Rapport de projet

Adan Bougherara Vivien Demeulenaere

21 novembre 2021

Table des matières

1 Présentation

Le langage choisit pour mener le projet est le Python. Les packages utilisés sont math, numpy, matplotlib, time, random ainsi que graphviz.

1.1 Echauffement

Le code relatif à cette section est contenu dans le fichier echauffement.py.

1.1.1 Représentation d'un entier de taille arbitraire

Afin de représenter un entier en base 2 de taille arbitraire, nous avons utilisé des listes de booléens offertes par Python. Ainsi, un entier représenté sur n bits correspondra à un tableau de taille n et son i^{eme} bit correspondra au i^{eme} élément du tableau. Par exemple on aura :

$$38_{10} = 011001_2 = \text{[False, True, True, False, False, True]}$$

1.1.2 Fonctions decomposition et completion

La fonction `decomposition` se chargera d'effectuer le passage d'un entier en base 10 vers notre représentation binaire. De plus, une fonction `completion` prenant en argument une liste de bits l et un entier naturel n renverra la liste l complétée par des bits `False` à droite ou tronquer de manière à obtenir une nouvelle liste de taille n.

1.1.3 Fonction table

Enfin, une fonction `table` prenant en entrée deux entiers n et m décomposera n en base 2 et le complétera de manière à ce qu'il soit de taille m. On obtiendra alors une *table de vérité* de $log_2(n)$ variables et n sorties.

2 Arbre de décision et compression

Le code relatif à cette section est contenu dans le fichier abd.py

2.1 Structure de données

Afin d'encoder un arbre binaire de décision, il a été décidé d'implémenter la structure suivante :

```
class abd:
    def __init__ (self, etiquette=None, faux=None, vrai=None):
        self.etiquette = etiquette
        self.faux = faux
        self.vrai = vrai
```

Un arbre contient donc une étiquette correspondant à un nom de variable ou à un booléen. De plus, ses fils gauche et droit (représentés respectivement par `faux` et `vrai`) sont également des arbres. Ceux-ci peuvent éventuellement être vides, auquel cas, on utilisera la valeur `None`.

2.2 Fonctions cons_arbre et luka

Afin de manipuler cette structure, une méthode statique `cons_arbre` a été définie. Elle prend en entrée une *table de vérité* et construit l'arbre de décision T qui lui est associé. Une fois T construit avant de pouvoir le compresser, une méthode consiste à associer à chacun de ses noeuds le mot de *Lukasiewicz* enrichi associé au sous arbre enraciné en ce noeud. Ce travail sera effectué par la méthode `luka`.

2.3 Fonction compression

Il est à présent possible de définir la méthode `compression` qui va éliminer les redondances d'arbres en redirigeant les parents d'arbres isomorphes vers un arbre unique (*Merging Rule*). De plus, les seules feuilles possibles dans l'arbre seront les singletons `True` et `False` (*Terminal Rule*). Cette méthode est une première étape vers la compression de notre structure de donnée.

2.4 Fonction dot

Les résultats de cette compression peuvent être visualisés à l'aide de la méthode `dot` qui génère un fichier du graphe au format dot.

3 Arbre de décision et ROBDD

Le code relatif à cette section est contenu dans le fichier abd.py

3.1 Fonction compression_bdd

La méthode `compression` de la section précédente n'est pas tout à fait suffisante pour obtenir un ROBDD. En effet, pour obtenir une telle structure, il est nécessaire de retirer de l'arbre tous les noeuds ayant le même fils à droite et à gauche (*Deletion Rule*). La méthode `compression_bdd` effectue le même travail que `compression` en appliquant cette dernière règle en plus.

3.2 Analyse de complexité

Nous allons nous intéresser dans cette partie à la complexité permettant de compresser un arbre de décision en ROBDD. Celle-ci sera mesurée en terme de nombre de comparaisons de caractères.

3.2.1 Majoration du mot de Lukasiewicz associé à la racine

Pour ce faire, on définit la longueur d'un mot de *Lukasiewicz* l_h de la racine d'un arbre de hauteur $h \in \mathbb{N}$.
Afin d'établir une mesure de complexité de notre algorithme, il nous sera utile de prouver les propriétés suivantes pour un arbre de décision A de hauteur h.

P1(h) : A contient exactement $2^h - 1$ noeuds internes.

P2(h) : A contient exactement 2^h feuilles.

Montrons par récurrence sur $h \in \mathbb{N}^*$ **P1(h)** et **P2(h)**.

