Travaux Pratiques n° 2
UNE BIBLIOTHÈQUE DE CONSTRUCTION ET MANIPULATION DE DIAGRAMMES DE DÉCISION BINAIRE

E. Encrenaz-Tiphene, M. Pouzet
Programmation, Licence 2000-2001
Université Paris VI

Les diagrammes de décision binaire (Binary Decision Diagram ou BDD) ont été introduits par Bryant en 1986 pour représenter et surtout manipuler efficacement des expressions booléennes.

C'est une représentation de fonctions booléennes sous forme de graphes acycliques présentant un partage maximal de sous-graphes isomorphes, et -sous certaines contraintes- garantissant l'unicité de la représentation. Cette dernière propriété est un atout majeur par rapport aux autres représentations de fonctions booléennes (somme de *minterms*, produit de *maxterms*, tableau de Karhaugh) lorsque l'on cherche à déterminer l'égalité de deux fonctions booléennes.

De fait, cette structure de données s'est très largement répandue dans le domaine de conception d'outils de CAO pour la micro-électronique, notamment pour déterminer si deux circuits, ou portions de circuits sont équivalentes fonctionnellement, mais également dans d'autres domaines où ils permettent de manipuler des ensembles et graphes finis mais de très grande taille (par exemple la vérification automatique de programmes).

Le but du TP est de développer une petite bibliothèque de manipulation des diagrammes de décision binaire et de l'évaluer sur différents problèmes, illustrant, à moindre échelle, ses applications réelles.

1 PRÉSENTATION

1.1 Différentes représentations de fonctions booléennes

1.1.1 Quelques représentations classiques

La représentation la plus classique est la table de vérité. Chaque ligne de la table associe une configuration des variables de la fonction à sa valeur. Cette représentation est énumérative, et si la fonction est d'arité n, la table contiendra 2^n entrées.

Voici par exemple la table de vérité d'une fonction $f : \mathbb{B}^3 \to \mathbb{B}$

[h]

a	**b**	**c**	**f**
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

TAB. 1 – Représentation d'une fonction sous forme de table de vérité

Deux autres représentations également très employées sont la somme de *minterms* et le produit de *maxterms*.

Un *minterm* est une conjonction de variables ou de leur complément. Une somme de *minterms* est une disjonction de ces *minterms*. Ainsi, la fonction précédente peut être réécrite en :

$$f = \overline{a}.\overline{b}.c + \overline{a}.b.\overline{c} + \overline{a}.b.c + a.\overline{b}.c + a.b.\overline{c} + a.b.c$$

Remarque : le symbole $\overline{x}$ représente la complémentation de la variable x, le symbole + représente la disjonction (le OU logique), et le symbole . représente la conjonction (le ET logique).

Cette écriture est la transcription littérale des lignes de la table de vérité pour lesquelles la fonction f vaut 1. Mais cette autre écriture de f est tout aussi valable, et c'est également une somme de *minterms*. (Elle est obtenue à partir de la précédente en appliquant des factorisations et simplifications).

$$f = \overline{b}.c + b$$

Un *maxterm* est une disjonction de variables ou de leur complément. Un produit de *maxterms* est une conjonction de ces *maxterms*. Ainsi, la fonction f peut encore s'écrire :

$$f = (a + b + c).(\overline{a} + b + c)$$

C'est l'énumération du complémentaire des lignes de la table de vérité pour lesquelles la fonction vaut 0. Comme précédemment, cette autre écriture, toujours sous forme de produit de *maxterms*, est encore équivalente.

$$f = b + c$$

En fait, ces représentations souffrent d'un manque de représentant canonique de taille raisonnable. De plus, la manipulation (la complémentation, les opérateurs booléens,...) de telles représentations est souvent complexe.

1.1.2 Décomposition de Shannon

Les diagrammes de décision binaire sont basés sur la décomposition de Shannon de fonctions booléennes. Soit $f : \mathbb{B}^n \rightarrow \mathbb{B}$, sur un jeu de variables $x_1, ...x_n$ *cofacteurs* de f de la façon suivante :

$$f = x_i.cof(f, x_i, 1) + \overline{x_i}.cof(f, x_i, 0)$$

La fonction $f : \mathbb{B}^n \rightarrow \mathbb{B}$ est définie à partir de deux sous-fonctions de $\mathbb{B}^{n-1} \rightarrow \mathbb{B}$.

