SCIENCES SORBONNE UNIVERSITÉ

Sorbonne université
Projet d'ALGAV

Algorithmique Avancée

Maxime DERRI & **Yaël ZARROUK**

Année : 2022 - 2023

Table des Matières

1 | Présentation

L'objectif du projet consiste à générer des diagrammes de décision binaires, réduits et ordonnés avec une approche analogue à celle présentée dans l'article de Newton et Verna.

Pour représenter ces arbres, nous avons choisi d'utiliser le langage de programmation C++ et Python.

2 | Organisation des structures utilisées

Dans notre code, les arbres de décisions sont organisés avec deux classes principales :

— La classe "node" : contenant comme attributs deux pointeurs vers des "node *" (fils G/D), un string pour le mot de Lukasiewicz (construit par appel à une fonction), un entier qui sert à stocker le numéro de variable et deux booléens.
Un premier booléen qui sert de marqueur pour ne pas parcourir les DAG/ROBDD plusieurs fois, et le deuxième qui sert à indiquer quels noeuds sont à supprimer ou non après la transformation DAG -> ROBDD ("mark and sweep").

— La classe "tree" : qui sert essentiellement de stockage du noeud principal, et qui indique également quelques autres informations telles que :
  — le nombre de variables
  — un booléen qui indique dans quelle situation se trouvent les booléens des "node" (celui pour le parcours)
  — un autre booléen qui sert à indiquer à la construction du mot de Lukasiewicz s'il faut stocker les mots pour chaque noeud ou seulement le principal (gain de place)

  Aussi, notre fonction qui transforme l'arbre de décision en DAG n'utilise que le mot du noeud principal : on le décompose à chaque appel récursif), un autre booléen qui est utilisé pour indiquer une suite de noeud à supprimer après la transformation en ROBDD.

— La classe "result" : qui sert de stockage des résultats. Elle garde de côté les secondes et millisecondes calculées, le nombre de variables des ROBDD analysées, ainsi qu'un vecteur de pair (correspondance nombre de noeuds - fonction booléennes).

Nous utilisons également des structures de données du langage C++ :

- Les "vector" : un conteneur classique permettant des accès en $\mathcal{O}(1)$, car définit par un tableau qui augmente de taille quand le maximum de ce tableau est atteint (allocation d'un nouveau tableau - recopie - suppression de l'ancienne zone mémoire).

- Les *unordered_set* : des tables de hachage de type set (ne contient que des clés uniques).

La table est organisée comme suit : un tableau qui contient des buckets (on trouve l'entrée correspondante avec une fonction de hachage). Chaque bucket contient une liste chainée qui sert à résoudre les collisions.

- Les "queue" : qui permettent l'implémentation classique d'une structure FIFO (push/-pop/empty,...). Utilisée pour la gestion de la mémoire, nous y reviendons rapidement.

Ces structures ne seront pas plus détaillées, du fait que l'implémentation est réalisable plutôt facilement (parcours de tableau, utilisation de liste chainées).

Pour les fonctions de hachage il existe cependant plusieurs implémentations possibles (utilisation de XOR et du modulo par exemple).

3 | Gestion de la mémoire

Pour revenir sur la gestion de la mémoire rapidement cité dans l'organisation des structures, nous avons fait le choix de ne pas utiliser les *unique_ptr* et *shared_ptr*. Nous voulions pouvoir contrôler l'allocation et la désallocation.
Il faut cependant noter que ces objets permettent une gestion efficace de la mémoire du aux contrôles sur les objets pointés : un compteur est utilisé, et arrivé à 0, le destructeur est appelé.

Concernant notre cas, nous faisons un parcours en largeur des "node *" (avec "une queue") et on insère dans un vecteur chaque noeud (si le noeud n'a pas encore été inséré, on utilise un booléen). Quand le parcours est fini, on itère sur le vecteur et on appel le destructeur. En effet, les DAG et ROBDD ayant des noeuds possiblement référencés plusieurs fois par des pointeurs, faire un destructeur classique poserait des problèmes de mémoire (segfault...).

