 3.13

Nous avons montré que la complexité de l'algorithme est $O(2^{2h})$ en fonction de h. Nous avons aussi montré qu'il y a 2^h feuilles et $2^h - 1$ noeuds internes dans un arbre.
Pour trouver la complexité du passage de l'arbre de décision au ROBDD associé nous devons ajouter les complexités de tous les étapes comme précédemment.
Le passage de l'arbre de décision à l'arbre de décision enrichi avec les mot de Lukasiewicz se fait

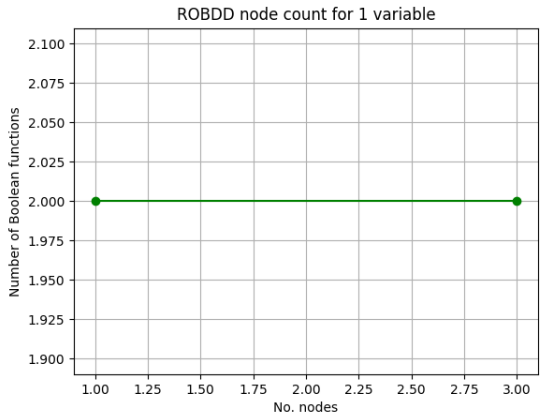
en $O(n)$ avec n le nombre de noeuds de l'arbre.
Nous savons qu'il y a n = $2 * 2^h$ - 1 noeuds au total dans le passage de l'arbre de décision avec les mot de Lukasiewicz au DAG compressé, et donc h = $log_2(n+1)-1$ ce qui implique une complexité pour cette étape en $O(2^{2h}) = O(2^{2*log_2(n+1)-2}) = O(2^{log_2((n+1)^2)-2}) = O(2^{log_2((n+1)^2)} - 2^2) = O((n+1)^2 - 2^2) = O(n^2)$.
Pour le passage du DAG compressé au ROBDD associé nous avons $2^h - 1$ noeuds internes et 2 feuilles (True + False) ce qui nous donne un nombre totale de noeuds $n = 2^h - 1 + 2 = 2^h + 1$, et donc $h = log_2(n-1)$, ce qui implique une complexité pour cette étape en $O(2^{2h}) = O(2^{2*log_2(n+1)}) = O(2^{log_2((n+1)^2)}) = O(n-1)^2 = O(n^2)$.

Finalement nous ajoutons tous ces complexités et donc pour passer de l'arbre de décision au ROBDD nous avons une complexité en nombre de comparaisons en fonction de n en $O(n + n^2 + n^2) = O(n^2)$.

Chapitre 4

Etude expérimentale

Question 4.14



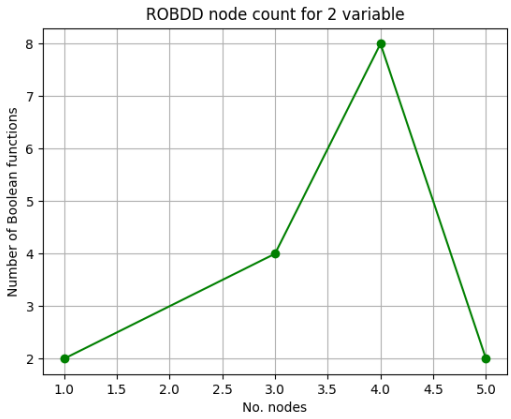
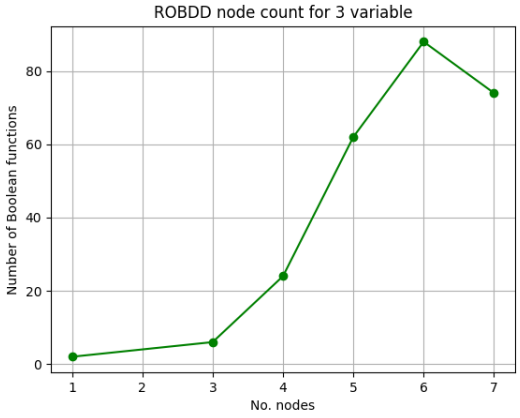