On peut alors construire un arbre pour toute fonction f, en appliquant récursivement la décomposition de Shannon tant que les sous-fonctions ne sont pas constantes, comme indiqué sur la figure 1. De plus, si la décomposition est effectuée dans le même ordre dans toutes les sous-fonctions, l'arbre est dit ordonné.

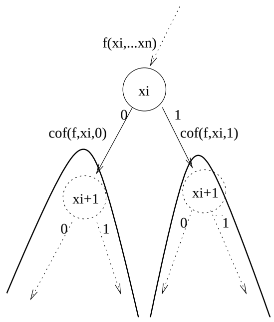

FIG. 1 – Représentation arborescente de la décomposition de Shannon

Cette représentation arborescente a une taille exponentielle avec l'arité de la fonction représentée, de fait, elle est inexploitable en l'état. Par contre, elle présente une très forte potentialité de réduction, ce qui la rend efficace dans la plupart des cas.

1.1.3 Réduction

Le partage des sous-graphes isomorphes permettent de réduire considérablement la taille de l'arborescence en général. Toutefois, cette réduction n'a lieu que si il existe effectivement des sous-graphes isomorphes. L'existence de ces sous-graphes dépend :

- de l'ordre d'occurrence des variables x_i le long des chemins menant de la racine de l'arborescence aux feuilles 0 et 1 ;
- de la fonction représentée. On a montré qu'il existait des fonctions pour lesquelles la représentation restait exponentielle quelquesoit l'ordre des variables (notamment les fonctions représentant des multiplications en binaire). En pratique, la plupart des fonctions couramment rencontrées ont une représentation quadratique.

La réduction d'un arbre de décision binaire ordonné suit l'algorithme :

1. regrouper toutes feuilles terminales 1 d'une part et toutes les feuilles 0 de l'autre.
2. appliquer les règles R1 et R2 suivantes tant qu'elles sont applicables.
   R1 : si un noeud a deux successeurs identiques (au sens de l'égalité physique), alors il est superflu.
   R2 : si deux noeuds ont les mêmes fils gauches et les mêmes fils droits (au sens de l'égalité physique), alors l'un des deux noeuds est superflu.

La figure 2 montre l'arbre issu de la décomposition de Shannon (à gauche), et le BDD réduit associé (à droite) de la fonction $f(x_1, x_2, x_3) = x_2.x_3 + x_1.\overline{x_2}.\overline{x_3} + \overline{x_1}.x_3$, pour l'ordre $x_1 < x_2 < x_3$.

Un résultat très important concernant ces diagrammes de décision binaire réduits et ordonnés est leur canonicité : Soient deux fonctions booléennes f et g de $\mathbb{B}^n - > \mathbb{B}$.

$$f = g \Leftrightarrow BDD(f) = BDD(g).$$

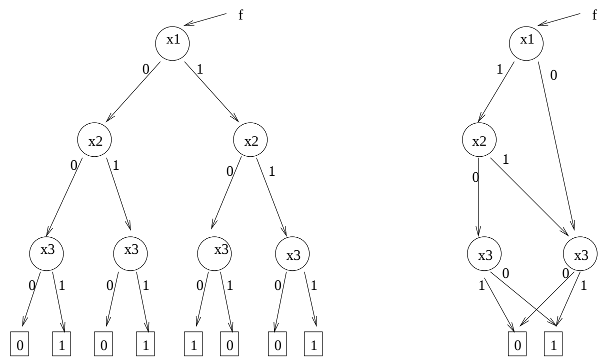

FIG. 2 – L'arbre de décision et le BDD de la fonction f

Ainsi, tester l'égalité de deux fonctions, revient à tester l'égalité structurelle des deux BDD les représentant (c'est-à-dire parcourir les arborescences des BDD, ce qui est de complexité proportionnelle à la taille des BDD). En fait, ceci peut encore être amélioré en représentant une seule fois les sous-graphes isomorphes *de tous les BDD*, et en les partageant. Dans ce dernier cas, seul un test sur l'égalité physique des noeuds racine des BDD des fonctions dont on souhaite connaître l'égalité est nécessaire (de complexité constante si les noeuds sont en mémoire).

