SIMON PAQUETTE

ÉVALUATION SYMBOLIQUE DE SYSTÈMES PROBABILISTES À ESPACE D'ÉTATS CONTINU

Mémoire présenté
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de maîtrise en informatique
pour l'obtention du grade de maître ès sciences (M.Sc.)

DÉPARTEMENT D'INFORMATIQUE ET DE GÉNIE LOGICIEL
FACULTÉ DES SCIENCES ET DE GÉNIE
UNIVERSITÉ LAVAL
QUÉBEC

2005

Résumé

Nous présentons une méthode symbolique pour représenter des modèles probabilistes à espace d'états continu (LMP). Ces modèles représentent des systèmes de transitions qui ont un nombre d'états non-dénombrable. L'objectif de l'utilisation d'une méthode symbolique est de consommer moins de mémoire lors de l'enregistrement du modèle. Notre technique pour enregistrer symboliquement les modèles est l'utilisation des diagrammes de décision binaire à terminaux multiples (MTBDD). Ces diagrammes offrent la possibilité d'enregistrer des fonctions à variables booléennes, ainsi que des graphes, de façon plus concise que les techniques plus fréquemment utilisées. Nous employons les MTBDD dans un vérificateur de modèles de type LMP nommé CISMO.

La vérification de modèles est une technique entièrement automatique qui est utilisée pour vérifier si un modèle respecte ou non sa spécification. L'espace mémoire nécessaire pour faire la vérification dépend entre autres des structures utilisées pour enregistrer le modèle. Dans la première version de CISMO, le modèle était enregistré à l'aide de structures de données traditionnelles, comme des vecteurs. Cette façon de faire est qualifiée d'*explicite*. Nos modifications de CISMO lui permettent maintenant de faire la vérification de façon symbolique, à l'aide des MTBDD, en plus de la façon explicite. Nos principales contributions sont la description de deux heuristiques qui, dans certains contextes, réduisent la taille des diagrammes de décision binaire ordonnés, ainsi que la modification de CISMO. L'analyse de la mémoire consommée par CISMO après nos modifications nous montre que l'utilisation des diagrammes de décision permet de sauver de la mémoire.

Remerciements

Je veux d'abord remercier grandement ma directrice de recherches, Josée Desharnais, avec qui j'ai eu un réel plaisir à travailler et discuter. Merci Josée !

Je désire remercier mes camarades du LSFM pour nos nombreuses discussions et pour leur aide. Je souhaite aussi remercier le personnel de soutien du département d'informatique et de génie logiciel, en particulier Lynda Goulet, Jacques Ménard, Jean-Sébastien Landry et Louis Demers, pour leur réelle volonté à nous aider à régler nos « petits » problèmes.

Je tiens à remercier Fernand Beaudet, sans qui je n'aurais pas eu la chance d'étudier les mathématiques. Merci Fernand de m'avoir transmis une de tes passions.

Finalement, je voudrais dire un merci spécial à mes parents qui m'ont encouragé et permis d'étudier toutes ces années et à Sarah, pour être ma soeur préférée.

Table des matières

Table des figures

Chapitre 1

Introduction

Depuis quelques années, le matériel électrique et les logiciels occupent une très grande place dans nos vies. Par exemple, les voitures sont maintenant munies d'ordinateurs qui contrôlent certaines pièces du véhicule, les avions ont des systèmes de pilotage automatiques et presque chaque famille possède un ordinateur. Il est important de s'assurer du bon fonctionnement de tous ces systèmes avant de les mettre sur le marché, en particulier pour les applications où la sécurité est critique comme les appareils médicaux et les systèmes de contrôle aérien. Dans ces applications, la vie des gens est en jeu, mais la sécurité est aussi importante dans certaines autres applications, telles celles utilisées dans le commerce électronique. Dans celles-ci, la sécurité est primordiale d'un point de vue économique.

Au cours des dernières années, plusieurs erreurs qui se sont produites auraient pu être évitées si une meilleure vérification avait été effectuée. Par exemple, l'explosion de la fusée Ariane 5 en 1996 qui a été causée par une erreur de logiciel [23]. Il y a aussi eu l'erreur de division dans le processeur Pentium en 1994 [15], erreur qui a coûté environ 500 millions de dollars !

La vérification commence à occuper une place importante dans le processus de conception du matériel et des logiciels, plus spécialement dans la fabrication de matériel informatique. Les compagnies veulent éviter de construire des pièces ne respectant pas leur spécification et surtout d'avoir à faire des rappels qui sont souvent très coûteux et qui entachent la réputation de la compagnie. Cependant, s'assurer qu'un circuit logique ou qu'un programme fonctionne correctement est devenu plus difficile au fur et à mesure que les systèmes se sont complexifiés. Pour ces raisons, plusieurs techniques de vérifications ont été développées pour améliorer et faciliter la vérification.

L'objectif de mes recherches était d'étudier la vérification de systèmes de transitions probabilistes. Pour situer la technique de vérification que nous étudions, nous décrirons dans cette introduction quatre types de techniques de vérification. Ensuite nous donnerons quelques définitions concernant les systèmes probabilistes. Finalement, nous expliquerons la problématique sur laquelle nous nous sommes penchés et comment nous l'avons abordée.

1.1 Vérification

La vérification des programmes ou du matériel électrique vise à détecter les erreurs qu'ils contiennent ou à démontrer qu'ils n'en contiennent pas. La vérification peut se faire directement sur le système ou sur une représentation de celui-ci qu'on appelle un modèle. Il y a 4 grandes classes de techniques de vérification : la simulation, les tests, la démonstration de théorème et la vérification par évaluation de modèle. Ces deux dernières techniques sont des méthodes formelles qui peuvent nous assurer qu'une propriété est *toujours* respectée.

1.1.1 Simulation et tests

Les techniques les plus utilisées sont la simulation et les tests. Ces deux méthodes effectuent la vérification en observant les sorties obtenues sur certaines entrées. La simulation est effectuée sur le modèle tandis que les tests sont faits sur le système lui-même. Ces deux méthodes donnent de bons résultats, *i.e.* elles nous permettent de détecter des erreurs. Cependant, elles ne peuvent pas nous assurer de l'absence d'erreurs car il est rarement possible de vérifier toutes les entrées possibles et il peut être très long de connaître la sortie associée à une entrée donnée.

1.1.2 Démonstration de théorème

La démonstration de théorème [55] est une approche basée sur les preuves. Le système à vérifier est décrit par un ensemble de formules Γ et la spécification du système, c'est-à-dire les propriétés désirées du système, est elle aussi décrite par des formules ϕ_i. Ensuite, pour chaque propriété, le démonstrateur tente de démontrer ϕ_i à partir de Γ. La preuve est syntaxique et permet donc d'obtenir $\Gamma \vdash \phi_i$. Il est possible d'automatiser en partie la démonstration de théorème mais chaque preuve peut nécessiter une

intervention humaine ce qui oblige la personne qui guide la démonstration à avoir une certaine expertise. Un des gros avantages de cette technique est qu'il est quasiment possible de tout démontrer (théoriquement). Mais étant donné que la technique n'est pas entièrement automatique, il n'est pas possible d'obtenir une borne sur le temps et la mémoire qu'il faudra pour faire la démonstration. Malgré cela, la démonstration de théorème est très étudiée et elle a été utilisée avec succès dans plusieurs cas : la conjecture de Robbins a été démontrée à l'aide d'un démonstrateur automatique de théorème [32].

1.1.3 Vérification par évaluation de modèle

La vérification par évaluation de modèle (*model-checking* en anglais) est une technique entièrement automatique basée sur les modèles. Elle est surtout utilisée pour vérifier des systèmes réactifs. Ceux-ci sont des systèmes qui ne terminent pas et qui réagissent à leur environnement : par exemple, une machine à café. Cette machine est en fonction sans arrêt et elle répond aux demandes des utilisateurs. Pour utiliser cette technique, il nous faut d'abord construire un modèle et ensuite spécifier la propriété à vérifier. Le modèle représente en fait l'ensemble des états possibles du système ainsi que les transitions entre les états. La vérification se fait en parcourant chaque état du modèle et en vérifiant quels états satisfont la propriété vérifiée. De plus, lorsqu'une propriété n'est pas valide, plusieurs outils de vérification sont en mesure de nous donner une trace d'exécution qui montre comment le problème peut arriver, ce qui est très utile pour corriger ce problème.

La vérification par évaluation de modèle est surtout utilisée pour vérifier qu'un modèle satisfait des propriétés temporelles telles la sûreté et la vivacité. Une propriété de sûreté sert à représenter le fait que *quelque chose de mauvais ne va jamais arriver*. Par exemple, un utilisateur ne veut pas que son ordinateur envoie des fichiers personnels sur Internet sans sa permission. Un propriété de vivacité représente le fait que *quelque chose de bien va nécessairement arriver*. Par exemple, lorsque quelqu'un lance une impression, il veut qu'elle s'exécute éventuellement. Une propriété d'un système réel peut prendre plusieurs formes comme « il n'y a jamais de blocage » et « le système va éventuellement répondre ».

Cette méthode a été utilisée avec succès à plusieurs reprises et elle est devenue très populaire, spécialement dans le domaine du matériel électronique. Il faut cependant savoir qu'il existera toujours des applications importantes que l'on ne pourra pas vérifier par cette technique étant donné la taille du modèle. Outre les limites de mémoire, il est important de remarquer que la vérification par évaluation de modèle est dépendante de

la qualité du modèle, c'est-à-dire que si le modèle comporte des erreurs, la vérification sera faussée. Avec assez de ressources (temps et mémoire), la terminaison du processus de vérification est souvent assurée.

Remarquons la différence entre *système* et *modèle*, le premier représente le système réel tandis que le deuxième représente la modélisation faite du système en synthétisant et parfois en faisant des abstractions. On illustre souvent le modèle par un graphe orienté dont les sommets sont les états du modèle et les transitions représentent la possibilité de passer d'un état à l'autre dans le modèle. Ainsi, nous parlerons parfois de graphe au lieu de modèle. Puisque notre travail porte seulement sur la vérification par évaluation de modèle, pour abréger nous dirons seulement *vérification*. De plus, les outils de vérification par évaluation de modèle seront nommés vérificateurs de modèle (*model-checkers* en anglais).

1.2 Systèmes probabilistes

De façon générale, nous nous intéressons aux systèmes réactifs, c'est-à-dire aux systèmes qui interagissent avec leur environnement qui, lui, peut être composé d'utilisateurs et d'autres systèmes. Le système réagit donc aux stimuli qui viennent de l'environnement. Notre étude se concentre sur la réaction du système à ces stimuli que nous pouvons observer. Comme exemple de système réactif, outre l'exemple classique de la machine à café, pensons à un serveur en réseau qui doit répondre à des requêtes : selon le type des requêtes et l'ordre dans lequel elles se présentent, le serveur peut réagir de différentes façons.

Au cours des années, les techniques de spécification et de modélisation se sont perfectionnées et on a cherché à élargir la classe des modèles sur lesquels il est possible de faire de la vérification. En particulier, certains chercheurs se sont intéressées aux systèmes probabilistes. Beaucoup de systèmes réels ont un caractère stochastique, c'est-à-dire qu'il n'est pas possible d'être certain du prochain état du système après une transition.

Les systèmes probabilistes se retrouvent partout : en physique, en finance, en informatique, dans le transport, etc. Ainsi, il est devenu nécessaire de pouvoir vérifier ces systèmes. Parfois l'information obtenue lors d'une étude sans probabilité peut être insuffisante. Par exemple si une telle étude indique qu'un arrêt du système peut survenir, il est difficile de conclure si nous pouvons ou non prendre la chance qu'un arrêt survienne car nous ne savons pas à quelle fréquence cela peut se produire. Par contre, si nous savons que la probabilité que le système arrête est de 0.01%, il se peut que

nous acceptions ce risque puisque la probabilité est faible. Nous nous intéressons donc à savoir si une propriété est vérifiée avec une certaine probabilité. Nous pouvons par exemple souhaiter vérifier si le système répondra avec une probabilité d'au moins 0.83 ou calculer avec quelle probabilité le système peut faire une action non souhaitée.

Les systèmes probabilistes sont caractérisés par le fait que, pour une action donnée, chaque transition a une probabilité qui lui est associée ; ainsi, le système « choisit » une transition selon une loi de probabilité. Pour chaque état, il peut y avoir plusieurs transitions possibles avec une même action et chacune d'elles est étiquetée par la probabilité qu'elle a d'être empruntée. Par exemple, il se peut qu'un serveur choisisse la requête à exécuter de façon uniforme parmi celles demandées. Ainsi, chaque requête aurait une probabilité égale d'être exécutée. Nous pourrions aussi demander que la probabilité accordée aux requêtes venant de l'utilisateur A soit plus grande que celle accordée aux requêtes de l'utilisateur B. La figure 1.1 est un exemple de modèle probabiliste.

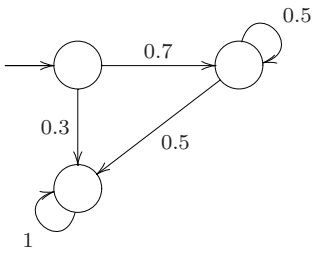


Fig. 1.1 – Exemple d'un modèle probabiliste.

Ainsi, un système probabiliste est très semblable à un système de transitions classique : il a un ensemble d'états, un ensemble de transitions et un ensemble d'étiquettes. Cependant, les transitions d'un système probabiliste sont pondérées par une probabilité. Il y a des systèmes qui sont naturellement probabilistes. Par exemple, plusieurs versions de protocoles d'élection d'un chef sont probabilistes. Toutefois, il y a des systèmes qui ne sont pas probabilistes et que l'on choisit de modéliser à l'aide de modèles probabilistes pour quantifier leur caractère non déterministe. Par exemple, le système de pilotage d'un avion n'est pas un système probabiliste. Dans cet exemple, les probabilités sont utilisées pour simplifier la modélisation du système. Sans elles, il faudrait déterminer exactement le comportement du système dans chaque circonstance.

1.2.1 Systèmes continus

Dans ce travail, nous nous intéressons particulièrement aux systèmes probabilistes dont l'espace d'états est continu et dont le temps est discret. On dit qu'un système

a un espace de temps discret si les transitions se font de façon discrète. Par exemple, lorsqu'un système fait une transition à toutes les secondes. On dit que l'espace de temps d'un système est continu lorsque les transitions sont faites de façon continue.

Les systèmes à espace d'états continu et à espace de temps discret ont été introduits par Blute et al. [9] en 1997. Un système a habituellement un espace d'états continu lorsque des paramètres physiques interviennent, tels la position, la pression et la température. En plus de la différence au niveau de l'espace d'états, une grande différence entre ces systèmes et ceux à espace d'états dénombrable, est le fait que les probabilités sont décrites à l'aide de fonctions continues (de densité ou de répartition).

Par exemple, si un système donne en sortie une quantité de liquide (comme une machine à café), cette quantité peut être quantifiée par une probabilité. Dans cet exemple, les probabilités sont utilisées pour modéliser la possibilité que le système ne donne pas toujours la même quantité de liquide. La loi de probabilité indique donc la probabilité qu'il y ait peu ou beaucoup de liquide. Puisque le volume est continu, il est modélisé par une loi continue, par exemple une loi normale. Il pourrait ainsi y avoir une faible probabilité d'obtenir insuffisamment ou trop de liquide.

Il existe plusieurs vérificateurs de modèles probabilistes à espace d'états fini. Cependant, il y en a peu qui vérifient les modèles probabilistes à espace d'états continu. En fait, il n'y en a qu'un seul et il se nomme CISMO (de l'anglais *ContInuous State space MOdel-cheker*). Notre travail porte justement sur ce vérificateur de modèles qui a été développé par un collègue, Richard [48]. Les seuls autres vérificateurs qui travaillent sur des modèles continus sont les vérificateurs de systèmes hybrides. Notre travail porte sur le vérificateur CISMO.

Les systèmes hybrides [3] sont de plus en plus étudiés. Un système hybride est un système à espace d'états continu qui est combiné à un processus de contrôle qui est discret. L'exemple classique est un passage à niveau constitué d'un train, d'une barrière et d'un contrôleur qui lève la barrière lorsqu'il n'y a pas de train et qui la baisse dans le cas contraire. Cet exemple est un système hybride, car d'une part on a le train, dont le déplacement est continu et d'autre part on a le contrôleur (et la barrière) dont l'espace d'états est discret.

1.3 Réduction de l'espace d'états

À ses débuts, la vérification se faisait en explorant explicitement tous les états du modèle. Un problème majeur de cette approche est que l'espace d'états est beaucoup trop grand. En fait, le nombre d'états est exponentiel par rapport à la mémoire utilisée par le système. Dans le cas des systèmes parallèles, le modèle du système est la composition des modèles des processus en parallèle et une explosion combinatoire survient lorsque cette composition est faite. Par exemple, si le modèle a deux composantes à N états, le modèle final aura $N \times N$ états puisque pour chaque état d'une composante il y a N états possibles pour l'autre composante. Ainsi, nous remarquons que le nombre d'états dans le système final est exponentiel par rapport au nombre de composantes. De plus, lors de la vérification, le calcul des états qui satisfont une formule entraîne aussi une explosion combinatoire. En effet, l'ensemble des états à déterminer est un sous-ensemble des états et le nombre de sous-ensembles est exponentiel par rapport au nombre d'éléments de l'ensemble de départ.

Le problème lié au grand nombre d'états est nommé le problème d'*explosion du nombre d'états* ou le problème d'*explosion combinatoire*. Dû à ce problème, il est habituellement trop long de faire la vérification en parcourant explicitement tous les états. Il y a actuellement beaucoup de recherche effectuées afin d'essayer de diminuer l'effet de l'explosion combinatoire et plusieurs méthodes ont été proposées. Ces méthodes sont généralement divisées en trois catégories : abstraction, réduction et compression.

1.3.1 Abstraction et symétrie

L'abstraction est employée pour transformer un modèle complexe en un modèle abstrait plus simple [14, 57]. Il y a plusieurs façon de faire de l'abstraction. Nous en survolerons ici quelques-unes.

Une abstraction souvent utilisée est de donner une même valeur abstraite aux variables qui sont dans un certain intervalle. Par exemple, nous pourrions donner la valeur v_0 à toutes les variables qui prennent comme valeurs des nombres entiers positifs. Il faut ensuite redéfinir les opérations sur les valeurs abstraites pour finalement effectuer la vérification directement sur le modèle abstrait. Nous pouvons aussi voir la réduction par symétrie comme une forme d'abstraction. Celle-ci réduit l'espace d'états en remplaçant des structures symétriques d'un modèle par une seule structure. Il est possible de voir cette opération comme un quotient par rapport à la symétrie. Finalement, l'abstraction se présente aussi sous la forme de l'élimination de variables. Nous pouvons fixer

la valeur d'une variable lorsque nous savons que cette valeur n'influencera pas le résultat de la vérification ce qui résulte en un nombre réduit d'états.

1.3.2 Réduction

La réduction, nommée *partial order reduction* en anglais, consiste à diminuer le nombre d'états en enlevant des états et des exécutions qui n'influencent pas une propriété donnée. Pour réduire l'espace d'états, on utilise l'indépendance des actions.

Cette méthode est surtout utilisée pour réduire l'espace d'états d'un système composé de plusieurs systèmes concurrents. Dans la composition de deux processus concourants, on nomme *actions indépendantes* les actions qui ne sont pas communes aux deux processus. Les actions indépendantes apparaissent dans le modèle composé sous forme de losanges, comme le montre la figure 1.2.

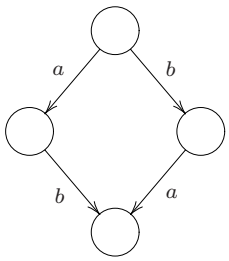


FIG. 1.2 – Losange représentant deux actions indépendantes.

On remarque dans cette figure qu'il est possible que l'action a soit effectuée avant l'action b ou l'inverse et ceci est causé par le fait que les processus ne sont pas synchronisés sur les actions non-communes aux deux processus.

De façon générale, la méthode restreint la vérification aux exécutions dont l'ordre des actions influence la validité d'une propriété. L'exemple suivant montre comment une telle réduction peut avoir lieu.

Exemple 1.3.1 *Pour vérifier si la formule $F\neg\mathsf{p}$ (dans tous les chemins la formule* p *sera éventuellement non-valide) est vraie sur le modèle a) de la figure 1.3, nous pouvons réduire ce modèle à celui de b) et vérifier la formule sur celui-ci.*

Dans le modèle de la figure 1.3 a), peu importe que l'action a soit exécutée avant

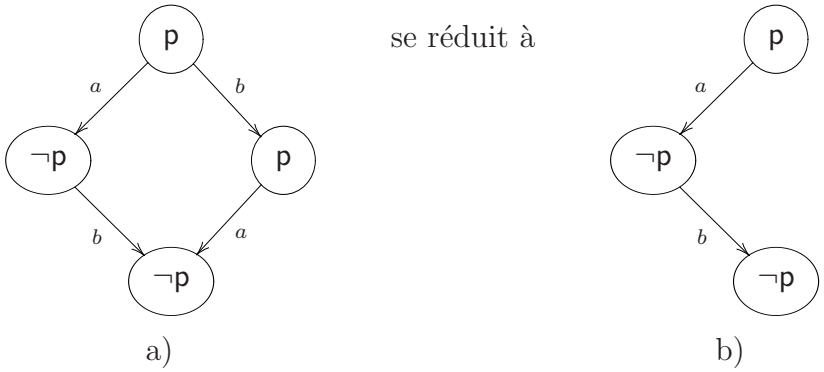


FIG. 1.3 – Exemple d'une réduction.

l'action b ou le contraire, l'état final ne satisfait pas la proposition p*. Ainsi, le modèle satisfait* $F\neg$p*. Mais puisque l'ordre des actions n'a pas d'importance, il est plus rapide de vérifier la formule sur le modèle b) de la même figure. De cette façon, moins d'états doivent être parcourus.*

Cette technique peut être employée lors de la vérification de toutes les formules d'une logique temporelle linéaire qui ne contient pas d'opérateur « suivant » (*next* en anglais).

En appliquant cette méthode à un gros modèle il est alors possible de réduire considérablement l'espace d'états. Dans Baier [6], on mentionne avoir diminué de 25% à 30% le nombre d'états en utilisant la réduction sur le problème des philosophes. Pour plus de détails sur la réduction, consulter Godefroid [24].

1.3.3 Compression et vérification symbolique

Finalement, la compression utilise des structures de données afin de représenter de façon concise des ensembles d'états. Les opérations se font alors sur des ensembles d'états plutôt que sur des états explicites. Cette technique est souvent qualifiée de *symbolique*. Une méthode symbolique bien connue est la représentation par diagrammes de décision binaire (BDD, de l'anglais *Binary Decision Diagrams*). L'utilisation des BDD est aujourd'hui tellement répandue que BDD est souvent utilisé comme synonyme de méthode symbolique.

Les BDD sont des diagrammes binaires qui servent à représenter efficacement une formule booléenne. En vérification, ils sont utilisés pour représenter un ensemble d'états.

L'utilisation des BDD a grandement augmenté la taille des systèmes pour lesquels il est possible de faire la vérification, passant de 2^{17} états à plus de 2^{100} états [31]. Évidemment, il n'est pas possible de faire la vérification sur tous les systèmes de 2^{100} états, mais les BDD on rendu la chose possible sur certains systèmes. Malheureusement, même un système de 2^{100} états est un petit système en comparaison avec plusieurs systèmes réels. La vérification symbolique est devenue très populaire, spécialement dans la vérification de circuits. Les résultats obtenus dans ce domaine ont été suffisamment bons pour que les BDD soient maintenant utilisés dans la vérification de logiciels et même en apprentissage automatique [5, 18].

1.4 Objectif

L'objectif de ce travail était d'étudier la vérification symbolique de systèmes probabilistes à espace d'états continu. Nous souhaitions approfondir nos connaissances sur les BDD dans le but de les utiliser dans le vérificateur de modèles CISMO et ainsi réduire la mémoire utilisée par celui-ci.

1.5 Organisation du mémoire

Au chapitre 2, deux types de modèles probabilistes seront introduits : les DTMC et les LMP. Les DTMC ont un espace d'états fini et ils nous servent de premier exemple pour présenter les systèmes probabilistes, alors que les LMP généralisent les DTMC aux espaces d'états continu. Le vérificateur de modèles CISMO porte sur une classe des modèles de type LMP.

Au chapitre 3, nous définirons formellement ce qu'est un BDD et donnerons quelques algorithmes de manipulation de cette structure. De plus, nous discuterons de l'utilisation des BDD pour la représentation des fonctions booléennes et des graphes.

Au chapitre 4, différentes variantes des BDD seront présentées. Plusieurs d'entre elles ont été développées pour augmenter l'efficacité de la structure dans des applications spécifiques. Nous présenterons entre autres les MTBDD, une généralisation naturelle des BDD. Nous survolerons quelques-unes des variantes pour présenter les choix de structures qu'il nous était possible de faire dans notre implémentation.

Au chapitre 5, nous étudierons les causes de variation de la taille des BDD. Sans que

cela soit essentiel pour notre travail, de beaux problèmes se posent et nous souhaitions les aborder. De plus, nous proposerons des heuristiques pour améliorer l'utilisation des BDD.

Nous terminerons au chapitre 6 en expliquant les modifications apportées au vérificateur de modèles CISMO. Pour vérifier si l'utilisation des MTBDD diminue la mémoire utilisée dans CISMO, nous ferons une comparaison de performance entre la méthode avec BDD et celle qui existait déjà dans la première version de CISMO.

Chapitre 2

Systèmes probabilistes

Nous présenterons dans ce chapitre deux types de modèles stochastiques : les chaînes de Markov à temps discret (DTMC) et les processus de Markov étiquetés (LMP). Les DTMC ont un espace d'états fini et les LMP ont un espace d'états continu. Notre projet porte sur les LMP mais il est intéressant de connaître les DTMC pour voir les différences entre un modèle à espace d'états discret et un modèle à espace d'états continu. De plus, plusieurs idées utilisées dans notre implémentation et dans l'utilisation des BDD sont inspirées des implémentations qui portent sur les DTMC. Dans les prochaines sections, nous donnerons les définitions et des exemples de chacun de ces types de systèmes.

2.1 Classes de modèles

Nous avons introduit informellement les systèmes de transitions dans l'introduction. En voici un rappel :

Définition 2.1.1 *Un système de transitions est un tuple* $(S, i, \rightarrow)$ *où* S *est un ensemble fini ou dénombrable d'états,* $i \in S$ *est l'état de départ et* $\rightarrow \subseteq S \times S$ *est l'ensemble des transitions.*

Les modèles probabilistes sont des systèmes de transitions dont les transitions sont pondérées par des probabilités. Il y a plusieurs types de modèles probabilistes et ceux-ci sont caractérisés selon

- leur nature déterministe ou non,

- leur traitement du temps et
- la cardinalité de leur espace d'états.

Les modèles probabilistes déterministes ont une seule loi de probabilité qui détermine les transitions. Par exemple, le modèle de la figure 2.1 est déterministe car à chaque état une seule distribution de probabilité décrit les transitions vers les autres états qui sont atteignables. Les modèles probabilistes non-déterministes ont plusieurs lois de probabi-

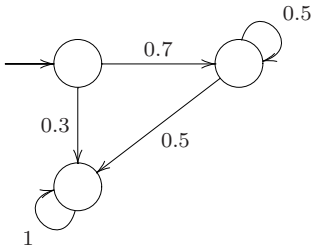


FIG. 2.1 – Exemple d'un modèle probabiliste déterministe.

lité possibles et à chaque transition le choix parmi celles-ci est indéterminé. La figure 2.2 montre un modèle non-déterministe. Il y a deux distributions de probabilités, une

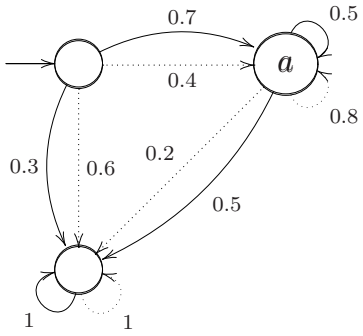


FIG. 2.2 – Exemple d'un modèle probabiliste non-déterministe.

avec les arcs pleins (A) et l'autre avec les arcs pointillés (B). Avant chaque transition, le système doit choisir entre ces deux distributions et prendre ensuite une transition selon cette loi. Comme conséquence, remarquons que si le système de l'exemple 2.2 est dans l'état a et que la distribution A est choisie, la probabilité d'être encore dans l'état a après une transition est plus petite que si la distribution B avait été choisie.

