Rapport de projet

13 novembre 2022

Compression des arbres de décision binaire

Auteurs:
Amaury Curiel
Ben Idriss Abdillahi

Table des matières

1 Presentation

L'objectif de ce projet était de compresser efficacement les arbres de décision binaire. Pour ce faire, nous avons utiliser les mots de Lukasievic. Enfin, nous avons réaliser quelques experiences mettant en lumière l'efficacité de notre approche. Nous avons utiliser le langage Ocaml car son paradygme fonctionnel facilitant la récursion, il nous semblait être la meilleure option pour réaliser ce projet. pour compiler le projet vous devez entrer la commande :

```
ocamlfind ocamlopt -o projet -linkpkg -package num  -thread projet.ml
```

sous un environement Opam. De plus, pour lancer le projet, vous devez posseder un environement python3 ainsi que l'application eog

2 Algorithme

2.1 Construction d'un BDD exemple

Pour implémenter l'algorithme de compression des arbres de décision binaire,(plus tard abrégé en BDD) nous avons tout d'abbord remarquer qu'il etait possible de représenter n'importe quelle table de vérité en décomposant un nombre en binaire et en associant aux 1 la valeur True et aux 0 la valeur False. Ainsi, lorsque l'on décompose le nombre 42 en binaire, nous obtenons la décomposition: 101010 que nous traduisons en [False;True;False;True;False;True].
Une fois ceci fait, nous completons cette table afin qu'elle soit de taille compatible pour un BDD. En effet, le nombre de feuille d'un BDD est toujours inclu dans
$TaillePossible = \{2^{2^n} \mid n \in \mathbf{N}\}$. Ainsi, nous completons notre table en rajoutant des False en fin de liste pour que la taille de la décomposition obtenue soit incluse dans l'ensemble $TaillePossible$. Ensuite, nous pouvons construire le BDD en fusionnant 2 a 2 les elements de la liste. Ainsi, l'arbre obtenu grâce à l'entier 42 sera ,l'arbre obtenu avec la table de vérité [False;True;False;True;False;True;False;False] qui sera a la première itteration de l'algorithme de construction :

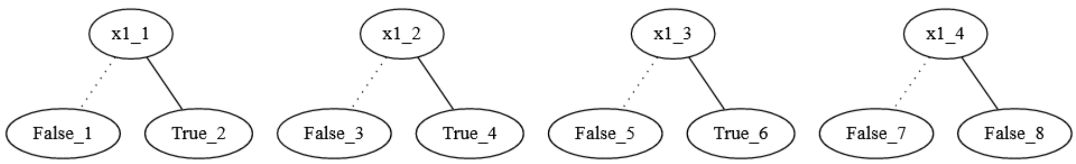


Figure 1: Etat du graphe après une itération

puis à la 2eme itération:

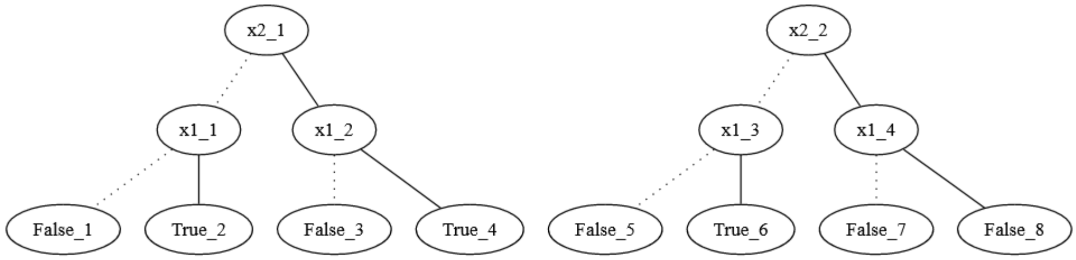


Figure 2: Etat du graphe après 2 itérations

puis enfin :