1.1.4 Construction directe d'un BDD réduit

Plutôt que de construire un arbre complet puis de le réduire ensuite, il est beaucoup plus efficace de le construire directement réduit. Pour se faire, il convient de disposer d'une *table de partage* qui stocke l'identité de chaque noeud du BDD, ainsi que celle de chacun de ses successeurs. Avant de créer un nouveau noeud, on consulte cette table pour déterminer si le noeud à créer ne serait pas redondant avec un noeud existant. Si le noeud à créer existe déjà dans la table, alors c'est le noeud de la table qui est utilisé, et s'il n'existe pas dans la table, alors il est effectivement créée puis ajouté à la table de partage.

Le schéma de la figure 3 montre deux BDD f et g partageant certaines sous-structures. L'identité des noeuds est représentée ici par une lettre majuscule située en haut à gauche de chaque noeud.

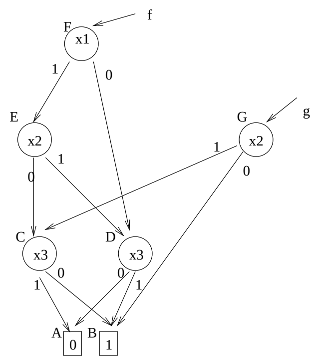

FIG. 3 – Plusieurs BDD partageant les mêmes noeuds

La table de partage correspondant à cette situation est représentée ci-après.

noeud	index	FilsZ	FilsU
A	-	-	-
B	-	-	-
C	3	B	A
D	3	A	B
E	2	C	D
F	1	D	E
G	2	B	C

TAB. 2 – Table de partage des BDD de la figure 3

1.2 Opérations sur les BDD

Les opérations booléennes classiques sont transposées en parcours et construction de BDD. Ces opérations sont linéaires ou quadratiques avec la taille (le nombre de noeuds) des BDD représentant les fonctions booléennes sur lesquelles portent les opérations.

1.2.1 Négation

Lorsque les BDD ne sont pas partagés, (les feuilles du BDD représentant une fonction f et celles du BDD représentant une autre fonction g sont différentes), la négation d'une fonction consiste uniquement à inverser les feuilles du BDD qui la représente : la feuille 0 devient la feuille 1 et vice-versa.

Lorsque plusieurs BDD partagent les mêmes feuilles, on ne peut plus inverser brutalement les feuilles, car alors on inverse non plus une seule fonction mais toutes celles partageant ces feuilles. Dans ce cas, il convient de propager la négation sur tous les arcs du BDD représentant la fonction, et, le cas échéant, de rétablir le partage avec les autres BDD.

En effet,

$$\begin{aligned} not f &= not\ (x\ .\ cof(f,x,1) + \overline{x}\ .\ cof(f,x,0) \\ &= x\ .\ not\ (cof(f,x,1) + \overline{x}\ .\ not\ (cof(f,x,0)) \end{aligned}$$

1.2.2 Opérations binaires

Les opérations binaires sont réalisées par un parcours attelé des BDD représentant les deux opérandes.

En effet,

$$\begin{aligned} f\ op\ g &= (x\ .\ cof(f,x,1) + \overline{x}\ .\ cof(f,x,0))\ op\ (x\ .\ cof(g,x,1) + \overline{x}\ .\ cof(g,x,0)) \\ &= x\ .\ (cof(f,x,1)\ op\ cof(g,x,1)) + \overline{x}\ .\ (cof(f,x,1)\ op\ cof(g,x,1)) \end{aligned}$$

1.2.3 Cofacteurs

Le calcul du cofacteur $cof(f,x,0)$ ou $cof(f,x,1)$ d'une fonction suit exactement la construction d'un BDD : il suffit de remplacer dans le BDD de la fonction f chaque noeud représentant la variable x par son sous-BDD représentant son cofacteur 0 ou 1 respectivement.

