Arbre de décision binaire - *ROBDD* en anglais
Rapport pour le projet Mathématique et Informatique de L3 de 2018-2019

Sébastien Palmer et Xavier Durand

Encadré par Sedki Boughattas

19 mai 2019

Table des matières

Introduction

Présentation du sujet

Ce document est un projet de Mathématiques et d'Informatique suivi et encadré par un enseignant chercheur à l'Université Paris Diderot.
Nous allons ici traiter des arbres de décisions binaires. Le projet se base sur l'article de Henrik Reif Andersen « *An Introduction to Binary Decision Diagrams* ».
Ce document traite de ce qu'est un arbre de décision binaire, et de la représentation de toutes expressions propositionnelles en un arbre de décision binaire. Enfin, il expose un algorithme permettant de construire l'arbre de décision binaire correspondant à une expression propositionnelle quelconque.

Qu'est ce qu'une expression propositionnelle

Dans un premier temps, on va présenter ce qu'est une expression propositionnelle.

Variable propositionnelle

Une variable propositionnelle correspond à une variable comme en Mathématiques. Cependant, son ensemble de définition correspond à l'ensemble $\{0, 1\}$, où ici on peut apparenter 0 à *faux* et 1 à *vrai*.
Pour lier différentes variables propositionnelles entre elles, on va introduire des connecteurs logiques.

Connecteurs logiques

On va présenter ici les cinq symboles logiques les plus utilisés en logique propositionnelle :

- la négation : on la notera $\neg$. Elle correspond à une fonction ne prenant qu'une expression propositionnelle en argument et renvoie vrai si et seulement si l'argument est faux.
- la conjonction : on la notera $\wedge$. Elle correspond à une fonction prenant deux expressions propositionnelles en argument et renvoie vrai si et seulement si les deux arguments sont vrais.
- la disjonction : on la notera $\vee$. Elle correspond à une fonction prenant deux expressions propositionnelles en argument et renvoie faux si et seulement si les deux arguments sont faux (donc renvoie vrai si au moins l'un des deux arguments est vrai).
- l'implication : on la notera $\Rightarrow$. C'est une fonction prenant deux expressions propositionnelles en arguments et renvoyant faux si et seulement si le premier argument est vrai et le deuxième est faux.
- l'équivalence : on la notera $\Leftrightarrow$. C'est une fonction prenant deux expressions propositionnelles en argument et renvoyant vrai si et seulement si les deux arguments ont la même valeur de vérité.

Voici les tables de vérités de chacun des connecteurs logiques :

	$\neg$
0	1
1	0

$\wedge$	0	1
0	0	0
1	0	1

$\vee$	0	1
0	0	1
1	1	1

$\Rightarrow$	0	1
0	1	1
1	0	1

$\Leftrightarrow$	0	1
0	1	0
1	0	1

FIGURE 1 – Tables de vérités des cinq connecteurs logiques

Ces connecteurs logiques ne sont bien évidemment pas les seuls à exister, mais ce sont ceux qu'on accepte dans notre algorithme qu'on présentera plus tard.

Expression propositionnelle

Une expression propositionnelle correspond à une suite de variables propositionnelles liées par des connecteurs logiques.
Voici un exemple d'expression propositionnelle :

$$(x \wedge y) \vee ((\neg z) \Rightarrow (x \Leftrightarrow t))$$

avec x, y, z et t des variables propositionnelles.

Par soucis de lisibilité, lorsqu'on écrira une expression propositionnelle, on ajoutera des parenthèses.
Prenons l'expression $x \wedge y \vee 1$. On ne va pas avoir la même table de vérité si on écrit $(x \wedge y) \vee 1$ (qui sera toujours vrai) et si on écrit $x \wedge (y \vee 1)$ (qui sera faux si x est faux).
Cependant, il y a des expressions qui sont commutatives, et donc on peut omettre les parenthèses dans ces cas précis. Ces expressions commutatives sont la succession de conjonction et la succession de disjonction. Les tables de vérité des deux expressions propositionnelles suivantes ne change pas quelque soit la position des parenthèses :

$$x \wedge y \wedge z \qquad et \qquad x \vee y \vee z$$

Qu'est ce qu'une ROBDD (présentation rapide)

Nous allons ensuite introduire la notion d'arbre de décision binaire, on le notera *ADB*. Cela se présente de la sorte :

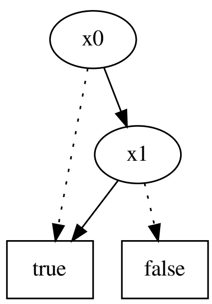


FIGURE 2 – ADB d'un implication simple : $x_0 \Rightarrow x_1$

Cet arbre se lit de haut en bas : on commence par évaluer la première variable, soit à *vrai* soit à *faux*. Ensuite on arrive dans un autre état où la première variable n'existera plus et sera remplacé par son évaluation.
On réitère ensuite ce processus pour les variables qui suivent.
La flèche en pointillée représente le chemin à emprunter si la variable correspondant au nœud a 0 pour valeur de vérité et la flèche pleine correspond à la valeur de vérité 1.
Ici, on voit bien que si x_0 est faux, alors l'implication est vrai, quelque soit la valeur de vérité de

x_1.
Si x_0 est vrai, alors il faut regarder la valeur de vérité de x_1 : si x_1 est vrai, alors l'implication est vraie et si x_1 est faux alors l'implication est fausse.

Ici, on a aussi introduit l'idée que cet arbre est ordonné, qu'on notera *ADBO* (arbre de décision binaire ordonné).
Il faut donner un ordre aux variables pour savoir dans quel ordre on va attribuer une valeur de vérité aux variables pour former l'arbre.
On le présentera plus en détail plus tard, mais il s'avère que l'arbre dépend de l'ordre choisi : on peut obtenir un arbre plus petit avec un certain ordre et un autre beaucoup plus grand avec un autre ordre.