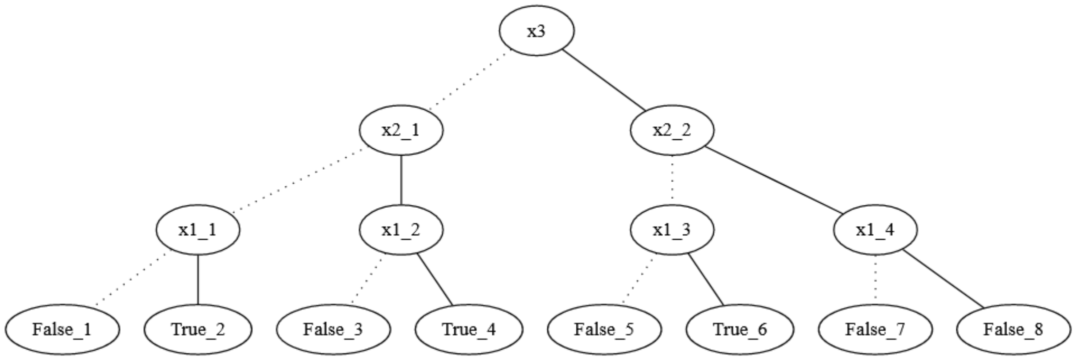


Figure 3: BDD obtenu grâce à l'entier 42

2.2 Compression du BDD obtenu

Maintenant que nous avons reussis a construire un BDD à partir d'un nombre, nous devons le compresser pour obtenir un ROBDD. Pour ce faire, nous avons enrichis le bdd pour rajouter les mots de Lukasievic a chaque noeud. ainsi, nous sommes passé du type

```
type robdd =
|Node of string*robdd*robdd*int
|Feuille of bool*int*int
|Vide ;;
```

au type

```
type lukarobdd =
|NodeL of string*lukarobdd*lukarobdd*string
|FeuilleL of bool*int*string
|VideL;;
```

Pour construire le lukarobdd, nous utilisions la fonction *buildLukArbre* (ligne 150 dans le code actuel) et la fonction *fillTab* (ligne 221 dans le code actuel). remarquons tout

d'abord que dans un mot de Lukasievic il est facile d'extraire le mot de Lukasievic du fils droit et du fils gauche : par exemple dans le mot de lukasievic $x0(x1(T)(F))(x1(F)(T))$, le fils gauche est $x1(T)(F)$ et le fils droit est $x1(F)(T)$. La récuperation de fils gauche et droit est facilement implémentable avec la fonction du module String : *String.split_on_char*. Grace à ceci, nous pouvons facilement implémenter la fonction *buildLukArbre*.
Cette fonction prends en paramètre un BDD et renvoie un BDD enrichis par les mots de Lukasievic. Lors de cette construction, la fonction remplis 2 tables de hachage *Luka* et *revLuka* (nous verrons dans la section amélioration & optimisation en quoi ce choix est discutable et comment nous pouvons l'améliorer), contenant respectivement une association mot de Lukasievic vers entier et entier vers mot de Lukasievic.

```
-----------------------------------------------------------
x2(0)(1)     2
true     0
x0(4)(5)     6
x1(2)(3)     4
false     1
x1(3)(3)     5
x2(1)(1)     3

-----------------------------------------------------------
6     x0(4)(5)
2     x2(0)(1)
3     1
5     3
4     x1(2)(3)
0     true
1     false
```

Figure 4: Etat des tables de hachage pour un arbre de 3 variables

Une fois ceci fait, nous pouvons passer à la construction du ROBDD. La construction du ROBDD est assurée par la fonction $fillTab$ qui prends en parametttre une table de hachage et renvoie un le ROBDD construit. Pour ce faire, $fillTab$ vas detecter lorsqu'un des fils du noeud en cours est deja construit et vas faire pointer le nouveau noeud nouvellement créer vers le noeud deja vu. Par exemple pour la table de hachage en Figure 4, $fillTab$ vas chercher le noeud pour la valeur 0 qui vaut *true* puis, vas passer a la valeur 1 qui vaut $false$ puis vas passer a la valeur 2 qui crée un noeud de label $x2$ puis qui fait pointer son fils gauche vers la valeur 0 donc *true* et son fils droit vers la valeur 1 donc *true*. et ainsi de suite. $fillTab$ s'appuie sur une fonction $folloWay$ (ligne 191 du code actuel) qui tant qu'elle detecte qu'un noeud est deja vu, remonte la table de Hachage pour trouver le fils corrspondant. Par exemple $followWay$ 5 vas appeler $followWay$ 3 qui lui même vas appeler $followWay$ 1 qui vas renvoyer $false$.

3 Complexité

3.1 Complexité de la construction du BDD

Tout d'abord, notons que la décomposition de l'entier n nécéssaire a la construction du BDD se fait en $O(log(n))$ en nombre de division. En effet, dans l'algorithme des divisions successives, la taille de n est réduite d'un facteur 2 à chaque itteration. Maintenant, attardons nous sur l'étape de fusion. Pour la première itteration, nous devons fusionner chacune des feuille cette etape nécéssite donc $|n_2|$ etapes pour etre effectuée. puis a l'étape suivante $|n_2|/2$ etapes et ainsi de suite($log_2(n)$ fois). Notons tout d'abbord que $|n_2| = log_2(n)$ on a donc besoin de $log_2(n)^2$ opérations pour créer le BDD. la complexité de la création du BDD est donc en $log_2(n)^2$ fusions de noeud.

