UQAC
Université du Québec
à Chicoutimi

**MÉMOIRE**

PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

**PAR**

RAOUF BOUKLAB

# ÉNUMÉRATION DE TRANSVERSAUX DE CIRCUITS DE CARDINALITÉ MINIMALE À L'AIDE D'ARBRES ET/OU

JANVIER 2016

## TABLE DES MATIÈRES

## TABLE DES FIGURES

## LISTE DES TABLEAUX

## RÉSUMÉ

Le problème de trouver un transversal de circuits de cardinalité minimale dans un graphe orienté est l'un des problèmes NP-difficiles classiques définis par Karp en 1972. Il consiste à identifier un sous-graphe acyclique à partir d'un graphe donné en enlevant le moins de sommets possible. L'ensemble de sommets enlevés constitue alors un transversal de circuits de cardinalité minimale (TCCM).

Un graphe peut posséder plusieurs transversaux de circuits de même cardinalité minimale. Pour pouvoir énumérer et enregistrer la famille de tous les ensembles de solutions (TCCM) d'un graphe, il est indispensable d'utiliser une structure de données permettant de les stocker implicitement et de manière compacte afin d'optimiser l'espace mémoire occupé par la famille et de bien gérer les sommets redondants. Dans cette perspective, nous introduisons une structure de données notée *arbre et/ou*, qui est un arbre dans lequel les nœuds internes sont des connecteurs logiques ($\wedge$ et $\vee$) et les feuilles sont des valeurs de l'ensemble de solutions.

Dans ce mémoire, nous présentons la définition et l'implémentation de la structure de base des *arbres et/ou*, ainsi que la description et la mise en œuvre des différents modificateurs, requêtes et opérations qui peuvent être appliqués à cette structure dans un contexte d'énumération. Nous terminons notre travail par l'introduction d'un algorithme permettant la construction

efficace d'un arbre et/ou représentant tous les TCCM d'un graphe orienté.

**Mots clés :** complexité, NP-complétude, diagramme binaire de décision, fonction booléenne, transversal de circuits de cardinalité minimale, arbre et/ou.

## REMERCIEMENTS

En préambule à ce mémoire, je souhaite adresser tous mes remerciements et l'expression de ma profonde gratitude et ma haute reconnaissance à mon codirecteur de recherche M. Alexandre Blondin Massé. Grâce à son implication dans le projet, son dévouement, ses conseils, ses encouragements, ainsi qu'à son soutien moral et financier, ce mémoire a pu être concrétisé et mené à terme. Merci !

Je tiens également à remercier mon directeur de recherche M. Sylvain Hallé pour ses remarques, ses contributions et son aide considérable.

De même, je remercie aussi tout le corps professoral et administratif du département d'informatique et de mathématique de l'Université du Québec à Chicoutimi pour tous les enrichissements que j'ai eus ainsi que leur aide inestimable.

Je voudrais également exprimer mes sincères remerciements à mes parents, mes frères Redouane et Ramzi et ma sœur Houda qui m'ont fourni tout ce dont j'avais besoin pour que je sois dans les meilleurs conditions.

J'exprime ma reconnaissance ultime à notre Seigneur, le tout puissant, pour m'avoir permis d'arriver à cette étape de ma vie.

# INTRODUCTION

En 1971, Stephen Cook a démontré que le problème de satisfaisabilité d'une formule logique (SAT) est un problème *NP-complet* (intraitable dans un temps polynomial) [Cook, 1971]. Depuis lors, de nombreux autres problèmes ont été identifiés comme étant NP-complets dans plusieurs domaines de l'informatique (des problèmes de satisfaction de contraintes, des problèmes de graphes, des problèmes combinatoires, etc).

En 1972, Karp a dressé une liste de 21 problèmes *NP-difficiles* (c'est-à-dire dont la version décisionnelle est un problème NP-complet) [Karp, 1972]. Plus tard, en 1979, Garey et Johnson ont recensé plus de 300 problème NP-difficiles dans leur livre *Computers and Intractability : A Guide to the Theory of NP-Completeness* [Garey et Johnson, 1979]. Parmi ces problèmes NP-difficiles apparaît le problème de trouver un transversal de circuits de cardinalité minimale (TCCM). Ce problème, qui remonte au début des années 60 et également l'un des problèmes NP-difficiles classiques de la liste de Karp, consiste à identifier un sous-graphe acyclique à partir d'un graphe donné en enlevant le moins de sommets possible (c'est-à-dire casser tous les circuits par la suppression des sommets et les arcs liés à ces sommets). Par conséquent, un transversal de circuits de cardinalité minimale est un ensemble minimal de sommets dont la suppression rend le graphe acyclique. Dans l'exemple de la figure 1, l'ensemble $S =$

$\{37, 36, 16, 2, 24, 10, 13, 18, 33, 35, 6, 15\}$ est un transversal de circuits (TC) du graphe illustré. Le graphe 1(b) est le graphe acyclique obtenu en supprimant les sommets de l'ensemble $S$.

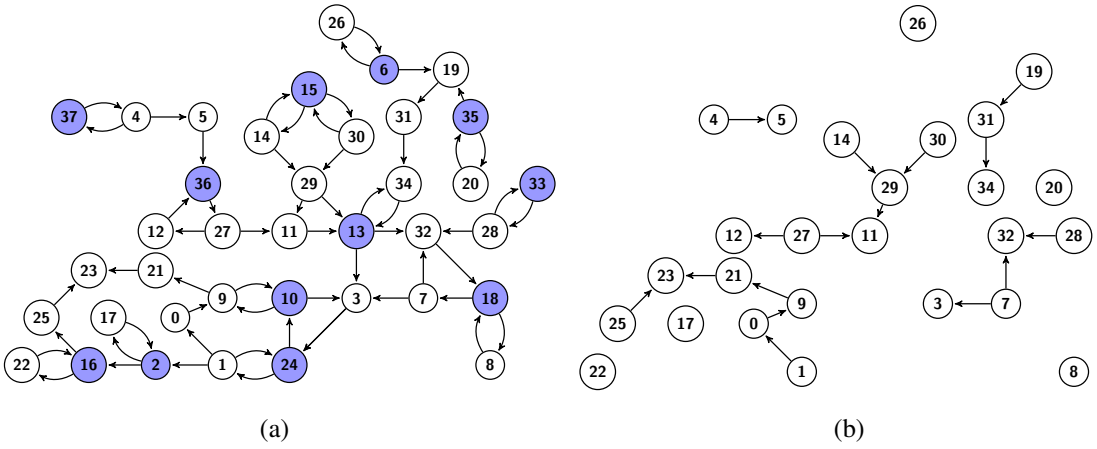



**Figure 1: Le sous-graphe acyclique induit d'un graphe. (a) Le graphe $G = (V, A)$. (b) Le sous-graphe acyclique induit par la suppression d'un transversal de circuits. L'ensemble des sommets coloriés est un transversal de circuits (TC).**

L'origine du problème TCCM vient des applications de conception des circuits logiques, mais il a trouvé sa place dans de nombreuses autres applications, telles que la prévention et la détection des inter-blocages qui peuvent se produire entre des processus concurrentiels dans les systèmes d'exploitation [Chen et al., 2008], la satisfaction des contraintes et l'inférence bayésienne [Bar-Yehuda et al., 1994], la conception assistée par ordinateur [Lin et Jou, 2000], la sécurité informatique [Gusfield, 1988], et dernièrement dans la bio-informatique [Mihai et al., 2004, Soranzo et al., 2012].

De nombreuses approches algorithmiques ont été proposées dans le but de résoudre le problème TCCM depuis sa définition, y compris les algorithmes d'approximation [Bar-Yehuda et al., 1994, Bafna et al., 1999, Even et al., 2000], la programmation linéaire [Chudak et al., 1998], la recherche locale [Brunetta et al., 2000, Zhang et al., 2013], les algorithmes probabilistes [Pardalos et al., 1999], les algorithmes paramétrés [Guo et al., 2005, Dehne et al., 2007], etc.

Parmi les premiers travaux qui ont été proposés, l'algorithme proposé par Smith et Walford qui permet de retourner un TCCM en temps exponentiel pour tout graphe, mais pour une classe de graphes appelée *Smith-WaIford one-reducible*, l'algorithme trouve un TCCM en temps polynomial égal à $O(|A||V|^2)$, où $V$ est l'ensemble de sommets et $A$ l'ensemble d'arcs [Smith et Walford, 1975]. En outre, un algorithme appelé *algorithme D*, proposé en 1979 par Shamir, permet de calculer un TCCM dans un graphe de flot réductible en temps polynomial [Shamir, 1979]. L'algorithme D a été modifié en 1982 par Rosen (*l'algorithme E*) [Rosen, 1982]. Il s'agit d'un algorithme qui retourne deux sous-ensembles de sommets $S_1$ et $S_2$ tel que $S_1 \cup S_2$ est un TCCM. Les graphes pour lesquels les algorithmes D et E retournent un TCCM sont appelés les graphes *quasi-réductibles*.

En 1985, Wang, Lloyd et Soffa ont proposé un algorithme d'ordre $O(|A||V|^2)$ pour une classe de graphes appelés *graphes réductibles acycliques* [Wang et al., 1985]. En 1988, Levy et Low ont proposé un algorithme polynomial en se basant sur une suite d'opérations de contraction permettant de réduire la taille du graphe tout en conservant les propriétés pertinentes pour trouver un TCCM [Levy et Low, 1988]. En 2000, Lin et Jou ont ajouté trois autres opérations de réduction et ont présenté un algorithme exact basé sur huit opérations dont la complexité est polynomiale pour une classe particulière de graphes appelés graphes *DOME-contractibles* [Lin et Jou, 2000].

Jusqu'à 2006, on ne connaissait pas d'algorithme exact traitant le cas général des graphes orientés et permettant de trouver un TCCM en un temps plus rapide que $O(2^n)(n = |V|)$ sauf pour certaines classes spécifiques de graphes comme les graphes bipartis ($O(1.8621^n)$), les graphes de degré maximal égal à 4 ($O(1.945^n)$) ou les graphes réductibles. Le premier algorithme général qui a brisé cette barrière est l'algorithme proposé en 2006 par Razgon qui a un temps d'ordre $O(1.8899^n)$ [Razgon, 2006]. Cette borne a été réduite par Fomin et al. à $O(1.7548^n)$ [Fomin et al., 2007].

Un autre problème intéressant lié au problème TCCM consiste à énumérer l'ensemble de tous les transversaux de circuits de cardinalité minimale d'une manière efficace. En effet, pour le même graphe, on peut trouver différents transversaux de circuits de même cardinalité minimale. Par exemple, dans le graphe de la figure 1, on peut prendre le nœud 4 à la place du nœud 37, ou le nœud 27 à la place du nœud 36. Aussi, pour une famille des ensembles de TCCM d'un graphe donné, on trouve souvent des sous-ensembles de sommets qui sont en conjonction entre eux et qui apparaissent dans plusieurs TCCM ou dans tous les TCCM comme le cas des sommets ayant des boucles. On trouve également des groupes de sommets pour lesquels exactement un des sommets apparaît dans un TCCM.

Le nombre de TCCM et la taille minimale d'un TCCM se changent d'un graphe à un autre. Dans [Fomin et al., 2007], les auteurs ont démontré qu'un graphe peut posséder au plus $O(1.8638^n)$ transversaux de circuits minimaux, où $n$ est le nombre de nœuds du graphe. Ils ont démontré également qu'il existe une grande partie de graphes possédant $O(1.5956^n)$ transversaux de circuits minimaux.

Généralement, pour résoudre un problème d'énumération, il est fréquent de devoir stocker d'une manière compacte une grande quantité d'informations qui présente des redondances afin de minimiser l'espace mémoire occupé par la famille de solutions. Différentes structures de données dédiées à ce genre de situations ont été proposées, tels que les diagrammes binaires de décision (DBD) et les diagrammes binaires de décision zéro-supprimés (DBDZ), qui sont très utilisés en combinatoire énumérative, en théorie de graphes et en théorie des jeux [Bryant, 1986]. En effet, plusieurs algorithmes d'énumération qui reposent sur la technique classique "*séparation et évaluation progressive (branch and bound)*" utilisent les DBD ou les DBDZ comme structure pour stocker les solutions combinatoires. Toutefois, ces structures ne sont pas toujours efficaces de sorte que les DBD et DBDZ sont parfois moins pratiques, surtout quand une énumération exhaustive des éléments de la famille n'est pas possible. Par exemple,

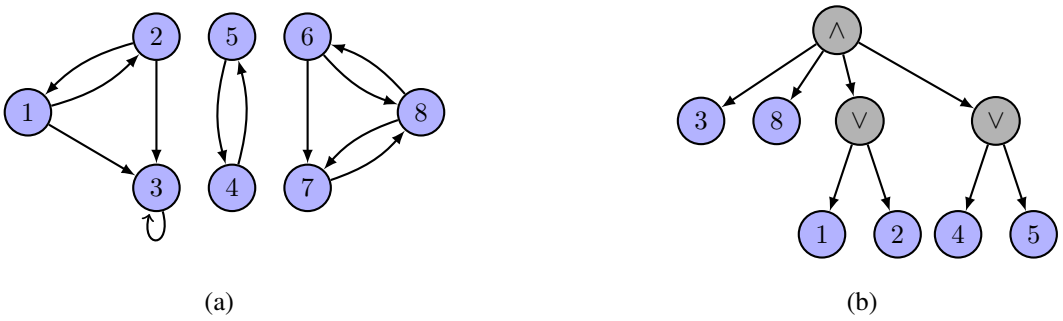
dans [Lin et Jou, 2000], les auteurs ont proposé un algorithme basé sur la technique séparation et évaluation pour calculer un TCCM dans un graphe orienté avec deux règles de branchement :

1. Lorsque le graphe n'est pas connexe, on applique l'algorithme sur chacune de ses composantes connexes ;
2. Lorsque toutes les règles ne sont pas applicables, un sommet bien choisi doit être inclus ou exclu de la solution.

Les DBD et DBDZ gèrent facilement la deuxième règle, mais il semble assez complexe de gérer la première règle. Dans l'objectif de résoudre ce problème et d'améliorer l'efficacité de l'algorithme, nous proposons une structure de données pour enregistrer la famille de solutions notée *arbre et/ou*. Un *arbre et/ou* est un arbre dans lequel les nœuds internes sont des connecteurs logiques ($\wedge$ et $\vee$) et les feuilles sont des valeurs de l'ensemble de solutions. Le connecteur logique $\wedge$ est utilisé pour rassembler les sommets et les sous-ensembles de sommets qui sont en conjonction, et le connecteur logique $\vee$ est utilisé pour rassembler les sommets et les sous-ensembles de sommets qui sont en disjonction.

Les arbres et/ou sont utilisés principalement dans le domaine de l'intelligence artificielle et aussi comme une technique pour explorer les arbres de jeu de grand espace [Kumar et N. Kanal, 1983]. En revanche, leur utilisation dans un contexte d'énumération est nouvelle et la littérature ne fait pas état de son application en tant que structure de données permettant l'énumération de familles d'ensembles de grande taille. La figure 2 montre l'arbre et/ou représentant l'ensemble des transversaux de circuits de cardinalité minimale (dans cet exemple la cardinalité minimale est égale à 4) d'un graphe orienté composé de trois composantes connexes. Pour la composante connexe de gauche, on doit éliminer le nœud 3 parce qu'il a une boucle et le nœud 1 ou le nœud 2. Pour la composante au milieu, on élimine le nœud 4 ou le nœud 5. Pour la composant de droite, on élimine le nœud 8. Ainsi, la famille des ensembles de tous les TCCM de ce graphe est $\mathcal{F} = \{\{3,1,4,8\},\{3,2,4,8\},\{3,1,5,8\},\{3,2,5,8\}\}$. Dans

la figure 2(b), le symbole $\vee$ signifie qu'on peut choisir un nœud entre les différents enfants et le symbole $\wedge$ signifie qu'on doit prendre tous les enfants.




(a) (b)

**Figure 2: L'arbre et/ou représentant la famille des TCCM d'un graphe. (a) Un graphe orienté composé de trois composantes connexes. (b) L'arbre et/ou représentant la famille des transversaux de circuits de cardinalité minimale $\mathcal{F}$ ={{3,1,4,8},{3,2,4,8},{3,1,5,8},{3,2,5,8}}.**

Ce mémoire propose deux principales contributions à l'énumération des TCCM dans des graphes orientés. La première est l'introduction de la structure de données arbre et/ou dans un contexte d'énumération, tout en donnant la définition et la mise en œuvre des différentes opérations, requêtes et modifications qui peuvent être effectuées sur cette structure, afin de la rendre plus efficace pour stocker d'une manière compacte un grand ensemble de solutions. La deuxième est l'implémentation d'un algorithme permettant de construire l'arbre et/ou représentant l'ensemble de tous les TCCM d'un graphe orienté.

La première partie de ce mémoire, qui vient de s'achever, présente une introduction au contexte des travaux, à la problématique spécifique traitée dans le mémoire, ainsi qu'à la contribution proposée dans celui-ci. Il permet au lecteur de bien voir le positionnement de la contribution proposée dans notre contexte applicatif. Le reste de ce mémoire est structuré comme suit.

Le chapitre 1 décrit quelques concepts généraux et donne des définitions et des notations nécessaires à la compréhension du reste du mémoire. Il présente des notions sur les algorithmes et leur complexité, les problèmes décisionnels, ainsi que les problèmes NP-complets et leur

réduction polynomiale. De même, il introduit des définitions et des notions sur les fonctions booléennes et l'expansion de Shannon et termine par des notions sur les graphes.

Le chapitre 2 présente une revue de la littérature sur les DBD. Il débute par l'introduction des anciens modèles de représentation des fonctions booléennes, tout en comparant les avantages et les inconvénients de ces modèles. Ensuite, il introduit la définition et les méthodes de construction des DBD et quelques uns de ses variantes. Il termine par la discussion de quelques opérations sur les DBD et les limites de leur utilisation.

Le chapitre 3 se consacre à l'étude de la structure d'arbres et/ou. On en donne la définition formelle et la description de la structure de base. Ensuite, on discute des algorithmes de manipulation de cette structure de données et des différentes opérations, requêtes et modifications qui peuvent être effectuées dans un contexte d'énumération. On termine le chapitre par la mise en place des algorithmes de conversion des arbres et/ou en DBD et vice versa.

Le chapitre 4 contient la description du problème TCCM et la mise en évidence des différents travaux liés à celui-ci. On commence le chapitre par la démonstration de la NP-complétude de la version décisionnelle du problème TCCM. Puis, on explique les travaux et les approches qui ont été proposées par les chercheurs pour résoudre ce problème. On parle particulièrement des opérations de contraction définies par Levy et Low, ainsi que les trois opérations supplémentaires introduites par Lin et Jou tout en présentant les algorithmes proposés par ceux-ci.

Le chapitre 5 se consacre à la description détaillée de notre contribution théorique, qui prend la forme d'introduction de la structure arbre et/ou pour concevoir l'algorithme permettant l'énumération de la famille de tous les TCCM d'un graphe orienté, et à l'évaluation de la contribution du mémoire d'un point de vue pratique et expérimental. La première partie présente notre algorithme et explique la façon de faire pour construire l'arbre représentant la

famille des TCCM. La deuxième partie présente l'implémentation des différents algorithmes proposés au cours de ce mémoire ainsi que les différents testes effectuer pour valider l'efficacité de ces algorithmes et la structure de données proposée.

Finalement, on conclut le mémoire en présentant un compte rendu du projet en mettant en évidence la contribution de ce travail en rapport aux recherches précédentes. Ce chapitre abordera également les limites de l'application et offrira des pistes pour les travaux futurs découlant de cette recherche.

# CHAPITRE 1

# DÉFINITIONS ET NOTATIONS

Dans ce chapitre, nous introduisons les définitions et les notions générales relatives au problème TCCM utilisées au cours de ce mémoire. Nous commençons par la définition des différents concepts qui ont trait aux algorithmes et leur complexité, en parlant de l'efficacité des algorithmes et la relation entre les différentes classes de complexité. Nous discutons aussi de la réduction polynomiale des problèmes NP-complets et de la grande problématique de la théorie de la complexité. Ensuite, nous rappelons la définition d'algèbre de Boole, des fonctions booléennes et du problème de satisfaisabilité. Et, nous terminons par des notions classiques sur les graphes. Les principales ressources sur lesquelles nous nous basons dans la description de ce chapitre sont [Liedloff, 2007, Mary, 2013, Delisle, 2015, Tourniaire, 2013, Bachelet, 2011, Lopez, 2008].

## 1.1 ALGORITHMES, COMPLEXITÉ ET NP-COMPLÉTUDE

Un *algorithme* est une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre un problème donné. Il peut être simple ou compliqué dépendamment de la difficulté du problème à traiter mais il doit obtenir une solution en un nombre fini d'étapes.

Un algorithme doit être correct, bien structuré et le temps nécessaire à son exécution ne doit pas être trop important. Il doit être simple de le traduire en un langage de programmation pour produire un programme exécutable. Parmi les problèmes que l'on est susceptible de résoudre par un algorithme, on peut distinguer plusieurs types :

1. Les problèmes de décision : déterminer si le problème possède une solution ou non.
2. Les problèmes de recherche : trouver une solution et la retourner si elle existe.
3. Les problèmes de comptage : calculer le nombre de solutions d'un problème.
4. Les problèmes d'énumération : calculer toutes les solutions d'un problème.

Dans plusieurs cas, ces problèmes sont tellement complexes qu'il n'est pas possible à l'être humain de compter uniquement sur la puissance de son ordinateur pour les résoudre. Ainsi, pour un problème donné, on cherche toujours à avoir l'algorithme le plus efficace possible. Cette efficacité dépend de ce qu'on cherche à optimiser : il peut s'agir du temps d'exécution de l'algorithme dans le pire cas, le temps d'exécution moyen de l'algorithme, ou encore l'espace mémoire qu'il utilise.

Généralement, l'*efficacité* d'un algorithme dépend crucialement de la difficulté du problème à résoudre car le degré de difficulté varie d'un problème à l'autre. Afin de pouvoir classer les problèmes selon leur difficulté et de mesurer l'efficacité d'un algorithme résolvant un problème donné indépendamment de la machine et du matériel utilisé, une notion dite *théorie de la complexité* a été introduite. Cette théorie, qui s'est beaucoup développée au cours des cinquante dernières années, permet d'évaluer le nombre d'opérations nécessaires à un algorithme pour s'exécuter et ainsi de classifier les problèmes informatiques selon leur difficulté.

La complexité d'un algorithme est généralement mesurée par une fonction $T(n)$, qui indique le nombre d'opérations effectuées lors de l'exécution de l'algorithme en fonction de la taille $n$ des données en entrées (c'est-à-dire les paramètres que l'on donne à l'algorithme). Quand la

taille des données du problème traité devient de plus en plus grande, on calcule la complexité asymptotique (à l'aide de la notation *grand-O*) plutôt que d'utiliser une mesure exacte du temps d'exécution, ce qui est d'ailleurs impossible puisque le temps réel d'exécution dépend de la machine utilisée, du langage de programmation, etc.

Le critère du temps n'est pas toujours celui qui nous intéresse. On peut aussi vouloir estimer l'espace mémoire utilisé par un algorithme. Dans ce cas, on parle de *complexité spatiale*. Le tableau 1.1 représente les différentes classes de la complexité selon le grandeur $O$ (de la plus complexe à la plus simple).

| **Complexité** | **Vitesse** | **Temps** | **Grandeur** $O$ |
|---|---|---|---|
| Factorielle | très lent | proportionnel à $n^n$. | $O(n!)$ |
| Exponentielle | lent | proportionnel à une constante $k \geq 2$ à la puissance $n$. | $O(k^n)$ |
| Polynomiale | moyen | proportionnel à $n$ à une puissance donnée $k \geq 2$. | $O(n^k)$ |
| Quasi-linéaire | assez rapide | intermédiaire entre linéaire et polynomial. | $O(nlog(n))$ |
| Linéaire | rapide | proportionnel à $n$. | $O(n)$ |
| Logarithmique | très rapide | proportionnel au logarithme de $n$. | $O(log(n))$ |
| Constante | le plus rapide | indépendant de la donnée. | $O(1)$ |

**Tableau 1.1: Les classes de la complexité temporelle selon la grandeur O.**

On dit qu'un algorithme est *polynomial* (ou, dans notre cas, *efficace*) si sa complexité est bornée dans le pire cas par un polynôme ayant la taille des données d'entrées comme variable. Ainsi, l'algorithme est efficace s'il reste d'ordre polynomial même si la taille des données en entrée devient très importante. Dans le tableau 1.1, les algorithmes de type *factoriel* et *exponentiel* ne sont pas efficaces car le temps d'exécution explose lorsque la taille des données d'entrée augmente. En revanche, les autres classes sont utilisables.

**Exemple 1.** Considérons les algorithmes $A$, $B$ et $C$ ayant les complexités montrées dans le

tableau 1.2.

| Algorithme | A | B | C |
|---|---|---|---|
| Complexité | $80 \cdot n$ | $10 \cdot n^2$ | n ! |
| Pour n=4 | 320 | 160 | 24 |
| Pour n=20 | 1600 | 4000 | $24 \cdot 10^{18}$ |

**Tableau 1.2: Tableau comparatif entre la complexité des algorithmes A, B et C pour n=4 et n=20.**

L'algorithme le plus efficace pour 4 éléments est l'algorithme *C*. Mais, pour 20 éléments, on s'aperçoit tout de suite que l'algorithme *C* n'est plus utilisable. Par contre, *A* et *B* restent applicables. Cet exemple justifie le fait que, si le nombre d'opérations n'explose pas avec une augmentation de la taille des données en entrée, l'algorithme est considéré efficace.

La théorie de la complexité se concentre sur l'étude de la complexité des différents types de problèmes (décision, optimisation, dénombrement,...). Pour des raisons de simplicité, on limite notre étude sur la complexité des problèmes de décision. *Un problème de décision* (ou décisionnel) est un problème dont la réponse est oui ou non. On représente fréquemment un problème de décision par un ensemble $X$ qui est un sous-ensemble de $Y$. Alors, pour chaque instance $y \in Y$, le problème consiste à décider si $y \in X$ ou non. Cela justifie que les problèmes qui ne sont pas décisionnels, comme les problèmes d'optimisation, peuvent être transformés en un problème de décision de difficulté équivalente ou supérieure.

**Exemple 2.** Le problème de décision TCCM est formulé comme suit :
***Instance :*** Un graphe orienté $G = (V, A)$, un entier $k$ ;
***Question :*** Le graphe $G$ possède t-il un transversal $S$ de $k$ sommets $S = \{v_1, v_2, ..., v_k\}$, c'est-à-dire que la suppression des sommets de $S$ casse tous les circuits de $G$ ?

La théorie de la complexité permet de diviser les problèmes décisionnels selon le temps d'exécution nécessaire à leur résolution en plusieurs classes, les plus connues étant les suivantes.

1. **La classe P.** C'est la classe des problèmes décidables par une machine de Turing déterministe en temps polynomial par rapport à la taille des donnée en entrée (le plus court chemin, l'arbre de poids minimal, flot maximal, etc.).
2. **La classe NP.** Il s'agit de la classe des problèmes décidables par une machine de Turing non déterministe en temps polynomial par rapport à la taille de l'entrée (systèmes d'équations linéaires, problèmes des nombres premiers, cycle hamiltonien, etc.).
3. **La classe NP-complets.** Un problème de décision est NP-complet lorsqu'il est NP, et s'il est au moins aussi difficile que tous les autres problèmes NP (satisfaisabilité d'une formule booléenne, problème de sac à dos, voyageurs de commerce, problème du coloriage de graphe, etc.).
4. **La classe NP-difficiles :** est la classe des problèmes qui sont au moins aussi difficiles que tout problème de NP (problème de l'arrêt, somme des sous-ensembles, transversaux de circuits de cardinalité minimale, etc.).

Les problèmes NP-complets et NP-difficiles sont souvent confondus, la différence entre ces classes de problèmes est qu'un problème NP-difficile peut ne pas être dans NP et n'est pas nécessairement un problème de décision. Les problèmes NP sont généralement difficiles à résoudre, le souci reste celui de trouver une technique de résolution dont le temps augmente polynomialement lorsque la taille du problème à résoudre augmente. En fait, la problématique centrale la plus connue, en théorie de la complexité, est la fameuse question : Est ce que P = NP ? En d'autres termes, est-ce qu'un problème NP-complet peut être résolu en temps polynomial ? Ce problème est ouvert depuis que Stephen Cook a démontré son théorème en 1971 qui dit que le problème de satisfaisabilité d'une formule booléenne (SAT) est NP-complet et que tous les problèmes de la classe NP sont polynomialement réductibles au problème SAT [Cook, 1971]. En 2001, le problème P = NP ? a été classé parmi les sept problèmes non résolus jugés les plus importants et dotés chacun d'un prix d'un million de dollars.

Le théorème de Cook a permis d'identifier beaucoup d'autres problèmes comme étant NP-complets par l'application des réductions polynomiales. Ainsi, pour prouver qu'un problème est NP, il suffit de démontrer qu'il est polynomialement réductible à un problème NP déjà connu, donc l'équivalence entre les problèmes NP est généralement obtenue à l'aide de réductions polynomiales.

Plus précisément, on dit qu'un problème $\Pi_1$ est *polynomialement réductible* à $\Pi_2$ (noté $\Pi_1 <_p \Pi_2$) s'il existe une fonction $f$ calculable en temps polynomial telle que toute instance $I$ de $\Pi_1$ admet une solution si et seulement si $f(I)$ admet une solution pour le problème $\Pi_2$.

Un problème de décision $\Pi_1$ est dit *NP-complet* si, $\Pi_1 \in NP$ et $X <_p \Pi_1, \forall X \in NP$. Si $\Pi_1$ est polynomialement réductible à $\Pi_2$ et si $\Pi_2$ est résolu en un temps polynomial, alors $\Pi_1$ est également résoluble en temps polynomial.