Enfin, introduisons la notion d'unicité de cet arbre. La construction de l'arbre est unique en fonction de l'ordre choisi, c'est à dire qu'en ayant fixé un ordre, toutes les expressions équivalentes posséderont le même ADB. On le notera *ADBOC* (arbre de décision binaire ordonné canonique).

Organisation

On va ici vous présenter les résultats de notre recherche sur le sujet.
Dans un premier temps, on va vous présenter les propriétés d'un ADBOC et la raison pour laquelle on peut générer pour toutes expressions propositionnelles un ADBOC.
Ensuite, nous présenterons l'algorithme que nous avons implémenté en Ocaml pour pouvoir générer ces ADBOCs. Nous discuterons aussi de la complexité de cet algorithme.
Enfin, nous présenterons les intérêts des ADBOCs ainsi que des réflexions qu'on peut avoir sur leurs utilisations.

Chapitre 1

Représentation sous ROBDD

1.1 Opérateur If-then-else et INF

Soit $x \to y_0, y_1$ l'opérateur *if-then-else* défini par

$$x \to y_0, y_1 = (x \land y_0) \lor (\neg x \land y_1)$$

Ainsi, $t \to t_0, t_1$ est *vrai* si t et t_0 sont *vrais* ou si t est *faux* et t_1 est *vrai*, et est faux sinon.
Tout les operateurs peuvent assez facilement être traduit en utilisant uniquement l'opérateur *if-then-else*, 1 et 0 (Pour *vrai* et *faux*) :

- $\neg A = A \to 0, 1$
- $(A \land B) = A \to B, 0$
- $(A \lor B) = A \to 1, B$
- $(A \Rightarrow B) = A \to B, 1$
- $(A \Leftrightarrow B) = A \to B, (B \to 0, 1)$

où A et B sont des variables propositionnelles.

On peut aussi représenter une variable propositionnelle A par $A \to 1, 0$
On introduit donc la *Forme Normale If-then-else* (INF : *If-then-else Normal Form*) comme étant une expression Booléen construite entièrement avec l'opérateur *if-then-else*, 1 et 0 appliqué aux variables propositionnelles de la formule qu'elle traduit.

En notant $t[0/x]$ l'expression obtenu en remplaçant la variable x par 0 dans t, on remarque assez facilement que :

$$t = x \to t[1/x], t[0/x]$$

pour chaque variable de t.
On appelle se résultat l'*Expansion de Shannon* de t sur x.
Grâce à ce résultat, on peut générer pour chaque expression booléen une INF en réitérant l'expansion de Shannon sur chaque variable de l'expression.
On a vu précédement que toute expression Booléen est équivalentes à une expression en INF, on a maintenant un moyen de la trouver.

Exemple : Soit $t = (x_1 \Leftrightarrow y_1) \land (x_2 \Leftrightarrow y_2)$.
On peut donc appliquer l'expansion de Shannon :

$$t = x_1 \to t_1, t_0)$$
$$t_0 = y_1 \to 0, t_{00})$$
$$t_1 = y_1 \to t_{11}, 0)$$
$$t_{00} = x_2 \to t_{001}, t_{000})$$
$$t_{11} = x_2 \to t_{111}, t_{110})$$
$$t_{000} = y_2 \to 0, 1)$$
$$t_{001} = y_2 \to 1, 0)$$
$$t_{110} = y_2 \to 0, 1)$$
$$t_{111} = y_2 \to 1, 1)$$

1.2 Les ROBDD et leurs propriétés

On introduit les *Diagrammes de Décision Binaire* (BDD) définis avec :

- Les noeuds représentent les expressions booléens t la variable x dans l'expansion de Shannon.
- Les arêtes en pointillé représentent la valuation de x en 0 dans l'expansion de Shannon.
- Les arêtes pleines représentent la valuation de x en 1 dans l'expansion de Shannon.

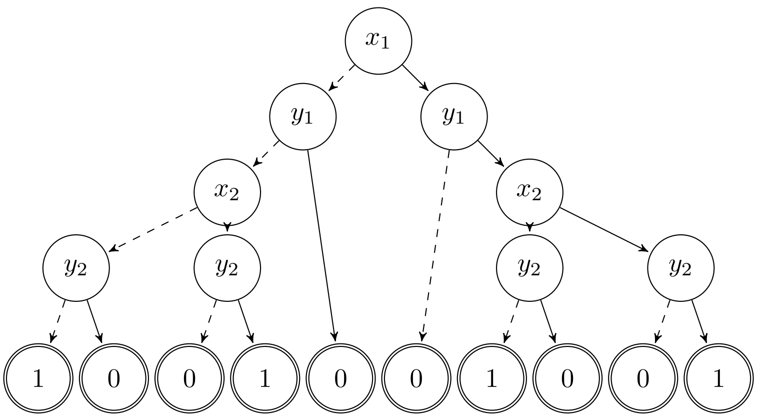
Ainsi, chaque noeud non-terminaux représente une extansion de Shanon $t = x \to t_1, t_0$ et est représenté par un triplet (var, low, high) où :

- var est x
- low est le noeud représentant t_0
- high est le noeud représentant t_1

Les noeuds terminaux sont 0 et 1

On représente donc une expression par le noeud racine.

Avec l'exemple précédent, on obtient le BDD suivant :



Si les variables apparaîssent dans un même ordre tout au long de la construction du BDD, on dit que le diagramme est *ordonné* (OBDD).