3.2 Complexité de la création du BDD enrichis par les mots de Lukasievic

la fonction *buildLukArbre* telle qu'elle est implémentée nécéssite de calculer le mot de Lukasievic pour chaque noeud selon un parcours top down (cette implémentation est très couteuse et nous proposerons des optimisations dans la section dediée). Elle a donc une complexité exponentielle la taille de l'arbre. Quand a la fonction $fillTab$, ella admet une complexité quadratique en la taille de la table de hachage (on parcours a chaque itteration au pire la totalité de la table d ehachage).

3.3 Questions du sujet

3.3.1 Question 3.11

Introduisons tout d'abbort le lemme suivant: "Un BDD de hauteur h est obtenu par la fusion de deux BDD de hauteur $h-1$".
On raisone par récurrance.
On cherche a verifier la propriété $P(n)$: *le mot de Lukasievic à la racine d'un arbre de hauteur n est egal à* $(10 + c_h) * 2^h - (5 + c_h)$
Initialisation:
Vérifions $P(0)$:
Un arbre de hauteur 0 est un arbre réduit a une feuille.
Or dans un BDD, la taille d'une feuille est majorée par la taille de $False$ qui vaut
$5 \leq (10 + c_h) * 2^0 - (5 + c_h) = 5$

Hérédité: Soit $n \in \mathbf{N}$ tel que $P(n)$. Montrons alors $P(n+1)$. Remarquons que la taille du mot de Lukasievic d'un noeud interne est egal a :
$|x| + c_h + |(| + |sousArbreGauche| + |)| + |(| + |sousArbreDroit| + |)|$ ce qui equivaut)
$5 + c_h + |sousArbreGauche(detailleh)| + |sousArbreDroit(detailleh)|$. ce qui est majoré d'après l'hypothèse de récurence par: $5+c_h+(10+C_h)2^h-(5+c_h)+(10+C_h)2^h-(5+c_h) = (10 + c_h)2^{h+1} - (5 + c_h)$
La propriété etant héréditaire et initialisée, elle est donc vraie pour tout entier naturel d'ou l'énoncé. De plus, le fait qu'un BDD soit un arbre complet avec toutes les feuilles au même niveau nous garentit trivialement les 2 lemmes suggerés dans l'énoncé.

3.3.2 Question 3.12

Soit A un BDD de hauteur h. Soit $0 \leq k \leq h$.
Notons tout d'abord qu'au niveau k, un BDD possède 2^k noeuds.
Au niveau k il est donc nécéssaire pour chaque noeud de se comparer a tout les autres noeuds. On a donc que pour chaque niveau k,il est nécéssaire d'effectuer $O(2^{2k})$. D'apres la question précédente, il est nécéssaire de comparer $(10 + c_h)2^{h-k} - (5 + c_h)$ caracteres.
ainsi le nombre de comparaison nécéssaire pour un niveau est donc
$= O((10 + c_h) * 2^{h+k} - (5 + c_h) * 2^{2k})$
Maintenant, pour avoir la complexité sur tout l'arbre, sommons sur k.
$\sum_{k=0}^{h}(10 + c_h) * 2^{h+k} - (5 + c_h) * 2^{2k}$
$\leq \sum_{k=0}^{h}(10 + c_h) * 2^{h+k}$
$\leq \sum_{k=0}^{h}(10 + c_h) * 2^{2h}$
$= \sum_{k=0}^{h}(10 + c_h) * 2^{2h}$
$= (10 + c_h) * \sum_{k=0}^{h} 2^{2h}$
$= (10 + c_h)h2^{2h} = O(2^{2h})$
d'ou l'énoncé.

3.4 Question 3.13

Un arbre de hauteur h possède 2^{h+1} feuilles.la complexité est donc $O(2^{2*2^{h+1}})$

4 Experimentations

Afin de définir les limites de notrer algorithme, nous avons réaliser une séries d'experimentation dont voici les résultats. Tout d'abord, nous avons pu vérifier que la complexité en pratique de notre algorithme correspondait bien a la complexité empirique en traçant le graphe du temps d'execution en fonction du nombre de variable. Nous avons obtenu le graphe ci dessous.

