Test de propriétés et BDD

Alpha DIALLO, Lamine KEITA, Sacha MEMMI

Avril 2020

Encadrant : Antoine Genitrini

Université Pierre et Marie Curie
Paris, France
Pstl

Master 1, Semestre 2

Table des matières

1 Abstract

Les fonctions booléennes sont très largement utilisées en science. Elles sont aussi largement utilisées parce que simple à manipuler et à mettre en place leur utilité est également très large. On les utilise par exemple en électronique pour travailler sur les circuits logiques.

Les structures avec lesquelles nous travaillons dans ce rapport sont ce que l'on appelle des robdd, Reduced Ordered Binary Decision Diagram. Elles représentent les fonctions booléennes en tant que graphes (dirigés et acycliques). Elles sont l'une des méthode préférée pour représenter celles-ci, en particulier, en informatique. Elles sont préférées car facile à implémenter, quelques ligne de codes dans n'importe quel langage sont suffisantes pour les représenter, mais également parce qu'elles sont simples à manipuler.

En particulier nous cherchons à énumérer le nombre de robdd unique représentant une fonction ayant un nombre défini de variables selon leurs tailles. Ce problème a intéressé le domaine scientifique depuis que ces structures ont vu le jour, il présente de nombreux intérêts dans tous les domaines où l'on travail avec des robdd.

En plus de rappeler la méthode standard de génération des robdd, nous donnerons un moyen de reconstruire ces structures. Nous obtenons par cette nouvelle méthode un générateur uniforme de robdd que nous pouvons insérer dans des outils tels que Quickcheck, outil au sujet duquel nous consacrerons un chapitre dans ce rapport.

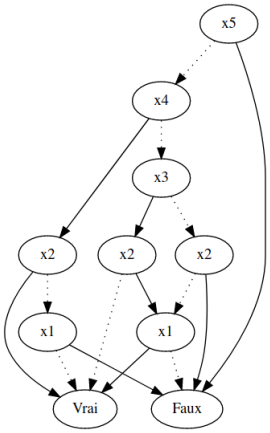

FIGURE 1 – Robdd généré avec la méthode du unranking

Avec notre méthode, nous avons pu générer le robdd de taille 8 sur 5 variables de la figure 1.

La méthode de génération de robdd présenter dans ce rapport s'appuie sur celle présenter dans [4], en plus d'implémenter en pseudo-code des algorithmes introduits dans ce papier nous en fournissons certains non décrits dans celui-ci.

2 Introduction

L'idée derrière les arbres de décisions binaires a été introduite par Shannon en 1938 dans [8]. C'est en 1959 que Lee introduit l'idée de bdd dans [6].

Les diagrammes de fonction binaires sont principalement des outils pour les ingénieurs (et autre professions travaillant dans le hardware en informatique) qui manipulent des circuit électroniques, cas où l'utilisation de tables de vérité se révèle efficace. Dans ce cas là, diviser leur circuit complexe en sous circuits simples comme le permettent les diagrammes de décisions binaires (concept que l'on appelle "diviser pour régner") est très efficace.

Plus particulièrement, les circuits électroniques sont devenus si compliqués que les robdd sont à peine suffisants pour les traiter comme l'a noté Knuth il y a de ça plus de 10 ans [5].

Un bdd (binary decision diagram) est un diagramme de décision binaire pouvant représenter une fonction booléenne. Il est important de souligner que l'utilisation des BDD est bien plus vaste que la seule application aux fonctions booléennes. Cependant, nous nous concentrerons sur celles-ci durant cette étude. Nous pouvons voir un exemple de bdd, celui de la figure 2 peut être utilisé pour représenter la fonction booléenne : $\bar{x3}(x2 + \bar{x3}) + x3(\bar{x2} + x1)$, ainsi que la table de vérité ci dessous :

x_3	x_2	x_1	R
$\bot$	$\bot$	$\bot$	$\top$
$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\top$
$\bot$	$\top$	$\top$	$\bot$
$\top$	$\bot$	$\bot$	$\bot$
$\top$	$\bot$	$\top$	$\top$
$\top$	$\top$	$\bot$	$\top$
$\top$	$\top$	$\top$	$\top$

Un robdd (reduced ordered binary decision diagram) est comme son nom l'indique la forme réduite et ordonnée d'un bdd. Dans la littérature le terme bdd est presque toujours utilisé pour évoquer un robdd, nous ferons d'abord la distinction entre ces deux termes pour mettre l'accent sur l'aspect réduction. Les bdd que nous considérons dans ce rapport sont toujours ordonnés. Dans cette étude, lorsque nous ferons parlerons de BDD, nous ferons référence aux obdd (ordered binary decision diagrams).

Les robdd n'ont été introduit qu'en 1986 [2]. Assez tard dans l'histoire de l'informatique. Les robdd ont été une telle révolution que [2] est pendant des années resté le papier le plus cité en science de l'informatique comme l'a noté Knuth :

"His (Randal E. Bryant's) introduction to the subject [IEEE Trans. C-35 (1986),677-691] became for many years the most cited papers in all of computer science"[5]

De nombreuses autres variantes de robdd ont également été créées depuis [9].

Le principe d'un robdd est simple, on cherche à conserver de l'espace mémoire en se débarrassant des structures redondantes, la mémoire étant une préoccupation importante en informatique. Pour réduire un bdd, nous supprimons les sous structures redondantes, nous supprimons également les noeuds non essentiels. Nous représentons les bdd en tant qu'arbre plat et les robdd en tant que DAG (directed acyclic diagrams) [7]. Plus généralement, les robdd sont des sous types de bdd et tous les types de bdd peuvent être représentés par un dag.

"A function in Boolean algebra. The function is written as an expression formed with binary variables (taking the value 0 or 1) combined by the dyadic and monadic operations of Boolean algebra."

- Oxford Reference

Il existe de nombreux moyens pour représenter des fonctions booléennes. Ceux préférés en science de l'informatique sont les diagrammes, pas seulement grâce à la manière dont ces diagrammes peuvent être réduits, mais aussi parce qu'ils peuvent être conservés en mémoire en tant que suites de noeuds qui ne sont pas forcement adjacents en mémoire les uns par rapport aux autres. Ceci rend leur conservation plus simple et efficace.

Un exemple d'arbre de décision binaire et un DAG représentant sa forme réduite sont donné dans la figure 2. Nous suivons ici le consensus dans la littérature et représentons les liens vers les fils haut par une ligne solide, et ceux vers les fils bas par une ligne en pointillé. Le fils haut (respectivement fils bas) d'un noeud est le noeud successeur de celui marqué par un arc 1 (respectivement 0), la valeur de transition vers ce noeud dans la fonction de transition est donc 1 (respectivement 0).

En plus de l'utilité évidente des fonctions booléennes ont peut aussi noter que les bdd sont devenus aussi populaires, car comme Knuth l'a noté dans [5], les bdd simples sont relativement facile à fusionner pour obtenir des bdd plus complexes.

Dans ce rapport notre but est de présenter un nouveau procédé permettant d'énumérer des robdd sans passer par le processus de compression, procédé introduit dans [4].