FIGURE 4.1



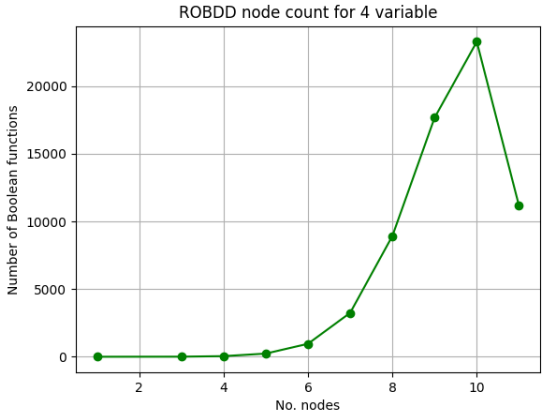


FIGURE 4.2

Dans les figures 4.1 et 4.2 nous pouvons remarquer les courbes représentant le nombre de variables booléennes en fonction du nombre de noeuds dans un ROBDD pour un nombre de variables allant de 1 à 4.

Ces courbes sont crées en créant tous les combinaisons possibles de ROBDD pour une variable n donnée. Il existe 2^{2^n} combinaisons possibles.
Ces 4 premières courbes sont identiques aux 4 premiers courbes de l'article puisque le nombre d'échantillons est relativement petit et ceci est calculé assez rapidement par nos machines. Ce n'est plus le cas à partir de 5 variables, ou même avec un pas plus grand ou des pas aléatoires le temps est reste long et donc difficile a calculer par nos machines en un temps convenable.

