Algorithmique avancée
Rapport

Projet compression ROBDD

Élève :
Yacine IDOUAR
Hamid KOLLI

Enseignant :
Antoine Genitrini

16 décembre 2022

Table des matières

1 Echauffement

Nous vous proposons dans cette section le pseudo-code des 3 fonctions demandées.

1.1 Décomposition

La fonction "décomposition" convertit un entier écrit en base 10 vers son écriture en base 2.

```
decomposition (nombre) :
    L : liste de booleen
    Tant Que nombre différent de 0 Faire :
        AjoutFin(L,nombre mod 2)
        nombre = nombre/2
    Fin tant que
    retourne L
```

1.2 Complétion

La fonction "complétion" retourne "n" bits d'une liste de bit donnée en entrée, si $n > size(liste)$, on complète la liste avec des "false".

```
completion (Liste_bit, n) :
    diff = n - size(Liste_bit)
    Pour i allant de 0 a diff :
        AjoutFin(Liste_bit,false)
    Fin pour
    retourne ListeBit[0..n]
```

1.3 Table

La fonction "Table" définit une table de vérité à partir d'un entier et de la taille de la table.

```
Table (nombre, tailleTable) :
   retourne completion(decomposition(nombre),tailleTable)
```

2 Structure des arbres de décision et leurs construction

2.1 Structure d'encodage de l'arbre

Nous vous présentons dans cette section les deux structures que nous avons utilisées pour encoder nos arbres dans les deux versions "C++" et "python".

2.1.1 Structure C++ :

Pour cette partie nous avons utilisé la classe Noeud suivante :

```
class Noeud {
    identifiant : entier
    fils_droit : Pointeur sur fils droit
    fils_gauche : Pointeur sur fils gauche
    pere : Pointeur sur le pere
    mot_Lukasiewicz : mot de lukasiewicz du sous-arbre
    }
```

Remarque : Nous avons utilisé des pointeurs enrichit défini dans la librairie standard du "C++" qui sont nommés "shared pointer"(shared_ptr). C'est des pointeurs avec un compteur de référence associé, ces derniers facilitent la gestion mémoire dans le cas de multiples réferences sur un même objet (arbre après compression).

2.1.2 Structure Python :

Nous avons utilisé la classe Tree suivante qui represente un noeud interne ou une feuille :

```
ident = 1
class Tree:
    def __init__(self,label):
        global ident
        self.label = label
        self.false = None
        self.true = None
        self.taille = 0
        self.hauteur = 0
        self.visited = False
        self.id = ident
        ident+=1
    def __repr__(self) -> str:
        return f"{self.label}"
```

Pour pouvoir dessiner un arbre à partir d'un fichier **.dot** nous avons utilisé un **id** qui est unique car les labels des noeuds ne sont pas uniques. Les champs, taille et hauteur de la classe sont utilisés que pour la racine de l'arbre, et sont calculés une fois lors de la création de l'arbre. "Visited" est utilisé pour les fonctions **dot** et **taille** pour ne pas passer deux fois par le même noeud. Vu qu'on engendre un effet de bord, on utilise une fonction qui met ce champ à "false". Le champ **"false"** est le fils gauche, **"true"** est le fils droit, si l'un des deux est **None** donc on est sur une feuille.

2.2 Construction de l'arbre

2.2.1 En C++

Nous présentons dans cette section l'algorithme utilisé dans la version "C++" pour la construction d'un arbre de décision.

Dans notre construction les feuilles sont des noeuds avec un identifiant 0. Les noeuds internes ont un identifiant : $id > 0$.

Les primitives utilisées :

- liste_noeud : liste doublement chainée contenant des noeuds.

- ajoutFin(liste_noeud,noeud) : ajoute un noeud a la fin de liste.

- retireXElem(liste_noeud) : retire les X premiers élements de la liste et les retourne.

```
1     constructionArbre(table) :
2         Pour (bool1,bool2) dans table :
3              feuille = constructionFeuille(bool1,bool2)
4              AjoutFin(liste_noeud,feuille)
5         Tant que size(liste_noeud) > 1 Faire :
6             (noeud1,noeud2) = retire2Elem(liste_noeud)
7             AjoutFin(liste_noeud,constructionNoeud(noeud1,noeud2)
8         Fin tant que
9         retourne retireElem(liste_noeud)
```

Remarque : Cette construction fonctionne car on travaille avec des tables de vérité qui sont des puissances de deux, donc nous pouvons retirer 2 éléments de la table sans se soucier de la parité du nombre d'éléments.