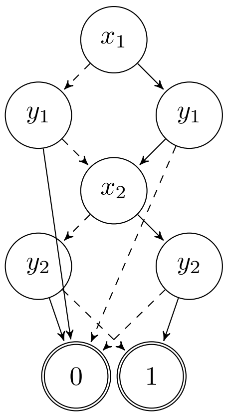
On remarque certains noeuds sont dupliqué.
Ainsi, dans l'exemple lors de la construction, on réduit par :

$$t_{000} = t_{110}$$
$$t_{001} = t_{111}$$
$$t_{00} = t_{11}$$

Et on obtient :

$$t = x_1 \to t_1, t_0)$$
$$t_0 = y_1 \to 0, t_{00})$$
$$t_1 = y_1 \to t_{00}, 0)$$
$$t_{00} = x_2 \to t_{001}, t_{000})$$
$$t_{000} = y_2 \to 0, 1)$$
$$t_{001} = y_2 \to 1, 0)$$

Le diagramme devient donc :



Si tout noeud redondant ont été supprimé et tout les noeuds identique ont été réduit, on dit que le diagramme est *réduit* (ROBDD).

Une ROBDD admet donc plusieurs propriété :

- Contient un ou deux noeud terminaux (0 et/ou 1)
- Contient un ensemble de noeuds u non-terminaux de degré 2 ($var(u)$ la variable x du noeud, $low(u)$ le noeud de $t[0/x]$, $high(u)$ le noeud de $t[1/x]$)
- Les var(u) des noeuds respectent un ordre en fonction de leur place dans le diagramme ($x_1 < x_2 < ... < x_n$)
- Soit 2 noeuds u et v, si $var(u) = var(v)$, $low(u) = low(v)$ et $high(u) = high(v)$ alors $u = v$
- Soit u un noeud alors $low(u) \neq high(u)$

1.3 Une unique ROBDD pour chaque fonction binaire

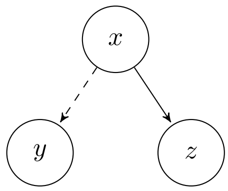
On a vu précédement que l'on peut associer à toutes formules propositionnelles un ROBDD.

Mais la valuation d'une formule formule n'est rien d'autre qu'une fonction binaire $f : \mathbb{B}^n \to \mathbb{B}$.

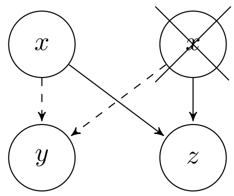
Lemme (Canonicité) : Pour toute fonction $f : \mathbb{B}^n \to \mathbb{B}$ il existe exactement une ROBDD u avec comme ordre $x_1 < x_2 < ... < x_n$ notée $f^u = f(x_1, x_2, ..., x_n)$
Pour simplifier la démonstration, on pose :

$$f_b(x_2, ..., x_n) = f(b, x_2, ..., x_n) \text{ où } b \in \mathbb{B}$$

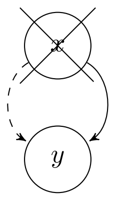
Ordonnée : x < y, x < z



Unique



Non-redondant



Démonstration Par induction sur n :

Si $n = 0$
Alors il n'existe que 2 fonctions binaires sans argument : $f = 0$ et $f = 1$.
Et il n'y a que 2 ROBDD qui n'utilisent pas de variable : $0(= false)$ et $1(= true)$
Or 0 est un ROBDD pour $f = 0$ mais pas pour $f = 1$
et 1 est un ROBDD pour $f = 1$ mais pas pour $f = 0$
Donc il y a une unique ROBDD pour chaque $f : \mathbb{B}^0 \to \mathbb{B}$

Si $n \neq 0$
Supposons que le Lemme est vrai pour tout $k <= n$, montrons que le Lemme est aussi vrai pour $n + 1$.

Soit $f : \mathbb{B}^{n+1} \to \mathbb{B}$. Alors $f(x_1, ..., x_{n+1}) = x_1 \to f_1(x_2, ..., x_{n+1}), f_0(x_2, ..., x_{n+1})$
Par induction : Il y existe des uniques u_0 et u_1 tels que $f^{u_0} = f_0$ et $f^{u_1} = f_1$.

Il y a deux cas à considérer :

Si $u_1 = u_0$ alors $f_0 = f^{u_0} = f^{u_1} = f_1$ donc $u_0(= u_1)$ est aussi un ROBDD pour f. Qui est aussi unique dû à l'ordre.
En effet, si x_1 se trouvait dans le ROBDD enracinée dans u alors $var(u) = x_1$.
Cependant $f = f^u$ alors $f_0 = f^u[0/x_1] = f^{low(u)}$ et $f_1 = f^u[1/x_1] = f^{high(u)}$ mais $f_0 = f_1$ alors par induction $low(u) = high(u)$ ce qui viol une des conditions des ROBDD.

Si $u_1 \neq u_0$ alors par induction $f_{u_0} \neq f_{u_1}$.
Posons u tel que $var(u) = x_1$, $low(u) = u_0$ et $high(u) = u_1$ ainsi $f^u = x_1 \to f^{u_1}, f^{u_0}$
Par induction $f^{u_1} = f_1$ et $f^{u_0} = f_0$ Or $f(x_1, ..., x_{n+1}) = x_1 \to f_1(x_2, ..., x_{n+1}), f_0(x_2, ..., x_{n+1})$ donc $f = f^u$
Supposons v un noeuds tel que $f = f^v$. Clairement f^v dépend de x_1. Donc $f^v[0/x_1] \neq f^v[1/x_1]$ (Sinon cela violerai le ROBDD).
Grâce à l'ordre $var(u) = x_1 = var(v)$.
De plus $f^u = f$ donc $f^{low(v) = f_0 = f^{u_0}}$ et $f^{high(v) = f_1 = f^{u_1}}$ par induction $low(v) = u_0 = low(u)$ et $high(v) = u_1 = high(u)$.
Donc par unicité des noeuds $u = v$

Chapitre 2

Construction d'une ROBDD

On a implémenté tout un algorithme (en Ocaml) prenant en argument une expression propositionnelle et permettant d'obtenir l'ADBOC de celle-ci.
On va vous présenter les fonctions principales.