La figure 1.1 représente les frontières entre les différentes classes de la complexité. La partie gauche représente le cas où les classes P, NP et NP-complet sont différentes et la partie droite représente le cas où les classes P, NP, NP-complet sont égales.

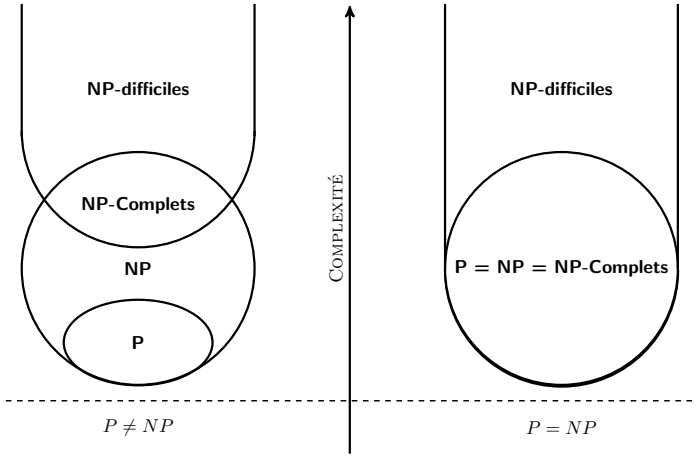



**Figure 1.1: Diagrammes de Venn pour les classes des problèmes P, NP, NP-complets et NP-difficiles dans le cas où $P = NP$ et le cas $P \neq NP$.** `https://en.wikipedia.org/wiki/NP-hardness`**.**

Jusqu'à ce jour, on ne peut donc pas affirmer s'il existe ou non un algorithme polynomial pour résoudre chacun des problèmes NP. On sait que P est inclus dans NP mais rien ne nous permet d'affirmer que P n'est pas NP. La réponse à cette question permettrait de résoudre plusieurs autres grandes questions de la théorie de la complexité. En effet, l'obtention d'un algorithme polynomial pour résoudre *un seul* problème NP-complet permettrait immédiatement de résoudre *n'importe quel* problème NP en un temps polynomial. Inversement, si l'un d'eux est prouvé être intraitable en temps polynomial, alors la classe NP est distincte de la classe P. L'attente de la plupart des informaticiens et mathématiciens est que l'égalité soit fausse. Malheureusement, il n'existe pas d'argument rigoureux appuyant cette intuition.

## 1.2 ALGÈBRE DE BOOLE ET FONCTIONS BOOLÉENNES

L'algèbre de Boole forme une composante importante de l'informatique et de la conception des systèmes numériques. Soit un ensemble des valeurs de vérité $\mathbb{B} = \{0, 1\}$ où 0 représente la valeur faux et 1 la valeur vraie. L'ensemble $\mathbb{B}$ muni des opérateurs logiques *et* ($\wedge$), *ou* ($\vee$) et *non* ($\neg$) forme une *algèbre de Boole*. Soit $n$ un entier supérieur ou égal à 1. On note $\mathbb{F}_n$ l'ensemble des fonctions $f : \mathbb{B}^n \longrightarrow \mathbb{B}$ et $\mathbb{F} = \bigcup_{n=0}^{\infty} \mathbb{F}_n$. L'ensemble $\mathbb{F}$ muni des mêmes opérateurs que $\mathbb{B}$, est aussi une algèbre de Boole.

Une fonction définie sur une algèbre de Boole est appelée *fonction booléenne*. Une *fonction booléenne* (ou *fonction binaire*) à $n$ variables $x_1, x_2, \ldots, x_n$ est donc toute application $f$ définie de $\mathbb{B}^n$ vers $\mathbb{B}$ comme suit :

$$f : \mathbb{B}^n \longrightarrow \mathbb{B}$$

$$(x_1, x_2, \ldots, x_n) \longmapsto f(x_1, x_2, \ldots, x_n).$$

Une telle fonction booléenne $f$ est entièrement définie par la donnée d'une table associant à

chacune des $2^n$ valeurs possibles pour le n-uplet $(x_1, x_2, \ldots, x_n)$ la valeur de $f(x_1, x_2, \ldots, x_n)$.

Pour la simplification des fonctions booléennes et pour mieux comprendre le fonctionnement de certains circuits usuels on utilise souvent la *décomposition de Shannon*. C'est en effet sur ce théorème de décomposition que s'appuient la plupart des techniques modernes de manipulation et d'optimisation des fonctions booléennes. Le théorème de Shannon a rendu le traitement d'une fonction booléenne d'une manière récursive assez facile ainsi que sa décomposition par rapport à ses variables.

Si une variable $x_i$ de la fonction $f$ est remplacée par une constante $b$, la fonction résultante est appelé une *restriction* ou *cofacteur* de $f$ par rapport à $x_i$, et noté $f|_{x_i=b}$. Pour toute variable $x_i \in \{x_1,x_2,...,x_i,...,x_n\}$ on a :

$$f|_{x_i=b}(x_1,\ldots,x_i,\ldots,x_n) = f(x_1,\ldots,x_{i-1},b,x_{i+1}\ldots,x_n).$$

Ainsi, pour toute variable $x_i$ de $X = \{x_1,x_2,\ldots,x_i,\ldots,x_n\}$, la *décomposition de Shannon* de $f$ sur la variable $x_i$ s'énonce par la formule

$$f(x_1,x_2,\ldots,x_i,\ldots,x_n) = (\neg x_i \wedge f(x_1,x_2,\ldots,0,\ldots,x_n)) \vee (x_i \wedge f(x_1,x_2,\ldots,1,\ldots,x_n))$$

Si on introduit la notion du cofacteur dans la décomposition de Shannon de $f$ sur $x_i$, on obtient

$$f(x_1,x_2,\ldots,x_i,\ldots,x_n) = (x_i \wedge f|_{x_i=1}) \vee (\neg x_i \wedge f|_{x_i=0})$$

où $f|_{x_i}$ est le cofacteur de $f$ par rapport à $x_i$.

Plusieurs problèmes de conception logique, de vérification, d'intelligence artificielle et de combinatoire peuvent être exprimées comme une séquence d'opérations par des fonctions

booléennes. Cependant, plusieurs problèmes liés à ces fonctions tels que le problème de satisfaisabilité (*SAT*), le problème de vérification de la tautologie ou l'équivalence de deux expressions booléennes sont NP-complets ou coNP-complets (complément d'un problème NP-complet). Parmi ces problèmes les plus importants, le problème SAT qui consiste à décider si une formule booléenne mise sous forme normale conjonctive (voir définition 2) admet ou non une valuation qui la rend vraie.

Le problème *SAT* est formulé comme suit : Étant donné un ensemble $X = \{x_1, x_2, ..., x_n\}$ de $n$ variables, chaque variable peut prendre deux valeurs (vrai, faux). Un *littéral* est une expression logique formée d'une seule variable $x_i$ ou de sa négation $\neg x_i$ avec $i \in [1..n]$. Une *clause* est une *expression logique* formée uniquement de la disjonction de littéraux (exemple : $x_1 \vee \neg x_2 \vee x_3$). Soit $m$ clauses $C_i$ formées à l'aide de $n$ variables (littéraux) et la *formule* $\Phi = C_1 \wedge C_2 \wedge ... \wedge C_m$. Le problème *SAT* consiste à savoir si on peut affecter une valeur à chaque variable $x_i$ telle que la valeur de la formule $\Phi$ soit vraie.

Le problème *SAT* a plusieurs variantes. On citera $k$-SAT dans lequel on cherche la satifaisabilité d'une formule constituée d'une conjonction de disjonctions de $k$ littéraux. Le problème 3-SAT (pour $k = 3$) est particulièrement utile et il est démontré lui aussi NP–complet. Pour le démontrer, il suffit de prouver que le problème *SAT* est polynomialement réductible à 3-SAT. En revanche, 2-SAT est de classe P.

Beaucoup de problèmes d'optimisation dans différents domaines comme la vérification formelle de circuits, la cryptographie, la configuration de produits, la planification, et même la bio-informatique, peuvent se réduire assez naturellement à l'une ou l'autre des variantes de *SAT* et ainsi peuvent être modélisés par des formules booléennes. La complexité pour résoudre ces problèmes a fait l'objet de nombreuses publications. À l'heure actuelle, on ne connaît pas d'algorithme meilleur que $O(2^n)$ en pire cas pour résoudre *SAT* [Tourniaire, 2013].

## 1.3 GRAPHES

Les graphes sont des structures de données qui jouent un rôle fondamental en informatique et en mathématiques. Ils permettent de représenter naturellement de nombreux problèmes et ils sont considérés comme l'un des instruments les plus efficaces pour résoudre des problèmes posés au sein de différentes disciplines telles que la chimie, la biologie, les sciences sociales et les réseaux. Ils permettent de représenter simplement la structure, les connexions, les relations, les dépendances et les cheminements possibles entre les éléments d'un ensemble comprenant un grand nombre de situations.

Un *graphe* est un ensemble fini de sommets (nœuds dans le contexte des réseaux) et d'arcs (arêtes dans le cas des graphes non orientés) défini comme un couple $G = (V,A)$ où $V = \{v_1, v_2, ..., v_n\}$ est l'ensemble des sommets et $A = \{a_1, a_2, ..., a_m\}$ l'ensemble des arcs. Un *arc* $a_1 = (v_1, v_2)$ est un élément du produit cartésien $V \times V = \{(v_1, v_2) \mid v_1, v_2 \in V\}$. Le sommet $v_1$ est l'*extrémité initiale* de l'arc et le sommet $v_2$ l'*extrémité finale* (ou bien *origine* et *destination*).


**Figure 1.2: Graphe orienté et le graphe non orienté lui associé. (a) Graphe orienté $G = (V,A)$, l'arc $u = (v_3, v_3)$ est une boucle. (b) Le graphe non orienté induit du graphe orienté de la figure 1.2(a).**

Dans beaucoup d'applications, les relations entre les éléments d'un ensemble sont orientées, un élément $x$ peut être en relation avec un autre $y$ sans que $y$ soit nécessairement en relation avec $x$, on parle alors de *graphes orientés*. Dans d'autres cas, l'orientation des arcs ne joue

aucun rôle, on s'intéresse simplement à l'existence d'arc entre deux sommets (sans en préciser l'ordre), dans ce cas, on parle de *graphes non orientés*. Dans ce mémoire, on s'intéresse à l'étude du problème TCCM dans les graphes orientés.

Soit $G = (V,A)$ un graphe orienté, $V$ est un ensemble fini de sommets, et $A \subseteq (V \times V)$ l'ensemble des arcs. On note $n = |V|$ le nombre de sommets et $m = |A|$ le nombre d'arcs. Pour tout sous-ensemble $U \subseteq V$, on note $G[U]$ le *sous-graphe induit* par $U$ sur $G$, et $A(U) = A \cap U^2$ les arcs de $G[U]$. Soit $u \in V$ un sommet, on note $N^-(u) = \{v \in V \mid (v,u) \in A\}$ l'ensemble des *prédécesseurs* de $u$, et $N^+(u) = \{v \in V \mid (u,v) \in A\}$ l'ensemble des *successeurs* de $u$. Également, on note $A^-(u) = \{(v,u) \mid v \in N^-(u)\}$ l'ensemble des arcs entrants à $u$, et $A^+(u) = \{(u,v) \mid v \in N^+(u)\}$ l'ensemble des arcs sortants de $u$. Deux sommets sont *adjacents (ou voisins)* s'ils sont joints par un arc. Deux arcs sont *adjacents* s'ils ont au moins une extrémité commune. L'ensemble de sommets *adjacents (voisins)* de $u$ est $N(u) = \{N^-(u) \cup N^+(u)\}$, et l'ensemble d'*arcs adjacents* de $u$ est $A(u) = \{A^-(u) \cup A^+(u)\}$. *Le degré intérieur* $deg^-(u) = |N^-(u)|$ de $u$ est le nombre d'arcs ayant $u$ comme extrémité finale. *Le degré extérieur* $deg^+(u) = |N^+(u)|$ de $u$ est le nombre d'arcs ayant $u$ comme extrémité initiale. *Le degré* d'un sommet $u$ est $deg(u) = deg^-(u) + deg^+(u)$. Un sommet $u$ est une *source* si $deg^-(u) = 0$. Un sommet $u$ est un *puits* si $deg^+(u) = 0$.

Dans un graphe orienté $G = (V,A)$, un *chemin* est une suite de sommets $(u_0, u_1, ..., u_k)$ tel que $(u_i, u_{i+1}) \in A$ pour $i = 0, 1, ..., k-1$. Le nombre $k$ est appelé la *longueur du chemin*. Un *circuit* $c$ est un chemin non vide qui commence et termine par le même sommet. Un graphe orienté est *acyclique* s'il ne contient aucun circuit. *Un transversal de circuits* est un sous-ensemble de sommets $U \subseteq V$ ayant la propriété que le sous-graphe induit $G' = (V - U, A \cap (V - U)^2)$ est acyclique. Un transversal de circuits est dit *minimum* s'il est de taille minimale.

Un graphe $G = (V,A)$ est *connexe* si pour tous sommets $u, v \in A$, il existe un chemin entre $u$ et $v$.

On note $u \to v$ s'il y a un chemin (peut être vide) de $u$ vers $v$, et $u \leftrightarrow v$ s'il y a $u \to v$ et $v \to u$. Le sous-graphe induit par les classes d'équivalence de $\leftrightarrow$ sont appelées *composantes fortement connexes* de $G$. Autrement dit, une composante fortement connexe est un sous-ensemble de sommets tel qu'il est possible d'atteindre chacun des sommets à partir de l'autre.

Un graphe orienté est *complet* si quels que soient deux sommets, il existe un arc les reliant dans un sens et un arc les reliant dans l'autre sens, c'est-à-dire, pour tous sommets $u, v \in V, (u,v) \in A$ et $(v,u) \in A$. Une *clique* de $G$ est un ensemble de sommets $U \subseteq V$ tel que $G[U]$ est un graphe complet sans boucles, c'est-à-dire, pour tout sommet $u \in U, (u,u) \notin A$ et quels que soient $u, v \in U, (u,v) \in A$ tel que $u \neq v$.

Si $G = (V,E)$ un graphe non orienté, une *couverture par sommets* de $G$ est un ensemble de sommets $U \subseteq V$ tel que chaque arrête de $G$ est incidente à au moins un sommet de $U$, c'est-à-dire, un sous-ensemble de sommets $U \subseteq V$ tel que pour chaque arrête $(u,v) \in E$ on a $u \in U$ ou $v \in U$. Une *couverture minimale par sommets* est une couverture par sommets de taille minimale.

Les graphes sont très utilisés dans la résolution des problèmes. Cependant, en théorie des graphes, on rencontre plusieurs problèmes difficiles qui ne sont pas résolubles en temps polynomial (le problème de décider si un graphe contient un cycle hamiltonien, le problème de coloriage, le problème de la clique maximale, le problème de couverture minimale, le problème TCCM, le problème de la coupe maximum, etc.).

# CHAPITRE 2

# DIAGRAMMES BINAIRES DE DÉCISION

Depuis l'origine de l'informatique, l'étude des fonctions booléennes s'est grandement développée et plusieurs représentations de celles-ci ont été suggérées, car le moyen de représenter ces fonctions influe grandement sur l'efficacité avec laquelle on les utilise par la suite. Parmi les premières représentations des fonctionnes booléennes, on trouve *les formules booléennes*, *les tables de vérité* et *les diagrammes de Karnaugh*. L'inconvénient majeur de ces représentations est qu'elles occupent un espace exponentiel par rapport au nombre de variables en entrée.

Afin de répondre aux problèmes posés par l'accroissement de la taille des fonctions booléennes, de nouvelles techniques de représentation ont été développées. Ces nouvelles techniques sont généralement basées sur la récursivité du théorème de Shannon et non seulement sur les théorèmes habituels du calcul booléen. Parmi les représentations les plus connues, on compte *les arbres binaires de décision (ABD)* et *les diagrammes binaires de décision (DBD)*. Cette dernière représentation constitue un modèle particulièrement efficace pour représenter et manipuler les fonctions booléennes.

En 1978, Akers a introduit les DBD pour la première fois [Akers, 1978]. Huit ans plus tard, Bryant a proposé une représentation canonique des DBD notée *diagrammes binaires de décision ordonnés et réduits (DBDOR)*, ainsi que des algorithmes pour calculer efficacement

les opérations booléennes sur ces structures de données [Bryant, 1986]. Depuis lors, les diagrammes binaires de décisions n'ont cessé de gagner en popularité et plusieurs variantes de ceux-ci ont été développées telles que les diagrammes binaires de décision zéro-supprimés (DBDZ), les diagrammes de moments binaires (DMB), les diagrammes binaires de décision à terminaux multiples (DBDTM), etc.

Nous verrons dans ce chapitre les différentes modèles les plus souvent utilisés pour représenter les fonctions booléennes ; nous nous intéressons particulièrement à l'étude des diagrammes binaires de décision et ses variantes ainsi qu'aux algorithmes de manipulation de base, et aux opérations qui peuvent être effectuées sur ceux-ci, tout en expliquant les avantages et les inconvénients d'utilisation de chaque modèle. Nous terminons le chapitre par l'étude de quelques variantes des DBD et la discussion du fait que la taille des *diagrammes binaires de décision ordonnés (DBDO)* peut varier selon l'ordre de variables choisi et qu'il y a des façons de la réduire. Le lecteur intéressé en savoir davantage sur les fonctions booléennes et les DBD est référé aux articles [Leblet, 2004, Bryant, 1986].

## 2.1 FORMULES BOOLÉENNES, TABLES DE VÉRITÉ ET DIAGRAMMES DE KARNAUGH

Nous commençons avec l'introduction de quelques définitions de base.

**Définition 1.** *Une* formule booléenne *est une formule construite récursivement à partir des constantes booléennes 0 et 1, des variables $x_i$ de $\mathbb{B}^n$ et des connecteurs logiques et ($\wedge$), ou ($\vee$) et non ($\neg$). Plus précisément,*

– 0 *et* 1 *sont des formules ;*

– *Les variables $x_1, x_2, \ldots, x_n$ de $\mathbb{B}^n$ sont des formules ;*

– *Si $f$ est une formule, alors $\neg f$ est une formule ;*

– *Si $f_1$ et $f_2$ sont des formules, alors $f_1 \wedge f_2$ et $f_1 \vee f_2$ sont des formules de $\mathbb{B}^n$.*

**Exemple 3.** Si l'on note $X = \{x_1, x_2, x_3\}$ l'ensemble des variables de $\mathbb{B}^3$, alors la formule $f = (x_1 \vee \neg(x_2 \wedge x_3)) \wedge (\neg x_2 \vee (x_1 \wedge x_3))$ est une formule booléenne de $\mathbb{B}^3$.

**Définition 2.** *Une formule booléenne $f$ définie sur X est dite en forme normale conjonctive (FNC) si elle est de la forme : $f = \bigwedge_{i=1}^{m} f_i$ avec $f_i = \bigvee_{j=1}^{n_i} x_{ij}$ et chaque $x_{ij} \in X \cup \bar{X}$, où $\bar{X}$ représente l'ensemble des négations des variables de X.*

**Définition 3.** *Une formule booléenne $f$ définie sur X est dite en forme normale disjonctive (FND) si elle est de la forme : $f = \bigvee_{i=1}^{m} f_i$ avec $f_i = \bigwedge_{j=1}^{n_i} x_{ij}$ et chaque $x_{ij} \in X \cup \bar{X}$.*

Toute formule booléenne est équivalente à une expression en forme normale conjonctive et une expression en forme normale disjonctive. Il est facile de passer d'une forme à l'autre, il suffit de développer les termes dans les parenthèses (voir l'exemple 4).

**Exemple 4.** Soit la formule $f = (x_1 \wedge (\neg x_3 \vee \neg x_4)) \vee (x_2 \wedge (\neg x_3 \vee \neg x_4))$.

1. On peut écrire $f$ en FNC comme suite : $(x_1 \vee x_2) \wedge (\neg x_3 \vee \neg x_4)$.
2. On peut écrire $f$ en FND comme suite : $(x_1 \wedge \neg x_3) \vee (x_1 \wedge \neg x_4) \vee (x_2 \wedge \neg x_3) \vee (x_2 \wedge \neg x_4)$.

L'inconvénient du FNC et FND est qu'une formule booléenne peut avoir plusieurs représentations en FNC et plusieurs représentations en FND.

**Définition 4.** *La table de vérité d'une fonction booléenne $f \in \mathbb{F}^n$ est un tableau de $2^n$ lignes et $n+1$ colonnes. Les n premières colonnes représentent les valeurs prises par les variables $x_1, x_2, ..., x_n$ et la dernière colonne représente l'évaluation de la fonction $f(x_1, x_2, ..., x_n)$.*

**Exemple 5.** Le tableau 2.1 montre la table de vérité associée à la fonction booléenne $f = (a \wedge b) \vee (\neg b \wedge c)$.

| a | b | c | *f(a,b,c)* |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

**Tableau 2.1: Table de vérité associée à la fonction f** = $(\mathbf{a} \wedge \mathbf{b}) \vee (\neg \mathbf{b} \wedge \mathbf{c})$

**Définition 5.** *Le code Gray est un code construit de telle façon qu'à partir du chiffre 0, chaque nombre consécutif diffère du précédent immédiat d'un seul digit.*

La méthode la plus simple et la plus utilisée pour calculer le code Gray d'un nombre est celle qui consiste à calculer le *OU Exclusif* entre le code binaire de ce nombre et ce même code binaire décalé d'un rang à droite. Cette méthode permet de passer d'une ligne à la suivante en inversant le bit le plus à droite conduisant à un nombre nouveau. Dans la figure 2.1, le tableau 2.1(a) représente le code Gray des nombres 0 à 7 et la figure 2.1(b) explique comment calculer le code Gray du nombre 7 en utilisant le *OU Exclusif*, le premier opérande est le nombre 7 codé en binaire, le deuxième opérande est ce codage binaire décalé d'un rang à droite, le symbole (^) désigne le *OU Exclusif*, le résultat de l'opération est le code Gray du nombre 7.

| Décimal | Binaire | Gray |
|---|---|---|
| **0** | 000 | 000 |
| **1** | 001 | 001 |
| **2** | 010 | 011 |
| **3** | 011 | 010 |
| **4** | 100 | 110 |
| **5** | 101 | 111 |
| **6** | 110 | 101 |
| **7** | 111 | 100 |

(a)

$$\begin{array}{r} 0111 \\ \hat{} \ 0011 \\ \hline 0100 \end{array}$$

(b)

**Figure 2.1: Méthode de codage Gray. (a) Le code Gray des nombres 0 à 7. (b) Le code Gray du nombre 7 calculé par le OU Exclusif.**

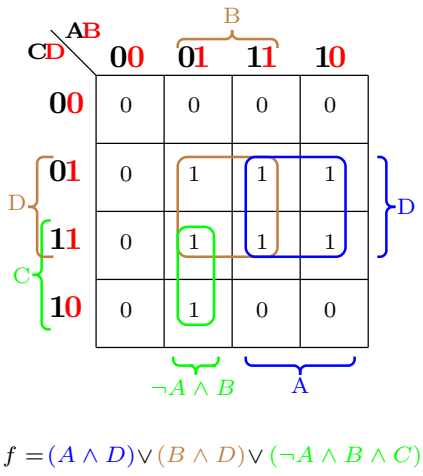
**Définition 6.** *Un* tableau de Karnaugh *est un tableau de* $2^n$ *cases, n étant le nombre de variables. Sur les lignes et les colonnes du tableau, on place l'état des variables d'entrée en suivant le code Gray et dans chacune des cases du tableau, on place l'état de la sortie pour les combinaisons d'entrée correspondantes.*

Les diagrammes de Karnaugh servent à calculer la FNC et la FND d'une formule, simplifier les formules booléennes et trouver la formule booléenne correspondante à une table de vérité d'une fonction booléenne. Pour calculer la formule booléenne à partir d'un tableau de Karnaugh, on commence par la constitution des groupes (carrés, rectangles) de $1, 2, 4, 8, 16, ..., 2^n$ cases adjacentes (on choisit des cases contenant des 1 ou des 0 pas les deux à la fois). Deux cases sont adjacentes si une seule variable change l'état d'une case à l'autre. Les groupes doivent contenir un nombre de cases égal à une puissance de 2. On doit avoir le plus petit nombre de groupes possible et chaque groupe doit contenir le maximum de termes 1 (ou 0). Les termes d'un même groupement sont liés par l'opérateur logique ($\wedge$). Les groupements sont liés par l'opérateur logique ($\vee$). La figure 2.2 explique comment calculer la formule $f(A,B,C,D) = (A \wedge D) \vee (B \wedge D) \vee (\neg A \wedge B \wedge C)$ correspondante à la table de vérité de la figure 2.2(a) en utilisant les diagrammes de Karnaugh. On commence par la construction des groupes de cases contenant des 1. Puis, pour chaque groupe, on calcule la formule correspondante, le carré bleu correspondant à la formule $(A \wedge D)$, le carré brun correspondant à la formule $(B \wedge D)$ et le rectangle vert correspondant à la formule $(\neg A \wedge B \wedge C)$.

Les formules booléennes sont très intéressantes parce qu'elles permettent de travailler facilement sur les fonctions booléennes. Leur grande expressivité leur permet d'être comprises rapidement par tout le monde. De plus, n'importe quelle fonction booléenne peut être représentée par une formule booléenne. Réciproquement, chaque formule booléenne représente bien une fonction booléenne. Cependant, leur inconvénient majeur est qu'il n'y a pas d'unicité de représentation d'une fonction booléenne par une formule booléenne. Ainsi, deux formules

| | A | B | C | D | $f$ |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 2 | 0 | 0 | 1 | 0 | 0 |
| 3 | 0 | 0 | 1 | 1 | 0 |
| 4 | 0 | 1 | 0 | 0 | 0 |
| 5 | 0 | 1 | 0 | 1 | 1 |
| 6 | 0 | 1 | 1 | 0 | 1 |
| 7 | 0 | 1 | 1 | 1 | 1 |
| 8 | 1 | 0 | 0 | 0 | 0 |
| 9 | 1 | 0 | 0 | 1 | 1 |
| 10 | 1 | 0 | 1 | 0 | 0 |
| 11 | 1 | 0 | 1 | 1 | 1 |
| 12 | 1 | 1 | 0 | 0 | 0 |
| 13 | 1 | 1 | 0 | 1 | 1 |
| 14 | 1 | 1 | 1 | 0 | 0 |
| 15 | 1 | 1 | 1 | 1 | 1 |

(a)




(b)

**Figure 2.2: Calcul de la formule booléenne correspondante à une table de vérité en utilisant les tableaux de Karnaugh. (a) Table de vérité d'une fonction $f(A,B,C,D)$. (b) Le diagramme de Karnaugh et la formule booléenne correspondants à la table de vérité de (a).**

booléennes différentes peuvent représenter la même fonction booléenne (voir l'exemple 6).

**Exemple 6.** On peut représenter la fonction *XOR* par la formule $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$ ou par $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$.

Le problème majeur des formules booléennes est de pouvoir savoir si deux formules booléennes différentes sont bien égales comme fonctions booléennes. Si oui, trouver cette égalité reste aussi un problème très difficile.

L'intérêt des tables de vérité et des tableaux de Karnaugh est qu'une fonction booléenne possède une unique représentation. Par contre, elle utilise un espace exponentiel par rapport au nombre de variables ($2^n$ entrées pour les tables et $2^n$ cases pour les tableaux de Karnaugh) et elle est peu lisible. L'ensemble des tables de vérité et des diagrammes de Karnaugh est en bijection avec l'ensemble des fonctions booléennes, ceci explique l'unicité de représentation mais explique aussi la difficulté d'utilisation.

## 2.2 ARBRES BINAIRES DE DÉCISION

*Un arbre binaire de décision* est un arbre composé d'une racine, des nœuds internes et des feuilles valant 0 ou 1. La racine et les nœuds internes sont étiquetés par les variables $x_1, x_2, ..., x_n$ et chacun possède deux nœuds fils, un fils gauche est atteint en empruntant la branche 0 (faux) et un fils droit en empruntant la branche 1 (vrai). Un arbre binaire de décision est obtenu en appliquant récursivement le théorème de Shannon sur l'ensemble des variables de la fonction (voir section 1.2). Si la décomposition de Shannon est appliquée à une fonction booléenne $f$ pour $x_1$, puis aux deux sous-fonctions obtenues pour $x_2$, et ainsi de suite jusqu'à $x_n$, on obtient un arbre binaire de décision.

**Exemple 7.** Soit la fonction booléenne $f(a,b) = (a \wedge \neg b) \vee \neg a$. La décomposition de Shannon de $f$ sur la variable $a$ s'écrit :

$$f(a,b) = (\neg a \wedge f(0,b)) \vee (a \wedge f(1,b)) \quad \text{où} \quad f(0,b) = 1, f(1,b) = \neg b$$

La décomposition de Shannon de $f(0,b) = 1$ sur la variable $b$ est :

$$f(0,b) = (\neg b \wedge f(0,0)) \vee (b \wedge f(0,1)) \quad \text{où} \quad f(0,0) = 1, f(0,1) = 1$$

Le premier argument de $f$ représente l'assignation de la variable $a$ et le deuxième argument représente l'assignation de la variable $b$. Si la valeur de l'argument est égale à 0, on crée un fils gauche indexé par 0 sinon, on crée un fils droit indexé par 1. Dans la fonction $f(0,0) = 1$, la valeur 1 est la valeur de $f$ pour l'assignation $a = 0$ et $b = 0$ donc l'image de $f$ sera représentée par une feuille valant 1. On fait la même chose avec $f(0,1) = 1$ et on continue par la décomposition

de Shannon de $f(1,b) = \neg b$ par rapport à la variable $b$, qui donne :

$$f(1,b) = (\neg b \wedge f(1,0)) \vee (b \wedge f(1,1)) \quad \text{où} \quad f(1,0) = 1, f(1,1) = 0$$

La figure 2.3 représente l'arbre binaire de décision correspondant à la fonction $f(a,b) = (a \wedge \neg b) \vee \neg a$.