Initialisation : Montrons **P1(1)** et **P2(1)**.

Pour $h = 1$, A est l'arbre de décision réduit à une variable et deux feuilles.

On a alors bien $2^1 - 1 = 1$ noeud interne (il s'agit de la racine)

De plus on a bien $2^1 = 2$ feuilles

D'où **P1(1)** et **P2(1)**.

Hérédité : Soit $h \in \mathbb{N}^*$, supposons **P1(h)** ainsi que **P2(h)**, montrons **P($h+1$)** et **P2($h+1$)**

Pour obtenir un arbre de décision A_2 de hauteur $h+1$ à partir d'un arbre de décision A_1 de hauteur h, il suffit de remplacer les feuilles de A_1 par des noeuds internes et de leur ajouter deux feuilles pour fils. Ainsi, A_2 contiendra par hypothèse de récurrence $2^h - 1 + 2^h = 2^{h+1} - 1$ noeuds.

De plus, il contiendra $2.2^h = 2^{h+1}$ feuilles.

D'où **P1($h+1$)** et **P2($h+1$)**.

D'où **P1(h)** et **P2(h)**.

Soit $h \in \mathbb{N}$ tel que $1 \leq$ h ≤ 9.
Avec une telle hauteur, chaque noeud interne x_i sera représenté par deux caractères. De plus, chaque feuille est étiquetée par `True` ou `False` et est donc représentée par 5 caractères au maximum. Enfin, chaque noeud interne utilise 4 parenthèses pour entourer ses deux enfants (2 par enfant).
Étant donné qu'un arbre de hauteur h contient $2^h - 1$ noeuds internes (d'après **P1(h)**) et 2^h feuilles (d'après **P2(h)**), on peut déduire la valeur de l_h :

$$\begin{aligned} l_h &= 2.(2^h \text{ - } 1) + 5.2^h + 4.(2^h \text{ - } 1) \\ &= 2.2^h \text{ - } 2 + 5.2^h + 4.2^h \text{ - } 4 \\ &= (2 + 5 + 4).2^h \text{ - } 6 \\ &= 11.2^h \text{ - } 6 \end{aligned}$$

3.2.2 Complexité au pire cas de l'algorithme de compression en fonction de la hauteur

Soit $i < h$, un étage de l'arbre Pour chaque étage on a $l_i \,.\, 2^{h-i}$ comparaisons (cas des noeuds internes). De plus, pour les feuilles, $5 \,.\, 2^h$ comparaisons au maximum sont nécessaires. Ainsi, on obtient la complexité maximale suivante :

$$\begin{aligned} C = \sum_{i=0}^{h-1} (l_i \,.\, 2^{h-i}) + 5 \,.\, 2^h &= \sum_{i=0}^{h-1} ((11.2^i - 6) \,.\, 2^{h-i}) + 5 \,.\, 2^h \\ &= \sum_{i=0}^{h-1} (11.2^{i+h-i} - 6.2^{h-i}) + 5 \,.\, 2^h \end{aligned}$$

$$= \sum_{i=0}^{h-1} (l_h + 6 - 6.2^{h-i}) + 5 \ . \ 2^h$$

$$= h(l_h + 6) - 6 \sum_{i=0}^{h-1} (2^{h-i}) + 5 \ . \ 2^h$$

$$= h(l_h + 6) - 6.(2^h - 1) + 5 \ . \ 2^h$$

$$= h.l_h + 6.h - 6.2^h + 6 + 5 \ . \ 2^h$$

$$= h.l_h + 6.h - 2^h + 6$$

$$= h.(11.2^h - 6) + 6.h - 2^h + 6$$

$$= h.11.2^h - 6.h + 6.h - 2^h + 6$$

$$= h.11.2^h - 2^h + 6$$

$$= (11.h - 1).2^h + 6$$

3.2.3 Complexité au pire cas de l'algorithme de compression en fonction du nombre de noeuds

Exprimons maintenant C en fonction du nombre de noeud n. On sait que $n = 2^{h+1} - 1$ avec h la hauteur de l'arbre binaire. Ainsi, on peut réécrire C de la manière suivante :

$$\begin{aligned} \text{C} &= (11.log_2(n+1) - 1 - 1).2^{log_2(n+1)-1} + 6 \\ &= \frac{(11.log_2(n+1)-2).(n+1)}{2} + 6 \end{aligned}$$