2.1 présentation de l'algorithme

Il faut déjà comprendre que l'algorithme implémenté est dans un paradigme impératif et qu'on va utiliser une structure correspondante à une Hashmap : l'accès à un élément est linéaire amorti et l'ajout est en temps linéaire amorti, c'est à dire qu'on peut faire ces opérations en temps constant dans un moyenne des temps mais en temps linéaire dans le pire des cas.
Notons avant de comprendre l'algorithme en détail que nous avons une structure, noté S, qui nous permet d'associer un nœud de l'arbre à ses deux nœuds enfants : on notera *low* pour le nœud suivant la valuation à *faux* et *high* le nœud correspondant à la valuation *vrai* de la variable du nœud parent.
Enfin, on indicera les variables par leur rang dans l'ordre : x_n est la n-ième variable, $n \in \mathbb{N}$ et $n \geq 0$.

L'algorithme est basé sur un principe récursif, on refait la même étape jusqu'à atteindre la condition de fin :

1. On détermine l'ordre d'évaluation des variables en fonction de l'ordre donné, pour obtenir un arbre ordonné.
2. On évalue l'expression propositionnelle donné en argument pour voir si la valuation n'est pas déjà déterminable. Par exemple, si on a $0 \wedge x$ avec x une variable propositionnelle, alors on sait déjà que quelque soit la valuation de x, l'expression est fausse.
3. Si la valuation est déterminé, on renvoie la nœud de l'arbre correspondant à la valeur de vérité correspondante (*vrai* ou *faux*).
4. Sinon, on crée le nœud *low* et le nœud *high* :
   - *low* correspond au nœud créé quand on repasse par l'étape 2 où on a remplacé la variable évaluée par 0 et où on évalue par la variable suivante dans l'ordre de l'étape 1.
   - *high* correspond au nœud créé quand on repasse par l'étape 2 où on a remplacé la variable évaluée par 1 et où on évalue par la variable suivante dans l'ordre de l'étape 1.
5. Ensuite, on regarde si le nœud avec comme suivant *low* et *high* n'a pas déjà été créé, auquel cas on le crée, sinon on récupère le nœud déjà créé. Cela permet d'être sur qu'on obtient bien un arbre canonique.
6. Enfin, on renvoie le nœud créé. A la fin de l'algorithme, le dernier nœud renvoyé correspondra à la racine de l'arbre.

2.2 Création d'un nœud

On vous présente en simplifier le code de la fonction qui s'occupe de créer un nouveau nœud en cas de besoin. Ensuite nous commenterons les différentes lignes.

```
let mk (i : indice de la variable) (low : noeud) (high : noeud) : node =
    if low = high then low      (* #1 *)
    else
    if member i low high then   (* #2 *)
      lookup i low high         (* #3 *)
    else
      create_node i low high    (* #4 *)
```

On regarde d'abord si on obtient le même résultat quelque soit la valuation de la i-ème variable (#1). Si c'est le cas, on renvoie le nœud correspondant à une des deux valuations (on garde la canonicité).
Si ce n'est pas le cas, on regarde si on n'a pas déjà créé un nœud ayant le même *low* et le même *high* (#2), auquel cas on renvoie ce nœud (#3).
Enfin, si le nœud n'existe pas, alors on le crée (#4).
Dans le meilleur des cas, la création se fait en temps constant et dans le pire des cas, elle se fait en temps linéaire (à cause de la Hashmap).

2.3 Construction de l'arbre

On va ensuite pouvoir exposer l'algorithme correspondant à la construction même de l'arbre.

```
let build (f : exp propositionnelle) : noeud =
    let tab = Array.init (!n) (function a -> None) in
    let rec aux (f : exp propositionnelle) (i : indice de la variable) : node =
      match getBool f tab with                         (* #1 *)
      | (None, nf) -> begin                           (* #2 *)
          if i >= (!n) then failwith "exception de getBool"   (* #3 *)
          else begin
            let v0 = tab.(i) <- (Some false); aux nf (i+1) in (* #4 *)
            let v1 = tab.(i) <- (Some true); aux nf (i+1) in  (* #5 *)
            tab.(i) <- None;
            mk i v0 v1                               (* #6 *)
          end
        end
      | (Some b, _) ->                               (* #7 *)
        if b then 1
        else 0
    in
    aux f 0
```

L'initialisation du tableau *tab* correspond à l'évaluation partielle de chacune des variables.
Ensuite, on fait une fonction récursive qui se déroule en 7 étapes :

1. On fait l'évaluation partielle de l'expression propositionnelle pour voir si on ne peut pas déterminer sa valeur sans avoir donné une valuation à chaque variable (#1).

2. Si ce n'est pas le cas (#2), on vient effectuer le corps de l'algorithme en construisant de nouveaux nœuds. Ici, nous avons aussi effectué une optimisation de l'algorithme en faisant en sorte que, lors de l'évaluation partielle, on factorise l'expression propositionnelle dans un même temps (correspond à la variable *nf* dans #2).

3. Si on se retrouve dans la situation où on a déjà donné une valeur de vérité à toutes les variables et qu'on est dans une situation indéterminée, alors on a une erreur d'algorithme (#3).

4. Si on ne rencontre pas cette situation, on va récupérer le nœud correspondant à la valuation. Et donc on vient récupérer le nœud pour l'évaluation de la variable à 0 (#4), donc ce qui correspond au *low*.

5. Ensuite, on récupère pour la valuation à 1 (#5).

6. Enfin, on récupère le nœud correspondant (création ou récupération d'un nœud déjà existant : dépend de la fonction *mk*), et on renvoie ce nœud.