Question 4.15

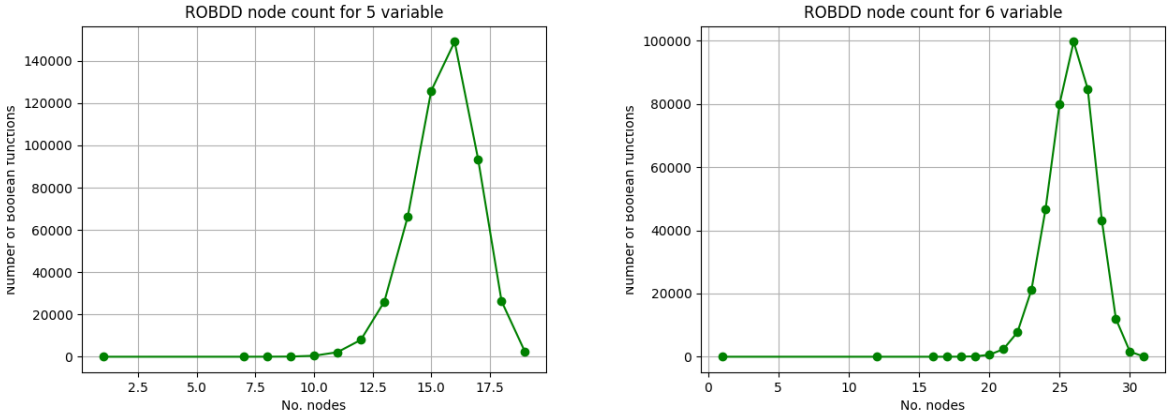


FIGURE 4.3

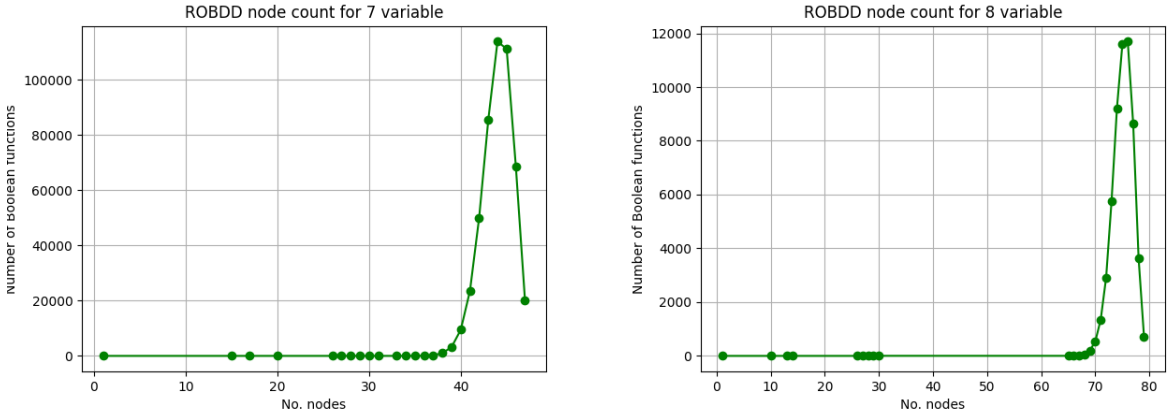


FIGURE 4.4

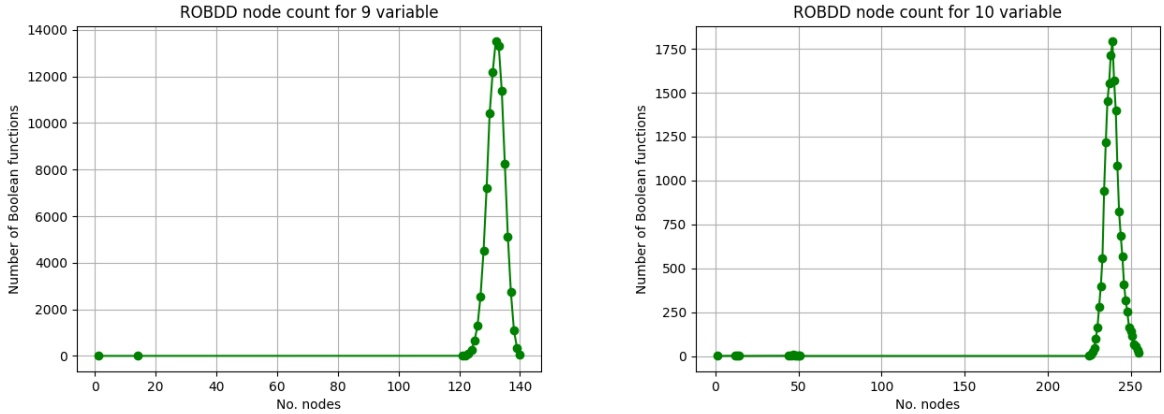


FIGURE 4.5

Dans les figures 4.3, 4.2 et 4.3 nous pouvons remarquer les courbes représentant le nombre de variables booléennes en fonction du nombre de noeuds dans un ROBDD pour un nombre de variables allant de 5 à 10.
Ces courbes sont des courbes similaires a celles de l'article puisque les échantillons calculés ne sont pas exhaustifs.
Pour tracer ces courbes nous avons parcourut le nombre d'échantillons complets avec un pas = 2^{2^n} / nombre de samples proposé dans la figure 11 de l'article. Avec n le nombre de variables.

Voir le code de **parcours_nb_noeuds(tree, tab) :** qui permet de compter le nombre de noeuds différents d'un arbre et **create_coubres(nbVariables, affichage)** qui permet de créer ces graphes pour un nombre de variables $nbVariables$ et afficher tous les ROBBD crées si $affichage$ est a True (par défaut a False puisque pour $nbVariables$ grand cela devient très long).
Pour tester, lancer dans le terminal *python main.py*, parmi les affichages de la partie IV il est indiqué le dictionnaire des fonctions booleenes et nombre de noeuds et un tableau similaire a celui de l'article pour un nombre de variables allant de 1 a 4. Les graphes sont visibles dans le dossier $./Figures/$.
Pour tester avec plusieurs variables changer l.418 $nombreVariables$ par x pour avoir les courbes et les informations allant jusqu'à (x-1). De plus, un affichage de tous les combinaisons est possible si l.421 on change False par True pour avoir tous les affichages dans des dossiers.