On distingue aussi les modèles par leur traitement du temps et par la cardinalité de leur espace d'états. Il y a des modèles à temps discret ou continu et à espace d'états discret ou continu. Les exemples précédents ont un espace d'états discret ; en effet, les chaînes de Markov à temps discret [38] [25], sont le type de modèle probabiliste le plus simple, ils ont un espace d'états discret et, comme leur nom l'indique, ils traitent le

temps de façon discrète. Il existe aussi des modèles très populaires : les chaînes de Markov à temps continu [4], les processus de Markov étiquetés [9] et les processus de décision de Markov [47]. Parmi ces modèles, seuls les processus de décision de Markov sont non-déterministes, seules les chaînes de Markov à temps continu traitent le temps de façon continue et seuls les processus de Markov étiquetés ont un espace d'états continu.

Tel que mentionné dans l'introduction, il existe en plus des modèles hybrides qui modélisent l'interaction entre un modèle à espace d'états discret et un modèle à espace d'états continu.

2.2 Chaînes de Markov à temps discret (DTMC)

Les chaînes de Markov à temps discret (DTMC, de l'anglais *Discrete Time Markov Chains*), introduites par Lehmann et Shelahen [38], sont les modèles les plus simples représentant des systèmes probabilistes. Les DTMC ont un espace d'états discret, un espace de temps discret et sont déterministes. Chaque transition est étiquetée par un nombre réel compris entre 0 et 1 : ce nombre représente la probabilité que le système prenne cette transition pour passer d'un état à un autre. La loi de probabilité est donc discrète. Pour la suite de ce travail, nous noterons AP l'ensemble des propriétés atomiques utilisées dans la modélisation du système. Les propriétés atomiques sont utilisées dans les états pour distinguer les états qui satisfont ou non certaines propriétés.

Définition 2.2.1 *Une chaîne de Markov à temps discret est un tuple* (S, i, P, L) *où*

- S *est un ensemble fini d'états,*
- $i \in S$ *est l'état initial,*
- $P : S \times S \to [0, 1]$ *est une fonction telle que* $\sum_{s' \in S} P(s, s') \in \{0, 1\}$ *pour tout* s *et*
- $L : S \to 2^{AP}$ *est une fonction d'étiquetage.*

P est représentée par une matrice nommée matrice de transitions qui attribue les probabilités aux transitions. L'élément $P(s, s')$ de la matrice de transitions représente la probabilité d'aller de l'état s à l'état s'. Un état dont la somme des probabilités est 0

est un état terminal. La fonction d'étiquetage L associe à chaque état un ensemble de propositions atomiques, sous-ensemble de l'ensemble des propositions atomiques AP. Les propriétés atomiques sont utilisées pour marquer les états qui ont des propriétés qui nous intéressent. Par exemple, si la proposition atomique danger est associée à un état, il est clair qu'il ne faut pas que notre système se retrouve dans cet état !

Exemple 2.2.2 *La figure 2.3 est un DTMC avec* $S = \{0, 1\}$ *et* $i = 0$. *La matrice* P *et la fonction* L *sont données à la figure 2.4.*

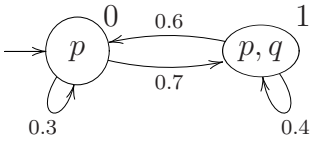


FIG. 2.3 – Exemple d'un DTMC.

$$P = \begin{pmatrix} 0.3 & 0.7 \\ 0.6 & 0.4 \end{pmatrix} \qquad \begin{aligned} L : S &\rightarrow 2^{\{p,q\}} \\ 0 &\mapsto \{p\} \\ 1 &\mapsto \{p, q\} \end{aligned}$$

FIG. 2.4 – Les fonctions P et L du DTMC de la figure 2.3.

Quelques variantes des DTMC sont aussi utilisées. Nous pourrions représenter les états terminaux en ajoutant une boucle avec probabilité 1, de sorte que la somme des probabilités serait toujours 1. Les résultats obtenus seraient les mêmes mais il serait alors impossible de distinguer un état terminal d'un état qui a une boucle sur lui-même avec probabilité 1. Malgré cela, les deux définitions se retrouvent dans la littérature. Il est aussi possible d'ajouter une fonction d'étiquetage des transitions à la définition des DTMC. De cette façon, chaque transition est associée à une action. Cette action représente une interaction entre le système et son environnement. Si on note A l'ensemble des actions, la fonction de transition P devient $P : S \times A \times S \rightarrow [0, 1]$. Cette fonction permet la synchronisation entre plusieurs modèles. Dans cette variante, on exige que pour un état et une action donnée, la somme des probabilités soient inférieure ou égale à un. La figure suivante présente l'exemple simple d'un DTMC avec actions. La section suivante présentera un type de modèle probabiliste à espace continu, les processus de Markov étiquetés. Ces modèles seront très importants pour la suite du travail puisque CISMO, le vérificateur de modèles que nous avons modifié, vérifie des modèles de ce type.

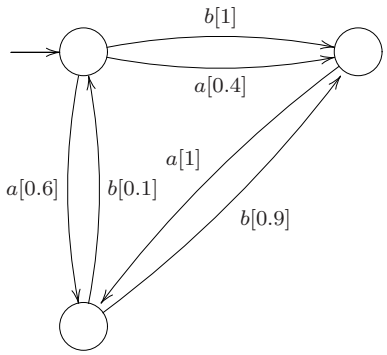


FIG. 2.5 – Exemple d'un DTMC avec actions.

2.3 Processus de Markov Étiquetés (LMP)

Les processus de Markov étiquetés (LMP, de l'anglais *Labelled Markov Processes*) sont utilisés pour représenter des systèmes réactifs et probabilistes ayant un espace d'états continu, c'est-à-dire des systèmes qui réagissent aléatoirement aux événements de leur environnement et qui possèdent une infinité non-dénombrable d'états.

Exemple 2.3.1 *Le système de pilotage d'un avion est un exemple de système qui admet une infinité non-dénombrable d'états. L'état d'un avion dépend de plusieurs paramètres comme sa position, son inclinaison, sa vitesse, etc. Puisque le déplacement de l'avion se fait, fort heureusement, de façon continue, la position de l'avion est un paramètre continu. Le temps est considéré comme discret. En effet, l'ordinateur de bord de l'appareil doit prendre les mesures des différents paramètres mais il ne peut le faire de façon continue, il les lit plutôt à intervalles réguliers. Le système de pilotage doit aussi répondre à des événements comme les augmentations de turbulence et les variations de pression. Ainsi, des paramètres autres que la position, tels la température et la pression, peuvent donner un espace d'états continu.*

Un LMP est un système de transitions étiquetées, c'est-à-dire que chaque transition du modèle est étiquetée par une action. De plus, comme dans tout processus de Markov, les LMP sont probabilistes et ils ont la propriété de Markov, c'est-à-dire que chaque probabilité dépend seulement de l'état présent du système et non pas des états visités précédemment. Les LMP sont aussi une généralisation des DTMC aux espaces d'états continus.

Dans les DTMC, on exige que la somme des probabilités des transitions à un état soit 0 ou 1 mais dans les LMP, on exige plutôt que la somme de ces probabilités soit

comprise entre 0 et 1, inclusivement. Lorsque la somme est strictement comprise entre 0 et 1, nous interprétons la probabilité manquante comme étant la probabilité que le système ne réagisse pas à l'action. Par exemple, il arrive qu'une machine distributrice ne réponde pas à la demande d'un consommateur lorsque celui-ci appuie sur un bouton. Remarquons que dans un LMP, nous pourrions exiger que la somme des probabilités soit de 0 ou de 1. Ce choix dépend de notre interprétation mais il n'a pas d'impact puisque tous les résultats sont valides dans les deux cas. Il faut donc choisir la définition qui convient le mieux à nos besoins.

Dans le cas d'un espace d'états dénombrable, il est possible de donner directement la probabilité de chaque transition d'état à état (par exemple à l'aide d'une distribution de probabilités). Par contre, dans le cas continu, il y a beaucoup de cas où une telle probabilité serait nulle. Nous allons nous intéresser à la probabilité qu'à partir d'un état il soit possible d'aller vers un *ensemble* d'états. Pour ce faire, il faut avoir une mesure de probabilité sur nos ensembles d'états et une σ-algèbre d'ensembles mesurables. Plus de détails sur les σ-algèbres et les mesures de sous-probabilité se trouvent à l'annexe A. Les mesures sont naturelles pour généraliser les distributions de probabilités, par exemple, la mesure d'une union d'ensembles est la somme des mesures des ensembles.

La prochaine définition nous permettra de bien définir un LMP. Dans celle-ci, $\mu(x, \cdot)$ est une mesure de sous-probabilité dont le premier argument est fixé et le deuxième est libre.

Définition 2.3.2 *Soient X un ensemble et Σ une σ-algèbre sur X. Une fonction de transition probabiliste partielle sur un espace mesurable (X, Σ) est une fonction $\mu : X \times \Sigma \to [0, 1]$ telle que $\mu(x, \cdot)$ est une mesure de sous-probabilité et pour chaque $E \in \Sigma$, la fonction $\mu(\cdot, E)$ est mesurable.*

La probabilité $\mu(x, E)$ est une probabilité conditionnelle puisque c'est la probabilité d'aller dans un état de E *sachant* que le système se trouve en x. Par contre, en aucun cas la probabilité ne dépend des états précédents. Voici maintenant la définition formelle d'un LMP.

Définition 2.3.3 *Un LMP est un tuple de la forme $(S, \Sigma, i, Act, label, \{P_a \mid a \in Act\})$ où*

- *$S \in \Sigma$ est l'ensemble des états,*
- *Σ est une σ-algèbre sur S,*

- $i \in S$ *est l'état initial,*
- *Act est l'ensemble des actions,*
- $label : AP \rightarrow \Sigma$ *est la fonction mesurable retournant l'ensemble des états qui satisfont une étiquette donnée et*
- *pour toute action* $a \in Act$, $\mu_a : S \times \Sigma \longrightarrow [0,1]$ *est une fonction de transition probabiliste partielle.*

Dans CISMO, notre application des LMP, nous restreignons toujours l'ensemble d'états à un sous-ensemble de $\mathbb{R}$. Ainsi nous choisissons pour Σ la σ-algèbre de Borel sur $\mathbb{R}$.

La définition originale des LMP ne contenait pas de propositions atomiques sur les états ni de fonction *label*. De plus, dans la littérature, lorsque la fonction *label* est présente, elle associe à chaque état un ensemble de propriétés atomiques, mais pour faciliter l'implémentation de CISMO, son concepteur a préféré utiliser la définition donnée précédemment. Nous définissons donc un LMP avec une fonction *label* qui associe un ensemble d'états à chaque propriété atomique. Les deux définitions sont équivalentes, leur seule différence est au niveau de la convivialité.

2.3.1 Exemple

La figure 2.6 est un exemple de LMP [16] sur lequel nous donnerons des exemples de calculs de probabilités. Cette façon d'illustrer un LMP n'est pas formelle mais elle

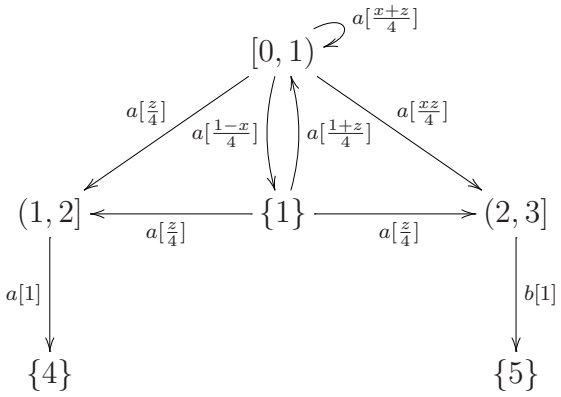


FIG. 2.6 – Exemple d'un LMP.

permet de visualiser le modèle. Dans cet exemple, x est l'état de départ de la transition

et $z \in (0, 1]$. Voici la description formelle de ce même exemple.

L'ensemble d'états S est $[0, 3] \cup \{4, 5\}$. L'état initial est 1, l'ensemble des actions possibles est $\{a, b\}$ et voici la liste des transitions :

- si $x \in [0, 1]$

$$\begin{aligned} P_a(x, [0, z)) &= \frac{x + z}{4} \\ P_a(x, \{1\}) &= \frac{1 - x}{4} \\ P_a(x, (1, 1 + z]) &= \frac{z}{4} \\ P_a(x, (2, 2 + z]) &= \frac{xz}{4} \end{aligned}$$

- si $x \in (1, 2]$

$$P_a(x, \{4\}) = 1$$

- si $x \in (2, 3]$

$$P_b(x, \{5\}) = 1.$$

Certaines probabilités ne sont pas définies : par exemple, pour aller à l'état $\{5\}$ à partir d'un état dans $[0, 1]$. Dans un tel cas nous considérons la probabilité comme étant nulle, c'est-à-dire qu'il n'y a pas de transition possible. Comme Σ est engendré par les ensembles de la forme $[0, z], (1, 1 + z], (2, 2 + z], \{4\}$ et $\{5\}$, $P_a(x, \cdot)$ peut-être étendu à tout Σ par un théorème d'extension classique en théorie de la mesure [8].

Voyons maintenant comment calculer la probabilité qu'a le système de faire certaines transitions. Les exemples de calculs suivants sont tirés du mémoire de Richard [48].

Essayons d'abord de déterminer si l'action a est possible dans l'état initial. Ceci revient à vérifier que la probabilité d'aller vers un état quelconque du système à partir de l'état 1, avec l'action a, est plus grande que zéro. Calculons $P_a(x, S)$,

$$\begin{aligned} P_a(x, S) &= P_a(x, [0, 3]) + P_a(x, \{4\}) + P_a(x, \{5\}) \\ &= P_a(x, [0, 1)) + P_a(x, \{1\}) + P_a(x, [1, 2)) + P_a(x, [2, 3)]) + 0 + 0 \\ &= \frac{x + 1}{4} + \frac{1 - x}{4} + \frac{1}{4} + \frac{x}{4} \\ &= \frac{3 + x}{4}. \end{aligned}$$

Ainsi, pour l'état initial $x = 1$, la probabilité est 1, donc le système répond toujours à l'action a.

Calculons maintenant la probabilité qu'à partir de l'état initial, le système se retrouve dans l'état $\{0\}$. Puisque cette probabilité n'est pas définie explicitement et que le singleton contenant l'élément 0 est l'intersection de intervalles de la forme $[0, z)$, nous calculons la probabilité en faisant tendre z vers 0.

$$\begin{aligned} P_a(1, \{0\}) &= \lim_{z \to 0} P_a(1, [0, z)) \\ &= \lim_{z \to 0} \frac{1+z}{4} \\ &= \frac{1}{4} \end{aligned}$$

Il est tout aussi facile de calculer $P_a(1, [0, 1)) = \frac{1}{2}$ car il suffit de prendre la limite quand x tend vers 1 dans la première branche de la fonction de probabilité.

Maintenant que nous avons un modèle, il nous faut une logique pour spécifier les propriétés à vérifier sur un LMP. La prochaine section décrira l'essentiel de la logique L_0.

2.3.2 Logique L_0

Pour valider un système représenté par un LMP, une façon de faire est d'avoir une logique pour décrire les propriétés que nous souhaitons vérifier. La logique L_0 est utilisée dans la vérification d'un LMP et elle est inspirée de la logique de Larsen et Skou [36]. Cette dernière est une adaptation de la logique de Hennessy et Milner [27]. Voici la syntaxe de la logique L_0 :

$$\phi := \mathsf{T} \mid \mathsf{p} \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \langle a \rangle_q \phi$$

où $a \in Act$ est une action, $\mathsf{p} \in AP$ est une proposition atomique qui représente une propriété d'état et q est un nombre rationnel compris dans l'intervalle $[0, 1)$.

Pour $s \in S$, on écrit $s \models \phi$ si s satisfait la propriété ϕ. Voici maintenant la sémantique de cette logique pour un LMP $L = (S, \Sigma, i, Act, label, \{P_a\})$.

- $s \models \mathsf{T}$ pour tout $s \in S$

- $s \models \mathsf{p}$ si et seulement si $s \in \mathit{label}(\mathsf{p})$
- $s \models \neg\phi$ si et seulement si $s \not\models \phi$
- $s \models \phi_1 \wedge \phi_2$ si et seulement si $s \models \phi_1$ et $s \models \phi_2$
- $s \models \langle a \rangle_q \phi$ si et seulement s'il existe un ensemble $E \in \Sigma$ tel que $\forall s' \in E, s' \models \phi$ et tel que $P_a(s, E) > q$

On dit qu'un modèle L satisfait une propriété ϕ, noté $L \models \phi$, si l'état initial du modèle la satisfait.

Exemple 2.3.4 *Dans la section précédente, nous avons vérifié si l'action a est possible dans l'état initial du LMP de l'exemple 2.6. La logique que nous venons de décrire nous permet d'exprimer cette propriété par* $\langle a \rangle_0 \mathsf{T}$. *Par nos calculs, nous savons que le modèle de notre exemple satisfait* $\langle a \rangle_0 \mathsf{T}$.

La façon habituelle de vérifier une formule de la forme $\langle a \rangle_p \langle b \rangle_q f$ *est de d'abord déterminer les états qui satisfont la formule f. Ensuite on trouve les états pour lesquels il est possible de faire une transition avec l'action b et avec une probabilité de plus de q. On notera l'ensemble de ces états E. Finalement, on vérifie s'il est possible, à partir de l'état initial du système, de se rendre dans un état de l'ensemble E, par l'action a, et avec une probabilité supérieure à p.*

Voyons maintenant si le modèle satisfait la formule $\langle a \rangle_{0.5} \langle b \rangle_{0.6} \mathsf{T}$. *Tous les états satisfont* T, *alors cherchons les états pour lesquels il est possible de faire une transition avec l'action b et avec une probabilité de plus de* 0.6. *Ces états sont ceux de l'intervalle* $(2, 3]$. *À partir de l'état initial du système, il est possible de se rendre dans l'intervalle* $(2, 3]$. *Cependant, ces transitions ont une probabilité d'au plus* 0.25. *Puisque la formule à vérifier indique que les états désirés doivent pouvoir faire une telle transition avec une probabilité supérieure à* 0.5, *on conclut que le modèle ne satisfait pas la formule.*

Dans ce chapitre, nous n'avons pas élaboré sur les algorithmes de vérification puisque cela n'influence pas notre travail. Il y a des algorithmes de vérification qui enregistrent explicitement le modèle et d'autres qui utilisent des méthodes symboliques. Comme nous l'avons mentionné dans l'introduction, les méthodes explicites explorent l'ensemble des états un à un, tandis que les méthodes symboliques traitent des ensembles d'états. Avec ces algorithmes, il est possible pour les modèles de types LMP et la logique L_0 de vérifier si oui ou non un modèle satisfait une formule. Nous introduirons dans le prochain chapitre une méthode symbolique et nous justifierons plus en détail son utilisation.

Chapitre 3

Diagrammes de décision binaire

La vérification d'un système est effectuée en explorant toutes les exécutions possibles de celui-ci, qu'il s'agit d'un programme ou d'une pièce de matériel informatique. Pour ce faire, l'outil de vérification doit analyser chaque état et chaque chemin dans l'espace d'états. D'une manière ou d'une autre, il faut pouvoir mettre en mémoire les états et les transitions qui existent entre ceux-ci. Un des principaux obstacles rencontrés lors de la vérification est le problème d'*explosion du nombre d'états*. Comme nous l'avons dit dans l'introduction, il y a en général un très grand nombre d'états dans les modèles. Ainsi, lorsqu'on représente explicitement les états, il est souvent très difficile d'avoir assez de mémoire et, même quand c'est possible, le temps de traitement d'une telle structure peut être beaucoup trop long. Des approches ont été développées pour diminuer la taille en mémoire des modèles. En 1959, Lee [37] a proposé une structure de données nommée diagramme de décision binaire pour représenter les circuits électriques et, en 1978, Akers [2] a repris cette idée. Mais c'est seulement huit années plus tard, en 1986, que l'intérêt pour les diagrammes de décision binaire est devenu plus important, lorsque Bryant [12] proposa une version modifiée des diagrammes de décision binaire, nommée diagrammes de décision binaire *ordonnés* (OBDD, de l'anglais *Ordered Binary Decision Diagrams*). Cette structure était aussi destinée à représenter des circuits électriques. Par la suite, cette technique a été introduite en vérification de logiciels. Les OBDD ont rapidement gagné en popularité et ils ont permis la vérification de modèles contenant beaucoup plus d'états. Étant donnés les bons résultats obtenus par l'utilisation des OBDD dans la vérification du matériel informatique, il y a de plus en plus d'outils de vérification qui permettent la vérification *symbolique* avec OBDD, par exemple les vérificateurs de modèles SMV, PRISM, etc.

Nous décrirons dans ce chapitre les diagrammes de décision binaire ordonnés ainsi que les algorithmes de manipulation de base. De plus, nous expliquerons les avantages et

désavantages de l'utilisation des diagrammes de décision binaire ordonnés. Nous verrons que la taille de cette structure peut varier et qu'il y a des façons de la réduire. Nous terminerons en donnant des idées d'heuristiques pouvant réduire la taille des OBDD.

3.1 Représentation des fonctions booléennes

Il est bien connu que les tables de vérité peuvent servir à représenter les fonctions booléennes, c'est-à-dire les fonctions de la forme $f : \{0,1\}^n \to \{0,1\}$. Malheureusement, cette représentation occupe un espace qui est exponentiel par rapport au nombre de variables d'entrées. En effet, pour toutes fonctions ayant n variables booléennes, il y a 2^n lignes dans la table de vérité. C'est donc dire que même pour une fonction valide, une fonction qui est toujours vraie, mettre la fonction en mémoire à l'aide de sa table de vérité prend beaucoup d'espace. Il est nécessaire d'avoir une méthode plus compacte pour représenter les fonctions booléennes. Dans ce qui suit, 0 et 1 représentent respectivement les valeurs de vérité *faux* et *vrai*.

L'utilisation des diagrammes de décision binaire (BDD) est une autre façon de représenter les fonctions booléennes. La représentation par diagramme de décision binaire est inspirée de la représentation de Shannon pour les fonctions booléennes. Cette représentation nous permet, pour un i fixé, de réécrire la fonction $f : \{0,1\}^n \to \{0,1\}$ ainsi :

$$f(x_1, x_2, ..., x_n) = (x_i \wedge f[x_i := 1]) \vee (\neg x_i \wedge f[x_i := 0])$$

où $f[x_1 := 1]$ veut dire $f(1, x_2, ..., x_n)$, c'est-à-dire la valeur de la fonction lorsque $x_1 = 1$, et similairement pour $f[x_1 := 0]$. Il est plus facile de comprendre les diagrammes de décision binaire par un exemple, ce que nous ferons à l'aide de l'exemple suivant.

Exemple 3.1.1 *La figure 3.1 montre le BDD de la fonction* $f(x_1, x_2) = x_1 \wedge x_2$. *Les arcs pointillés sortant d'un noeud étiqueté par la variable* v *signifient que la variable* v *prend la valeur* 0 *et les arcs pleins signifient que la variable* v *prend la valeur* 1. *Ainsi,*

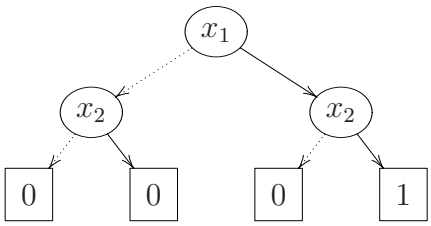


FIG. 3.1 – Exemple d'un diagramme de décision binaire.

pour déterminer $f(0,1)$*, nous devons suivre l'arc pointillé à partir du noeud* x_1 *vers le noeud* x_2 *car* $x_1 = 0$*. Ensuite, du noeud* x_2 *nous suivons l'arc plein (puisque* $x_2 = 1$*) pour atteindre une feuille étiquetée* 0*. Nous savons alors que* $f(0,1) = 0$*. Comparons le BDD avec la représentation de Shannon. Regardons encore la valeur de la fonction en* $x_1 = 0$ *et* $x_2 = 1$*. Elle est déterminée ainsi :*

1. *décomposition de Shannon de* $f(x_1, x_2)$ *avec* $x_1 = 0$ *:*

$$f(0,1) = (0 \wedge f[x_1 := 1]) \vee (\neg 0 \wedge f[x_1 := 0]),$$

   remarquons que la fonction $f[x_1 := 1]$ *représente le sous-graphe de droite de la figure 3.1,*

2. *simplifications :*

$$f(0,1) = 1 \wedge f[x_1 := 0] = f[x_1 := 0]$$

3. *décomposition de Shannon de* $f(0, x_2)$ *avec* $x_2 = 1$ *:*

$$f(0,1) = (1 \wedge f[x_1 := 0, x_2 := 1]) \vee (\neg 1 \wedge f[x_1 := 0, x_2 := 0]),$$

4. *simplifications :*

$$f(0,1) = f[x_1 := 0, x_2 := 1] = 0$$

À la ligne 1*, l'équation nous indique de prendre* $f[x_1 := 0]$ ou $f[x_1 := 1]$ *selon la valeur de* x_1*. C'est exactement ce que l'on fait dans le BDD. Il faut choisir la branche pointillée* ou *la branche pleine selon que* $x_1 = 0$ *ou* $x_1 = 1$*. De même à la ligne* 3*, le choix est fait selon la valeur de* x_2 *comme dans le diagramme de décision binaire. Nous comprenons ainsi que la représentation par diagramme de décision binaire a exactement la même signification que la représentation de Shannon.*

Dans la littérature, on voit souvent les BDD avec les transitions non orientées. Cette façon de faire n'est pas ambiguë car le BDD est toujours parcouru de haut en bas. Dans ce mémoire, nous avons choisi de laisser les transitions orientées, nous pouvons alors parler d'arcs. Voyons maintenant une définition plus exacte d'un BDD et la définition d'une valuation, un terme qui sera utile par la suite.

Définition 3.1.2 *Un diagramme de décision binaire est un arbre. À l'exception des feuilles, il sort de chaque noeud deux arcs et chaque noeud est étiqueté par une variable booléenne. Tous les noeuds qui sont à la même distance de la racine sont étiquetés par la même variable booléenne. Les feuilles sont étiquetées par* 0 *ou* 1*.*

Définition 3.1.3 (Valuation) *Une valuation d'une fonction booléenne est l'affectation d'une valeur à chaque variable de la fonction.*

Dans un BDD représentant la fonction f, pour un noeud dont l'étiquette est x, il y a deux arcs sortants. Un qui pointe vers un sous-BDD qui représente la fonction f avec $x = 0$ et l'autre pointant sur un sous-BDD représentant la fonction f avec $x = 1$. Graphiquement, nous distinguons les deux arcs en mettant celui de $x = 0$ en pointillé et l'autre plein. Lorsqu'un chemin est parcouru de la racine à une feuille, toutes les valeurs des variables rencontrées sont fixées et l'étiquette de la feuille représente la valeur de la fonction avec cette valuation de variables.

Une fois encore la représentation de f est de taille exponentielle car le nombre de noeuds dans un BDD est $\sum_{i=0}^{i=n-1} 2^i = 2^n - 1$, n étant le nombre de variables. Lorsque nous déterminons la taille d'un tel arbre, nous comptons seulement les noeuds étiquetés par les variables. Remarquons qu'il y a une redondance d'information dans les diagrammes de décision binaire : dans l'exemple 3.1.1, il y a trois feuilles étiquetées 0. Il aurait été plus « économique » en terme d'espace, de n'avoir qu'une seule feuille étiquetée 0. Ensuite, le noeud étiqueté x_2 qui suit l'arc pointillé sortant de x_1 a pour seul enfant la feuille 0, c'est donc dire que la valeur de la variable x_2 n'a pas d'impact sur la valeur de la fonction lorsque $x_1 = 0$. Nous pourrions ainsi omettre ce noeud. Dans l'exemple 3.1.1, ces deux règles suffisent à réduire l'arbre mais en général il peut arriver qu'une autre simplification soit possible : en effet, si deux noeuds ont les mêmes fils, il est préférable de garder une seule copie de ces noeuds. Nous avons décrit ici les idées générales derrière les réductions données par Bryant lorsqu'il a défini les OBDD. Leur définition formelle se trouve dans la section suivante.