2.2.2 En python

Nous présentons dans cette section l'algorithme utilisé dans la version "Python" pour la construction d'un arbre de décision.
On utilise pour cette construction deux fonctions, une fonction principale d'initialisation et une autre de récurssion.

Les primitives utilisées :

- makeLeaf(label) : qui permet de crée une feuille

- makeNode(label,filsGauche,filsDroit) : qui permet de crée un noeud interne

- table(numeroFonction,tailleTable) : qui permet de crée une table de vérité

```
cons_arbre(number) :
    table = table de verite de number agrandit a une puissance de 2

    arbre =  __cons_arbre(taille de arbre,la premiere moitier de la table,la deuxieme moitier de la table)

    affectation de la hauteur et la taille de arbre

    retourner  arbre
```

```
__cons_arbre(number:int,arrayLeft : list, arrayRight: list) -> Tree:

    # len(arrayLeft) = len(arrayRight) because len(arrayLeft)+len(arrayRight) is a power of 2

    si taille de premiere moitier est 0 et taille de deuxieme est 1 :
        retourne une feuille avec comme label element de la deuxieme moitie

    si taille de premiere moitie est 1 et taille de deuxieme est 0
        retourne une feuille avec comme label element de la premiere moitier

    si taille de premiere moitie est 1 et taille de deuxieme est 1
        retourner un noeud avec comme fils deux feuilles la premiere avec label element de la premiere moitie et la deuxieme avec label element de la deuxieme moitie

    sinon retourner un noeud avec fils les arbres resultant des deux appels recursifs en divisant la premiere moitie en deux et la deuxieme moitie en deux
```

3 Compression

3.1 Approche naïve

3.1.1 Mot de lukasiewicz

La construction des mot de lukasiewicz se fait par récurssion.

```
motLukasiewicz(noeud) :
    retourne racine.identifiant + "(" + motLukasiewicz(noeud.fils_gauche)+ ")" + "(" + motLukasiewicz(noeud.fils_doit) + ")"
```

3.1.2 Fonction de compression

Le principe de la fonction de compression consiste à parcourir l'arbre et fusionner tous les sous-arbres isomorphes (même mot de Lukasiewicz).

Les primitives utilisées :

- liste_noeud_mot : un tableau de couple (mot de lukasiewicz, pointeur sur sous-arbre définit par le mot).

- AjouterCoupleTableau : ajoute un noeud et son mot de lukasiewicz au tableau de couples.

```
1     compressionArbre(racine, tableau_noeud_mot) :
2         Si racine vide :
3             retourne nulle
4         Sinon :
5             racine.mot_Lukasiewicz = motLukasiewicz(racine)
6             Si mot dans tableau :
7                 retourne le noeud associer au mot
8             Sinon :
9                 racine.fils_gauche = compressionArbre(racine.fils_gauche,
10                                      tableau_noeud_mot)
11                racine.fils_droit = compressionArbre(racine.fils_droit,
12                                      tableau_noeud_mot)
13                AjouterCoupleTableau(tableau_noeud_mot,racine)
14                retourne racine
15            Fin si
16        Fin si
```

3.2 Approche par hachage

3.2.1 Mot de Lukasiewicz

- La construction des mots de lukasiewicz se fait par récurssion.

- Au lieu de construire une chaine de caractères pour représenter un mot de Lukasiewicz, on construit une valeur hachée pour éviter les concatenations des chaines (remplacer les concatenations par les additions et des multiplications *plus efficace en complexité*). Mais aussi éviter les comparaisons entre des chaines de caractères (remplacer les comparaisons entre deux chaines par des comparaisons entre deux entiers *plus efficace en complexité*).

- Utilisation de deux tables de hachage (dictionnaire en python), une qui contient les structures et leurs encodages, et une autre pour les haches et les pointeurs vers les noeuds qui les représente, si deux noeuds ont le même haché, on garde que le dernier parcouru et on écrase le premier. Pour la première table l'encodage est égal à l'encodage de la structure précédente incrémenté de 1, cette table est utilisée pour avoir des petits haches et pour ne pas calculer le haché du père à partir des hachés des fils mais à partir des encodage des fils dans la première table (Normalisation des valeurs).

- La construction se fait avec deux fonctions, une principale qui initialise les dictionnaires et une secondaire qui fait la récurssion.

- La fonction principale retourne un arbre de décision étiqueté par les haches des mots de Lukasiewicz et la table des haches avec leurs pointeurs comme valeur.

Les primitives utilisées :

- makeLeaf(label) : qui permet de crée une feuille.

- makeNode(label,filsGauche,filsDroit) : qui permet de crée un noeud interne.