Question 4.16

No. Variables(n)	No. Samples	No. Unique Sizes	Compute Time hh::mm::ss	Seconds per ROBDD
1	4	2	0:00:00	1.98483e-05
2	16	4	0:00:00	0.000135407
3	256	6	0:00:00	0.000134168
4	65536	10	0:00:04	5.57667e-05
5	500055	14	0:00:48	9.62368e-05
6	400004	18	0:01:25	0.00021372
7	486893	25	0:03:25	0.000421237
8	56344	24	0:00:52	0.000915054
9	95000	22	0:03:02	0.001913
10	17976	42	0:01:18	0.00436046

FIGURE 4.6 – Informations similaires a celles du tableau de l'article

Chapitre 5

Pour Aller Plus Loin...

Fusion du ROBDD

Nous utilisons les informations du cours pour réaliser la fusion. Notre fusion applique l'opération booléenne xor (ou-exclusif).

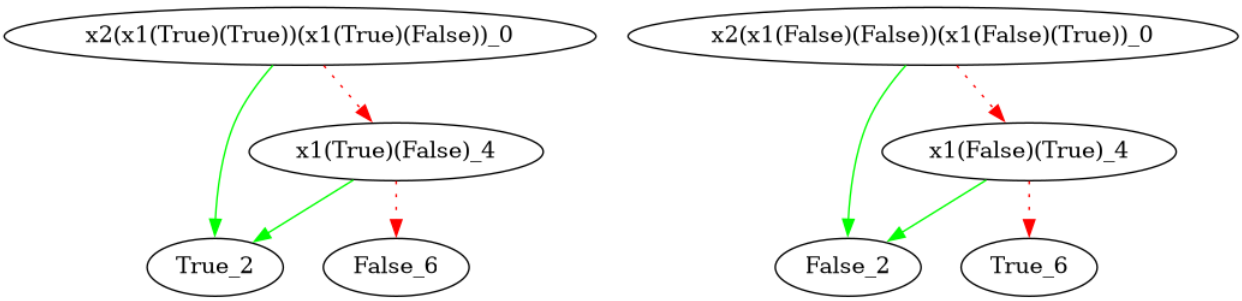


FIGURE 5.1 – ROBDDs a fusionner

Les tables de verités de ces deux ROBDDs sont : table(int('0111', 2), 4) et table(int('0111', 2), 4). Nous remarquons que les étiquettes des racines de ces deux ROBDDs contiennent plus d'informations que nécessaire ce qui ne doit pas être le cas dans notre algorithme de fusion puisque l'étiquette de la racine nous indique les opérations a faire.
Une mise a jour des mots de Lukasiewicz est effectué avec la fonction **luka_after(tree, text)**. Ce qui nous donne les ROBDDs suivants :

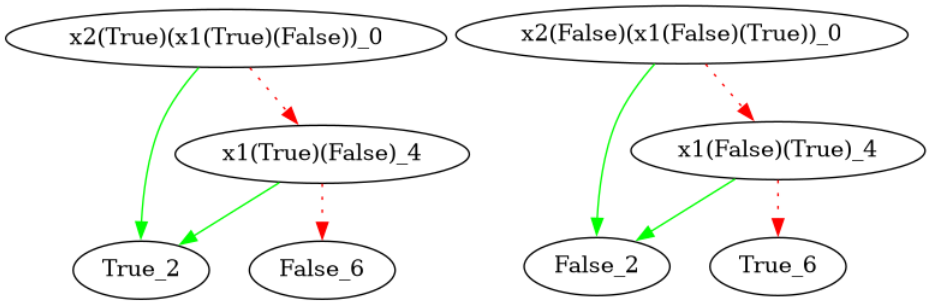


FIGURE 5.2 – ROBDDs avec les mots de Lukasiewicz à jour

Cette mise a jour permet de se fier a l'étiquette de la racine et donc la fusion peut avoir lieu, ce

qui donne l'ROBDD fusionné ci-dessous :