3.2 Diagrammes de décision binaire ordonnés (OBDD)

En utilisant 3 règles de réduction, il est possible de transformer un diagramme de décision binaire en un graphe qui représente la même fonction booléenne mais qui est de taille moindre. De plus, cette représentation est unique (nous verrons que l'unicité dépend de l'ordre des variables). Cette structure est appelée un diagramme de décision binaire ordonné (OBDD). Nous présentons ici la définition des OBDD, des exemples d'applications des règles de réduction ainsi que certaines opérations utiles sur cette structure. Les résultats énoncés ne seront pas démontrés puisqu'il s'agit de résultats classiques dans le domaine de la vérification symbolique et, pour la même raison, les

algorithmes ne seront pas tous donnés explicitement.

Définition 3.2.1 *Soient S un ensemble d'états, V un ensemble non-vide de variables booléennes, $<$ un ordre total sur les variables de V et $var : S \to V \cup \{0,1\}$ une fonction qui associe au sommet $s \in S$ une variable. Un OBDD sur $\langle V, < \rangle$ est un graphe orienté acyclique avec un ensemble non-vide S de sommets qui sont de deux types :*

- *des sommets terminaux s étiquetés par une valeur booléenne, $var(s) = 0$ ou $var(s) = 1$ et*
- *des sommets non-terminaux s étiquetés par une variable booléenne, $var(s) \in V$, qui ont deux enfants left(s) et right(s) tels que pour tout sommet t :*

$$t \in \{left(s), right(s)\} \Rightarrow ((var(s) < var(t)) \text{ ou } (t \text{ est terminal})).$$

La deuxième condition exige que chaque variable soit rencontrée au plus une seule fois et dans le même ordre pour tous les chemins qui commencent à la racine et qui se terminent à une feuille. C'est pour cette raison qu'on dit que le BDD est ordonné (**O**BDD). Tous les exemples de BDD présentés précédemment étaient en fait des OBDD. La figure 3.2 est un exemple de BDD non ordonné. Remarquons qu'un OBDD n'est plus nécessairement

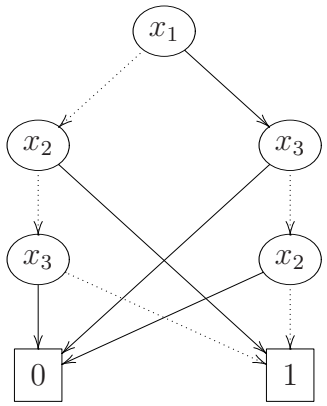


FIG. 3.2 – Exemple d'un graphe orienté sans cycle non-ordonné.

un arbre contrairement aux BDD car la version non-orientée du graphe contient des cycles. C'est pourquoi un OBDD est défini comme un graphe orienté sans cycle. Nous discuterons de l'importance de l'ordre des variables dans la prochaine section. Cette définition est moins restreignante que celle des BDD et elle nous permet d'imposer des contraintes aux OBDD pour que ceux-ci aient une taille plus petite.

3.2.1 Règles de réduction des OBDD

Pour réduire la taille d'un BDD, Bryant [12] a ajouté trois contraintes à la définition des OBDD [3.2.1]. Voici ces trois contraintes :

1. pour tous noeuds terminaux s et t,
$$var(s) = var(t) \Rightarrow s = t$$
2. pour tout noeud s,
$$left(s) \neq right(s)$$
3. pour tous noeuds s et t,
$$((var(s) = var(t)) \wedge (left(s) = left(t)) \wedge (right(s) = right(t))) \Rightarrow s = t$$

Ces trois contraintes sont appelées les règles de réduction parce qu'à chacune correspond une transformation du OBDD pour que celui-ci respecte la contrainte. Un OBDD est réduit lorsque toutes les règles de réduction ont été appliquées et qu'aucune d'elles ne peut encore changer le graphe (nous verrons plus loin que l'ordre d'application n'est pas important). Les OBDD réduits sont souvent notés **R**OBDD de l'anglais *Reduced OBDD*. Dans ce texte, lorsque nous parlerons d'un OBDD, il sera réduit.

Voyons à l'aide d'exemples ce que les règles de réduction impliquent. Pour appliquer la règle 1, il suffit de conserver une seule copie de chaque feuille. Nous voyons sur la figure 3.3 une utilisation de la règle 1. Il ne doit y avoir que sommets terminaux ayant les valeurs 0 et 1. Pour appliquer la règle 2, il faut éliminer les noeuds dont les deux

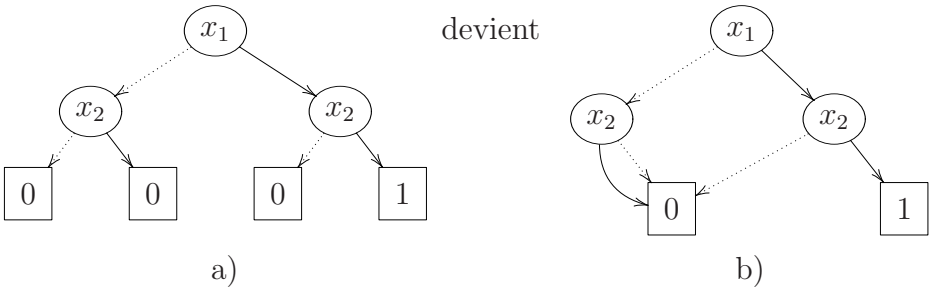


FIG. 3.3 – Exemple de réduction par la règle 1.

arcs sortants ont la même destination. La figure 3.4 nous montre une utilisation de la règle 2 sur le graphe b) de la figure 3.3. Puisque les deux arcs sortants du noeud x_2 de gauche pointent sur 0, il faut enlever ce noeud et rediriger l'arc pointillé, sortant de x_1,

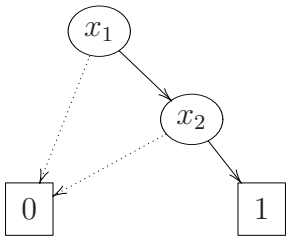


FIG. 3.4 – Exemple de réduction par la règle 2.

vers l'unique enfant de x_2. Pour appliquer la règle 3, nous devons conserver une seule copie des noeuds qui ont des enfants étiquetés par les mêmes variables. Cette règle sert donc à éliminer les sous-graphes isomorphes. L'OBDD de la figure 3.4 est réduit car les trois contraintes sont satisfaites. Réduisons maintenant l'OBDD de la figure 3.5. Pour

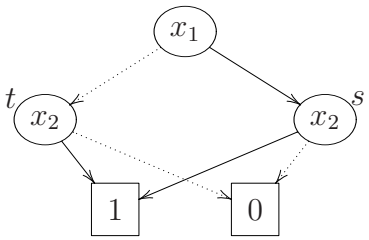


FIG. 3.5 – Exemple de graphe contenant deux sous-graphes isomorphes.

mieux les distinguer, nous nommerons t le noeud de gauche étiqueté x_2 et s celui de droite. Remarquons que les sous-graphes de la figure 3.5 qui ont comme racine t et s respectivement, sont isomorphes. En effet, l'arc pointillé du noeud t pointe vers 0 et son arc plein vers 1. De même pour le noeud s. Par la règle 3, il faut garder une seule copie de ces sous-graphes. Nous obtenons alors l'OBDD de la figure 3.6. Il reste à appliquer

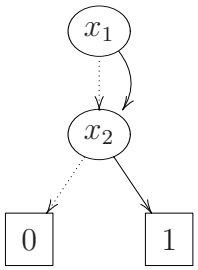


FIG. 3.6 – Exemple de réduction par la règle 3.

la règle 2 sur ce graphe pour obtenir un OBDD complètement réduit, ce qui donne le graphe de la figure 3.7.

L'algorithme suivant sert à réduire un BDD. La réduction se fait en traversant le BDD de bas en haut en donnant des étiquettes à chaque noeud. L'étiquette du noeud

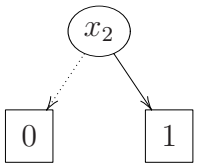


FIG. 3.7 – OBDD réduit.

v est notée $id(v)$ et E est l'ensemble des étiquettes.

Algorithme 3.3.3 Réduction(BDD B)

Pour toute feuille v
  Si $var(v) = 0$ alors $id(v) = 0$
  Si $var(v) = 1$ alors $id(v) = 1$
Fin Pour tout
Pour toute hauteur à partir de la hauteur 1
  Pour tout sommet v
    Si $id(left(v)) = id(right(v))$ alors $id(v) = id(left(v))$
    Sinon
      Pour tout v'
        Si $var(v) = var(v')$, $id(left(v')) = id(left(v))$ et $id(right(v')) = id(right(v))$
          alors $id(v) = id(v')$
        Sinon
          $id(v) = x$ où $x \in E$
          $E := E \setminus \{x\}$
      Fin Pour tout
  Fin Pour tout
Fin Pour tout
Fin algorithme

Après l'étiquetage des noeuds, nous gardons une seule copie des noeuds qui ont la même étiquette. La réduction se fait en $O(|B| \cdot \log |B|)$ où $|B|$ représente la taille du BDD B [30]. Cette façon de faire est simple mais elle nécessite d'avoir le BDD qui représente la fonction booléenne pour pouvoir le réduire et ce BDD peut être très gros. Il est possible aussi de construire et de réduire l'OBDD en même temps. La description de cette technique se fera plus en détail lorsque nous discuterons de l'implémentation des outils de manipulation des OBDD.

3.3.2 Ordre des variables et taille des OBDD

La taille de l'OBDD pour une fonction booléenne donnée dépend essentiellement de l'ordre des variables dans celui-ci. Par exemple, la fonction $f(x_0, x_1, x_2, x_3) = (x_0 \vee x_1) \wedge (x_2 \vee x_3)$ avec l'ordre de variable $x_0 < x_1 < x_2 < x_3$ est représentée par l'OBDD de la figure 3.8. Par contre, si pour la même fonction nous choisissons l'ordre $x_0 < x_2 <$

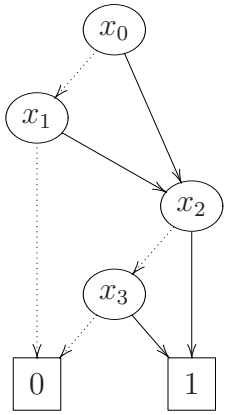


FIG. 3.8 – OBDD de la fonction f avec $x_0 < x_1 < x_2 < x_3$.

$x_1 < x_3$, nous obtenons un OBDD de taille plus grande, celui de la figure 3.9. Plus

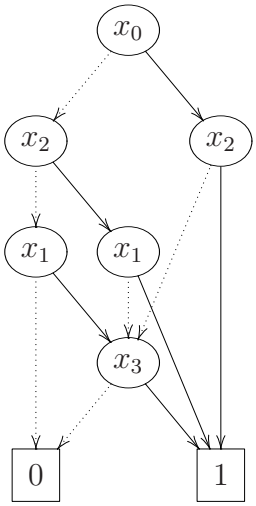


FIG. 3.9 – OBDD de la fonction f avec l'ordre $x_0 < x_2 < x_1 < x_3$.

généralement, si la fonction est $g(x_0, ..., x_{2n-1}) = (x_0 \vee x_1) \wedge ... \wedge (x_{2n-2} \vee x_{2n-1})$ et que nous choisissons l'ordre $x_0 < x_2 < x_4 < ... < x_1 < x_3 < ... < x_{2n-1}$ nous aurons un OBDD de taille $2^{n+1} - 2$, c'est-à-dire toujours exponentiel (exemple tiré de Huth et Ryan [30]). Tandis qu'avec l'ordre $x_0 < x_1 < ... < x_{2n-1}$, nous aurons $2n$ noeuds dans

l'OBDD (attention, ici n n'est pas le nombre de variables). L'ordre des variables joue donc un rôle très important dans la taille d'un OBDD. Malheureusement, déterminer l'ordre qui minimise la taille de l'OBDD est un problème $\mathcal{NP}$-*Difficile* [10]. De plus, il a été démontré qu'on peut faire une approximation de l'ordre optimal avec un certain ratio de façon polynomiale si et seulement si $\mathcal{P} = \mathcal{NP}$ [52]. Le meilleur algorithme connu pour résoudre ce problème s'exécute en $O(3^n \cdot n^2)$ [19], n étant le nombre de variables de l'OBDD. Il est généralement inutilisable sur des fonctions qui ont plus de 25 variables. Il faut alors se rabattre sur des heuristiques pour déterminer un ordre qui rendra l'OBDD plus petit. Nous discuterons de certaines heuristiques qui peuvent diminuer la taille des OBDD à la fin de ce chapitre.

Remarquons que puisque nous n'obtenons pas le même OBDD selon l'ordre des variables, il peut y avoir deux OBDD différents qui représentent la même fonction. Le théorème suivant nous assure cependant que pour un ordre de variable donné, la représentation par OBDD est canonique [12]. Dans ce qui suit, f_B est la fonction booléenne que l'OBDD B représente.

Théorème 3.3.2 *Soit $V = \{x_0, x_1, ..., x_n\}$ un ensemble de variables et $<$ un ordre total sur V. Pour toute paire de OBDD B et B' sur $\langle V, < \rangle$ on a*

$$f_B = f'_B \Leftrightarrow B \text{ et } B' \text{ sont isomorphes.}$$

Une conséquence du théorème 3.3.2 est que l'ordre d'application des règles de réduction n'a pas d'importance puisqu'une fois réduit, l'OBDD obtenu sera toujours le même.

La taille des tables de vérité et des BDD ne dépend pas de la fonction qu'ils représentent mais plutôt du nombre de variables dans celle-ci. Ce n'est cependant pas le cas avec les OBDD ; ceux-ci n'ont pas toujours une taille exponentielle par rapport au nombre de variables. Ainsi, l'ordre de grandeur de la taille d'un OBDD *peut* être moindre que celui des tables de vérité. Malheureusement, la représentation reste exponentielle en pire cas.

Nous avons donc une représentation *canonique* des fonctions booléennes dont la taille peut être polynomiale par rapport au nombre de variables d'entrées.

3.3.3 Manipulations des OBDD

Les tests

Quand on définit une logique, on s'intéresse à l'équivalence sémantique, à la satisfiabilité et à la validité des formules. Voyons comment ces tests peuvent être faits sur les OBDD.

La canonicité des OBDD est essentielle pour pouvoir effectuer certains tests de façon très efficace : si deux OBDD représentant des fonctions f_1 et f_2 ne sont pas isomorphes pour un ordre des variables donné, nous pouvons déduire immédiatement que f_1 et f_2 ne sont pas sémantiquement équivalentes. Par exemple, si nous souhaitons savoir si $f_1 = x_0 \wedge (x_1 \vee x_2)$ est sémantiquement équivalente à $f_2 = (x_0 \wedge x_1) \vee x_2$, nous pouvons nous fixer un ordre des variables et construire l'OBDD représentant chacune des fonctions. La figure 3.10 a) représente l'OBDD de la fonction f_1 et la figure 3.10 b) représente l'OBDD de la fonction f_2. Puisque ces graphes ne sont pas isomorphes, nous

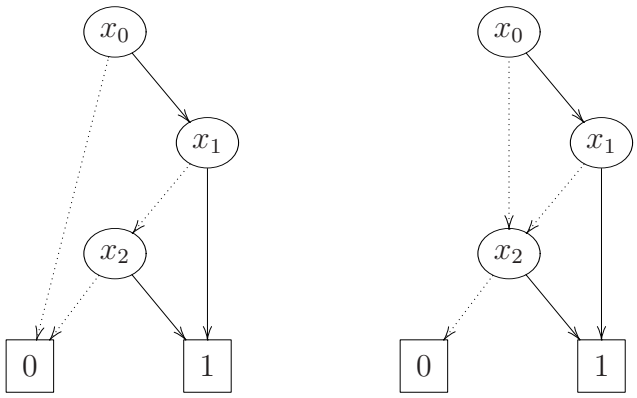


FIG. 3.10 – OBDD des fonctions f_1 et f_2 avec l'ordre $x_0 < x_1 < x_2$.

pouvons déduire que la fonction f_1 est différente de la fonction f_2.

Les tests de satisfiabilité et de validité peuvent aussi être faits efficacement. Déterminer si une formule est satisfiable revient seulement à vérifier que l'OBDD correspondant ne consiste pas seulement en la feuille étiquetée 0. Pour le problème de validité, il suffit de vérifier si l'OBDD est constitué seulement de la feuille étiquetée 1. Si ce n'est pas le cas, la formule n'est pas valide.

Nous savons que les problèmes de satisfiabilité et de validité d'une formule booléenne

sont $\mathcal{NP}$-*Complet* [21]. Ici les solutions à ces problèmes sont très faciles à obtenir car nous supposons que nous avons l'OBDD de la formule. Cependant, obtenir cet OBDD peut être très difficile.

Les opérations

Les formules de logique sont souvent composées de sous-formules liées par des opérateurs. Divers algorithmes ont été développés pour effectuer ces opérations sur les OBDD. Les algorithmes présentés ici sont ceux de Bryant [12] et de Huth et Ryan [30]. Dans ce qui suit, nous ne parlerons pas de ce qu'il advient des noeuds qu'on élimine dans les OBDD. Une question à se poser est : est-ce qu'on détruit immédiatement ces noeuds ou si on les conserve ? Nous répondrons à cette question lorsque nous discuterons des méthodes d'implémentation.

Négation Ayant l'OBDD O représentant une fonction f, pour déterminer l'OBDD de $\neg f$ nous n'avons qu'à échanger les deux feuilles 0 et 1 de O. Il est clair que cette opération donne bien la négation voulue et que l'OBDD est réduit.

Opérations binaires La méthode *apply* est utilisée pour calculer des opérations binaires, comme le *et* et le *ou* logique, entre deux OBDD. Étant donnés deux OBDD B_f et B_g, représentant les fonctions f et g, l'appel de la fonction $apply(op, B_f, B_g)$ avec $op \in \{\vee, \wedge, \oplus, ...\}$, retourne l'OBDD réduit représentant la fonction booléenne f op g. L'algorithme s'exécute de façon récursive sur les deux OBDD de la façon suivante :

Algorithme $apply(op, B_f, B_g)$

1. Soit x la première variable (selon l'ordre de variable) qui apparaît dans B_f ou B_g.
2. On applique l'algorithme récursivement sur les deux sous-graphes du noeud de cette variable, celui de la branche pointillée et celui de la branche pleine, si ces deux noeuds sont étiquetés par la même variable. Si telle n'est pas le cas, l'algorithme est appliqué sur le noeud dont l'étiquette est la plus petite selon l'ordre des variables.
3. aux feuilles, on applique l'opération binaire op directement
4. si nécessaire, on réduit l'OBDD résultant.

L'algorithme implémenté de façon récursive a un temps exponentiel. Cependant, lorsque la programmation dynamique est utilisée, le temps d'exécution de la fonction

apply appliquée à deux OBDD est $O(|B_f| \cdot |B_g|)$, donc linéaire par rapport à la taille des OBDD.

Restriction de variables L'algorithme *restrict* sert à fixer la valeur d'une variable dans l'OBDD. Ainsi, $restrict(B, x_i, b)$ retourne l'OBDD B dans lequel la valeur de la variable x_i a été fixée à la valeur booléenne b. Cette opération se fait facilement en dirigeant tous les arcs pointant sur chaque noeud n étiqueté par x_i vers le fils gauche (arc pointillé) de n si $b = 0$ et vers le fils droit (arc plein) de n si $b = 1$. Avec les méthodes utilisées à l'implémentation, présentées à la fin du chapitre, l'OBDD obtenu est réduit. Cette méthode prend un temps en $O(|B|)$.

Abstraction de variables Un autre algorithme utile est l'algorithme *exists*. Celui-ci sert à *supprimer* des variables dans un OBDD en considérant les deux valeurs possibles pour ces variables, c'est-à-dire, $exists(B_f, x_i) = f([x_i := 0]) \vee f([x_i := 1])$. Énoncé ainsi, il est facile de voir que calculer $apply(\vee, restrict(B, x_i, 0), restrict(B, x_i, 1))$ nous donne $exists(B, x_i)$, il est cependant possible d'améliorer l'algorithme en remplaçant directement dans l'OBDD les noeuds étiquetés x_i par le résultat de l'application du $\vee$ aux deux fils des noeuds.

En plus des articles de Bryant, plusieurs documents offrent une bonne documentation sur les OBDD. Par exemples, les livres *Representations of Discrete Functions* [50] et *Algorithms and Data Structures in VLSI Design* [41] sont de bonnes références et plus particulièrement *Binary Decision Diagrams and Applications for VLSI CAD* [43] et *Binary Decision Diagrams : Theory and Implementation* [17].

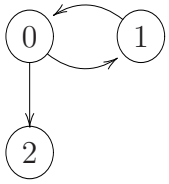
Dans la section suivante, nous présenterons une utilisation des OBDD de plus en plus populaire : la représentation des graphes à l'aide d'OBDD. Ensuite, nous donnerons certain aspects intéressants de l'implémentation des outils de manipulation des OBDD.

3.4 Représentation symbolique d'un graphe

Dans plusieurs applications informatiques il est important de pouvoir enregistrer des graphes en utilisant le moins de mémoire possible. Comme nous l'avons mentionné précédemment, la modélisation des systèmes se fait à l'aide de graphes dont les sommets représentent les états du système et les arcs représentent les transitions possibles à partir de chaque état. Il y a deux façons traditionnelles de mettre un graphe en mémoire, la

première est l'utilisation de la matrice de transitions (ou d'adjacence) du graphe et la seconde est l'utilisation des listes chaînées.

La première méthode utilise la matrice T définie par $T(i,j) = 1$ s'il existe une arête entre les sommets i et j (on dit alors que i et j sont voisins) et $T(i,j) = 0$ sinon. La matrice de la figure 3.11 b) est la matrice d'adjacence du graphe présenté en a). Il est



a)

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

b)

FIG. 3.11 – Matrice de transitions.

très fréquent que le graphe n'ait pas beaucoup d'arêtes. Dans de tels cas, la matrice d'adjacence contient beaucoup de 0 et est dite creuse. La représentation par matrice de transitions prend beaucoup trop d'espace quand la matrice est creuse car l'information nécessaire pour savoir entre quels sommets il y a une arête n'est pas la seule information enregistrée. À chaque fois qu'il n'y a pas d'arête entre deux sommets, on enregistre un 0. Cette dernière information prend plus d'espace que la précédente et elle n'est pas très utile. Cependant, avec cette représentation, plusieurs manipulations sur le graphe sont faciles à effectuer. Par exemple, il est facile de voir qu'il existe un chemin de longueur deux entre le sommet i et le sommet j si la valeur de l'élément (i,j) dans la matrice T^2 est non-nulle.

Une deuxième façon de mettre un graphe en mémoire est l'utilisation des listes chaînées pour représenter le graphe. Pour ce faire, l'ensemble des sommets est mis dans une liste chaînée L dont chaque noeud est un sommet du graphe. À chaque noeud de L, une nouvelle liste chaînée l_i débute, dont les noeuds représentent les sommets voisins du sommet i de la liste L. La figure 3.12 est une représentation du graphe de la figure 3.11 à l'aide d'une liste chaînée. Cette méthode peut théoriquement donner des bons résultats mais elle est moins utilisée que les méthodes présentées dans les sections suivantes. Ce manque de popularité peut être dû au fait que les algorithmes pour traiter les listes chaînées sont parfois plus complexes et que le stockage des structures et des pointeurs augmente la quantité de mémoire utilisée.

Nous présenterons deux méthodes de stockage de matrices basées sur l'utilisation de la matrice de transitions. La méthode standard est sans doute la plus populaire et est surtout utilisée en mathématiques appliquées. La deuxième méthode en est une

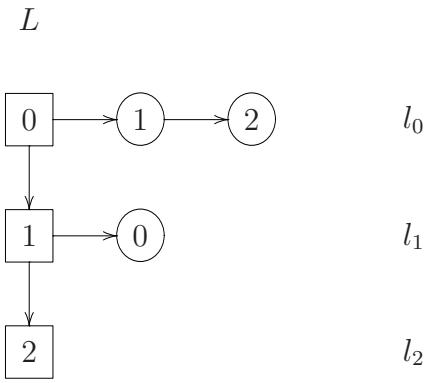


FIG. 3.12 – Exemple de liste chaînée.

symbolique, qui utilise les OBDD pour représenter les états et les transitions du graphe. Cette dernière est de plus en plus utilisée, spécialement dans le domaine de la vérification et de l'apprentissage automatique en intelligence artificielle.

3.4.1 Méthode standard de représentation des matrices creuses

Le défaut de la représentation par matrice de transitions est que stocker directement la matrice prend beaucoup de mémoire. Nous cherchons donc à stocker cette matrice de façon plus efficace. Le problème de stockage des matrices creuses est très connu en mathématiques appliquées. D'une certaine façon, l'utilisation des listes chaînées peut être vue comme une représentation plus compacte des matrices creuses.

La méthode standard attaque le problème en éliminant les éléments nuls de la matrice. Ainsi, seuls les éléments non-nuls sont conservés en mémoire. Voyons cette méthode à l'aide d'un exemple.

Exemple 3.4.1 *La figure 3.13 montre une matrice creuse M à mettre en mémoire. Pour mettre cette matrice en mémoire, nous utilisons trois tableaux A, B et C. Le*

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 7 & 0 & 0 & 5 \end{pmatrix}$$

FIG. 3.13 – Exemple d'une matrice creuse.

tableau A contient la valeur des éléments non-nuls. Les éléments sont mis dans le tableau A ligne par ligne, en gardant l'ordre d'occurrence dans la matrice. Voici le tableau A de la matrice de la figure 3.13 :

$$A \quad \begin{bmatrix} 1 & 2 & 1 & 7 & 5 \end{bmatrix}$$

Dans le tableau B, à la position i, on indique la colonne dans la matrice de l'élément i du tableau A. Voici le tableau B du même exemple :

$$B \quad \begin{bmatrix} 3 & 1 & 3 & 1 & 4 \end{bmatrix}$$

Finalement, dans le tableau C, on met à la case i un pointeur sur l'élément du tableau A qui est le premier à apparaître sur la ligne i. Voici le tableau C de la matrice de la figure 3.13 :

$$C \quad \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

Lorsque la matrice est suffisamment creuse, cette méthode permet de beaucoup diminuer l'espace occupé. Dans cet exemple, la taille est seulement réduite de deux entrées. Par contre, lorsque la dimension de la matrice est très grande, la quantité d'espace sauvé est significative. En fait, si e est le nombre d'éléments non-nuls dans la matrice $n \times n$, la taille de la représentation par la méthode standard sera de $2e + n$, car le tableau A et le tableau B contiennent e éléments, et C a autant d'éléments que de la matrice a de lignes. Ainsi, lorsque $2e + n < n^2$, il est préférable d'utiliser cette technique. Cette méthode, ainsi que certaines variantes de celle-ci, sont très souvent utilisées en pratique, et ce, avec d'assez bons résultats. Pour plus de détails sur cette méthode de stockage des matrices creuses, voir Saad [49].

Un problème avec la méthode standard est que si la matrice contient plusieurs fois la même valeur, chaque valeur est gardée en mémoire plusieurs fois. Ainsi, dans le cas de la matrice de transitions, comme elle contient seulement des 0 et des 1, le tableau A contiendra seulement des 1 et cette redondance engendre une utilisation inutile de la mémoire. Un des objectifs de la prochaine méthode présentée est de mettre en mémoire une seule fois chaque valeur.

3.4.2 Représentation symbolique avec OBDD

Nous allons utiliser les OBDD pour représenter la matrice de transitions d'un graphe. Cette méthode peut s'appliquer à n'importe quel type de matrice mais notre utilisation sera seulement appliquée aux matrices de transitions. Puisque les OBDD servent

à représenter les fonctions booléennes, afin de les utiliser dans l'enregistrement d'un graphe, nous devons être en mesure de décrire l'ensemble des états et des transitions du graphe par une fonction booléenne.