- hash_luka : qui calcule un haché à partir des encodages des fils,la hauteur et la taille de l'arbre.

```
1     luka (racine: Tree) :
2         initaliser la table hache-noeud
3
4         initaliser la table Structure(triplet(label,encodage fils gauche,
      encodage fils droit), encodage) ou (label noeud, encodage)
5
6         mettre {true -> 1},{false -> 0} dans la table Structure-encodage
7
8         arbreLuka = __luka(racine, table hache-noeud, table Structure-
      encodage, racine)
9
10        retourner (arbreLuka, table hache-noeud)
```

```
1     __luka(noeudATraiter, table hache-noeud, table Structure-encodage,
      racine):
2
3         Si le noeudATraiter est une feuille :
4             ajouter la feuille dans la table hache-noeud avec comme hache
      son encodage dans la table Structure-encodage
5             retourner le noeud et son encodage
6          Sinon :
7             recuperer encodage de la variable dans la table Structure-
      encodage si existe dans table
8             Sinon :
9                 ajouter la variable dans table structure-encodage
10
11        (noeud gauche,encodage du fils gauche) = __luka(tree.false, dict,
      dictl , racine)
12
13        (noeud droit,encodage du fils droite) = __luka(tree.true, dict,
      dictl, racine)
14
15        recuperer encodage de la structure (lable du noeudATraiter,encodage
       du fils gauche,encodage du fils droit) dans la table Structure-encodage
       si existe
16        Sinon :
17            ajouter la structure dans la table
18
19        calculer le hache Lukasiewicz du noeudATraiter et ajouter dans la
      table hache-noeud
20
21        retourner le noeud et son encodage dans la table Structure-encodage
```

Complexité temporelle :

On mesure la complexité de cet algorithme par le nombre de haches calculés.
On sait qu'un accès en lecture et en écriture à une table de hachage est en O(1). La Complexité de la fonction de hachage est en O(1).
On parcourt tous les noeuds une seule fois donc on a une complexité en O(n) avec "n" la taille de l'arbre.

3.2.2 Fonction de compression

La compression se fait en utilisant deux fonctions, une principale qui appelle la fonction Luka qui renvoie une table de hachage et un arbre étiqueté par les haches, et une fonction secondaire qui compresse.

```
1     compression(tree) :
2         tester si taille de arbre > 4 (une variable)
3         Si oui :
4             retourne arbre
5         Sinon :
6             calculer arbre Lukasiewicz et table hache-noeud
7             retourne __compression(arbre Lukasiewicz, table hache-noeud)
```

```
1      __compression(arbre Lukasiewicz, table hache-noeud) :
2         Si feuille :
3             retourner le noeud qui corresond au hache de l'etiquette de l'
      arbre en utilisant la table
4         Sinon :
5             on recupere le noeud qui corresond au hache de l'etiquette de l
      'arbre en utilisant la table
6             fils gauche =__compression(fils gauche,table hache-noeud)
7             fils droit =__compression(fils droit,table hache-noeud)
8         retourne arbre resultant
```

4 Arbre de décision et ROBDD

4.1 Compression BDD

Nous allons vous présenter dans cette section les deux méthodes implémentées pour réaliser la construction d'un ROBDD.

4.1.1 Méthode C++

Nous remrquons que dans la fonction compression, 2 des 3 règles de la construction des ROBDD sont respectées, "Terminal rule" et "Merging rule". Il suffit alors de satisfaire la dernière "Deletion Rule".
Nous avons repris le même code avec un test supplémentaire pour appliquer la règle.

```
1     compression_bdd(racine, tableau_noeud_mot) :
2         Si racine vide :
3             retourne nulle
4         Sinon :
5             racine.mot_Lukasiewicz = motLukasiewicz(racine)
6             Si mot dans tableau :
7                 retourne le noeud associer au mot
8             Sinon :
9                 fils_gauche_tempo = compressionArbre(racine.fils_gauche,
10                                    tableau_noeud_mot)
11                fils_droit_tempo = compressionArbre(racine.fils_droit,
12                                    tableau_noeud_mot)
13                Si  fils_gauche_tempo  == fils_droit_tempo :
14                    Si racine.pere différent de null :
15                        Si racine = racine.pere.fils_gauche :
16                            racine.pere.fils_gauche = fils_gauche_tempo
17                        Sinon :
18                            racine.pere.fils_droit = fils_gauche_tempo
19                        Fin si
20                        retourne fils_gauche_tempo
21                    Sinon :
22                        retourne racine.fils_gauche
23                    Fin si
24                Sinon :
25                    racine.fils_gauche = fils_gauche_tempo
26                    racine.fils_droit =  fils_droit_tempo
27                AjouterCoupleTableau(tableau_noeud_mot,racine)
28                retourne racine
29            Fin si
30        Fin si
```

4.1.2 Méthode Python

Nous avons utilisé 3 fonctions chaqu'une pour une loi (terminal,mergin, deletion) qui utilisent la table hache-noeud et l'arbre géneré par la fonction luka.