Nous avons donc pris la décision de redéfinir l'opérateur "delete" de la classe "node".

4 | Algorithmes importants

4.1 Primitives

$fin(string, char)$: true si la chaine se fini par le char

$compter(string, char)$: retourne le nombre d'occurence de char dans string

$debut\_gauche(string)$: retourne l'index du debut, du sous mot de gauche

$debut\_droit(string)$: symétrique de $debut\_gauche$, mais pour la droite

$taille(string)$: retourne le nombre de caractère

$couper\_mot(string, ind1, ind2)$: retourne la sous chaine entre les ind1 et ind2

$luka(node)$, $variable(node)$, $valeur\_verite(node)$, $parcours(node)$: attributs node()

$node(var, verite, parcours)$: constructeur

$vector()$, $queue()$: constructeur

$push(fifo, val)$: ajouter val à la fifo

$pop(fifo)$: retirer un élément de la fifo

$tete(fifo)$: retourne la tête sans la supprimer

$estVide(struct)$: true si aucun éléments

$filsGauche(node)$: retourne un pointeur vers le sous noeud gauche

$filsDroit(node)$: symetrique de filsGauche(node) pour la droite

$ajoute(struct, elt)$: ajoute elt dans la structure

$ajoute(hash_set, elt < cle, val >)$: ajoute elt si possible, retourne <elt,true> si inséré, sinon <elt_deja_present,false> où elt_dej_present référence une pair <cle,val>

$second(< cle, val >)$: retourne val et premier(<cle,val>) retourne clé

$marque(node)$: true si le node est marqué

$marquer(node)$: marque le node

liberer$Memoire(elt)$: libère la memoire

4.2 Arbres de décision vers DAG

```
to_dag(hash_set<mot_luka, node> map, string mot, node n):
string g,d
<mot_luka, nod> tmp # <cle,val>

        si filsGauche(n) == NULL || filDroit(n) == NULL):
            retourner 0 # plus rien à traiter

        si couper_luka(mot, g, d) == -1:
            retourner -1 # probleme
        #gauche
        tmp = ajouter(map, <g, filsGauche(n)>):
        #ajoute une entrée si la clé 'g' n'est pas présente (utilise une fonction de hachage)
        si second(tmp) #true -> insertion
            to_dag(map, g, filsGauche(n)
        sinon:
            si @filsGauche(n) != @(second(premier(tmp)):
            # les adresses des noeuds sont différentes | prend le noeud dans tmp
                delete(filsGauche(n)) #suppression
                filsGauche(n) = second(premier(tmp))#change la reference

        #droit
        tmp = ajouter(map, <d, filsdroit(n)>):
        #ajoute une entrée si la clé 'd' n'est pas présente (utilise une fonction de hachage)
        si second(tmp) #true -> insertion
            to_dag(map, d, filsDroit(n)
        sinon:
            si @filsDroit(n) != @(second(premier(tmp)):
            # les adresses des noeuds sont différentes
                delete(filsDroit(n)) #suppression
                filsDroit(n) = second(premier(tmp))#change la reference
        retourner 0
#fin
```

4.3 DAG vers ROBDD

```
to_robdd_aux(node n, vector<node> tab,  bool w): # effectue la transformation
    si filsGauche(n) == NULL || si filsDroit(n) == NULL
        retourner # rien n'est à retourner

#parcourir le DAG
    si filsGauche(filsGauche(n)) == NULL || filsDroit(filsGauche(n)) == NULL:
        retourner
    sinon:
        to_robdd_aux(filsGauche(n), tab, w)

    si filsDroit(filsDroit(n)) == NULL || filsGauche(filsDroit(n)) == NULL:
        retourner
    sinon:
        to_robdd_aux(filsDroit(n), tab, w)

# gauche
    si filsGauche(filsGauche(n)) == filsDroit(filsGauche(n)):
        si parcours(filsGauche(n)) != w:
            parcours(filsGauche(n)) = w
            ajoute(tab, filsGauche(n))
        filsGauche(n) = filsGauche(filsGauche(n))

# droite
    si filsDroit(filsDroit(n)) == filsGauche(filsDroit(n)):
        si parcours(filsDroit(n)) != w:
            parcours(filsDroit(n)) = w
            ajoute(tab, filsDroit(n))
        filsDroit(n) = filsDroit(filsDroit(n))

#fin
```