1.2.4 Quantification, substitution

Les algorithmes de quantification existentielle et universelle suivent les définitions :

$$\begin{aligned} exists(f,x) &= cof(f,x,1) + cof(f,x,0) \\ forall(f,x) &= cof(f,x,1).cof(f,x,0) \end{aligned}$$

De même, la substitution d'une expression g à une variable x dans une fonction f peut s'écrire de la façon suivante :

$$subst(f,x,g) = g.cof(f,x,1) + \overline{g}.cof(f,x,0)$$

Remarque : des algorithmes plus efficaces combinant en un seul parcours le calcul du cofacteur et la conjonction avec g ou $\overline{g}$ ont été proposés.

1.3 Facteurs d'amélioration

L'implantation des algorithmes décrits précédemment sur des BDD représentés par des arbres partagés permet d'effectuer de petits calculs sur des fonctions comportant quelques variables à quelques dizaines de variables. Au delà, les parcours répétés sur des gros graphes deviennent trop lents ; de plus, en cas de création et destruction dynamique de BDD, les BDD qui ne sont plus utilisés restent présents en mémoire et grossissent artificiellement celle-ci.

Examinons plusieurs pistes, implantées dans toute bonne bibliothèque, pour améliorer les temps d'exécution et l'espace mémoire nécessaires à la manipulation des fonctions.

Pour le TP, il vous sera demandé d'introduire la gestion du partage des noeuds des BDD ainsi qu'un cache d'opérations. Les deux dernières améliorations citées dans cette partie le sont pour mémoire.

1. **Gestion du partage** : tous les BDD manipulés sont stockés dans une unique table de partage implantée à l'aide d'une table de hachage (hash) ; c'est ce que l'on appelle communément *le manager*. Chaque entrée de cette table associe à un triplet `(identité du noeud, identité du filsZ, identité du FilsU)` l'identité du noeud[1]. De plus, cette table stocke les feuilles O et 1 (cf. §1.1.4 et figure 3 et table 2).

[1] en général, cette identité est l'adresse du noeud

2. **Réduction du nombre de parcours** : lors de l'application d'opérateurs binaires sur des BDD partagés, il est fréquent que l'on ait à réappliquer plusieurs fois le même opérateur aux mêmes noeuds. Pour éviter d'avoir à reparcourir les BDD et réeffectuer des calculs préalablement obtenus, on utilise un *cache d'opérations*. Lorsque l'on a à appliquer un opérateur sur deux BDD, on regarde dans le cache si ce calcul n'y est pas déjà mémorisé ; dans l'affirmative, le résultat du calcul est immédiatement disponible, et dans la négative on effectue le calcul puis on stocke la paire `(calcul à effectuer, résultat du calcul)`.
   En pratique, le cache d'opérations est réalisé par une table de hachage.
3. **Modification de l'ordre des variables** : la taille d'un BDD correspond au nombre de noeuds qui le constituent. Cette taille d'un BDD (réduit et ordonné) dépend de l'ordre d'occurrence des variables le long de chemins de la racine vers les feuilles (cf. Figure 4). Le problème est qu'en général, étant donné une fonction, on ne sait pas a priori quel est l'ordre (et même s'il en existe un) minimisant la taille du BDD. Pourtant, lors de calculs à un ordre fixé, les BDD deviennent souvent très vite énormes et il est nécessaire de pouvoir changer l'ordre d'occurrence des variables le long des chemins menant de la racine vers les feuilles, dans l'espoir de trouver un meilleur ordre, réduisant la taille de la représentation et de fait accélérant les calculs. En même temps, puisque tous les BDD sont partagés (ils sont stockés dans une unique table de partage, cf. §1.1.4, on ne peut changer l'ordre des variables d'un seul BDD. Ainsi, les critères de choix d'un bon ordre ne doivent pas uniquement réduire le BDD qui nous intéresse actuellement, mais ils doivent également éviter de faire croître la taille des autres BDD stockés dans la table de partage. Différents algorithmes ont été proposés, tous sont basés sur :
   (a) une permutation des variables du (ou des) BDD,
   (b) puis reconstruction suivant le nouvel ordre et
   (c) test pour déterminer si l'ordre testé est meilleur que le précédent.

