brouillon-pstl-bdd.v4

Alpha DIALLO, Lamine KEITA, Sacha MEMMI

Avril 2020

Contents

1 Abstract

Les fonction 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.

Les structures avec lesquelles nous travaillons dans ce papier sont ce que l'on appelle des robdd, reduced ordered binnary decision diagram. Ils représentent des fonctions booléennes en tant que diagramme avec leurs noeuds ordonnés et le nombre de ces noeuds réduit. Ils sont la méthode préférée pour représenter les fonctions booléenne en particulier en informatique. Quelques ligne de codes dans n'importe quel langage sont suffisantes pour les représenter.

En particulier nous cherchons à énumérer le nombre de robdd unique représentant une fonction ayant un nombre définie de variable 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 methode standard de génération des robdd, nous donnons un nouveau moyen d'obtenir ces structures. Nous obtenons par cette nouvelle méthode un générateur uniforme de robdd.

2 Introduction

L'idée derrière les diagrammes binaire a été introduite par Shannon en 1938 dans [9]. 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. Plus particulièrement les circuits électroniques sont devenus si compliqués que les robdd sont à peine suffisant pour les traiter comme l'a noté Knuth il y a de ca plus de 10 ans [6].

Les fonctions booléennes ont de très nombreux usages, dans [3] P. Camion convertie de nombreux problemes du domaine des entiers en problèmes équivalent en algèbre booléenne et dans ce même papier R. Fortet a introduit l'idée de colorer un graphe selon 4 couleurs (theoreme des quatres couleurs [8]) en assignant deux variables booléenne à chaque vertex.

Un bdd (binary decision diagram) est un diagramme de décision binaire pouvant représenter une fonction booléenne, l'utilité des bdd ne se limite a représenter des fonctions booléenne mais dans ce papier nous nous limiterons à celle ci. Un robdd (reduced ordered binary decision diagram) est comme son nom l'indique la forme réduite et ordonné d'un bdd. Dans la literrature le terme bdd est presque toujours utilisé pour évoquer un robdd, nous différencierons tout d'abord entre ces deux termes pour mettre l'accent sur l'aspect réduction. Les bdd que nous considerons dans ce papier sont toujours ordonné, quand on parle de bdd nous faisons en fait référence à un 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 dans la littérature scientifique, et de nombreuses autres variantes de robdd ont été crées depuis [10].

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.

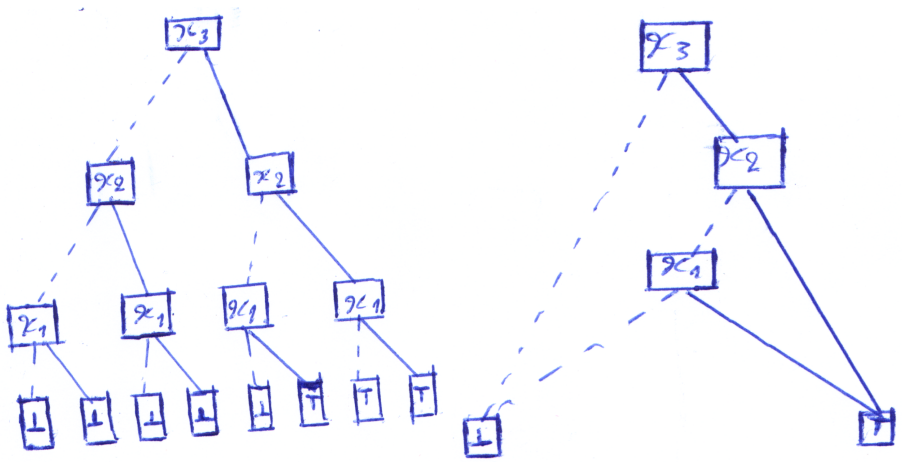

Figure 1: Arbre représentant un bdd à gauche et dag représentant le robdd qui lui correspond à droite

Un exemple d'arbre représentant un bdd et un dag représentant le robdd correspondant à ce bdd sont donné dans la figure 1. 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.

En plus de l'utilité évidente des fonctions booléennes ont peut aussi noter que les bdd sont devenus aussi populaire parce que comme Knuth l'a noté dans [6] les bdd simples sont relativement facile à fusionner pour obtenir des bdd plus complexes.

Le but de ce papier est de présenter un nouveau procédé permettant d'énumérer des robdd sans passer par le processus de compression.

Le processus de compression est relativement simple avec une complexité P (polynomial), mais la méthode standard de génération des bdd d'un nombre donné de variable k utilisé jusqu'ici était impraticable pour $k > 4$ [5]. 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.