3.2.4 Cas d'une hauteur supérieure à 9

Si h n'est plus majoré par 9, deux caractères ne suffisent plus pour représenter une variable de la forme x_i avec $i \in [|1; 9|]$. En effet, il faudrait ajouter un caractère à chaque fois que la hauteur est multipliée par 10. En reprenant l'équation de la question 3.11, on a alors :

$$l_h = (1 + \lceil log_{10}(h) \rceil).(2^h - 1) + 5.2^h + 4.(2^h - 1)$$

Or $\lceil log_{10}(h) \rceil \geq 1$ car h > 0. Donc $1 + \lceil log_{10}(h) \rceil \geq 2$. Donc la longueur maximale d'un mot de Lukasiewicz à la racine d'un arbre de hauteur h > 0 est supérieure ou égale à l_h quand h est majorée par 9. Par conséquent, la complexité C est augmentée à chaque fois que l'on multiplie la hauteur par 10.

4 Etude expérimentale

Dans cette partie, nous allons comparer nos résultats à ceux présentés dans l'article *A Theoretical and Numerical Analysis of the Worst-Case Size of Reduced Ordered Binary Decision Diagrams.*

4.1 ROBDD de 1 à 4 variables

Pour des ROBDD n'excédant pas 4 variables, nous avons pu générer la totalité des ROBDD possibles afin d'obtenir nos courbes. Celles-ci représentent le nombre de fonctions booléennes en fonction du nombre de noeuds.
Comme nous pouvons le constater, nos résultats sont identiques avec ceux de l'article.

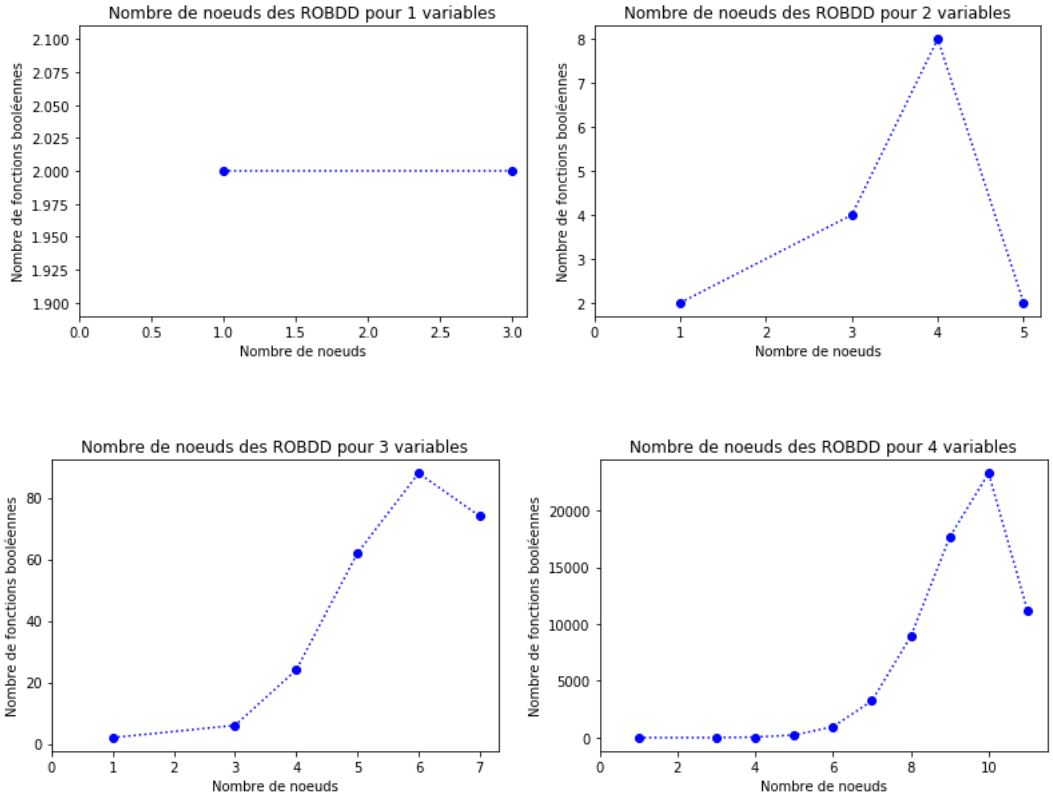


FIGURE 1 – Histogrammes illustrant la répartition des tailles de ROBDD pour 1 à 4 variables

4.2 ROBDD de 5 à 10 variables

À partir de 5 variables, on ne peut plus tester de manière exhaustive l'ensemble des ROBDD. En effet, il y a 2^{2^i} ROBDD avec i le nombre de variables. Nous avons donc du extrapoler nos résultats en prenant un nombre restreint d'arbres aléatoirement. Afin de compenser cela, nous avons multiplié le nombre de fonctions booléennes par le coefficient :

$$c = \frac{nombre\ de\ ROBDD\ possibles}{taille\ de\ l'chantillon}$$

Nous trouvons également des résultats identiques à ceux de l'article.