   Les efforts portent essentiellement sur le choix de « bonnes » permutations, pour éviter d'avoir à énumérer les $n!$ solutions possibles.

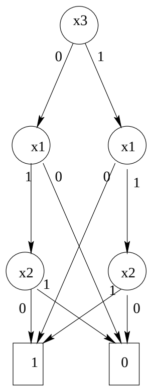

FIG. 4 – Le BDD de la figure 2 selon l'ordre $x_3 < x_1 < x_2$

4. **Suppression des noeuds non utilisés**. Toute bibliothèque dispose d'un « ramasse-miettes » ou « garbage collector » : souvent, les applications nécessitent la création et la suppression dynamiques de BDD. La création ne pose pas de difficulté (sinon qu'elle peut accroître la taille de la table de partage), par contre la suppression de BDD de la table partagée demande quelque attention. En effet, un BDD détruit peut très bien être reconstruit par la suite, aussi n'est-il pas forcément judicieux de le supprimer immédiatement. D'autre part, un BDD détruit peut toujours être référencé par un autre BDD (du fait du partage), ou par une ligne du cache d'opérations, que faire alors ? Le plus simple est :
   (a) de stocker le degré de partage d'un noeud dans une variable compteur attribuée à chaque noeud, c'est le *compteur de réferences*, d'incrémenter ce compteur chaque fois qu'un autre BDD fait référence à ce noeud, et de le décrémenter chaque fois qu'il y a une demande de destruction de ce noeud.
   (b) La destruction effective des noeuds est réalisée lors de l'exécution d'une commande explicite de destruction des noeuds, qui parcourt la table de partage en supprimant les noeuds dont le compteur de références est nul.

2 TRAVAIL A EFFECTUER

Le but de ce TP est de développer une petite bibliothèque de construction et de manipulation de BDD. Cette bibliothèque n'implantera évidemment pas toutes les subtilités permettant de la rendre compétitive avec les bonnes bibliothèques actuellement disponibles, mais elle permettra de traiter efficacement de petites expressions booléennes.

La construction de cette bibliothèque se fera progressivement :

Une première étape consistera à représenter les BDD par des arbres ordonnés, non réduits et non partagés, et à implanter les opérations booléennes sur cette structure. La seconde partie consistera à adjoindre un cache d'opérations dans ces fonctions, implanté à l'aide d'une table de hachage ; cette partie ne présente aucune difficulté mais vous familiarisera avec le module de gestion des fonctions de hachage de Camllight. La dernière partie consistera en l'implantation des BDD partagés (directement réduits), en utilisant une table de partage représentée par une table de hachage.

Lors de ces trois parties, vous pourrez tester votre programme sur de petites expressions booléennes. Enfin nous vous proposons d'utiliser votre bibliothèque pour résoudre un problème très combinatoire.

2.1 Descriptif de l'environnement de travail

L'environnement de travail qui vous est proposé vous permet de saisir au clavier une expression booléenne qui sera convertie en un arbre de syntaxe abstraite que vous pourrez par la suite transformer en un BDD. Il s'agit d'un ensemble de fichiers, dont voici la liste triée par ordre alphabétique.

- `Makefile` : le fichier contenant les directives de compilation.
- `algo.ml` : les fonctions de la bibliothèque de BDD. À COMPLETER.
- `bdd.ml` : les types des BDD et opérateurs booléens. À COMPLETER.
- `hash.ml` : les créations et manipulation de tables de hachage. À COMPLETER.
- `lexer.ml, lexer.ml, lexer.mll` : l'analyse lexicale des expressions saisies au clavier.
- `load.ml` : tous les modules à charger avant d'exécuter vos fonctions (de `algo.ml` ou `main.ml`) en session interactive.
- `main.ml` : les fonctions de construction d'une expression en syntaxe abstraite à partir d'une expression saisie au clavier, puis l'appel aux fonctions de construction du BDD (CETTE DERNIÈRE PARTIE EST À REALISER).
- `parser.ml, parser.mli, parser.mly` : le parser des expressions saisies au clavier.
- `reines.ml` : les fonctions permettant la construction de l'expression booléenne traduisant le problème des reines.
- `syntaxe_abstraite.ml` : la définition de la syntaxe abstraite représentant les expressions booléennes saisies au clavier.