- Terminal rule :

```
terminal_rule(table hache-noeud, arbre Luka):
  Si feuille :
      retourne le noeud qui corresond au hache de etiquette de arbre en utilisant la table
  Sinon :
      retourne un noeud avec etiquette de arbre et comme fils le parcours recursif sur les deux fils de arbre
```

- Mergin rule :

```
mergin_rule(table hache-noeud_hache, arbre Luka):
   verifier si le hache qui est etiquette du noeud est dans la table :
      retourne le noeud associe
   Sinon :
      Si feuille :
         on met le noeud dans la table avec comme cle son hache
         retourne le noeud
      Sinon :
         on cree un noeud avec etiquette arbre et comme fils appel recursif sur les fils de arbre
         on met le noeud dans la table avec comme cle son hache
         retourne noeud
```

- Deletion rule :

```
1   deletion_rule(table hache-noeud,table hache-noeud_hache, arbre Luka):
2       verifier si le hache qui est etiquette du noeud est dans la table
    hache-noeud_hache
3       retourne le noeud associe
4
5       Sinon :
6           Si feuille :
7               on met le noeud dans la table hache-noeud avec comme cle
    son hache
8               retourne le noeud
9
10          Si les hache des deux fils sont egaux :
11              Si les fils sont des feuille :
12                  retourne le fils gauche
13
14              Sinon :
15                  on cree un noeud avec etiquett du fils gauche et comme
    fils les fils du fils gauche et on fait un appel recursif sur le noeud
    cree et on ajoute dans la table  hache-noeud_hache avec comme cle le
    hache du noeud courant
16                  retourne noeud
17
18        Sinon :
19              on cree un noeud avec etiquette du noeud qui correspond au
    hache du noeud courant dans la table hache-noeud et comme fils les
    appels recursifs de la fonction sur les deux fils de arbre et on ajoute
    dans dans la table hache-noeud_hache avec comme cle le hache du noeud
    courant,
20              retourne noeud
```

4.2 Preuves formelles

La complexité du code implémenté en "C++" est prouvé dans cette section, car l'algorithme suit la démarche de l'énoncé.

4.2.1 Longueur du mot de Lukasiewicz

Le mot de Lukasiewicz d'un arbre de hauteur "h" est définit comme suit :

- Mot associé aux feuilles : "True", "False" dont on peut majorer la taille par 5, dans un arbre binaire complet nous avons 2^h feuilles.

- Un nombre de caractères fixe qui sont le"x", le chiffre qui identifie le noeud "C_h" et 4 parenthèses (arbre binaire complet) : $Taille_{noeudInterne} = 5 + C_h$

- Nous avons $(2^h - 1)$ noueds internes.

En reproupant les informations, on se retrouve alors pour un arbre de hauteur "h" comme suit :

$$Taille = (5 + C_h)(2^h - 1) + 2^h.5$$

En dévloppant la première équation on retrouve alors :

$$Taille = (10 + C_h)2^h - (5 + C_h)$$

4.2.2 Complexité pire cas de l'algorithme

- Nous pouvons définir la taille d'un mot au niveau "i" de l'arbre :

$$Taille_i = (10 + C_h)2^{h-i} - (5 + C_h)$$

- Le nombre de noeuds au niveau "i" d'un abre binaire complet :

$$Nombre_{noeud} = 2^i$$

- Nous comparons chaque noeud avec tous ses voisins du même niveau donc pour chaque noeud nous avons 2^{i-1} comparaisons.

En regroupant toute les informations et pour chaque niveau de l'abre : i allant de 0 à "h" on retoruve :

$$\sum_{i=0}^{h} 2^i.2^{i-1}.2^{h-i} * C_{st}$$

La constante représente toutes les valeurs constantes de la taille du mot de Lukasiewicz qu'on peut ignorer car ne dépendent pas de la hauteur.
On se retrouve alors avec :

$$\sum_{i=0}^{h} 2^{i+i-1+h-i}$$

En sortant le 2^{h-1} de la somme :

$$2^{h-1} \sum_{i=0}^{h} 2^i$$

On conclut alors : $2^{h-1}.2^{h+1} = 2^{2h}$
L'algorithme est alors en $O(2^{2h})$