7. Si, par contre, on se retrouve à avoir une valuation déjà déterminée, alors on se retrouve dans la situation #7, et la on renvoie le nœud correspond à *vrai* ou à *faux*.

Notre algorithme suit la logique où on donne une valuation 1 ou 0 à chaque variable, et donc on aurait une complexité de 2^n à chaque fois. Cependant, il fait cela de façon intelligente, et on ne se retrouve pas deux fois dans une même situation. On peut dire que c'est une sorte d'algorithme dynamique (à l'aide des hashmap).

2.4 Sat-solveur

On a ensuite défini des fonctions de parcours de cette arbre et qui nous permettent d'obtenir des informations intéressantes avec notre arbre déjà construit.

2.4.1 SatCount

La première fonction est une fonction en temps linéaire du nombre de nœud de notre arbre et qui permet de déterminer combien est ce qu'il y a de valeurs de vérité qui satisfassent une expression propositionnelle .

```
let satCount (u : noeud) : int =
    let node_of_var = ... in                          (* #1 *)
    let rec aux (u : noeud) : (distance au noeud true, nombre de possibilites)  =
      if u = 0 then (0,0)
      else if u = 1 then (0,1)                          (* #2 *)
      else
        let ind = node_of_var.(u) in
        let som =
          let (i,res) = aux (low u) in two_power (ind - i - 1) * res (* #3 *)
                        +
          let (i, res) = aux (high u) in two_power (ind - i - 1) * res
        in (ind, som)
    in
  let size = Array.length !var_tab in
    if u = 0 then 0                               (* #4 *)
    else if u = 1 then two_power size                 (* #5 *)
    else let (i,res) = aux u in two_power (size - i) * res   (* #6 *)
```

Pour pouvoir compter le nombre de valeurs de vérité possible, on a besoin de savoir à quelle variable correspond chaque nœud (plus précisément, on renvoie l'indice de la variable et qui commence à 1). C'est dans ce but là qu'on initialise le tableau *node_of_var* (#1).
Ensuite, on va faire une fonction récursive locale à la fonction *aux* : elle prend un nœud de l'arbre en argument et renvoie un couple contenant la distance du nœud à *true* et le nombre de valuation possible depuis ce nœud là.
Si on tombe sur le nœud *vrai*, on renvoie le couple $(0, 1)$, car la distance à lui-même est à 0 et on a une solution possible, et $(0, 0)$ pour *faux* (#2).
Ensuite, si on n'est pas sur une constante, on calcule pour chaque valuation de la variable combien on a de solutions possibles (#3) :

- On récupère le couple de *low* et de *high*.
- Si la variable suivante (*low* ou *high*) possède un indice dont la différence est supérieure à 1, alors lors de la construction, on s'est retrouvé à faire une simplification du nœud parce qu'il amenait à la même valeur de vérité. Donc le nombre de solutions correspond à $2^{difference\ des\ distances\ -\ 1}$.

Lorsqu'on sort de la boucle, il ne nous reste plus qu'à multiplier le nombre de solution de la racine par $2^{distance\ avec\ le\ nombre\ de\ variables}$.
On obtient ainsi le nombre de solutions possibles.

2.4.2 AnySat

Lorsque la construction de l'arbre a été effectué, on peut déterminer en tant constant si une expression propositionnelle est une antilogie (s'il n'y a qu'un seul nœud et qu'il correspond à *faux*). De plus, on peut trouver, en temps linéaire du nombre de variables, une valuation satisfaisant notre expression propositionnelle .

```
let rec anySat (u : noeud) : (variable * bool) list =
    let rec aux u =
      if u = 0 then raise (No_SAT "Solution␣introuvable")
      else if u = 1 then []
      else if low u = 0 then (!var_tab.(var u), true) :: (aux (high u))
      else (!var_tab.(var u), false) :: (aux (low u))
    in
    fill_var 0 (aux u)
```

Il suffit de parcourir notre nœud racine et d'arriver jusqu'au nœud *vrai*. Pour cela, on a choisi de passer par *low*, mais on aurait très bien pu passé par *high*, car si *low* et *high* existe et sont différents de *faux*, alors il y a un chemin permettant d'aller à *vrai*.
On est en temps linéaire du nombre de variable car à chaque itération, on descend d'un cran vers *vrai*, et donc on donne une valuation à chaque variable.
La dernière opération, appelé *fill_var*, permet de remplir la solution en attribuant une valeur de vérité à toutes les variables dont un nœud correspondant n'a pas été visité. Cela permet d'obtenir une solution attribuant une valeur de vérité à chaque variable.

2.4.3 AllSat

De la même façon, on peut aussi obtenir l'ensemble des solutions possibles en temps linéaire selon le nombre de nœud de l'arbre :

```
let allSat (u : node) : (variable * bool) list list =
    let rec aux (u : noeud) l r =
      if u = 0 then r
      else if u = 1 then l::r
      else aux (high u) ((!var_tab.(var u), true)::l) (aux (low u) ((!var_tab.(
var u), false)::l) r)
    in
    List.rev (List.rev_map (fun a -> fill_var 0 a) (aux u [] []))
```

Ici, à chaque nœud, on a comme argument l'ensemble des solutions possibles pour les nœuds parents. Ensuite, on teste si on arrive à *vrai* en évaluant la variable, et si c'est possible, on ajoute la valuation à l'ensemble des valuations possibles.
On se retrouve ainsi à obtenir l'ensemble des solutions possibles en temps linéaire selon le nombre de nœud.

On vous a beaucoup parlé de l'algorithme, mais maintenant passons à la pratique.