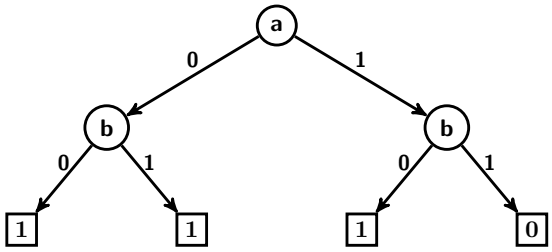



**Figure 2.3: L'arbre binaire de décision associé à la fonction : f(a,b) = (a ∧ ¬b) ∨ ¬a**

Les arbres binaires de décision constituent un modèle simple et très populaire pour représenter les fonctions booléennes. Toute fonction booléenne peut être représenter par un arbre binaire de décision, ainsi que, de nombreuses opérations utiles sur les fonctions booléennes peuvent être effectuées efficacement en temps polynomial. De plus, la représentation est unique lorsqu'on choisit un ordre sur les variables. L'inconvénient principal des arbres binaires de décision est qu'ils comportent plusieurs nœuds redondants et sous-arbres identiques, donc, ils occupent beaucoup d'espace mémoire. Nous voyons dans la section suivante comment trouver une représentation équivalente plus compacte en fusionnant des sous-arbres identiques.

## 2.3 GRAPHES BINAIRES DE DÉCISION

On peut simplifier la représentation par les arbres binaires de décision en ne conservant qu'une seule représentation des sous-arbres identiques. Tout d'abord, on peut remplacer toutes les feuilles valant 0 par un unique nœud étiqueté 0. De la même façon, toutes les feuilles valant 1 devraient pointer vers un nœud unique étiqueté par 1. En étendant cette idée, les sous-arbres identiques redondants devraient être remplacés par un seul sous-arbre. Cette manière de faire permet de concevoir des nouvelles structures appelées *diagrammes binaire de décision*.

### *2.3.1 DIAGRAMMES BINAIRES DE DÉCISION (DBD)*

[Bryant, 1986] a donné la définition et les notations utilisées pour décrire les DBD. *Un DBD* est un graphe orienté acyclique avec une racine et des nœuds internes appelés *nœuds de décisions*, et deux nœuds terminaux appelés *0-terminal* et *1-terminal*. Chaque nœud de décision $u$ est étiqueté par une variable booléenne ($u.var$) et a deux nœuds fils, appelés *fils bas ($u.bas$)* et *fils haut ($u.haut$)*. L'arête d'un nœud à un fils bas représente l'affectation de la variable à 0, et l'arête d'un nœud à un fils haut représente l'affectation de la variable à 1.

Soit $f$ une fonction booléenne définie sur $X = \{x_1, x_2, ..., x_n\}$. Le DBD correspondant à la fonction $f$ est défini par récurrence comme suit :

– Si $X = \varnothing$, le DBD est la feuille 1 ou 0 selon la valeur de $f$.
– Sinon, le DBD est un arbre binaire défini comme suit :
  – La racine est une variable $x \in X$.
  – Le fils bas (indexé par 0) est un DBD du cofacteur $f|_{x=0}$ défini sur $X \backslash \{x\}$
  – Le fils haut (indexé par 1) est un DBD du cofacteur $f|_{x=1}$ défini sur $X \backslash \{x\}$

En utilisant la deuxième forme de la décomposition de Shannon $f = (x_i \wedge f|_{x_i=1}) \vee (\neg x_i \wedge$

$f|_{x_i=0}$), on peut représenter n'importe quelle fonction booléenne $f$ sous la forme d'un DBD. L'exemple 8 explique comment construire le DBD d'une fonction booléenne $f$ en utilisant la décomposition de Shannon.

**Exemple 8.** Soit la fonction booléenne $f(a,b,c) = (\neg a \wedge b \wedge c) \vee (a \wedge c)$. Pour construire le DBD correspondant à $f$, on commence par l'application de la décomposition de Shannon sur la variable $a$ qui donne les deux sous fonctions notées $h(b,c)$ et $g(b,c)$.

$$f(a,b,c) = (a \wedge c) \vee (\neg a \wedge (b \wedge c)) \Rightarrow \begin{cases} f|_{a=1} = h(b,c) & \text{où } h(b,c) = c \\ f|_{a=0} = g(b,c) & \text{où } g(b,c) = b \wedge c \end{cases}$$

donc la racine du DBD est la variable $a$, le fils haut de $a$ indexé par 1 est la décomposition de Shannon de l'équation $h(b,c) = c$ et le fils bas de $a$ indexé par 0 est la décomposition de Shannon de $g(b,c) = b \wedge c$ (voir la figure 2.4(a)). Si on réalise la décomposition de Shannon de $g(b,c)$ sur $b$, on obtient :

$$g(b,c) = (b \wedge c) \vee (\neg b \wedge 0) \Rightarrow \begin{cases} g|_{b=1} = g'(c) & \text{où } g'(c) = c. \\ g|_{b=0} = g''(c) & \text{où } g''(c) = 0. \end{cases}$$

Le fils bas de $b$ est le nœud 0-terminal, le fils haut de $b$ est la décomposition de Shannon de $g'(c) = c$ (voir la figure 2.4(b)).

La décomposition de Shannon de $h(b,c) = c$ qui est égale aussi à la décomposition $g'(c) = c$ par rapport à $c$ donne :

$$h(b,c) = (c \wedge 1) \vee (\neg c \wedge 0) \Rightarrow \begin{cases} h|_{c=1} = 1 & \text{le fils haut de } c \text{ indexé par 1 est 1-terminal.} \\ h|_{c=0} = 0 & \text{le fils bas de } c \text{ indexé par 0 est 0-terminal.} \end{cases}$$

À la fin de la décomposition de Shannon, on obtient :

$$f(a,b,c) = [a \wedge ((c \wedge 1) \vee (\neg c \wedge 0))] \vee [\neg a \wedge ((b \wedge ((c \wedge 1) \vee (\neg c \wedge 0))) \vee (\neg b \wedge 0))].$$

On arrête la décomposition de Shannon lorsqu'on arrive à des nœuds terminaux 0 ou 1. La figure 2.4 montre les différente décompositions de Shannon effectuées sur la fonction

$f(a,b,c) = (\neg a \wedge b \wedge c) \vee (a \wedge c)$ par rapport à ces variables *a*, *b* et *c*. La figure 2.4(d) représente le DBD associé à $f$ en faisant des fusions des nœuds terminaux valant 0 (1) à un seul nœud 0-terminal (1-terminal).

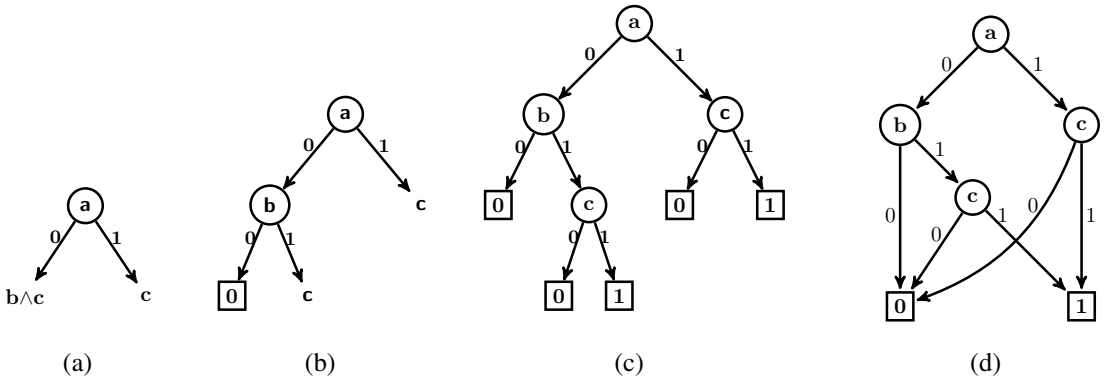



**Figure 2.4: Construction du DBD associé à la fonction f$(\mathbf{a},\mathbf{b},\mathbf{c}) = (\neg\mathbf{a} \wedge \mathbf{b} \wedge \mathbf{c}) \vee (\mathbf{a} \wedge \mathbf{c})$. (a) l'expansion de Shannon par rapport à *a*. (b) l'expansion par rapport à *b*. (c) l'expansion par rapport à *c*. (d) le DBD associé à f en remplaçant les nœuds terminaux 0 (1) par un seul nœud terminal 0-terminal (1-terminal).**

Le problème des DBD est que la représentation n'est pas unique, on peut avoir $x_j$ comme index sur un fils d'un nœud et $x_k$ sur l'autre fils avec $k \neq j$ (dans l'exemple 2.4(d) $(a.bas).var \neq (a.haut).var$). Aussi, on peut trouver des nœuds redondants (le nœud *c* est redondant). De plus, si la fonction $f$ est une tautologie, le DBD de $f$ ne sera pas forcément égal au graphe constitué uniquement de la feuille 1.

### *2.3.2 DIAGRAMMES BINAIRES DE DÉCISION ORDONNÉS (DBDO)*

Pour qu'un DBD représentant une fonction booléenne soit unique, on doit imposer un ordre d'utilisation des variables pendant la décomposition de Shannon. On peut alors définir un ordre sur les variables et à partir de cette ordre on obtient un DBD ordonné et unique.

Soit $f$ une fonction booléenne définie sur un ensemble de variables $X = \{x_1, x_2, ..., x_n\}$. On note $x_1 < x_2 < \ldots < x_n$ un ordre sur les variables de X. *Un diagramme binaire de décision ordonné*

représentant $f$ est un DBD de $f$ respectant l'ordre défini sur X. Donc, l'ordre d'occurrence des variables le long de tout chemin de la racine à un nœud terminal est cohérent avec l'ordre $\prec$. Soient $x_i$ la variable d'un nœud $v$ du DBD, $x_g$ et $x_d$ les variables des fils bas et haut respectivement de $v$, alors on doit avoir $x_i \prec x_g$ et $x_i \prec x_d$.

**Exemple 9.** La figure 2.5 représente les deux DBDO correspondants à la fonction : $f(a,b,c,d) = (a \wedge b \wedge c) \vee (\neg b \wedge d) \vee (\neg c \wedge d)$ selon les deux ordres $a < b < c < d$ et $b < c < a < d$.

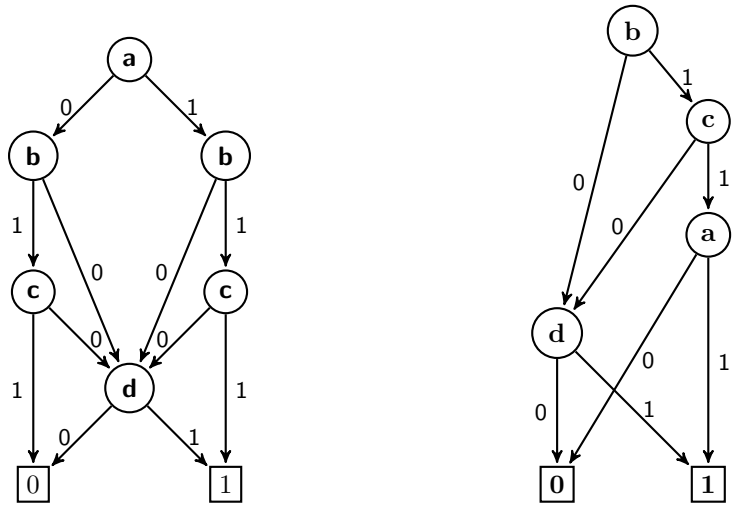



**Figure 2.5: Le DBDO représentant la fonction $\mathbf{f(a,b,c,d) = (a \wedge b \wedge c) \vee (\neg b \wedge d) \vee (\neg c \wedge d)}$. (a) le DBDO selon l'ordre $a < b < c < d$. (b) le DBDO selon l'ordre $b < c < a < d$.**

En imposant un ordre d'apparition sur les variables, les DBD permettent alors d'obtenir des implémentations très efficaces en termes de mémoire. Cependant, la compacité de la représentation d'une fonction booléenne par un DBDO dépend crucialement de l'ordre choisi. En effet, un DBD peut être très compact pour un ordre donné et être exponentiellement plus grand avec un ordre différent (dans la figure 2.5, l'ordre $b < c < a < d$ est plus performant que l'ordre $a < b < c < d$). Déterminer l'ordre optimal pour un DBD est un problème NP-complet qui, dans la pratique, se résout avec des méthodes approchées qui constituent un compromis entre le temps de recherche d'un ordre et sa compacité [Rudell, 1993].

### *2.3.3 DIAGRAMMES BINAIRES DE DÉCISION ORDONNÉS ET RÉDUITS (DBDOR)*

En revenant à la figure 2.4(d), on peut remarquer que le DBD comporte des nœuds redondants (des sous-graphes isomorphes). En effet, les deux sous-graphes enracinés par *c* sont isomorphes. On peut alors simplifier le DBD, en fusionnant les deux sous-graphes (on supprime le fils haut de *a* et on le remplace par un arc partant de *a* vers le fils haut de *b* qui est le sous-graphe enraciné par *c*).

En fusionnant tous les sous-graphes isomorphes et en supprimant les nœuds inutiles d'un DBDO, on obtient un *diagramme binaire de décision ordonné et réduit (DBDOR)*. Les *DBDOR* sont basés sur un ordre fixé des variables et ont la propriété supplémentaire d'être réduits. Cela signifie que les successeurs bas et hauts de chaque sommet sont distincts, et il n'existe pas deux sommets distincts testant la même variable ayant les mêmes successeurs. Alors, un DBDOR est un DBDO qui ne contient aucun sommet *u* tel que ses deux fils soient égaux ($u.bas \neq u.haut$. Si $u.bas = u.haut$ on élimine *u* et on redirige tous les arcs y entrants vers $u.bas$), et aucune paire de sommets distincts *u* et *v* tels que les sous-graphes enracinés par *u* et *v* soient isomorphes (si $u.var = v.var$, $u.bas = v.bas$ et $u.haut = v.haut$ alors $u = v$, on élimine l'un des deux sommets et on redirige tous les arcs entrants au sommet éliminé vers l'autre sommet).

Un algorithme de réduction d'un DBD a été proposé par Bryant [Bryant, 1986]. Il a pour objectif d'éliminer les nœuds inutiles et les sous-graphes isomorphes, et transforme le DBD représentant une fonction en un graphe réduit (voir l'algorithme 1).

Partant des nœuds terminaux vers la racine, une étiquette (*id*) et une clé (*clé*) sont attribue à chaque sommet *v*. Un sommet non terminal *v* doit avoir un $id(v)$ égal à celui d'un sommet qui a été déjà marqué si et seulement si l'une des deux conditions suivantes est remplie :

**Algorithme 1** Réduction du DBD

```
1:  fonction REDUCE(v : sommet)
2:      Q ← liste vide ; idSuivant ← 1
3:      u ← 0-terminal; u.id ← 0; clé ← 0;
4:      Ajouter u au DBDOR ; Ajouter <clé,u> à Q ;
5:      u ← 1-terminal; u.id ← 1; clé ← 1;
6:      Ajouter u au DBDOR ; Ajouter <clé,u> à Q ;
7:      pour i ← n − 1, 1 faire                          ▷ n est le nombre de niveaux
8:          l_i ← les sommets du niveau i
9:          pour tout u de l_i faire
10:             si u.bas.id = u.haut.id alors            ▷ Sommet inutile
11:                 u.id ← u.bas.id
12:             sinon
13:                 clé ← (u.bas.id, u.haut.id)
14:                 Ajouter <clé,u> à Q ;
15:             fin si
16:         fin pour
17:         Trier les éléments de Q ;
18:         cléAncien ← (−1,−1)
19:         pour tout <clé,u> de Q faire
20:             si clé = cléAncien alors                 ▷ Sommet existe déjà
21:                 u.id ← idSuivant
22:             sinon                                    ▷ Nouveau sommet
23:                 idSuivant ← idSuivant + 1
24:                 u.id ← idSuivant
25:                 u.bas ← le sous graphe enraciné par u.bas.id
26:                 u.haut ← le sous graphe enraciné par u.haut.id
27:                 cléAncien ← clé
28:                 v ← u
29:                 Ajouter v au DBDOR ;
30:             fin si
31:         fin pour
32:     fin pour
33:     retourner v                                      ▷ La racine du DBDOR
34: fin fonction
```

– si $v.bas.id = v.haut.id$, alors $v$ est inutile, on doit fixer $id(v) = id(bas(v))$.

– s'il y a un sommet $u$ étiqueté par $id(u) = k$, et $v.bas.id = u.bas.id$, et $v.haut.id = u.haut.id$, alors les sous-graphes enracinées par ces deux sommets sont isomorphes, on met $id(v) = k$.

On commence par les sommets terminaux, on assigne $id(v) = 0$ aux nœuds terminaux valant 0, et $id(v) = 1$ aux nœuds terminaux valant 1. On initialise le DBDOR par un nœud 0-terminal ($id = 0$, $clé = 0$) et un nœud 1-terminal ($id = 1$, $clé = 1$), on ajoute les tuples $<clé, sommet>$ à une liste $Q$. On continue récursivement avec les nœuds non terminaux de bas vers haut. Pour chaque niveau $i$ on construit une liste $l_i$ de sommets. Puis, pour chaque sommet $v$ de la liste, on associe une $clé$ de la forme $<v.bas.id, v.haut.id>$ et une étiquette $id$. Si $v.bas.id = v.haut.id$ alors le sommet est inutile, on met $id(v) = id(v.bas)$. Si $v.bas.id \neq v.haut.id$ on parcourt la liste $Q$ de gauche à droit, si on trouve un sommet $u$ ayant la même $clé$ que $v$, on met $id(v) = id(u)$ sinon, on met $id(v) = idSuivant$ et on ajoute le tuple $<clé, sommet>$ à la liste $Q$. On ajoute $v$ au DBDOR avec des arcs allant aux sommets dont l'$id$ est égale à ceux de $v.bas$ et $v.haut$ (voir l'algorithme 1).

**Exemple 10.** La figure 2.6 représente les différentes étapes de l'application de l'algorithme de réduction sur le DBD représentant la fonction $f(a,b,c) = (a \vee b) \wedge c$.

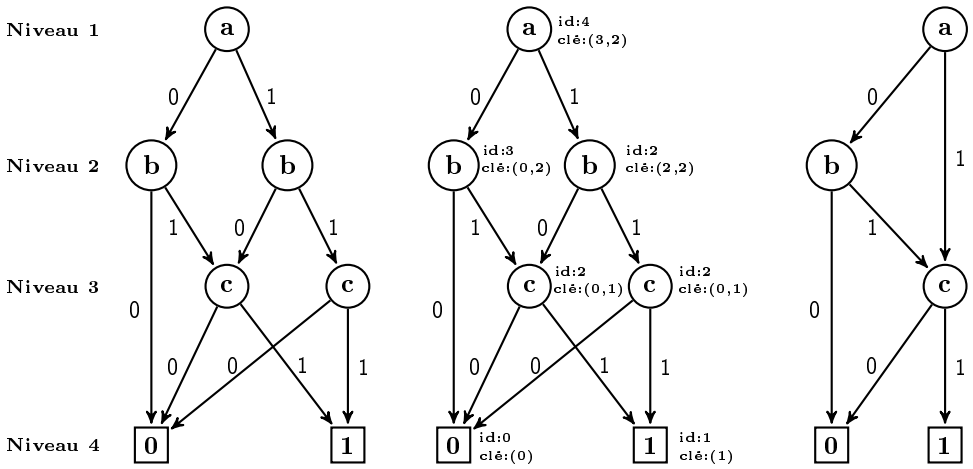



**Figure 2.6: Le DBDOR représentant la fonction $\mathbf{f(a,b,c) = (a \vee b) \wedge c}$.**

Les sous-graphes enracinés par $c$ sont isomorphes, ils ont la même clé, donc on élimine un sommet et on redirige les arcs entrant à ce sommet vers l'autre sommet. Le sous-graphe

enraciné par $b$ (fils haut de $a$) est inutile car les étiquettes de ses fils bas et haut sont égaux ; on redirige les arcs entrant à $b$ vers son fils bas.

**Théorème 1.** *Soit G le graphe de n sommets représentant le DBD D, la réduction de ce DBD à un DBDOR a une complexité* $O(n\log(n))$.

**Démonstration :** Le traitement à chaque niveau nécessite un temps proportionnel au nombre de sommets à ce niveau. Chaque niveau est traité une fois, donc la complexité globale de l'algorithme est $O(n\log(n))$ en nombre de nœuds.

**Théorème 2** ( [Bryant, 1986])**.** *Soit* $f$ *une fonction booléenne définie sur X muni d'un ordre* $<$*. Il existe un unique DBDOR qui représente* $f$*. De plus, chaque sous-graphe de ce DBDOR est réduit.*

La démonstration du théorème 2 est dans l'article [Bryant, 1986], elle est omise car elle est longue et elle ne fait pas l'objet du présent mémoire.

Ce théorème garantit l'unicité du DBDOR d'une fonction. Par conséquent, il permet notamment de pouvoir tester l'équivalence entre deux formules simplement par comparaison des diagrammes correspondants. Le test *SAT* est aussi très simple, la formule représentée par un DBDOR est satisfaisable si et seulement si ce DBDOR n'est pas la feuille 0. De plus, les DBDOR permettent d'avoir des algorithmes polynomiaux en la taille du graphe pour les opérations telles que la conjonction ou la disjonction de deux diagrammes. La négation peut s'effectuer en temps linéaire en inversant les deux feuilles 0 et 1 (certaines implémentations permettent même la négation en temps constant).

## 2.4 OPÉRATIONS SUR LES DIAGRAMMES BINAIRES DE DÉCISION

Plusieurs opérations liées aux fonctions booléennes telles que la satisfaisabilité, la vérification de la tautologie ou l'équivalence de deux expressions booléennes et l'application des opérations logiques entres les fonctionnes booléennes peuvent être vérifiées par les DBD. Dans ce qui suit, on présente la résolution des opérations les plus intéressantes des fonctions booléennes par les DBD.

### *2.4.1 ÉVALUATION*

Soit $f$ une fonction booléenne sur $X = \{x_1, x_2, ..., x_n\}$ et une assignation $a : X \to \{vrai, faux\}$ des variables de $X$. Le problème d'évaluation sert à déterminer la valeur de la fonction booléenne $f(x_1, x_2, ..., x_n)$ pour l'assignation $a$ des variables $x_i, i \in [1, n]$.

**Théorème 3.** *L'évaluation d'une assignation par un DBD s'effectue en* $O(n)$.

**Démonstration :** La valeur de la fonction $f$ pour une assignation des variables d'entrées peut être trouvée en traversant le DBD depuis la racine jusqu'à une feuille en empruntant la branche gauche ou droite de chaque sommet selon la valeur de la variable d'entrée correspondante. Dans le pire cas, le DBD correspondant à une fonction $f$ est composé de $n$ branches entre la racine et une feuille. Donc, la complexité d'une procédure d'évaluation d'une fonction, c'est à dire le parcours d'un DBD depuis sa racine est $O(n)$.

### *2.4.2 TAUTOLOGIE*

On dit qu'une fonction booléenne est une tautologie si elle est toujours vraie pour toute assignation, c'est-à-dire vraie quelle que soit les valeurs de variables. On dit autrement, la

table de vérité de cette fonction prend toujours la valeur vrai $(1)$. On peut poser le problème de tautologie d'autre manière : Une fonction booléenne $f$ est une tautologie si et seulement si la fonction $f$ est la fonction identiquement égale à 1.

**Théorème 4.** *La vérification de la tautologie par un DBD est d'ordre* $O(1)$.

**Démonstration :** Pour montrer que la fonction $f$ est une tautologie, il suffit de vérifier si la racine du DBD correspondant à $f$ est la feuille 1 ou non. Il est évident que cela a le coût $O(1)$.

### *2.4.3 ÉGALITÉ*

L'un des plus grand problèmes des fonctions booléennes est de déterminer si deux fonctions booléennes sont égales, surtout, parce que une fonction booléenne peut avoir plusieurs représentations différentes. Ce problème peut paraître simple, mais deux fonctions booléennes ayant des représentations complètement différentes peuvent être égales. Ce problème n'a pas lieu d'être si chaque fonction booléenne $f$ a une unique représentation dans un système de représentation. Donc, pour vérifier l'égalité de deux fonctions, il suffit de vérifier l'égalité de ses représentations.

**Théorème 5.** *La vérification de l'égalité de deux fonctions booléennes par les DBD est d'ordre* $O(n)$.

**Démonstration :** D'après le théorème d'unicité du DBDOR d'une telle fonction (voir théorème 2), pour tester l'égalité de deux fonctions booléennes $f$ et $g$, il suffit de comparer les DBD qui les représentent. Soit $n$ la taille du graphe représentant $f$ et $m$ la taille du graphe représentant $g$, la complexité de l'opération d'égalité dans le pire cas est égale au minimum de $n$ et $m$, donc elle est d'ordre $O(n)$.

### *2.4.4 SATISFAISABILITÉ*

Le problème de SAT a été discuté dans le premier chapitre (voir section 1.2 pour plus de détails). On peut poser le problème SAT de cette manière. Soient $f(x_1, x_2, ..., x_n)$ une fonction booléenne de $n$ variables, $a$ une assignation des variables, et $S_f = \{a \in \mathbb{B}^n | f(a) = 1\}$ l'ensemble des assignations qui rendent $f$ vraie. On peut définir trois types de problèmes de satisfaisabilité :

- **Satisfaisabilité** : On dit que $f$ est satisfaisable si $S_f \neq \varnothing$, c'est-à-dire, elle n'est pas équivalente à la fonction booléenne constante 0.
- **Satisfait-Un** : Pour une assignation $a$, Satisfait-Un consiste à déterminer si $a \in S_f$.
- **Satisfait-Tous** : Satisfait-Tous consiste à calculer $S_f$.

**Théorème 6.** *La complexité des opérations de vérification de la satisfaisabilité par les DBD est d'ordre* $O(1)$*, de Satisfait-Un est d'ordre* $O(n)$ *et de Satisfait-Tous est d'ordre* $O(n.|S_f|)$.

**Démonstration :** Pour montrer que la fonction $f$ est satisfaisable, il suffit de vérifier si la racine du DBD correspondant à $f$ est la feuille 0 ou non. Il est évident que cela a le coût $O(1)$. Pour montrer Satisfait-Un, il suffit de partir de la feuille 1 du DBD correspondant à $f$ et de remonter un chemin jusqu'à la racine. La complexité est en $O(n)$. Pour montrer Satisfait-Tous, Il suffit de réitérer autant de fois que nécessaire le Satisfy-Un. Le coût de cette procédure est d'ordre $O(n.|S_f|)$.

### *2.4.5 APPLICATION D'UN OPÉRATEUR LOGIQUE*

L'algorithme permettant de construire le DBD résultant de l'application d'une opération logique entre deux fonctions se trouve également dans l'article de Bryant (voir la fonction APPLY 2). Les opérations applicables sur les DBD sont les opérations de test d'implication,

de complémentation, de ET logique, de OU logique et de OU exclusif.

Étant données deux fonctions booléennes $f$ et $g$, un ordre *ord* sur les variables de $f$ et $g$, et un opérateur logique binaire $<op> \in (\wedge, \vee, \rightarrow, \ldots)$. Le DBD représentant la fonction $h = f<op>g$ est basé sur l'application récursive de la formule suivante dérivée de l'expansion de Shannon.

$$f < op > g = (\neg x_i \wedge (f|_{x_i=0} < op > g|_{x_i=0})) \vee (x_i \wedge (f|_{x_i=1} < op > g|_{x_i=1})).$$

On commence à appliquer la fonction APPLY sur les racines des deux DBD, et on itère récursivement le processus jusqu'à la génération des sommets terminaux en respectant l'ordre des variables et en tenant compte des différents cas possibles. Supposant que $u$ est la racine du DBD représentant $f$ et $v$ la racine du DBD représentant $g$. Si $u$ et $v$ sont des sommets terminaux alors le DBD résultant est composé d'un seul sommet terminal ayant une valeur égale à $u.var <op> v.var$. Sinon, si $u.var = v.var$, on crée un sommet $w$ avec $w.var = u.var$ et on applique la fonction APPLY sur les fils bas de $u$ et $v$ pour obtenir le fils bas de $w$, et on fait la même chose pour le fils haut de $u$ et $v$ pour avoir le fils haut de $w$. Si $u.var \neq v.var$, $w$ prend la valeur du sommet qui a l'ordre minimal. Supposant que $u$ qui a l'ordre minimal, donc $w.var = u.var$, le fils bas de $w$ est obtenu par l'application de APPLY sur $u.bas$ et $v$, et le fils haut de $w$ est obtenu par l'application de APPLY sur $u.haut$ et $v$. Si l'une des racines est terminale et l'autre non terminale, on suppose que $u$ n'est pas terminal, dans ce cas, la valeur de la racine du DBD résultant $w.var = u.var$ et les fils bas et haut de $w$ sont obtenus par l'application de la fonction APPLY sur les fils de $u$ et la valeur du sommet terminal $v$ qui est 0 ou 1. Et enfin, après la construction du DBD final, on applique la fonction de réduction sur le graphe obtenu car il ne l'est pas.