Pour déterminer cette fonction, nous numérotons chaque état du modèle. S'il y a une transition entre l'état 2 et l'état 5, mais pas entre les états 2 et 4, nous voulons que la fonction f qui représente l'ensemble des transitions soit telle que $f(2,5) = 1$ et $f(2,4) = 0$. Cette fonction est donnée par la matrice de transitions. Pour transformer cette fonction en fonction booléenne, nous établissons une bijection entre les ensembles $\{x \in \mathbf{N} : x < 2^{\lceil log_2 |S| \rceil}\}$ et $\{0,1\}^{\lceil log_2 |S| \rceil}$, où $|S|$ est le nombre d'états dans le graphe.

La bijection la plus utilisée est certainement la représentation en base 2 des nombres. Ainsi, s'il y a au plus 8 états, puisque la représentation binaire de 2 est 010, celle de 4 est 100 et celle de 5 est 101, la fonction f doit être telle que $f(010,101) = 1$ et $f(010,100) = 0$. Nous obtenons donc une fonction booléenne $f : \{0,1\}^{2\lceil log_2 |S| \rceil} \rightarrow \{0,1\}$ qui représente les transitions d'un graphe.

Nous utiliserons la représentation binaire tout au long de ce travail mais il aurait été possible de faire autrement. Par exemple, nous aurions pu utiliser le code Gray. Le code Gray est utilisé pour représenter des nombres entiers par une suite finie de 0 et de 1 de telle sorte que d'un entier au suivant il n'y a qu'un seul bit qui change.

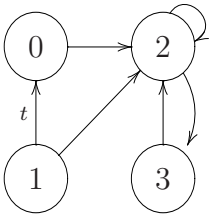
Nous pouvons maintenant voir la matrice de transitions $n \times n$ comme une fonction booléenne $f : \{0,1\}^{2\lceil log_2 (n) \rceil} \rightarrow \{0,1\}$:

$$f(x_1, ..., x_{\lceil log_2 (n) \rceil}, y_1, ..., y_{\lceil log_2 (n) \rceil}) = \begin{cases} 1 & \text{s'il existe un arc entre les sommets } \bar{x} \text{ et } \bar{y} \\ 0 & \text{sinon} \end{cases}$$

et cette fonction est entièrement définie par la matrice de transitions. $\bar{x}$ et $\bar{y}$ représentent respectivement une valuation sur les variables x_i et une valuation sur les variables y_i.

Une transition est identifiée par son état de départ et son état d'arrivée. Pour pouvoir distinguer ces deux types d'états, nous noterons les états de départ par les variables booléennes x_i et ceux d'arrivée par y_i. Comme nous pouvons utiliser les OBDD pour représenter les fonctions booléennes, nous pouvons aussi les utiliser pour représenter les transitions d'un graphe. L'exemple suivant montre comment représenter les transitions du graphe de la figure 3.14 à l'aide d'un OBDD. En particulier, nous y montrons comment déterminer une forme propositionnelle qui s'évalue avec les mêmes valeurs de vérité que la fonction déterminée par la matrice de transitions. Ainsi, en utilisant la méthode *apply*, il sera facile de construire l'OBDD.

Exemple 3.4.2 *À la figure 3.14, nous pouvons voir un graphe et sa matrice de transitions sur laquelle nous avons indiqué la numérotation des lignes et des colonnes en notation binaire. Nous voulons représenter ce graphe à l'aide d'un OBDD. La transi-*



	00	01	10	11
00	0	0	1	0
01	1	0	1	0
10	0	0	1	1
11	0	0	1	0

FIG. 3.14 – Un graphe et sa matrice de transitions.

tion t de la figure 3.14 va de l'état 1 à l'état 0. Comme il y a 4 états dans le graphe, nous encodons les numéros d'états sur deux bits et donc nous écrivons 1 en notation binaire 01. Nous avons donc que la variable x_1 prend comme valeur 0 et la variable x_2 la valeur 1. Pour noter qu'une variable v prend la valeur 0, nous écrivons $\neg v$. L'état 1 est représenté par $\neg x_1 \wedge x_2$. L'état 0 a comme représentation binaire 00, donc nous écrivons $\neg y_1 \wedge \neg y_2$ pour le représenter. La transition est alors représentée par la forme propositionnelle :

$$(\neg x_1 \wedge x_2) \wedge (\neg y_1 \wedge \neg y_2).$$

L'ensemble des transitions est représenté par la disjonction de chaque arc. Nous obtenons donc

$$f(x_1, x_2, y_1, y_2) = (\neg x_1 \wedge x_2 \wedge \neg y_1 \wedge \neg y_2) \vee (\neg x_1 \wedge x_2 \wedge y_1 \wedge \neg y_2) \vee (\neg x_1 \wedge \neg x_2 \wedge y_1 \wedge \neg y_2) \vee (x_1 \wedge \neg x_2 \wedge y_1 \wedge \neg y_2) \vee (x_1 \wedge \neg x_2 \wedge y_1 \wedge y_2) \vee (x_1 \wedge x_2 \wedge y_1 \wedge \neg y_2)$$

pour l'exemple de la figure 3.14. L'OBDD représentant cette fonction booléenne avec l'ordre des variables $x_1 < y_1 < x_2 < y_2$ est celui de la figure 3.15.

Nous avons donc une façon simple de représenter les transitions d'un graphe par une fonction booléenne et par le fait même un OBDD.

Numérotation par propriétés atomiques

Dans ce qui précède, nous avons utilisé une numérotation arbitraire des états. Pour représenter un graphe, nous distinguions les états à l'aide de variables artificielles qui étaient utilisées dans l'OBDD. Cependant, dans certaines utilisations, comme en vérification, on peut vouloir conserver plus d'informations sur les états. Comme nous

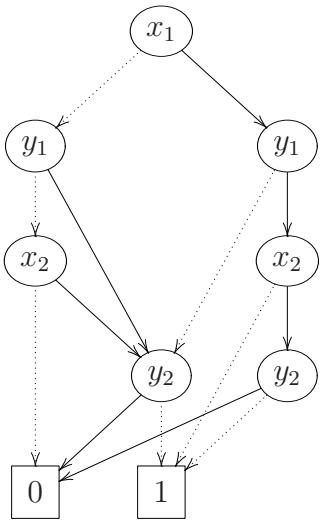


FIG. 3.15 – OBDD représentant le graphe de la figure 3.14.

l'avons vu dans les DTMC et les LMP, il arrive que l'on étiquette les états par des propositions atomiques pour représenter des propriétés de ces états. Nous pouvons alors utiliser ces propriétés pour distinguer les états plutôt que d'utiliser une numérotation arbitraire.

Dans le cas d'une telle numérotation il n'est pas nécessaire de construire un OBDD pour représenter les états, il suffit d'avoir le nombre total d'états. Par contre, lorsque nous utilisons la numérotation par propositions atomiques nous devons avoir un OBDD qui identifie quelles combinaisons des propositions représentent les états.

Nous désirons donc construire un OBDD des états d'un modèle, où les états sont identifiés par des propositions atomiques. Il sera alors facile de pouvoir déterminer s'il existe un état qui satisfait les propriétés p_1, p_2 et p_3 par exemple, en vérifiant que l'évaluation de l'OBDD donne 1. Ensuite nous pourrons construire un OBDD qui représente les transitions du modèle, toujours en nous basant sur les propositions atomiques.

Voyons en détail comment représenter les états à l'aide d'un OBDD et de propositions atomiques. Notons les propriétés atomiques $p_1, ..., p_k$ et nommons chaque état par les propriétés qui y sont vraies. Pour ce faire, chaque état est étiqueté avec p_i lorsque la propriété i est vraie dans cet état et $\neg p_i$ lorsqu'elle y est fausse. Pour un même état, nous faisons la conjonction des propriétés ou de leur négation. Nous pouvons donc représenter l'ensemble des états par des disjonctions de conjonctions. Évidemment, il faut avoir suffisamment de propriétés pour pouvoir distinguer tous les états. La figure 3.16 donne un exemple pour identifier les états à partir des propositions atomiques.

Pour ne pas alourdir le graphe, nous n'avons pas indiqué les propositions qui ne sont pas vraies dans les états ; ainsi, l'absence d'une proposition veut dire que l'état satisfait sa négation. Remarquons que cette façon de faire engendre une numérotation des états. Comme p_1 et p_2 sont vraies dans l'état a et que p_3 y est fausse, a est numéroté (en

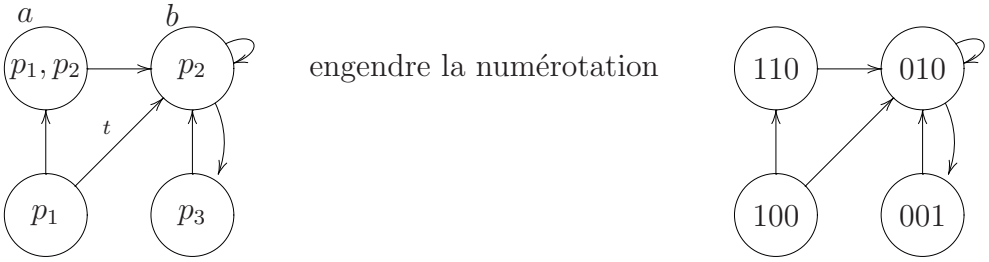


FIG. 3.16 – Numérotation induite par les propositions atomiques p_1, p_2 et p_3.

binaire) par 110 et l'état est représenté par $p_1 \wedge p_2 \wedge \neg p_3$. De même pour l'état b où p_1 et p_3 sont fausses et p_2 est vraie, cet état est numéroté 010 et l'état est représenté par $\neg p_1 \wedge p_2 \wedge \neg p_3$. Pour représenter l'ensemble des états nous faisons une disjonction de tous les états. Les états de ce graphe sont représentés par :

$$f(p_1, p_2, p_3) = (p_1 \wedge p_2 \wedge \neg p_3) \vee (p_1 \wedge \neg p_2 \wedge \neg p_3) \vee (\neg p_1 \wedge p_2 \wedge \neg p_3) \vee (\neg p_1 \wedge \neg p_2 \wedge p_3).$$

Et nous pouvons représenter cette fonction booléenne par un OBDD, comme le montre la figure 3.17. Nous utilisons la même numérotation pour représenter les transitions.

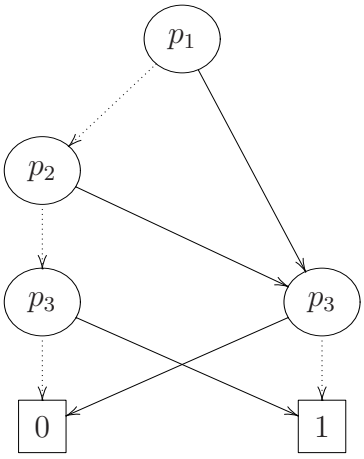


FIG. 3.17 – OBDD représentant les états du graphe de la figure 3.16.

Comme précédemment, il faut être en mesure de distinguer les variables qui représentent les états de départ de celles qui représentent les états d'arrivée. Avec la numérotation engendrée par les propositions atomiques, nous distinguons les variables de départ et d'arrivée en primant les variables qui représentent les états d'arrivée. La transition t est donc représentée par les états étiquetés p_1 et p_2 (dans cet ordre). En gardant la

numérotation induite par les propositions atomiques, $p_1 \wedge \neg p_2 \wedge \neg p_3 \wedge \neg p'_1 \wedge p'_2 \wedge \neg p'_3$ représente la transition t. Pour décrire l'ensemble des transitions, nous faisons une disjonction des conjonctions comme précédemment.

3.5 Implémentation

Nous avons vu que les OBDD peuvent être une structure très efficace pour la manipulation des fonctions booléennes. Cependant, pour qu'ils soient vraiment efficaces, il faut que les opérations de base le soient. Ainsi, beaucoup de recherches ont été faites afin d'optimiser ces opérations. Il y a donc maintenant des concepts fondamentaux qui sont utilisés dans la plupart des implémentations des paquetages des OBDD. Nous présenterons dans cette section quelques-uns de ces concepts.

3.5.1 Taille d'un noeud

Il est clair que chaque noeud d'un OBDD doit avoir au minimum trois champs : l'index de la variable que le noeud représente, un pointeur vers le fils gauche et un pointeur vers le fils droit. En ajoutant certaines informations dans chaque noeud, il est possible d'améliorer considérablement l'efficacité des algorithmes de manipulations des OBDD. Cependant, ajouter trop d'information peut entraîner une sur-utilisation de la mémoire. Il est donc crucial de trouver un bon compromis entre l'amélioration souhaitée pour le temps de calcul et la mémoire utilisée.

3.5.2 OBDD partagé

Dans un logiciel qui utilise des OBDD, l'idée naturelle pour enregistrer les fonctions booléennes consiste à créer un OBDD pour chaque fonction. Cependant, il est possible de garder toutes les fonctions dans un même OBDD nommé OBDD *partagé*. L'idée est simple : dans deux OBDD différents, ayant le même ordre des variables, il est possible qu'il y ait dans chacun d'eux un noeud qui est identifié par la même variable et dont les fils sont aussi identifiés par les mêmes variables. Ainsi, nous pouvons utiliser une seule copie de ce noeud. Un OBDD partagé est identique à un OBDD tel que défini précédemment à l'exception qu'il possède autant de racines qu'il encode de fonctions.

Exemple 3.5.1 *La figure 3.18, empruntée de Meinel et Theobald [41], montre un OBDD qui représente les fonctions* $f_1 = (x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2)$, $f_2 = \neg x_2$ *et* $f_3 = x_1 \wedge \neg x_2$. *Cet OBDD partagé occupe moins d'espace en mémoire que les trois OBDD.*

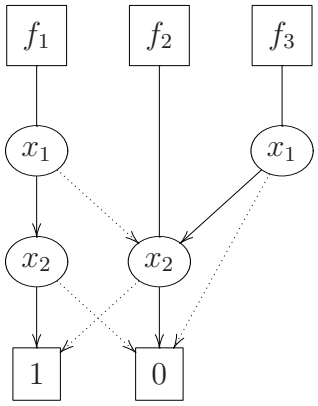


FIG. 3.18 – OBDD partagé des fonctions f_1, f_2 et f_3.

L'espace utilisé par un OBDD partagé n'est jamais supérieur à celui utilisé par les OBDD représentés séparément (s'ils utilisent le même ordre des variables et si la taille des pointeurs utilisés est la même), au pire il n'y aura pas d'isomorphisme entre les sous-graphes. Il est cependant possible que le stockage de plusieurs fonctions booléennes avec plusieurs OBDD, dont l'ordre des variables est *différent*, soit de moindre taille qu'un OBDD partagé qui les représente. Cependant, lorsque les OBDD sont enregistrés avec des ordres de variables différents, les algorithmes de manipulation doivent être modifiés et sont parfois beaucoup moins efficaces.

Par le théorème de canonicité 3.3.2, lorsque deux fonctions équivalentes sont enregistrées, elles pointent sur le même noeud dans l'OBDD partagé. Pour vérifier si deux fonctions sont équivalentes, plutôt que de devoir vérifier si leurs représentations par OBDD sont identiques noeuds à noeuds, il est possible de faire la vérification en comparant seulement les deux pointeurs. Cette caractéristique est nommée la canonicité forte. Ainsi, en plus de sauver de la mémoire, cette technique permet d'accélérer certaines manipulations.

3.5.3 Table unique

Afin de garder la canonicité forte tout au long d'une application, il est important de garder l'OBDD réduit. Ainsi, lorsqu'un nouveau noeud doit être créé, il faut d'abord vérifier s'il n'existe pas déjà un noeud avec la même variable et les mêmes enfants. Si

tel est le cas, le nouveau noeud ne sera pas créé puisqu'on utilisera celui qui existe déjà. Les noeuds déjà existants dans l'OBDD partagé sont enregistrés dans ce que nous appelons la table unique. Pour vérifier efficacement si un noeud est déjà présent dans la table unique, celle-ci est très souvent implémentée à l'aide d'une table de hachage.

Dans la section 3.2.1, nous avons donné un algorithme de réduction d'un OBDD et cet algorithme utilise le BDD non réduit. La table unique permet de construire directement l'OBDD. Les OBDD des variables sont d'abord créés (le OBDD de la variable x_i est constitué d'un noeuds interne étiqueté x_i, dont le fils de gauche est la feuille étiquetée 0 et le fils de droite est la feuille étiquetée 1) et ensuite on manipule ces OBDD avec l'algorithme *apply*. Si les variables ou les autres noeuds à créer existent déjà dans la table unique alors on utilise ceux-ci. Comme chaque noeud apparaît une seule fois dans la table, l'OBDD obtenu est déjà réduit.

3.5.4 Mémoire cache

Lors de l'utilisation des OBDD, il arrive souvent que plusieurs OBDD soient calculés. Pour éviter de recalculer souvent les mêmes OBDD intermédiaires, une fois qu'une opération est effectuée, le résultat est conservé dans une mémoire, nommée *cache*, et nous pouvons ainsi le réutiliser. Plus la mémoire cache est petite, plus il est rapide de vérifier si un calcul a déjà été fait. Cependant, plus elle est petite, moins il est possible de garder des calculs, et les mêmes calculs sont refaits plus souvent. Ainsi, il est important de bien choisir la taille de cette mémoire. Il n'y a pas de taille parfaite pour toutes les applications, celle-ci peut varier beaucoup en fonction du domaine d'application. Par exemple, dans le domaine de la vérification de logiciels, il y a plus de calculs répétitifs que dans le domaine de la vérification de circuits électriques. Il est donc naturel de prévoir une plus grande mémoire cache dans le cas de la vérification des logiciels.

3.5.5 Ramasse-miettes

Pour éviter d'allouer et de libérer de la mémoire constamment, la mémoire est gérée par un ramasse-miettes. Dans chaque noeud on garde comme information le nombre de prédécesseurs du noeud et on ne détruit pas celui-ci aussitôt que le compteur est à zéro. En fait, s'il manque de mémoire pour créer de nouveaux noeuds, le ramasse-miettes est activé et il détruit les noeuds qui n'ont pas de prédécesseur. Parfois le ramasse-miettes est aussi activé lorsqu'un grand nombre de noeuds n'a pas de prédécesseur. Cette façon de faire accélère la création des noeuds. Par contre, lorsque le ramasse-miettes doit être

activé, beaucoup de temps de calcul est perdu à son profit. En général, les expériences montrent qu'il est tout de même préférable d'utiliser un ramasse-miettes.

3.5.6 Colorado University Decision Diagrams

Un paquetage très populaire pour l'utilisation des OBDD est le *Colorado University Decision Diagrams* (CUDD) [54]. Ce paquetage a été développé et rendu public en 1996 par F. Somenzi et son groupe de travail. Une des principales caractéristiques de ce paquetage est qu'il inclut plusieurs algorithmes d'ordonnancement des variables. Ainsi, selon l'utilisation que nous en faisons, nous pouvons choisir l'algorithme qui nous convient. De plus, CUDD donne la possibilité de travailler avec les OBDD, les MTBDD et les ZBDD. Ce paquetage a été développé pour Linux et Unix et il est écrit en C et C++.

CUDD est le paquetage que nous avons utilisé dans notre implémentation. Dans le dernier chapitre nous expliquerons notre utilisation de ce paquetage et les bénéfices et problèmes qu'il nous a emmenés.

Chapitre 4

Variantes des OBDD

Dans ce chapitre nous étudierons quelques variantes des OBDD. Une méthode pour réduire la taille des OBDD ou pour améliorer l'efficacité des algorithmes consiste à transformer la structure de l'OBDD et de l'adapter au type précis des problèmes à traiter. Par exemple, en vérification de circuits électriques, il arrive souvent qu'un circuit n'ait pas de représentation en OBDD de petite taille. Ces circuits ont été une source de motivation pour développer des variantes des OBDD. Cette façon de faire peut parfois donner de très bons résultats pour certaines classes de fonctions booléennes ; par contre, cela ne nous donne pas de méthode générale.

Il y a plusieurs extensions aux OBDD que nous pouvons considérer, nous en présenterons ici quelques-unes parmi les plus populaires. Pour chacune d'elles, nous expliquerons brièvement la structure et donnerons quelques caractéristiques. Nous n'entrerons pas dans les détails d'utilisation et de manipulation de ces structures puisque ceci sort du cadre de ce travail, à l'exception du cas des MTBDD qui sont utilisés dans notre implémentation.

4.1 Diagrammes de décision pour matrices binaires

Dans cette section nous présenterons des variantes qui sont utilisées pour représenter des matrices strictement booléennes.

4.1.1 Diagrammes de décision sans zéro

Les diagrammes de décision sans zéro (ZBDD, de l'anglais *Zero-suppressed Binary Decision Diagrams*) sont des diagrammes de décision binaire dont la réduction est différente de celle utilisée pour obtenir un OBDD. Pour réduire un diagramme de décision binaire dans le but d'obtenir un ZBDD, les noeuds dont l'arc plein pointe directement sur le noeud terminal 0 sont enlevés, comme le montre l'exemple suivant.

Exemple 4.1.1 *La figure 4.1 est un exemple d'élimination d'un noeud dont l'arc plein pointe sur* 0. *En a) nous avons le graphe à réduire, en b) le graphe en partie réduit et en c) le graphe totalement réduit.*

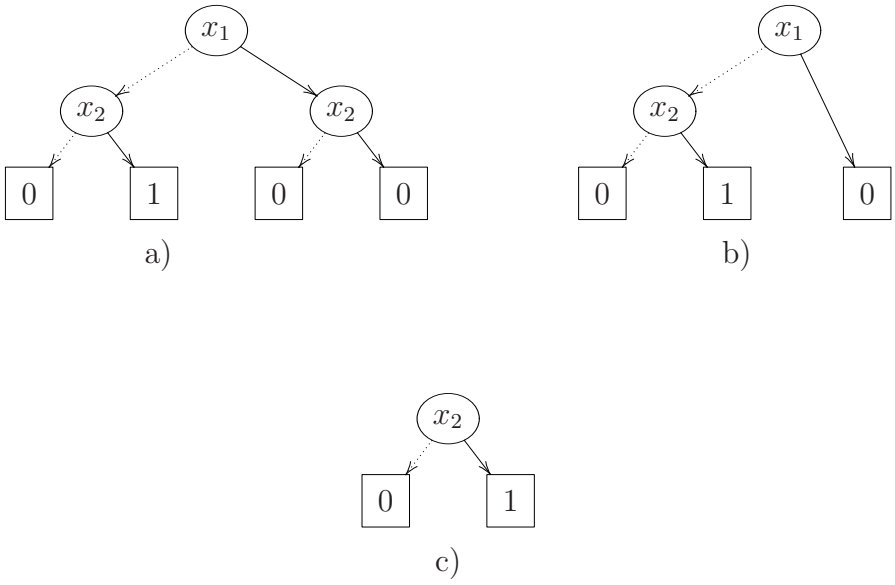


FIG. 4.1 – Réduction d'un ZBDD représentant la fonction $f = \neg x_1 \wedge x_2$.

Comme dans les OBDD, nous gardons une seule copie de chaque feuille. Cependant, nous ne supprimons pas les noeuds qui ont leurs deux arcs sortants pointant sur le même noeud (voir la figure 4.2), sauf si ce noeud est étiqueté 0. La lecture d'un ZBDD n'est pas la même que celle d'un OBDD. Dans ces derniers, lorsqu'un noeud est absent, nous considérons que ses deux arcs sortants ont la même destination. Dans les ZBDD, quand un noeud est absent, nous considérons que son arc plein se dirige vers la feuille 0. C'est pourquoi les noeuds dont les arcs sortants qui pointent vers un même noeud doivent être conservés. Sinon, l'absence d'un noeud dans le ZBDD ne pourrait pas être

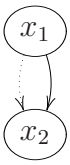


FIG. 4.2 – Il est possible de voir ce sous-graphe dans un ZBDD.

interprétée. Si le nombre et l'ordre des variables sont fixes, cette représentation est canonique.

Pour certaines applications, par exemple celles qui utilisent les ensembles de combinaison, cette représentation est plus pratique et concise que les OBDD. Un ensemble de combinaison est un ensemble de n-tuples de bits. Par exemple, $\{001, 010\}$ est un ensemble de combinaison. L'OBDD et le ZBDD de la figure 4.3 représentent justement cet exemple, tiré de Minato [42]. Lorsque nous devons enregistrer plusieurs ensembles de combinaison, l'amélioration due à l'usage des ZBDD est encore plus grande. Comparons un ZBDD et un OBDD représentant cet ensemble de combinaison.

Exemple 4.1.2 *La fonction à représenter est celle-ci :*

$$f = (\neg x_1 \wedge \neg x_2 \wedge x_3) \vee (\neg x_1 \wedge x_2 \wedge \neg x_3).$$

L'ordre de variable choisi est $x_1 < x_2 < x_3$. La figure 4.3 montre la représentation de f en a) par un OBDD et en b) par un ZBDD.

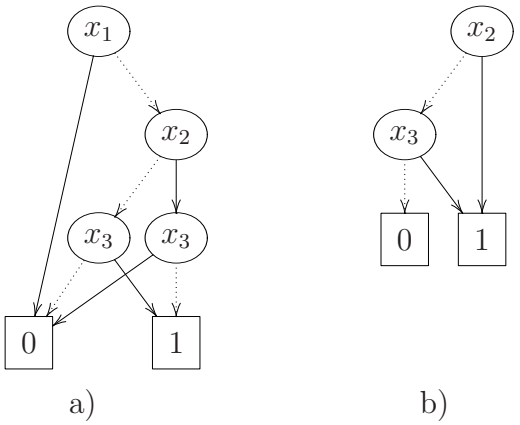


FIG. 4.3 – La représentation de f par un OBDD et un par ZBDD.

Remarquons que les ZBDD peuvent être plus compacts que les OBDD. Cependant, les exemples donnés dans la littérature portent essentiellement sur les ensembles de com-

binaison, ce qui laisse croire que les OBDD seraient plus efficaces dans des applications générales. Pour plus de détails sur les ZBDD, consulter Minato [43].

4.1.2 Diagrammes de décision binaire ordonnés et partitionnés

Nous présentons maintenant les diagrammes de décision binaire ordonnés et partitionnés (POBDD, de l'anglais *Partitioned Ordered Binary Decision Diagrams*). Cette structure utilise en fait plusieurs OBDD. Le but est de représenter une formule booléenne avec plusieurs OBDD qui pourront avoir un ordre des variables différent. Pour ce faire, le domaine de la fonction à représenter est décomposé en sous-ensembles pas nécessairement disjoints. Chaque OBDD utilisé représentera une fonction g_i égale à la fonction f sur le ième sous-ensemble du domaine et nulle ailleurs. Ainsi, lors de l'évaluation du POBDD, si au moins un des OBDD donne la valeur 1 comme résultat à l'entrée donnée, alors la valeur de la fonction f avec cette même entrée est 1. Un avantage d'une telle représentation est que chaque OBDD peut avoir un ordre des variables différent. De plus, il se peut qu'il n'existe pas de bon ordre des variables pour une fonction donnée mais qu'il en existe un (le même pour tous les g_i) qui donne de bons résultats pour le POBDD. Si les sous-ensembles sont fixés, ainsi que l'ordre des variables, alors cette représentation est canonique. Voyons un exemple simple.

Exemple 4.1.3 (POBDD) *Soit la fonction*

$$f = (\neg x_1 \wedge \neg x_2 \wedge x_3) \vee (x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2 \wedge x_3).$$

Pour illustrer la décomposition du domaine, voyons celle-ci sur la table de vérité de f à la figure 4.4.

sous-ensembles	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
A	0	0	0	0
	0	0	1	1
	0	1	0	0
B	0	1	1	0
	1	0	0	1
C	1	0	1	0
	1	1	0	0
	1	1	1	1

FIG. 4.4 – La décomposition du domaine de f.

Définissons $g_A = (\neg x_1 \wedge \neg x_2 \wedge x_3)$ qui est égale à f sur A, $g_B = (x_1 \wedge \neg x_2 \wedge \neg x_3)$ qui est égale à f sur B et $g_C = (x_1 \wedge x_2 \wedge x_3)$ qui est égale à f sur C. Ainsi, la représentation de f en POBDD est l'union des trois OBDD qui représentent g_1, g_2 et g_3. Lorsque nous voulons évaluer f, nous évaluons chacun des OBDD des fonctions g_A, g_B et g_C et nous faisons la disjonction des résultats.