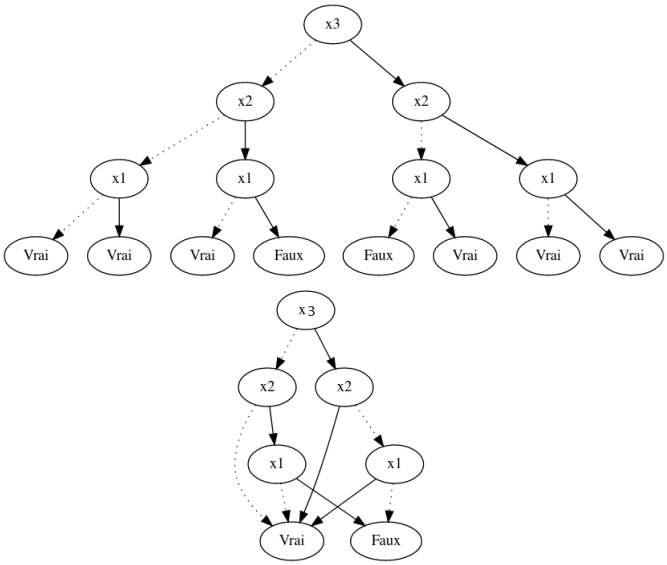

FIGURE 2 – Arbre de decision binaire en haut et dag représentant sa forme réduite en bas

Le processus de compression est relativement simple avec une complexité polynomiale, mais la méthode standard de génération des bdd d'un nombre donné de variables k utilisé jusqu'ici était impraticable pour $k > 4$ [3]. Pour construire les bdd ayant k variable dans la méthode standard nous construisons tout d'abord toutes les fonctions booléennes ayant k variables, puis leurs bdd, puis compressons ces bdd afin d'obtenir leurs robdd et enfin nous supprimons les robdd apparaissant plus d'une fois. Ceci avait une complexité doublement exponentielle de $O(2^{2^k})$.

Nous définissons la taille d'un robdd par le nombre de noeuds internes (non terminaux) qu'il contient.

Pour énumérer ces structures dans la méthode que nous introduisons ici, nous utilisons de nombreuses propriétés des robdd. Nous utilisons ensuite des méthodes dites de unranking et ranking basées sur le modèle standard introduit dans [1] en définissant un ordre total sur les robdd et en décomposant chacun par une constitution de sous robdd.

Le processus de ranking consiste à attribuer à chaque objet d'un ensemble un identifiant unique. Celui du unranking consiste à retrouver, à partir d'un identifiant, un objet au sein d'un ensemble.

Nous obtenons grâce à cette méthode un générateur uniforme de robdd.

Avec la méthode standard utilisée dans [3] les robdd de taille proche de la taille maximale étaient crées avec une grande probabilité, mais dans celle ci chaque robdd a un identifiant unique dans l'ensemble. On peut également

modifier ce générateur pour ne créer que des robdd ayant certains types de propriétés.

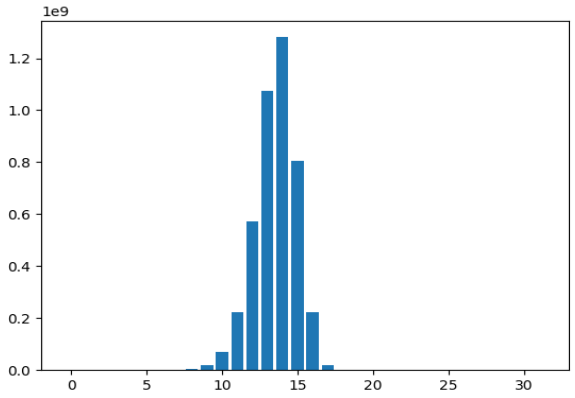

FIGURE 3 – Distribution des taille de bdd ayant 5 variable

Avec notre méthode nous arrivons a comptabiliser les bdd ayant 5 variables, avec une implantation en python et un matériel limité, en 14 secondes.

En faisant une comparaison des résultats trouvés dans [3] et visibles sur la figure 4 on remarque que nos résultats visibles sur la figure 3 y sont similaires (on rappelle qu'ici on ne prend pas en compte les feuille dans le calcul de la taille d'un bdd).

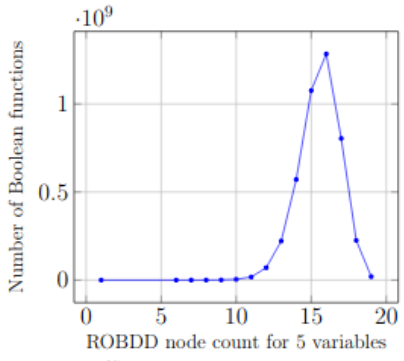

FIGURE 4 – Distribution des taille de bdd ayant 5 variable selon [3]

Dans ce rapport nous considérerons que nous ne rencontrons jamais le robdd réduit au noeud unique $\top$ ou $\bot$.

3 BDD et ROBDD

Dans cette section nous cherchons à définir les structures basiques avec lesquelles nous travaillons, les bdd et les robdd.

Un bdd est composé de noeuds ayant un label. Chaque noeud non terminal possède deux enfants, ou fils, un bas et un haut liés à leurs parents par des liens notés respectivement 0 et 1. Chaque noeud a un index qui détermine son ordre dans le bdd. Les noeuds sont traités dans l'ordre décroissants de leur index.

La manière dont nous ordonnons les noeuds est importante comme noté dans [3] et trouver l'ordre le plus efficace est un problème NP (nondeterministic polynomial time).

La racine a un index k qui est l'index maximal de la bdd (ou robdd), et les noeuds terminaux ont des index nuls. Tous les noeuds internes (quand ils existent) ont des indexes allant de k à 0, non inclus. Pour un bdd l'index de la racine est le nombre de variables de la fonction booléenne qu'il représente, pas forcement pour un robdd.

Un bdd associe à chaque affectation de variables une valeur de retour, pour connaître cette valeur nous allons tout d'abord à la racine, puis nous suivons un chemin en se fiant à la valeur de chaque variable données en entrée selon l'index du noeud courant. Si la variable d'index du noeud courant est 0 nous nous rendons à l'enfant bas du noeud courant, l'enfant haut si la variable a pour valeur 1. Nous finissons par arriver à un noeud terminal, que dans le reste de ce rapport nous appellerons feuille, de valeur $\top$ (vrai) ou $\bot$ (faux) qui est la valeur de retour de la bdd pour les variables considérées.

Pour un bdd, de la figure 2 si l'on donnait les variables 101 en tant qu'entrée elle retournerait $\top$ (vrai).

La méthode classique pour obtenir un robdd est de réduire un arbre de décision binaire.

Pour effectuer cette réduction nous suivons tout d'abord deux règles, que nous notons R et M.

Considérons Δ un dag représentant une fonction booléenne f. Considérons également α et β deux noeuds distincts dans Δ .

Règle M : Si α et β sont des racine des sous graphes isomorphes alors fusionner β et α.

Règle R : Si les deux fils de β sont isomorphes à α, alors faire pointer tous les liens entrant de β vers α et supprimer β.

Nous appliquons récursivement ses deux règles à tous les noeuds de Δ .

Deux ordres de parcours peuvent être considérés, $post_o$ et pre_o.

Dans l'ordre de parcours $post_o$ nous visitons tous d'abord le fils bas du noeud courant, puis son fils haut et enfin le noeud lui-même. Dans l'ordre de parcours pre_o nous visitons tous d'abord le noeud courant, puis son fils bas, et enfin son fils haut.