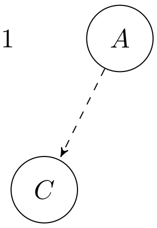
2.5 Exemple d'exécution de l'algorithme

Prenons l'expression propositionnelle suivante :

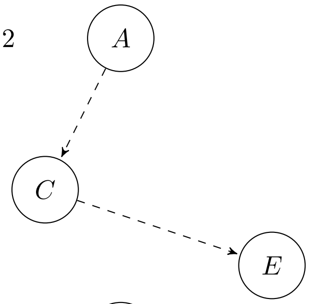
$$A = ((x_1 \wedge x_3) \Leftrightarrow (x_3 \vee x_4)) \Rightarrow ((\neg x_2) \wedge x_1)$$

On va faire les différentes étapes de l'algorithme.
Première chose, on va définir l'ordre suivante : $x_1 < x_2 < x_3 < x_4$.

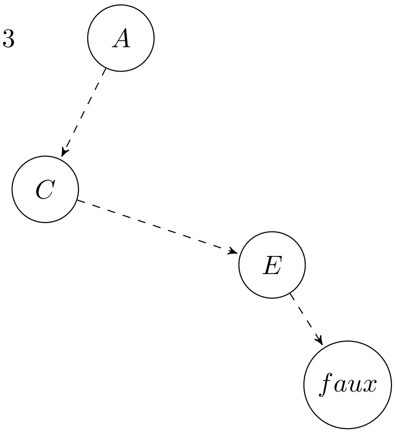
Étape 1



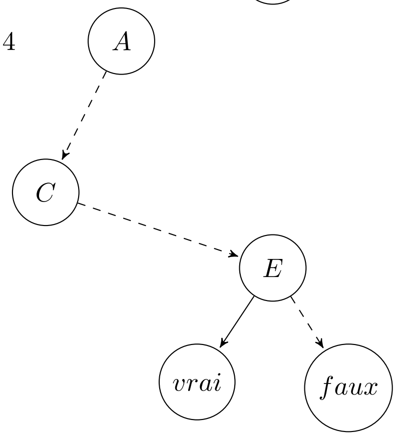
Étape 2



Étape 3

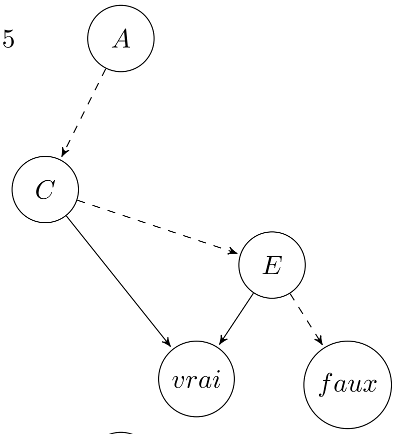


Étape 4

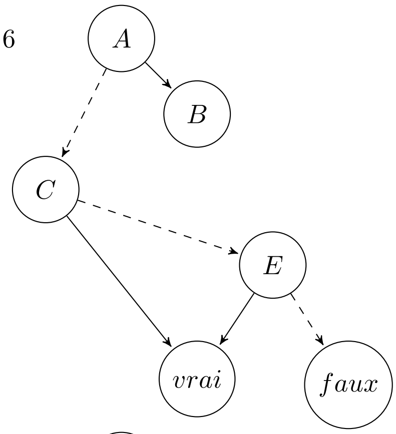


Première étape, on prend notre expression propositionnelle , et on cherche *low* et *high* donc on donne une valeur de vérité à la variable x_1.
Tout d'abord, on associe 0 à x_1, et on arrive en $C = x_3 \vee x_4$.
Ensuite, C n'est pas un état terminal, donc on doit déterminer son *low* et *high* pour la variable x_3 car ni la variable x_1, ni la variable x_2 n'existe.
Donc à l'étape 2, on cherche *low* (on associe 0 à x_3) et on arrive dans le nœud $E = x_4$.
On n'est toujours pas dans un état terminal, donc on cherche le *low* et le *high* de E.
On commence d'abord par *low*, donc on associe 0 à x_4 et là on se retrouve dans l'état terminal *faux* (étape 3).
On peut ensuite chercher *high*. A l'étape 4, on associe 1 à x_4 et on arrive sur l'état terminal *true*.