Un autre avantage potentiel de cette façon de représenter une formule booléenne est que l'ordre optimal pour chaque OBDD pourrait être déterminé par un algorithme exact. Cet algorithme a un temps de calcul exponentiel par rapport au nombre de variables mais si le domaine est suffisamment décomposé, le temps consommé pourrait être très acceptable. Ainsi, chaque OBDD serait de taille minimale. Il est important de remarquer que même si chaque OBDD est de taille minimale, la structure n'est pas nécessairement de taille minimale. Par exemple, il pourrait exister une autre décomposition du domaine qui engendrerait une diminution de la taille totale. Les POBDD ont aussi le désavantage qu'en plus du problème de l'ordre des variables s'ajoute le problème de décomposition du domaine. Ce problème semble difficile à résoudre, comme l'était l'ordre des décompositions dans les FDD. Cependant, dans certaines utilisations comme l'apprentissage par renforcement, il arrive que la décomposition du domaine soit naturellement décrite par la problématique. Dans ce cas, le problème de trouver une décomposition est évité et les POBDD peuvent donner de bons résultats. Pour plus de détails sur les POBDD, un lecteur intéressé est invité à lire Narayan et al. [44].

4.2 Diagrammes de décision pour matrices non-binaires

Dans cette section nous présenterons des variantes qui sont utilisées pour représenter des matrices dont les entrées peuvent être quelconques.

4.2.1 Diagrammes de moments binaires

Étudions maintenant les diagrammes de moments binaires (BMD, de l'anglais *Binary Moment Diagrams*). Les BMD peuvent représenter les matrices qui ne sont pas composées seulement de 0 et de 1. Ils sont plus précisément utilisés pour représenter le polynôme dont l'image est l'ensemble des valeurs de la matrice : un polynôme p

représentant une matrice M est tel que $p(i,j) = M[i][j]$. La figure 4.5 donne un exemple d'un tel polynôme. Dans cet exemple, nous remarquons qu'en évaluant le polynôme en

$$\begin{array}{c} \\ x=0 \\ x=1 \end{array} \begin{array}{c} \begin{array}{cc} y=0 & y=1 \end{array} \\ \begin{pmatrix} 2 & 3 \\ 0 & 1 \end{pmatrix} \end{array} \quad \text{est représenté par} \quad 2+y-2x$$

FIG. 4.5 – Une matrice et son polynôme associé.

$x = 0$ et $y = 1$ nous obtenons la valeur 3, et pour chacune des autres valuations, nous obtenons la valeur correspondante de la matrice.

Voyons comment déterminer un polynôme à variables binaires dont l'évaluation donne la valeur voulue dans une matrice donnée. Comme dans le chapitre 3.4, nous utilisons des variables booléennes pour numéroter les lignes et les colonnes de la matrice. Les lignes de la matrice sont numérotées par les variables x_i et les colonnes par y_i. Les variables du polynôme sont exactement celles-ci. Pour trouver le polynôme associé à une matrice, il faut déterminer les termes du polynôme. Chaque terme (du polynôme non simplifié) représente une entrée de la matrice. Si l'élément de la matrice, lorsque les variables ont une certaine valeur, est a, alors le terme du polynôme pour cette entrée est a multiplié par : $(1 - v)$ lorsque la variable v vaut 0 et v lorsqu'elle vaut 1, et ce pour toutes les variables. En additionnant tous les termes ainsi obtenus, nous obtenons le polynôme désiré.

Construisons le polynôme de l'exemple précédent. Nous commençons par le 2 situé en $x = 0$ et $y = 0$, puisque les deux variables sont égales à 0, nous avons les facteurs $1 - x$, $1 - y$ et le facteur 2 représentant la valeur dans la matrice. Nous multiplions ces facteurs pour obtenir le terme $2(1 - x)(1 - y)$. Nous faisons de même pour les autres entrées de la matrice et nous additionnons le tout pour obtenir

$$2(1-x)(1-y) + 3(1-x)y + 0x(1-y) + 1xy.$$

En simplifiant, nous obtenons le polynôme de la figure 4.5. Remarquons qu'ici la multiplication joue le même rôle que le *ET* logique utilisé dans les formes propositionnelles. En effet, on peut lire $x(1-y)$ comme ceci : la variable x prend la valeur 1 *ET* la variable y prend la valeur 0.

Pour représenter le polynôme, nous utilisons les BMD. Cette structure est un diagramme de décision binaire, qui a comme feuilles les coefficients du polynôme déterminé précédemment. La figure 4.6 montre le BMD associé à la matrice de la figure 4.5. Il est important de remarquer qu'avec les BMD, les règles de réduction des OBDD ne sont pas utilisées. Ainsi, dans l'exemple de la figure 4.6, lorsque nous voyons que la variable

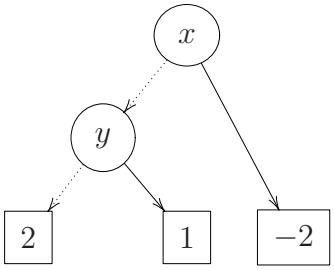


FIG. 4.6 – Le BMD associé au polynôme $2 + y - 2x$.

y est absente quand $x = 1$ cela ne veut pas dire que peu importe la valeur de y le coefficient -2 est présent, mais plutôt que y n'apparaît pas dans le terme qui a comme coefficient -2 dans le polynôme.

Le polynôme qui représente une matrice donnée est unique. Ainsi, le BMD qui le représente l'est aussi. De cette façon, si deux BMD sont isomorphes, nous savons que les matrices qu'ils représentent sont les mêmes. En ce qui concerne la taille des BMD, le polynôme ne peut pas être beaucoup simplifié en général, et ainsi il a presque autant de termes que la matrice a d'entrées. Les BMD ne sont donc pas toujours efficaces. Constatons tout de même que l'idée est jolie ! De plus, des améliorations ont été apportées afin de donner une structure, nommée *BMD, qui a une taille moindre qu'un BMD. Sans entrer dans les détails, mentionnons que les techniques utilisées pour transformer un BMD en *BMD sont semblables à celles utilisées dans les EVBDD, la prochaine variante de diagramme binaire présentée. Pour plus de détails sur les BMD et les *BMD consulter Bryant [13].

4.2.2 Diagrammes de décision binaire à arcs valués

Les diagrammes de décision binaire à arcs valués (EVBDD, de l'anglais *Edge Valued Binary Decision Diagrams*) sont des diagrammes de décision binaire qui ont une seule feuille (étiquetée 0) et dont les arcs sont munis d'une valeur. Ils sont utilisés pour représenter les fonctions à valeurs entières et à variables binaires, c'est-à-dire $f : \{0,1\}^n \to \mathbf{Z}$. Pour conserver l'unicité de la représentation, on exige que la valeur sur les arcs pointillés soit nulle. Les EVBDD sont très semblables aux OBDD ; un ordre de variable est imposé pour tous les chemins et les noeuds représentent des variables booléennes. Cependant, la sémantique d'un noeud est différente de celle des OBDD. Dans ces derniers, le noeud s étiqueté par la variable x signifie $(\neg x \wedge f[x := 0]) \vee (x \wedge f[x := 1])$. Pour un EVBDD, il faut parcourir les arêtes selon la procédure habituelle, selon la valeur des variables, tout en additionnant les valeurs rencontrées sur

les arêtes.

Nous allons présenter deux exemples d'EVBDD. Le premier est très simple et servira à expliquer comment lire l'EVBDD. Le deuxième nous montrera un exemple d'utilisation d'un EVBDD pour représenter une fonction booléenne.

Exemple 4.2.1 *La figure 4.7 est l'EVBDD de la fonction* $g(x, y, z, w) = 3x + 2y - 9z + w + 2$. *Pour connaître la valeur de la fonction* g *en* $x = 1,\ y = 1,\ z = 0$ *et* $w = 0$ *à partir de l'EVBDD, nous additionnons les valeurs rencontrées sur les arcs. Dans cet exemple nous obtenons :* $g(1, 1, 0, 0) = 2 + 3 + 2 + 0 + 0 = 7$. *Remarquons que l'arc entrant sur*

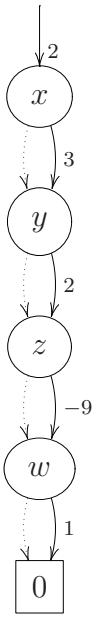


FIG. 4.7 – L'EVBDD associé à la fonction g.

le noeud x *représente la valeur de la constante dans la fonction. Il est possible que cet arc ait la valeur* 0 *si le polynôme n'a pas de terme constant.*

Exemple 4.2.2 *Regardons maintenant la figure 4.8. Dans cet exemple un peu plus complexe, l'EVBDD est utilisé pour représenter le polynôme*

$$f(x, y, z) = xy + yz + zx – 2xyz,$$

avec x, y, z *des variables booléennes. Ce polynôme représente la formule booléenne* $(x \wedge y) \vee (z \wedge x) \vee (y \wedge z)$ *et est obtenu par le même procédé que nous avons présenté dans la section sur les BMD.*

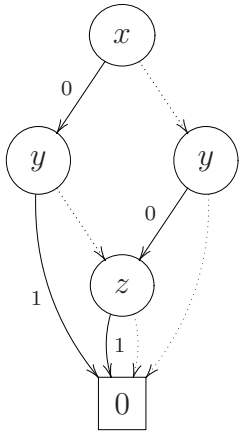


FIG. 4.8 – L'EVBDD associé à la fonction f.

Avec les EVBDD, il est aussi possible de représenter des fonctions dont les variables sont à valeurs entières autres que 0 et 1 si ces valeurs sont bornées. Pour ce faire, ces variables sont remplacées par des variables booléennes. Nous utilisons la notation binaire pour substituer chaque variable à valeur entière par plusieurs variables binaires. Par exemple, si $x \in \{0, 1, 2, 3, 4, 5, 6, 7\}$ alors x est remplacé par $x_0 + 2x_1 + 4x_2$. Cette combinaison de trois nouvelles variables booléennes permet de représenter les 2^3 valeurs que prend x. Ainsi, toutes les fonctions à variables entières peuvent être représentées par un EVBDD. L'exemple suivant illustre cette méthode.

Exemple 4.2.3 *Si nous souhaitons représenter la fonction* $g(x, y, z) = 3x + 4y - 5xz$ *avec* $x \in \{0, 1, ..., 7\}$ *et* $y, z \in \{0, 1\}$*, nous écrirons*

$$g(x_0, x_1, x_2, y, z) = 3(x_0 + 2x_1 + 4x_2) + 4y - 5(x_0 + 2x_1 + 4x_2)z.$$

Le lecteur intéressé trouvera plus d'information sur les EVBDD dans Lai et Sastry [35].

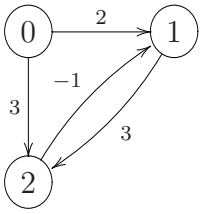
4.2.3 Diagrammes de décision binaire à terminaux multiples (MTBDD)

Les diagrammes de décision binaire à terminaux multiples (MTBDD, de l'anglais *Multi Terminal Binary Decision Diagrams*), parfois nommés diagrammes de décision algébrique (ADD, de l'anglais *Algebraic Decision Diagrams*) sont une généralisation des OBDD dans laquelle des valeurs différentes de 0 ou 1 sont permises pour les feuilles. Le MTBDD est construit de la même façon qu'un OBDD à l'exception que d'autres feuilles

que 0 et 1 sont admises. Alors que les OBDD représentent les fonctions booléennes, les MTBDD peuvent représenter des fonctions à valeurs quelconques.

Nous pouvons ainsi utiliser les MTBDD pour représenter des graphes dont les arcs sont étiquetés par un poids. Dans ce cas, les entrées de la matrice de transitions contiennent le poids de la transition représentée plutôt que la valeur 1 comme précédemment avec les graphes non-valués. Ce poids peut représenter par exemple la distance entre des villes en kilomètres, ou la probabilité d'une transition comme dans les DTMC.

La figure 4.9 est un exemple de graphe dont les arcs sont valués. De la matrice,



$$\begin{pmatrix} 0 & 2 & 3 \\ 0 & 0 & 3 \\ 0 & -1 & 0 \end{pmatrix}$$

FIG. 4.9 – Un graphe valué et sa matrice de transitions.

nous pouvons déduire la forme propositionnelle qui représente les transitions. Voici la fonction booléenne associée à la matrice de l'exemple 4.9 avec la numérotation suggérée par la figure (selon la même méthode qu'avec les OBDD)

$$\begin{aligned} f(x_0, x_1, y_0, y_1) \quad = \quad & (\neg x_0 \wedge \neg x_1 \wedge \neg y_0 \wedge y_1) \vee (\neg x_0 \wedge \neg x_1 \wedge y_0 \wedge \neg y_1) \vee \\ & (\neg x_0 \wedge x_1 \wedge y_0 \wedge \neg y_1) \vee (x_0 \wedge \neg x_1 \wedge \neg y_0 \wedge y_1). \end{aligned}$$

Évidemment, sous la forme de fonction booléenne, l'information du poids de chaque arc est perdue. Mais cette information se trouve dans le MTBDD, puisque chaque feuille représente le poids de l'arc dans le graphe. Ce n'est donc pas la formule f que nous utilisons pour construire le MTBDD. Pour garder les poids des arcs, nous devons trouver le polynôme dont l'image est l'ensemble des valeurs de la matrice. Ce polynôme est le même que nous avons utilisé dans les BMD : sa construction, ainsi que sa signification sont les mêmes. Voici le polynôme associé à la matrice de la figure 4.9

$$2y_1 - 5y_1y_0 - 2y_1x_1 + 2y_1x_1y_0 - 3y_1x_0 + 6y_1x_0y_0 + 3y_1x_0x_1 - 3y_1x_0x_1y_0 + 3y_0 - 3x_0y_0.$$

Contrairement aux BMD, les MTBDD n'ont pas comme feuilles les coefficients du polynôme. Les feuilles sont exactement les valeurs des éléments de la matrice. Cependant, le polynôme peut être utilisé pour construire le MTBDD.

Il existe des algorithmes de manipulations des MTBDD semblables à ceux utilisés sur les OBDD, par contre ces algorithmes n'exécutent pas des opérations booléennes. Ils

implémentent plutôt des opérations arithmétiques comme l'addition et la multiplication. Ainsi, à partir du polynôme, on multiplie et additionne des MTBDD pour obtenir celui qui représente la matrice. La figure 4.10 est le MTBDD qui représente la matrice de transitions de la figure 4.9 avec l'ordre des variables $x_0 < x_1 < y_0 < y_1$. Remarquons que

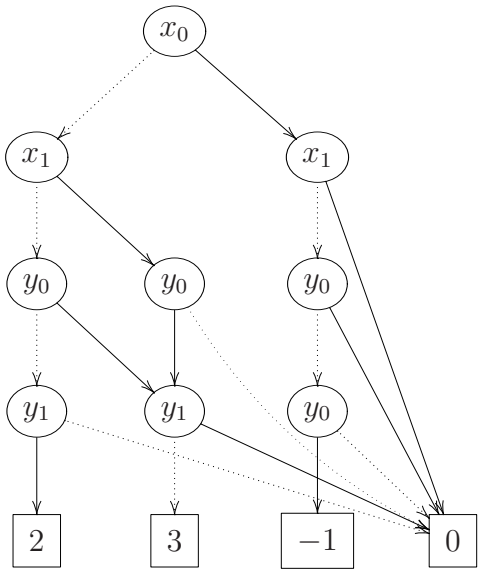


FIG. 4.10 – Le MTBDD associé au graphe de la figure 4.9.

les MTBDD ont les mêmes caractéristiques que les OBDD, à l'exception de la valeur des feuilles. Les règles de réduction sont les mêmes et la taille des MTBDD est aussi affectée par l'ordre des variables ainsi que par la numérotation des états. À l'instar des OBDD, les MTBDD sont efficaces quand la matrice est creuse, c'est-à-dire lorsqu'il y a beaucoup plus de zéro que d'autres valeurs. Cependant, les MTBDD peuvent aussi être très efficaces même s'il y a peu de zéro, par exemple lorsque beaucoup d'entrées de la matrice ont la même valeur. De cette façon, il y a beaucoup de réductions dues à la redondance des valeurs. En apprentissage par renforcement, les MTBDD sont parfois utilisés de la même façon que le sont les OBDD dans un POBDD. Ainsi le domaine est décomposé et un MTBDD est créé pour chaque partition.

Contrairement à la plupart des variantes que nous présentons dans ce travail, les MTBDD sont très populaires et beaucoup utilisés. Pour plus de détail sur les MTBDD ou sur les applications de ceux-ci, vous pouvez voir Bahar et al. [5], Baier et al. [7], Fujita et al. [39], Fujita et al. [20], Hermanns et al. [28, 29], Kropf et Ruf [33] et Kwiatkowska et al. [34].

Les MTBDD sont très importants pour la suite de ce travail puisque nous les avons utilisés pour enregistrer les LMP dans le vérificateur de modèles CISMO.

4.2.4 Diagrammes avec noeuds de décision

Les diagrammes avec noeuds de décision (DNBDD, de l'anglais *Decision Nodes Binary Decision Diagrams*) sont une autre façon de stocker les matrices à entrées non booléennes. Un des principaux problèmes des MTBDD est que si les valeurs non-nulles de la matrices sont toutes distinctes alors il n'y a pas beaucoup de réduction par isomorphisme dans la structure. Une façon de contrer ce problème est de construire un OBDD qui détermine quelles entrées ne sont pas nulles. Cet OBDD profite des isomorphismes puisque les feuilles ne sont que 0 ou 1, donc il y a beaucoup de redondance. Ensuite, il faut déterminer la valeur exacte de l'entrée dans la matrice. Avant d'expliquer comment trouver la valeur exacte, voyons une définition qui nous sera utile.

Définition 4.2.4 (Chemin) *Un chemin dans un OBDD O est une chaîne de noeuds qui débute à la racine et se termine à une feuille. Il est noté $(x_0 = b_0, ..., x_m = b_m, b_f)$ avec x_0 la variable de la racine de O, $b_i \in \{0, 1\}$, $x_0 < ... < x_m$ respecte l'ordre des variables du OBDD et b_f est la valeur de la feuille. Les b_i indiquent quel arc a été emprunté à la sortie de chaque noeud étiqueté x_i.*

Un chemin où $b_f = 1$ est appelé un vrai-chemin. La longueur d'un chemin est égale au nombre de variables présentes dans celui-ci.

Un seul chemin dans l'OBDD est associé à chaque valuation. Cependant, plus d'une valuation peut être associée à un même chemin. Pour un chemin c de longueur k il y aura 2^{n-k} valuations associées au chemin c, où n est le nombre de variables. Remarquons que $n - k$ est en fait le nombre de variables non-présentes dans le chemin.

Exemple 4.2.5 *La figure 4.11 montre un OBDD à 4 variables. Regardons dans cet OBDD le chemin $c := (x_0 = 0, x_1 = 0, 0)$. Dans le chemin c, les variables x_2 et x_3 sont absentes. Pour toute valuation v, si v fixe $x_0 = 0$ et $x_1 = 0$ et donne des valeurs quelconques à x_2 et x_3, la valuation v sera associée au chemin c. Combien y a-t-il de valuations qui ont c comme chemin ? Il y en a 4 puisque les deux variables manquantes peuvent prendre chacune deux valeurs, donc 2^2 valuations différentes. Avec ce que avons dit précédemment, la longueur du chemin c est 2 et il y a 4 variables, donc $n - k = 4 - 2 = 2$ valuations associées à ce chemin.*

Afin de conserver les valeurs de la matrice, il faut associer une liste de valeurs à chaque *vrai-chemin*. La longueur de cette liste sera exactement le nombre de valuations associées à ce chemin. Ainsi, à l'aide de l'OBDD, il est possible de savoir si la valeur est nulle

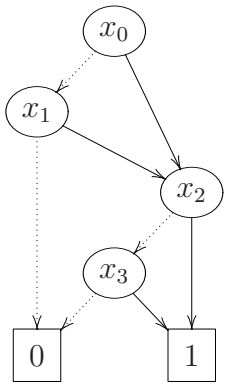


FIG. 4.11 – OBDD de $f(x_0, x_1, x_2, x_3)$ avec $x_0 < x_1 < x_2 < x_3$.

ou non ; si elle n'est pas nulle, par la liste associée au chemin, la valeur exacte de l'entrée voulue dans la matrice est déterminée. Le problème est alors de trouver comment associer une liste aux chemins. C'est ici qu'interviennent les noeuds de décision.

Définition 4.2.6 (Noeud de décision) *Un noeud de décision est un noeud de l'OBDD qui n'a pas la feuille étiquetée* 0 *comme enfant.*

Une autre façon de décrire un noeud de décision serait de dire qu'il est possible, à partir de ce noeud, d'atteindre la feuille étiquetée 1 en passant par l'un ou l'autre de ses fils. Pour chaque *vrai-chemin*, il faut déterminer le dernier noeud de décision, c'est-à-dire le noeud de décision le plus près des feuilles. Si s_i est le dernier noeud de décision sur le chemin et s_{i+1} le noeud suivant s_i dans le même chemin, alors c'est sur l'arc (s_i, s_{i+1}) que nous fixons la liste de valeurs. La figure 4.12 montre un premier exemple d'un DNBDD qui représente une matrice.

$$\begin{pmatrix} 0 & 0 & 1 & 2 \\ 0 & 0 & 3 & 4 \\ 0 & 0 & 5 & 6 \\ 7 & 8 & 0 & 0 \end{pmatrix}$$

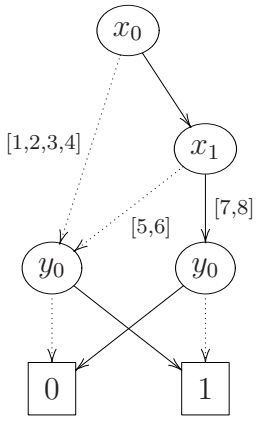


FIG. 4.12 – Une matrice et sa représentation en DNBDD.

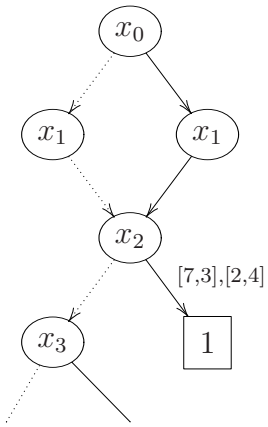


FIG. 4.13 – Deux chemins ayant le même noeud de décision.

Exemple 4.2.7 (Le DNBDD représentant une matrice) *Dans le DNBDD de la figure 4.12, considérons le chemin* $c := (x_0 = 0, y_0 = 1, 1)$ *. Nous remarquons qu'il y a* 2^2 *valuations associées à ce chemin et celles-ci représentent les entrées* $1, 2, 3$ *et* 4 *dans la matrice* A*. Soit* $v =< x_0 = 0, x_1 = 1, y_0 = 1, y_1 = 0 >$ *une de ces* 4 *valuations. Puisque* c *est le chemin associé à* v *et qu'il est un vrai-chemin (le dernier noeud est étiqueté* 1*), nous savons que la valeur de l'entrée dans la matrice est non-nulle. La valeur exacte se trouve dans la liste attachée à l'arc* (x_0, y_0)*. Il est facile de trouver cette valeur, la valuation* $x_1 = 0, y_1 = 0$ *est l'entrée* 0 *de la liste, la valuation* $x_1 = 0, y_1 = 1$ *est l'entrée* 1 *de la liste et ainsi de suite. Dans le cas de* v*, nous avons* $x_1 = 1$ *et* $y_1 = 0$ *donc la bonne valeur est la valeur à la position* 2*, soit* 3*. Ainsi, si le DNBDD de la figure 4.12 représente la fonction* f*, alors* $f(0, 1, 1, 0) = 3$*.*

Malheureusement, il arrive que deux chemins distincts aient le même dernier noeud de décision et dans ce cas, il y a plus d'une liste de valeurs attachées à un même arc. Nous devons donc savoir quelle liste va avec quel chemin. Une façon facile de le faire est d'utiliser l'ordre lexicographique des chemins. Par exemple, le chemin $(x_0 = 0, x_1 = 0, x_2 = 1)$ de la figure 4.13 précède le chemin $(x_0 = 1, x_1 = 1, x_2 = 1)$ car $001 < 111$. Ainsi, il suffit de conserver les listes de façon ordonnée. Cependant, dans Siegle [51], on suggère d'ajouter une autre structure à celle du DNBDD pour simplifier les algorithmes de manipulation des DNBDD. La structure peut être de plus petite taille que les MTBDD étant donné le gain d'espace par la fusion des sous-graphes isomorphes. Remarquons cependant que s'il y a beaucoup de redondance dans les entrées de la matrice, il y a beaucoup d'information inutilement stockée dans les listes, c'est-à-dire que la même valeur est stockée plusieurs fois. Dans ce cas, l'utilisation des MTBDD est recommandée.

Chapitre 5

Minimisation de la taille de l'OBDD représentant un graphe

Nous avons évoqué précédemment l'importance de trouver un bon ordre des variables pour diminuer la taille des OBDD. Rappelons qu'il est $\mathcal{NP}$-*Difficile* de trouver l'ordre des variables optimal et ainsi, pour résoudre en partie ce problème, nous ne pouvons que trouver des méthodes (heuristiques) qui donnent de bons résultats sans toutefois être optimales. Il y a deux catégories d'heuristiques pour améliorer l'ordre des variables. La première catégorie regroupe les heuristiques dites *dynamiques*. Ces algorithmes sont utilisés dynamiquement, c'est-à-dire pendant les manipulations. L'ordre des variables peut donc changer plusieurs fois tout au long du processus de calcul. Le plus connu et efficace de ces algorithmes est sans doute le *shifting* qui consiste à changer l'ordre de deux variables en les permutant. En appliquant cette technique à plusieurs paires de variables on arrive à choisir un ordre qui diminue la taille de l'OBDD. Il existe des algorithmes qui permettent de permuter deux variables très rapidement et ce, en gardant l'OBDD réduit. Ces techniques donnent de bons résultats mais elles sont toutefois assez coûteuses en temps de calcul si on décide de tester plusieurs permutations, ce qui est nécessaire si on veut réduire de beaucoup la taille de l'OBDD. Les heuristiques dynamiques sont souvent utilisées dans la vérification des circuits électriques.

La deuxième catégorie regroupe les heuristiques *statiques*. Ces algorithmes sont utilisés avant les manipulations pour trouver l'ordre des variables qui sera utilisé tout au long des calculs. Nous avons déjà parlé d'une telle heuristique lorsque nous avons mentionné que l'ordre $x_0 < y_0 < ... < x_n < y_n$ donne en général de bons résultats. Les heuristiques portant sur la numérotation sont aussi une forme d'heuristiques statiques puisqu'elles s'appliquent une seule fois, avant de débuter les calculs.

Notre étude des OBDD nous a amenés à étudier ces heuristiques. Lors de nos lectures, nous avons eu quelques idées d'heuristiques qui pourraient réduire la taille des OBDD ; nous les présenterons plus loin.

Comme les OBDD ont d'abord été utilisés pour la représentation des circuits électriques, beaucoup de techniques ont été développées pour réduire la taille des OBDD dans ce domaine. Nous présenterons la méthode classique utilisée sur les circuits et discuterons ensuite des méthodes qui peuvent être utilisées sur les OBDD qui représentent des graphes.

5.1 Heuristique classique sur les circuits et ses limitations

Une des principales méthodes utilisées sur les OBDD représentant des circuits est la recherche en profondeur sur le circuit. L'algorithme s'exécute ainsi : le parcours débute par la sortie du circuit et se complète par une recherche en profondeur, comme sur un graphe. L'ordre des variables est déterminé par l'ordre de rencontre des entrées. L'exemple suivant montre une application de cette heuristique à un circuit.