$ICICHOISIRBONPARAGRAPHESELONCOMPUTATION$

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.

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

Avec la méthode standard introduite dans [5] les robdd de taille proche de la taille maximal étaient crées avec une grande probabilité [7], 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 certaine classes de propriétés.

Dans ce papier 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 basique avec lesquelles nous travaillons, les bdd et les robdd. Aussi bien que comment nous les construisons.

"A Boolean function is a function in mathematics and logic whose arguments, as well as the function itself, assume values from a two-element set (usually {0,1})."

- Wikipedia

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

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 indexe qui détermine son ordre dans le bdd. Les noeuds sont traités dans l'ordre décroissants de leurs indexes.

La manière dont nous ordonnons les noeuds est importante comme noté dans [5] et trouver l'ordre le plus efficace est un problème NP (non-polynomial). Nous savons que plus un bdd a de variable plus sa taille tend à se rapprocher de celles du robdd qui lui correspond.

La chose importante ici est que les bdd vu en pratique ont tendance à avoir au moins un ordonnancement efficace de leurs noeuds où la taille de leurs robdd est bien inférieur à la leurs [4]. Raison pour laquelle déterminer l'ordre le plus efficace est aussi important.

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

Pour exécuter un bdd 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'indexe du noeud courant. Si la variable d'indexe 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 papier nous appellerons feuille, de valeur $\top$ (true) ou $\bot$ (false) qui est la valeur de retour de la bdd pour les variables considérées.

Pour le bdd de la figure 1 si l'on donnait les variables 101 en tant qu'entrées elle retournerait $\top$ (true).

La méthode classique pour obtenir un robdd est de réduire un bdd non réduit. Définissons maintenant comment on effectue cette réduction.

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 de 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 ordre de traversés peuvent être considérés, $post_{ordre}$ et pre_{ordre}. Dans le $post_{ordre}$ nous visitons tous d'abord l'enfant bas du noeud courant, puis son enfant haut et enfin le noeud lui-même. Dans le pre_{ordre} nous visitons tous d'abord le neoud courant, puis son enfant bas, et enfin son enfant haut.

Nous choisissons ici de considérer l'ordre de traversé $post_{ordre}$ 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 traversé 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 que la traversé de Δ achevé, le dag résultant est le robdd de f.

Voir Appendix 10.1 pour l'algorithme de compression.

Définissons maintenant formellement un robdd.

Considérons un robdd Δ = (Q, I, r, δ). Q est l'ensemble de noeuds que l'on obtient par une traversé en $post_{ordre}$ de Δ. r est la racine de Δ, I est la fonction qui associe pour tout noeud $\lambda \in$ Q son indexe 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 $post_{ordre}$ et [feuille2] la seconde)
- $I = Q \to \{0, \ldots, k\}$ où k est l'indexe de r
- $\delta = (Q \setminus \{\top, \bot\}) * \{0, 1\} \to$ Q

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

4 (RO)BDD : propriétés

Dans cette section nous définissons des propriété que l'on assigne aux bdd. Pour ce faire nous mettant en place un ensemble de notation et d'opérations. Toute les notations utilisés dans cette section seront utilisés dans le reste de ce papier. Les opérations que nous définissons pour les liste sont également applicables aux ensembles.

Pour deux noeuds α et β d'un dag Φ nous écrivons $\alpha \Leftarrow_{post} \beta$ si le neoud α est visité avant le noeud β en $post_{ordre}$ lors de la traversé de Φ, et $\alpha \Leftarrow_{pre} \beta$ si le noeud α est visité avant le noeud β en pre_{ordre} lors de la traversé 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'élément dans l, et $\mid\mid l \mid\mid$ la somme de tous ses éléments.

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

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

Spine d'un bdd : Considérons un bdd Δ = (Q, I, r, δ) enraciné en r. Le spine 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 $post_{ordre}$ de Δ en faisant omission des deux feuilles. Les liens du spine 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.

Décrit simplement un spine est un bdd auquel on retire les deux feuilles et les pointeurs.

Dans le reste de ce papier 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 spine 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 transition 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:

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

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

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

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

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

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

Voir Appendix 10.2 pour des exemples de certaines propriétés vu dans cette section.

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

FACT 1.1 : *De la règle r nous faisons la conclusion que deux noeuds de même indexe 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 diagrammes binaire
2. Appliquer le processus de compressions pour obtenir leurs 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 leurs récurrence

Nous proposons ici une méthode récursive permettant d'effectuer cette énumération sans passer par le processus de compression.

