KAYA Delphine

Rapport Projet Algorithme Avancée

Le projet est implémenté en python et contient 4 parties dont la dernière concerne la partie expérimentation.

Partie Echauffement :

Dans cette première partie, on retrouve les fonctions suivantes :

- **decompostion(x)** : décompose le nombre entier **x** en liste de booléen qui représente la liste de bits en base 2 de cet entier.
  decomposition(38) == [False, True, True, False, False, True]
  decomposition(89) == [True, False, False, True, True, False, False, True]
  decomposition(0) == []

- **completion(l, n)** : prend la liste de bits **l**. Si sa taille est inférieure à l'entier **n**, alors elle retourne les n premiers éléments de **l**. Sinon, elle complète **l** pour qu'elle soit de taille **n**, elle rajoute alors si besoin des False en bout de liste.

- **table(x, n)** : réunit les deux fonctions précédentes pour renvoyer la liste de bits en base 2 du nombre **x** ayant une longueur **n**.

Remarques : la fonction **decomposition** raise une erreur si le nombre à traiter n'est pas un entier naturel, c.à.d. si **n < 0**.

Partie Arbre de décision et compression :

Tout d'abord, nous implémentons une structure qui va nous permettre de manipuler des arbres de décision. Pour cela, on crée une classe **Abr**.

Celle-ci est définie telle que chaque nœud contient les attributs racine, fils gauche et fils droit. De plus, la classe fournit notamment la méthode **nœud(self)**. Celle-ci renvoie la description d'un arbre de décision dont la racine/tête est **self** sous la forme d'une liste.

Puis, en ce qui concerne l'implémentation des fonctions :

- **cons_arbre(T) :** prend en entrée une table de vérité **T** (tel le résultat de la fonction ***table(x, n)***) et retourne l'arbre de décision qui correspond. Pour cela, **cons_arbre** utilise la structure **Abr** et sa fonction ***nœud()***.

cons_arbre(table(8, 4)) == [‘x2’, [‘x1’, False, False], [‘x1’, False, True]]

- **luka(arb) :** prend en entrée la sortie de la fonction précédente, c.à.d. un arbre de décision et l’enrichit avec les mots de Lukasiewicz. Pour cela, elle appelle la fonction récursive ***racine_luka*** sur la tête de l’arbre **arb**.

- **racine_luka(A) :** associe un mot de Lukasiewicz à chaque nœud de l’arbre **A** et renvoie la chaine de caractère de son mot de Lukasiewicz (Hyp : la taille du nœud n’excède jamais 3, s i l’arbre provient de ***cons_arbre***, il n’y a pas de problème).
racine_luka(cons_arbre(table(8, 4))) == ‘x2(x1(False)(False))(x1(False)(True))’

- **luka_table(T) :** effectue la même opération que ***luka***, mais elle prend en paramètre une table de vérité **T**. Elle construit l’arbre de décision et l’enrichit avec les mots de Lukasiewicz en même temps.

- **list_racine(arb) :** prend un arbre enrichi ou non des mots de Lukasiewicz et renvoie la liste des racines de l’arbre **arb**.

- **racine_count(lr) :** prend une liste **lr** de racines et renvoie un dictionnaire avec pour clés les noms de racines et pour valeurs leur nombre d’occurrences dans l’arbre (c.à.d. dans la liste **lr**).

- **compression(luka, dico) :** Une fonction récursive qui, depuis un arbre de décision enrichi des mots de Lukasiewicz **luka**, construit un arbre compressé, c.à.d. tel que les sous arbres communs sont fusionnés. Le dictionnaire **dico** est vide au premier appel, et est rempli au fur et à mesure des noms de racines d’origines (x1, x2…) avec leur nombre d’occurrences et les noms de racines Lukasiewicz associés à leur nouveau nom de racine numéroté.
compression(luka(cons_arbre(table(8, 4)))) == [‘x21’, [‘x11’, False, False], [‘x12’, False, True]]
compression(luka(cons_arbre(table(0, 4)))) == [‘x21’, [‘x11’, False, False], [‘x11’, False, False]]

- **dot(arb, compresse) :** prend un arbre de décision **arb** (enrichi ou non des mots de Lukasiewicz) et renvoie un dot qui va permettre ensuite, avec la fonction *render()* (de graphviz) d’afficher le graphe de l’arbre. La fonction **dot** initialise un nouveau dot, lui ajoute le premier nœud qui est la racine de **arb** puis appelle la fonction **node_lien_dot** sur cette même tête **arb** qui se charge d’ajouter récursivement les sous nœuds et les liaisons de l’arbre en entier. En fonction de la valeur du paramètre booléen ***compresse***, la fonction va compresser le graphe en ne gardant qu’un seul

nœud unique en cas de redondance et en reliant les liens vers cet unique nœud (utile après pour la **compression_bdd**).