Nous choisissons ici de considérer l'ordre de traversée $post_o$ pour effectuer chacune de nos compressions, puisque c'est celui préféré en science de l'informatique. Choisir un ordre en particulier est important pour s'assurer que des bdd équivalents soient compressés en tant que robdd identique une fois l'ordre des variables établie.

Nous pouvons désormais définir formellement le processus de compression.

Compression : Considérons Δ le bdd de la fonction f. Quand le noeud λ est visité, λ étant l'enfant d'un noeud μ. Si un sous-arbre Λ, celui qui a pour racine λ, a déjà été vu lors de la traversée de Δ, sous arbre dont la racine est ω, alors Λ est supprimé de Δ et le noeud μ obtient un pointeur vers ω, remplaçant le lien allant précédemment de μ à λ. Une fois la traversée de Δ achevée, le DAG résultant est le robdd de f.

Définissons maintenant formellement un robdd, cette définition ainsi que les notations que l'on utilise se basent sur celles définies dans [4].

Considérons un robdd Δ = (Q, I, r, δ). Q est l'ensemble de noeuds que l'on obtient par une traversé en $post_o$ de Δ. r est la racine de Δ, I est la fonction qui associe pour tout noeud $\lambda \in$ Q son index et δ est la fonction de transition complète qui associe à chaque noeuds ses fils.

- Q = ([feuille1],[feuille2], ..., r) ([feuille1] etant la premiere feuille que l'on rencontre lorsque l'on parcourt Δ en $post_o$ et [feuille2] la seconde)
- $I = Q \to \{0, \ldots, k\}$ où k est l'index de r
- $\delta = (Q \setminus \{\top, \bot\}) * \{0, 1\} \to$ Q

On note que Q a une taille forcément supérieure ou égale à 3.

On peut ainsi representer le dag de la figure 2 par un robdd Δ = (Q, I, r, δ) ayant les propriétés suivantes :

- Q = $(\bot, \top, x1, x1, x2, x2, x3)$
- $I = \{\bot, \top, x1, x1, x2, x2, x3\} \to \{0, 0, 1, 1, 2, 2, 3\}$
- $\delta = \{x1 * 0 \to \bot, x1 * 1 \to \top, x1 * 0 \to \top, x1 * 1 \to \cdot x2 * 0 \to x1, x2 * 1 \to \top, x2 * 0 \to \top, x2 * 1 \to x1, x3 * 0 \to x2, x3 * 1 \to x2\}$ Q
- r = x3

Dans le reste de ce rapport nous utiliserons le terme bdd pour faire référence aux robdd.

4 (RO)BDD : propriétés

Dans cette section nous définissons des propriétés que l'on assigne aux bdd. Pour ce faire, nous mettons en place un ensemble de notations et d'opérations. Toutes les notations utilisées dans cette section seront utilisées dans le reste de ce rapport. Les opérations que nous définissons pour les listes sont également applicables aux ensembles. On se contente ici de redéfinir les notations définies dans [4].

Pour deux noeuds α et β d'un dag Φ nous écrivons $\alpha \Leftarrow_{post} \beta$ si le noeud α est visité avant le noeud β en $post_o$ lors de la traversée de Φ, et $\alpha \Leftarrow_{pre} \beta$ si le noeud α est visité avant le noeud β en pre_o lors de la traversée de Φ.

Pour deux listes l et l' de tailles respectives m et n, où l = (p0,p1...,pm) et l' = (p'0,p'1,...,p'n) nous écrivons L=l+l' l'opération telle que si m<n avec L=(P0,P1,...,Pn) alors nous avons pour $i \leq$ m Pi=pi+p'i et pour $m < i \leq n$ Pi=p'i.

Pour une liste l nous notons $\mid l \mid$ le nombre d'éléments dans l, et $\mid\mid l \mid\mid$ la somme de tous ses éléments.

Profil d'un ensemble : Le profil P=profil(S) d'un ensemble S est simplement une liste du nombre d'élément de S ayant un certain index, on note P = (p0,p1...,pj,... pk) où pj est le nombre d'éléments dans l'ensemble S d'index j et k l'index maximal de S.

Le processus d'extension de cette définition de profil à des graphes est évident.

Squelette d'un bdd : Considérons un bdd Δ = (Q, I, r, δ) enraciné en r. Le squelette T = (Q', I, r, δ') de Δ est un dag tel que Q' = Q / $\{\top, \bot\}$ où chaque noeud $\lambda \in$ Q' a été obtenu par la traversée en $post_o$ de Δ en faisant omission des deux feuilles. Les liens du squelette sont décrit en utilisant une fonction de transition partielle δ' : Q' $\times$ {0, 1} $\rightarrow$ Q' $\cup$ {nil} où nil est un symbole spécial désignant une transition non définie.

Plus simplement, un squelette est un bdd auquel on retire les deux feuilles et les pointeurs.

Dans le reste de ce rapport nous nous référerons aux valeurs non définis de la fonction de transition partielle en tant que nil transition.

Nous notons n le nombre de noeud d'un dag Δ d'un bdd, nous savons que son squelette S a le même nombre de noeud que Δ à l'exception des deux feuilles. Nous pouvons ainsi noter que S a (n-2) noeuds. Nous savons également que tous les noeuds non terminaux de Δ ont deux enfants, ainsi le nombre de transition dans Δ est (n-2)*2, ce qui est le nombre total de transition dans S (a la fois les transition nil et non nil). Nous savons également que chaque noeud $\lambda \in$ S a nécessairement un degré entrant de 1 à l'exception de la racine. Ainsi le

nombre restant de transitions non nil dans S est (n-3). Nous pouvons maintenant conclure que S a un nombre de nil transition égal à (n-1).

Sous arbre : Un sous arbre TA(λ) est un arbre enraciné en λ appartenant à un autre arbre.

Pool : Considérons un dag D, la pool PT(λ) d'un noeud $\lambda \in$ D est :

$$\text{PT}(\lambda) = \{\tau\text{'} \in \text{D} \mid \tau\text{'} \Leftarrow_{pre} \lambda \text{ et } \text{I}(\tau\text{'}) < \text{I}(\lambda)\} \cup \{\top, \bot\} \ .$$

Pool profil : Le pool profil de λ est comme son nom l'indique le profil de sa pool que l'on note : profil(PT(λ)).

Level set : Considérons un dag D, le level set $\lambda \in$ D est :

$$\text{ST}(\lambda) = \{\theta\text{'} \in \text{D} \mid \theta\text{'} \Leftarrow_{pre} \lambda \text{ et } \text{I}(\theta\text{'}) = \text{I}(\lambda)\} \ .$$

Level rank : Considérons, ST(λ) le level set du noeud λ, nous écrivons RS(λ) son sibling rank tel que :

$$RS(\lambda) = \mid ST(\lambda) \mid$$

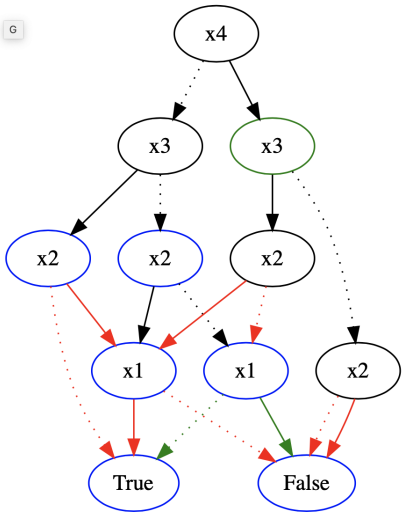

FIGURE 5 – Dag

On considère la figure 5 contenant le dag d'un bdd Δ. Les nil-transitions de Δ sont représentés en rouge, son squelette est le graphe réduit aux transitions en noirs.