Proposition 1 : Considérons T = (Q', I, r, δ') un spine 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 spine 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 $\delta'(\lambda, 0)$!= nil et $\delta'(\lambda, 1)$!= nil alors
$$\lambda\sigma = \delta'(\lambda, \sigma) \text{ pour } \sigma \in \{0, 1\}.$$
2. Si $\delta'(\lambda, 0) = \delta'(\lambda, 1) = \text{nil}$, alors
$$\lambda\sigma \in \text{PT}(\lambda) \text{ pour } \sigma \in \{0, 1\} \text{ et } \lambda 0 \mathrel{!}= \lambda 1, \text{ et il n'y a aucun noeud } \lambda' \mathrel{!}= \lambda \text{ avec } \text{I}(\lambda)=\text{I}(\lambda') \text{ tel que } \delta(\lambda', \cdot) = \delta(\lambda, \cdot).$$
3. Si $\delta'(\lambda, 0) = \text{nil}$ et $\delta'(\lambda, 1)$!= nil, alors
$$\lambda 0 \in \text{PT}(\lambda) \text{ et } \lambda 1 = \delta'(\lambda, 1).$$
4. Si $\delta'(\lambda, 0)$!= 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 puisque nous devons étendre la fonction de transition uniquement où 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 indexe 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 choisie dans la pool de λ puisque l'on doit préserver le spine. 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 spine donné, nous appellerons le nombre de bdd correspondant à un spine S le poids de S.

Poids d'un noeud : Considérons un spine T = (Q', I, δ', r), le poids wT(λ) d'un noeud $\lambda \in$ Q' est le nombre de transition possible pour compléter la fonction de transition partielle $\delta'(\lambda, \cdot)$ et former un bdd valide de spine T.

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

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

Avec pts(λ) = $|| \, profile(PT(\lambda)) \, ||$ et pts0 = $|| \, profile(TA(\delta'(\lambda, 0))) \, ||$.

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 définie). 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).

Le poids d'un sous arbre est le poids cumulé de ses noeuds.

Poids d'un spine : Considérons un spine 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 spine T est invalide parce que T n'est pas le spine du moindre bdd. Ceci arrive uniquement lorsque λ a ses deux enfants égaux à nil (puisque l'on a nécessairement $|| \, profile(PT(\lambda)) \, || \geq 2$).

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

Nous notons T_{nk} l'ensemble des spine de taille n et de nombre de variable k, et N_{nk} le nombre de bdd de taille n et de nombre de variable k. Pour déterminer N_{nk} tout ce que nous avons à faire est déterminer la totalité des spine de taille n et de nombre de nombre de variable k et ensuite déterminer leurs poids, nous faisons ensuite l'addition de leurs poids.

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

6 Combinatoire sur les spine

Ce que nous voulions dans la section précédente était d'énumérer tout les bdd de taille n et de nombre de variable k. Pour ce faire nous devons déterminer les spine de taille n et de nombre de variable k. Ceci n'est un processus aisé à informatiser.

Nous définissons ici un processus récursive où nous construisons les spine par chacun de leurs noeuds et des que l'on a déterminé que ses spine ne sont pas valides on les ignore.

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

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

où N_i est un noeud d'indexe i, ceci signifie que le spine est soit reduit à nil soit à une racine et deux spine. Nous designerons dans la suite les spine T' et T" qui composent Tt par le terme de sous spine.

Pour considerer un spine enraciné en r en particulier nous prendrons en compte trois de ses paramètres. Sa pool pT(T), le level rank de sa racine ST(r) et sa taille que nous designerons par $|| T ||$.

Ces paramètre sont important parce qu'ils sont ce dont on a besoin pour déterminer le poids des spine que nous considérons.

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

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

Nous avons ainsi un algorithme où nous considérons toute les tailles possible ainsi que les indexes des sous-spines, le level rank de la racine étant déjà donné par le pool profile du spine courant (ainsi que par le profile 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) $W(T) = WTP(N_k, T', T'') * W(T') * W(T'')$
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 considere que les elements ayant un indexe plus petit ou egale à i et p_k lorsqu'on designe l'element d'indexe k. On a egalement p1=p+profile(TA(T')) et K = {0,1,...,k}. Enfin on note W(T) le poids du spine T et WTP(N,T',T") le poids du noeud N qui a pour fils bas le spine T' et pour fils haut le spine T".