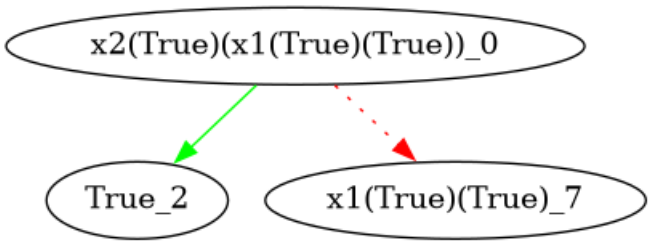


FIGURE 5.3 – ROBDD fusionné avec les deux ROBDDs précédents

Cette fusion se fait avec la fonction **fusion_ROBDD(tree1, tree2, treeRes)**.
Pour tester, lancer dans le terminal *python main.py*, parmi les affichages de la partie V il est indiqué que les ROBDD a fusionner et l'ROBDD fusionné sont dans le dossier *./Graphes/*.
Pour tester d'autres combinaisons changer les bits dans la table de vérité inverse des deux ROBDDs l.448 et l.466
A noter que une partie "Autres tests" existe et ou des tests similaires a ceux de la soutenance peuvent être effectués, notamment en changeant l.501 le nombre de variables et l.502 le nombre dans la table.
Le pseudo code de la fonction **fusion_ROBDD(tree1, tree2, treeRes)** se trouve ci-dessous :

Algorithm 10 Pseudo-code Fusion_ROBDD

Entrée(s) ROBDD *tree*1, ROBDD *tree*2, ROBDD résultat *treeRes*

si l'un des deux arbres est null **alors**
rendre l'autre
fin du si
si les deux arbre sont null **alors**
rendre null
fin du si
*val*1 ←valeur de tree1
*val*2 ←valeur de tree2 -1 si True et -2 si False
si val1 $<$ 0 and val2 $<$ 0 **alors**
rendre l'étiquette du résultant de l'opération xor entre *val*1 et *val*2 pour *treeRes*
fin du si
génération des enfants gauche et droit pour *treeRes* avec une étiquette vide
si val1 $<$ 0 and val2 $>$ 0 **alors**
fusion_ROBDD(tree1, tree2.enfant_gauche, treeRes.enfant_gauche) genère l'étiquette pour enfant gauche de treeRes
fusion_ROBDD(tree1, tree2.enfant_droit, treeRes.enfant_droit) genère l'étiquette pour enfant droit de treeRes
fin du si
si val1 $>$ 0 and val2 $<$ 0 **alors**
fusion_ROBDD(tree1.enfant_gauche, tree2, treeRes.enfant_gauche) genère l'étiquette pour enfant gauche de treeRes
fusion_ROBDD(tree1.enfant_droit, tree2, treeRes.enfant_droit) genère l'étiquette pour enfant droit de treeRes
fin du si
si val1 $>$ 0 and val2 $>$ 0 **alors**
si *val*1 = *val*2 **alors**
fusion_ROBDD(tree1.enfant_gauche, tree2.enfant_gauche, treeRes.enfant_gauche) genère l'étiquette pour enfant gauche de treeRes
fusion_ROBDD(tree1.enfant_droit, tree2.enfant_droit, treeRes.enfant_droit) genère l'étiquette pour enfant droit de treeRes
fin du si
si val1 $>$ val2 **alors**
fusion_ROBDD(tree1, tree2.enfant_gauche, treeRes.enfant_gauche) genère l'étiquette pour enfant gauche de treeRes
fusion_ROBDD(tree1, tree2.enfant_droit, treeRes.enfant_droit) genère l'étiquette pour enfant droit de treeRes
fin du si
si val1 $<$ val2 **alors**
fusion_ROBDD(tree1.enfant_gauche, tree2, treeRes.enfant_gauche) genère l'étiquette pour enfant gauche de treeRes
fusion_ROBDD(tree1.enfant_droit, tree2, treeRes.enfant_droit) genère l'étiquette pour enfant droit de treeRes
fin du si
fin du si

Sortie(s) *treeRes.valeur*