La pool du noeud entouré en vert est : $\{x_2, x_2, x_1, x_1, \top, \bot\}$.

Le profil de Δ est : {2,2,4,2,1}.

Nous savons de la manière dont nous définissons les bdd que pour être valide un bdd β doit avoir les propriétés suivantes :

FACT 1.1 : *De la règle r nous faisons la conclusion que deux noeuds de même index ne peuvent avoir les même descendants, autrement ils auraient été fusionnés.*

FACT 1.2 : *A partir de la règle m nous savons qu'un noeud ne peut avoir deux enfants identiques, autrement ce noeud aurait été supprimé.*

5 Comptage

La méthode classique permettant d'énumérer les bdd ayant un nombre de variable k et une taille n est la suivante :

1. Énumérer toute les 2^{2^k} fonctions booléennes et dresser leurs arbres de décision binaires
2. Appliquer le processus de compression pour obtenir leur bdd
3. Filtrer les bdd de taille n
4. Supprimer les bdd réapparaissant plus d'une fois afin de ne conserver qu'une seule de leur occurrence

Nous proposons ici une méthode récursive permettant d'effectuer cette énumération sans passer ni par le processus de compression ni celui du filtrage. Ces deux processus étant hautement coûteux avec une complexité doublement exponentielle $O(2^{2^k})$.

Proposition 1 : Considérons T = (Q', I, r, δ') un squelette avec un ensemble de noeuds Q', racine r et fonction de transition partiel $\delta' : Q' \times \{0, 1\} \to Q' \cup \{\text{nil}\}$. La fonction de transition entière $\delta : Q' \times \{0, 1\} \to Q' \cup \{\top, \bot\}$ est la fonction de transition d'un bdd avec un squelette T si et seulement si pour tout noeuds $\lambda \in Q'$, noté $\lambda 0 = \delta(\lambda, 0)$ et $\lambda 1 = \delta(\lambda, 1)$, la paire $(\lambda 0, \lambda 1)$ satisfait :

1. Si $\delta'(\lambda, 0) \,!= \text{nil}$ et $\delta'(\lambda, 1) \,!= \text{nil}$ alors
$$\lambda\sigma = \delta'(\lambda, \sigma) \text{ pour } \sigma \in \{0, 1\}.$$
2. Si $\delta'(\lambda, 0) = \delta'(\lambda, 1) = \text{nil}$, alors
$$\lambda\sigma \in \text{PT}(\lambda) \text{ pour } \sigma \in \{0, 1\} \text{ et } \lambda 0 \,!= \lambda 1, \text{ et il n'y a aucun noeud } \lambda' \,!= \lambda \text{ avec I}(\lambda)=\text{I}(\lambda') \text{ tel que } \delta(\lambda', \cdot) = \delta(\lambda, \cdot).$$
3. Si $\delta'(\lambda, 0) = \text{nil}$ et $\delta'(\lambda, 1) \,!= \text{nil}$, alors
$$\lambda 0 \in \text{PT}(\lambda) \text{ et } \lambda 1 = \delta'(\lambda, 1).$$
4. Si $\delta'(\lambda, 0) \,!= \text{nil}$ et $\delta'(\lambda, 1) = \text{nil}$, alors
$$\lambda 0 = \delta'(\lambda, 0) \text{ et } \lambda 1 \in \text{PT}(\lambda) \cup \text{TA}(\lambda 0) \setminus \lambda 0 \ .$$

Preuve. Puisque $\delta(\cdot, \cdot)$ doit étendre $\delta'(\cdot, \cdot)$, le cas 1. est triviale. En effet, nous devons étendre la fonction de transition uniquement lorsque l'on a $\delta(\lambda, \sigma) = \text{nil}$. Dans le cas 2., nous devons choisir pour $(\lambda 0, \lambda 1)$ deux noeuds dans $\text{PT}(\lambda)$. De plus $\lambda 0 \mathrel{!}= \lambda 1$ comme déclaré dans FACT 1.2 et un autre noeud avec le même index que λ ne peut pas avoir les même enfants $(\lambda 0, \lambda 1)$ comme déclaré dans FACT 1.1. Dans le cas 3., l'enfant bas doit être choisi dans la pool de λ puisque l'on doit préserver le squelette. Dans le cas 4., l'enfant haut de λ est aussi choisi dans la pool de λ ou dans $\text{TA}(\lambda 0)$ (et doit être diffèrent de $\lambda 0$).

Nous définissons maintenant un procédé permettant de calculer le nombre de bdd correspondant à un squelette donné, nous appellerons le nombre de bdd correspondant à un squelette S le poids de S.

Poids d'un noeud : Considérons un squelette T = (Q', I, δ', r), le poids $\text{wT}(\lambda)$ d'un noeud $\lambda \in$ Q' est le nombre de transitions possibles pour compléter la fonction de transitions partielles $\delta'(\lambda, \cdot)$ et former un bdd valide de squelette T.

Proposition 2 (Poids d'un noeud) : Considérons T = (Q', I, δ', r) un squelette , le poids $\text{wt}(\lambda)$ d'un noeud $\lambda \in$ T est :

$$wt(\lambda) = \begin{cases} 1 & \text{si } \delta'(\lambda, 0) \mathrel{!}= \text{nil et } \delta'(\lambda, 1) \mathrel{!}= \text{nil} \\ pts(\lambda)(pts(\lambda) - 1) - ST(\lambda) & \text{si } \delta'(\lambda, 0) = \delta'(\lambda, 1) = \text{nil} \\ pts(\lambda) + pts0(\lambda) - 1 & \text{si } \delta'(\lambda, 0) = \text{nil et } \delta'(\lambda, 1) \mathrel{!}= \text{nil} \\ pts(\lambda) & \text{si } \delta'(\lambda, 0) \mathrel{!}= \text{nil et } \delta'(\lambda, 1) = \text{nil} \end{cases} \quad (1)$$

Avec $\text{pts}(\lambda) = || \, profil(PT(\lambda)) \, ||$ et $\text{pts0} = || \, profil(TA(\delta'(\lambda, 0))) \, ||$.

Preuve. A partir de la proposition 1 nous énumérons tous les liens possibles (liant à un noeud) nous pouvons utiliser pour compléter la fonction de transition partielle. Dans le premier cas nous ne pouvons compléter que avec les liens déjà existant (comme déclaré dans 1) de proposition 1) le cas est déjà défini). Dans le second cas nous énumérons tous les liens possible de 2) de la proposition 1. Dans le troisième et quatrième cas nous faisons la même chose pour respectivement 3) et 4).

Poids d'un squelette : Considérons un squelette T = (Q', I, δ', r), le poids W(T) de T est le poids cumulé de ses noeuds.

$$W(T) = \prod wt(\lambda) \mid \lambda \in Q'$$

Dans le cas où le poids d'un noeud $\lambda \in Q'$ est nul ou négative nous savons que le squelette T est invalide parce que T n'est le squelette d'aucun BDD. Ceci arrive uniquement lorsque λ a ses deux enfants égaux à nil (puisque l'on a nécessairement $|| \, profil(PT(\lambda)) \, || \geq 2$).