Remarques : Avec l'implémentation de notre fonction **dot**, on n'est pas obligé de passer par la fonction **compresse** pour fournir un dot représentant un arbre compressé. Il suffit de créer un arbre « normal » avec **cons_arbre**, de créer son arbre enrichi des mots de Lukasiewicz à l'aide de la fonction **luka**, et donc de lancer l'opération « **dot(luka(cons_arb(x,n)), True)** ». En indiquant **compresse == True** dans notre appel, la fonction **dot** va elle-même se charger de créer un graphe compressé, en détectant les redondances de nœuds identiques. Elle va alors créer des instances uniques pour chaque nœud et relier tous les liens vers ces nœuds identiques vers ce seul unique nœud dans le graphe.

Partie Arbre de décision et ROBDD :

- **compression_bdd(luka) :** Cette fonction de compression prend en paramètre un arbre enrichi des mots de Lukasiewicz **luka** et renvoie un nouvel arbre qui reprend le premier mais qui le compresse. En d'autres termes, **compression_bdd** copie **luka**, mais lorsqu'il voit que pour un de ses sous-nœuds **k**, il a **deux sous-fils identiques**, il remplace ce nœud **k** par la valeur de ses fils. On obtient alors un ROBDD.
  Il faut tout de même faire attention au nombre de variables de départ, par exemple, on pourrait avoir :
  compression_bdd(luka(cons_arbre(table(0, 4)))) == False
  Alors qu'avec :
  cons_arbre(table(0, 4)) == ['x2', ['x1', False, False], ['x1', False, False]]
  On a deux variables x1 et x2 au départ.

  compression_bdd(luka(cons_arbre(table(8, 4)))) == ['x2(False)(x1(False)(True))', False, ['x1(False)(True)', False, True]]
  On peut voir que le nœud **(x1(False)(True**)) est répété deux fois. La compression des nœuds redondants sera faite soit avec la fonction **dot** (avec son paramètre ***compresse*** à **True**) soit avec la fonction **compresse.**