4.2.3 Complexité en fonction de n

Dans un arbre binaire complet, nous avons $h = Log_2(n)$
En remplaçant dans l'équation trouvé à la question précédante :

$$2^{2*log_2(n)}$$

Avec les propriétés de la fonction logarithme :

$$2^{log_2(n^2)}$$

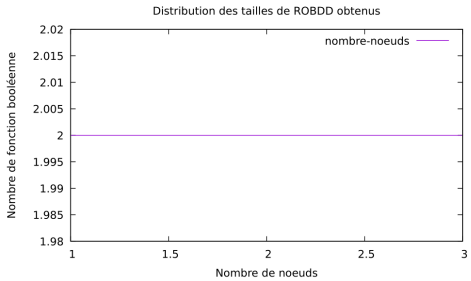
L'algorithme est alors en $O(n^2)$

5 Etude expérimentale

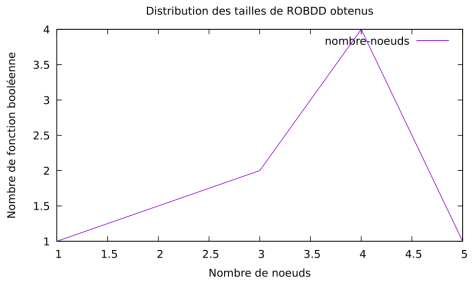
Nous avons reproduit les éxprémentations de l'article dans les deux versoins de notre code et nous avons eu les résultas suivants.