Cette formule peut être aisément décrite de manière récursive, via une traversée en $post_o$ où toutes les informations nécessaires pour calculer le poids du noeud ont déjà été traitées.

Nous notons T_{nk} l'ensemble des squelettes de taille n et de nombre de variables k, et N_{nk} le nombre de bdd de taille n et de nombre de variables k. Pour déterminer N_{nk} tout ce que nous avons à faire est déterminer la totalité des squelettes de taille n et de nombre de nombre de variables k et ensuite déterminer leurs poids, nous les additionnons par la suite.

$$N_{nk} = \sum W(T) \mid T \in T_{nk}$$

6 Combinatoire sur les squelettes

Ce que nous voulions dans la section précédente était d'énumérer tout les bdd de taille n et de nombre de variables k. Pour ce faire nous devons déterminer les squelette de taille n et de nombre de variable k. Il ne s'agit pas d'un processus aisé à informatiser.

Nous définissons ici un processus récursif où nous construisons les squelettes à l'aide de chacun des noeuds composant ceux-ci et dès lors que l'on a déterminé qu'un squelette n'est pas valide on l'ignore.

Nous définissons un squelette Tt en tant que tuple tel que :

$$Tt = T' \cup N_i \cup T'' \mid nil$$

où N_i est un noeud d'index i. Ceci signifie que le squelette est soit réduit à nil soit à une racine et deux squelettes. Nous désignerons dans la suite les squelettes T' et T" qui composent Tt par le terme de sous squelette.

Pour considérer un squelette en particulier nous prendrons en compte trois de ses paramètres. Sa pool, le level rank de sa racine et sa taille.

Ces paramètres sont fondamentaux pour déterminer le poids des squelettes considérés.

Nous utiliserons la notation $T_{m,p,s}$ pour considérer le squelette enraciné en r de taille m, de pool profil p et de level rank s (level rank de r) représenté par un tuple Tt tel décrit plus haut.

Proposition 3 : Pour $T_{m,p,s}$ nous considérons des tailles pour chaque sous squelette T' ∪ T" ∈ T de respectivement i et m – 1 – i, avec $0 \leq i \leq m-1$. Pour chacune des tailles de sous squelette considérée, nous déterminons chacune des valeurs d'indexes que l'on peut assigner à leurs racines.

Nous avons ainsi un algorithme dans lequel nous considérons toute les tailles possible ainsi que les indexes des sous-squelettes, le level rank de la racine étant déjà donné par le pool profil du squelette courant (ainsi que par le profil de T' dans le cas du level rank de T"). k correspond dans la suite à la taille de p.

1. Si m>0
   (a) $T' \in \bigcup_{k_0 \in K} T_{i,p[:k_0-1],p_{k_0-1}}$
   (b) $T'' \in \bigcup_{k_0 \in K} T_{j,p1[:k_0-1],p1_{k_0-1}}$
   (c) $WTP(N_k, T', T'') * W(T') * W(T'')$
2. Si m = 0
   (a) T=nil
   (b) W(T)=1

Avec j = m-1-i, on note p[:i]=$(p_0, p_1 ... p_i)$ pour designer dans un ensemble p que l'on ne considère que les éléments ayant un index plus petit ou égal à i et p_k lorsqu'on désigne l'élément d'index k. On a également p1=p+profil(TA(T')) et K = {0,1,...,k}. Enfin on note W(T) le poids du squelette T et WTP(N,T',T") le poids du noeud N qui a pour fils bas le squelette T' et pour fils haut le squelette T".

Avec notre algorithme, nous ne considérons, pour chaque récursion, uniquement les squelettes ayant les même valeurs m, p et s. En s'inspirant de ce procédé on peut écrire un algorithme COUNT(m,p,s) prenant en argument une taille de squelette m, un profil p et un level rank s et retournant un dictionnaire au sein duquel chaque clé est le profil d'un squelette, et la valeur, la somme des poids des squelettes ayant ce profil.

Quelques exemples d'exécution de la fonction COUNT :

— COUNT(3,(2,0,0),0) nous retourne {(0,1,1,1) : 56, (0,2,0,1) : 2, (0,0,2,1) : 2}
— COUNT(2,(2,0),0) nous retourne {(0,1,1) : 8}
— COUNT(1,(2),1) nous retourne {(0,1) : 1}

Dans le premier cas le dictionnaire retourné nous indique qu'il existe 56 bdd de taille 3 ayant un squelette ayant pour profil (0,1,1,1), 2 ayant un squelette ayant pour profil (0,2,0,1) et encore 2 ayant un squelette ayant pour profil (0,0,2,1).

S'appuyant sur l'algorithme COUNT on définit N(n,k) qui renvoie le nombre de bdd de taille n et d'index k :

$$N(n,k) = \sum_{(t,w) \in COUNT(n-2,(2,0,...,0),0)} w$$

$e^{(k)}$ est une liste de k+1 composants tous nuls sauf le dernier qui est égal à 1 tel que $e^{(k)} = (0, 0, ..., 1)$.

La complexité de l'algorithme COUNT est exponentiel de l'ordre de $O(2^{3k^2/2+k})$.

L'algorithme COUNT s'appuie sur la méthode de comptage et de génération des squelettes que l'on a décrit plus haut. Cet algorithme vérifie tout d'abord si le noeud courant est vide (m=0). Si tel est le cas, on sait qu'on est sur une nil-transition. Si le noeud courant n'est pas vide (m>0) on génère la totalité des sous arbres gauche et droit et déterminons récursivement leurs poids. Nous stockons enfin leur poids dans un dictionnaire selon leurs profil.

```
1: function COUNT(m,p,s)
2:     d ← {}
3:     k ←Len(p)-1                          ▷ Len(p) : renvoie la taille de la liste p
4:     if m=0 then
5:         S ← (∑_{k-1}^{j=0} p_j)(∑_{k-1}^{j=0} p_j − 1)
6:         if S > 0 then
7:             d ← {e^(k) : S}
8:         end if
9:     else
10:        for i ← 0 to m−1 do
11:            d_0 ← {}
12:            if i=0 then d_0 ← {() : ∑_{k-1}^{i=0} p_i}
13:            else
14:                for k_0 ← 1 to k − 1 do d_0 ← d_0∪COUNT(i, p[: k_0 − 1], p_{k_0})
15:                end for
16:            end if
17:            for (l, w_0) ← d_0 do
18:                d_1 ← {}
19:                p' ← p + l
20:                if n-1-i=0 then d_1 ← {() : −1 + ∑_{k-1}^{i=0} p'_i}
21:                else
22:                    for k_1 ← 1 to k − 1 do
23:                        d_1 ← d_1∪COUNT(n − 1 − i, p'[: k_1 − 1], p'_{k_1})
24:                    end for
25:                end if
26:                for (r, w_1) ← d_1 do
27:                    w ← w_1 ∗ w_0
28:                    t ← l + r + e^k
29:                    if t ∈ d then d[t] ← d[t] + w
30:                    elsed ← d ∪ {t : w}
31:                    end if
32:                end for
33:            end for
34:        end for
35:    end if
36:    return d
37: end function
```

7 Unranking

Notre approche pour le unranking s'appuie sur la méthode introduite dans [1] où l'on décompose un arbre binaire en ses sous arbres gauche et droits et en attribuant à chaque décomposition un ordre total pour un bdd de taille et d'index fixé.