4.4 DAG vers ROBDD

```
to_robdd(node n, bool w): # w indique la valeur contraire du boolean walk/parcours du code
    si filsGauche(n) == NULL && filsDroit(n) == NULL: # rien à faire
        retourer # rien n'est retourné

    vector<node> tab # noeud à suppr
    to_robdd_aux(n, tab, w) #traitement
    node r = NULL

    #traitement de la racine
    si  filsGauche(n) == filsDroit(n): # il faut réduire
            varible(n) = variable(filsGauche(n))
            verite(n) = verite(filsGauche(n))
            luka(n) = ""

            si filsGauche(filsGauche(n)) != NULL:
                r = filsGauche(n)
                filsGauche(n) = filsGauche(filsGauche(n))
                delete(filsGauche(r))

            si filsDroit(filsDroit(n)) != NULL:
                r = filsDroit(n)
                filsDroit(n) = filsDroit(filsDroit(n))
                delete(filsDroit(r))

            si r == NULL: #n'a pas été modif
                delete(filsGauche(n))
                filsGauche(n) = NULL
                filsDroit(n) = NULL
            sinon:
                delete(r)

    #nettoyage pour toutes les entrées elt de tab:
        filsGaucche(elt) = NULL # ne pas delete les autres noeuds !!!
        filsDroit(elt) = NULL # pareil
        delete(elt)
#fin
```

4.5 Décomposition du mot de Lukaziewicz

```
couper_luka(string src, string g, string d): # coupe src en deux et place dans les pointeurs
    si !fin(src,')') # pas correct
        retourner -1

    si compter(src, '(') != compter(src, ')'): # probleme !
        retourer -1

    int a,b # index x(a)(b)
    int c # index de fin

    a = debut_gauche(src) # x(a...
    b = debut_droit(src) # x(...)(b...
    c = taille(src)
    l = couper_mot(src, a, b-2)
    # prend le premier couple de parenthèse x(ici)(autre) | b-2 pour excluse les parenthèses:
    r = couper_mot(src, b, c-1) # extrait de b à la fin (privé de la dernière parenthèse)

    retourner 0
# fin
```

4.6 Copie d'une arborescence de noeuds

Remarque : l'algorithme ne vérifie pas s'il est déjà passé sur un noeud :
- il peut donc décompresser un DAG
- il peut servir pour décompresser un ROBDD (cependant, les noeuds/variables qui ont été supprimés ne sont pas ajoutés)

```
copier_noeud(node dst, node src): # copier la source src dans la destination dst | ce sont
des pointeurs donc pas besoin de les retourner
        si src == NULL:
            retourner -1 # erreur
        dst = node(variable(src), valeur_verite(src), parcours(src))
        # parcours correspond au champ walk dans notre code - pas important ici
       # valeur de verite est pris en compte dans le reste que si c'est une
       # feuille, mais besoin pour copie...
        luka(dst) = luka(rsc)

        si filsGauche(src) != NULL:
            copier_noeud(filsGauche(dst),filsGauche(src))
        si filsDroit(src) != NULL:
```

```
            copier_noeud(filsDroit(src), filsGauch(src))

        retourner 0 #fin correct
```

4.7 Opérateur delete (pour expliquer plus en détail l'approche)

```
operator delete(node n): # supprimer un noeud et son arborescence de la mémoire
    si n == NULL:
        retourner

    vector<node> map = vector()
    queue<node> fifo = queue()
    node cur = node()
    push(fifo, n)

    tant que !estVide(fifo):
        cur = tete(fifo)
        si filsGauche(cur) != NULL:
            push(fifo, filsGauch(cur))
        si filsDroit(cur) != NULL:
            push(fifo, filsDroit(cur))
        si !marque(cur):
            marquer(cur)
            ajoute(map, cur)

    pour toutes les entrées elt dans map:
        libererMemoire(elt)
```

5 | Extensions

Nous avons implémenté dans notre programme de la concurrence, avec une pool de thread.
- La classe "job" définit les tâches que les threads devront traiter.
- Redéfinition d'une classe "queue", pour s'adapter à notre implémentation (surtout pour contrôler les data race).
- Une classe "pool" qui contient la queue de job, les threads, et permet aux threads de traiter leurs tâches.