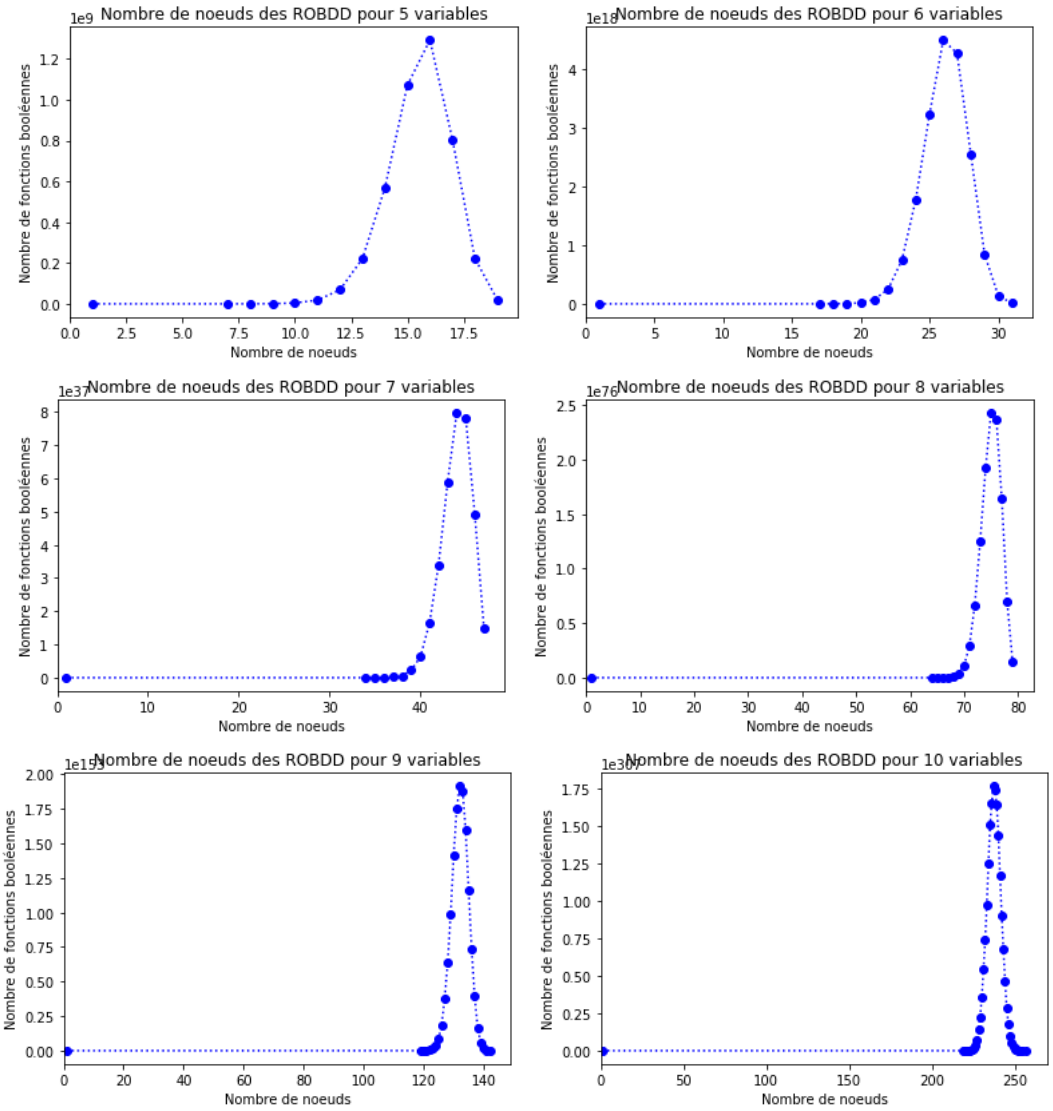


FIGURE 2 – Histogrammes illustrant la répartition des tailles de ROBDD pour 5 à 10 variables

Pour chaque variables, nous avons relevé le temps de calcul et le nombre de tailles uniques avec un échantillon de 100000 arbres :