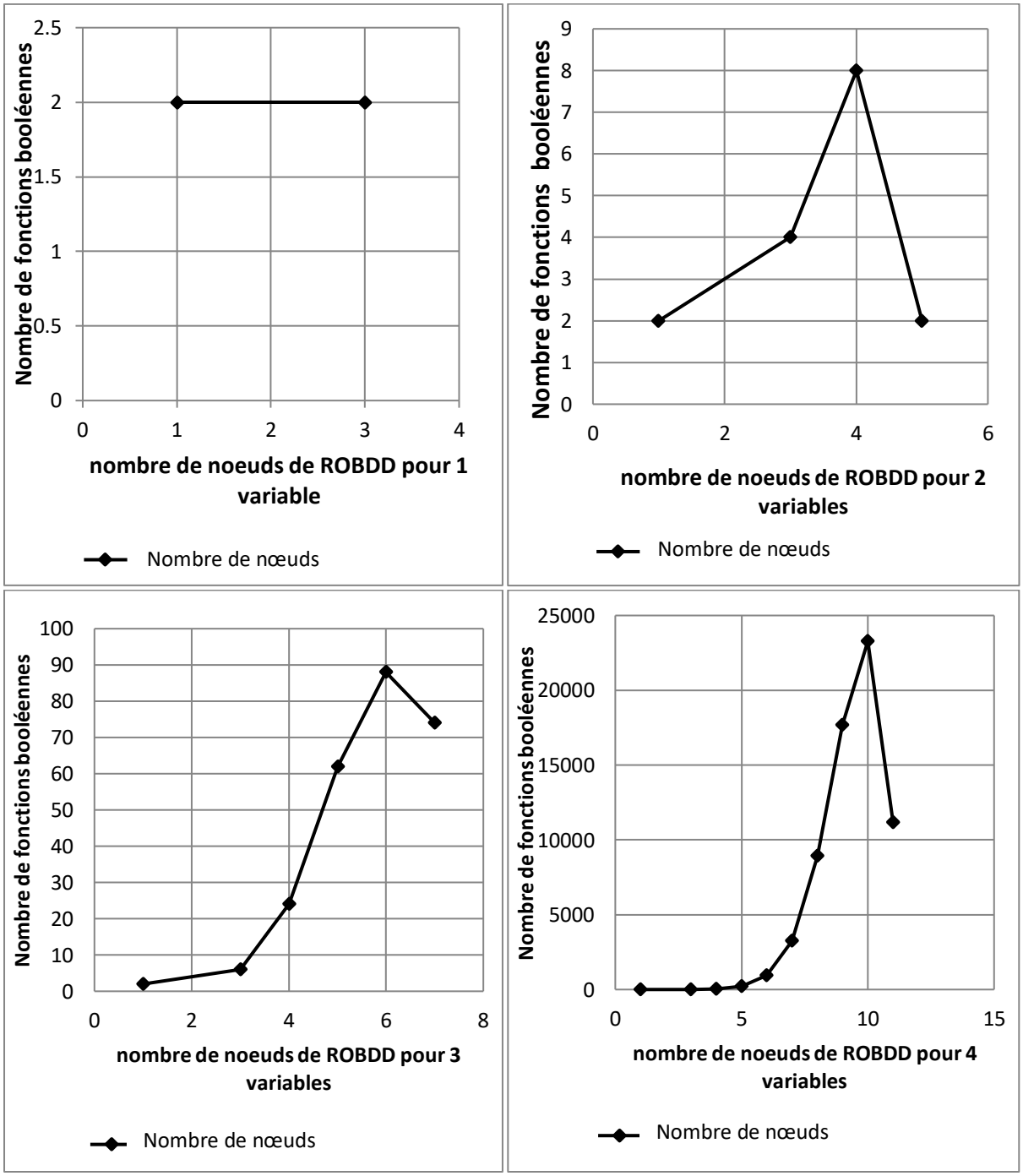
Partie Etude Expérimentale :

Dans cette dernière partie, on cherche à comparer nos résultats aux résultats décrits dans l'article « *A Theoretical and Numerical