**Théorème 7.** *Soient $f$ et $g$ deux fonctions booléennes, $D_1$ le DBD représentant $f$ et $D_2$ le DBD représentant $g$. $n$ est la taille de $D_1$ et $m$ est la taille de $D_2$. La complexité de la procédure*

**Algorithme 2** Application d'un opérateur

```
1:  fonction APPLY(u, v : sommet ; <op>: oprateur)
2:      si (u.var = 0 or u.var = 1) and (v.var = 0 or v.var = 1) alors
3:          w.var ← u.var <op> v.var
4:      sinon
5:          si (u.var = 0 or u.var = 1) and (v.var ≠ 0 and v.var ≠ 1) alors
6:              w.var ← v.var ;
7:              w.bas ← APPLY(u, v.bas, <op>) ;
8:              w.haut ← APPLY(u, v.haut, <op>) ;
9:          sinon
10:             si (u.var ≠ 0 and u.var ≠ 1) and (v.var = 0 or v.var = 1) alors
11:                 w.var ← u.var ;
12:                 w.bas ← APPLY(v, u.bas, <op>) ;
13:                 w.haut ← APPLY(v, u.haut, <op>) ;
14:             sinon
15:                 si (u.var = v.var) alors
16:                     w.var ← u.var
17:                     w.bas ← APPLY(u.bas, v.bas, <op>) ;
18:                     w.haut ← APPLY(u.haut, v.haut, <op>) ;
19:                 sinon
20:                     w.var ← (min(u.ord, v.ord)).var ;
21:                     si (u.ord < v.ord) alors
22:                         w.bas ← APPLY(u.bas, v, <op>) ;
23:                         w.haut ← APPLY(u.haut, v, <op>) ;
24:                     sinon
25:                         w.bas ← APPLY(u, v.bas, <op>) ;
26:                         w.haut ← APPLY(u, v.haut, <op>) ;
27:                     fin si
28:                 fin si
29:             fin si
30:         fin si
31:     fin si
32:     retourner REDUCE(w) ;                          ▷ La racine du DBD
33: fin fonction
```

*d'application d'un opérateur entre $D_1$ et $D_2$ est d'ordre $O(nm)$.*

**Démonstration :** La fonction APPLY génère la première fois dans le pire cas $2n$ appels récursifs pour $f$ et $2m$ appels récursifs pour $g$. Pour chaque appel, les différentes opérations

s'effectuent en temps constant. Par conséquent, la complexité totale de l'algorithme est $O(nm)$.

Il existe d'autres opérations qui peuvent être effectuées sur les DBD en temps polynomial telles que la négation (qui peut s'effectuer en temps constant en inversant les deux feuilles 0 et 1), le calcul du cofacteur par rapport à une variable $x_i$ ($\neg x_i$) (obtenu par la suppression du sommet $x_i$ ($\neg x_i$) et en liant les sommets pointant sur $x_i$ au fils bas (haut) de $x_i$), le test d'inclusion d'un monôme dans la fonction, etc.

## 2.5 ORDRE DES VARIABLES ET TAILLE DES DIAGRAMMES BINAIRES DE DÉCISION ORDONNÉS

En observant la complexité des différents opérations sur les DBD, on voit que la complexité de la plupart de ces opérations est très peu élevée et donc il peut paraître intéressant d'utiliser les DBD. Cependant, toutes ces complexités sont en fonction de la taille du graphe (nombre de sommets du DBD). Si la taille du DBD est exponentielle par rapport à l'ensemble de variables ($k = |X|$) de la fonction booléenne, on se retrouve à nouveau avec des complexités plus élevées. On a vu précédemment que la taille du graphe dépend grandement de l'ordre des variables. Il est donc intéressant d'étudier le problème de déterminer l'ordre optimal qui minimise la taille du DBDO. Malheureusement, il a été démontré que ce problème est NP-Complet [Bollig et Wegener, 1996]. En pratique, on peut rechercher l'ordre optimal jusqu'à 20 variables. Cependant, si on a plus de variables, on utilise des heuristiques pour trouver l'ordre qui minimise le mieux la taille du graphe. Par conséquent, on ne teste pas tous les ordres possibles ($k!$ ordres possibles). Pour déterminer l'ordre optimal, on peut distinguer deux grandes familles de techniques [Rudell, 1993] :

– Le DBDOR est déjà construit. Cette famille part du principe que le DBDOR a été déjà construit, et donc qu'il existe déjà un ordre sur celui-ci. On essaye d'améliorer dynamique-

ment cet ordre en le modifiant légèrement pour réduire la taille.

– Le DBDOR n'est pas encore construit. Cette famille d'algorithmes permet de rechercher l'ordre optimal d'une fonction booléenne avant de construire le DBDOR. On utilise pour cela des heuristiques qui essaient de prédire le meilleur ordre pour la fonction booléenne étudiée. Certaines heuristiques réalisent de bons résultats sur des familles particulières de fonctions booléennes. Mais il n'existe pas d'heuristique ayant de bons résultats dans le cadre général.

## 2.6 EXTENSIONS ET VARIANTES DES DIAGRAMMES BINAIRES DE DÉCISION

Dans la littérature, il y a plusieurs extensions aux DBD qui ont été proposées dans le but de réduire la taille de ces structures ou pour améliorer l'efficacité des algorithmes liés à ceux-ci dans certains contextes. Parmi les extensions les plus populaires, on trouve les suivantes.

Les diagrammes de décision binaire ordonnés et partitionnés (DBDOP) sont particulièrement utilisés pour représenter une formule booléenne avec plusieurs DBDO ayant des ordres de variables différents. Pour ce faire, la fonction à représenter est décomposée en sous-ensembles. Chaque DBDO utilisé représente une fonction $g_i$ égale à la fonction $f$ sur le i-ième sous-ensemble (pour plus de détails sur les DBDOP, consulter [Narayan et al., 1996]).

Les diagrammes de moments binaires (DMB) sont utilisés pour représenter les fonctions qui ne sont pas composées uniquement des 0 et 1. Plus précisément, ils sont utilisés pour représenter les polynômes (pour plus de détails voir [Bryant et Chen, 1995]).

Les diagrammes binaires de décision à terminaux multiples (DBDTM, parfois nommés diagrammes de décision algébrique DDA) sont une généralisation des DBD dans laquelle

des valeurs différentes de 0 ou 1 sont permises pour les feuilles. Contrairement aux autres variantes que nous présentons, les DBDTM sont très populaires et beaucoup utilisés (pour plus de détail sur les DBDTM, voir [Bahar et al., 1993] et [Fujita et al., 1997]).

Les diagrammes binaires de décision sans zéro (ou zéro-supprimés DBDZ) sont une autre structure réduite des diagrammes binaires de décision de telle sorte que, les nœuds dont l'arc indexé par 1 pointe directement sur le nœud 0-terminal sont enlevés [Minato, 1993, Minato, 1996]. Dans la figure 2.7(b) les nœuds $a$ et $c$ sont supprimés parce qu'ils pointent sur le nœud 0-terminal par 1. Les DBDZ sont plus concis que les DBD pour certaines applications qui utilisent les ensembles de combinaison. Un ensemble de combinaison est un ensemble de n-tuples de bits. Par exemple l'ensemble $S = \{001, 010\}$ est un ensemble de combinaison. L'exemple 11 explique comment représenter cet ensemble par un DBD et un DBDZ.

**Exemple 11.** Pour représenter l'ensemble $S = \{001, 010\}$, on commence par la définition de la fonction booléenne lui correspondant. Chaque élément de $S$ est composé de 3 chiffres, donc, on a besoin de 3 variables $\{a, b, c\}$. Pour chaque chiffre, si sa valeur est égale à 0 on prend la négation de la variable, sinon on prend la variable elle même. La fonction $f$ à représenter est celle-ci :

$$f(a,b,c) = (\neg a \wedge \neg b \wedge c) \vee (\neg a \wedge b \wedge \neg c).$$

La figure 2.7 montre la représentation de $f$ par un DBD et par un DBDZ selon l'ordre $x_1 < x_2 < x_3$.

On trouve dans la littérature d'autres variantes des DBD qui sont moins utilisées comme les diagrammes binaires de décision à arcs valués (DBDAV), les diagrammes avec nœuds de décision (DBDND), etc.

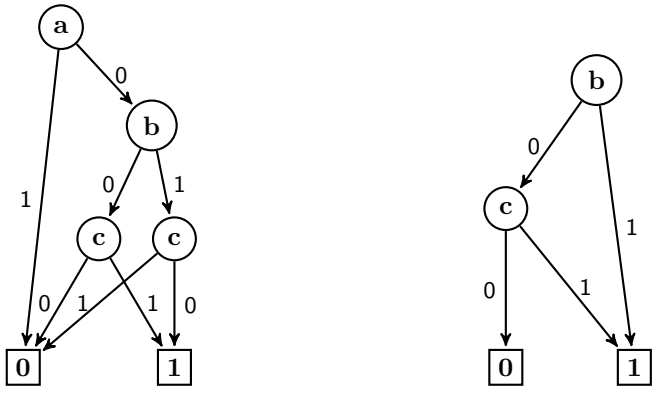



**Figure 2.7: La représentation de $f$ par un DBD et un DBDZ. (a) le DBD représentant $f$. (b) le DBDZ représentant $f$.**

## 2.7 CONCLUSION

Les DBD et ces variantes constituent des structures de données intéressantes dans plusieurs domaines de l'informatique. Ils sont utilisés en particulier pour la vérification et de l'apprentissage automatique en intelligence artificielle et pour enregistrer efficacement des ensembles de combinaisons. Ils offrent des modèles de représentation très compacts et disposent d'algorithmes efficaces pour plusieurs opérations et problèmes combinatoires réductibles à l'un des problèmes liés aux fonctions booléennes. Cependant, l'efficacité de ces opérations et algorithmes dépend crucialement de la taille de ces structures qui, dans certains cas, nécessitent une représentation de taille exponentielle.

# CHAPITRE 3

# ARBRES ET/OU

Plusieurs algorithmes d'énumération qui reposent sur la technique classique de séparation et évaluation utilisent les DBD ou les DBDZ pour stocker les solutions combinatoires. Toutefois, ces structures ne sont pas toujours pratiques, surtout quand le branchement n'est pas binaire (plus de deux cas possibles à un branchement). Dans ce chapitre, nous introduisons une classe particulière d'arbres permettant l'énumération et l'enregistrement de manière compacte de larges familles d'ensembles de données. Nous allons donner son implémentation et étudier son efficacité dans le contexte d'énumération par la conception d'un algorithme résolvant le problème TCCM en utilisant cette structure.

Dans un premier temps, les arbres et/ou ont été introduits pour représenter graphiquement la réduction d'un problème à des conjonctions et disjonctions de sous-problèmes. Chaque nœud d'un arbre et/ou représente un sous-problème ; en particulier, le nœud racine représente le problème initial à résoudre. Les nœuds ayant des nœuds enfants (ou successeurs) sont appelés *nœuds non-terminaux*. Chaque nœud non terminal est soit de type *et*, soit de type *ou*. Une solution d'un problème dont le nœud correspondant est de type *ou* est obtenue en récupérant la solution de l'un de ses successeurs, tandis qu'une solution à un problème dont le nœud sous-jacent est de type *et* est obtenue en récupérant les solutions de tous ses nœuds successeurs.

Les nœuds sans successeurs sont appelés *terminaux* [Kumar et N. Kanal, 1983]. La figure 3.1 représente l'arbre et/ou représentant la décomposition du problème $P$ en sous-problèmes. Le nœud $P$ est de type *ou* et le nœud $Q$ est de type *et*, les nœuds $P$ et $Q$ sont des nœuds non terminaux, les nœuds $R, S, T, U$ sont des nœuds terminaux. La résolution du $P$ est obtenue en récupérant la solution de $Q$ *ou* $R$ *ou* $S$. La résolution de $Q$ est obtenue en récupérant la solution de $T$ *et* de $U$. Le problème $P$ se formule par $P = (Q \vee R \vee S)$ avec $Q = (T \wedge U)$ donc $P = ((T \wedge U) \vee R \vee S)$.

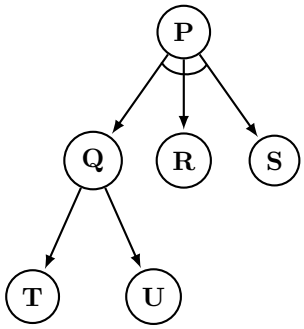



**Figure 3.1: Décomposition d'un problème $P$ par un arbre et/ou.**

Plus tard, les arbres et/ou ont été utilisés comme une stratégie de recherche dans les espaces d'états et les espaces de recherche des jeu à deux personnes. Dans l'arbre d'un jeux à deux joueurs, les nœuds *ou* représentent les mouvements d'un joueur, où il peut choisir lequel des mouvements à prendre et les nœuds *et* représentent les mouvements de l'adversaire [Kumar et N. Kanal, 1983]. Les arbres et/ou sont également utilisés pour l'optimisation combinatoire dans les modèles graphiques [Marinescu et Dechter, 2009, Joo et al., 2014] et pour optimiser l'espace des contraintes dans les problèmes de satisfaction [Marinescu et Dechter, 2008].

Dans le reste de ce chapitre, nous présentons la structure de base de la structure arbre et/ou que nous avons conçue, et nous discutons les différents algorithmes de manipulation de cette structure et les différentes opérations, requêtes et modifications qui peuvent être effectuées sur

cette structure dans un contexte d'énumération.

## 3.1 STRUCTURE DE BASE

Comme on a déjà vu, un arbre et/ou est un arbre dont lequel les nœuds internes sont des opérateurs logiques *et* et *ou* ($\wedge$, $\vee$) et les feuilles sont des valeurs de l'ensemble de solution. Pour chaque nœud interne de type *ou*, la solution est obtenue en résolvant l'un de ses successeurs, tandis que la solution d'un nœud de type *et* est obtenue en résolvant tous ses successeurs.

**Exemple 12.** Soit une famille d'ensembles $\mathcal{F} = \{\{0,1\},\{0,3\},\{1,2\}\}$. La figure 3.2 montre la représentation de cette famille par un DBD, DBDZ et un arbre et/ou selon l'ordre $0 < 1 < 2 < 3$.

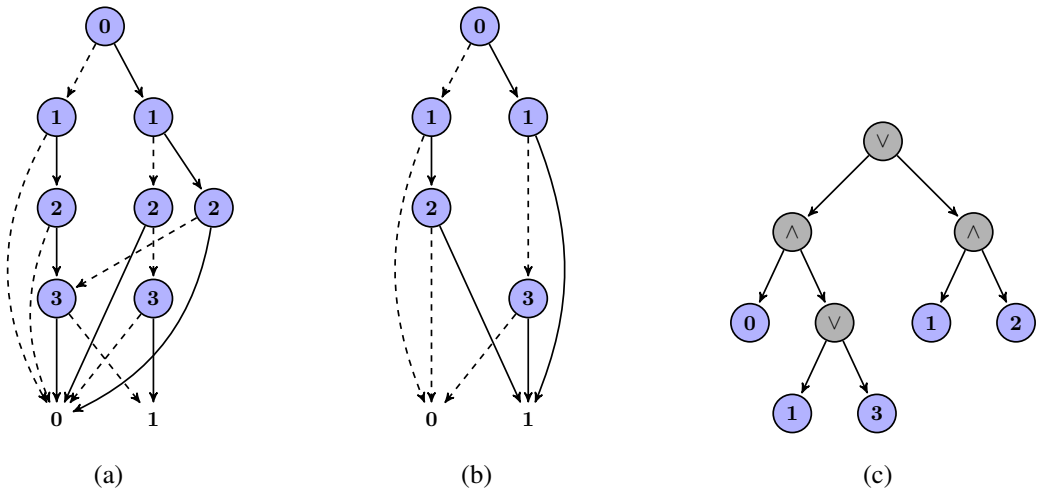



**Figure 3.2: Représentation de la famille $\mathcal{F}$ ={{0,1},{0,3},{1,2}} par DBD, DBDZ et arbre et/ou. (a) Représentation de $\mathcal{F}$ par un DBD. (b) Représentation de $\mathcal{F}$ par un DBDZ. (c) Représentation de $\mathcal{F}$ par un arbre et/ou.**

Dans 3.2(a) et 3.2(b), les arcs pointillés sortant d'un nœud étiqueté par la variable $v$ signifient que la variable $v$ prend la valeur 0 et donc qu'elle est exclue de l'ensemble de solution. Les arcs pleins signifient que la variable $v$ prend la valeur 1 et donc elle est incluse dans l'ensemble de solution. Un ensemble $E$ appartient à $\mathcal{F}$ s'il existe un chemin partant de la racine vers la

feuille 1 en traversant les arcs pointillés ou les arcs pleins. Dans 3.2(c), pour calculer la famille des ensembles, on part de la racine vers les feuilles, si le nœud est étiqueté par $\vee$, on récupère la solution de l'un de ses enfants, s'il est étiqueté par $\wedge$, on récupère la solution de tous ses enfants, si le nœud est une feuille, on l'ajoute à l'ensemble de solution courant.

**Définition 7.** *Soit S un ensemble fini, T un arbre et V l'ensemble des nœuds de l'arbre T. Chaque ensemble $\mathcal{F} \subseteq 2^S$ est appelé* famille *dans S, c'est-à-dire $\mathcal{F}$ est une collection de sous-ensembles de S. Généralement, $|S|$ est petite, mais $|\mathcal{F}|$ pourrait être très grande. L'arbre T est un arbre et/ou s'il existe trois fonctions* CONTENT *: $V \to S \cup \{\wedge, \vee\}$,* VALUES *: $V \to 2^S$ et* FAMILY *: $V \to 2^{2^S}$ de telle sorte que les conditions suivantes soient vérifiées.*

1. *Chaque feuille v a un contenu dénoté par* $\textsc{Content}(v) \in S$ *;*
2. *Pour chaque nœud interne v,* $\textsc{Content}(v) = \wedge$ *ou* $\textsc{Content}(v) = \vee$ *;*
3. *Pour chaque nœud* $v \in V$*,*
$$\textsc{Values}(v) = \begin{cases} \{\textsc{Content}(v)\} & \text{si } v \text{ est une feuille ;} \\ \bigcup_{w \in v.fils} \textsc{Values}(w) & \text{si } v \text{ est un nœud interne.} \end{cases}$$
4. *Pour chaque nœud* $v \in V$*,*
$$\textsc{Family}(v) = \begin{cases} \{\{\textsc{Content}(v)\}\} & \text{si } v \text{ est une feuille ;} \\ \bigcup_{w \in v.fils} \textsc{Family}(w) & \text{si } \textsc{Content}(v) = \vee \text{ ;} \\ \prod_{w \in v.fils} \textsc{Family}(w) & \text{si } \textsc{Content}(v) = \wedge. \end{cases}$$
*où le produit $\prod$ de deux familles $\mathcal{F}$ et $\mathcal{G}$ est défini par $\mathcal{F} \cdot \mathcal{G} = \{S \cup T | S \in \mathcal{F}, T \in \mathcal{G}\}$.*
5. *Pour tout noeud v tel que* $\textsc{Content}(v) = \vee$ *et w, $w' \in v.fils$, la condition* $\textsc{Family}(w) \cap \textsc{Family}(w') \neq \varnothing$ *implique* $w = w'$*.*
6. *Pour tout noeud v tel que* $\textsc{Content}(v) = \wedge$ *et w, $w' \in v.fils$, la condition* $\textsc{Values}(w) \cap \textsc{Values}(w') \neq \varnothing$ *implique* $w = w'$*.*

Le point (1) est pour dénoter la valeur (clé) des nœuds feuilles. Le point (2) pour dénoter la valeur (clé) des nœuds internes $(\wedge, \vee)$. Le point (3) indique que $\textsc{Values}(v)$ est l'ensemble

de toutes les valeurs qui apparaissent dans au moins une feuille du sous-arbre induit par $v$. Le point (4) indique comment calculer récursivement, pour chaque nœud $v$, la famille des ensembles de toutes les solutions représentées par le sous-arbre dont la racine est $v$. Finalement, les conditions (5) et (6) imposent certaines restrictions sur le type d'arbre qui peut être manipulé. Dans le point (5), on veut s'assurer que tout ensemble apparaissant dans une branche ne peut apparaître dans une autre (par conséquent, chaque ensemble apparaît une seule fois dans l'arbre). Quant au point (6), il permet d'optimiser l'utilisation des nœuds de contenu $\wedge$ puisqu'il interdit le fait qu'une clé puisse apparaître dans différentes branches.

**Exemple 13.** Soit la même famille $\mathcal{F} = \{\{0,1\},\{0,3\},\{1,2\}\}$ de l'exemple 12. L'arbre et/ou de la figure 3.3 fournit une représentation de CONTENT, VALUES et FAMILY de $\mathcal{F}$ . Pour des raisons de lisibilité, nous utilisons souvent le dessin correspondant à CONTENT uniquement.

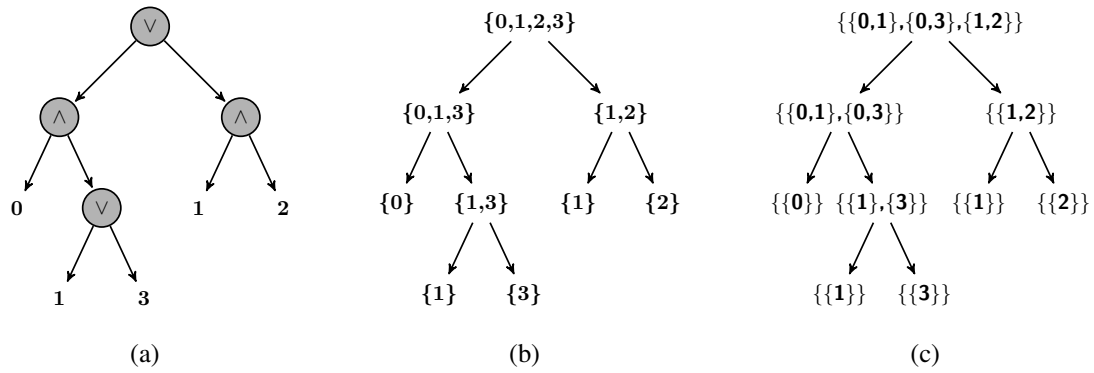



**Figure 3.3: Arbres représentant CONTENT, VALUES et FAMILY de la famille $\mathcal{F} = \{\{0,1\},\{0,3\},\{1,2\}\}$. (a) L'arbre et/ou représentant la famille $\mathcal{F}$ , Pour chaque nœud $v$, on écrit la valeur CONTENT$(v)$ à l'intérieur du nœud. (b) Le même arbre et/ou avec VALUES$(v)$ à la place du CONTENT$(v)$ pour chaque nœud $v$. (c) Le même arbre et/ou avec FAMILY$(v)$ à la place du CONTENT$(v)$ pour chaque nœud $v$.**

**Définition 8.** *Un arbre et/ou $T$ est appelé* k-uniforme *si $|E| = k$ pour chaque $E \in$ FAMILY$(r)$, où $r$ est la racine de $T$. Si $T$ est k-uniforme, l'ordre de $T$, noté* ORDER$(T)$*, est le nombre $k$.*

Dans de nombreuses situations, on se concentre à l'étude des familles d'ensembles ayant une taille constante. Une famille $\mathcal{F}$ est appelé *k-uniforme* (ou simplement uniforme si la valeur de

$k$ est sans importance) si $|E| = k$ pour chaque $E \in \mathcal{F}$ . L'arbre de la figure 3.3 représente une famille 2-uniforme.

## 3.2 OPÉRATIONS SUR LES ARBRES ET/OU

La plupart des opérations de base et les opérations avancées appliquées sur les arbres et/ou peuvent être définies de manière récursive par un parcours en profondeur de la racine vers les feuilles. Pour chaque nœud de l'arbre actuellement visité, les actions appliquées se diffèrent selon le type du nœud et l'opération à effectuer. Dans ce qui suit, nous présentons quelques uns des requêtes et modificateurs les plus intéressants effectués sur les arbres et/ou.

### *3.2.1 REQUÊTES*

En général, les requêtes sont des opérations utilisées pour récupérer des informations à partir d'une structure de données sans la modifier. Nous ne pouvons pas énumérer toutes les requêtes possibles qui peuvent être effectuées sur les arbres et/ou en raison du manque d'espace, mais nous limitons notre étude à celles qui nous seront le plus utiles. Soient $T$ un arbre et/ou, $root(T)$ est la racine de $T$, $\mathcal{F}$ une famille représentée par l'arbre $T$ définie sur un ensemble fini de valeurs $S$, nous sommes intéressés aux requêtes suivantes :

1. Calculer l'ensemble de valeurs de $S$ à partir de l'arbre $T$ (VALUES($T$)) ;
2. Calculer la famille $\mathcal{F}$ représentée par un arbre $T$ (FAMILY($T$)) ;
3. Calculer la cardinalité (le nombre d'ensembles) de la famille $\mathcal{F}$ (CARD($\mathcal{F}$)) ;
4. Vérifier si un ensemble donné $E$ est appartient à la famille $\mathcal{F}$ ($E \in \mathcal{F}$) ;
5. Calculer un ensemble aléatoire $E$ avec une probabilité uniforme (RANDOM($\mathcal{F}$)) ;
6. Vérifier si l'arbre est uniforme (ISUNIFORM($\mathcal{F}$)).

Pour calculer VALUES($T$) et FAMILY($T$), on procède tel qu'il est décrit dans les point 3 et 4 de la définition de la structure de base des arbres et/ou. L'ensemble de valeurs $S$ et la famille de solutions $\mathcal{F}$ de l'arbre $T$ sont égaux à l'ensemble de valeurs et la famille de solutions du nœud racine de $T$. Pour plus de détails sur le calcul du FAMILY($T$), voir l'algorithme 10.

Pour calculer CARD($\mathcal{F}$ ), il suffit de définir de manière récursive CARD des nœuds feuilles, des nœuds $\wedge$ et des nœuds $\vee$ en montant jusqu'à la racine de l'arbre. Ainsi, CARD($\mathcal{F}$ ) = CARD($r$), où $r$ est la racine de $T$. CARD est définie récursivement par

$$\text{CARD(v)} = \begin{cases} 1 & \text{si } v \text{ est une feuille ;} \\ \sum_{w \in v.fils} \text{CARD}(w) & \text{si } v \text{ est de type } ou \text{ ;} \\ \prod_{w \in v.fils} \text{CARD}(w) & \text{si } v \text{ est de type } et. \end{cases}$$

Pour vérifier si un ensemble $E$ est appartient à la famille $\mathcal{F}$ , on procède comme suit. Pour chaque nœud $v$ :

1. Si $v$ est une feuille, retourner vrai si $E = \{v\}$ ;
2. Si $v$ est un nœud de type *ou*, retourner vrai si et seulement si $E$ appartient à au moins une famille induite par un nœud enfant de $v$ ;
3. Si $v$ est un nœud de type *et*, pour chaque nœud enfant $w$ de $v$, vérifier si $E \cap \text{VALUES}(w)$ appartient à la famille induite par le sous-arbre enraciné par $w$ (si c'est le cas, retourner vrai, sinon, retourner faux).

Pour extraire un ensemble aléatoire en garantissant que tous les ensembles ont la même probabilité d'être choisis, on introduit pour chaque nœud $v$ de l'arbre une probabilité uniforme dépend de CARD(v) et CARD($\mathcal{F}$ ). Dans la figure 3.3(a), le sous-arbre gauche de la racine représente deux ensembles et le sous-arbre droit représente un seul ensemble. Afin que le calcul soit équitable, on introduit la probabilité $\frac{2}{3}$ pour le sous-arbre gauche et $\frac{1}{3}$ pour le sous-arbre droit. Pour calculer $E$, il suffit de suivre les étapes suivantes sur chaque nœud $v$.

1. Si $v$ est une feuille, retourner $\{\text{CONTENT}(v)\}$ ;
2. Si $v$ est un nœud de type *et*, retourner $\bigcup_{w \in v.fils} \text{RANDOM}(w)$ ;
3. Si $v$ est un nœud de type *ou*, choisissez un nœud enfant $w$ de $v$ avec une probabilité de $\text{CARD}(w)/\text{CARD}(v)$ pour chaque enfant de $v$, puis retourner $\text{RANDOM}(w)$.

Pour vérifier que $\mathcal{F}$ est uniforme, on procède comme suite, Soit $k$ un entier et $v$ un nœud de $T$. $\text{ISUNIFORM}(v,k)$ est défini comme suit :

1. Si $v$ est une feuille, $\text{ISUNIFORM}(v,k)$ est vrai si et seulement si $k = 1$ ;
2. Si $v$ est un nœud de type *ou*, $\text{ISUNIFORM}(v,k)$ est vrai si et seulement si $\text{ISUNIFORM}(w,k)$ pour chaque nœud enfant $w$ de $v$ ;
3. Si $v$ est un nœud de type *et*, $\text{ISUNIFORM}(v,k)$ est vrai si et seulement si $\text{ISUNIFORM}(w,k_w)$ pour chaque nœud enfant du nœud $v$ et un entier $k_w$ avec $k = \sum_{w \in v.fils} k_w$.

**Théorème 8.** *Soit T un arbre et/ou de n nœuds. Les opérations* CARD*,* RANDOM*,* $\in$*,* ISUNIFORM *ont une complexité de* $O(n)$.

**Démonstration :** CARD effectue un seul parcours de l'arbre et/ou, qui est trivialement de complexité $O(n)$. L'opération $\in$ est également linéaire, parce que $\text{VALUES}(v)$ est connue pour chaque nœud $v$ de $T$. Il suffit de stocker cette information directement dans la structure de données ou la pré-calculer en temps linéaire. RANDOM est également linéaire, car elle s'effectue en un seul parcours de l'arbre en utilisant CARD, ce qui est $O(n)$. Enfin, ISUNIFORM est clairement $O(n)$.