Exemple 5.1.1 *Nous souhaitons déterminer un ordre de variable efficace pour l'OBDD qui représente le circuit de la figure 5.1. En appliquant l'heuristique classique, l'ordre*

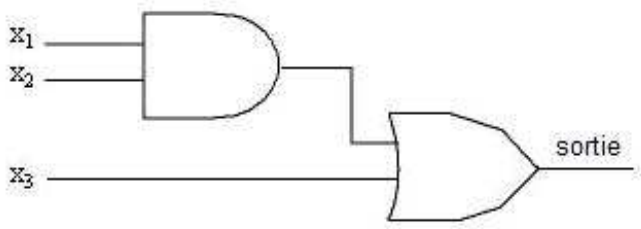


FIG. 5.1 – Exemple de circuit.

trouvé pourrait être $x_1 < x_2 < x_3$ ou $x_3 < x_2 < x_1$ selon l'implémentation de l'algorithme de recherche en profondeur.

Beaucoup de recherches ont été faites pour trouver des heuristiques qui déterminent un ordre des variables efficace pour diminuer la taille d'un OBDD qui représente un circuit

électrique. Par contre, il y a moins de recherche sur des heuristiques statiques pour les OBDD représentant un graphe. Dans ce dernier cas, on utilise souvent des paquetages de gestion des OBDD dans lesquels sont implémentées plusieurs heuristiques, le plus souvent dynamiques. Ces heuristiques ne sont pas toujours bien adaptées aux OBDD qui représentent un graphe ou, lorsqu'elles donnent de bons résultats, elles utilisent beaucoup de temps de calcul. Étonnamment, nous n'avons pas rencontré de remarque à ce sujet dans la littérature. Nous allons présenter d'autres avenues de recherche que l'heuristique classique portant sur les circuits électriques mais voyons d'abord plus en détail pourquoi nous pensons que l'heuristique classique n'est pas efficace dans le contexte des graphes.

Nous ne pensons pas que les techniques utilisées avec les circuits électriques soient adéquates dans le cas des graphes car le type de fonctions booléennes qui représentent un graphe est très particulier. En effet, ces fonctions booléennes sont sous une forme k-normale disjonctive, ce qui donne une fonction de la forme $(x_1 \wedge ... \wedge x_k) \vee (\neg x_1 \wedge ... \wedge x_k) \vee ... \vee (x_1 \wedge ... \wedge \neg x_k)$ (avec $k = 2\lceil \log_2 n \rceil$ et où n est le nombre de sommets du graphe). Le circuit qu'elles engendrent est toujours semblable à celui de la figure 5.2. Pour simplifier le circuit, nous avons regroupé les conjonctions, qui sont en fait binaires,

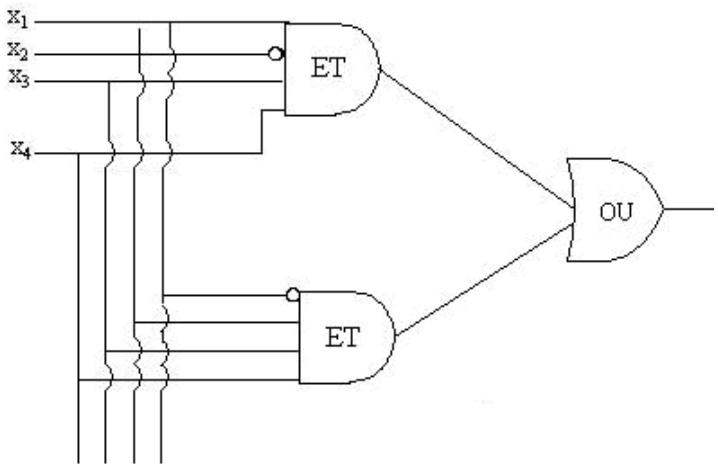


FIG. 5.2 – Circuit représentant les transitions d'un graphe.

en une seule, représentée par un *ET*. Chaque *ET* représente une transition puisqu'il fait la conjonction de chacune des variables ou de leur négation (représentée par un petit cercle à l'entrée du *ET*). De même pour le *OU* qui est en réalité $t - 1$ disjonctions qui prennent chacune deux entrées, où t est le nombre de transitions.

Nous avons remarqué que sur ce type de circuit, chaque entrée est à la même distance (profondeur) de la sortie. Ainsi, le parcours en profondeur du circuit numérotera les

variables d'entrée dans un ordre quasi quelconque selon l'implémentation de la recherche en profondeur. Pour que cette méthode donne de bons résultats, il faudrait que les variables soient déjà dans un bon ordre, ce qui revient au problème de départ. Pour cette raison, nous ne pensons pas que les heuristiques utilisées pour les OBDD représentant des circuits peuvent être efficaces pour les OBDD qui représentent des graphes. Dans le cadre de la vérification de logiciels, ce sont justement des graphes que nous souhaitons représenter.

Remarquons tout de même que pour adapter cette méthode à la représentation des graphes, nous pourrions utiliser des techniques (peut-être les mêmes que celles utilisées dans la recherche sur les circuits électriques) pour transformer la fonction booléenne qui représente les transitions d'un graphe en une fonction booléenne moins symétrique. Nous n'avons pas étudié cette avenue, nous avons plutôt étudié des heuristiques statiques qui reposent sur la structure du graphe.

5.2 Principes de base

On trouve dans Hermanns et al. [29] quelques principes de base qui servent de guide pour améliorer l'utilisation des MTBDD (ils s'appliquent généralement aussi aux OBDD). Sans être infaillibles, ces règles donnent une idée des choses à éviter ou à faire pour que les MTBDD soient de taille moindre. On peut voir ces principes comme étant des heuristiques. Sans entrer dans les détails, nous énonçons ici ces principes, le lecteur intéressé étant invité à consulter le document original.

Voici les principes de base de Hermanns pour utiliser les MTBDD efficacement :

- Il est recommandé d'utiliser l'ordre des variables qui alterne entre les variables de départ et celles d'arrivée.

- Il n'est pas toujours plus efficace d'utiliser $\lceil log_2(m) \rceil$ variables pour représenter m états. En fait, il est souvent préférable d'utiliser la structure d'une spécification de haut niveau.

- L'utilisation de techniques pour compresser l'espace d'états peut nuire à la taille des MTBDD puisque la régularité est perdue.

- Si la spécification a un opérateur de composition, il est préférable de choisir un meilleur encodage pour les niveaux les plus bas, plutôt que de choisir la

numérotation après la composition.

- L'exploitation de la régularité est la clé de la réussite. Si la matrice a des blocs identiques d'entrées, il faut les encoder « près » les uns des autres. Il peut en résulter un gain de mémoire exponentiel.

Certaines remarques faites par la suite sont inspirées de ces principes. Nous allons maintenant discuter de l'importance de la numérotation des états.

5.3 Influence de la numérotation des états

Avant de présenter des heuristiques, explorons ce qui influence la taille de l'OBDD qui représente une matrice de transitions. Nous avons déjà parlé de l'importance de l'ordre des variables. Peu importe le domaine d'utilisation des OBDD, l'ordre des variables affecte beaucoup leur taille. Mais dans un graphe, il y a un autre facteur qui influence beaucoup la taille de l'OBDD : l'encodage des états.

L'encodage est la numérotation des états. Nous avons vu que nous pouvons le faire avec les propositions atomiques ou de façon consécutive, ainsi il faut connaître les effets de ces numérotations. Voyons sur l'exemple de la figure 5.3, comment la façon de numéroter les états du graphe G peut influencer la taille de son OBDD.

FIG. 5.3 – Graphe G et sa matrice de transitions.

Nous pouvons déduire la forme propositionnelle associée à la matrice de G. Nous avons trois états, il faut donc deux variables booléennes pour identifier chaque état. Soient x_1, x_2 les variables des états de départ des arcs et y_1, y_2 celles des états d'arrivée. La forme propositionnelle est donc

$$(\neg x_1 \wedge \neg x_2 \wedge \neg y_1 \wedge y_2) \vee (\neg x_1 \wedge \neg x_2 \wedge y_1 \wedge \neg y_2).$$

La figure 5.4 est l'OBDD représentant les transitions du graphe G avec l'ordre des variables $x_1 < x_2 < y_1 < y_2$. Remarquons que cet OBDD contient 5 noeuds.

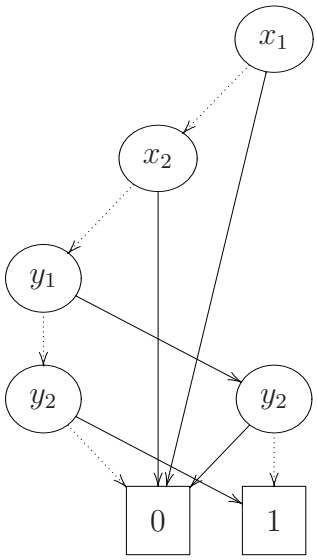
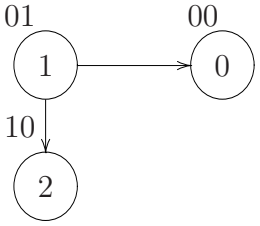


FIG. 5.4 – OBDD représentant les transitions du graphe G.

En changeant la numérotation, ici en inversant le 0 et le 1, nous obtenons une matrice de transitions différente et un OBDD de plus petite taille présentés respectivement aux figures 5.5 et 5.6.



donne la matrice $\begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$

FIG. 5.5 – Nouvelle numérotation du graphe G.

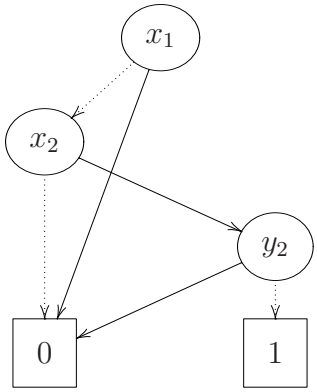


FIG. 5.6 – OBDD représentant le graphe G avec la nouvelle numérotation.

Nous remarquons que même sur un très petit graphe il peut y avoir une différence. Ainsi, en plus de bien choisir l'ordre, nous devons donc bien choisir la numérotation.

La question que nous nous sommes posée est : est-ce que bien numéroter les états est équivalent à bien choisir l'ordre des variables ? C'est sur ce problème que nous nous penchons à la section suivante.

5.4 Numérotation versus ordre des variables

Comme nous avons vu, il y a deux façons de numéroter les états : par les propositions atomiques et par la numérotation successive. Nous verrons dans cette section qu'il est nécessaire de bien choisir l'ordre des variables *et* la numérotation des états.

Tout d'abord, soulignons qu'il faut un minimum de $\lceil log_2 n \rceil$ variables booléennes pour représenter les n états. Voyons ce qu'il peut arriver :

1. si nous numérotons les états de façon successive, nous utilisons exactement $\lceil log_2 n \rceil$ variables ;
2. si nous utilisons les propositions atomiques, le nombre de variables dépend du nombre de propositions :
   a) si le nombre de propositions est plus petit que $\lceil log_2 n \rceil$, il faut ajouter des propositions car sinon nous ne pouvons pas distinguer tous les états. Ainsi nous avons au moins $\lceil log_2 n \rceil$ variables ;
   b) si le nombre de propositions atomiques est plus grand que $\lceil log_2 n \rceil$, nous utilisons toutes les propositions. Il y a donc plus de variables dans la forme propositionnelle qui représente l'ensemble des états que dans les cas 1 et $2a$), ce qui laisse croire, *à priori*, que la taille de l'OBDD est plus grande.

Proposition 5.4.1 *Lorsque les états d'un graphe sont numérotés avec p propositions atomiques, il y a*

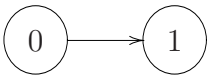
1. $p!$ *façons différentes de numéroter le graphe,*
2. $2p!$ *ordres de variables possibles dans l'OBDD qui représente les transitions du graphe et*
3. *moins de* $p!(2p)!$ *choix possibles pour une paire* $(numérotation, ordre)$.

Justifications *Pour un état, si la proposition p_1 est vraie et la proposition p_2 est fausse, nous pouvons numéroter cet état* 10 *ou* 01 *selon le cas où nous mettons p_1 avant ou après p_2. Dans le cas où on a p propositions, nous devons choisir la position des chacune d'elles et il y a $p!$ façons de le faire.*

Pour représenter les transitions, nous devons différencier les états de départ et d'arrivée. Ainsi nous avons $2p$ *variables dans l'OBDD. Ces* $2p$ *variables peuvent être dans n'importe quel ordre. Il y a* $(2p)!$ *ordres de variables possibles dans l'OBDD.*

Il arrive qu'un couple $(numérotation_1, ordre_1)$ *revienne au même qu'un autre choix* $(numérotation_2, ordre_2)$*. L'exemple suivant illustre le phénomène.*

Exemple 5.4.2 *Voici le graphe* G *à enregistrer. Avec cette numérotation et l'ordre*



de variable $x_0 < y_0$*, l'OBDD qui représente la transition du graphe* G *est celui de la figure 5.7. Cependant, si la numérotation et l'ordre de variable sont inversés, l'OBDD*

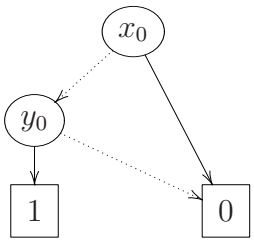


FIG. 5.7 – OBDD du graphe G.

obtenu est celui de la figure 5.8 et on remarque qu'il est identique au précédent (à un renommage de variables près).

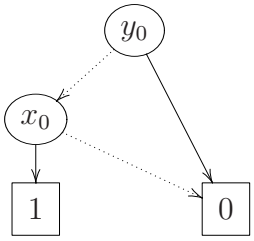


FIG. 5.8 – OBDD du graphe G avec un autre ordre et une autre numérotation.

Proposition 5.4.3 *Lorsque les* n *états d'un graphe sont numérotés sans propositions atomiques, il y a*

1. $n!$ *façons de numéroter les états,*

2. $(2\lceil log_2 n \rceil)!$ *ordres possibles pour les variables dans l'OBDD qui représente les transitions et*
3. *possiblement moins de* $n!(2\lceil log_2 n \rceil)!$ *combinaisons possibles pour le choix d'une paire* (*numérotation, ordre*).

Les justifications de ces énoncés sont identiques à celles de la proposition précédente.

Il existe des combinaisons (numérotation_1, ordre_1) qu'il n'est pas possible de retrouver en utilisant une autre paire de (numérotation_2, ordre_2). Ainsi, en se limitant au choix d'un seul des deux paramètres et en laissant l'autre fixe, il se peut que nous ne puissions jamais obtenir la combinaison optimale. Nous pouvons donc affirmer :

Proposition 5.4.4 *Pour diminuer la taille d'un OBDD au maximum, il n'est pas suffisant de bien choisir seulement la numérotation ou seulement l'ordre des variables.*

Dans les deux cas de numérotations, nous voyons qu'il y a beaucoup de combinaisons à étudier et il est bien sûr impensable de pouvoir toutes les essayer ! Certains auteurs suggèrent de choisir l'encodage des états et ensuite de choisir l'ordre des variables. Nous sommes plutôt en désaccord avec cette façon de faire pour plusieurs raisons. Premièrement, si nous savons que nous utilisons des méthodes dynamiques d'ordonnancement, le choix de l'encodage ne semble pas avoir de l'importance sur le fonctionnement des algorithmes. Par contre, dans le cas où nous utilisons une méthode statique, comme l'heuristique est utilisée une seule fois, il peut être intéressant d'étudier le fonctionnement de l'algorithme pour en déduire une numérotation qui le favoriserait. En particulier, il pourrait être très avantageux d'étudier un seul ordre des variables et de trouver quelles structures de matrice rend cet ordre plus efficace, pour ensuite chercher quel genre de numérotation engendre cette structure de matrice. C'est justement ce que nous nous proposons d'étudier avant de terminer ce chapitre.

Puisque la plupart du temps, n est beaucoup plus grand que p, nous nous attendons à ce qu'il y ait plus de choix possibles avec la numérotation successive et qu'alors, il peut être plus difficile de trouver la combinaison qui minimise la taille de l'OBDD.

Nous avons dit précédemment, qu'*a priori*, si $p > \lceil log_2 n \rceil$, p étant le nombre de propositions atomiques, l'OBDD risque d'être plus gros car il contient plus de variables. Ce n'est en fait pas toujours le cas. Il est connu qu'une modélisation de haut niveau d'un système favorise l'utilisation des méthodes symboliques puisqu'elle conserve la structure du système et que la taille des OBDD est moindre lorsque la matrice a une

certaine structure. Il est possible qu'une numérotation engendrée par un grand nombre de propositions atomiques, celui-ci étant causé par une modélisation de haut niveau, donne de très bons résultats en ce qui concerne la taille de l'OBDD.

5.4.1 Ordre des variables

Maintenant que nous avons étudié l'impact de l'ordre des variables et de la numérotation des états du graphe sur la taille des OBDD, nous présentons les choix que nous avons fait pour obtenir nos heuristiques. Nous avons d'abord choisi de fixer l'ordre des variables et de travailler sur la numérotation. L'ordre des variables choisi est celui qui alterne les variables de départ et d'arrivée :

$$x_1 < y_1 < x_2 < y_2 < ... < x_k < y_k.$$

Cet ordre est appelé l'*ordre alterné*. Ce choix est fait pour deux principales raisons. Tout d'abord, il facilite l'utilisation des OBDD dans plusieurs algorithmes qui nécessitent un accès rapide aux sous-matrices, comme les algorithmes récursifs. Par exemple, il existe un algorithme de multiplication de matrices récursif qui utilise la multiplication des sous-matrices. Les figures 5.9 et 5.10 illustrent cette multiplication matricielle. Cet exemple représente très bien l'utilité d'avoir accès rapidement aux sous-matrices.

$$A = \left(\begin{array}{c|c} a_1 & a_2 \\ \hline a_3 & a_4 \end{array}\right) \qquad B = \left(\begin{array}{c|c} b_1 & b_2 \\ \hline b_3 & b_4 \end{array}\right)$$

FIG. 5.9 – Décomposition des matrices en sous-matrices.

$$A \cdot B = \left(\begin{array}{c|c} a_1b_1 + a_2b_3 & a_1b_2 + a_2b_4 \\ \hline a_3b_1 + a_4b_3 & a_3b_2 + a_4b_4 \end{array}\right)$$

FIG. 5.10 – Multiplication récursive de matrice.

L'accès aux sous-matrices se fait efficacement. En effet, en observant la figure 5.11, nous voyons qu'après avoir déterminé la valeur de x_0 et de y_0 (les deux premières variables de l'OBDD) nous nous retrouvons avec une des 4 sous-matrices principales de la matrice. Pour accéder à la sous-matrice C dans cette même matrice, nous avons seulement à fixer la valeur des variables ainsi : $x_0 = 1$ et $y_0 = 0$.

Notre deuxième argument en faveur de cet ordre est qu'il est très souvent mentionné dans la littérature. Par exemple, l'utilisation de cet ordre des variables est le premier

$$\begin{array}{c} \\ x_0 = 0 \\ x_0 = 1 \end{array} \begin{array}{c} \begin{array}{cc} y_0 = 0 & y_0 = 1 \end{array} \\ \left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right) \end{array}$$

FIG. 5.11 – Accès à une sous-matrice.

principe de base présenté dans Hermanns et al. [29]. Il n'y a pas de démonstration théorique que cet ordre des variables soit meilleur que les autres, mais les résultats obtenus avec celui-ci sont, en général, très bons.

5.4.2 Numérotation

Nous avons donc choisi de fixer l'ordre des variables et nous allons maintenant nous intéresser à la numérotation. La numérotation choisie sera celle sans les propositions atomiques. Nous pensons qu'ainsi nous aurons plus de flexibilité. Il est connu que l'ordre alterné est plus performant lorsque les éléments non nuls de la matrice d'adjacence sont près de la diagonale principale de la matrice. Ceci est justifié par le fait que lorsque nous connaissons la valeur de x_0 et de y_0, nous pouvons savoir dans quelle sous-matrice nous nous trouvons. Si la matrice est presque diagonale, deux des quatre sous-matrices sont pratiquement nulles. Ceci se répercute dans l'OBDD par la disparition de certaines variables et nous connaissons la valeur de la fonction grâce aux valeurs d'un petit nombre de variables. Voyons un exemple.

Exemple 5.4.5 *Considérons la matrice de la figure 5.12.*

$$\begin{array}{c} \\ x_0 = 0 \\ x_0 = 1 \end{array} \begin{array}{c} \begin{array}{cc} y_0 = 0 & y_0 = 1 \end{array} \\ \left(\begin{array}{cc|cc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right) \end{array}$$

FIG. 5.12 – La matrice identité de dimension 4.

Nous voyons que si $x_0 = 0$ et que $y_0 = 1$, la valeur résultante est 0 peu importe la valeur de x_1 et y_1. Dans l'OBDD, dès que nous avons parcouru ces deux noeuds (x_0 et y_0), nous allons directement sur 0, ce qui réduit considérablement le nombre de noeuds. Dans cet exemple, l'OBDD obtenu est celui de la figure 5.13. Puisque x_0 et y_0 doivent nécessairement être égaux pour que la fonction s'évalue à 1, après avoir vérifié

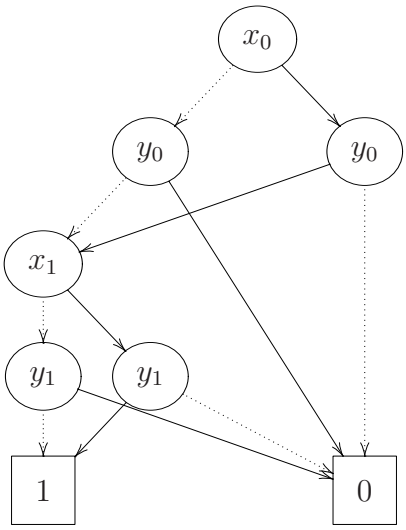


FIG. 5.13 – OBDD représentant la matrice identité.

seulement deux variables nous pouvons savoir si la fonction est nulle (dans le cas où x_0 et y_0 ne sont pas égaux). Et de même pour x_1 et y_1. Ainsi, nous devons tester toutes les variables seulement lorsque la fonction s'évalue à 1. *Étant donné que la matrice est creuse, nous ne devrions pas avoir à toutes les tester souvent.*

Dans le cas d'une matrice dont les éléments non-nuls sont près de la diagonale principale, le même effet que sur la matrice identité se produit généralement. Nous cherchons donc à *diagonaliser* la matrice pour obtenir l'effet de réduction de l'OBDD observé dans l'exemple précédent.

Une autre façon de justifier pourquoi nous voulons diagonaliser la matrice est que la régularité dans la matrice, c'est-à-dire le fait que la matrice ait des blocs de valeurs identiques, dans notre cas des blocs de zéros, se reflète en une diminution de la taille de l'OBDD. En effet, la règle 3 de réduction des OBDD implique que la taille des OBDD est réduite par la présence de sous-graphes isomorphes dans le diagramme de décision binaire. Ces sous-graphes isomorphes sont en partie engendrés par la régularité dans la matrice. Une numérotation qui favorise une telle régularité engendre donc un OBDD plus petit. Il est à noter que cette remarque respecte le cinquième principe de base de la section 5.2.

La section suivante montrera quelques méthodes pour trouver une numérotation qui favorisera l'utilisation de l'ordre alterné des variables en utilisant les observations susmentionnées.

5.5 Heuristiques

Contrairement aux heuristiques traditionnelles, les heuristiques présentées ici porteront sur la numérotation des états du modèle plutôt que sur l'ordre des variables puisque nous avons choisi de fixer celui-ci. Comme nous l'avons mentionné dans la section précédente, l'objectif de nos heuristiques est de mettre les éléments non-nuls le plus près possible de la diagonale principale de la matrice d'adjacence représentant les transitions d'un modèle.

Remarquons que dans plusieurs contextes comme la vérification, il est naturel de supposer que le nombre d'arcs dans le graphe est au plus $\frac{n^2}{2}$ (et possiblement beaucoup moins) puisqu'il n'y a pas beaucoup de possibilités de transitions à chaque état. Par contre, si ce n'était pas le cas, il serait alors préférable de mettre les éléments *nuls* sur les diagonales principales puisqu'il y aurait une majorité de 1 et qu'il serait possible de déterminer rapidement si la valuation donnée est une transition, plutôt que de déterminer si elle n'en est pas une. Dans la suite du texte, nous utiliserons le terme diagonale principale pour représenter la vraie diagonale ainsi que les diagonales adjacentes, celles au-dessus et celles en dessous (le nombre de diagonales incluses dépend de la dimension de la matrice : plus elle est grande, plus nous incluons de diagonales).

Nous présentons maintenant quelques heuristiques. Nous croyons que ces méthodes peuvent améliorer la performance des OBDD en établissant une numérotation des états qui favorise l'ordre des variables $x_0 < y_0 < x_1 < y_1 < ... < x_n < y_n$.

5.5.1 Chaîne de longueur maximale

Cette méthode consiste à trouver une chaîne de longueur maximale dans le graphe. Ici nous utilisons le terme *chaîne* pour désigner un parcours de noeuds dans un graphe, en respectant le sens des arcs. Les sommets de cette chaîne sont alors numérotés les uns après les autres. On poursuit en trouvant les chaînes de longueur moindre ou égale pour numéroter les sommets sans numéro et ainsi de suite jusqu'à ce que tous les sommets soient numérotés. Nous avons remarqué que les éléments non-nuls sur les diagonales forment une chaîne dans le graphe. En numérotant les noeuds de la chaîne la plus longue successivement, on s'assure d'avoir beaucoup d'éléments non-nuls sur la diagonale principale. De cette façon, nous favorisons l'utilisation de l'ordre des variables alterné. Voici un petit exemple où la technique a été utilisée et où les résultats escomptés ont effectivement été obtenus.

Exemple 5.5.1 *Voici comment nous avons numéroté les sommets du graphe de la figure 5.14. Remarquons que la chaîne* $[0, 1, 2, 3]$ *est une chaîne de longueur maximale (de longueur* 3*) ; ainsi ce sont ces noeuds que nous avons numérotés en premier (dans le sens de la chaîne). Il restait ensuite* 3 *noeuds. La chaîne* $[4, 5]$ *étant plus longue que* $[6]$*, nous la numérotons en premier. À titre indicatif, avec la numérotation obtenue par cette heuristique (tout en gardant l'ordre alterné des variables) l'OBDD qui représente les transitions de ce graphe a* 16 *noeuds. Avec* 20 *autres numérotations aléatoires, l'OBDD avait entre* 17 *et* 20 *noeuds inclusivement.*

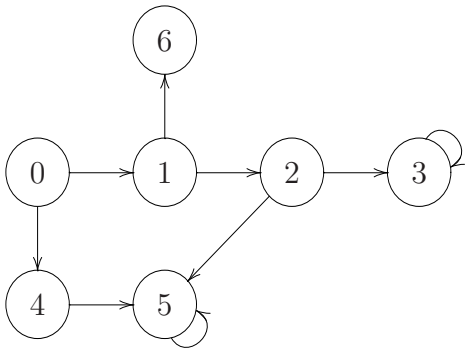


FIG. 5.14 – Numérotation à l'aide de la méthode de la chaîne de longueur maximale.

Cette méthode force les 1 à être sur la diagonale principale. En fait il y a autant de 1 sur la diagonale qu'il y a d'arcs dans les chaînes et de boucles sur les états. Les seules valeurs qui ne sont pas dans la diagonale sont causées par les arcs qui lient des noeuds appartenant à des chaînes différentes. Cette méthode peut aider à réduire la mémoire utilisée puisqu'elle donne une matrice très régulière et elle favorise l'utilisation de l'ordre des variables $x_0 < y_0 < x_1 < y_1 < ... < x_n < y_n$. Pour pouvoir faire la comparaison, la figure 5.15 montre la matrice de transitions du graphe avec la numérotation de la figure 5.14 et celle du même graphe avec une numérotation aléatoire.

Remarquons que dans la matrice a) les 1 sont plus près de la diagonale. Les 1 sur la diagonale sont en fait les arcs dans les chaînes déterminées par l'algorithme. Nous remarquons que plus la chaîne de longueur maximale est longue, plus il y aura de 1 dans la diagonale principale. L'effet est beaucoup plus marquant lorsque la matrice est de grande taille.

Malheureusement, trouver une chaîne de longueur maximale dans un graphe est un problème $\mathcal{NP}$*-Difficile* [21]. Par contre, si nous avons des heuristiques pour trouver une chaîne de longueur quasi maximale, alors nous pouvons les utiliser dans cette méthode

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

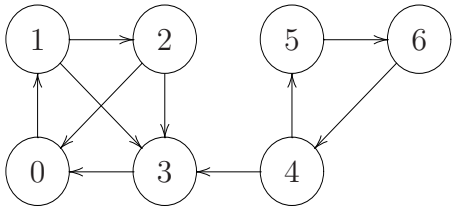
a) b)

FIG. 5.15 – Matrices de transitions du graphe de la figure 5.14.

puisqu'il n'est pas absolument nécessaire d'avoir toujours la plus grande chaîne. En effet, nous cherchons une heuristique et non pas une méthode exacte.