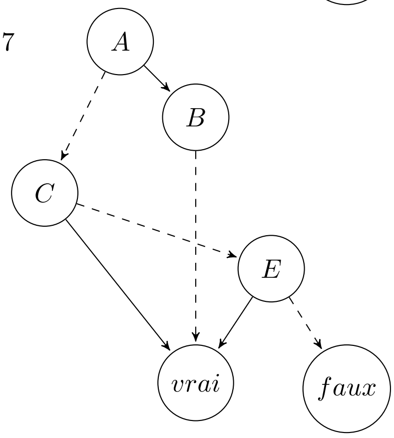
Étape 5



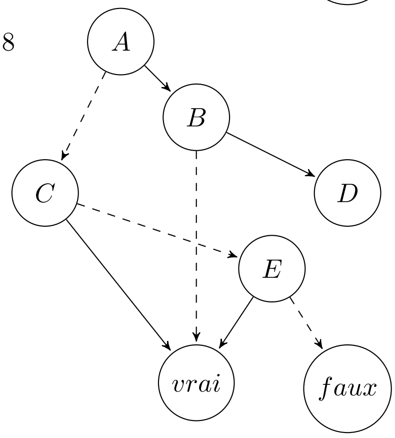
Étape 6



Étape 7

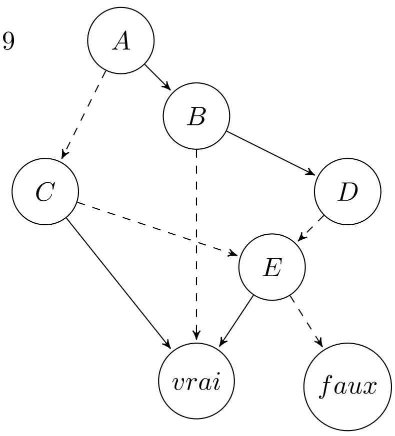


Étape 8

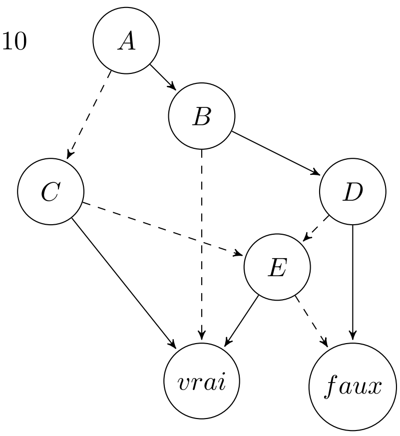


Ensuite, on a terminé de déterminer *low* et *high* de E, on a donc déterminé le *low* de C.
On peut donc à l'étape 5 déterminer *high*, et on arrive sur l'état terminal *vrai*.
Ensuite, on remonte à A où on va ensuite pouvoir déterminer son *high*. On trouve à l'étape 6 le nœud $B = (x_3 \Leftrightarrow (x_3 \vee x_4)) \Rightarrow (\neg x_2)$.
À l'étape 7, on détermine que *low* de B est l'état terminal *vrai*.
À l'étape 8, on cherche *high* et on arrive à l'état $D = \neg(x_3 \Leftrightarrow (x_3 \vee x_4))$.

Étape 9



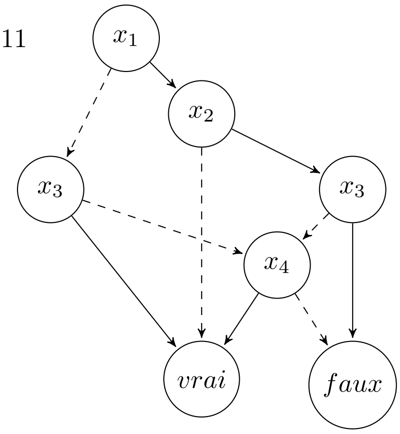
Étape 10



On détermine ensuite à l'étape 9 le *low* de D, et on retrouve un nœud ayant le même *low* et le même *high* que E, donc le *low* de D est E.
Ensuite, à l'étape 10, on détermine le *high* de D qui est *faux*.
Enfin, on a trouvé le *low* et le *high* du nœud racine (ici A), donc on a terminé l'algorithme.

En remplaçant chaque nom de nœud par la variable qui a été évalué pour le *low* et *high*, on obtient l'arbre de décision binaire ordonné canonique pour l'expression propositionnelle A suivant :

Étape 11



2.6 Etude des complexités

Maintenant, intéressons nous aux complexités des différentes fonctions. On peut se demander à juste titre si cet algorithme ne pourrait pas être intéressant.
Voici un tableau récapitulatif des complexités :

Fonction	Compléxité	Note
MK(i, low, high)	$O(1)$	En considérant que la hashmap est en temps constant
BUILD(exp)	$O(2^n)$	pire cas : calculer toutes les valeurs de vérité
SATCOUNT(racine)	$O(\|arbre\|)$	Linéaire selon le nombre de nœud de l'arbre
ANYSAT(racine)	$O(n)$	Linéaire selon le nombre de variables
ALLSAT(racine)	$O(max(2^{\|arbre\|}, 2^n))$	Nombre de solutions possibles

On remarque que la fonction ANYSAT est en tant constant du nombre de variable, mais il faut prendre en compte qu'on a construit au préalable l'arbre de décision binaire ordonné canonique correspondant.
Le plus gros de la complexité de l'algorithme se fait donc au niveau de la fonction BUILD.
On va voir dans la partie d'après nos réflexions là-dessus.

Chapitre 3

Intérêt et optimisation

3.1 Sat-solveur mais mauvais

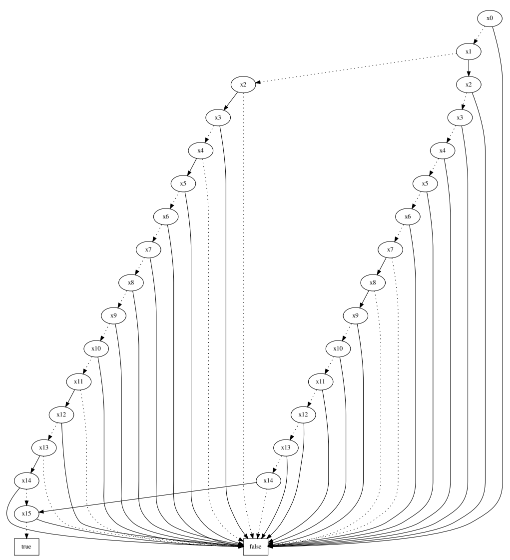
La première utilisation des ROBDD est en tant que Sat-solveur. En effet, le fait de savoir si une formule admet une solution se fait en temps constant, dès que l'on a son ROBDD, car si la formule n'admet pas de solution alors elle est une antologie et par unicité des ROBDD, le ROBDD qui lui est associé est 0.
Et donc il suffit de comparer le premier noeud du ROBDD à 0 pour savoir si il existe une solution.

Suite à ceci, il est possible de trouver l'ensemble de solution à partir du ROBDD en le parcourant et en retenant tout le chemin qui mènent au noeud 1.

On utilise ici les fonctions ANYSAT et ALLSAT (Et SATCOUNT si l'on n'a besoin uniquement du nombre de solution).

De là il est possible de résoudre des problèmes logique comme le Problème des N Reines, le Sudoku ou encore des BenchMarks.