En s'inspirant de la Proposition 3 on définie une méthode récursif b permettant de déterminer le nombre d'arbres binaires ayant une certaine taille n.

$$b_n = \begin{cases} 1 & \text{si n} = 0 \\ \sum_{k=0}^{n-1} b_k * b_{n-1-k} & \text{sinon} \end{cases} \tag{2}$$

En s'appuyant sur cette méthode on peut definir une decomposition sur un arbre de taille fixé n. Cette méthode s'appuie sur le fait que b_n est définie en une somme de produit, où chaque élément du produit peut être vu comme la repreésentation d'un arbre ayant un sous arbre gauche et un sous arbre droit de taille fixe. Par exemple $B_7 = 429$, et on peut le décomposer en :

$$B_7 = B_0 * B_6 + B_1 * B_5 + B_2 * B_4 + B_3 * B_3 + B_4 * B_2 + B_5 * B_1 + B_6 * B_0$$

$$429 = 1 * 132 + 1 * 42 + 2 * 14 + 5 * 5 + 14 * 2 + 42 * 1 + 132 * 1$$

Où $B_2 * B_4$ représente un arbre composé d'un sous arbre droit de taille 4 et d'un sous arbre gauche de taille 2.

Ainsi, si je cherche l'arbre n°250 de taille 7, j'obtiens : 250 = 132 + 42 + 28 + 25 + 23. On trouve donc que l'arbre 250 est composé d'un enfant gauche de taille 4 et d'un enfant droit de taille 2. Et plus particulièrement, parmi les 28 arbres ayant cette décomposition 4x2, c'est l'arbre numéro 23 qui nous intéresse.

Pour ensuite retrouver le numéro des sous arbre gauche et droit on procède comme ceci :

— 23 $//B_2$ pour le numero du sous arbre gauche

— 23 % B_2 pour le numero du sous arbre droit

En s'appuyant recursivement sur cette methode, on obtient l'arbre de la figure 6.

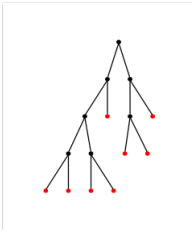

FIGURE 6 – Arbre de taille 7 et de rang 250 généré

En s'inspirant de cette même méthode, on peut définir une fonction DECOMPOSE(r',n',s,p') permettant de générer un bdd de taille n', ayant un profil p'et un rang r'. s est une variable contenant le squelette du bdd a générer.

```
1: function DECOMPOSE(rank, n, squelette, p)
2:     p_target ←list(p)
3:     r ← rank
4:     s ← squelette.get_profil()
5:     k ←len(p_target) − 1
6:     q ← s[: k]
7:     q[0] ← 2
8:     i ← n − 1
9:     while i ≥ 0 do
10:        if i = 0 then
11:            d0 ← () :sum(q, 1, k − 1) + 2
12:        else
13:            d0 ← {}
14:            for k0 ←range(1, k) do
15:                d0.update(count(i, q[: k0], q[k0]))
16:            end for
17:        end if
18:        for (l, w0) in d0.items() do
19:            q_prim ←sum_profil(q, l)
20:            if n − 1 − i = 0 then
21:                d1 ← {() :sum(q_prim, 1, k − 1) + 1}
22:            else
23:                d1 ← {}
24:                for k1 in range(1, k) do
25:                    d1.update(count(n-1-i, q_prim[: k1], q_prim[k1]))
26:                end for
27:            end if
28:            for (h, w1) ind1.items() do
29:                t ←sum_profil(sum_profil(e_k(k),list(l)),list(h))
30:                w ← w0 ∗ w1
31:                if t = p_target then
32:                    if w > 0 then
33:                        r ← r − w
34:                    end if
35:                    if r < 0 then
36:                        r ← r + w
37:                        r0 ← r%w0
38:                        r1 ← r//w0
39:                        return (i, r0, l, r1, h)
40:                    end if
41:                end if
42:            end for
43:            i ← i − 1
```

On décrit ensuite une fonction GENERATE(r,n,s,p) permettant de générer un squelette de taille n, de profil p et de rang r. La variable s est une variable globale qui contiendra en fin d'exécution le bdd généré. Cette fonction prend en argument un squelette s contenant les noeuds initialement dans la pool (ici $\top$ et $\bot$) et crée au fur et à mesure le bdd souhaité en faisant appel à la fonction DECOMPOSE et en mettant à jour le squelette.

```
1: function GENERATE(rank, n, squelette, p_target)
2:     k ←len(p_target) − 1
3:     if n = 0 then
4:         node ← squelette.unrank_singleton(rank)
5:         return node
6:     end if
7:     if n = 1 then lo, hi ← squelette.unrank_pair(rank, k)
8:     else
9:         i, r0, l, r1, h ←DECOMPOSE(rank, n, squelette, p_target)
10:        lo ←GENERATE(r0, i, squelette, l)
11:        if n − 1 − i = 0 then hi.unrank_singleton(r1)
12:            if hi = lo then hi ←GENERATE(r1 − 1, n − 1 − i, squelette, h)
13:            end if
14:        else
15:            hi ←GENERATE(r1, n − 1 − i, squelette, h)
16:        end if
17:    end if
18:    node ← squelette.add_node(rank, k, lo, hi)
19:    return node
```

Dans la fonction GENERATE on fait en sorte d'identifier les noeuds que l'on peut assigner aux nil-transitions du noeud courant. Pour un noeud v ayant deux nil-transitions on vérifie donc avant de lui assigner en tant que haut fils un noeud n et fils bas un noeud m que I(m)<I(v) et I(n)<I(v), on vérifie également qu'il n'existe pas v'∈ squelette tel que v !=v', où v'a pour fils bas m'et pour fils haut n', que m=m'et n=n'.

On fait ensuite de même lorsque le noeud courant a une seul nil-transitions en tant que fils bas ou en tant que fils haut tout en respectant les règles que l'on a déjà décrit pour la génération des bdd.

Pour cela on utilise ces fonctions :

— unrank_singleton(r) : cette fonction retourne un noeud compatible avec le noeud courant de rang r.

— unrank_pair(rank, k) : Cette fonction retourne une pair de noeud dont l'index est strictement inférieur à k, on omet également les pair de noeuds déjà assignés en tant que descendant d'un noeud précèdent de même index.

— add_node(lo, hi, k) : Cette fonction crée le noeud d'index k et ayant pour descendants bas et haut lo et hi.

Enfin on décrit une fonction GEN_BDD(r,n,k) permettant de générer un bdd de taille n, d'index k et de rang r en utilisant la fonction GENERATE décrite précédemment.

```
1: function GEN_BDD(rank, n, k)
2:     squelette ←Squelette(k)
3:     r ← rank
4:     p ← [0] ∗ k
5:     p[0] ← 2
6:     d ←count(n − 2, p, 0)
7:     for (t, w) in d.items() do
8:         r ← r − w
9:         if r < 0 then :
10:            r ← r + w
11:            generate(r, n − 2, squelette, t)
12:            return squelette
13:
```

Squelette est ici une classe d'objet permettant de représenter un squelette. Voir Annexe 10.1 pour une représentation en python de la classe squelette.