Pour les data race, nous utilisons des mutex pour les sections critiques ainsi que des $condition\_variable$ pour faire patienter les threads et les débloquer avec des $notify\_all$.

La concurrence intervient (dans le cas où elle est configurée dans le Makefile) pour l'étude des ROBDD des figures 9, 10 et 11 :

Plutôt que de traiter une étude à la fois (arbre -> DAG -> ROBDD -> analyse), le programme en lancera plusieurs en nourrisant la queue (d'un objet pool) avec un nouvel objet job avec les informations qui lui sont nécessaires (le nombre de variables, la valeur utilisée pour obtenir la table de véritée, un booléen pour forcer la création des graphes avec GraphViz, ainsi qu'un objet "result").

Remarque : result à été protégé de la concurrence par des mutex.

Nous n'utilisons pas de concurrence pour la création de ROBDD uniques, car nous avons besoins des compressions précédentes pour traiter les futures.
Il aurait été possible d'utiliser des threads, mais cela force à faire plusieurs parcours pour lier le tout. Cela pourrait augmenter la vitesse, mais peut-être de pas beaucoup.

Nous aurions voulu nous intéresser à paralléliser notre code dans une nouvelle version avec un code utilisant les ressources GPU, mais il y a deux points :

- Nous ne savons pas si cela est possible. De plus, il est possible que cela impacte encore plus la RAM (pour beaucoup de variables) si nous traitons plus de constructions à la fois.
- Nous n'avons malheureusement pas eu le temps de réaliser des essaies.

L'idée aurait été d'utiliser CUDA (pour GPU nvidia - l'un de nos PC en est equipé).

6 | Arbres de décision et ROBDD

6.1 Question 3.11

On veut prouver que $l_h = (10+c_h)2^h - (5+c_h)$. On note h la hauteur de l'arbre et c_h le nombre de caractères des chiffres des variables (par exemple si $c_h = 2$, on va être borné jusqu'à 99).
Les feuilles peuvent prendre soit la valeur $True$, soit la valeur $False$. Chaque noeuds internes : $|"x" + c_h + "(" + ")" + "(" + ")"| = 5 + c_h$.

- $2^h - 1$ noeuds internes :
$h = 0$: donc l'arbre est réduit à 1 seul noeud.
$h = 1$: donc l'arbre a 2 feuilles.

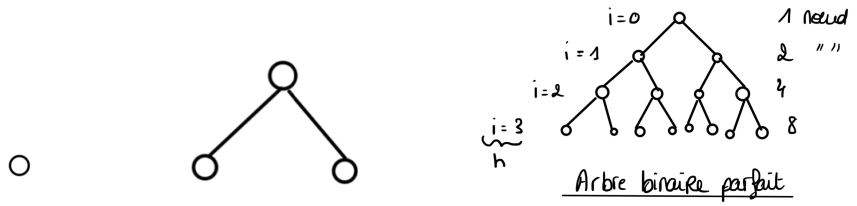

FIGURE 6.1 – La hauteur d'un arbre diffère en fonction du nombre de noeuds qu'il possède

Donc, si chaque nouvel étage $i+1$, on double le nombre de noeuds qu'il y avait à l'étage i, pour $i \geq 0$, on a : $2^0 + 2^1 + 2^2 + ... + 2^h$ = nombre de noeuds + feuilles. Or, nous voulons les noeuds privés des feuilles, il faut donc faire la somme des noeuds pour $0 \leq i \leq h-1$.