La syntaxe des expressions booléennes que vous pouvez saisir au clavier est la suivante :

```
<expr> ::= <nom>
       |   <expr> + <expr>
       |   <expr> * <expr>
       |   ! <expr>
       |   <expr> = <expr>
       |   <expr> => <expr>
       |   ( <expr> )
       |   exists <nom>. <expr>
       |   forall <nom>. <expr>
       |   <expr> [<nom> \ <expr>]
```

Un <nom> désigne une suite ininterrompue de caractères représentant un nom de variable. Les symboles +, * et ! désignent respectivement la conjonction, la disjonction et la complémentation. = et => désignent respectivement l'égalité et l'implication. `exists` et `forall` désignent les quantifications existentielle et universelle. La dernière ligne indique la syntaxe de la substitution : substituer, dans <expr> de gauche, à la variable <nom>, l'expression <expr> de droite.

La syntaxe abstraite représentant ces expressions vous est donnée dans le fichier `syntaxe_abstraite.ml`.

Les fonctions `read_expr_string` et `read_expr_on_input_channel` (définies dans le fichier `main.ml` vous permettent de construire un arbre de syntaxe abstraite à partir d'une expression représentée par une chaîne de caractères ou saisie au clavier.

Question préliminaire 1 : visualiser le résultat de `read_expr_string "0"` puis `read_expr_string "x"`, puis de `read_expr_string "(x + y) * exists v. (t \ !v)"`

Question préliminaire 2 : représenter le BDD réduit et ordonné suivant l'ordre $x < y < t < v$ de cette dernière expression.

2.2 Implantation de BDD non réduits et non partagés

1. À partir d'un arbre de syntaxe abstraite représentant une expression,
   - Construire la liste de toutes les variables de l'expression (libres et liées) et lui associer un ordre.
   - Construire deux tables ou listes d'association établissant la correspondance : index $\leftrightarrow$ nom de variable, ainsi que les fonctions d'accès à ces associations.
2. En considérant le type suivant :

   `type bdd = Noeud of int * bdd * bdd | Feuille of bool ;;`
   - Écrire la fonction `ite i b0 b1` (`ite` pour « `if then else` »), qui crée un BDD dont la racine est d'index i avec deux successeurs : b0 comme cofacteur0 et b1 comme cofacteur1.

- Écrire les fonctions `not b`, `apply op b1 b2`, `cofactor b x v`, qui retournent respectivement le BDD représentant la négation de la fonction représentée par `b`, l'application de l'opérateur `op` à ses deux opérandes représentés par `b1` et `b2`, et le cofacteur de la fonction représentée par `b`, pour la variable `x` ayant la valeur `v`. Pour la fonction `apply`, vous serez amené à définir un type pour décrire les opérateurs booléens binaires.
- Écrire la fonction `make_bdd expr` qui retourne le BDD (ordonné) de l'expression représentée par un arbre de syntaxe abstraite.
- Écrire la fonction `print_bdd b` qui affiche soit le BDD `b` complet, soit uniquement les valuations des variables pour lesquelles la fonction représentée par `b` est vraie, soit uniquement les valuations des variables pour lesquelles la fonction représentée par `b` est fausse.

3. Tester ces fonctions sur de petites expressions booléennes. Vous incluerez ces tests dans votre compte-rendu.

2.3 Introduction aux tables de hachage : implantation d'un cache d'opérations

En vous référant aux fonctions proposées dans le module de gestion de tables de hachage de Camllight (`hashtbl`),

1. Créer une table de hachage `cache` qui a un triplet `(op,b1,b2)` de type `int*bdd*bdd` associe un bdd `b3`.
2. Créer les fonctions `find_in_cache_table` et `add_in_cache_table` associées à cette table.
3. Insérer ce cache dans la fonction `apply`.