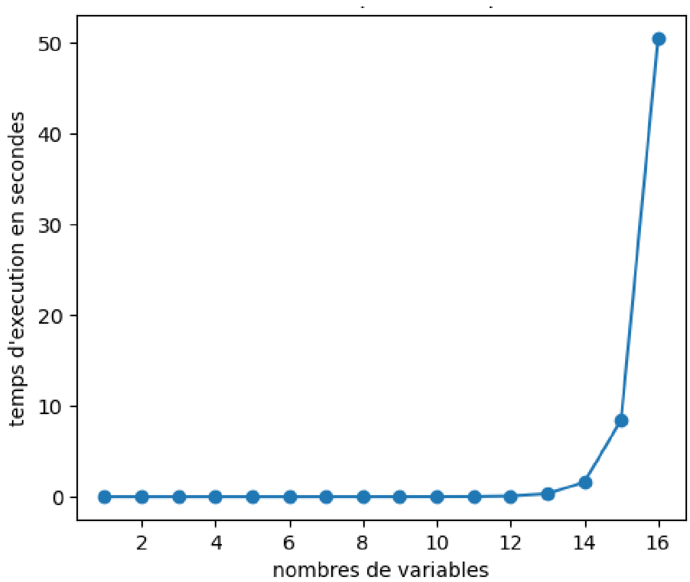
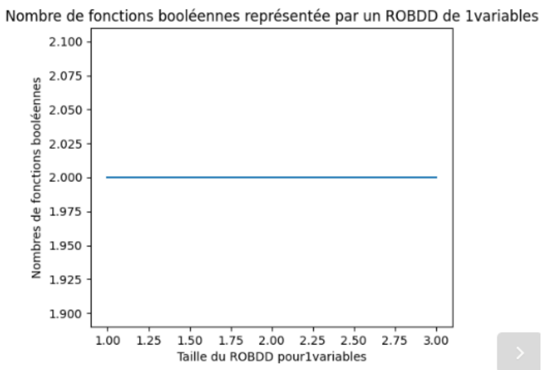


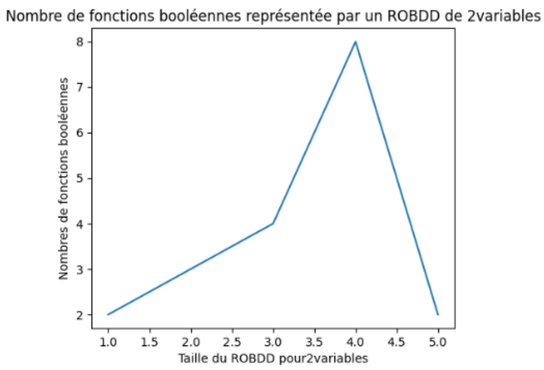
Figure 5: Graphique du temps d'execution en fonction du nombre de variable

4.1 resultat

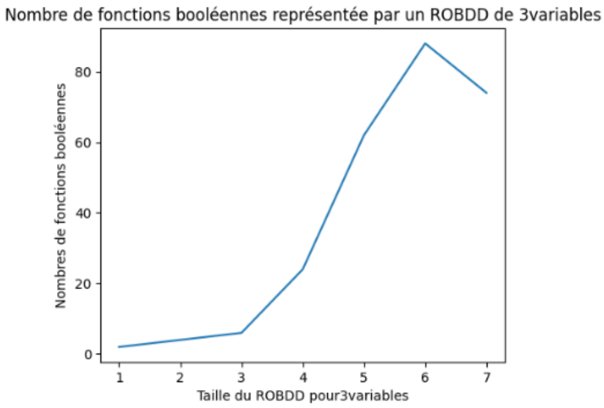
Nous avons aussi pu comparer nos résultats à ceux de l'article et nous avons obtenus les courbes suivantes:



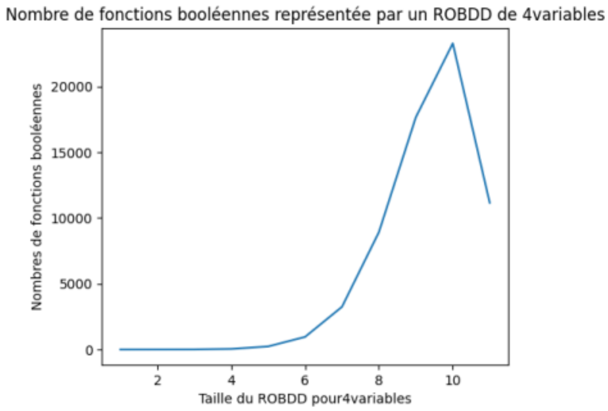
Graphique la distribution des ROBDD en fonction du nombre de noeud pour 1 variable