### *3.2.2 MODIFICATEURS*

Plusieurs modificateurs peuvent être définis sur les arbres et/ou. Nous nous intéressons ici à l'aplatissement et à la binarisation de l'arbre :

1. Aplatir l'arbre $T$ (FLATTEN($T$)) ;
2. Binariser l'arbre $T$ (BINARIZE($T$)).

L'aplatissement d'un arbre et/ou peut être considéré comme un moyen de compression de données afin d'économiser l'espace mémoire occupé par la structure. Il est aussi un moyen d'obtenir une structure de données où les nœuds de type *et* alternent avec les nœuds de type *ou* à chaque niveau. Plus précisément, l'aplatissement effectue les opérations suivantes :

1. Si un nœud interne $v$ n'a pas enfant, alors on le supprime ;
2. Si un nœud interne $v$ a un seul enfant $w$, alors on supprime $v$ et on le remplace par $w$ ;
3. Si un nœud interne $v$ a un enfant $w$ du même type $(\text{CONTENT}(v) = \text{CONTENT}(w))$, alors on déplace tous les enfants du nœud $w$ à la liste des enfants du nœud $v$ et on supprime $w$ (voir l'algorithme 3).

**Algorithme 3** Aplatissement d'un arbre et/ou

```
1:  fonction FLATTEN(T : arbre et/ou) : arbre et/ou
2:      retourner FLATTEN(T.root)
3:  fin fonction

4:  fonction FLATTEN(v : Leaf) : arbre et/ou                    ▷ Nœud feuille
5:      retourner v
6:  fin fonction

7:  fonction FLATTEN(v : InternalNode) : arbre et/ou            ▷ Nœud interne
8:      si |v.children| = 0 alors
9:          Supprimer v
10:     sinon
11:         si |v.children| = 1 alors
12:             v ← FLATTEN(v.children[1])
13:         sinon
14:             pour tout w ∈ v.children faire
15:                 si w.value = v.value alors
16:                     pour tout z ∈ w.children faire
17:                         Ajouter FLATTEN(z) à v.children
18:                     fin pour
19:                     Supprimer w
```

```
20:                 sinon
21:                     w ←FLATTEN(w)
22:                 fin si
23:             fin pour
24:         fin si
25:     fin si
26:     retourner v
27: fin fonction
```

**Exemple 14.** La figure 3.4, montre l'aplatissement d'un arbre représentant la famille $\mathcal{F}$ =$\{\{1,4\},\{2,4\},\{3,4\}\}$.

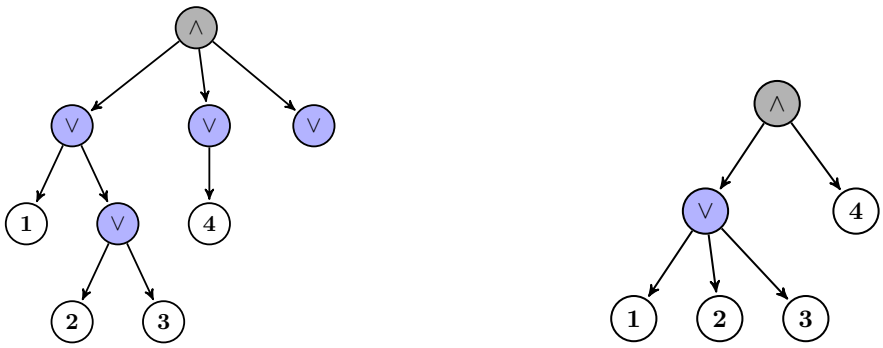



**Figure 3.4: Aplatissement d'un arbre et/ou.**

Inversement, l'opération BINARIZE permet à un nœud interne d'avoir au plus deux nœuds enfants. Cela se fait par un parcours de la racine vers les feuilles. Si le nœud courant a un seul enfant, on le remplace par son enfant unique et on continue la binarisation, sinon, si le nœud courant possède plus de deux nœuds enfants, on insère autant de nœuds internes de même type que le nœud courant selon les besoins. La binarisation de l'arbre simplifie la mise en œuvre de tous les opérations définies sur les arbre et/ou mais donne un arbre plus grand en taille donc elle occupe plus d'espace mémoire.

**Exemple 15.** La figure 3.5, montre la binarisation d'un arbre représentant la famille $\mathcal{F}$ =$\{\{1\},\{2,3\},\{4,5,6\}\}$.

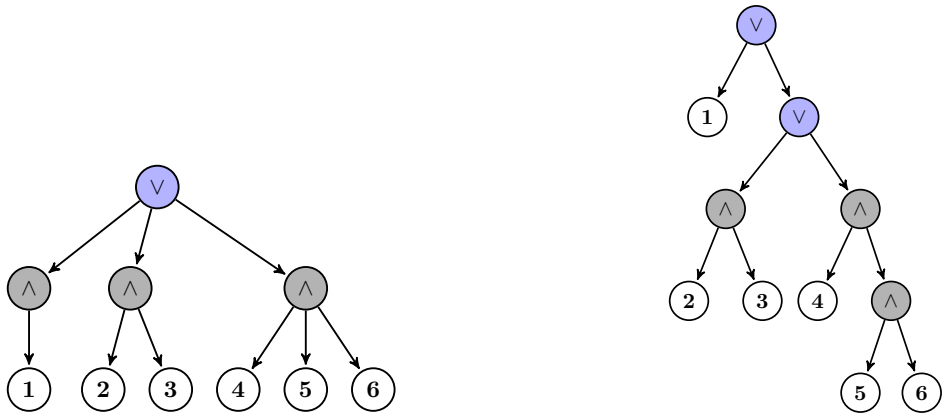



**Figure 3.5: Binarisation d'un arbre et/ou.**

**Théorème 9.** *Soit $T$ un arbre et/ou de n nœuds. Les opérations* FLATTEN*$(T)$ et* BINARIZE*$(T)$ ont une complexité de $O(n)$ dans le pire cas.*

**Démonstration :** Il est clair que les opérations FLATTEN et BINARIZE s'effectuent en un seul parcours de l'arbre. Pour chaque nœud $v$, FLATTEN supprime $v$ s'il n'a pas d'enfants ou elle le supprime et déplace ses enfants vers la liste des enfants de son père. La suppression et le déplacement s'effectuent en temps constant ($O(1)$). Donc FLATTEN se réalise en $O(n)$. BINARIZE effectue l'insertion d'un nouveau nœud ($O(1)$) et le déplacement des nœuds enfants du nœud courant vers la liste des enfant du nouveau nœud créé qui sont dans le pire cas $n$ déplacements, donc en total BINARIZE se fait en $O(n)$.

Plusieurs autres opérations de vérification et de recherche peuvent être définies sur les arbres et/ou comme la recherche d'une clé dans l'arbre, la recherche des sous-ensembles de solutions et des sous-familles, le calcul de la famille de solution d'un nœud interne, l'extraction des sous-familles qui ne contient certains valeurs, etc. Ainsi que d'autres opérations ensemblistes comme les opérations d'union, d'intersection, de complétude et de différence peuvent être aussi définis sur les arbres et/ou, ces dernières opérations est de préférable les appliquer sur

les DBD et DBDZ correspondants aux arbres et/ou car il est facile de convertir un arbre et/ou en un DBD ou un DBDZ comme nous le voyons dans la section suivante.

## 3.3 RELATION AVEC LES DIAGRAMMES BINAIRES DE DÉCISION

Dans cette section on s'intéresse au lien entre les arbres et/ou et les DBD/DBDZ en expliquant les conversions effectuées pour transformer un arbre et/ou en un DBD/DBDZ et vice versa. Notant que dans un contexte d'énumération, le fait d'avoir des négations de valeurs dans les feuilles de l'arbre et/ou et ainsi dans le DBD correspondant n'a aucun sens, on ne trouve donc pas d'arcs indexés par 1 (0) allant d'un nœud interne vers le nœud 0-terminal (1-terminal) dans le DBD.

### *3.3.1 CONVERSION D'UN ARBRE ET/OU EN UN DIAGRAMME BINAIRE DE DÉCISION*

Pour construire le DBD/DBDZ correspondant à un arbre et/ou, il suffit de parcourir l'arbre de la racine vers les nœuds feuilles en construisant récursivement le DBD correspondant à chaque nœud actuellement visité selon son type. Si le nœud courant est une feuille, le DBD correspondant est composé d'un seul nœud interne étiqueté par la valeur de la feuille et ayant le nœud 0-terminal comme fils bas et le nœud 1-terminal comme fils haut (voir figure 3.6(a)). Si le nœud courant est un nœud interne, on construit récursivement les DBD correspondants à ses enfants puis on applique récursivement la fonction APPLY (voir la fonction 2) sur ces DBD pour avoir le DBD correspondant au nœud courant (voir section 2.4.5). L'opérateur logique appliqué à chaque étape est selon le type du nœud, s'il est de type *et*, on applique l'opérateur logique $\wedge$ sinon on applique l'opérateur logique $\vee$. Si le nœud courant est de type *et*, alors les DBD correspondants à ses enfants sont liés par des arcs indexés par 1 ou des arcs pleins (voir la figure 3.6(b)), sinon ils sont liés par des arcs indexés par 0 ou des arcs pointillés (voir la

figure 3.6(c)). En pratique la procédure de construction du DBD correspondant à un arbre et/ou commence par la construction des DBD correspondants aux feuilles puis elle remonte jusqu'à la racine de l'arbre. L'algorithme 4 résume le processus de conversion, la fonction APPLY est inspirée de la fonction APPLY définie par Bryant, elle traite juste les deux opérateurs *et* et *ou* dans le cas où il n'y a pas de négation. Cette fonction redéfinie permet d'optimiser le temps d'exécution et donne un DBD ordonné et réduit.

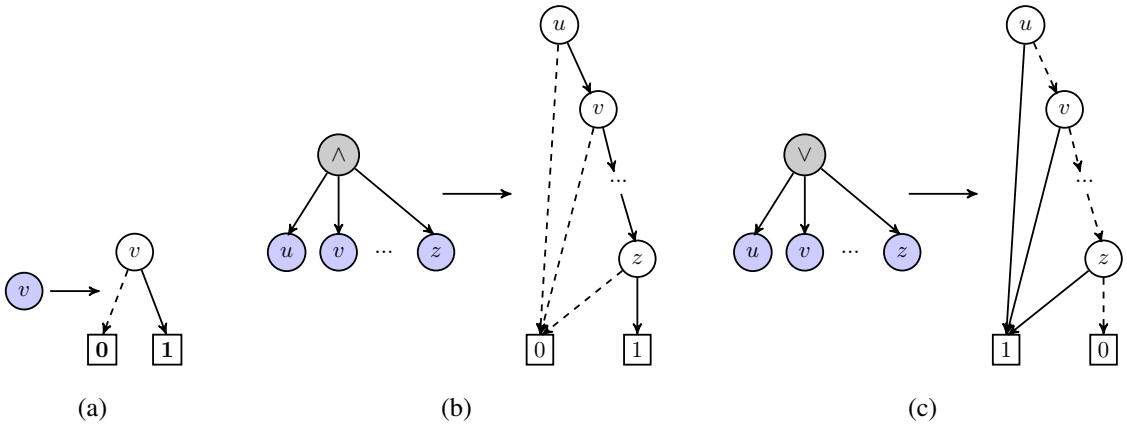



**Figure 3.6: Conversion d'un arbre et/ou en un DBD. (a) Le DBD correspondant à un nœud feuille. (b) Le DBD correspondant à un nœud de type et. (c) Le DBD correspondant à un nœud de type ou.**

**Algorithme 4** Convertir un arbre et/ou en un DBD

**fonction** CONVERTTREETOBDD($T$ : arbre et/ou, $id : entier, zero, un : DBD$) : DBD
    $zero \leftarrow$NEWTERMINALNODE(0,"0", $false$)
    $un \leftarrow$NEWTERMINALNODE(1,"1", $false$)
    **retourner** CONVERTTREETOBDD($T.root, 2, zero, un$)
**fin fonction**

**fonction** CONVERTTREETOBDD($v$ : Leaf, $id : entier, zero, un : DBD$) : DBD
    $dbd \leftarrow$NEWNOTTERMINALNODE($id, v.value, false, zero, un$)
    **retourner** $dbd$
**fin fonction**

**fonction** CONVERTTREETOBDD($v$ : OrNode, $id : entier, zero, un : DBD$) : DBD
    $dbd1 \leftarrow$CONVERTTREETOBDD($v.children[1], id, zero, un$)
    $dbd \leftarrow dbd1$

```
13:     id ← dbd.root.id + 1
14:     pour i ← 2 à |v.children| faire
15:         dbd2 ←ConvertTreeToBDD(v.children[i], id, zero, un)
16:         dbd ← Apply(dbd1, dbd2, ∨)
17:         id ← dbd.root.id + 1
18:         dbd1 ← dbd
19:     fin pour
20:     retourner dbd
21: fin fonction

22: fonction ConvertTreeToBDD(v : AndNode, id : entier, zero, un : DBD) : DBD
23:     dbd1 ←ConvertTreeToBDD(v.children[1], id, zero, un)
24:     dbd ← dbd1
25:     id ← dbd.root.id + 1
26:     pour i ← 2 à |v.children| faire
27:         dbd2 ←ConvertTreeToBDD(v.children[i], id, zero, un)
28:         dbd ← Apply(dbd1, dbd2, ∧)
29:         id ← dbd.root.id + 1
30:         dbd1 ← dbd
31:     fin pour
32:     retourner dbd
33: fin fonction
```

**Exemple 16.** La figure 3.7 représente les différentes étapes à suivre pour convertir l'arbre et/ou représentant la famille $\mathcal{F} = \{\{1,4\},\{2,3,4\}\}$ en un DBDOR.

**Théorème 10.** *Soit $T$ un arbre et/ou de n nœuds et D le DBDOR correspondant à $T$. L'opération de conversion de l'arbre $T$ en un DBDOR a une complexité de $O(n^3)$.*

**Démonstration :** Le parcours de l'arbre est d'ordre $O(n)$, la fonction Apply est d'ordre $O(n^2)$ donc la conversion de l'arbre $T$ en un DBDOR est d'ordre $O(n^3)$.

### *3.3.2 CONVERSION D'UN DIAGRAMME BINAIRE DE DÉCISION EN UN ARBRE ET/OU*

Pour convertir un DBD en un arbre et/ou, il suffit de parcourir le DBD de la racine vers les nœuds terminaux en ajoutant récursivement des nœuds internes ($\wedge$ et $\vee$) et en faisant des

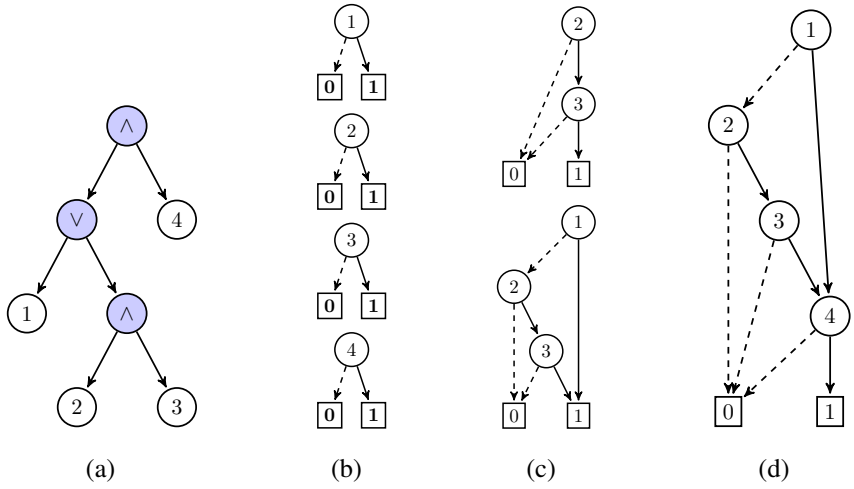



**Figure 3.7: Conversion de l'arbre et/ou représentant la famille $\mathcal{F} = \{\{1,4\},\{2,3,4\}\}$ en un DB-DOR. (a) L'arbre et/ou. (b) Les DBD correspondant aux feuilles. (c) Le DBD correspondant à $(2 \wedge 3)$ et le DBD correspondant à $(1 \vee (2 \wedge 3))$. (d) Le DBDOR représentant $\mathcal{F}$ .**

liaisons entre l'arbre correspondant au nœud courant et les arbres correspondants à ses fils bas et haut. L'arbre correspondant à un nœud interne dépend du type de ses nœuds fils, pour chaque nœud interne $v$ du DBD :

1. Si son fils bas est le nœud 0-terminal et son fils haut est le nœud 1-terminal, l'arbre correspondant est une feuille de valeur $v$ (voir figure 3.8(a)) ;
2. Si son fils bas est le nœud 0-terminal et son fils haut est un nœud de décision ayant $w$ comme variable, l'arbre et/ou correspondant est composé d'un nœud interne de type *et* ayant la feuille $v$ et l'arbre correspondant au fils haut ($AOT(w)$ : And/Or Tree (w). i.e. l'arbre et/ou de racine $w$) comme enfants (voir figure 3.8(b)) ;
3. Si son fils haut est le nœud 1-terminal et son fils bas est un nœud de décision ayant $w$ comme variable, l'arbre et/ou correspondant est composé d'un nœud interne de type *ou* ayant la feuille $v$ et l'arbre correspondant au fils bas ($AOT(w)$) comme enfants (voir figure 3.8(c)) ;
4. Si son fils haut est un DBD de racine $u$ et son fils bas est un DBD de racine $w$, l'arbre

et/ou correspondant est composé d'un nœud interne de type *ou* ayant l'arbre correspondant au fils bas ($AOT(w)$) et un nœud de type *et* comme enfants, ce nœud de type *et* de son tour, a la feuille $v$ et l'arbre correspondant au fils bas ($AOT(u)$) comme enfants (voir figure 3.8(d)).

A la fin de l'opération de conversion, on applique la fonction d'aplatissement sur l'arbre résultant. L'algorithme 5 résume le processus de conversion. La fonction IsNotTerminalNode($v$) est vraie si $v$ est un nœud non terminal, IsZeroTerminal($v$) est vraie si $v$ est un nœud 0-terminal et IsOneTerminal($v$) est vraie si $v$ est un nœud 1-terminal.

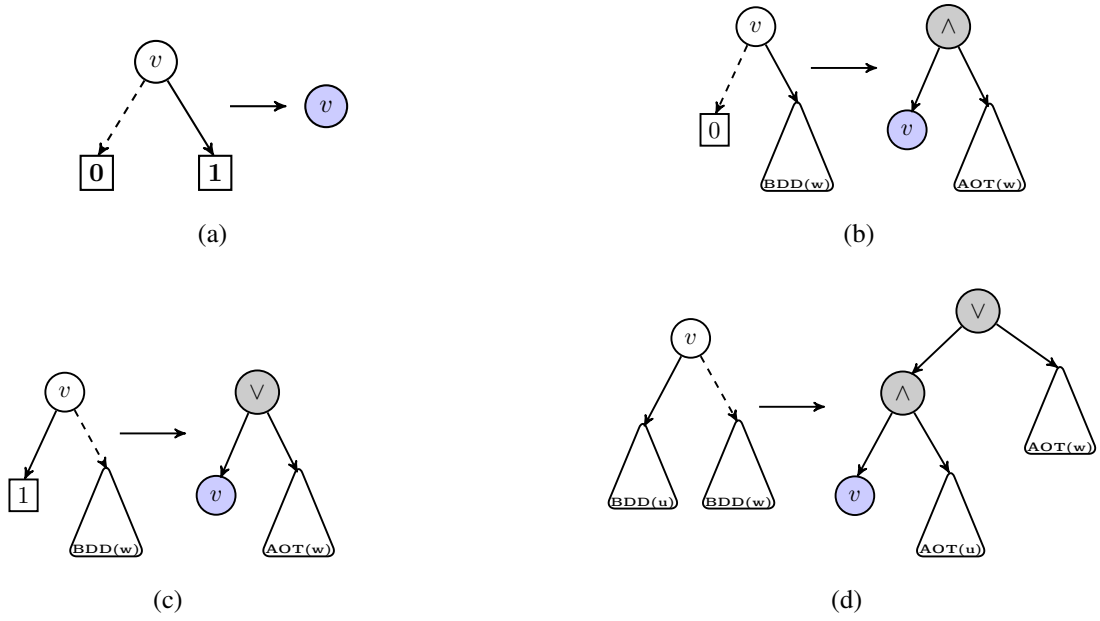



**Figure 3.8: Les différents cas possibles pendant la conversion d'un DBD en un arbre et/ou. (a) Les fils de la variable sont des nœuds terminaux. (b) Le fils bas 0-terminal et le fils haut non terminal. (c) Le fils haut 1-terminal et le fils bas non terminal. (d) les deux fils non terminaux.**

**Exemple 17.** La figure 3.9, représente les différentes étapes à suivre pour convertir le DBD représentant la famille $\mathcal{F} = \{\{a,c,d\},\{b\},\{e\}\}$ en un arbre et/ou.

**Théorème 11.** *Soit D un DBD de n nœuds et T l'arbre et/ou correspondant à D ayant m nœuds. L'opération de transfert de D à un arbre et/ou a une complexité de* $O(n+m)$.

**Algorithme 5** Convertir un DBD en un arbre et/ou

```
1:  fonction CONVERTBDDTOTREE(D : DBD, id :entier) : arbre et/ou
2:      v ← D.root
3:      si ISNOTTERMINALNODE(v) alors
4:          low ← v.low, high ← v.high
5:          si ISZEROTERMINAL(low) et ISONETERMINAL(high) alors
6:              id ← id + 1, x ←NEWNODE(id, v.value)
7:          sinon
8:              si ISNOTTERMINALNODE(high) et ISZEROTERMINAL(low) alors
9:                  id ← id + 1, x ←NEWNODE(id, ∧, nul)
10:                 id ← id + 1, y ←NEWNODE(id, v.value)
11:                 Ajouter y à x.children
12:                 Ajouter CONVERTBDDTOTREE(high) à x.children
13:             sinon
14:                 si ISNOTTERMINALNODE(low) et ISONETERMINAL(high) alors
15:                     id ← id + 1, x ←NEWNODE(id, ∨, nul)
16:                     id ← id + 1, y ←NEWNODE(id, v.value)
17:                     Ajouter y à x.children
18:                     Ajouter CONVERTBDDTOTREE(low) à x.children
19:                 sinon
20:                     id ← id + 1, x ←NEWNODE(id, ∨, nul)
21:                     id ← id + 1, y ←NEWNODE(id, ∧, nul)
22:                     id ← id + 1, z ←NEWNODE(id, v.value)
23:                     Ajouter z à y.children
24:                     Ajouter CONVERTBDDTOTREE(high) à y.children
25:                     Ajouter y à x.children
26:                     Ajouter CONVERTBDDTOTREE(low) à x.children
27:                 fin si
28:             fin si
29:         fin si
30:     fin si
31:     retourner FLATTEN(x)
32: fin fonction
```

**Démonstration :** La construction de l'arbre et/ou se fait par un parcours du DBD qui est d'ordre $O(n)$, et l'aplatissement s'effectue en $O(m)$, donc le tout en $O(n+m)$

Les arbres et/ou peuvent représenter n'importe quelle famille d'ensemble de solutions. Ils produisent des représentations plus compactes et plus expressives que les DBD et les DBDZ

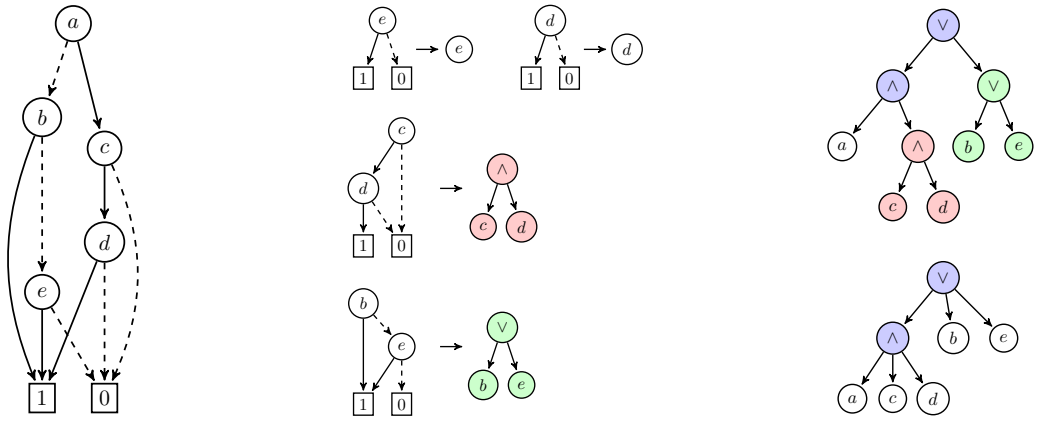



**Figure 3.9: Conversion du DBD représentant la famille $\mathcal{F} = \{\{a,c,d\},\{b\},\{e\}\}$ en un arbre et/ou. (a) Le DBDZ représentant la famille $\mathcal{F}$ . (b) Les arbres correspondants aux DBD représentants $e$, $d$, $(c \wedge d)$ et $(b \vee e)$. (c) En haut, l'arbre et/ou représentant $\mathcal{F}$ avant l'aplatissement, en bas, l'arbre et/ou représentant $\mathcal{F}$ après l'aplatissement.**

dans certaines situations. Sa construction semble complexe pour représenter une petite famille d'ensembles, mais elle est une alternative compacte lors du calcul de grandes familles, en particulier lorsque les nœuds $\wedge$ partitionnent efficacement l'ensemble des solutions. L'utilisation de cette structure de données semble nouvelle et d'après nos premières observations, elle pourrait être utilisée dans plusieurs contextes.

## 3.4 CONCLUSION

Les arbres et/ou offrent un modèle de représentation alternatif et compact pour stocker les familles de solutions combinatoires. Ils peuvent être utilisés comme structure pour enregistrer et manipuler les ensembles de solutions des différents problèmes d'énumération, surtout, parce que la plupart des opérations et modifications appliquées sur ceux-ci s'effectuent en temps polynomial.

# CHAPITRE 4

# TRANSVERSAUX DE CIRCUITS

Le problème d'énumérer les transversaux de circuits de cardinalité minimale (TCCM) dans un graphe orienté est un problème d'optimisation combinatoire fondamental de la classe NP-difficile. Il existe une version pour les graphes orientés et les graphes non orientés, toutes les deux étant NP-difficiles. D'autres variations possibles consistent à ajouter des poids sur les sommets ou les arcs/arêtes du graphe et de chercher un ou plusieurs transversaux de poids minimal [Festa et al., 1999]. Dans ce chapitre, nous nous intéressons seulement aux graphes orientés non pondérés dans le but de calculer un transversal de cardinalité minimale.

Depuis la définition du problème TCCM par Karp et la démonstration qu'il est NP-difficile [Yannakakis, 1978], le problème TCCM est largement étudié et plusieurs approches ont été proposées afin de trouver des algorithmes optimaux permettant de le résoudre.

Au début, l'étude du problème est limitée à des classes particulières de graphes. Plusieurs algorithmes polynomiaux ont été proposés en ce sens [Smith et Walford, 1975, Shamir, 1979, Levy et Low, 1988, Lin et Jou, 2000]. Plus tard, quelques chercheurs se sont intéressés à l'étude du problème dans le cadre général [Razgon, 2006, Fomin et al., 2007]. Cependant, sans surprise, tous les algorithmes proposés dans le cas général sont exponentiels en la taille du graphe.

Dans ce chapitre, nous étudions en détails l'approche proposée par Levy et Low et étendue par Lin et Jou ; qui permet de calculer un TCCM en temps polynomial pour certains types de graphes appelés *DOME-contractibles*. Mais avant d'entrer dans les détails de cette approche, on démontre d'abord la NP-complétude de la version décisionnelle du problème TCCM.

## 4.1 COMPLEXITÉ DU PROBLÈME

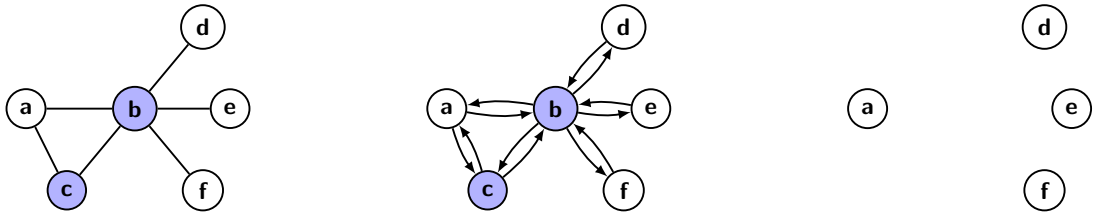
Le problème TCCM est un problème NP-difficile. Pour démontrer sa complexité il suffit de démontrer que le problème de couverture par sommets (*vertex cover*) est polynomialement réductible à ce problème.

**Théorème 12.** *Soit $G = (V,A)$ un graphe orienté et k un entier non négatif. Le problème de décider s'il existe un ensemble $U$ de k sommets tel que $G[V-U]$ est acyclique est NP-complet.*

**Démonstration :** Premièrement, nous devons montrer que le problème TCCM est dans NP. Un TCCM est un sous-ensemble de sommets $U \subseteq V$ tels que, en supprimant tous les sommets de $U$, le sous-graphe induit $G[V-U]$ est acyclique. Le problème est dans NP, car la donnée de $U$ est un certificat facilement vérifiable en temps polynomial (la construction du graphe $G[V-U]$ et la vérification qu'il est acyclique est vérifiable en $O(|V|+|A|)$).

Ensuite, nous démontrons que le problème TCCM est NP-Complet par la réduction du problème de couverture par sommets à ce dernier. Soit $G = (V,A)$ un graphe non orienté et $k$ un entier positif. L'algorithme de vérification est défini comme suit : pour une instance de couverture par sommets de taille $k$ dans $G$ on construit un TCCM dans $G' = (V,A')$ tel que $G' = (V,A')$ est le graphe orienté ayant les mêmes sommets que $G$ et pour chaque arrête $(u,v)$ de $G$ on crée deux arcs $(u,v)$ et $(v,u)$ dans $G'$ (voir figure 4.1).