Analysis of the Worst-Case Size of Reduced Ordered Binary Decision Diagrams*».

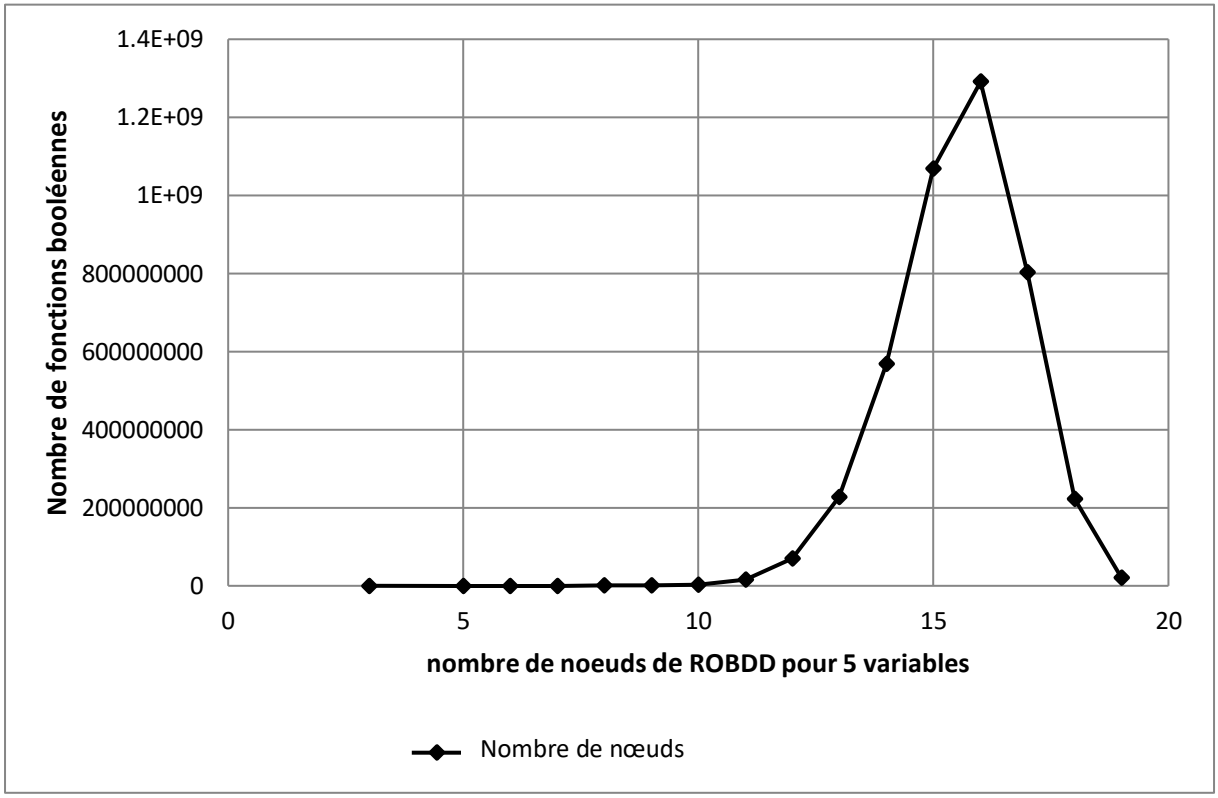
Ainsi on compare à chaque fois nos valeurs aux figures 9, 10 et 11 de l'article.