5.5.2 Composantes k-connexes

Voici une autre de nos méthodes qui tente de mettre les 1 le plus près possible de la diagonale principale. L'idée consiste à numéroter les états qui sont liés par plusieurs arcs de façon à ce que leurs numéros soient le plus près possible les uns des autres. Ainsi nous tentons de regrouper les 1 et de les disposer près de la diagonale principale. La figure 5.16 est un exemple simple sur lequel l'algorithme a été appliqué. Sur ce graphe



a) b)

FIG. 5.16 – Numérotation à l'aide de l'étude des composantes dont les noeuds sont très liés.

simple, deux composantes sont rapidement identifiables, soit la composante a) et la composante b). Nous avons choisi de numéroter la composante a) en premier (choix arbitraire).

Définition 5.5.2 *Une composante connexe d'un graphe est un ensemble de sommets pour lesquels il existe un chemin entre chaque paire de sommet de l'ensemble.*

Ainsi le graphe de la figure 5.16 a une seule composante connexe. Ce qui nous intéresse est de déterminer les sommets qui sont fortement liés, d'où l'utilisation des composantes k-connexes.

Définition 5.5.3 *Une composante k-connexe est une composante connexe dans laquelle il existe plus d'un chemin entre chaque paire de sommets. Nous dirons k-connexe quand il existe au moins k chemins différents entre chaque paire de sommets.*

L'heuristique doit déterminer les composantes k-connexes pour un k élevé et ensuite numéroter de façon successive les sommets de chaque composante. Il faut aussi choisir dans quel ordre les composantes sont numérotées. Encore une fois, il faut les numéroter pour que celles qui sont liées soient près, et pour ce faire on peut regarder les composantes $(k-1)$-connexes et ainsi de suite. Il est important de remarquer que de trouver les composantes k-connexes d'un graphe est un problème $\mathcal{NP}$-*Difficile* [21].

Encore une fois dans un but indicatif, l'OBDD représentant le graphe de la figure 5.16 (avec la numérotation déterminée par notre heuristique) a 17 noeuds. Avec 20 autres numérotations aléatoires, nous avons obtenu des OBDD qui ont entre 19 et 22 noeuds. La figure 5.17 montre la forme que donne la méthode à la matrice de transitions en comparaison à une numérotation aléatoire.

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

a)

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

b)

FIG. 5.17 – Matrices de transitions du graphe de la figure 5.16.

Nous voyons que les 1 sont moins éparpillés dans la matrice a) même s'ils ne sont pas exactement sur la diagonale principale. Sur un petit exemple comme celui-ci, il est moins clair que les 1 sont regroupés près de la diagonale, mais sur un gros exemple nous verrions des amas de 1 concentrés autour de la diagonale principale.

5.5.3 Problème de la largeur de bande

Il y a depuis longtemps un problème couramment rencontré en mathématique et plus précisément en théorie des graphes et en analyse numérique. Ce problème est celui de la largeur de bande (*bandwidth reduction problem* en anglais). Les stratégies développées pour résoudre ce problème sont de bonnes stratégies pour mettre les éléments non-nuls le plus près possible de la diagonale. Voici l'énoncé de ce problème.

Énoncé 5.5.4 (Problème de la largeur de bande [53]) *Soit $G = \langle V, E \rangle$ un graphe avec V l'ensemble de ses sommets et E l'ensemble de ses arêtes. Les éléments de V sont numérotés. Trouver quelle permutation p des sommets de V minimise la grandeur de la plus grande arête lorsque les sommets sont alignés, c'est-à-dire choisir p tel que*

$$max_{(i,j)\in E} \mid p(i) - p(j) \mid$$

est minimal.

Par exemple, si nous résolvons ce problème sur le graphe de la figure 5.18, nous pourrions obtenir la permutation présentée à la figure 5.19.

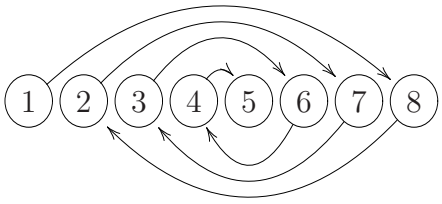


FIG. 5.18 – Graphe G.

FIG. 5.19 – Solution au problème de la largeur de bande sur le graphe G.

Ce problème intervient, par exemple, en théorie des graphes lorsqu'on veut aligner un ensemble de composantes d'un circuit sur une plaque et minimiser le temps de communication entre les composantes. En analyse numérique, ce problème intervient lorsqu'on veut résoudre un système linéaire de façon plus efficace. Il est possible d'améliorer considérablement le temps d'exécution de l'algorithme d'élimination de Gauss lorsque la distance d qu'ont les éléments non-nuls à la diagonale principale est petite. La résolution du problème de la largeur de bande sur le graphe représenté par la matrice minimise

justement la distance d. Ainsi, le problème de la largeur de bande est exactement notre problème de mettre les éléments non-nuls près de la diagonale.

Malheureusement, comme nous pouvions nous en douter, ce problème fait partie de la classe des problèmes $\mathcal{NP}$-*Difficile*. Il n'y a pas non plus d'algorithme d'approximation pour ce problème. Cependant, il a beaucoup été étudié et plusieurs bonnes heuristiques ont été proposées. Les deux algorithmes les plus populaires sont celui de Cuthill-McKee et celui de Gibbs-Poole-Stockmeyer. Il semble qu'il y ait plus de 45 différentes heuristiques pour le problème de la largeur de bande. La plupart des algorithmes exécutent un parcours en largeur sur le graphe à partir d'un sommet de départ. Le choix de ce sommet ainsi que d'autres détails d'implémentation diffèrent d'un algorithme à l'autre. Pour une description et une comparaison de quelques-uns des algorithmes voir Saad [49] et Gibbs et al. [22].

En vérification, il y a peu d'heuristiques qui tentent de réduire la taille d'un OBDD représentant un graphe en changeant la numérotation des états du graphe. De plus, les personnes qui ont remarqué que la structure de la matrice de transitions influence la taille du OBDD la représentant n'ont pas fait le lien avec le problème de largeur de bande, lequel est pourtant bien connu en mathématique.

Il serait très intéressant de comparer les résultats obtenus par les algorithmes présentés précédemment et les heuristiques couramment utilisées en mathématique. Il sortait cependant du cadre de ce travail d'implémenter ces algorithmes et de vérifier s'ils diminuent de beaucoup la taille des OBDD représentant des graphes.

Comme nous l'avons mentionné au début de ce chapitre, l'objectif de ce travail n'est pas d'étudier et d'élaborer des heuristiques. Cependant, nous avons été amenés à nous poser des questions sur ce sujet que nous trouvons intéressant. Nous n'utiliserons donc pas ces heuristiques dans la suite de ce travail.

Chapitre 6

Implémentation et résultats

Dans ce chapitre, nous présenterons CISMO, un vérificateur de modèles probabilistes à espace d'états continu. De plus nous discuterons des modifications que nous avons apportées à ce vérificateur de modèles pour lui permettre de faire de la vérification *symbolique* à l'aide des MTBDD. Mais tout d'abord, nous verrons qu'il existe plusieurs vérificateurs de modèles et que plusieurs d'entre eux utilisent les OBDD.

6.1 OBDD dans les vérificateurs de modèles

Rappelons qu'un vérificateur de modèles est un outil qui permet de faire de la vérification automatique de modèles. L'utilisateur doit donner en entrée le modèle ainsi que les propriétés à vérifier et l'outil répond que le modèle respecte ou non ces propriétés. Si le modèle ne satisfait pas une propriété, certains outils offrent la possibilité d'obtenir une trace d'exécution qui témoigne du non-respect de la propriété.

Il y a beaucoup d'outils qui permettent de faire la vérification. On retrouve sur la page de la bibliothèque virtuelle à propos des méthodes formelles [11] une liste de projets en méthodes formelles qui contient plusieurs vérificateurs de modèles. Pour ne nommer que quelques-uns des vérificateurs de modèles existants, nommons SMV [40], SPIN [1], ProbVerus [26], PRISM [46] et CISMO [48]. Les deux premiers outils sont utilisés pour valider des systèmes finis non-probabilistes avec respectivement la logique CTL et LTL. Les trois derniers sont des outils utilisés pour valider des systèmes probabilistes. PRISM vérifie les systèmes à espace d'états discret et CISMO vérifie les systèmes à espace d'états continu. Avec certains de ces outils, la vérification a été faite

efficacement pour vérifier autant le matériel électrique que les logiciels.

Les diagrammes de décision binaire sont utilisés pour représenter l'espace des états d'un système ainsi que ses transitions. Cette technique a été un élément qui a aidé au succès de la vérification. Elle a permis de valider des modèles comportant beaucoup plus d'états ; par exemple, Burch [31] rapporte avoir pu vérifier, en utilisant des OBDD, un modèle ayant $5 \cdot 10^{120}$ états. De nos jours, beaucoup de vérificateurs de modèles les utilisent. Le vérificateur de logiciel SMV fut le premier à faire de la vérification en utilisant les OBDD. Le vérificateur de modèles ProbVerus est aussi un vérificateur de modèles probabilistes qui utilise les OBDD.

Étant donné le succès obtenu avec les OBDD pour la vérification des modèles non-probabilistes, une équipe de l'University de Birmingham a décidé d'ajouter à l'outil PRISM l'option de faire de la vérification symbolique de modèles probabilistes. Pour gérer ses MTBDD, ce vérificateur de modèles utilise CUDD, le paquetage de manipulation d'OBDD et de MTBDD présenté à la section 3.5.6. Il y a aussi dans PRISM une option qui permet d'utiliser la méthode standard de stockage des matrices creuses pour manipuler le modèle. Ainsi, grâce à ce logiciel, on peut comparer la performance des deux méthodes. Cependant, les résultats escomptés par l'utilisation des MTBDD dans PRISM ne furent pas observés. C'est pourquoi une méthode dite hybride, qui est un mélange de la méthode standard et de la méthode par MTBDD, a été ajoutée. Cette méthode semble donner de bons résultats. Une description complète de la méthode hybride est donnée par Parker [46]. Nous verrons en détail pourquoi les MTBDD n'ont pas été efficace dans PRISM et pourquoi nous croyons qu'ils peuvent l'être dans notre application.

6.2 CISMO

CISMO est un outil de vérification de systèmes probabilistes à espace d'états continu. Il peut vérifier une certaine classe de LMP (restreinte par le langage utilisé pour décrire les LMP dans le vérificateur). Ce programme a été développé en JAVA par Richard [48] dans le cadre de sa maîtrise. Avec cet outil, il est possible de mettre en mémoire un modèle et de vérifier si celui-ci satisfait des propriétés décrites par la logique présentée à la section 2.3.2.

6.3 Utilisation des MTBDD dans CISMO

Dans l'implémentation initiale de CISMO, les modèles sont enregistrés en utilisant des structures de données offertes en JAVA. Ces structures sont optimisées pour améliorer la vitesse des manipulations. Par exemple, l'utilisation d'un *hashmap* facilite la recherche d'un élément qui a été associé à une clef. De cette façon, la première version de CISMO met l'accent sur la rapidité et non la minimisation de la mémoire utilisée. Pour des modèles de petite à moyenne taille, cette implémentation donne de bons résultats. Cependant, nous appréhendons le jour où une longue formule devra être vérifiée sur un modèle de grande taille et que CISMO ne pourra le faire par manque de mémoire. Nous avons tenté de créer un tel modèle mais sans succès. En fait, un tel modèle est très long à construire. À l'aide de CISMO, nous avons réussi à vérifier une formule, contenant 12 opérateurs probabilistes imbriqués, sur un modèle possédant un peu plus de 150 transitions en quelques secondes.

Notre objectif était d'ajouter l'option « MTBDD » à l'outil. Cette option permet à l'utilisateur de choisir le type de structures de données utilisées pour stocker le modèle probabiliste.

Les modifications ayant été faites, le modèle peut maintenant être enregistré d'une façon ou d'une autre. De plus, il est possible de changer de structure lors de la manipulation du modèle. De cette façon, il nous est possible de vérifier les petits modèles avec les structures de la première version, ce qui accélère la vérification. Nous pouvons aussi faire de la vérification symbolique avec MTBDD, ce qui permet de vérifier des modèles de plus grande taille.

6.3.1 Enregistrement des LMP

Un LMP est composé d'intervalles et de fonctions de répartition. En plus de devoir garder en mémoire ces informations, il faut conserver l'association entre ceux-ci. Pour enregistrer les transitions, nous construisons un MTBDD dont les feuilles sont les fonctions de répartition associées aux transitions. Les variables du MTBDD représentent les intervalles de départ et d'arrivée des transitions. Ces variables sont déterminées par une numérotation des intervalles de la même façon que dans les modèles à espace d'états discret. Chaque intervalle, action et fonction de répartition est associé à un entier. Cette numérotation se fait de façon aléatoire.

Pour bien comprendre comment il est possible de représenter un LMP à l'aide d'un

MTBDD, nous allons construire le MTBDD qui représente le LMP de la figure 6.1, que nous avons déjà rencontré à la figure 2.6. De plus, nous utiliserons cet exemple tout au long de l'analyse pour faciliter la compréhension des explications. Avant de construire

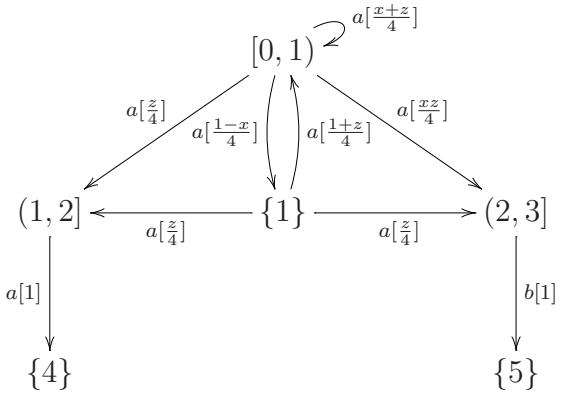


FIG. 6.1 – LMP de l'exemple 2.6.

le MTBDD, voyons sous quel format un LMP est donné en paramètre à CISMO. La figure 6.2 montre le fichier d'entrée donné à CISMO pour charger le LMP de la figure 2.6.

Remarquons qu'il faut spécifier d'abord l'ensemble des états possibles du système, l'état initial ainsi que le nombre d'actions. Ensuite, les intervalles de départ et d'arrivée sont spécifiés et chaque transition est décrite. Pour chaque transition, il faut donner l'intervalle de départ, l'action, la fonction de répartition et l'intervalle d'arrivée. La fonction de répartition détermine l'état d'arrivée de la transition lorsque le système est dans un état de l'intervalle de départ et qu'il fait l'action donnée.

Pour construire le MTBDD, nous devons numéroter les intervalles et les actions. Les variables du MTBDD seront déterminées par cette numérotation. Nous prenons une variable pour chaque action (nous verrons pourquoi dans ce qui suit) et les autres variables représentent les états de départ et d'arrivée. Pour notre exemple, une numérotation possible est la suivante :

$$\begin{array}{ll} a := 0, & b := 1, \\ [0,1) := 0, & (1,2] := 1, \\ \{1\} := 2, & (2,3] := 3, \\ \{4\} := 4, & \{5\} := 5. \end{array}$$

Puisqu'il y a 2 actions il faut deux variables d'action. Les intervalles sont considérés comme les sommets d'un graphe et le nombre de variables est alors déterminé comme

```
states : [0,3] U {4,5};
init : 1.0 ;

NbAction : 2;

X : [0,1), {1}, (1,2], (2,3];
Y : [0,1), {1}, (1,2], (2,3], {4},{5};

transition
        [0,1) -[a]-> z/4:(1.0,2.0];
        [0,1) -[a]-> 1:{4.0};
        [0,1) -[a]-> x/4*(y-2):(2.0,3.0];
        [0,1) -[a]-> (1-x)/4:{1.0};
        {1} -[a]-> y/4:(0.0,1.0);
        {1} -[a]-> x/4:{0.0};
        {1} -[a]-> y/4:(1.0,2.0];
        (1,2] -[a]-> y/4:(1.0,2.0];
        (2,3] -[b]-> 1:{5.0};
endtransition
```

FIG. 6.2 – Exemple d'un fichier d'entrée de CISMO.

dans les systèmes à espace d'états discret. Comme il y a 6 intervalles différents, il faut $\lceil log_2(6) \rceil = 3$ variables.

Ensuite, il faut choisir un ordre pour les variables et il reste à construire le MTBDD comme lorsque nous représentions un graphe. Dans le cas des LMP, les feuilles sont les fonctions de répartition. Si nous choisissons de mettre les variables d'action au début et ensuite d'alterner les variables d'intervalles, le MTBDD qui représente le LMP de la figure 6.1 est celui de la figure 6.3. Pour faciliter sa lecture, nous avons omis de mettre la feuille étiquetée par la fonction 0 ainsi que tous les arcs qui pointent sur celle-ci.

Comparativement au fichier d'entrée, le MTBDD peut paraître gros. Cependant, le format utilisé pour enregistrer le LMP dans la première version de CISMO utilise aussi beaucoup d'espace. En fait, comme nous le verrons par la suite, il en utilise plus que la version avec MTBDD.

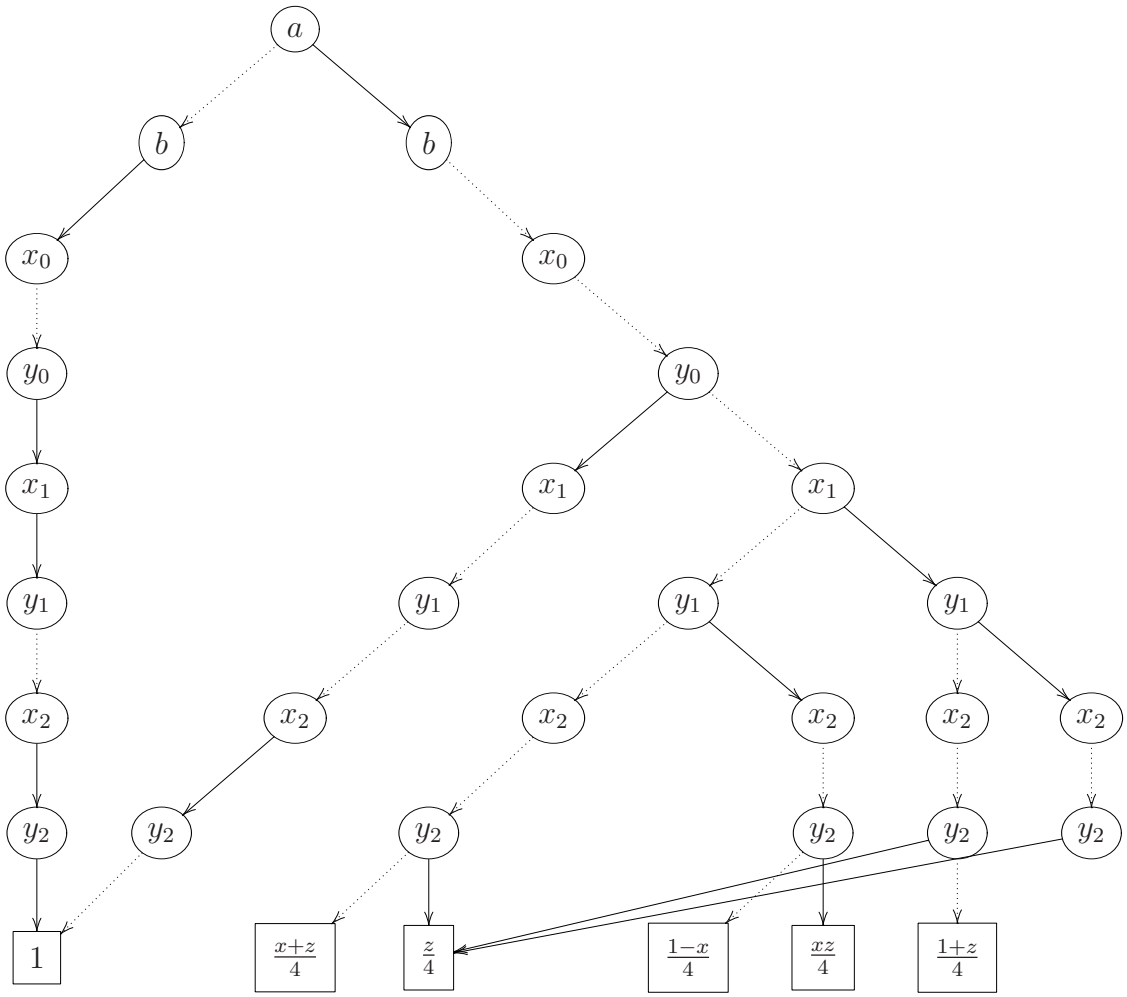


FIG. 6.3 – MTBDD représentant le LMP de l'exemple 2.6.

6.3.2 Choix de la structure

Pour enregistrer un LMP en mémoire de façon symbolique, nous avions le choix entre quatre structures qui permettent d'autres feuilles que 0 et 1. Ces quatre structures sont celles présentées à la section 4.2, les BMD, les EVBDD, les DNBDD et les MTBDD. Nous avons éliminé les BMD et les EVBDD car ils ne semblaient pas donner de bons résultats. Nous avons mis de côté les DNBDD puisque ceux-ci enregistrent chaque occurrence des entrées tandis que les MTBDD les enregistrent une seule fois. Dans notre application, nous croyons qu'il est possible qu'il y ait beaucoup de redondance, alors il était préférable d'opter pour les MTBDD.

Mais pourquoi avoir choisi les MTBDD malgré les résultats décevants de l'expérience

de PRISM ? Il faut savoir ce qui faisait que les MTBDD dans PRISM devenaient trop gros. Comme nous l'avons vu dans le deuxième chapitre, lorsqu'un DTMC doit être vérifié, nous pouvons être amenés à résoudre un système d'équations linéaires. Il en est de même pour les MDP et les CTMC. Ces trois types de modèles sont ceux que PRISM vérifie. Dans Parker [46], on apprend que la croissance de la taille du MTBDD est surtout liée à la résolution de systèmes d'équations linéaires. Comme aucun système d'équations n'est résolu dans CISMO, nous avons donc décidé d'utiliser les MTBDD puisque les aspects négatifs décelés dans PRISM ne risquent pas de survenir.

6.3.3 Ordre des variables

Pour faciliter les manipulations et éviter de trop souvent changer l'ordre des variables, nous avons choisi de laisser l'ordre des variables fixe dans notre implémentation. Les premières variables représentent chacune une action. Nous avons fait ce choix puisque dans la thèse du concepteur de PRISM [46], on explique que des tests on été faits et qu'il est préférable de positionner les variables d'actions au début et de laisser une variable par action plutôt que de choisir $\lceil log_2\ n \rceil$ variables pour les n actions.

Les autres variables représentent les numéros d'intervalles de début et de fin des transitions. Les variables de début et de fin de transitions sont alternées. Nous obtenons donc comme ordre $a_0 < a_1 < ... < a_n < x_k < y_k < x_{k-1} < y_{k-1} < ... < x_0 < y_0$ où les a_i représentent les n variables d'actions, les x_j représentent les intervalles de départ et les y_j représentent les intervalles d'arrivée pour un système à 2^k intervalles. Remarquons que les variables ont été mises dans un ordre inverse (x_k avant x_{k-1}). Ce choix a été fait pour faciliter l'implémentation. Dans notre cadre de travail, cet ordre est équivalent à l'ordre alterné décrit auparavant.

6.3.4 Implémentation

Pour manipuler les MTBDD, nous avons utilisé le paquetage CUDD. Comme nous l'avons déjà mentionné, CUDD est écrit en C et en C++. Ainsi, pour l'utiliser dans CISMO, écrit en JAVA, il a fallu utiliser la JNI (*Java Native Interface*) qui permet d'utiliser du code natif dans un programme JAVA.

CUDD a beaucoup facilité la programmation puisque toutes les manipulations élémentaires se font par l'appel de fonctions déjà existantes. De plus, la gestion de la table unique et du ramasse-miette se fait sans l'intervention du programmeur.

La plus importante restriction imposée par CUDD est le manque de portabilité. En effet, CISMO fonctionne maintenant seulement sur les plates-formes Linux et Unix puisqu'il en est ainsi pour CUDD. Cependant, PRISM utilise ce même paquetage et la dernière version est disponible pour Windows. Ceci indique qu'il serait possible et intéressant de rendre CISMO accessible pour Windows.

Un des problèmes rencontrés lors de l'implémentation est que CUDD ne permet pas qu'un MTBDD ait comme feuilles des fonctions. En effet, CUDD accepte seulement des nombres réels. C'est pour cette raison que nous avons dû associer un numéro à chaque fonction. De cette façon, lorsqu'on évalue le MTBDD, on obtient un numéro et ensuite on trouve la fonction associée dans le tableau qui les contient toutes.

6.4 Comparaison théorique de l'utilisation de la mémoire

Regardons maintenant l'efficacité de l'utilisation des MTBDD dans CISMO en comparaison avec la méthode sans MTBDD. Nous comparerons d'abord la quantité théorique de mémoire utilisée avec et sans les MTBDD. Ensuite, nous expliquerons les difficultés rencontrées dans la validation de ces résultats en pratique.

La seule façon de calculer la taille d'un MTBDD est de compter son nombre de noeuds et de le multiplier par la taille d'un noeud (la taille d'un noeud est fixe). Pour comparer la mémoire utilisée par les deux techniques, avec et sans MTBDD, nous devons donc calculer la taille, en *octets*, de chaque structure utilisée. De cette façon, nous pourrons comparer le nombre d'octets utilisés par les deux méthodes.

Difficulté

Le problème majeur lors de cette analyse est qu'il est très difficile d'évaluer la mémoire utilisée par un objet dans un programme écrit en JAVA. Pour un objet donné, il n'existe pas de méthode qui nous donne la mémoire occupée par l'objet. De plus, la taille d'un objet dépend de l'implémentation de la JVM car il n'y a pas de standard à ce sujet. Il est donc incontournable de faire une estimation de celle-ci, ce qui pourrait biaiser les résultats si l'estimation est très différente de la réalité.

Estimation de la taille

Pour estimer la mémoire de chaque objet, nous avons utilisé du code JAVA (voir annexe B), disponible sur le site de la compagnie *Sun* [58]. Ce programme construit plusieurs fois le même objet (des milliers de fois pour des petits objets) et calcule la mémoire utilisée par tous ces objets. Ensuite, cette quantité est divisée par le nombre d'objets créés. Une des faiblesses de cette technique est causée par le ramasse-miettes de JAVA. Lorsque des milliers d'objets sont créés, le ramasse-miettes peut s'activer et libérer de la mémoire préalablement allouée et inutilisée. Ainsi, il pourrait arriver que la mémoire utilisée par un objet apparaisse négative si la mémoire libérée est supérieure à la mémoire utilisée par les objets. Une façon de contrer cet effet est de tenter de forcer l'activation du ramasse-miettes avant la création des objets. Mais en aucun cas, nous ne pouvons être certain qu'il sera activé.

Dans notre analyse, lorsque nous parlerons d'un objet JAVA, nous mettrons le nom de l'objet en italique pour le différentier de l'objet mathématique. Par exemple, si nous écrivons « intervalle » nous faisons référence à un intervalle de nombres réels tandis que « *Interval* » représente un objet de la classe *Interval*.

L'analyse qui suit est basée sur les estimations faites par l'algorithme décrit précédemment. S'il advenait que l'estimation de la taille des objets considérés soit différente de la réalité, l'analyse devrait être revue. Voici la taille estimée des objets de base qui sont manipulés dans CISMO ; ces estimations nous permettrons d'évaluer le nombre d'octets utilisés par le logiciel.