La méthode du unranking nous permet de mettre au point un générateur aléatoire uniforme de bdd.

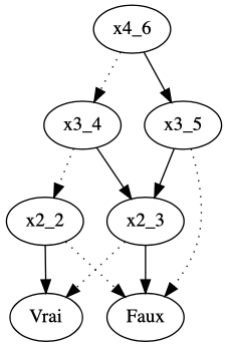

FIGURE 7 – Graphe(A) rang = 3104

L'ensemble dans lequel on a choisi le squelette du graphe(A) de la figure 7 est : {(0, 2, 1, 1, 1) : 584, (0, 1, 2, 1, 1) : 1256, (0, 2, 2, 0, 1) : 74, (0, 1, 1, 2, 1) : 1112, (0, 2, 0, 2, 1) : 74, (0, 0, 2, 2, 1) : 74}.

Avec un rang=3104, on déduit le profile du squelette du graphe est (0, 0, 2, 2, 1). Puisqu'en faisant la somme 584+1256+74+1112+74 = 3100, on remarque que l'on a pas atteint 3104, le seul choix possible reste le dernier.

Avec un rang=3104 et le profile de squelette généré on se retrouve avec x1=0 ; x2=2 ; x3=2 ; x4 = 1 et le graphe de la figure 7.

8 Tests Quickcheck

8.1 Combinaison de BDD(fonction melding)

Melding fonction est une combinaison de deux BDD, dont le résultat est la conjonction logique ET. Cette fonction que nous utilisons provient de l'ouvrage de KNUTH [5]. Le code consiste à combiner les deux BDD en comparant les indexes de noeuds, pour deux noeuds de même index, en résulte un noeud commun. le BDD (C 13) est une combinaison(melding) des BDD (A 7) et (B 12).

$$\alpha \diamond \alpha' = \begin{cases} (v,\ l \diamond l',\ h \diamond h'), & \text{if } v = v'; \\ (v,\ l \diamond \alpha',\ h \diamond \alpha'), & \text{if } v < v'; \\ (v',\ \alpha \diamond l',\ \alpha \diamond h'), & \text{if } v > v'. \end{cases}$$

FIGURE 8 – Combinaison de $\alpha(v, l, h)$ et $\alpha'(v', l', h')$

```python
# This is a methode combining 2 root of spine (node)
def melding(node1, node2, label=2):
    if (node1.index == 0 and  node2.index == 0):
        if node1.label * node2.label == 1:
            return label+1, Node(1,0,0,-1,-1)
        else:
            return label+1, Node(0,0,0,-1,-1)
    if node1.index == node2.index:
        lab1,n1 = melding(node1.left,node2.left,label+1)
        lab2,n2 = melding(node1.right,node2.right,lab1+1)
        return lab2,Node(label,0,node1.index,n1,n2)

    elif node1.index > node2.index:
        lab1,n1 = melding(node1.left,node2,label+1)
        lab2,n2 = melding(node1.right,node2,lab1+1)
        return lab2,Node(label,0,node1.index,n1,n2)

    elif node1.index < node2.index:
        lab1,n1 = melding(node1,node2.left,label+1)
        lab2,n2 = melding(node1,node2.right,lab1+1)
        return lab2,Node(label,0,node2.index,n1,n2)

    else:
        raise Exception("Fail")
```

FIGURE 9 – fonction melding

8.2 Stratégie de Test

Dans nos tests, nous utilisons Quickcheck pour python. Nous intégrons notre générateur dans des tests Quickcheck.

```python
from hypothesis.strategies._internal.core import *
from hypothesis.strategies._internal.strategies import SearchStrategy
import random


class ListUniformBoundedIntStrategy(SearchStrategy):
    """A strategy for providing integers in some interval with inclusive
    endpoints."""

    def __init__(self, start, end, n):
        SearchStrategy.__init__(self)
        self.start = start
        self.end = end
        self.n = n

    def __repr__(self):
        return "UniformBoundedIntStrategy(%d, %d)" % (self.start, self.end)

    def do_draw(self, data):
        #return d.integer_range(data, self.start, self.end)
        return [random.randint(self.start, self.end) for n in range(self.n)]


@cacheable
@defines_strategy_with_reusable_values
def list_integers(min_value: int, max_value: int, n: int) -> SearchStrategy[int]:
    """Returns a strategy which generates uniformly integers.

    If min_value is not None then all values will be >= min_value. If
    max_value is not None then all values will be <= max_value.

    We suppose min_value <= max_value
    """

    return ListUniformBoundedIntStrategy(min_value, max_value, n)
```

FIGURE 10 – génération uniforme d'entiers

À partir d'un ensemble de 10 entiers générés uniformément10, nous construisons un ensemble BDD Pour chaque couple (A,B) de cet ensemble nous les combinons avec la fonction fonction melding pour obtenir un BDD C. Pour une confirmation du test nous comparons les tables de vérités des BDD A,B,C par la formule suivante : table(A) ET(logique) table(B) égal à table(C) 11.

```python
@given(list_integers(min_rank,max_rank,nb_element))
def test_rank(R):
    Spines = []
    for r in R:
        Spines.append(gen_bdd(r,n,k).root())
    for i in range(0,len(Spines)):
        for j in range(i,len(Spines)):

            tA   = truth_table(Spines[i])
            tB   = truth_table(Spines[j])

            _,ml = melding(Spines[i],Spines[j])
            tC = truth_table(ml)

            assert and_truth_table(tA,tB) == tC
            print('passed '+str(j)+'<==>'+str(i))
```

FIGURE 11 – Tests des tables de vérités

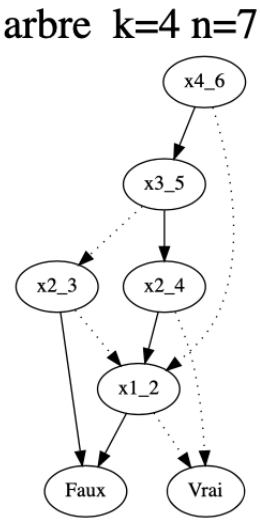

FIGURE 12 – BDD(B) rang(rank) = 1577

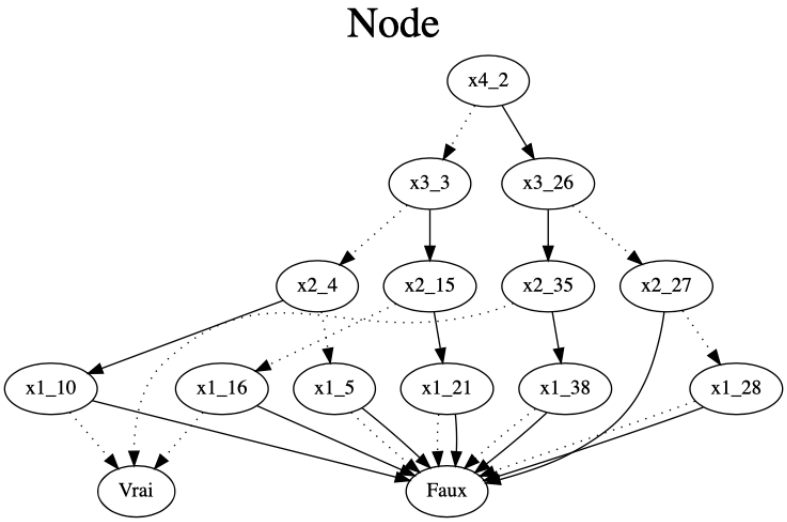

FIGURE 13 – BDD(C) combinaison(melding) de A et B

9 Conclusion

Il ne fait aucun doute que la méthode présenté dans ce rapport offre un gain en temps et en espace gigantesque pour la résolution du problème présenté, la génération de bdd. Nous avons vu que même avec un matériel limité, nous pouvions générer un grand nombre de bdd en un temps record. Ce qui n'était pas possible en pratique avec la méthode standard.