Cela revient à écrire la somme suivante :

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

- 2^h feuilles (arbre binaire parfait) :
D'après la partie précédente, on a vu qu'à l'étage (profondeur) i, on a 2^i noeuds. Donc à profondeur h (étages des feuilles), on a 2^h noeuds (qui sont les feuilles).

On a donc :

$$
\begin{aligned}
&(5 \times 2^h) + ((5 + c_h) \times (2^h - 1)) \\
&= (5 \times 2^h) + (5 \times 2^h) - 5 + (c_h \times 2^h) - c_h \\
&= 2^h \times (5 + 5 + c_h) - 5 - c_h \\
&= 2^h \times (10 + c_h) - (5 + c_h) \\
&= l_h
\end{aligned} \tag{6.1}
$$

(5×2^h) correspond aux feuilles et $((5 + c_h) \times (2^h - 1))$ correspond aux noeuds.

6.2 Question 3.12

On a l'idée suivante :

- On compare pour chaque étape i tous les noeuds entre eux une fois, tel que $0 \leq i \leq h$ (exemple : résultat dans un tableau ou une structure).
- Et pour chaque noeuds, on compare leur chaînes de caractères (mot de Lukasiewicz).

a. Nombre de noeuds à comparer par étage

Si à l'étage i on a n noeuds, alors on compare :

- n_0 avec $n_1, ..., n_{2^i}$
- n_1 avec $n_2, n_n .., n_{2^i}$ car on a déjà comparé n_0 et n_1.

On peut généraliser en disant qu'on compare n_x avec tous ses voisins à droite tel que $d > x, (n_d)$ et on ne compare pas avec ses voisins de gauche tel que $g < x, (n_g)$.

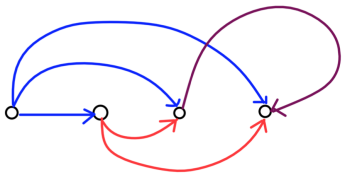

FIGURE 6.2 – Comparaison de chaque noeuds avec ses voisins

On a donc la somme suivante :

$\sum_{i=0}^{n-1} i = \frac{(n-1)(n-1+1)}{2} = \frac{n(n-1)}{2}$

Donc à l'étage i on a :

$\sum_{j=0}^{2^i-1} j = \frac{(2^i-1)(2^i-1+1)}{2} = \frac{2^i(2^i-1)}{2}$ qui correspond donc au nombre de noeuds à comparer à l'étage i.

b. Si à l'étage $i = 0$ on a $l_h = (10 + c_h)2^h - (5 + c_h)$

On sait que $l_h = (10 + c_h)2^h - (5 + c_h)$ correspond à la taille du mot de Lukasiewicz des noeuds à l'étage i.
Alors, à chaque étage on a $l_{h-i} = (10 + c_h)2^{h-i} - (5 + c_h)$.

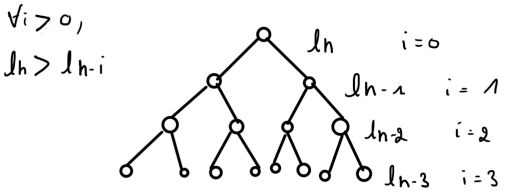

FIGURE 6.3 – Hauteur de l'arbre à chaque étape i

Entre 2 noeuds à l'étage i, on a l_{h-i} comparaisons à faire (chacune de même taille, c'est donc un arbre binaire parfait).
À l'étage i, on a donc :

$(\sum_{j=0}^{2^i-1} j \times ((10 + c_h)2^{h-i} - (5 + c_h)))$
De plus, les feuilles ne peuvent prendre que 2 valeurs : $true$ ou $false$, donc on fait 1 comparaison car on a 2 noeuds :

$\sum_{i=0}^{h-1}(\sum_{j=0}^{2^i-1} j \times ((10 + c_h)2^{h-i} - (5 + c_h)))$

Où $\sum_{j=0}^{2^i-1} j$ représente le nombre de noeuds à comparer et $(10 + c_h)2^{h-i} - (5 + c_h)$ représente la longueur des mots de Lukasiewicz.