Avec algorithme nous ne considérons pour chaque récursion que les spine de 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 spine m, un profile p et un level rank s et retournant un dictionnaire où chaque clé est le profile d'un spine et la valeur la somme des poids des spine ayant ce profile.

Voir 10.3 pour un exemple de l'algorithme count.

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

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

7 Unranking

Notre approche pour le unranking s'appuie sur la méthode introduite dans [1] où chaque composé a un ordre definie dans l'ensemble et chacun de ces objets peut être décomposé en tant qu'ensemble de sous objets. Dans la precomputation on s'appuie sur ce que l'on a déjà défini pour déterminé le nombre d'objet (ici bdd) ayant une certaine taille afin de leur assigner à chacun un ordre.

[BESOIN DONNER PLUS APRES FIN ALGORITHME!!!] Voir Appendix 10.4 pour l'algorithme de unranking ainsi que celui de precomputation.

La methode du unranking nous permet de mettre au point un generateur aleatoire uniforme de bdd.

8 Quickcheck

[décrire quick test et resultats]

9 Conclusion

En tant que conclusion nous notons que les méthodes introduites dans ce papier peuvent être étendues à d'autre types de bdd tel que les zbdd (zero-suppressed decision diagram), ou encore d'autre types d'arbre binaire ne traitant pas de fonctions booléennes. Nous aimerions également faire remarquer que nous pourrions tenter de simplifier l'algorithme de comptage en le rendant plus efficace, ou en faisant en sorte de ne considérer que les spine valide, nous considérons la complexité en temps comme bien plus importante que celle en espace mémoire. [A COMPLETER!!!!]

10 Appendix

10.1 Appendix 1

On présente ici l'algorithme de compression en pseudo code

Require:

Le noeud initial est celui de la racine de l'arbre à compresser
table est une valeur globale stockant un ensemble de sous arbre et initialement vide
count est un compteur globale initialisé à 0

```
1: function Compression(n)
2:     if is_terminal(n) then      ▷ is_terminal(n): Vérifie si n est une feuille
3:         if Label(n)=False then          ▷ Label(n): Renvoie le label de n
4:             return 0
5:         else
6:             return 1
7:         end if
8:     else
9:         lo ← Left(n)          ▷ Left(n): Renvoie le sous arbre gauche de n
10:        hi ← Right(n)
11:        lo ← Compression(lo)
12:        hi ← Compression(hi)
13:        if lo = hi then
14:            return lo
15:        else
16:            triple ← (INDEX(n), lo, hi)      ▷ Index(n): Renvoie l'index de n
17:            value_found ←FIND(triple, table)   ▷ Find(n): Renvoie l'arbre
   triple si il existe deja dans la table, None sinon
18:            if value_found = None then
19:                uid ← count
20:                INSERT((triple, uid), table)   ▷ INSERT((triple, uid), table):
   Insert triple dans la table avec l'identifiant uid.
21:                count ← count + 1
22:                return uid
23:            else
24:                return value_found
25:            end if
26:        end if
27:    end if
28: end function
```

On peut faire remarquer que cette algorithme a une complexité polynomial O(n) où n est le nombre de noeud de l'arbre a compresser.