5.1 Reproduction de la figure 9

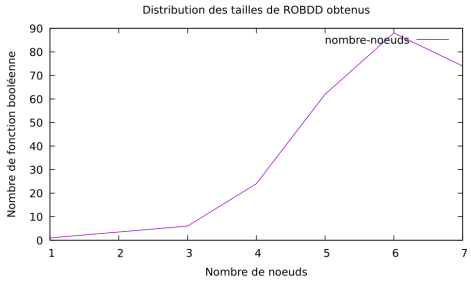
5.1.1 Résultats version C++



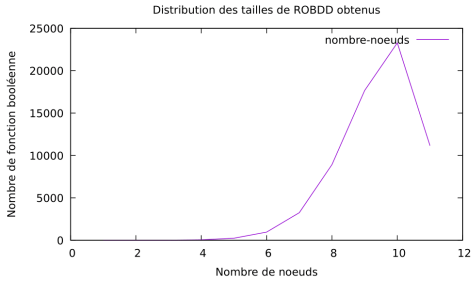
(a) Distribution pour 1 varible



(b) Distribution pour 3 varibles



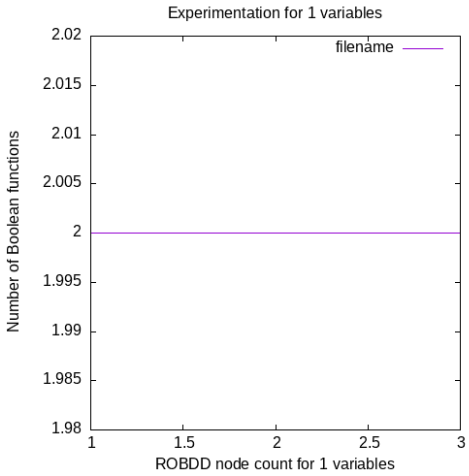
(c) Distribution pour 2 varibles



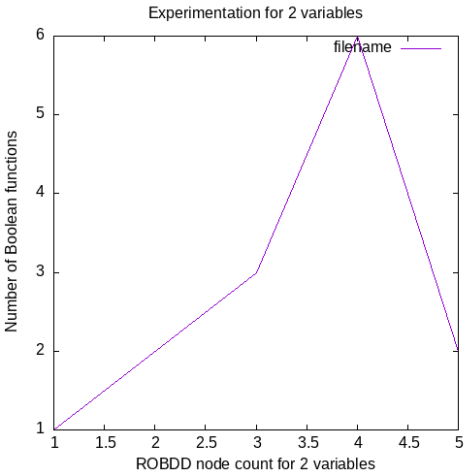
(d) Distribution pour 4 varibles

FIGURE 1 – Reproduction de la figure 9

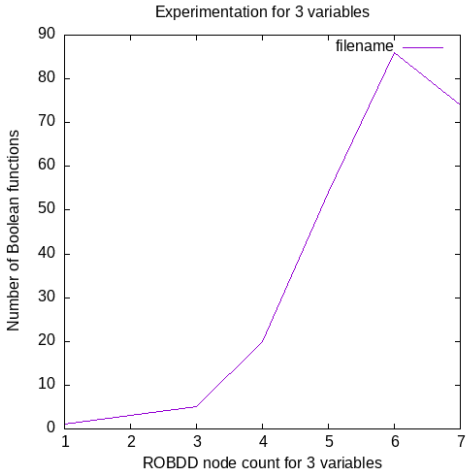
5.1.2 Résultats version Python



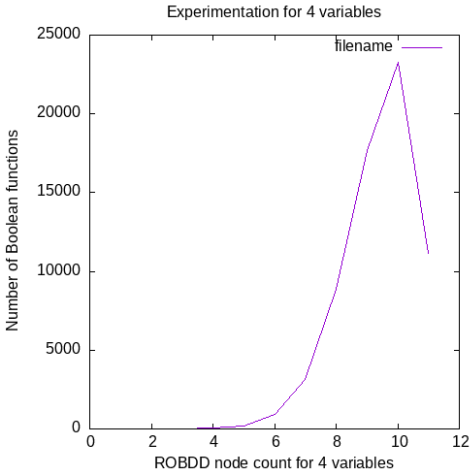
(a) Distribution pour 1 varible



(b) Distribution pour 3 varibles



(c) Distribution pour 2 varibles



(d) Distribution pour 4 varibles

FIGURE 2 – Reproduction de la figure 9

Nous Remarquons que pour une éxecution exhaustive sur des instances de 1 à 4 variables nous avons des résultats identiques à la figure présenté dans l'article de recherche.

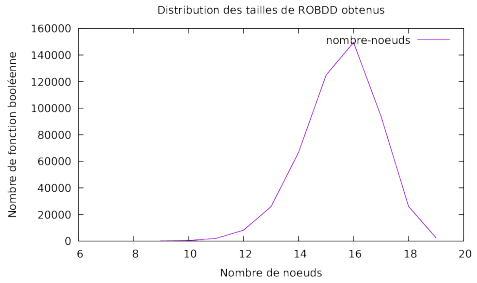
5.2 Reproduction de la figure 10

5.2.1 Résultats version C++

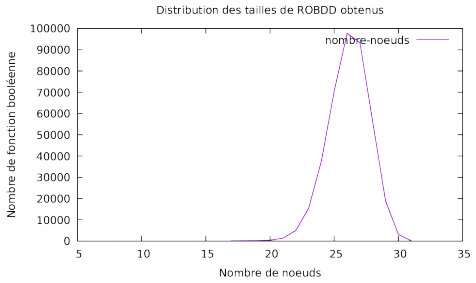
Dans cette version nous avons fait le choix de faire un échantillonnage par pas sur le nombre de valeurs utilisées dans le tableau de la –figure 11-, en appliquant l'équation suivante avec nb_v le nombre de variables.

$$Pas = (2^{2^{nb_v}} - 1)/nb_{echantillon}$$

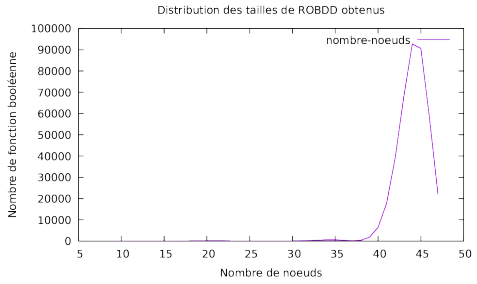
Et nous avons obtenus les figures suivantes :