$\sum_{i=0}^{h-1}((\frac{2^{2i}-2^i}{2}) \times ((10 + c_h)2^{h-i} - (5 + c_h)))$

Pour $i = 0$: $(\frac{1-1}{2}) \times ((10 + c_h 2^h) - (5 + c_h)) = 0$

Pour $i = 1$: $(2^{2-1} - 2^{1-1}) \times ((10 + c_h 2^{h-1} - (5 + c_h)) = 10 \times 2^{h-1} + c_h \times 2^{h-1} - 5 - c_h$

Pour $i = h - 1$:

$$
\begin{aligned}
&(2^{2(h-1)-1} - 2^{(h-1)-1}) \times ((10 + c_h 2^{h-(h-1)} - (5 + c_h)) \\
&= (2^{2h-3} - 2^{h-2}) \times (20 + 2c_h - 5 + c_h) \\
&= 20 \times 2^{2h-3} + 2c_h \times 2^{2h-3} - 5 \times 2^{2h-3} + c_h \times 2^{2h-3} - 20 \times 2^{h-2} - 2c_h \times 2^{h-2} + 5 \times 2^{h-2} - \\
&c_h \times 2^{h-2} \\
&= 15 \times 2^{2h-3} + c_h \times 2^{2h-3} - 15 \times 2^{h-2} + c_h \times 2^{h-2} \\
&= 2^{2h-3}(15 + c_h) + 2^{h-2}(-15 + c_h)
\end{aligned}
\tag{6.2}
$$

On a donc : $2^{2h-3} = \frac{2^{2h}}{8} = \frac{1}{8} \times 2^{2h} \approx \mathcal{O}(2^{2h})$
Donc la complexité au pire cas de l'algorithme de compression est bien majorée par $\mathcal{O}(2^{2h})$.

6.3 Question 3.13

On cherche la complexité de l'algorithme en fonction de n qui correspond aux noeuds et aux feuilles. On sait qu'un arbre binaire parfait a 2 fils par noeuds et ses feuilles sont à la même profondeur.

On a donc : $h = log_2(n)$.

$\mathcal{O}(2^{2h}) = 2^{2(log_2(n))} = 2^{(log_2(n^2))} = n^2$

De plus, d'après le cours de compression des arbres, on sait que la complexité de l'algorithme ROBDD est $\mathcal{O}(n^2)$.

Donc la complexité de l'algorithme au pire cas en fonction de n si n est le nombres de noeuds (internes et feuille) est $\mathcal{O}(n^2)$.

7 | Étude expérimentale

7.1 Question 4.14

En utilisant les threads on a un temps de 4 secondes pour 5 variables à 500000 arbres et sans les threads on a un temps de 9 secondes.
Pour 5 variables, la distribution exacte prendrait plusieurs heures.

8 | Graphes (complément de la soutenance

Jeudi matin lors de la soutenance, nous avons remarqué avec Mr. Genitrini que nos graphes n'étaient pas tous générés comme attendus.

En effet, ce problème aurait pu être évité pendant la soutenance :
Par défaut, les valeurs générées n'étaient pas bornées. La table de vérité était tronquée, mais cela produisait des courbes qui n'étaient pas similaires aux résultats deJim Newton et Didier Verna. Cependant, ce comportement était possible en modifiant une des macro utilisée pour la configuration du programme (depuis le Makefile des sources .cpp), qui permettait de borner par modulo les valeurs générées.

En conséquence, nous avons légèrement modifié le programme afin que ce comportement soit par défaut : suppression de la macro du Makefile, suppression de quelques lignes de code en rapport avec la macro et ajout du calcul de la borne dans tous les cas (figure 10 et 11), plutôt que de se baser sur cette macro.
De plus, comme la fonction *power* était naïve (n*n*...*n dans une boucle), la fonction mettait beaucoup de temps avant de finir. Nous l'avons donc redéfinit par un algorithme d'exponentiation rapide et maintenant le calcul est vraiment plus rapide.