Maintenant, nous montrons qu'il existe une couverture par sommets de taille $k$ dans $G$ si et




(a) : $G = (V,A)$ (b) : $G' = (V,A')$ (c) : Graphe acyclique

**Figure 4.1: Réduction du problème de couverture par sommets au problème TCCM. (a) Graphe non orienté $G = (V,A)$, l'ensemble $S = \{b,c\}$ est une couverture par sommets du graphe $G$. (b) Le graphe orienté $G' = (V,A')$ associé au graphe $G$, l'ensemble $S = \{b,c\}$ est un TCCM du graphe $G'$. (c) Le graphe acyclique induit par la suppression de $S$.**

seulement s'il y a un sous-ensemble de $k$ sommets dans $G'$ dont sa suppression casse tous les circuits. Tout d'abord, supposons qu'il existe une couverture par sommets de taille $k$ dans $G$ (dans la figure 4.1(a) l'ensemble $S = \{b,c\}$ couvre toutes les arrêtes de $G$). Supprimons ces $k$ sommets de $G'$ avec les arcs incidents de ces sommets. Sachant que pour chaque arc orienté $(u,v) \in G'$, au moins l'un des sommets $u$ et $v$ doit être enlevé, parce que l'un des deux doit être dans la couverture par sommets. Ainsi, après la suppression de ces $k$ sommets et leurs arcs incidents de $G'$, on ne trouve aucune paire de sommets ayant un arc entre eux, et par conséquent il n'existe aucun circuit.

Inversement, supposons qu'il existe un ensemble $S$ de $k$ sommets dans $G'$ dont sa suppression casse tous les circuits. Par construction de chaque paire de sommets $u$, $v$ de $S$, chaque circuit dans $G'$ doit contenir au moins un sommet de $S$. En particulier, chaque circuit de longueur 2 de $G'$ doit contenir au moins un sommet de $S$. Chaque circuit de longueur 2 correspond à une arrête dans $G$. Donc, chaque arrête dans $G$ a au moins l'une de ses extrémités dans $S$, et donc $S$ est une couverture par sommet de $G$.

Après avoir démontré que le problème TCCM est NP-difficile, on présente maintenant une approche permettant de calculer un TCCM en se basant sur la réduction du graphe.

## 4.2 OPÉRATEURS DE CONTRACTION DE LEVY ET LOW

Premièrement, on commence par l'introduction de la notation de trois opérations utilisées pour la description des opérateurs de contraction définis par Levy et Low. Soit $G = (V,A)$ un graphe orienté et $u$, $v$ et $w$ des sommets appartient à $V$. L'opération $\text{DELETE}(u)$ supprime le sommet $u$ et tous ses arcs incidents $A(u)$, l'opération $\text{MERGE}(u,v)$ fusionne les deux sommets $u$ et $v$ et l'opération $\text{VANISH}(u)$ supprime le sommet $u$ tout en préservant les liens entre ses prédécesseurs $N^-(u)$ et ses successeurs $N^+(u)$.

1. $G.\text{DELETE}(u)$ est le graphe $G' = (V',A')$, où $V' = V - \{u\}$ et $A' = A - A(u)$.
2. $G.\text{MERGE}(u,v)$ est le graphe $G' = (V',A')$, où $V' = V - \{v\}$ et $A' = (A \cup \{(w,u)|w \in N^-(v)\} \cup \{(u,w)|w \in N^+(v)\}) - A(v)$.
3. $G.\text{VANISH}(u)$ est le graphe $G' = (V',A')$, où $V' = V - \{u\}$ et $A' = (A \cup \{(v,w)|v \in N^-(u), w \in N^+(u)\}) - A(u)$.

Levy et Low ont défini cinq opérateurs de contraction qui simplifient le calcul d'un TCCM dans un graphe orienté, ces opérateurs sont les suivantes :

**Définition 9.** *Soit $G = (V,A)$ un graphe orienté et $u \in V$. Alors*

1. $G.\text{IN0}(u) = G.\text{DELETE}(u)$*, si u est une source ;*
2. $G.\text{OUT0}(u) = G.\text{DELETE}(u)$*, si u est un puits ;*
3. $G.\text{LOOP}(u) = G.\text{DELETE}(u)$*, si* $(u,u) \in A$ *;*
4. $G.\text{IN1}(u) = G.\text{MERGE}(v,u)$*, si* $N^-(u) = \{v\}$ *;*
5. $G.\text{OUT1}(u) = G.\text{MERGE}(v,u)$*, si* $N^+(u) = \{v\}$.

Soit $G' = (V',A')$ le graphe orienté résultant de $G$ par l'application répétée des opérateurs de contraction jusqu'à ce qu'aucune contraction ne soit possible, $G'$ est appelé *le graphe*

*contracté* de $G$. Pour tout graphe orienté $G$ ne contenant pas d'arcs parallèles, il existe un unique graphe contracté $G'$. L'importance de cette propriété est qu'elle permet d'appliquer les opérateurs de contraction dans n'importe quel ordre sans affecter le résultat final.

**Théorème 13.** *Soit $G = (V,A)$ un graphe orienté, $u \in V$ un sommet,*

1. *Si G est contracté en $G'$ par l'application d'un opérateur* $\textsc{In0}(u)$, $\textsc{Out0}(u)$, $\textsc{In1}(u)$ *ou* $\textsc{Out1}(u)$, *et $S'$ est un TCCM de $G'$ alors $S'$ est un TCCM de G ;*
2. *Si G est contracté en $G'$ par l'application de l'opérateur* $\textsc{Loop}(u)$, *et $S'$ est un TCCM de $G'$ alors $S = S' \cup \{u\}$ est un TCCM de G.*

**Démonstration :**

1. Soit $U \in TCCM(G)$, nous montrons que $U$ est un TC de $G'$. Soit $c$ un circuit de $G'$. Si $u$ est une source ou un puits, alors $u \notin c$, et parce que $c$ est aussi un circuit de $G$, il est donc couvert par $U$. Si $c$ n'est pas un circuit de $G$, il est forcément la réduction d'un circuit de $G$ issu de la suppression d'un sommet $u$ par l'application de $\textsc{In1}(u)$ ou $\textsc{Out1}(u)$, donc, il est couvert par $U$. Il reste à prouver la minimalité de $U$ pour $G$. Supposons qu'il existe $U' \in TCCM(G')$ tel que $|U'| < |U|$, donc, $U'$ est un TC de $G$, c'est une contradiction de la minimalité de $U$ pour $G$.
2. Soit $U \in TCCM(G)$. Premièrement, nous montrons que $U - \{u\}$ est un TC de $G'$. Soit $c$ un circuit de $G'$, alors $c$ est également un circuit de $G$ et il ne passe pas par $u$, donc, il est couvert par $U$. La minimalité de $U - \{u\}$ résulte de la minimalité de $U$ et le fait que l'arc $(u,u)$ doit aussi être couvert.

**Exemple 18.** La figure 4.2 montre un exemple de réduction d'un graphe avec les cinq opérateurs de Levy et Low. Tout d'abord, les opérateurs $\textsc{In0}$ et $\textsc{Out0}$ sont appliqués sur les sommets 1 et 2. Ensuite, les opérateurs $\textsc{Out1}$ et $\textsc{In1}$ sont appliqués sur 0 et 5 et, Enfin, l'opérateur $\textsc{Loop}$ est appliqué sur 3 et 4, ce qui donne le graphe vide. Donc, l'ensemble $S = \{3,4\}$ est un TCCM du graphe initial.

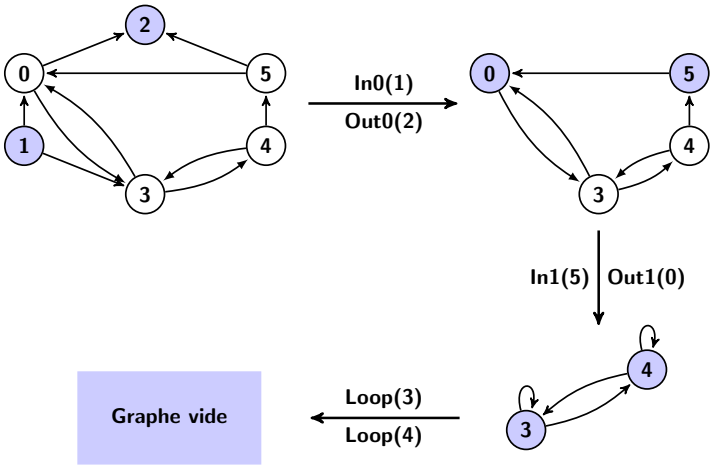



**Figure 4.2: Réduction d'un graphe en appliquant les cinq opérateurs de Levy et Low.**

**Théorème 14.** *Soit $G = (V, A)$ un graphe orienté de n sommets et m arcs. Si G est complètement réductible au graphe vide par l'application des opérateurs de Levy et Low. Alors, on peut calculer un TCCM de G en temps polynomial égal à $\mathcal{O}(m\log(n))$.*

**Démonstration :** La complexité de l'algorithme est liée au nombre de suppressions et de déplacement d'arcs, chaque suppression ou déplacement exige $\mathcal{O}(1)$, chaque arc peut être déplacé $\log(n)$ fois au maximum par les opérateurs IN1 et OUT1, donc la complexité totale est $\mathcal{O}(m\log(n))$.

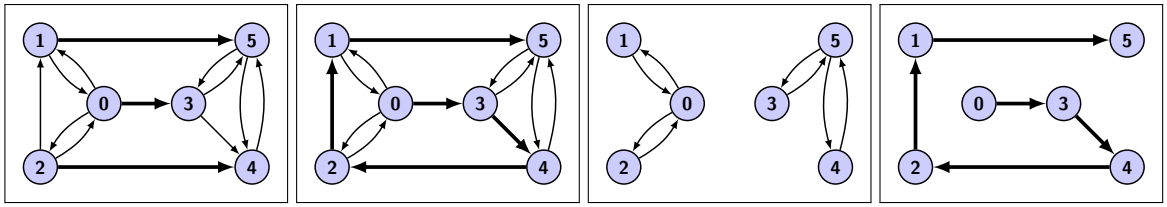
Si chaque nœud du graphe a plus d'un arc entrant et plus d'un arc sortant, l'ensemble des opérateurs de Levy et Low ne s'appliquent pas sur le graphe et ne permettent pas de calculer un TCCM. Néanmoins, cette ligne de travail a un impact significatif dans l'étude du problème TCCM pour deux raisons. Premièrement, on peut trouver un TCCM pour tout graphe réductible même s'il est de grande taille. Deuxièmement, la plupart des algorithmes exacts proposés ultérieurement utilisent ces opérations de réduction.

## 4.3 OPÉRATEURS DE CONTRACTION DE LIN ET JOU

En 2000, Lin et Jou [Lin et Jou, 2000] ont ajouté trois autres opérateurs de réduction et ont présenté un algorithme exact basé sur la technique séparation et évaluation permettant de calculer un TCCM en temps polynomial pour des graphes orientés.

### *4.3.1 L'OPÉRATEUR* PIE

Soient $G = (V,A)$ un graphe orienté, $u, v \in V$ des sommets et $e = (u,v) \in A$ un arc. L'arc $e$ est acyclique si aucun circuit de $G$ passe par $e$, dans la figure 4.3(a), les arcs en gras sont des arcs acycliques. L'arc $e = (u,v)$ est appelé $\pi - arc$ si $(v,u)$ est un arc de $G$, dans la figure 4.3(c), tous les arcs sont des $\pi - arcs$. L'arc $e$ est appelé $\pi - acyclique$ s'il est un arc acyclique dans le sous-graphe $G - \pi(G)$ tel que $\pi(G)$ est un sous-graphe de $G$ composé de tous les $\pi - arcs$ de $G$. Dans la figure 4.3(b), tous les arcs en gras sont de type $\pi - acyclique$ (voir figure 4.3(d)).




(a) : $G' = (V',A')$ (b) : $G = (V,A)$ (c) : $\pi(G)$ (d) : $G - \pi(G)$

**Figure 4.3: Les sous-graphes $\pi(G)$ et $G - \pi(G)$ d'un graphe $G = (V,A)$.**

**Lemme 1.** *Étant donné un graphe G, un arc $e = (u,v)$ et un circuit c de G.*

1. *L'arc $e = (u,v)$ est acyclique dans G si et seulement si u et v appartient à deux différentes composante fortement connexes de G ;*
2. *Tout circuit c passant par un arc $\pi -$ acyclique de G, passe aussi par un $\pi -$ arc de G.*

**Démonstration :**

1. Il existe un circuit qui passe par $(u,v)$ si et seulement si $u \to v$ et $v \to u$, ce qui est vrai si et seulement si $u$ et $v$ appartiennent à la même composante fortement connexe.
2. Supposant qu'il existe un circuit $c$ qui traverse un arc $\pi - acyclique\ e$, mais il ne traverse aucun $\pi - arc$ de $G$. Soit $G' = G - \pi(G)$, tel que $\pi(G)$ est le sous-graphe de $G$ induit par les $\pi - arcs$ de $G$. Donc $c$ est un circuit de $G'$. Mais $c$ traverse $e$, qui est acyclique dans $G'$ contredisant la condition 1-1 du lemme 1.

D'après le lemme 1, les arcs non utilisés par les circuits du graphe $G$ et les arcs non utilisés par les circuits du graphe dans lequel on a retiré les circuits de type $(u,v,u)$ ne sont pas nécessaires dans le calcul d'un TCCM. Donc la suppression de ces arcs n'affecte pas le calcul d'un TCCM.

**Définition 10.** *L'opérateur* PIE *supprime les arcs acycliques et* $\pi - acyclique\ de\ G$.

**Exemple 19.** La figure 4.4, montre un exemple de réduction d'un graphe $G = (V,A)$ par l'application de l'opérateur PIE et les opérateurs de Levy et Low. Au début, on ne peut appliquer aucun opérateur de Levy et Low car, pour chaque sommet $u$ de $G$, $deg^-(u) > 1$ et $deg^+(u) > 1$ et il n'existe pas d'arc de type $e = (u,u)$. L'application de l'opérateur PIE sur le graphe initial permet de supprimer les arcs acycliques (arcs lignés) et les arcs $\pi$-acycliques (arcs pointillés). L'application de l'opérateur LOOP sur les sommets 0, 1, 5 et 7 donne le graphe vide, cela signifie que l'ensemble $S = \{0,1,5,7\}$ est un TCCM du graphe initial.

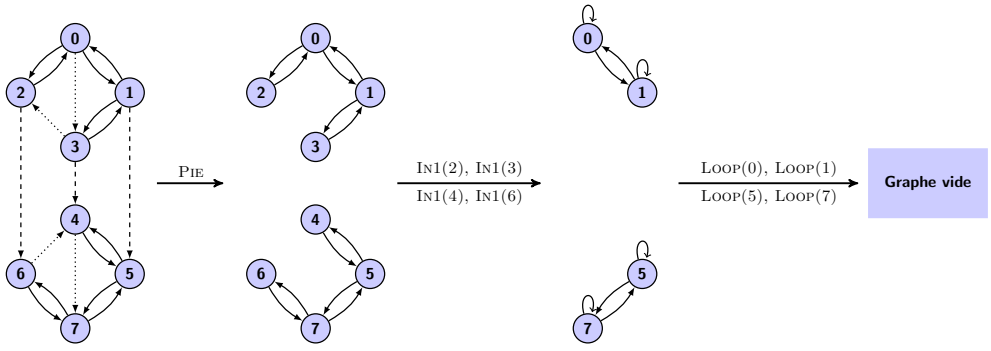



**Figure 4.4: Réduction d'un graphe en utilisant l'opérateur PIE et les opérateurs de Levy et Low.**

### *4.3.2 L'OPÉRATEUR* CORE

Soient $G = (V,A)$ un graphe orienté et $u \in V$ un sommet. On définit le voisinage du sommet $u$ par $\text{NEIGH}(u) = N(u) \cup \{u\}$. Soit $G' = (V',A')$ un sous-graphe de $G$. Si chaque sommet $u'$ de $V'$ n'a pas de boucle et quels que soient $u',v' \in V'$ tel que $u' \neq v'$, si $(u',v') \in A'$, alors, $G'$ est une *clique*. Un sommet $u$ est un *noyau* de $G' = (V',A')$ si $u \in V'$ et $\forall v \in V', (v,u) \in A'$. Dans la figure 4.5, le sous-graphe $G' = (V',A')$ tel que $V' = \{0,1,2\}$ et $A' = V' \times V'$ est une clique. Le sommet 0 est un noyau de $G'$.

**Lemme 2.** *Étant donné un graphe $G = (V,A)$. Soit $G' = (V',A')$ une clique de $G$ ayant $n = |V'|$ sommets. Alors, il existe au minimum $n-1$ sommets de $G'$ dans chaque TCCM de $G$.*

**Démonstration :** Soit $U$ un TCCM de $G$. S'il existe moins de $n-1$ sommet de $G'$ dans $U$ alors, le graphe $G-U$ contient au moins deux sommets $u$ et $v$ qui appartient à $V'$. Parce que $G'$ est une clique, il existe alors un $\pi - arc\ (u,v)$ dans $G-U$. Donc, il existe un circuit $(u,v,u)$ dans $G-U$. C'est une contradiction.

Si $\text{NEIGH}(u)$ forme une clique dans un graphe et $u$ un noyau de cette clique, on peut retirer $\text{NEIGH}(u)$ du graphe et on ajoute l'ensemble des sommets voisins de $u$ au TCCM courant.

**Définition 11.** *L'opérateur* CORE *retire les voisins d'un noyau d'une clique et les ajoute à la solution courante.*

**Exemple 20.** La Figure 4.5, montre un exemple de réduction d'un graphe $G = (V,A)$ par l'application de l'opérateur CORE et les opérateurs de Levy et Low. Au début, on ne peut appliquer aucun opérateur de Levy et Low, et l'opérateur PIE donne la composante fortement connexe composée du sommets $\{0,1,2\}$. La composante $\text{NEIGH}(0) = \{0,1,2\}$ forme une clique, le sommet $\{0\}$ est son noyau. L'opérateur CORE supprime $\text{NEIGH}(0)$ et ajoute $\{1,2\}$ à

la solution courante. L'opérateur LOOP(3) ajoute le sommet 3 à la solution, donc l'ensemble $S = \{1,2,3\}$ est un TCCM du graphe initial.

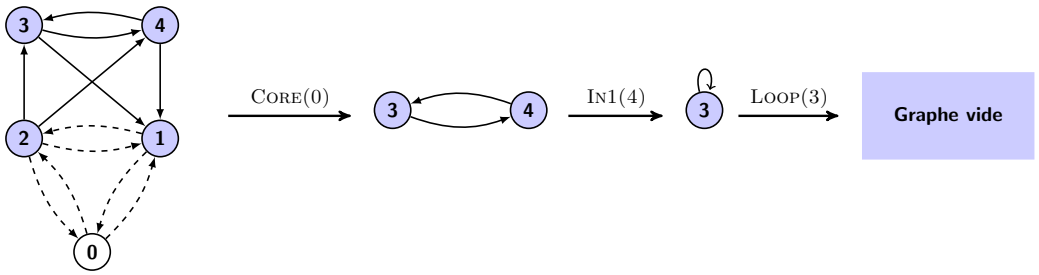



**Figure 4.5: Réduction d'un graphe en utilisant l'opérateur CORE et les opérateurs de Levy et Low.**

### *4.3.3 L'OPÉRATEUR* DOME

Soit $G = (V,A)$ un graphe orienté et $e = (u,v)$ un arc. Si $e = (u,v)$ est un $\pi - arc$, on appelle $u$ un $\pi-prédécesseur$ de $v$ et $v$ un $\pi - successeur$ de $u$. Dans le cas contraire, on dit que $u$ est un $non-\pi-prédécesseur$ de $v$ et $v$ est un $non-\pi-successeur$ de $u$. Soient $C_1 = (v_1, v_2, ..., v_n, v_1)$ et $C_2 = (u_1, u_2, ..., u_m, u_1)$ deux circuits. On dit que le circuit $C_2$ couvre le circuit $C_1$ si chaque sommet $v_i, i = 0, 1, ..., n$ de $C_1$ est traversé par $C_2$. Un circuit est *minimal* s'il ne couvre aucun autre circuit. Un arc $e = (u,v)$ de $G$ est dit *dominé* si tout $non-\pi-prédécesseur$ de $u$ est un *prédécesseur* de $v$ (l'arc en gras dans la figure 4.6), ou tout $non-\pi-successeur$ de $v$ est un *successeur* de $u$ (les arcs pointillés dans la figure 4.6).

**Lemme 3.** *Étant donné un graphe G, $e = (u,v)$ un arc dominé de G et c un circuit de G.*

1. *Pendant le calcul d'un TCCM de G, on a besoin de casser seulement les circuits minimaux ;*
2. *Supposons que c traverse e et c n'est pas minimal. Alors, il existe un circuit plus court $c'$ couvert par c et traversant e.*

**Démonstration :**

1. Pour tout circuit non minimal $c$, il existe un circuit minimal $c_m$ couvert par $c$. Casser le circuit $c_m$ va permettre de casser automatiquement $c$. Donc, on a besoin de casser seulement les circuits minimaux.
2. $c$ traverse $e = (u, v)$, i.e $c = (c_1, c_2, ..., c_i, u, v, c_{i+1}, c_{i+2}, ..., c_k, c_1)$ pour $i$, $k$ tel que $1 \leq i < k$. Il existe 3 cas possibles :
   (a) Si $c_i$ est un $\pi-$*prédécesseur* de $u$, le circuit $c' = (c_i, u, c_i)$ est couvert par $c$ ;
   (b) Si $c_{i+1}$ est un $\pi-$*successeur* de $v$, le circuit $c' = (c_{i+1}, v, c_{i+1})$ est couvert par $c$ ;
   (c) Si $c_i$ n'est pas un $\pi-$*prédécesseur* de $u$ et $c_{i+1}$ n'est pas un $\pi-$*successeur* de $v$. Dans ce cas, d'après la définition d'un arc dominé, soit $c_i$ prédécesseur de $v$ ou $c_{i+1}$ successeur de $u$. En premier cas, le circuit $(c_1, c_2, ..., c_i, v, c_{i+1}, c_{i+2}, ..., c_k, c_1)$ est couvert par $c$, dans le deuxième cas, le circuit $(c_1, c_2, ..., c_i, u, c_{i+1}, c_{i+2}, ..., c_k, c_1)$ est couvert par $c$.

On conclut du lemme 3 qu'uniquement les arcs non dominés sont nécessaires dans le calcul d'un TCCM.

**Définition 12.** *L'opérateur* DOME *supprime les arcs dominés.*

**Exemple 21.** La Figure 4.6 montre un exemple de réduction d'un graphe $G = (V, A)$ par l'application de l'opérateur DOME et les opérateurs de Levy et Low. Les opérateurs de Levy et Low et les opérateurs PIE et CORE ne sont pas applicables sur le graphe initial. On commence à appliquer l'opérateur DOME sur le graphe initial, cet opérateur supprime les arcs dominés, puis on continue par l'application des opérateurs de Levy et Low. L'ensemble $S = \{2, 4\}$ est un TCCM du graphe initial.

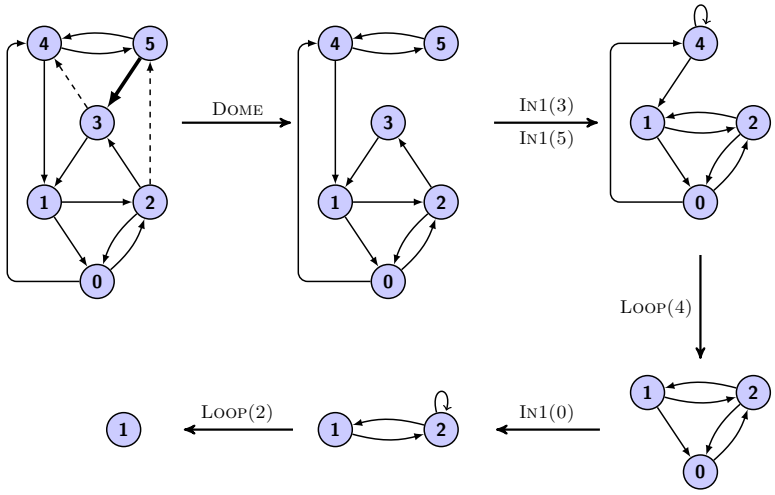



**Figure 4.6: Réduction d'un graphe en utilisant l'opérateur DOME et les opérateurs de Levy et Low.**

**Théorème 15.** *Soit $G = (V,A)$ un graphe orienté, $u \in V$ un sommet. Alors*

1. *Si G est contracté en $G'$ par l'application de l'opérateur* PIE *ou* DOME *et $S'$ est un TCCM de $G'$ alors $S'$ est un TCCM de G ;*

2. *Si G est contracté en $G'$ par l'application de l'opérateur* CORE. *Si le sommet u est un noyau de la clique retirée par* CORE *du graphe G et $S'$ est un TCCM de $G'$, alors $S' \cup N(u)$ est un TCCM de G ;*

**Démonstration :**

1. Pour l'opérateur PIE. Supposons $U \in TCCM(G)$. D'abord, on doit montrer que $U$ est un TC de $G'$. Soient $B$ l'ensemble des arcs acyclique de $G$, $C$ l'ensemble des arcs $\pi$-acycliques de $G$ et $P = B \cup C$. i.e. $P$ est l'ensemble des arc retirés par PIE. Soit $c$ un circuit de $G'$ et $A(c)$ les arcs de $c$. Si $P \cap A(c) = \varnothing$ alors $c$ est un circuit de $G$ donc il est couvert par $U$. Sinon, supposons qu'il existe un arc $(u,v) \in P \cap A(c)$. Il est clair que $(u,v) \notin B$ tant qu'il n' y a pas de circuit qui traverse un arc acyclique. Par conséquent $(u,v)$ est $\pi - acyclique$ donc $c$ passe par un arc $(u,v)$ d'après le lemme 1-2. Mais $(u,v,u)$ est un

circuit de $G$ et de $G'$ et il est couvert par $U$, ce qui implique que $c$ est aussi couvert par $U$. Maintenant, on doit montrer que $U$ est minimal. Par contradiction, supposons qu'il existe $U' \in TCCM(G')$ tel que $|U'| < |U|$. Comme $TCCM(G') \subseteq TCCM(G)$, on a $U' \in TCCM(G)$, contredisant la minimalité de $U$ dans $G$. Pour l'opérateur DOME, supposons que $U \in TCCM(G)$. On prouve que $U$ est un TC de $G'$. Soit $c$ un circuit de $G'$. Si le circuit $c$ ne traverse aucun arc dominé, alors il est aussi un circuit de $G$, et donc il est couvert par $U$. Sinon, supposons que $c$ traverse un arc dominé $e$. Alors il existe un circuit $c'$ plus court et couvert par $c$ traverse $e$ d'après le lemme 3-2. Alors $c'$ est aussi un circuit de $G$, donc il est couvert par $U$. La minimalité est triviale.

2. Supposons que $U \in TCCM(G)$. Soit $D$ la clique de $G$ retiré par CORE. On montre que $U$ est un TC de $G' \cup D$. Soit $c$ un circuit de $G' \cup D$. Si $c \notin D$, alors $c$ est un circuit de $G'$ et il est aussi un circuit de $G$ donc il est couvert par $U$. Sinon, d'après le lemme 2 pour casser tous les circuits de $D$, juste un seul sommet peut être resté et tous les arcs doivent être enlevés. Tant que tous les arcs incidents d'un noyau $u$ de $D$ sont dans $D$, l'ajout de $N(u)$ à $U$ casse tout circuit $c$ de $D$ donc $c$ est couvert par $U$. La minimalité de $U$ est garantie par le lemme 2.

Lin et Jou ont défini une classe de graphes orientés notés DOME-*contractible*. Plus précisément, on dit qu'un graphe orienté est DOME-*contractible* si l'application successive des huit opérateurs résulte en un graphe vide. L'ensemble des sommets ajoutés par les opérateurs LOOP et CORE forment un TCCM du graphe initial.

**Théorème 16.** *Soit $G = (V,A)$ un graphe orienté* DOME-*contractible avec n sommets et m arcs. Alors on peut calculer un TCCM de G en temps polynomial égal à $O(m^2 n)$.*

**Démonstration :** Les opérateurs IN0, OUT0, IN1, OUT1 et LOOP ont un temps égal à $O(m+n)$. L'opérateur PIE est aussi linéaire ($O(m+n)$). L'opérateur CORE s'exécute en temps égal

à $O(m+n\log n)$ et DOME prend un temps égal à $O(m.n)$. A chaque étape de réduction au moins un sommet ou un arc est supprimé, la complexité globale est bornée par $O((m.(mn))$ = $O(m^2n)$.

Lin et Jou ont présenté un algorithme exact basé sur la technique séparation et évaluation permettant de calculer un TCCM en temps polynomial pour n'importe quel graphe orienté (voir l'algorithme 6).

L'algorithme proposé par Lin et Jou peut être résumé comme suit :

1. Réduire le graphe autant que possible en utilisant les huit opérateurs ;
2. S'il y a plus d'une composante connexe, diviser le problème en sous-problèmes selon ces composantes.
3. Calculer une borne supérieure $U$ pour le graphe en cours ;
4. Si $U$ est la meilleure solution trouvée jusqu'à présent, la sauvegarder ;
5. Calculer une borne inférieure $L$ du problème pour le graphe en cours en suivant ces étapes :
   (a) Si le graphe est vide, arrêter ;
   (b) Sinon, soit $c$ un circuit de longueur minimal du graphe ;
   (c) Supprimer $c$ du graphe et incrémenter $L$ de 1 ;
   (d) Appliquer les huit opérateurs de contractions autant que possible ;
   (e) Revenir à l'étape 5(a).
6. Si la borne inférieure indique que la solution actuelle ne pourra pas être étendue à une meilleure solution, ne pas poursuivre avec cette solution ;
7. Sinon, choisir un sommet $u$ et effectuer un branchement en deux cas : soit on inclut le sommet $u$ dans la solution, soit on l'exclut.