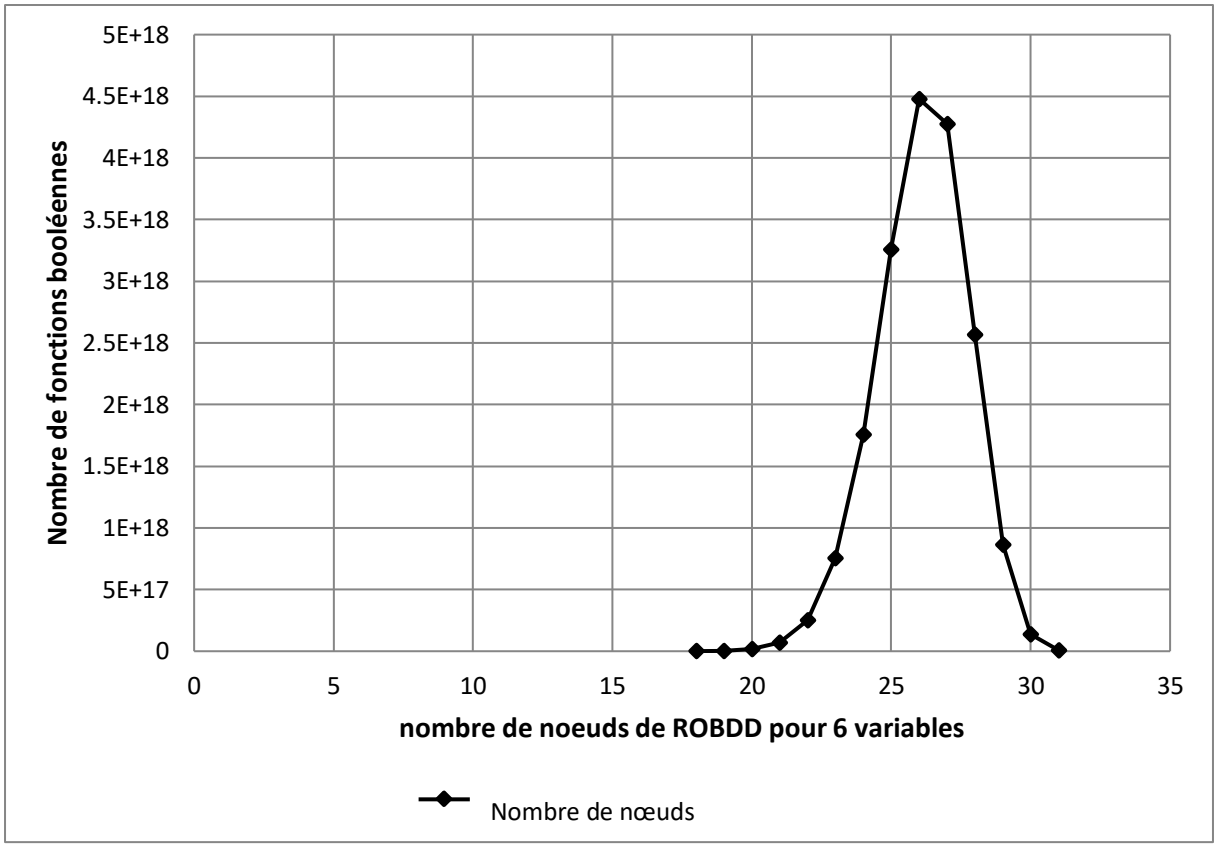
Pour la figure 9, on génère tous les graphes possibles en fonction du nombre de fonctions Pour cela, on récupère la structure de l'arbre compressé à l'aide de la fonction **compression_bdd** (sous forme d'une liste arborescente) et on l'aplatit pour pouvoir compter son nombre de nœuds. On rappelle que pour les graphes à **n** variables, il y a 2^(2^**n**) fonctions booléennes possibles.