Pour la figure 10 et 11, il ne faut tout de même pas oublier de configurer l'exécutable depuis le Makefile (principalement MIN/MAX pour le nombre d'arbres à construire entre cette borne, ainsi que INT_LENGTH pour délimiter la taille en caractères des valeurs. Attention, une taille trop grande peut provoquer un temps de calcul plus long (en particulier pour les petits choix de variables...)).
Nous avons initialement ajouté INT_LENGTH au programme car la fonction $big\_random(nb\_car)$ de la bibliothèque choisit génère une valeur d'un certain nombre de caractères, passé en argument.

Voici donc quelques graphes (figure 10), pour 200 000 ROBDD :

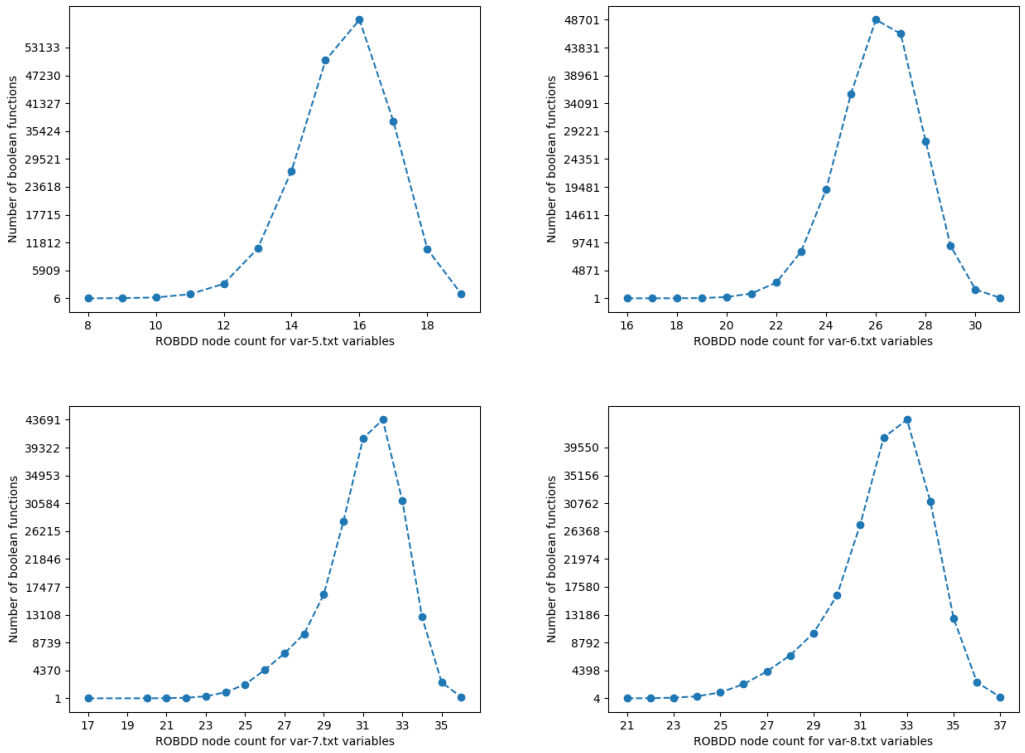

Figure 8.1 – Histogrammes illustrant les distributions de taille des ROBBDs de 5 à 8 variables.

Une figure 11 entre 300 000 et 500 000 ROBDD générée avec 20 threads :

	Standard	Standard	Standard	Standard	Standard
1	No.Variables(n)	No.samples	No.Unique Sizes	Compute time	Seconds per ROBDD
2	7	432231	19	0:0:5	1.33418e-05
3	8	457984	18	0:0:7	1.54837e-05
4	9	438831	18	0:0:8	1.9598e-05
5	10	476588	17	0:0:13	2.74896e-05
6	11	484856	18	0:0:21	4.47016e-05
7	12	421703	18	0:0:34	8.15263e-05
8	13	300947	18	0:0:41	0.00013837