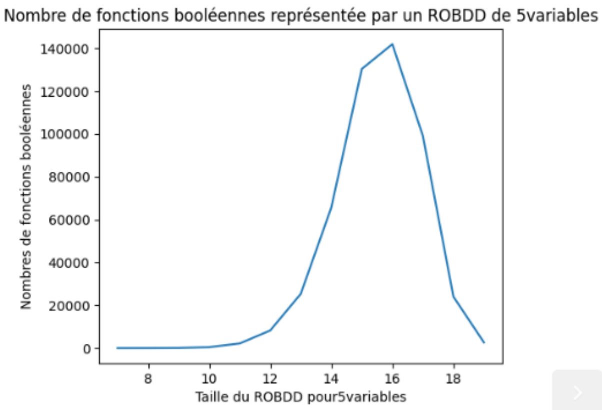
Graphique la distribution des ROBDD en fonction du nombre de noeud pour 2 variable



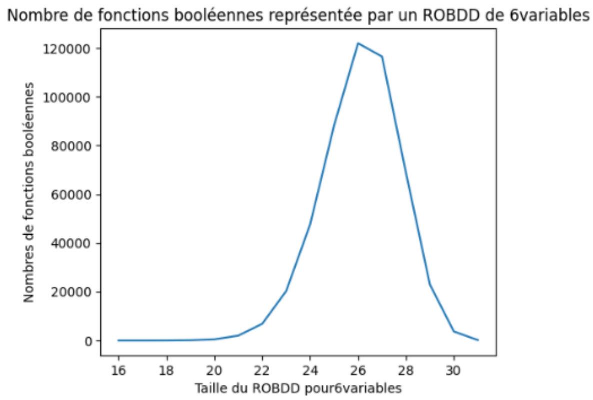
Graphique la distribution des ROBDD en fonction du nombre de noeud pour 3 variable



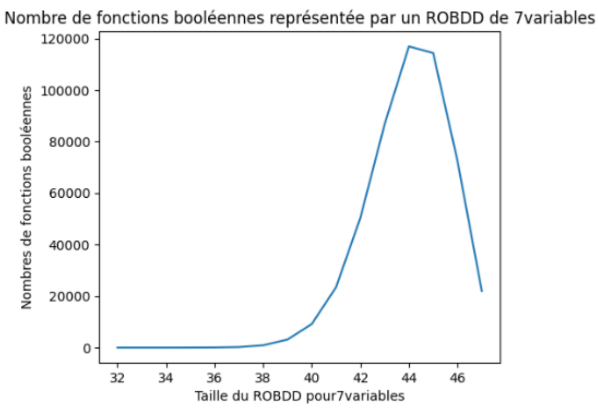
Graphique la distribution des ROBDD en fonction du nombre de noeud pour 4 variable



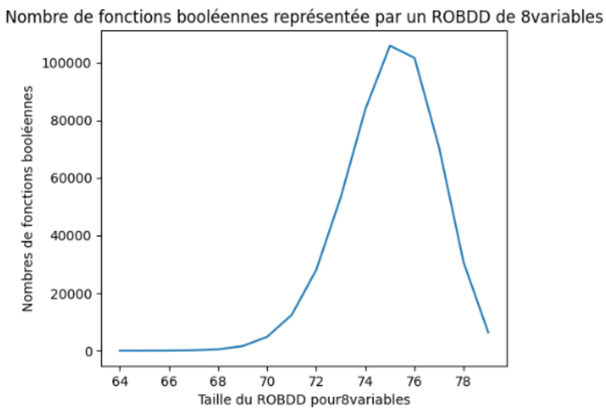
Graphique la distribution des ROBDD en fonction du nombre de noeud pour 5 variable



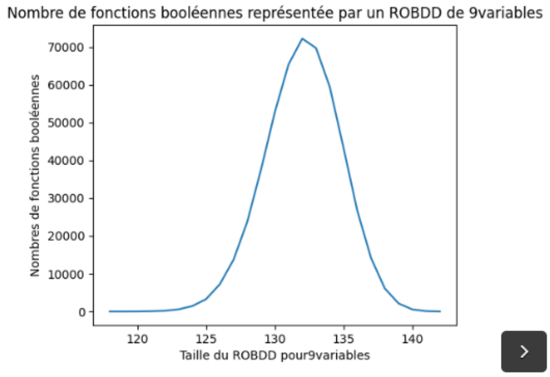
Graphique la distribution des ROBDD en fonction du nombre de noeud pour 6 variable