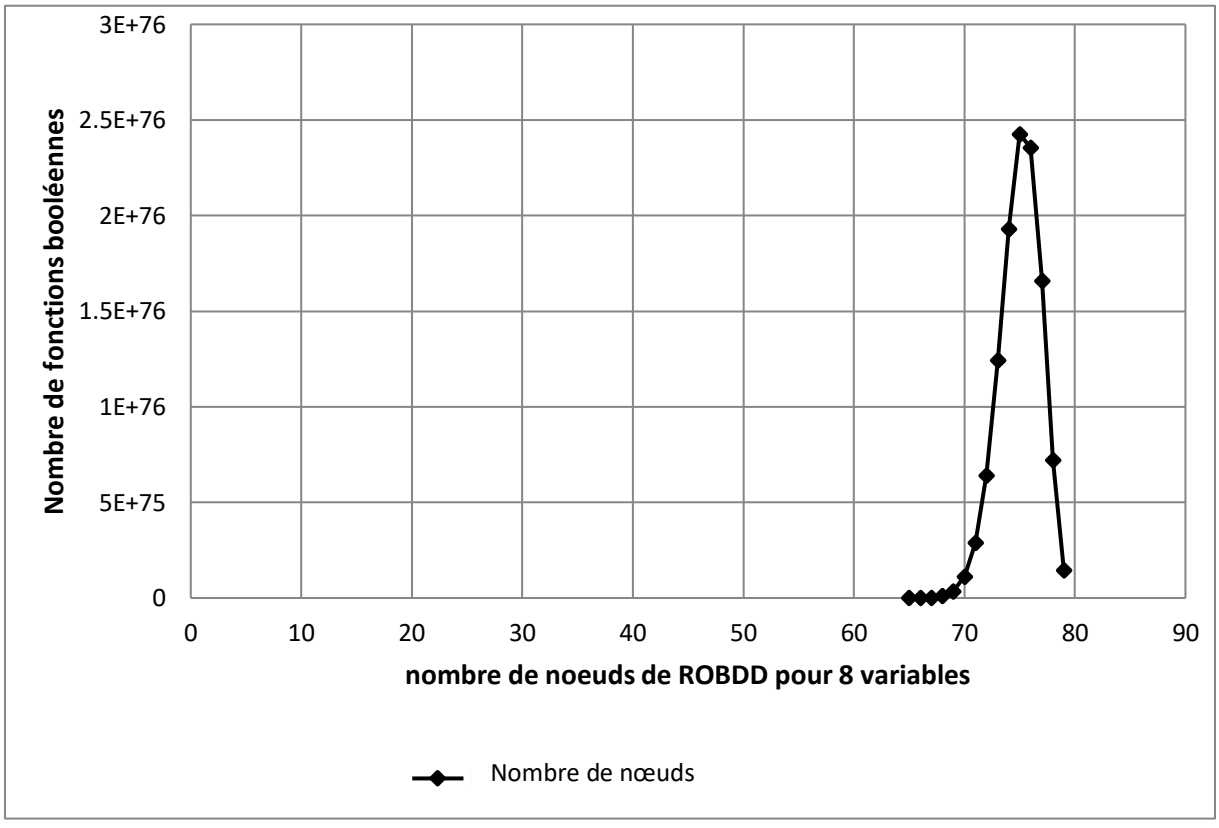
Avec notre expérimentation sur les graphes de variables allant de 1 à 4, on observe que l'on obtient exactement les mêmes nombres de nœuds que dans l'article:

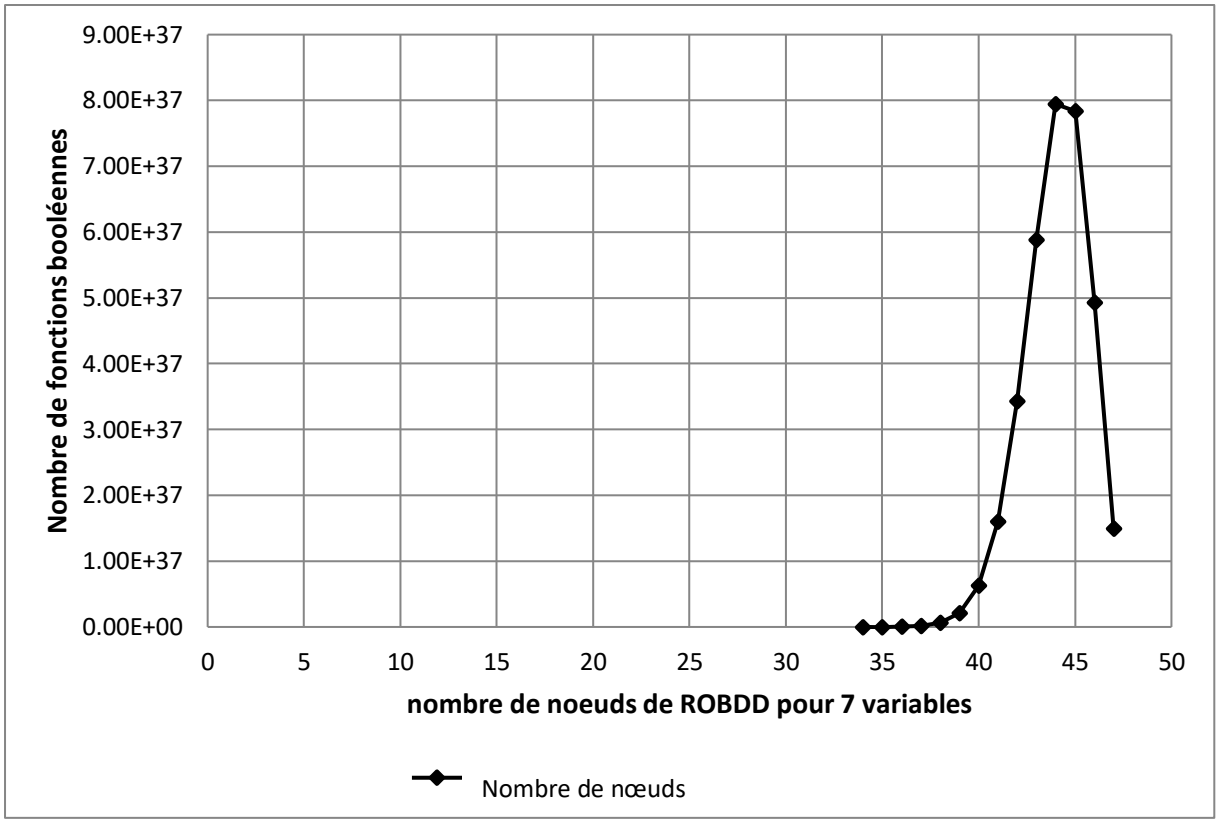


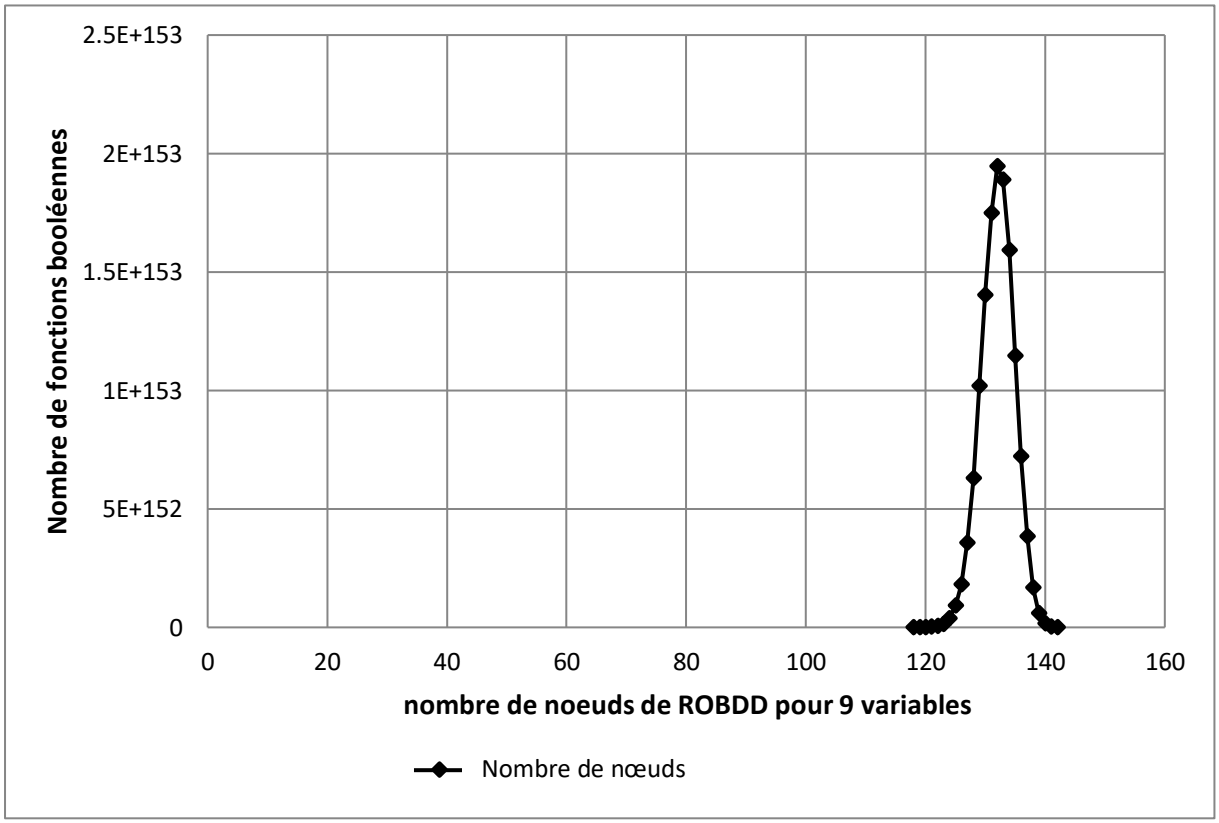
Pour la figure 10, avec le nombre de variables allant de 5 à 10, sachant que l'expérimentation traite la **compression_bdd** sur une partie des arbres à variables **n** et qu'on multiplie par un facteur pour « combler le manque ». Il s'agit donc d'extrapoler pour ces valeurs.