**Algorithme 6** L'algorithme de Lin et Jou pour calculer un TCCM

```
1:  fonction OneMFVS(G : graphe orienté, S,B :sommets, p,L :entiers) : sommets
2:      Appliquer les opérateurs de réduction sur G
3:      Soit S' l'ensemble de sommets enlevés par Loop et Core
4:      S ← S ∪ S'
5:      𝒞 ← ConnectedComponents(G)
6:      si |𝒞| > 1 alors
7:          pour tout C = (A', V') ∈ 𝒞 faire
8:              S ← S ∪ OneMFVS(C, 0, V', 0, 0)
9:          fin pour
10:         retourner S
11:     sinon
12:         lowerbound ← LowerBound(G)
13:         si L = 0 alors
14:             p ← lowerbound
15:             B ← S ∪ ApproximateSolution(G)
16:         fin si
17:         si |S| + lowerbound > |B| alors
18:             retourner B
19:         sinon
20:             si G = ∅ alors
21:                 retourner S
22:             fin si
23:         fin si
24:         v ← MaxDegreeVertex(G)
25:         S1 ← OneMFVS(G.Delete(v), S ∪ {v}, B, p, L + 1)
26:         si |S1| < |B| alors
27:             B ← S1
28:         fin si
29:         si p = |B| alors
30:             retourner B
31:         fin si
32:         S2 ← OneMFVS(G.Vanish(v), S, B, p, L + 1)
33:         si |S2| < |B| alors
34:             B ← S2
35:         fin si
36:         retourner B
37:     fin si
38: fin fonction
```

## 4.4 CONCLUSION

Le problème TCCM est largement étudié depuis son introduction et plusieurs approches ont été proposées pour le résoudre. Dans ce chapitre, on a présenté ce problème dans son cadre général et on a détaillé l'approche définie par Levy et Low, et étendue par Lin et Jou, qui se base sur le principe de réduction du graphe par la suppression des sommets et des arcs inutiles. Cette approche a permis de concevoir des algorithmes polynomiaux pour calculer un TCCM dans un graphe orienté et permet également de concevoir un algorithme énumérant l'ensemble de tous les TCCM comme nous le décrivons dans le chapitre suivant.

# CHAPITRE 5

# ÉNUMÉRATION DES TRANSVERSAUX À L'AIDE D'UN ARBRE ET/OU

Une question naturelle issue du problème TCCM est la suivante : combien de transversaux de circuits de cardinalité minimale contient un graphe orienté de $n$ sommets ? Clairement, il peut y en avoir un nombre exponentiel par rapport au nombre de sommets. Le graphe orienté $G = (V,A)$ à $n = |V|$ sommets de la figure 5.1 possède $2^{n/2}$ TCCM différents, chacun d'eux possède $n/2$ sommets. En effet, pour $i = 1,3,5,\ldots,n-3,n-1$, on peut inclure le sommet $v_i$ ou le sommet $v_{i+1}$ dans la solution de façon indépendante.

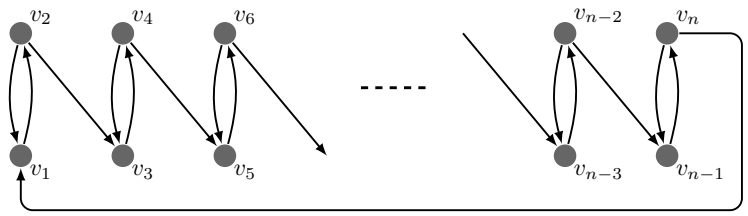



**Figure 5.1: Un graphe ayant $2^{n/2}$ transversaux de circuits minimaux.**

L'énumération de tous les TCCM d'un graphe orienté $G = (V,A)$ est un problème NP-difficile. En 2002, Schwikowski et Speckenmeyer [Schwikowski et Speckenmeyer, 2002] ont présenté un algorithme permettant d'énumérer tous les TCCM dans un temps polynomial. En 2007, Formin et al. [Fomin et al., 2007] ont démontré qu'il existe des graphes contenant un nombre exponentiel de TCCM et ont proposé un algorithme pour les énumérer en temps $O(1.8638^n)$.

## 5.1 ALGORITHME

Lors du traitement de grandes familles d'ensembles de solutions, il devient rapidement impossible de stocker explicitement chacun des ensembles dans la mémoire. Par conséquent, on recourt souvent à des structures de données qui représentent implicitement ces ensembles. Dans notre algorithme, nous utilisons les arbres et/ou qui sont vus dans le chapitre précédent (voir chapitre 3) pour stocker implicitement la famille des TCCM.

Notre manière de faire pour énumérer l'ensemble de TCCM d'un graphe orienté consiste à diviser le problème en deux phases principales. La première phase permet de construire l'arbre et/ou représentant l'ensemble de TCCM du graphe et la deuxième phase se consacre au calcul de la famille de solutions (TCCM) à partir de l'arbre construit.

Vu que les DBD sont des structures populaires et très utilisées puisqu'ils offrent différents avantages que les arbres et/ou n'offrent pas (unicité de la représentation, compacité, etc.), nous ajoutons une autre phase qui permet de convertir l'arbre construit en un DBD et vice versa.

Dans ce qu'il vient, nous décrivons notre algorithme et la manière de faire pour construire un arbre et/ou et énumérer la famille des TCCM d'un graphe orienté.

### *5.1.1 CONSTRUCTION DE L'ARBRE ET/OU REPRÉSENTANT LA FAMILLE DE TCCM*

Cette section est consacrée à la description de l'algorithme qui calcule l'arbre et/ou représentant l'ensemble des TCCM d'un graphe orienté $G = (V, A)$. Tout d'abord, nous commençons par l'introduction de quelques définitions utiles dans la description de l'algorithme de construction.

**Définition 13.** *Soit $G = (V, A)$ un graphe orienté, $u \in V$. On dit que u est*

1. *essentiel s'il appartient à tous les TCCM de G ;*

2. *inutile s'il n'appartient à aucun TCCM de G.*

Les sommets essentiels et les sommets inutiles peuvent être contractés sans compromettre l'espace des TCCM.

**Lemme 4.** *Soit $G = (V,A)$ un graphe orienté et $U \subseteq V$*

1. *Pour tout sommet essentiel u, U est un TCCM de G si et seulement si $U - \{u\}$ est un TCCM de $G$.*DELETE$(u)$*;*
2. *Pour tout sommet inutile u, U est un TCCM de G si et seulement si U est un TCCM de $G$.*VANISH$(u)$.

**Démonstration :**

1. Soit $U \in TCCM(G)$, nous montrons que $U - \{u\}$ est un TCCM de $G$.DELETE$(u)$. Soit $c$ un circuit de $G$. Si $c$ est un circuit de $G$.DELETE$(u)$ donc, il est couvert par $U - \{u\}$ ; sinon, pour chaque TCCM de $G$, $c$ est couvert par $u$ parce que ce sommet est essentiel.
2. Soit $U \in TCCM(G)$, nous montrons que $U$ est un TCCM de $G$.VANISH$(u)$. Soit $c$ un circuit de $G$.VANISH$(u)$. Si $c$ est un circuit de $G$, il est donc couvert par $U$. Si $c$ n'est pas un circuit de $G$, il est donc forcément issu de l'application de l'opération VANISH$(u)$ ; et parce que $u$ n'appartient à aucun TCCM (c'est-à-dire, tout circuit $c$ passant par $u$ est couvert) donc $U$ est un TCCM de $G$.VANISH$(u)$.

Un sommet possédant une boucle (self-loop) est nécessairement essentiel, tandis qu'un sommet supprimé par l'opérateur IN0 ou OUT0 est nécessairement inutile (voir la figure 5.2). Il n'est pas difficile de vérifier si un sommet est essentiel ou inutile (voir l'algorithme 8).

**Définition 14.** *Soit $G = (V,A)$ un graphe orienté, U un TCCM de G et $u, v \in V$. On dit que*

1. *u est dominé par v, noté $u \preceq v$, si $u \in U$ implique que $(U - \{u\}) \cup \{v\}$ est un TCCM de G ;*

2. *u et v sont équivalents, noté $u \equiv v$, si $u \preceq v$ et $v \preceq u$.*

Il est facile de voir si deux sommets $u$ et $v$ sont équivalents, c'est-à-dire pour chaque TCCM $U$ de $G$, vérifier si $u \in U$ implique que $v \notin U$. Si $u$ et $v$ appartient à $U$, et parce que chaque circuit $c$ passant par $u$ passe aussi par $v$ donc $c$ est couvert par $U$ même si on enlève juste l'un des deux sommets $u$ ou $v$. Si on enlève les deux sommets, c'est une contradiction de la minimalité de $U$, donc $u$ et $v$ ne peuvent pas appartenir les deux à $U$. Ainsi, pour chaque $U \in TCCM(G)$, si $u \in U$ et $u \equiv v$, on peut remplacer $u$ par $v$ dans $U$ (voir la figure 5.2). L'intérêt des sommets équivalents est qu'ils peuvent être fusionnés en un seul sommet. On a juste besoin de garder une trace des classes d'équivalence lors de la construction de l'arbre.

**Exemple 22.** Dans la figure 5.2, les sommets $\{2,10\}$ sont essentiels. Les sommets $\{1,3,7\}$ sont inutiles. Les trois sommets $\{4,5,6\}$ sont équivalents. Les deux sommets 8 et 9 sont équivalents et chacun d'eux domine l'autre.

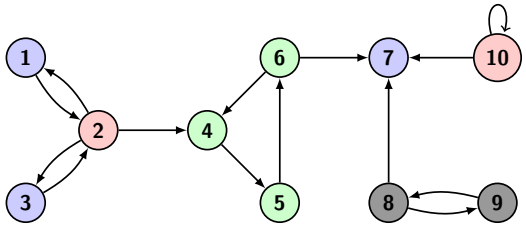



**Figure 5.2: Exemple de sommets inutiles, essentiels, dominés et équivalents.**

**Lemme 5.** *Soit $G = (V,A)$ un graphe orienté et $u,v \in V$.*

1. *Si v est l'unique successeur de u, alors $u \preceq v$ ;*
2. *Si u est l'unique prédécesseur de v, alors $v \preceq u$ ;*
3. *Si u est l'unique prédécesseur de v et v est l'unique successeur de u, alors $u \equiv v$.*

**Démonstration :** Soit $U \in TCCM(G)$ et $u \in U$.

1. Parce que chaque circuit passant par $u$ passe aussi par $v$ alors, on peut remplacer $u$ par $v$ dans $U$, donc $(U - \{u\}) \cup \{v\}$ est un $TCCM(G)$. Par conséquent, $u \leq v$.
2. C'est l'énoncé symétrique de (1).
3. Suit de (1) et (2).

D'autres propriétés reliant la relation $\leq$ avec les concepts de sommets essentiels et inutiles accélèrent l'algorithme de construction de l'arbre, en particulier lorsque les opérateurs IN1 et OUT1 sont appliqués plusieurs fois. En effet, on peut déduire que certains sommets sont inutiles sans avoir à les vérifier explicitement.

**Lemme 6.** *Soit $G = (V,A)$ un graphe orienté et $u, v \in V$.*

1. *Si $u \leq v$ et $v$ est essentiel, alors $u$ est inutile ;*
2. *Si $u \leq v$ et $v$ est inutile, alors $u$ est inutile ;*

**Démonstration :** Soit $U \in TCCM(G)$ et $u \in U$.

1. Si $v$ est essentiel donc $v \in U$ et parce que chaque circuit $c$ passant par $u$ passe aussi par $v$, alors $U \cup \{u\}$ est un TC de $G$. Contradiction de la minimalité de $U$.
2. La définition de $\leq$ implique que $(U - \{u\}) \cup \{v\} \in TCCM(G)$, contradiction à la supposition que $v$ est inutile.

On est maintenant prêt à décrire notre algorithme de construction. Il prend en entrée un graphe orienté et renvoie un arbre et/ou représentant l'ensemble de TCCM. Dans l'algorithme 9, la fonction NEWNODE() permet de créer un nouveau nœud qui est peut être de type *et*, de type *ou* ou une feuille. La stratégie de l'algorithme est comme suit :

1. Identifier les paires de sommets équivalents et les fusionner : Pour calculer les ensembles des sommets équivalents d'un graphe $G = (V,A)$, on commence par la construction du graphe représentant les paires de sommets dominés tel qu'il est décrit dans la fonction DOMINANCEDIGRAPH de l'algorithme 7.

**Algorithme 7** Calcul du paires de sommets équivalents

```
1:  fonction DOMINANCEDIGRAPH(G : graphe orienté) : graphe
2:      pour tout u ∈ A faire
3:          si deg⁻(u) = 1 et v est le prédécesseur de u alors
4:              G'.ADDEDGE(v,u)
5:          fin si
6:          si deg⁺(u) = 1 et w est le successeur de u alors
7:              G'.ADDEDGE(w,u)
8:          fin si
9:      fin pour
10:     retourner G'
11: fin fonction

12: fonction EQUIVALENTSETS(G : graphe orienté) : ensemble
13:     dg ←DOMINANCEDIGRAPH(G)
14:     𝒞 ←STRONGLYCONNECTEDCOMPONENTS(G)
15:     pour tout C ∈ 𝒞 faire
16:         si |C| >= 2 alors
17:             Ajouter C à S
18:         fin si
19:     fin pour
20:     retourner S
21: fin fonction
```

Les sommets de chaque composante fortement connexe du graphe construit par la fonction DOMINANCEDIGRAPH constitue un ensemble de sommets équivalents. Si la classe d'équivalence du sommet $u$ est vide, on crée une feuille contenant $u$ ; sinon, on crée un sous-arbre enraciné par $\vee$ et les feuilles sont les sommets équivalents à $u$.

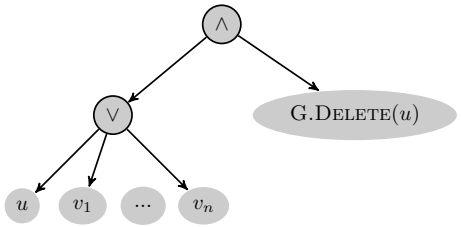



**Figure 5.3: Représentation des sommets équivalents au sommet $u$ par l'arbre et/ou.**

Dans la figure 5.3, les sommets $v_1, v_2, \ldots, v_n$ sont des sommets équivalents à $u$. On

enlève $u$ du graphe $G$ et on continue à appliquer l'algorithme sur $G$.DELETE$(u)$. Il est important de garder une trace des sommets équivalents pour les étapes à venir.

2. Identifier les sommets inutiles et les sommets essentiels : Pour calculer l'ensemble des sommets inutiles et les sommets essentiels d'un graphe $G = (V, A)$, on calcule un TCCM $U$ du graphe ; puis pour chaque sommet $u$ de $U$, on vérifie s'il est essentiel en appliquant la fonction ISESSENTIAL ; et pour chaque sommet $u$ de $V$ qui n'appartient pas à $U$, on vérifie s'il est inutile en appliquant la fonction ISINUTIL (voir l'algorithme 8).

**Algorithme 8** Vérifier si un sommet est inutile ou essentiel

```
1: fonction ISINUTIL(G : graphe orienté, u :sommet) : booléen
2:     si deg⁻(u) = 0 ou deg⁺(u) = 0 alors
3:         retourner Vrai
4:     fin si
5:     G' ← G.DELETE(u)
6:     retourner |ONEMFVS(G)| = |ONEMFVS(G')|
7: fin fonction

8: fonction ISESSENTIAL(G : graphe orienté, u :sommet) : booléen
9:     si (u,u) ∈ A alors
10:        retourner Vrai
11:    fin si
12:    G' ← G.VANISH(u)
13:    retourner |ONEMFVS(G')| > |ONEMFVS(G)|
14: fin fonction
```

3. Appliquer l'opérateur VANISH sur les sommets inutiles ;

4. Appliquer les opérateurs PIE et DOME ;

5. Éliminer les sommets essentiels et les ajouter à l'arbre et/ou. Comme on a déjà vu que le sous-ensemble des sommets essentiels appartient à chaque TCCM de $G$, donc ces sommets doivent être rassemblés par un nœud *et* dans l'arbre et/ou. La figure 5.4 montre comment représenter le sous-ensemble de sommets essentiels dans l'arbre et/ou. $E = \{e_1, e_2, ..., e_k\}$ est le sous-ensemble de sommets essentiels ;

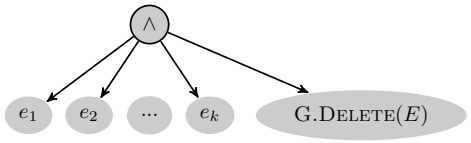



**Figure 5.4: Représentation des sommets essentiels dans l'arbre et/ou.**

6. Si le graphe a plus d'une composante connexe, diviser le problème selon ces composantes. Les arbres représentant les TCCM de chaque composante sont rassemblés par un nœud de type *et* (voir la figure 5.5).

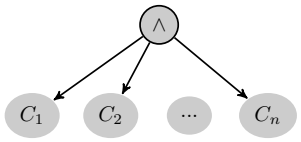



**Figure 5.5: Division du problème selon ses composantes connexes.**

7. Sinon, choisir un sommet (celui qui a le degré maximum) et faire la séparation selon ce sommet (ce sommet doit être inclus ou exclu de la solution) (voir la figure 5.6).

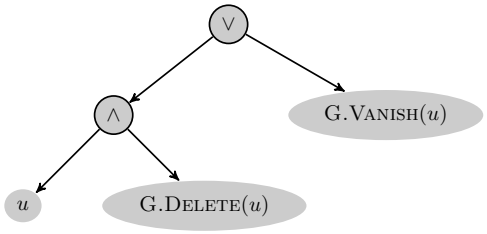



**Figure 5.6: Séparation et évaluation selon un sommet bien choisi.**

**Algorithme 9** Calcul d'un arbre et/ou représentant tous les transversaux de circuits minimaux

1: **fonction** GETTREE($G$ : graphe orienté, *id* :entier) : arbre et/ou
2: Fusionner toutes les paires de sommets équivalents
3: Appliquer l'opérateur VANISH sur tous les sommets inutiles
4: Appliquer les opérateur PIE et DOME
5: Soit $E$ l'ensemble des sommets essentiels
6: **si** $|E| > 0$ **alors** ▷ Supprimer les sommets essentiels et les ajouter à l'arbre

```
7:          id ← id + 1
8:          x ←NewNode(id, ∧, nul)
9:          pour tout u ∈ E faire
10:             id ← id + 1
11:             Ajouter NewNode(id, u.value) à x.children
12:         fin pour
13:         G′ ← G.Delete(E)
14:         Ajouter GetTree(G′) à x.children
15:     sinon
16:         𝒞 ←ConnectedComponents(G)
17:         si |𝒞| > 1 alors      ▷ Effectuer un branchement selon les composantes connexes
18:             id ← id + 1
19:             x ←NewNode(id, ∨, nul)
20:             pour tout C ∈ 𝒞 faire
21:                 Ajouter GetTree(C) à x.children
22:             fin pour
23:         sinon   ▷ Effectuer un branchement en incluant/excluant un sommet quelconque
24:             Soit u un sommet quelconque
25:             id ← id + 1
26:             x ←NewNode(id, ∨, nul)
27:             id ← id + 1
28:             y ←NewNode(id, ∧, nul)
29:             id ← id + 1
30:             y.children ←NewNode(id, u.value)
31:             G′ ← G.Delete(u)
32:             Ajouter GetTree(G′) à y.children
33:             Ajouter y à x.children
34:             G′ ← G.Vanish(u)
35:             Ajouter GetTree(G′) à x.children
36:         fin si
37:     fin si
38:     retourner x
39: fin fonction
```

### *5.1.2 CALCUL DE LA FAMILLE DES TCCM À PARTIR DE L'ARBRE ET/OU*

La deuxième phase de notre application consiste à calculer la famille de TCCM (solutions) du graphe $G$ à partir de l'arbre et/ou. La section 3.2.1 et l'algorithme 10 décrivent comment calculer cette famille.

**Algorithme 10** Énumération de la famille de solutions à partir d'un arbre et/ou

```
1: fonction AllMFVS(T : arbre et/ou) : List<List<value>>
2:     retourner AllMFVS(T.root)
3: fin fonction

4: fonction AllMFVS(v : Leaf) : List<List<value>>          ▷ Si le nœud est une feuille
5:     Lsol ← nul
6:     Ajouter v.value à Lsol
7:     retourner Lsol
8: fin fonction

9: fonction AllMFVS(v : OrNode) : List<List<value>>        ▷ Si le nœud est de type ou
10:    pour tout w ∈ v.children faire
11:        sc ←AllMFVS(w)
12:        pour tout s ∈ sc faire
13:            Ajouter s à Lsol
14:        fin pour
15:    fin pour
16:    retourner Lsol
17: fin fonction

18: fonction AllMFVS(v : AndNode) : List<List<value>>      ▷ Si le nœud est de type et
19:    pour tout w ∈ v.children faire
20:        sc ←AllMFVS(w), sol ← Lsol, Lsol ← nul
21:        si sol ≠ nul alors
22:            pour tout s1 ∈ sol faire
23:                pour tout s2 ∈ sc faire
24:                    Ajouter s1 + s2 à Lsol
25:                fin pour
26:            fin pour
27:        sinon
28:            pour tout s2 ∈ sc faire
29:                Ajouter s2 à Lsol
30:            fin pour
31:        fin si
32:    fin pour
33:    retourner Lsol
34: fin fonction
```

Les principaux avantages d'utiliser les arbres et/ou au lieu d'utiliser les DBD sont la compacité et la facilité de construction de la structure des arbres et/ou.

En utilisant les arbres et/ou, on a pas besoin de chercher l'ordre optimal qui minimise le mieux la taille de la structure et qui est un problème NP-complet comme nous avons vu dans la section 2.5, et on a pas également besoin de faire la réduction de la structure car l'arbre construit est déjà optimal et aplati. Ainsi que la décomposition de l'ensemble de solution sur les variables est très simple par rapport aux DBD surtout quand le nombre de variables et supérieur à 2. Dans la figure 5.7, le DBD représentant l'ensemble des TCCM du graphe est composé de 9 nœuds et cela en utilisant l'ordre optimal qui est $c < d < a < b$ (le DBD est plus grand si on utilise un ordre autre que celui-ci) tandis que l'arbre est composé de 6 nœuds.

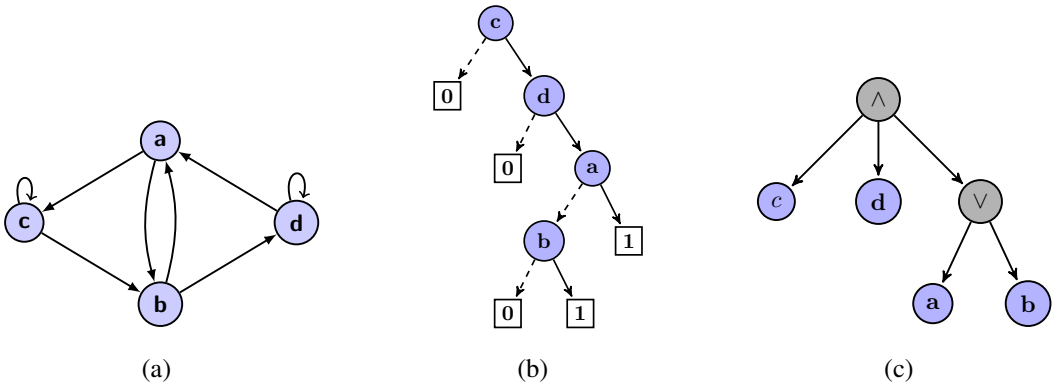



**Figure 5.7: Comparaison entre le DBD et l'arbre et/ou représentant la famille des TCCM d'un graphe. (a) Un graphe orienté. (b) Le DBD représentant les TCCM du graphe. (c) L'arbre et/ou représentant les TCCM du graphe.**

Les procédures de conversion des arbres et/ou en DBD et des DBD en arbres et/ou ont été décrites et détaillées dans les sections 3.3.1, 3.3.2, pour plus de détails voir aussi les algorithmes 3, 4 et 5.

## 5.2 IMPLÉMENTATION

Pour évaluer les différents algorithmes vus dans ce mémoire, notre choix s'est porté sur les langages de programmation **Java** et **Python**. L'algorithme qui permet de calculer l'arbre et/ou représentant la famille des TCCM a été implémenté en **Python 2.7**. Naturellement, nous avons choisi le langage Python parce qu'il offre des modules facilitant la créations, l'utilisation, l'affichage et la manipulation des graphes. Grâce aux méthodes et fonctions offertes par le module **networkx-1.10-py2.7** de Python, nous avons pu implémenté les différents opérations de calcul et de réduction de notre algorithme.

Le reste de l'application a été implémenté en utilisant le langage Java. Celui-ci a permis de mettre en place des interfaces homme-machine pour afficher nos résultats et intégrer les différents composants du système. Pour pouvoir sauvegarder, ouvrir et dessiner les arbres et/ou et les DBD créés (.dot, .txt, .png, etc) nous avons installé **Graphviz2.38** et l'avons intégré dans Java en ajoutant une classe **GraphViz.java**. Cette classe permet de gérer les interactions entre notre application Java et le programme **dot** de Graphviz2.38.

Dans notre application, il y a trois structures de données qui interviennent (les graphes orientés, les DBD et les arbres et/ou). La création des DBD et des arbres et/ou a été implémentée en Java, ainsi que les différentes opérations de calcul et de conversion permettant l'énumération et le stockage de la famille de TCCM.

La figure 5.8 représente le diagramme des classes intervenant dans notre application et les différentes relations et interactions entre elles, ainsi que les attributs et les méthodes de chaque classe.

Un arbre et/ou $T$ est défini par sa racine $T.root$ qui est de type $TNode$. Un TNode $v$ est de type $\text{ANDNODE}(id, \wedge, children)$, $\text{ORNODE}(id, \vee, children)$ ou $\text{LEAF}(id, value)$ tel que $id$

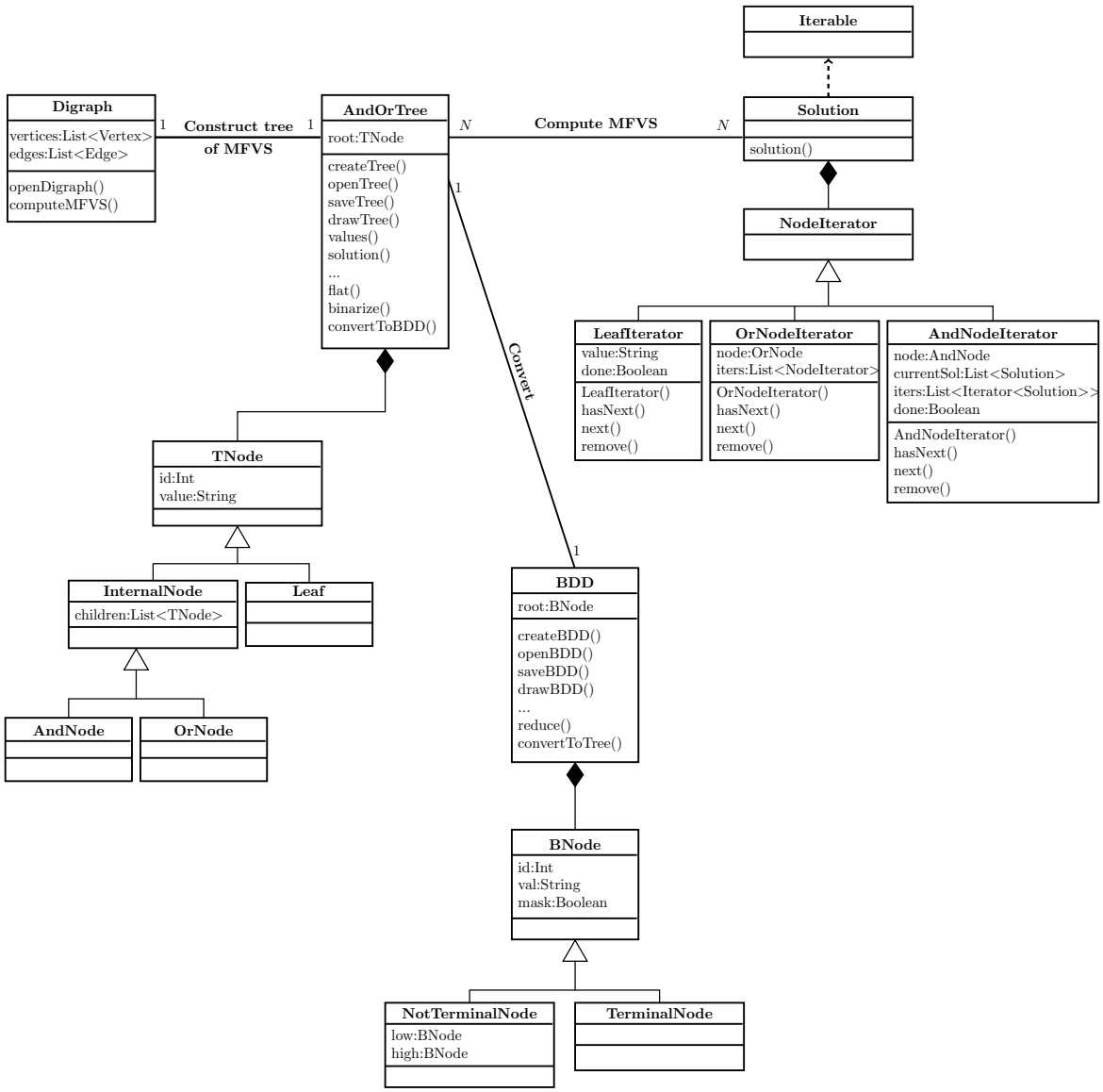



**Figure 5.8: Diagramme de classes**

est l'identifiant de $v$, *value* est la valeur (CONTENT) du $v$ et *children* est la liste des nœuds enfants de $v$ qui sont de type TNode. Un DBD $D$ est défini par sa racine $D.root$ qui est de type *BNode*. Un BNode $w$ est de type NOTTERMINALNODE$(id, val, mask, low, high)$ ou TERMINALNODE$(id, val, mask)$ tel que $id$ est l'identifiant de $w$, $val$ est la variable de $w$, $mask$ un booléen utilisé pour désigner si le nœud est déjà parcouru pendant le calcul d'une fonction

ou non, *low* (*high*) est le fils bas (haut) d'un nœud non terminal qui est de type BNode.