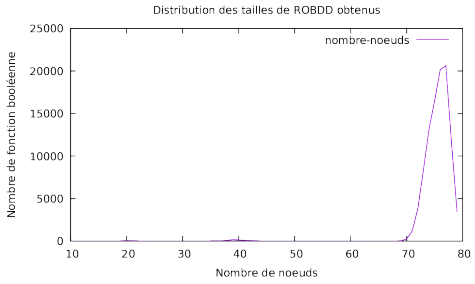
(a) Distribution pour 5 varibles



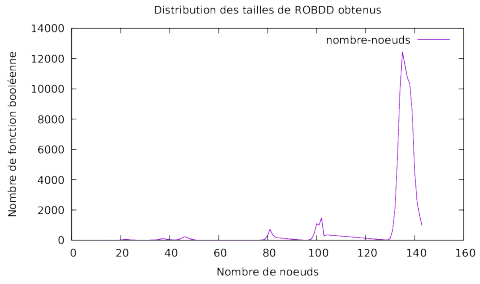
(b) Distribution pour 6 varibles



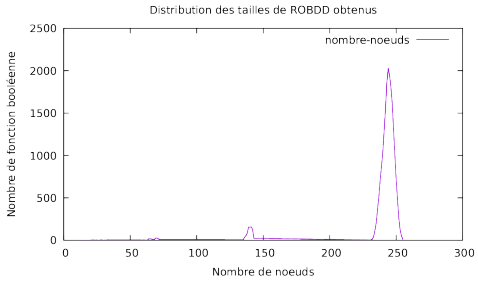
(c) Distribution pour 7 varibles



(d) Distribution pour 8 varibles



(e) Distribution pour 9 varibles



(f) Distribution pour 10 varibles

FIGURE 3 – Reproduction de la figure 10

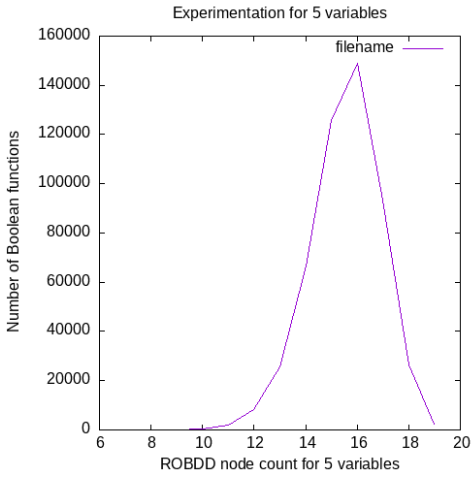
Remarque : Nous remaquons que les figures sont identiques à celles du rapport sur la concentration des tailles des ROBDD pour chaque nombre de variables, mais nous voyons aussi apparaitre quelques piques sur des tailles plus petites, et ce à cause du fait que notre nombre d'échantillons est très petit devant l'expérimentation exhaustive réaliser dans le document. Donc les petits clusters de taille ne sont plus négligeables

5.2.2 Résultats version Python

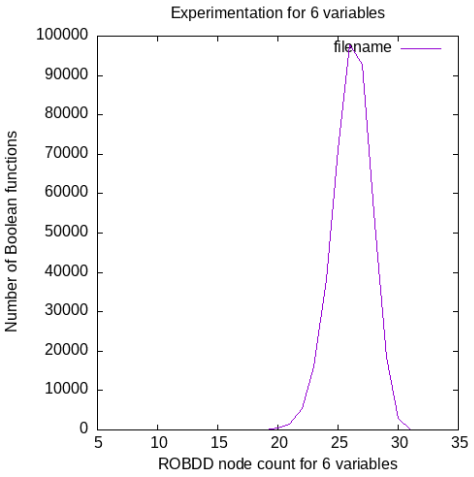
Dans cette version nous avons fait le choix de faire un échantillonnage par pas sur le nombre de valeurs utilisés dans le tableau de la figure 11, en suivant l'équation suivante :

$$Pas = ((2^{2^{nb_v}} - 1) - 2^{2^{nb_v - 1}})/nb_{echantillon}$$

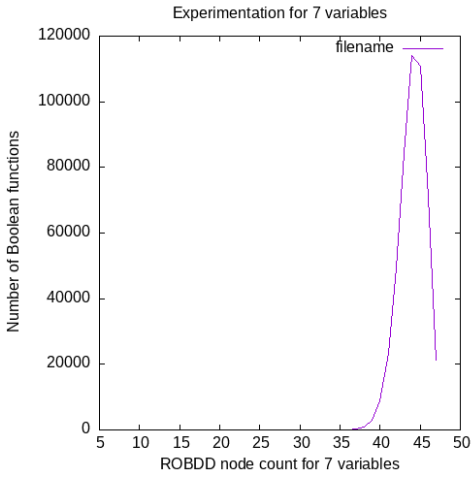
Nous testons alors uniquement sur des grands nombres s'écrivant sur $2^{2^{nb_v}}$ bits.
Nous avons ainsi obtenu :