10.2 Appendix 2

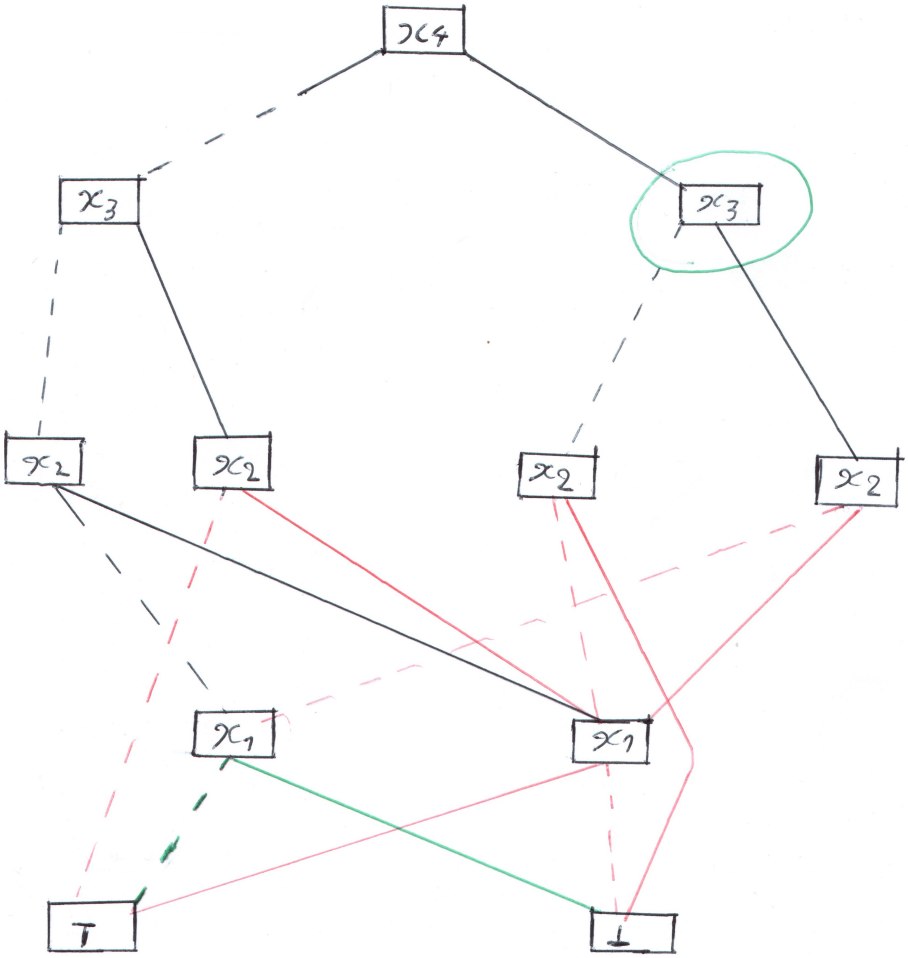

Figure 2: Dag d'une robdd

On considère dans la figure 2 le dag d'un robdd. Les pointeurs sont représentés en rouge. Le spine de ce robdd 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 profile de ce robdd est : $\{(2,2,4,2,1)\}$.

10.3 Appendix 3

On decrit l'algorithme COUNT en pseudo code ci-dessous.

```
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
13:                d_0 ← {() : ∑_{k-1}^{i=0} p_i}
14:            else
15:                for k_0 ← 1 to k − 1 do
16:                    d_0 ← d_0 ∪ COUNT(i, p[: k_0 − 1], p_{k_0})
17:                end for
18:            end if
19:            for (l, w_0) ← d_0 do
20:                d_1 ← {}
21:                p' ← p + l
22:                if n-1-i=0 then
23:                    d_1 ← {() : −1 + ∑_{k-1}^{i=0} p'_i}
24:                else
25:                    for k_1 ← 1 to k − 1 do
26:                        d_1 ← d_1 ∪ COUNT(n − 1 − i, p'[: k_1 − 1], p'_{k_1})
27:                    end for
28:                end if
29:                for (r, w_1) ← d_1 do
30:                    w ← w_1 ∗ w_0
31:                    t ← l + r + e^k
32:                    if t ∈ d then
33:                        d[t] ← d[t] + w
34:                    else
35:                        d ← d ∪ {t : w}
36:                    end if
37:                end for
38:            end for
39:        end for
40:    end if
41:    return d
42: end function
```

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

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

10.4 Appendix 4

[donner pseudo code unranking et precomputation]

10.5 Appendix 5

[donner graphique temps methode classique, la notre]

References

[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), pp. 677–691. DOI: https://doi.org/10.1109/TC.1986.1676819.

[3] *Cahiers du Centre d'Etudes et de Recherches Opérationnelle 1*. Université libre de Bruxelles. Centre d'études de recherche opérationnelle, 1959.

[4] Anand Srivastav Clemens Gröpl Hans Jürgen Prömel. "Ordered binary decision diagrams and the shannon effect". In: *Discrete Applied Mathematics* 142.1 (2004), pp. 67–85. DOI: https://doi.org/10.1016/j.dam.2003.02.003.

[5] 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), pp. 1–36. DOI: https://doi.org/10.1145/3274279.

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

[7] P. Sipala P. Flajolet and 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), pp. 220–234. DOI: https://doi.org/10.1007/BFb0032034.

[8] Benoît Rittaud. *Les Mathématiques*. Editions Le Cavalier Bleu, 2008, p. 34. ISBN: 9782846701969.

[9] Claude Shannon. *Trans. Amer. Inst. Electrical Engineers*. Vol. 57. 1938, pp. 713–723.

[10] Ingo Wegener. *Branching Programs and Binary Decision Diagrams: Theory and Applications*. Ed. by SIAM. Monographs on Discrete Mathematics and Applications (Book 4). Society for Industrial and Applied Mathematics, 2000. ISBN: 9780898714586. DOI: https://doi.org/10.1137/1.9780898719789.