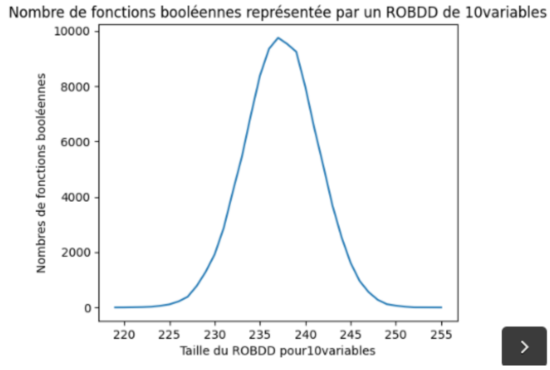
Graphique la distribution des ROBDD en fonction du nombre de noeud pour 7 variable



Graphique la distribution des ROBDD en fonction du nombre de noeud pour 8 variable



Graphique la distribution des ROBDD en fonction du nombre de noeud pour 9 variable



Graphique la distribution des ROBDD en fonction du nombre de noeud pour 10 variable

4.2 Interpretation

Ces experiences semblent nous montrer que la distribution semble tendre vers une gaussienne. De plus, il est innutile de generer tout les ROBDD; en effet, generer que la moitié des ROBDD suffit pour avoir une distribution exacte car la décomposition en binaire des nombres entre 0 et 2^{2n} est symétrique. On a maintenant un tableau qui a été obtenu avant de remarquer la symétrie. Il nous est cependant impossible de calculer en temps raisonable la distribution exacte avec 5 variables.

Ce tableau montre bien la complexité exponentielle de l'algorithme.

N° Variable	N°Samples	N°Unique Size	Compute Time(s)	Seconds per ROBDD
5	500 000	13	159.410526	0.000318
6	500 000	16	221.151752	0.000442
7	500 000	14	384.648485	0.000768
8	500 000	16	942.739182	0.001884
9	500 000	26	2166.711242	0.004332
10	500 000	35	5634.368102	0.011268

Figure 9: Graphique representant la distribution obtenue pour 8 variables et 500 000 ROBDD tirés au hasard

5 Critique et idées d'amélioration

Dans cette section, nous allons nous attarder a etudier des voies d'amélioration de l'algorithme.

5.1 Reduction de la complexité de la construction du BDD enrichis par les mots de Lukasievic

Dans notre approche, notre code a une complexité expoenentielle ce qui est très pénalisant pour les experimentations. afin de remedier à cela, nous avons reflechis à un algorithme. Nous construisons l'arbre via l'algorithme de la fusion mais au lieu de fusionner les noeuds 2 par 2 etage par etage, ce qui rends impossible la detection d'isomorphisme, nous proposons de construire l'arbre de maniere prefixe(detaillé dans la soutenance). Nous fusionnons les 2 premiers noeuds puis les 2 suivant puis nous fusionnons les résultats entre eux puis les 2 suivant et ainsi de suite. Cet algorithme aurait permis d'avoir un algo linéaire en nombre de feuille de l'arbre

5.2 Reduction de la complexité de *buildLukArbre*

Dans la fonction *buildLukArbre*, nous utilisons des tables de hachage. Cette structure de donnée, bien que permetant une recherche en temps constant en moyenne, coute 2 fois plus cher en mémoire et peut aisément etre remplacé par un tableau. Cette approche ne fais pas gagner en classe de complexité mais permets de gagner en temps d'execution.

5.3 Idées tentées mais avortées

Lors de la partie experimentation, nous avons tenter de diviser par 8(nombre de coeur de ma machine) le temps d'execution du programme en divisant la tache de generation des ROBDD avec 8 threads. Cependant, le GC d'Ocaml etant bloquant, l'implémentation par threads était au mieux aussi efficace que l'algoritme actuellement utilisé.

Il fut également question de generer directement le ROBDD a partir du nombre sans passer par un BDD mais l'idée fut abandonnée par manque de temps.

La fusion de 2 ROBDD n'as de plus pas été testé mais aurait pu l'etre en récuperant la table de vérité des 2 arbres en paramètre et en combinant les valeurs 2 a 2 avec l'opérateur fournis par l'utilisateur et en comparant la table ainsi obtenue avec la table extraite du ROBDD combiné.