Cette génération en plus de sa simplicité, présente d'autres gains tels que la possibilité de l'adapter selon nos besoins. Nous pouvons ainsi l'adapter pour pouvoir générer d'autre types d'arbre de décisions. Mais comme nous l'avons vu ce gain ne se limite pas a la seule génération de bdd. Cette méthode offre une toute nouvelle manière d'appréhender ainsi que que de nouveaux moyens d'études des Binary Decision Diagram. Dans ce rapport nous avons presentés la maniere avec laquelle nous avons incorporé cette méthode dans l'outil quick-check.

On note également que l'on pourrait encore améliorer cette méthode, par exemple en faisant en sorte de ne considérer que les squelettes valides, la generation des squelettes etant la partie la plus lourde de notre code, ou en améliorant le processus de comptage.

10 Annexe

10.1 Annexe 1

Description en python de la classe squelette.

```python
class Squelette:
    """ Une classe pour construire un squelette
    """
    def __init__(self,k):
        self.squelette  = [[] for i in range(k+1)]
        self.label      = 2
        self.profile    = [0]*(k+1)
        self.forbiden   = {}
        self.profile[0] = 2
        self.squelette[0].append(Node(0,-1,0,-1,-1))
        self.squelette[0].append(Node(1,-1,0,-1,-1))

    def print_squelette(self):
        print('squelette')
        for i in range(len(self.squelette)):
            print('indexe='+str(i))
            for j in range(len(self.squelette[i])):
                print("     "+str(self.squelette[i][j].label))

    def print_profile(self):
        print('profile',end='')
        print(self.profile)

    def print_forbiden(self):
        print('forbiden')
        for i in self.forbiden.keys():
            print('index='+str(i))
            for x in self.forbiden[i].keys():
                print('     x='+str(x.label)+'(index='+str(x.index)+') -> y-values=',end='')
                for y in self.forbiden[i][x]:
                    print(str(y.label)+'(index='+str(y.index)+')',end=',')
                print()

    def root(self):
        return self.squelette[len(self.squelette)-1][0]

    def size(self):
        return sum(self.profile,0,len(self.profile)-1)

    def variable(self):
        return len(self.profile)-1

    def affiche(self):
        g = ("arbre "
        +" k="
        +str(self.root().index)
        +" n="
        +str(sum(self.profile,0,len(self.profile)-1)))

        dot = 'digraph {graph [label="'+g+'", labelloc=t, fontsize=30];\n'
        dot += '1[label=Vrai];0[label=Faux];'
        return dot + self.root().dot_ch() + '}'
```

FIGURE 14 – partie 1 classe squelette

```
    def afficheNode(self,node):
        g =  'Node'
        dot = 'digraph {graph [label="'+g+'", labelloc=t, fontsize=30];\n'
        dot += '1[label=Vrai];0[label=Faux];'
        return dot + node.dot_ch() + '}'

    def stocke(self):
        dot = self.affiche()

        fic = open("arbre_taille_"
        + str(self.size())
        + "_variable_"
        + str(self.variable())
        +".dot", 'w')

        fic.write(dot)
        fic.close()
        return None

    def unrank_singleton(self,rank):
        """ Suivre l'ordre < pour le rank
        """
        r = rank
        for i in range(len(self.squlette)):
            for j in range(len(self.squelette[i])):
                node = self.squelette[i][j]
                if r == 0:
                    return node
                r = r - 1
        raise Exception('unrank_singleton fail index')
```

FIGURE 15 – partie 2 classe squelette

```
def unrank_pair(self,rank,k):
    r = rank
    for i in range(k):
        for a in range(len(self.squelette[i])):
            n = self.squelette[i][a]
            for j in range(0,k):
                for b in range(len(self.squelette[j])):
                    x = n
                    y = self.squelette[j][b]

                    if x == y:
                        continue

                    constructed = True
                    if  (k not in self.forbiden) or (x not in self.forbiden[k]) or (y not in self.forbiden[k][x]):
                        constructed = False

                    if constructed:
                        r = r - 1
                        continue

                    if not constructed:
                        r = r - 1

                    if r < 0:
                        return (x,y)
    raise Exception('unrank_pair fail')

def get_rank(self,i):
    return i.rank

def add_node(self,rank,index,left,right):
    n = Node(self.label,rank,index,left,right)
    self.squelette[index].append(n)
    self.label += 1
    self.profile[index] += 1
    if index not in self.forbiden:
        self.forbiden[index] = {}
    if n.left not in self.forbiden[index]:
        self.forbiden[index][n.left] = []
    self.forbiden[index][n.left].append(n.right)
    return n

def get_profile(self):
    return self.profile
```

FIGURE 16 – partie 3 classe squelette

Références

[1] Herbert S. Wilf Albert Nijenhuis. “Combinatorial algorithms : An update”. In : *SIAM* (1989).

[2] R.E Bryant. “Graph-based algorithms for boolean function manipulation”. In : *IEEE Trans. Computers* 35.8 (1986), p. 677-691. doi : https://doi.org/10.1109/TC.1986.1676819.

[3] Didier Verna Jim Newton. “A theoretical and numerical analysis of the worst-case size of reduced ordered binary decision diagrams”. In : *ACM Transactions on Computational Logic* 20.6 (2019), p. 1-36. doi : https://doi.org/10.1145/3274279.

[4] Antoine Genitrini Julien Clément. “Binary Decision Diagrams : from Tree Compaction to Sampling”. In : (2020).

[5] Donald E. Knuth. *The Art of Computer Programming*. T. 4A. Addison-Wesley, 2011. Chap. Combinatorial Algorithms. isbn : 9780201038040.

[6] C. Y. Lee. “Representation of Switching Circuits by Binary-Decision Programs”. In : *Bell System Technical Journal* 38 (1959), p. 985-999. doi : https://doi.org/10.1002/j.1538-7305.1959.tb01585.x.

[7] P. Sipala P. Flajolet et J.-M. Steyaert. “Analytic variations on the common subexpression problem [In Automata, languages and programming (Coventry, 1990)]”. In : *Lecture Notes in Comput. Sci.* 443 (1990), p. 220-234. doi : https://doi.org/10.1007/BFb0032034.

[8] Claude Shannon. *Trans. Amer. Inst. Electrical Engineers*. T. 57. 1938, p. 713-723.

[9] Ingo Wegener. *Branching Programs and Binary Decision Diagrams : Theory and Applications*. Sous la dir. de SIAM. Monographs on Discrete Mathematics and Applications (Book 4). Society for Industrial et Applied Mathematics, 2000. isbn : 9780898714586. doi : https://doi.org/10.1137/1.9780898719789.