Exemple : Problème des 5 Reines



Avec un total de 2 solutions.

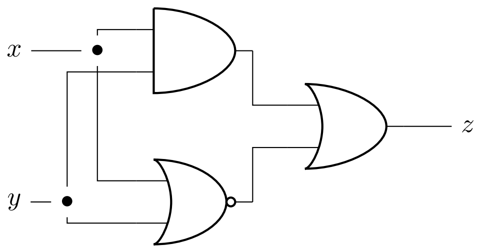
La raison pour laquelle ce Sat-solveur n'est pas utilisé est que ,bien que la compléxité de l'algorithme ANYSAT soit linéaire, la fonction BUILD est de compléxité exponentielle.

3.2 Correction et Equivalence de la combinatoire de circuit

Lorsque l'on a le ROBDD d'une formule, il est assez facile de voir qu'elles sont les distributions de valeur de vérité qui la satisfaite.
Donc une des principales utilisations des ROBDD et pour la correction ou équivalence de circuit (logique).

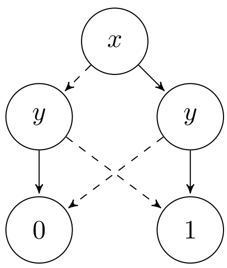
En effet, il est assez évident de voir qu'un circuit logique peut être représenté par une formule. Les entrées sont les variables et les portes les opérations logiques (souvent unaire comme la porte 'N', ou binaire comme les portes 'OR', 'AND', 'NOR', 'XOR' ...).

Comme par exemple : un Circuit d'équivalence



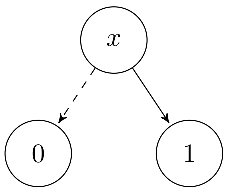
De formule : $((x \wedge y) \vee (\neg(x \vee y)))$

ROBDD d'ordre : $x < y$



Pour la traduction des circuits en ROBDD sans passer par la formule revient à créer des ROBDDs pour chaque variables :

Variable : x



Puis d'appliquer la fonction APPLY en fonction des portes logiques utiles.

```
let apply (op : node->node->node) (u1 : node) (u2 : node) : node =
    let g = Hashtbl.create (map_size*map_size) in
    let rec aux (u1 : node) (u2 : node) : node =
      if Hashtbl.mem g (u1, u2) then Hashtbl.find g (u1, u2)
      else
        let u =
          if unite u1 && unite u2 then op u1 u2
          else if var u1 = var u2 then                (* c1 *)
            mk (var u1)
              (aux (low u1) (low u2))
              (aux (high u1) (high u2))
          else if var u1 < var u2 then                (* c2 *)
            mk (var u1)
              (aux (low u1) u2)
              (aux (high u1) u2)
          else                                        (* c3 *)
            mk (var u2)
              (aux u1 (low u2))
              (aux u1 (high u2))
        in
        Hashtbl.add g (u1, u2) u;
        u
    in
    aux u1 u2
```

APPLY est un fonction qui nous permet de réunir deux ROBDD par un opérateur *op* (dans notre cas les différentes portes). Son principe est récursif : On crée un nouveau noeud de *var* prioritaire par rapport à l'ordre des *var* des deux noeuds à réunir. On réunit les noeuds *low* des deux noeuds en fonction de leur priorité dans l'ordre et fait de même pour les *high*. Ces deux nouveaux noeuds sont respectivement le *low* et le *high* du nouveau noeud. Si les noeuds à réunir sont des 1 et/ou des 0, on applique l'opérateur.

APPLY ne viole pas les règles des ROBDD car elle utilise mk qui gère l'unicité des noeuds et la non-redondance et l'ordre est gèré directement dans l'algorithme avec les conditions $c1, c2, c3$.

Utiliser APPLY nous permet alors de juste parcourir le circuit et construire le ROBDD en même temps.

Ainsi, vérifier qu'un circuit logique donne les bons résultats devient très simple avec son ROBDD et montrer que deux circuits soient équivalents revient à remarquer que les deux ROBDD soient exactement les même (par unicité des ROBDD).

3.3 Théorie sur les ordres

Conclusion

L'utilisation des arbres de décision binaires ordonnés canoniques est une nouvelle approche des expression propositionnelle qui est très intéressante. Ce projet a été une première approche à la recherche en mathématique et en informatique sur les expressions propositionnelles .

De plus, on a pu voir la complexité de trouver une méthode de résolution *SAT* même si cet algorithme n'est pas adapté pour être un *SAT-solveur*.

On a du faire un travail de recherche sur les différents ordres possibles pour une expression propositionnelle pour voir s'il n'existait pas un ordre qui permettrait de construire un arbre en temps polynomiale.

Si un tel ordre existait, on pourrait construire l'arbre en temps polynomiale et ensuite trouver une solution en temps linéaire : ce qui pourrait être très intéressant pour un Sat-solveur.

On a vu que pour une même expression propositionnelle , on peut avoir deux ordres différents nous donnant un nombre de nœud totalement différent.

Ceci est une toute nouvelle approche possible qu'on a pu avoir.

Cependant, l'utilisation d'un arbre de décision binaire ordonné canonique est surtout pour déterminer si deux expressions propositionnelles sont équivalentes : comme on construit un arbre canonique, on obtient, pour un même ordre, le même arbre si les deuxexpressions propositionnelles sont équivalentes.

Ensuite, un arbre de décision binaire ordonné canonique permet aussi de représenter une expression propositionnelle sous une forme plus compréhensible par un humain.

Nous tenions à remercier Monsieur Sedki Boughattas qui nous a encadré pour ce projet. Nous avons été très intéressé par ce sujet.

Bibliographie

Article de Henrik Reif Andersen : *An Introduction to Binary Decision Diagrams*