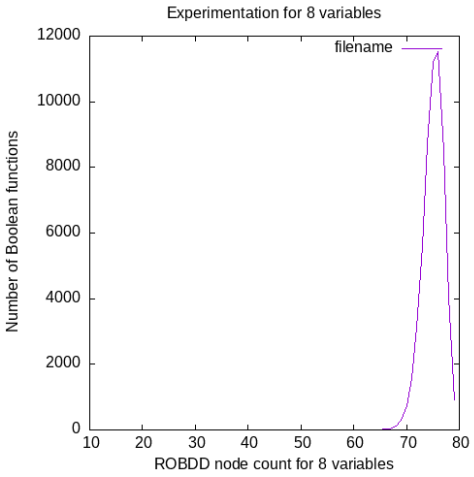
(a) Distribution pour 5 varibles



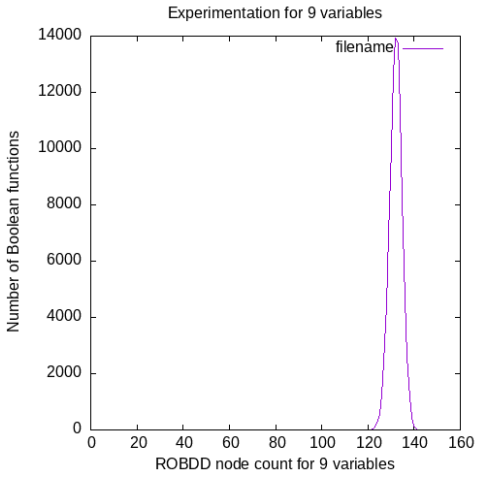
(b) Distribution pour 6 varibles



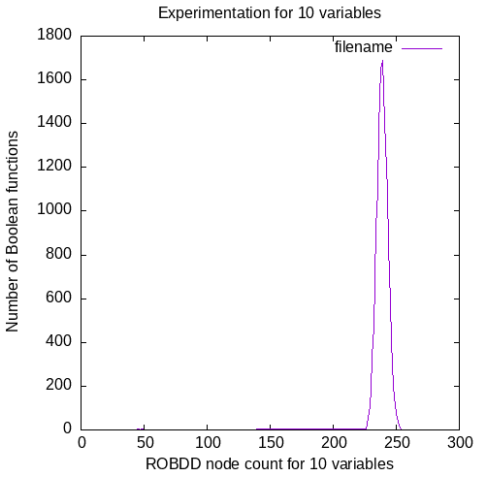
(c) Distribution pour 7 varibles



(d) Distribution pour 8 varibles



(e) Distribution pour 9 varibles



(f) Distribution pour 10 varibles

FIGURE 4 – Reproduction de la figure 10

Remarque : Nous remaquons que les figures sont identiques à celles du rapport sur la concentration des tailles des ROBDD pour chaque nombre de variable car nous avons trvaillé sur un interval plus restreint de valeurs à la différence de la première version.

5.3 Tableau des expérimentations

Nous vous présentons dans cette dernière section une représentation des deux tableaux résumant les expérimentations menées pour chaune des versions.

Nombre variables	Échantillon	Tailles uniques	Temps d'éxecution	Temps moyen d'un arbre
5	500000	14	0min:23sec:972ms	0.047944ms
6	400000	21	0min:42sec:533ms	0.106332ms
7	500000	32	2min:41sec:618ms	0.254045ms
8	100000	40	0min:55sec:67ms	0.55067ms
9	90000	70	1min:55sec:447ms	1.28274ms
10	20000	57	0min:56sec:777ms	2.83885ms

FIGURE 5 – Nombre d'échantillons et temps d'éxecution pour les graphes de la figure 3 (C++)

Nombre variables	Échantillon	Tailles uniques	Temps d'éxecution	Temps moyen d'un arbre
5	500003	13	03min:44sec	044803ms
6	400003	16	06min:29sec	0.97316ms
7	486892	25	16min:22sec	2.01720ms
8	56343	25	03min:29sec	3.71660ms
9	94999	26	13min:15sec	8.37591ms
10	17975	44	03min:59sec	13.35133ms

FIGURE 6 – Nombre d'échantillons et temps d'éxecution pour les graphes de la figure 4 (Python)

Les remarques :

— Nous avons un temps d'éxecutions plus rapide pour le code en C++ par rapport à python (∗8).

— La différence des intervalles d'échantillonnage fait en sorte que les tailles uniques de la première version soient supérieures à celle de la seconde car le spectre des valeurs testées est plus grand.

— Le deuxième tableau est celui qui se rapproche le plus des réusltats obtenu dans l'article de recherche.

6 Conclusion

— Nous pouvons toujours améliorer la version "C++" en utilisant des tables de hachage indexées par la taille du sous-arbre, pour avoir tous les noeuds de même taille dans la même liste et ainsi parcourir moins d'éléments.
— La version développée en "Python" est la plus optimale qu'on a pu imaginer car en complexité $O(n)$, mais on sacrifie de l'espace mémoire pour stocker les différentes tables.
— Nous aurions dû tester d'autres types d'échantillonnage pour parcourir plus de valeurs, mais la technique utilisée dans la version "C++" permet d'avoir le plus grand spectre possible.
— Travailler avec le même langage de programmation aurait permis de mettre en place une comparaison expérimentale des deux versions pour voir le gain réel de l'implémentation optimale.