2.4 Implantation de BDD réduits et partagés

Dorénavant, les BDD sont des arbres partagés stockés dans une table de partage implantée grâce à une table de hachage. Le module `hashtbl` de Camllight vous propose une fonction de hachage paramétrable basée sur l'égalité structurelle et non sur l'égalité physique, ce qui, dans notre cas, peut conduire en un inutile parcours des BDD dont on teste l'égalité lorsque ces derniers ne sont pas égaux. Pour contourner cette difficulté, nous allons introduire explicitement un identificateur de noeud entier pour chaque noeud de BDD, et nous appliquerons la fonction de hachage sur ces identificateurs de noeuds plutôt que sur les noeuds eux-mêmes.

En considérant maintenant que le type des BDD est le suivant :

```
type descr_bdd = Noeud of int * bdd * bdd | Feuille of bool
and bdd = {id = int; noeud = descr_bdd; mutable marque = bool;;
```

1. Créer la table de partage `ite` sous forme d'une table de hachage stockant chaque noeud, ainsi que les fonction `find_in_ite_table` et `add_in_ite_table` associées. (indication : hacher sur le triplet : `(index, identifiant(b1), identifiant(b2))` et non sur `(index, b1, b2)`).
2. Insérer les noeuds terminaux dans la table.
3. Modifier les fonctions de la bibliothèque précédente pour bénéficier automatiquement du partage.
4. Écrire une fonction `count_bdd` qui compte le nombre de noeuds d'un BDD réduit.
5. Tester la nouvelle bibliothèque sur de petits exemples de formules booléennes. A titre d'exemple, les différents circuits de la Figure 5 représentent-ils la même fonction ? Vous décrirez votre façon de procéder dans le compte-rendu.

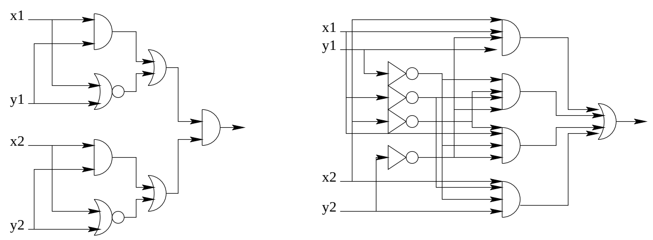

FIG. 5 – Les deux circuits présentent-ils la même fonctionnalité ?

2.5 Le problème des reines

Le problème des huit reines est extrèmement classique de par l'explosion combinatoire qu'il engendre. En fait, les BDD ne sont pas forcément le meilleur outil pour résoudre ce problème, mais nous vous le proposons car c'est un bon benchmark pour tester l'efficacité d'une bibliothèque BDD.

Le problème est simple : Étant donné un échiquier $(8 * 8)$, comment y placer huit reines sans qu'aucune d'elle ne soit en échec ? Plus généralement, étant donné un échiquier $(N * N)$, comment y placer N reines sans qu'aucune d'elle ne soit en échec ?

Une approche possible consiste à construire le BDD des solutions à ce problème : chaque case (i, j) de l'échiquier est représentée par une variable booléenne $v_{i,j}$ qui vaut 0 si la case ne peut pas contenir de reine, et 1 si elle en contient une. Il reste alors à construire l'expression de la position des huit reines sur l'échiquier (dans notre syntaxe abstraite), puis à construire le BDD associé grâce à notre fonction `make_bdd`, et enfin à imprimer les solutions.

L'expression de la position des reines est donnée par la conjonction des quatre expressions booléennes suivantes :

1. impact sur la ligne :
$$v_{i,j} \Rightarrow \prod_{1 \leq k \leq n, k \neq j} \overline{v_{i,k}}$$
2. impact sur la colonne :
$$v_{i,j} \Rightarrow \prod_{1 \leq k \leq n, k \neq i} \overline{v_{k,j}}$$
3. impact sur la diagonale montante :
$$v_{i,j} \Rightarrow \prod_{1 \leq k \leq n, 1 \leq i+j-k \leq n, k \neq i} \overline{v_{k,j+k-i}}$$
4. impact sur la diagonale descendante :
$$v_{i,j} \Rightarrow \prod_{1 \leq k \leq n, 1 \leq i+j-k \leq n, k \neq i} \overline{v_{k,j+i-k}}$$

... et de la contrainte qu'il y a n reines sur un échiquier $n * n$ (au moins une par ligne).

Tester cet exemple pour diverses tailles d'échiquiers.