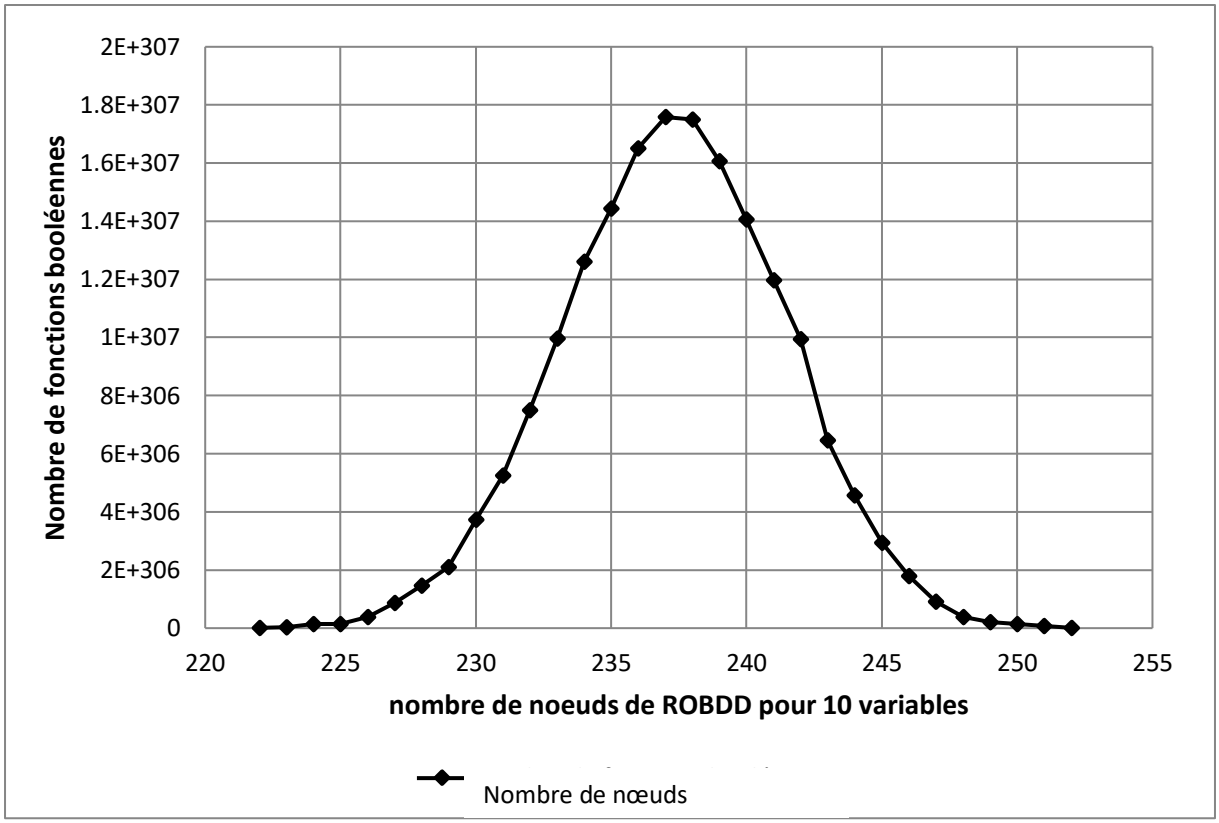



5E+18
4.5E+18
4E+18
3.5E+18
3E+18
2.5E+18
2E+18
1.5E+18
1E+18
5E+17
0
Nombre de fonctions booléennes
0
5
10
15
20
25
30
35
nombre de noeuds de ROBDD pour 6 variables
Nombre de nœuds


3E+76
2.5E+76
2E+76
1.5E+76
1E+76
5E+75
0
Nombre de fonctions booléennes
0
10
20
30
40
50
60
70
80
90
nombre de noeuds de ROBDD pour 8 variables
Nombre de nœuds


9.00E+37
8.00E+37
7.00E+37
6.00E+37
5.00E+37
4.00E+37
3.00E+37
2.00E+37
1.00E+37
0.00E+00
Nombre de fonctions booléennes
0
5
10
15
20
25
30
35
40
45
50
nombre de noeuds de ROBDD pour 7 variables
Nombre de nœuds


2.5E+153
2E+153
1.5E+153
1E+153
5E+152
0
Nombre de fonctions booléennes
0
20
40
60
80
100
120
140
160
nombre de noeuds de ROBDD pour 9 variables
Nombre de nœuds



On remarque sur les graphes que les nombres de nœuds trouvés sont toujours proches des nombres de l'étude faite dans l'article.

En ce qui concerne les temps d'exécution, nous avons repris la figure 11 de l'article à laquelle nous avons ajouté les graphes de la figure 9 également.

Nombre de variables	Nombre d'échantillons	Nombre de tailles uniques	Temps d'exécution (en secondes)	Secondes par ROBDD
1	4	2	7,16E-05	1,79E-05
2	16	4	0,000657764	4,11102E-05
3	256	6	0,02196176	8,57881E-05
4	65536	16	12,8578897	0,000196196
5	500003	14	162,05075	0,0003241
6	400 003	14	366,8953	0,000917231
7	486 892	16	1372,3693	0,002818632
8	56 343	17	550,28207	0,009766645
9	94 999	25	4426,6105	0,04659639
10	17 975	35	3741,4835	0,208149291

On peut voir qu'avec l'implémentation du projet, le calcul du nombre de nœuds dans un ROBDD n'excède jamais une seconde. Pour un ROBDD à 10 variables, le temps moyen de calcul est 0.21 secondes. Ce n'est rien comparé au résultat de l'article pour lequel le même calcul prenait en moyenne une minute.