La famille des TCCM de $G$ est une liste de solutions, chaque solution (TCCM) est une liste de feuilles de $T$ et implémente l'interface *Iterable* de Java. Pour l'énumération des solutions, nous avons implémenté des itérateurs pour chaque type de nœud de l'arbre. Cette façon de faire permet de garder juste une solution (la solution courante) dans la mémoire. Pour calculer la solution suivante, l'application teste les nœuds de la solution courante, si un nœud a une autre solution (hasNext()=vrai) alors on rend cette solution sinon on génère un nouvel itérateur. Nous avons introduit cette notion d'itérateurs pour ne pas occuper un grand espace mémoire par la famille de solution surtout quand elle est grande. Cette manière de faire permet de calculer rapidement la famille des TCCM même si le graphe est de grande taille.

L'application a été testée sur un ensemble de graphes générés aléatoirement. Les résultats des testes effectues seront affichés dans l'article publié pendant la session d'hiver 2016.

Dans ce qui suite une liste de figures capturés de notre applications. La figure 5.9 est la capture d'écran de notre application montrant l'arbre et/ou représentant la famille des TCCM du graphe de la figure 2. L'arbre et/ou de la figure 5.10(b) est l'arbre représentant la famille des TCCM du graphe de la figure 5.10(a) produit par notre application. Cet arbre utilise 19 nœuds pour représenter une famille de 9 TCCM, chaque TCCM contient 4 nœuds. La figure 5.11 est la capture d'écran de notre application montrant un exemple de conversion d'un arbre et/ou en un DBD. La figure 5.12 est une capture d'écran de notre application montrant un exemple de conversion d'un DBD en un arbre et/ou.

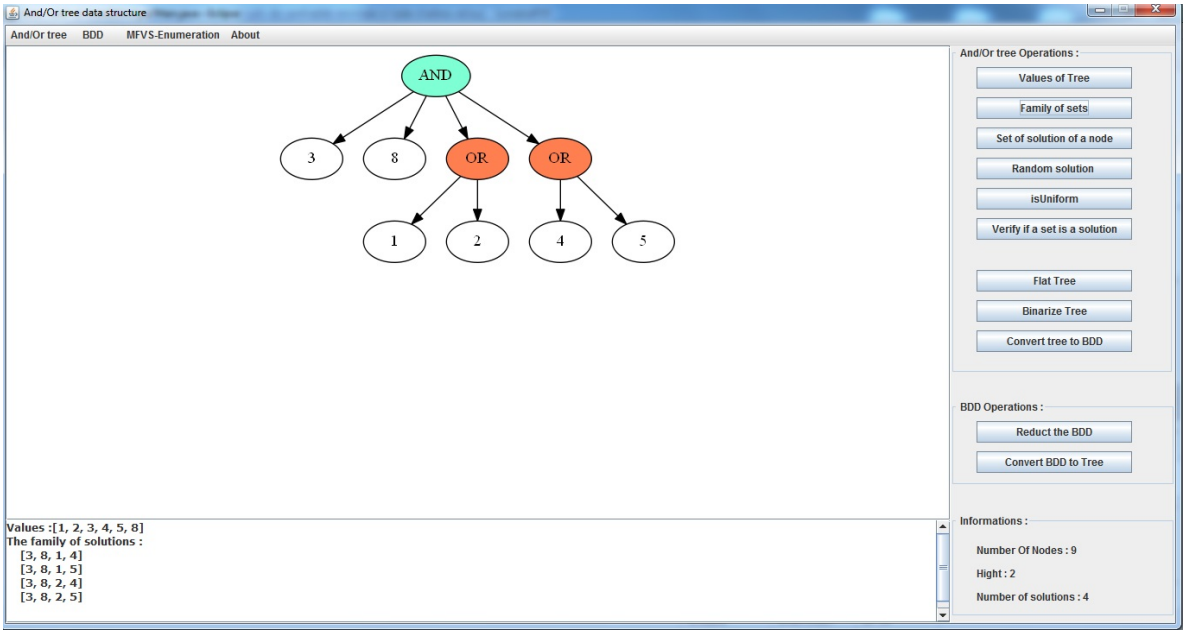
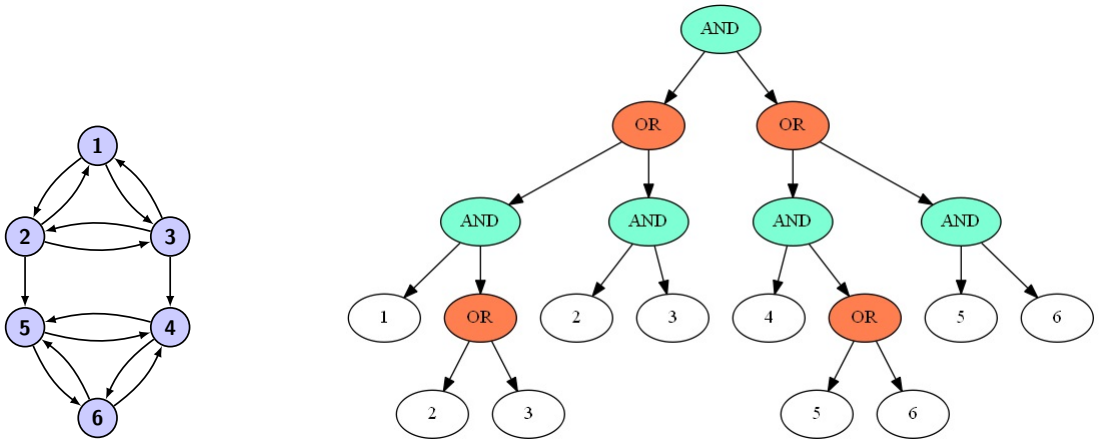



**Figure 5.9: L'arbre représentant la famille des TCCM du graphe de la figure 2 généré par notre application.**




(a) (b)

**Figure 5.10: L'arbre et/ou produit par notre application représentant la famille des TCCM d'un graphe. (a) Un graphe orienté. (b) Un arbre et/ou représentant la famille des TCCM de ce graphe.**

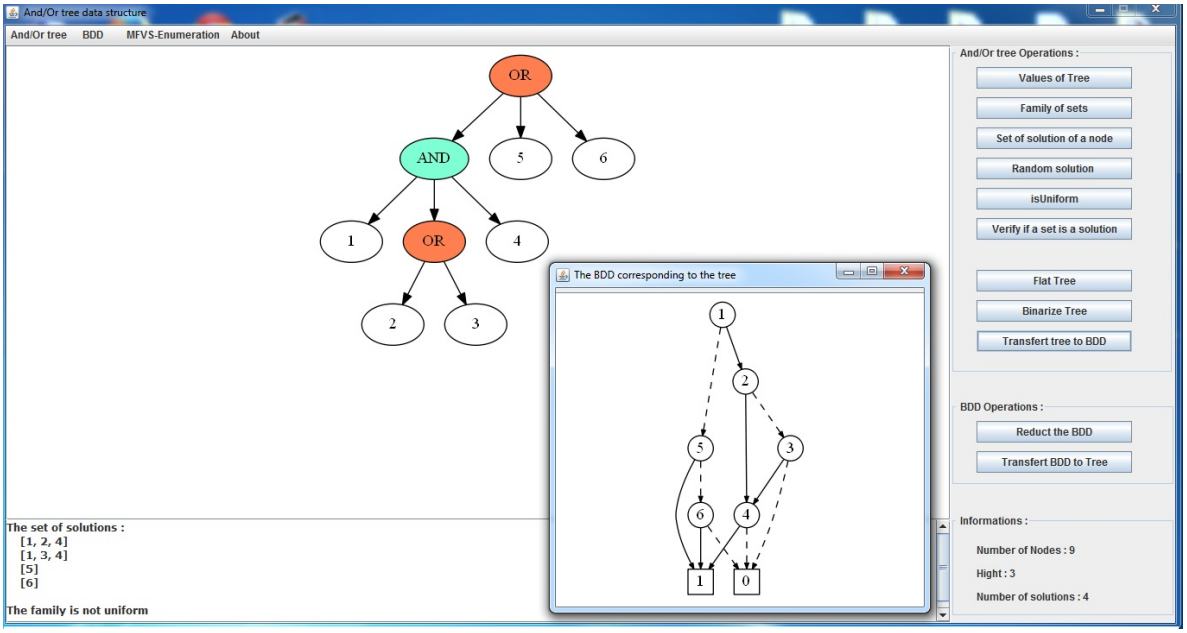



**Figure 5.11: Le DBD correspondant à un arbre et/ou produit par notre application.**

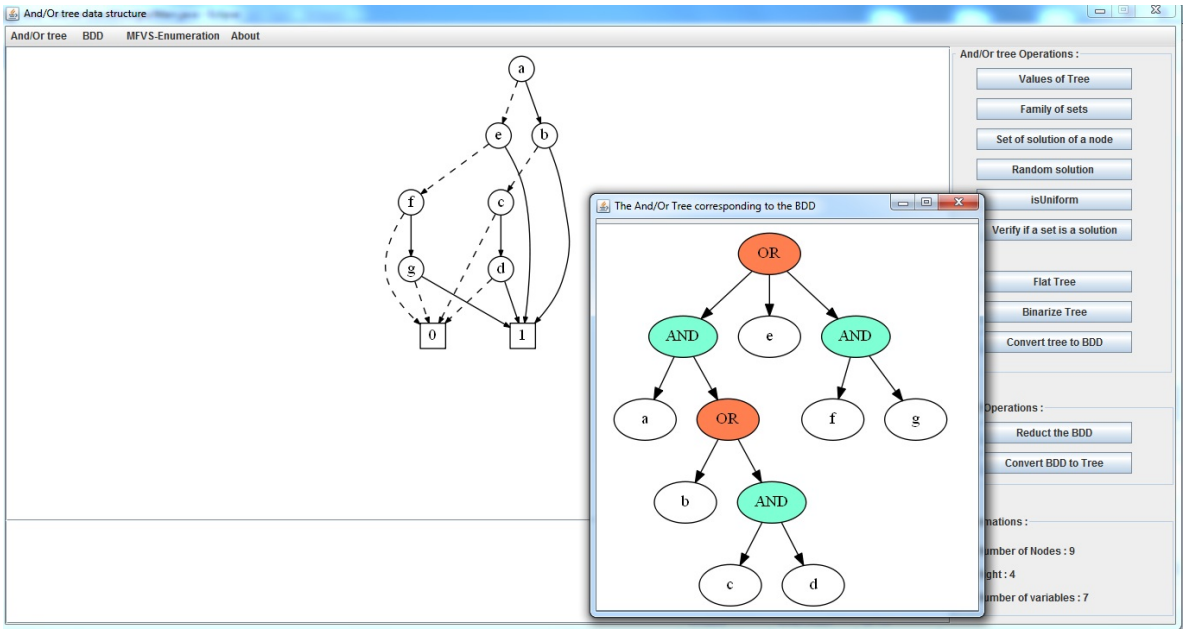



**Figure 5.12: L'arbre et/ou correspond à un DBD produit par notre application.**

## 5.3 CONCLUSION

Dans ce chapitre, nous avons présenté notre algorithme permettant l'énumération des transversaux de circuits de cardinalité minimale d'un graphe orienté à l'aide d'un arbre et/ou et l'implémentation de celui-ci. En premier lieu, nous avons détaillé l'aspect théorique et la manière de faire pour construire l'arbre et/ou représentant la famille de solutions et comment calculer cette famille à partir de cet arbre. Ensuite, nous avons exposé l'implémentation de ces algorithmes et quelques exemples de tests générés par notre application.

# CONCLUSION

Le problème de transversaux de circuits de cardinalité minimale est un problème NP-difficile classique, il a été étudié dans la littérature par plusieurs auteurs en utilisant différentes approches. Les principaux objectifs dans la résolution de ce problème sont le calcul d'un TCCM, l'énumération efficace de la famille de tous les TCCM, ainsi que le stockage compact de cette famille. Dans ce mémoire, nous avons étendu une méthode exacte basée sur la réduction du graphe afin de concevoir un algorithme exact permettant l'énumération et l'enregistrement de la famille de TCCM d'un graphe orienté non pondéré.

Afin d'élaborer notre algorithme et de permettre le stockage compact de la famille de TCCM, nous avons conçu et introduit la structure de données notée *arbre et/ou*. La définition de base de cette structure, la description théorique, la mise en œuvre de cette structure et des différentes opérations et modifications qui peuvent être effectuées sur cette structure dans un contexte d'énumération, ainsi que la mis en évidence de la relation entre cette structure et les DBD, ont été présentés et bien détaillés au cours de ce mémoire.

L'algorithme que nous avons proposé a permis de construire un arbre et/ou représentant la famille de tous les TCCM d'un graphe orienté. Cet arbre est de taille polynomiale par rapport à la taille du graphe traité si le graphe est *Dome-contractible* ou si on n'utilise pas les

opérateurs IN1, OUT1 et CORE pendant la réduction du graphe. Cependant, cette taille est non polynomiale pour certains graphes. Un autre avantage principale dans l'utilisation des arbres et/ou est qu'ils facilitent la gestion et la construction des sous-arbres correspondant aux sommets qui sont en conjonction ou en disjonction.

Les arbres et/ou sont introduit pour la première fois dans la résolution d'un problème combinatoire. Ils peuvent être utilisés dans de nombreux autres problèmes combinatoires pour stocker de manière compacte les familles d'ensembles. Les algorithmes proposés dans ce mémoire ne traitent que des graphes orientés non pondérés. Il y aurait encore des travaux à faire pour généraliser et améliorer les performances de nos algorithmes et de vérifier l'efficacité de la structure arbre et/ou dans la résolution d'autres problèmes, voici quelques travaux qui pourraient être faits dans un avenir rapproché :

– Utiliser les arbres et/ou dans la résolution d'autres problèmes combinatoires.
– Généraliser la structure des arbres et/ou de sorte que les nœuds peuvent comporter aussi des négations de valeurs.
– Faire une études comparative approfondie entre les arbres et/ou et les DBD.
– Déterminer la classe des graphes dont laquelle l'arbre correspondant est de taille polynomial en taille du graphe de l'entrée.
– Généraliser notre algorithme pour d'autres types de graphes (pondérés, non orientés, etc).

Le lecteur soucieux de découvrir et d'exécuter nos algorithmes sur les arbres et/ou, les DBD et l'énumération des transversaux, ainsi que de voir les exemples d'essai est invité à télécharger et regarder les codes source à la disposition du public hébergés sur Bitbucket sur le lien `https://bitbucket.org/Raouff27/mfvs-enumeration/downloads`. Je mentionne qu'une partie de ce travail a été faite par mon codirecteur M. A Blondin Massé.

## BIBLIOGRAPHIE


[Akers, 1978] Akers, S. 1978. « Binary decision diagrams », *IEEE Transactions on Computers*, vol. C-27, p. 509–516.

[Bachelet, 2011] Bachelet, B. 2011. Efficacite des algorithmes, complexite des problemes. `http://www.nawouak.net/?doc=course.operations_research+ch=complexity+lang=fr`.

[Bafna et al., 1999] Bafna, V., P. Berman, et T. Fujito. 1999. « A 2-approximation algorithm for the undirected feedback vertex set problem », *SIAM Journal on Discrete Mathematics*, vol. 12, Issue 3, p. 289–297.

[Bahar et al., 1993] Bahar, R., E. Frohm, C. Gaona, E. Massi, A. Pardo, et F. Somenzi. 1993. « Algebraic decision diagrams and their applications. dans proceedings of the 1993 ieee/acm international conference on computer-aided design », *IEEE Computer Society Press*, p. 188–191.

[Bar-Yehuda et al., 1994] Bar-Yehuda, R., D. Geiger, J. Naor, et R. Roth. Arlington, Virginia1994. « Approximation algorithms for the feedback vertex set problem with applications to constraint satisfaction and bayesian inference », *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, p. 344–354.

[Bollig et Wegener, 1996] Bollig, B. et I. Wegener. Septembre 1996. « Improving the variable ordering of obdds is np-complete », *IEEE Transactions on Computers*, vol. 45, Issue 9, p. 993–1002.

[Brunetta et al., 2000] Brunetta, L., F. Maffioli, et M. Trubian. 2000. « Solving the feedback vertex set problem on undirected graphs », *Discrete Applied Mathematics*, vol. 101, p. 37–51.

[Bryant, 1986] Bryant, R. 1986. « Graph-based algorithms for boolean function manipulation », *IEEE Transactions on Computers*, vol. C-35-8, p. 677–691.

[Bryant et Chen, 1995] Bryant, R. et Y. Chen. 1995. « Verification of arithmetic circuits with binary moment diagrams. dans proceedings of the 32nd acm/ieee conference on design automation conference », *ACM Press*, p. 535–541.

[Chen et al., 2008] Chen, J., Y. Liu, S. Lu, B. O'Sullivan, et I. Rozgan. New York, 2008. « A fixed-parameter algorithm for the directed feedback vertex set problem », *Proceedings of the 40th Annual ACM-SIAM Symposium on Discrete Algorithms*, p. 177–186.

[Chudak et al., 1998] Chudak, F. A., M. X. Goemans, D. S. Hochbaum, et D. P. Williamson. 1998. « A primal-dual interpretation of two 2-approximation algorithms for the feedback vertex set problem in undirected graphs », *Operations Research Letters*, vol. 22, p. 111–118.

[Cook, 1971] Cook, S. May 1971. « The complexity of theorem-proving procedures », *Proceedings Third Annual ACM Symposium on Thoery of Computing*, p. 151–158.

[Dehne et al., 2007] Dehne, F., M. Fellows, M. Langston, F. Rosamond, et K. Stevens. 2007. « An $o(2^{O(k)}n^3)$ fpt algorithm for the undirected feedback vertex set problem », *Springer Science+Business Media, Theory Computer Systems*, vol. 41, p. 479–492.

[Delisle, 2015] Delisle, P. 02 février 2015. *Problèmes NP-Difficiles : Résolution par méthodes exactes*. Université de Reims Champagne-Ardenne, Département de Mathématiques et Informatique.

[Even et al., 2000] Even, G., J. Naor, B. Schieber, et L. Zosin. 2000. « Approximating minimum subset feedback sets in undirected graphs with applications », *SIAM Journal on Discrete Mathematics*, vol. 13, Issue 2, p. 255–267.

[Festa et al., 1999] Festa, P., P. M. Pardalos, et M. G.C. Resende. 1999. « Feedback set problems », *Handbook of Combinatorial Optimization, Kluwer Academic*, vol. A, p. 209–258.

[Fomin et al., 2007] Fomin, F., S. Gaspers, A. Pyatkin, et I. Razgon. 26 November 2007. « On the minimum feedback vertex set problem : Exact and enumeration algorithms », *Springer Science+Business Media*, p. 293–307.

[Fujita et al., 1997] Fujita, M., P. McGeer, et J. Yang. 1997. « Multi-terminal binary decision diagrams : An efficient data structure for matrix representation. formal methods in system design », *Formal Methods in System Design : An International Journal*, p. 149–169.

[Garey et Johnson, 1979] Garey, M. R. et D. S. Johnson. 1979. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. Macmillan Higher Education.

[Guo et al., 2005] Guo, J., R. Niedermeier, et S. Wernicke. 2005. « Parameterized complexity of generalized vertex cover problems », *in Proceedings of the 9th International Workshop on Algorithms and Data Structures (WADS 2005)*, vol. 3608, p. 36–48.

[Gusfield, 1988] Gusfield, D. 1988. « A graph theoretic approach to statistical data security », *SIAM Journal on Computing*, vol. 17, Issue 3, p. 552–571.

[Joo et al., 2014] Joo, J., S. Wang, et S.-C. Zhuza. 2014. *Hierarchical Organization by And-Or Tree*. Oxford Handbook of Perceptual Organization.

[Karp, 1972] Karp, R. 1972. « Reducibility among combinatorial problems », *Complexity of Computer Computations*, p. 85–103.

[Kumar et N. Kanal, 1983] Kumar, V. et L. N. Kanal. 1983. « A general branch and bound formulation for understanding and synthesizing and/or tree search procedures », *Artificial Intelligence*, vol. 21,Issues 1–2, p. 179–198.

[Leblet, 2004] Leblet, J. 2004. « Représentation et approximation de fonctions booléennes : Application à la génération de requêtes ». Thèse de Doctorat, INSTITUT DE RECHERCHE EN INFORMATIQUE DE NANTES.

[Levy et Low, 1988] Levy, H. et D. Low. 1988. « A contraction algorithm for finding small cycle cutsets », *Journal of Algorithms*, vol. 9(4), p. 470–493.

[Liedloff, 2007] Liedloff, M. Décembre 2007. « Algorithmes exacts et exponentiels pour les problèmes np-difficiles : domination, variantes et généralisations ». Thèse de Doctorat, Université Paul Verlaine – Metz.

[Lin et Jou, 2000] Lin, H. et J. Jou. 2000. « On computing the minimum feedback vertex set of a directed graph by contraction operations », *IEEE Transactions on computer-aided design of integrated circuits and systems*, vol. 19, p. 295–306.

[Lopez, 2008] Lopez, P. 2008. Graphes. `http://homepages.laas.fr/lopez/cours/GRAPHES/graphes.html`.

[Marinescu et Dechter, 2008] Marinescu, R. et R. Dechter. 2008. « Advancing and/or search for optimization using diverse principles », *In Workshop on Inference Methods based on Graphical Structures of Knowledge of the European Conference on Artificial Intelligence (ECAI)*. Disponible à `http://www.ics.uci.edu/~dechter/publications/`.

[Marinescu et Dechter, 2009] ———. 2009. « And/or branch-and-bound search for combinatorial optimization in graphical models », *Artificial Intelligence*, vol. 173, p. 1457–1491.

[Mary, 2013] Mary, A. 2013. « Énumération des dominants minimaux d'un graphe ». Thèse de Doctorat, Université Blaise Pascal.

[Mihai et al., 2004] Mihai, P., D. Kosack, et S. Salzberg. 2004. « Hierarchical scaffolding with bambus », *Genome Res*, vol. 14, Issue 1, p. 149–159.

[Minato, 1993] Minato, S. 1993. « Zero-suppressed bdds for set manipulation in combinatorial problems. proceedings of the 30th international on design automation conference », *ACM Press*, p. 272–277.

[Minato, 1996] ———. 1996. *Binary Decision Diagrams and Applications for VLSI CAD*. Kluwer Academic Publisher.

[Narayan et al., 1996] Narayan, A., J. Jawahar, M. Fujita, et A. Sangiovanni-vincentelli. 1996. « Partitioned robdds-a compact, canonical and efficiently manipulable representation for boolean functions. dans proceedings of the 1996 ieee/acm international conference on computer-aided design », *IEEE Computer Society*, p. 547–554.

[Pardalos et al., 1999] Pardalos, P. M., T. Qian, et M. G. C. Resende. 1999. « A greedy randomized adaptive search procedure for the feedback vertex set problem », *Journal of Combinatorial Optimization*, vol. 2, p. 399–412.

[Razgon, 2006] Razgon, I. 2006. « Exact computation of maximum induced forest », *Comput. Sci., Springer Verlag*, p. 160–171.

[Rosen, 1982] Rosen, B. 1982. « Robust linear algorithms for cutsets », *Journal of Algorithms*, vol. 3, p. 205–217.

[Rudell, 1993] Rudell, R. 1993. « Dynamic variable ordering for ordered binary decision diagrams », *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, p. 42–47.

[Schwikowski et Speckenmeyer, 2002] Schwikowski, B. et E. Speckenmeyer. 2002. « On enumerating all minimal solutions of feedback problems », *Discrete Applied Mathematics*, vol. 117, p. 253–265.

[Shamir, 1979] Shamir, A. 1979. « A linear time algorithm for finding minimum cutsets in reduced graphs », *SIAM Journal On Computing*, vol. 08, p. 645–655.

[Smith et Walford, 1975] Smith, G. et R. Walford. 1975. « The identification of a minimal feedback vertex set of a directed graph », *IEEE Transactions on Circuits and Systems*, vol. 22, p. 9–15.

[Soranzo et al., 2012] Soranzo, N., F. Ramezani, G. Iacono, et C. Altafini. 2012. « Decompositions of large-scale biological systems based on dynamical properties », *Bioinformatics*, vol. 28, Issue 1, p. 76–83.

[Tourniaire, 2013] Tourniaire, E. Octobre 2013. « Problèmes np-difficiles : approximation modérément exponentielle et complexité paramétrique ». Thèse de Doctorat, Université Paris Dauphine - Paris IX.

[Wang et al., 1985] Wang, C.-C., E. Lloyd, et M. Soffa. 1985. « Feedback vertex sets and cyclically reducible graphs », *Journal of the ACM*, vol. 32, Issue2, p. 296–313.

[Yannakakis, 1978] Yannakakis, M. 1978. « Node and edge-deletion np-complete problems », *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, p. 253–264.

[Zhang et al., 2013] Zhang, Z., A. Ye, X. Zhou, et Z. Shao. 2013. « An efficient local search for the feedback vertex set problem », *Algorithms*, vol. 6, p. 726–746.

# ANNEXE

Le graphe de la figure 5.13 est un graphe créé par un anonyme représentant un dictionnaire composé de 38 mots. L'arbre et/ou de la figure 5.14 est l'arbre représentant la famille des TCCM du graphe de la figure 5.13 produit par notre application. Cet arbre utilise 38 nœuds et 37 arcs pour représenter une famille de 1152 TCCM, chaque TCCM contient 12 nœuds.

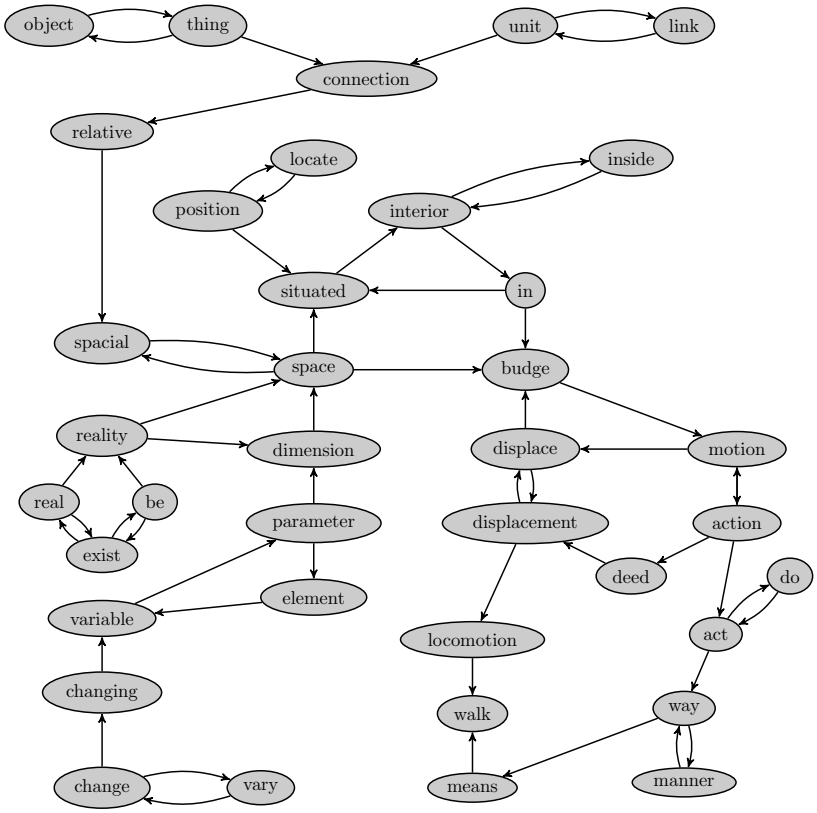



**Figure 5.13: Un graphe créé par un anonyme représentant un dictionnaire de mots.**

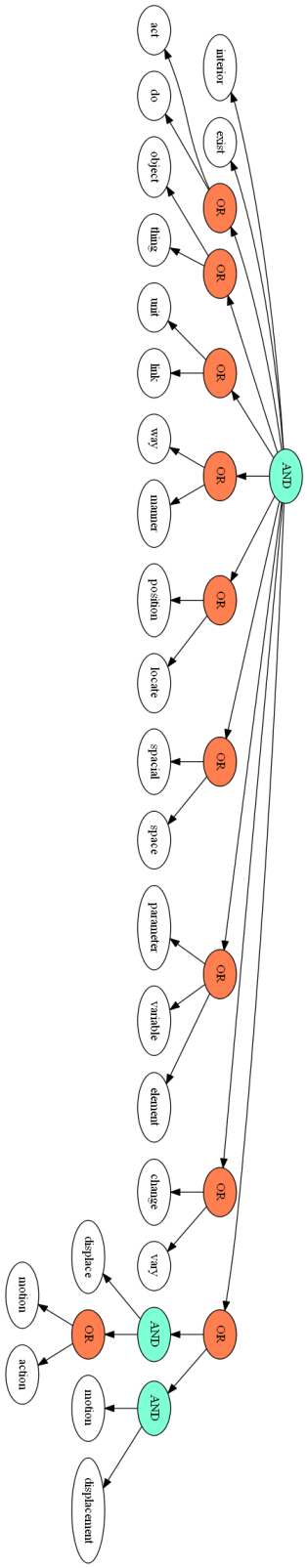



**Figure 5.14: L'arbre représentant la famille des TCCM du graphe de la figure 5.13 généré par notre application.**