Nb. variables	**Nb. arbres**	**Nb. tailles uniques**	**Temps de calcul (en s)**	**Temps de calcul par ROBDD (en s)**
5	100000	14	81.85602951	0.00081856
6	100000	16	87.1204493	0.000871204
7	100000	15	114.6158905	0.001146159
8	100000	17	194.0332394	0.001940332
9	100000	25	384.8936143	0.003848936
10	100000	38	821.0664122	0.008210664

Nous obtenons des résultats très similaires concernant le nombre de tailles uniques. Ces légers écarts s'expliquent en raison de différences en taille de nos échantillons par rapport à ceux de l'étude. Enfin, nous notons qu'en temps de calcul nos résultats sont largement meilleurs.

5 Pour aller plus loin...

5.1 Implémentation de la combinaison de deux ROBDD

Nous avons implémentés la synthèse de ROBDD. À partir de deux ROBDD représentant respectivement les fonctions f et f', on peut en effet construire un ROBDD représentant g avec g étant une combinaison de f et f'. Pour ce faire nous avons une méthode `fusion_ROBDD` qui fusionne deux ROBDD en suivant les règles suivantes :
On note $\alpha = (v, l, h)$ et $\alpha' = (v', l', h')$ deux ROBDD.

$$\alpha \diamondsuit \alpha' = \begin{cases} (v, l \diamondsuit l', h \diamondsuit h') & \text{si v = v'} \\ (v, l \diamondsuit \alpha', h \diamondsuit \alpha') & \text{si v < v'} \\ (v', \alpha \diamondsuit l', \alpha \diamondsuit h') & \text{si v > v'} \end{cases}$$

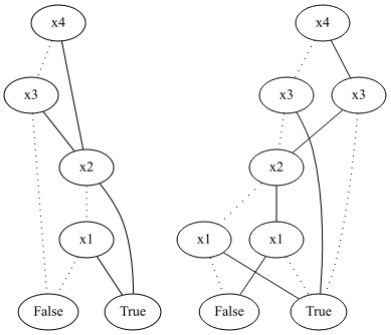
Enfin la fonction renomme chaque noeud en deux nombres binaires, en faisant remonter les tables du sous-arbre faux et du sous-arbre vrai.
On applique ensuite la méthode `simplification_et_reduction_ROBDD` sur l'arbre obtenu après l'étape de la fusion. Elle consiste dans un premier temps à réunir les deux binaires en un seul via l'opérateur booléen. Dans un second temps, la réduction consiste à enlever les noeuds doublons. Autrement dit, deux noeuds ayant pour étiquette le même nombre binaire deviennent un seul et même noeud.

5.2 Test de notre implémentation

On définit tout d'abord notre opérateur booléen (nous avons choisi dans ce test l'opérateur $\wedge$) et on créé nos deux ROBDD :

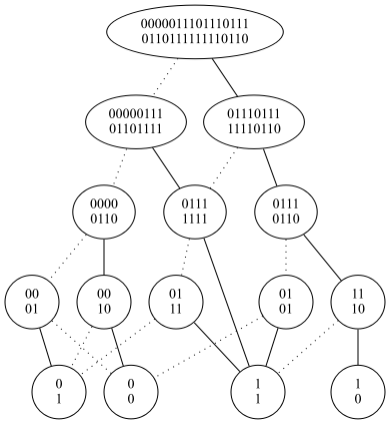
```
def et (table1, table2):
    res = ""
    for i in range (len(table1)):
        if table1 [i] == "1" and table2 [i] == "1":
            res += "1"
        else:
            res += "0"
    return res

arbre = abd.cons_arbre(table(61152,16))
arbre2 = abd.luka(arbre)
arbre3 = abd.compression_bdd(arbre2)

arbre4 = abd.cons_arbre(table(28662,16))
arbre5 = abd.luka(arbre4)
arbre6 = abd.compression_bdd(arbre5)
```

On obtient ainsi les arbres suivants :



Puis on appelle `fusion_ROBDD` avec ces deux arbres, on obtient alors :

```
fusion3_6 = abd.fusion_ROBDD (arbre3, arbre6)
```



Enfin, on appelle `simplification_et_reduction_ROBDD` avec le résultat de `fusion_ROBDD` ainsi que l'opérateur booléen. On obtient ainsi notre ROBDD final :

```
fusion3_6_bdd = abd.simplification_et_reduction_ROBDD(fusion3_6, et)
```