Objets	Estimation de la taille en octets
Action	24
Interval	32
ExpressionProgram	144

La taille d'une *Action* à été déterminée en calculant la taille d'une chaîne de caractères (*String*) à un seul caractère. Le nom d'une action pourrait être composé de plusieurs caractères, mais nous avons choisi de considérer le minimum. Pour la taille d'un *Interval*, nous avons calculé la taille d'un intervalle de nombre réel quelconque. Une *ExpressionProgram* est un objet qui représente une fonction de répartition. Encore une fois, l'estimation est faite sur une fonction de répartition quelconque.

Stratégie d'analyse

Pour simplifier l'évaluation de la mémoire, nous calculons la taille des éléments propres à chaque méthode et nous ne comptons pas les structures communes au deux méthodes. De plus, nous supposons qu'il y a autant d'intervalles de départ que d'intervalles d'arrivée. Voici la liste des symboles utilisés pour la suite de l'analyse :

a	le nombre d'actions dans le système
k	le nombre d'intervalles de départ et d'arrivée dans les transitions du système
P	le nombre de fonctions de densité dans le système
v_i	variables $\in \mathbf{R}$, $v_i \geq 1$

Nous omettons de compter les octets nécessaires aux structures englobant les données, par exemple, les *HashMap*, *ArrayList*, *actInitialPair*, etc. Nous comptons la mémoire des objets contenus dans ces structures sans ajouter les octets nécessaires pour gérer la structure. Les résultats de l'analyse seront ainsi inférieurs à la réalité mais seulement d'une constante.

Avant de calculer la mémoire utilisée par les structures, rappelons qu'en JAVA, chaque objet n'est stocké réellement qu'une seule fois, et que tout accès à l'objet se fait par *référence* (au sens JAVA), si la copie profonde n'est pas explicite. De plus, chaque référence utilise 4 octets de mémoire. Pour différencier un objet d'une référence à un objet, nous noterons la référence à l'objet O par O^*.

Pour chaque structure à analyser, nous donnerons d'abord sa description et nous compterons le nombre d'octets que la structure utilise théoriquement. Pour mieux comprendre les explications, nous visualiserons ces mêmes structures dans le cas de la figure 2.6 reprise en 6.1.

Première version de CISMO (méthode sans MTBDD)

Nous faisons maintenant quelques remarques sur la façon dont les objets sont enregistrés dans la première version de CISMO et sur le nombre de transitions dans un modèle de type LMP.

Dans CISMO, aucune copie profonde n'est faite. En mode sans MTBDD, le seul endroit où les objets sont réellement en mémoire est dans la pile d'exécution. Cependant,

il peut arriver qu'un même *Interval* soit enregistré plusieurs fois dans la pile (cela aurait pu être évité en changeant des détails d'implémentation, ce qui réduirait la mémoire utilisée). Les structures de données qui contiennent cet *Interval*, ne contiennent en fait qu'une référence.

Remarquons que dans un modèle, il y a $\frac{ak^2}{v_0}$ transitions car pour une transition, on peut choisir parmi a actions, k intervalles de départ et k intervalles d'arrivée. La variable v_0 exprime le fait qu'il y a généralement moins de ak^2 transitions dans le modèle. Par exemple, dans le LMP de l'exemple 2.6, il pourrait y avoir jusqu'à $2{\cdot}4{\cdot}6 = 48$ transitions, mais il n'y en a que 9. Dans ce modèle, la variable v_0 est donc $5{,}\overline{3}$.

6.4.1 Analyse de la méthode sans MTBDD

Voici la description et l'analyse des structures de données qui sont utilisées dans le mode sans MTBDD et qui ne sont pas utilisées dans le mode MTBDD.

- La première structure est un *HashMap* nommé *startingStates*. Cette structure sert à faire l'association entre chaque action et les intervalles à partir desquels ces actions sont possibles. Tous les objets dans cette structure sont des références.

  Pour mettre cette structure en mémoire, il faut d'abord enregistrer toutes les actions. Puisqu'il y en a a, nous avons besoin de $4a$ octets. Ensuite, il faut enregistrer les intervalles associés à ces actions. Il y a k intervalles de départ, donc au maximum k intervalles associés à chaque action. Au total, il faut stocker $\frac{ak}{v_1}$ références aux intervalles (la variable v_1 exprime le fait qu'il y a généralement moins de k intervalles pour chaque action). Ces références utilisent au total $\frac{4ak}{v_1}$ octets. Pour l'ensemble de la structure, $4a + \frac{4ak}{v_1}$ octets sont utilisés.

  La structure *startingStates* de l'exemple 2.6 se présente comme suit

  $$\begin{array}{lcl} a^* & : & [0,1)^*,\ (1,2]^*,\ \{1\}^* \\ b^* & : & (2,3]^* \end{array}$$

  elle nécessite 24 octets puisqu'elle enregistre 6 références. Il pourrait avoir 4 intervalles associés *à chaque* action, mais il y en a 4 *en tout*. Pour cet exemple, v_1 est égale à 2.

- La deuxième structure à enregistrer est aussi un *HashMap*, nommé *endingStates*. Cette structure sert à faire l'association entre les couples (*act*, *d*) et les intervalles

d'arrivée des transitions qui débutent dans l'intervalle d et fait l'action *act*. Tous les objets dans cette structure sont des références.

Par l'analyse de la structure précédente, nous savons qu'il y a $\frac{ak}{v_1}$ couples (action, intervalle de départ). Dans chacun de ces couples, il y a deux références, il faut donc $\frac{8ak}{v_1}$ octets pour enregistrer les couples. Comme il y a k intervalles d'arrivée, il y a au plus k intervalles associés à chaque couple, disons exactement $\frac{k}{v_2}$. Nous obtenons ainsi $\frac{4ak \cdot k}{v_1 v_2}$ octets pour enregistrer les références sur les *Interval*. La structure *endingStates* utilise au total $\frac{8ak}{v_1} + \frac{4ak^2}{v_1 v_2}$ octets.

La structure *endingStates* de l'exemple 2.6 se présente comme suit

$$\begin{array}{lcl} (a^*, [0,1)^*) & : & [0,1)^*,\ (1,2]^*,\ \{1\}^*,\ (2,3]^* \\ (a^*, (1,2]^*) & : & \{4\}^* \\ (a^*, \{1\}^*) & : & [0,1)^*,\ (1,2]^*,\ (2,3]^* \\ (b^*, (2,3]^*) & : & \{5\}^* \end{array}$$

elle nécessite 68 octets puisqu'elle enregistre 17 références. Il pourrait avoir 6 intervalles associés à chaque couple (donc 24 en tout), mais il y en a 9 en tout. La variable v_2 de cet exemple est donc $2,\overline{6}$.

- La dernière structure à analyser est un *HashMap* nommé *transitions*. Cette structure sert à faire l'association entre les triplets (action, intervalle de début, intervalle d'arrivée) et la transition elle-même, *i.e.* un objet de la classe *Transition*.

  Chaque triplet représente une transition. Il y a donc autant de triplets que de transitions et nous avons mentionné précédemment qu'il y en a $\frac{ak^2}{v_0}$. Les objets dans les triplets sont des références (trois par triplet), ainsi ils utilisent $\frac{12ak^2}{v_0}$ octets.

  Il reste à enregistrer les objets *Transition*. Un objet de cette classe contient l'intervalle de départ, l'action, l'intervalle d'arrivée et la fonction de répartition de la transition représentée. Ces objets sont aussi des références. Ainsi il faut $\frac{16ak^2}{v_0}$ octets.

  La structure *transitions* de l'exemple 2.6 se présente comme suit

$$
\begin{array}{lcl}
(a^*, [0,1)^*, [0,1)^*) & : & (a^*, [0,1)^*, [0,1)^*, \frac{x+z}{4}^*) \\
(a^*, [0,1)^*, (1,2]^*) & : & (a^*, [0,1)^*, (1,2]^*, \frac{z}{4}^*) \\
(a^*, [0,1)^*, \{1\}^*) & : & (a^*, [0,1)^*, \{1\}^*, \frac{1-x}{4}^*) \\
(a^*, [0,1)^*, (2,3]^*) & : & (a^*, [0,1)^*, (2,3]^*, \frac{xz}{4}^*) \\
(a^*, (1,2]^*, \{4\}^*) & : & (a^*, (1,2]^*, \{4\}^*, 1^*) \\
(a^*, \{1\}^*, [0,1)^*) & : & (a^*, \{1\}^*, [0,1)^*, \frac{1+z}{4}^*) \\
(a^*, \{1\}^*, (1,2]^*) & : & (a^*, \{1\}^*, (1,2]^*, \frac{z}{4}^*) \\
(a^*, \{1\}^*, (2,3]^*) & : & (a^*, \{1\}^*, (2,3]^*, \frac{z}{4}^*) \\
(b^*, (2,3]^*, \{5\}^*) & : & (b^*, (2,3]^*, \{5\}^*, 1^*)
\end{array}
$$

elle nécessite 252 octets puisqu'elle enregistre 63 références.

De plus, les objets sont enregistrés en mémoire transition par transition. Ainsi il faut enregistrer $\frac{ak^2}{v_0}$ actions, $\frac{ak^2}{v_0}$ intervalles de départ, $\frac{ak^2}{v_0}$ intervalles d'arrivée et $\frac{ak^2}{v_0}$ fonctions de répartition. Il faut donc $\frac{24ak^2}{v_0} + \frac{32ak^2}{v_0} + \frac{32ak^2}{v_0} + \frac{144ak^2}{v_0} = \frac{232ak^2}{v_0}$ octets pour enregistrer ces structures. Pour l'exemple en cours cela fait 2088 octets.

La version explicite de CISMO, *i.e.* qui n'emploie pas les MTBDD, utilise donc

$$4a + \frac{4ak}{v_1} + \frac{8ak}{v_1} + \frac{4ak^2}{v_1 v_2} + \frac{28ak^2}{v_0} + \frac{232ak^2}{v_0}$$

octets et pour l'exemple 2.6, enregistrer le LMP nécessite 2432 octets.

6.4.2 Analyse du mode avec MTBDD

Avant de calculer la mémoire utilisée par les structures propres au mode avec MTBDD, soulignons qu'un noeud dans un MTBDD construit avec le paquetage CUDD utilise 16 octets de mémoire selon la documentation de CUDD ; par contre, une autre source [56] affirme que ce serait plutôt 24 octets. De façon à avoir une approximation conservatrice, nous allons faire les calculs avec 24.

Il est nécessaire d'estimer le nombre de fonctions de probabilité différentes dans le modèle. Nous estimons ce nombre par le nombre de transitions en le réduisant d'un certain facteur puisque plusieurs transitions peuvent avoir les mêmes fonctions de répartition. Nous obtenons donc que $P \approx \frac{\text{nombre de transitions}}{v_3} = \frac{ak^2}{v_3 v_0}$. Pour l'exemple 2.6, il y a 9 transitions mais seulement 6 fonctions de répartition différentes. Ainsi, pour cet exemple, $v_3 = 1,5$.

Il est aussi nécessaire d'estimer le nombre de noeuds du MTBDD. Le MTBDD a au

plus $a^2 + 2ak^2 - a + P$ noeuds car :

- les premières variables du MTBDD sont les variables représentant les actions. Regardons le nombre de noeuds étiquetés par de telles variables. Comme il y a a actions et que chaque transition fait une seule action, il y a exactement une variable qui est *vraie* (prend la valeur 1) dans chaque chemin. Il y a au plus a chemins, ainsi, une approximation conservative nous donne a^2 noeuds étiquetés par une variable d'action.

- les variables qui suivent sont celles utilisées pour représenter les intervalles. Puisqu'il y a k intervalles, il faut $2log_2(k)$ variables pour les distinguer (départ et arrivée), la première variable apparaît au plus $2a$ fois (il y a deux arcs sortants de la dernière variable d'action) donc il y a au plus (si le MTBDD est complet) $a(2^1 + ... + 2^{2log_2(k)}) = a(2^{2log_2(k)+1} - 1) = 2ak^2 - a$ noeuds.

- il ne peut y avoir plus de feuilles que de fonctions de répartition, on trouve donc qu'il y a P feuilles. Comme nous avons dit précédemment, P peut être estimé par $\frac{ak^2}{v_3 v_0}$.

Calculons maintenant la mémoire utilisée par CISMO pour enregistrer un modèle à l'aide d'un MTBDD. Nous présentons ici les descriptions et l'analyse des structures de données qui sont utilisées dans le mode MTBDD et qui ne sont pas utilisées dans le mode sans MTBDD.

- La première structure est un *ArrayList* nommé *interInTrans*. Cette structure sert à enregistrer les intervalles (une seule fois chacun). Comme il y a k intervalles et que ceux-ci sont des références, cette structure utilise $k \times 4 = 4k$ octets.

  Pour l'exemple 2.6, la structure *interInTrans* contient les intervalles suivants :

  $$[0, 1), (1, 2], \{1\}, (2, 3], \{4\}, \{5\}$$

  ce qui nécessite 24 octets.

- Le deuxième objet à enregistrer est un entier nommé *nbInterval*. Cet entier est le nombre d'intervalles différents (de début et d'arrivée) présents dans le modèle. En JAVA, la taille d'un entier est généralement de 4 octets. Pour l'exemple 2.6, *nbInterval* est 6.

- L'objet suivant est un autre entier, il a comme nom *transitions*. Ce nombre est

un « pointeur[1] » sur le MTBDD des transitions. Ce pointeur utilise 4 octets.

Mesurons maintenant la mémoire utilisée par le MTBDD. Pour que le MTBDD soit complet il doit avoir $a^2 + 2ak^2 - a$ noeuds et il a P feuilles. Le MTBDD non complet utilise donc $24(\frac{a^2+2ak^2-a}{v_4} + \frac{ak^2}{v_3 v_0})$ octets (la variable v_4 rend le MTBDD non complet).

Le MTBDD qui représente le LMP de la figure 6.1 est celui de la figure 6.3. Il contient 32 noeuds. Ainsi, la mémoire utilisée par cette structure est de $32 \times 24 + 4 = 772$ octets. Selon nos calculs précédents, si le MTBDD était plein il contiendrait 80 noeuds (en admettant $k = 5$). Comme il n'en a que 32 noeuds, nous déduisons que $v_4 = 2,5$ pour cet exemple.

- La structure suivante est un *ArrayList*, nommé *fctRepartition*. Cette structure enregistre les fonctions de répartition (une seule fois chacune). Puisqu'il y a $\frac{ak^2}{v_3 v_0}$ fonctions de répartition, il faut $\frac{4ak^2}{v_3 v_0}$ octets pour enregistrer la structure *fctRepartition* (ce sont des références).

  Pour l'exemple 2.6, la structure *fctRepartition* contient les six fonctions de répartition suivantes :
  $$1,\ \frac{x+z}{4},\ \frac{z}{4},\ \frac{1-x}{4},\ \frac{xz}{4},\ \frac{1+z}{4}$$
  et la structure nécessite 24 octets.

- La dernière structure est un objet de la classe *Mtbdd* nommé *M*. Cet objet donne seulement accès aux fonctions de CUDD. L'objet M a une taille constante car ses seuls paramètres sont des fonctions. Mais sa taille est difficile à estimer. Nous dirons qu'il a une taille de x octets. Cependant, nous pouvons supposer que la taille d'un tel objet n'excède pas 100 octets puisque plusieurs structures du même genre n'en utilisent pas autant.

De plus, il faut compter les octets nécessaires pour enregistrer réellement les objets. Il faut donc $24a$ octets pour les actions, $32k$ octets pour les intervalles et $144\frac{ak^2}{v_3 v_0}$ octets pour les fonctions de répartition. Pour l'exemple en cours il faut donc 1104 octets pour enregistrer réellement ces objets.

Pour la nouvelle version de CISMO qui utilise les MTBDD, l'ensemble des structures

[1] Il n'y a pas de pointeur en JAVA, mais ce nombre est un pointeur lorsque le programme exécute une fonction en langage C par l'entremise de la JNI.

décrites précédemment utilise

$$4k + 8 + \frac{24a^2 + 2ak^2 - a}{v_4} + \frac{24ak^2}{v_3 v_0} + \frac{4ak^2}{v_3 v_0} + 100 + 24a + 32k + \frac{144ak^2}{v_0 v_3}$$

octets. Sur l'exemple de la figure 6.1, la méthode avec MTBDD utilise 2028 octets. Remarquons que la méthode avec MTBDD nécessite moins de mémoire que la méthode qui ne les utilise pas.

À l'aide du logiciel de mathématiques Maple, nous avons tracé le graphique de la différence de consommation de mémoire de ces deux méthodes (méthode sans MTBDD - méthode avec MTBDD). La figure 6.4 montre le résultat avec comme paramètre $v_0 = 2, v_1 = 2, v_2 = 2, v_3 = 1.2, v_4 = 2, x = 100$.

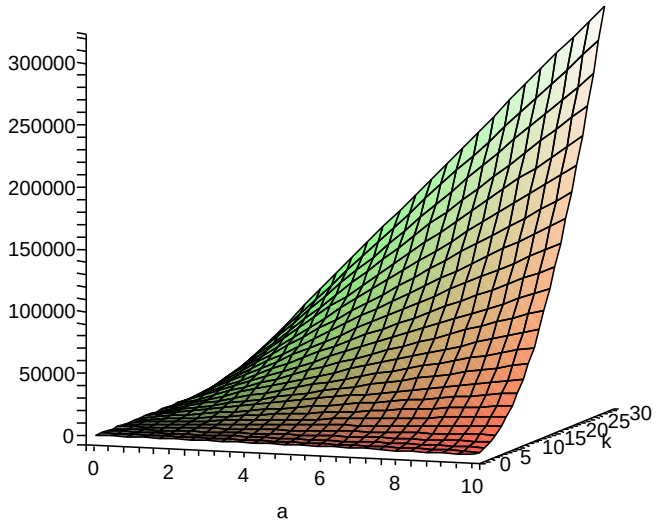


Fig. 6.4 – Fonction représentant la différence entre la mémoire utilisée sans les MTBDD et celle avec les MTBDD.

La figure 6.4 illustre que plus le nombre d'actions et le nombre d'intervalles croissent, plus la différence entre les deux fonctions croit elle aussi. De plus, pour d'autres valeurs de v_i, le graphe est semblable, même pour des grandes valeurs de v_0 et v_1 et des petites valeurs de v_2, v_3 et v_4. Nous pouvons donc conclure que l'emploi des MTBDD dans CISMO permet d'utiliser moins de mémoire que la technique sans MTBDD.

Il faut remarquer que nous affirmons que l'utilisation des MTBDD dans CISMO nous permet de sauver de la mémoire par rapport à la première version. Il est possible que d'autres implémentations sans MTBDD soient préférables à celle avec MTBDD. Cependant, les MTBDD sont souvent utilisés dans différents domaines et nous croyons qu'ils sont une bonne alternative pour améliorer les performances d'un programme (d'un point de vue de la mémoire).

6.5 Comparaison pratique de l'utilisation de la mémoire

S'il est difficile de comparer l'utilisation de la mémoire de façon théorique, il l'est encore plus en pratique. Trois problèmes majeurs nous empêchent de vérifier, avec confiance, nos résultats théoriques. Premièrement, nous n'avons pas de modèle suffisamment gros pour lesquels la première version de CISMO n'est pas capable de faire la vérification. Un tel modèle nous aurait permis de voir facilement si effectivement les MTBDD rendent possible la vérification de gros modèles. Ensuite, pour les modèles plus petits que nous possédons, il est probable que la différence ne soit pas notable. Finalement, nous avons un algorithme qui estime la taille d'un objet, alors pourquoi ne pas l'utiliser sur le modèle mis en mémoire ? Le problème est que pour avoir un bon degré de confiance, il faudrait mettre en mémoire le modèle des milliers de fois et ceci n'est pas possible même pour des petits modèles.

Nous n'avons donc pas été en mesure de vérifier si les résultats en pratique concordent avec nos résultats théoriques. Dans l'avenir, il serait intéressant de construire un très gros modèle pour tenter d'atteindre les limites de CISMO (sans les MTBDD) et voir comment il réagit avec les MTBDD.

Chapitre 7

Conclusion

Le but de la vérification est de s'assurer qu'un programme ou du matériel électrique fonctionnent comme on le souhaite. Certaines techniques de vérification permettent seulement de détecter des erreurs et d'autres, comme la vérification par évaluation de modèle, permettent de déterminer avec certitude si un certain type d'erreur peut ou non survenir.

Un des principaux problèmes en vérification est le problème d'explosion du nombre d'états. Ce problème est causé par une explosion combinatoire qui engendre un trop grand nombre d'états dans les modèles à vérifier. Ceci peut entraîner un manque de mémoire lors de la vérification. Une des solutions proposées pour ce problème est l'utilisation des OBDD pour représenter l'espace d'états du modèle ainsi que ses transitions. Cette méthode a été utilisée avec succès dans la vérification de systèmes déterministes et probabilistes à espace d'états discret.

Nous avons travaillé sur un vérificateur de modèles probabilistes nommé CISMO. Celui-ci vérifie un type de systèmes probabilistes qui ont un espace d'états continu : les LMP. Les résultats obtenus par l'utilisation des OBDD dans la vérification de d'autres types de modèles nous ont motivés à utiliser les OBDD pour vérifier des LMP. Nous avons ainsi modifié CISMO pour lui permettre de faire la vérification de façon symbolique en utilisant des MTBDD. Avant de les utiliser, nous avons approfondi nos connaissances sur les OBDD. Nous avons décrit plusieurs variantes des OBDD qui ont été développées pour améliorer les performances des OBDD dans des applications spécifiques.

Un problème connu dans le domaine des OBDD est la recherche d'un ordre des variables qui diminue au maximum la taille du OBDD. Trouver cet ordre est un problème

$\mathcal{NP}$-Difficile. Il y a peu d'information dans la littérature concernant la recherche sur l'ordre des variables d'un OBDD qui représente un graphe. La plupart des algorithmes mentionnés ont été faits spécialement pour les circuits électriques et nous avons expliqué à la section 6.3.2 pourquoi nous croyons qu'ils ne sont pas bien adaptés à nos applications.

Lors de notre étude des OBDD, nous avons remarqué qu'il ne fallait pas négliger la numérotation des états du graphe puisque celle-ci affecte la taille de l'OBDD qui le représente. À partir de cette observation, nous avons proposé des heuristiques pour diminuer la taille d'un OBDD en choisissant judicieusement la numérotation des états du graphe.

La modification que nous avons faite à CISMO permet maintenant d'utiliser ou non les MTBDD. De plus, il est possible de passer d'un mode à l'autre lors de la vérification. Notre comparaison des deux approches nous porte à croire que l'utilisation des MTBDD dans CISMO diminue la mémoire utilisée par le vérificateur. Rappelons toutefois que cette analyse est basée sur des estimations de la taille de certains objets JAVA : comme nous l'avons dit, il est difficile d'obtenir de bonnes estimations puisque la taille dépend de plusieurs paramètres, dont l'implémentation de la JVM. De plus, bien que nos résultats théoriques soient prometteurs, nous n'avons pas pu valider en pratique ces résultats puisque nous ne disposions pas de modèles suffisamment gros pour que la vérification sans MTBDD ne puisse se faire.

À la lumière de ces remarques, il y aurait encore des travaux à faire pour améliorer la vérification des LMP et les performances de CISMO, voici ceux qui pourraient être faits dans un avenir rapproché :

- construire des modèles de très grande taille pour mettre CISMO à l'épreuve,
- construire un générateur aléatoire de LMP et
- faire des études de cas plus approfondies.

Avec ces réalisations, nous serons plus en mesure de promouvoir et pousser plus loin la vérification de modèles probabilistes à espace d'états continus et par le fait même CISMO.

Bibliographie

[1] Spin. En ligne, page consultée le 16 mars 2005. http://spinroot.com/spin/whatispin.html#B.

[2] S. Akers. Binary decision diagrams. Dans *IEEE Transactions on Computers*, volume c-27. IEEE Computer Society Press, 1978.

[3] R. Alur, et G. J. Pappas, éditeurs. *Hybrid Systems : Computation and Control*, volume 2993 de *Lecture Notes in Computer Science*. Springer, mars 2004.

[4] A. Aziz, K. Sanwal, V. Singhal, et R. Brayton. Model-checking continuous-time markov chains. *ACM Trans. Comput. Logic*, 1(1) :162–170, 2000.

[5] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, et F. Somenzi. Algebraic decision diagrams and their applications. Dans *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 188–191. IEEE Computer Society Press, 1993.

[6] C. Baier, M. Größer, et F. Ciesinski. Partial order reduction for probabilistic systems. Dans *QEST*, pages 230–239, 2004.

[7] C. Baier, J.-P. Katoen, et H. Hermanns. Multi-terminal decision diagrams : a data structure for numerical integration (extended abstract), 1999.

[8] R. G. Bartel. *The Elements of Integration and Lebesgue Measure*. Wiley Classics Library, 1995.

[9] R. Blute, J. Desharnais, A. Edalat, et P. Panangaden. Bisimulation for labelled Markov processes. Dans *Proceedings of the Twelfth IEEE Symposium On Logic In Computer Science, Warsaw, Poland*, 1997.

[10] B. Bollig, et I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. Comput.*, 45(9) :993–1002, septembre 1996.

[11] J. Bowen. The world wide web virtual library : Formal methods. En ligne, page consultée le 16 mars 2005. http://vl.fmnet.info/.

[12] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.*, C-35(8) :677–691, août 1986.

[13] R. E. Bryant, et Y.-A. Chen. Verification of arithmetic circuits with binary moment diagrams. Dans *Proceedings of the 32nd ACM/IEEE conference on Design automation conference*, pages 535–541. ACM Press, 1995.

[14] E. M. Clarke, O. Grumberg, et D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5) :1512–1542, septembre 1994.

[15] B. Crothers. Intel confirms pentium "F0" bug. En ligne, page consultée le 16 mars 2005.
http://news.com.com/2100-1001-205207.html?legacy=cnet.

[16] J. Desharnais. *Labelled Markov Processes*. Thèse de doctorat, McGill University, novembre 1999.

[17] R. Drechsler, et B. Becker. *Binary Decision Diagrams : Theory and Implementation*. Kluwer Academic Publisher, 1998.

[18] Z. Feng, et E. A. Hansen. Approximate planning for factored POMDPs. Dans *Sixth European Conference on Planning (ECP-01)*, septembre 2001.

[19] S. Friedman, et K. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Trans. Comput.*, 39(5) :710–713, 1990.

[20] M. Fujita, P. McGeer, et J.-Y. Yang. Multi-terminal binary decision diagrams : An efficient data structure for matrix representation. *Formal Methods in System Design : An International Journal*, 10(2/3) :149–169, 1997.

[21] M. R. Garey, et D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[22] N. E. Gibbs, W. G. P. Jr., et P. K. Stockmeyer. Comparison of several bandwidth and profile reduction algorithms. *ACM Transactions on Mathematical Software*, 2(4) :322–330, 1976.

[23] J. Gleick. A bug and a crash. En ligne, page consultée le 16 mars 2005.
http://www.around.com/ariane.html.

[24] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems : An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., 1996.

[25] H. Hansson, et B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5) :512–535, 1994.

[26] V. Hartonas-Garmhausen, S. Campos, et E. Clarke. Probverus : Probabilistic symbolic model checking. Dans J.-P. Katoen, éditeur, *ARTS'99*, volume 1601 de *Lecture Notes in Computer Science*, pages 96–110. Springer-Verlag Berlin, 1999.

[27] M. Hennessy, et R. Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1) :137–161, 1985.

[28] H. Hermanns, M. Kwiatkowska, G. Norman, D. Parker, et M. Siegle. On the use of MTBDDs for performability analysis and verification of stochastic systems. *Journal of logic and algebraic programming*, 56(1-2) :23–67, août 2003.

[29] H. Hermanns, J. Meyer-Kayser, et M. Siegle. Multi Terminal Binary Decision Diagrams to Represent and Analyse Continuous Time Markov Chains. Dans B. Plateau, W. Stewart, et M. Silva, éditeurs, *3rd Int. Workshop on the Numerical Solution of Markov Chains*, pages 188–207. Prensas Universitarias de Zaragoza, 1999.

[30] M. Huth, et M. Ryan. *Logic in Computer Science : Modeling and Reasoning About Systems*. Cambridge University Press, 2000.

[31] J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, et D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4) :401–424, 1994.

[32] G. Kolata. Computer math proof shows reasoning power. En ligne, page consultée le 16 mars 2005.
http://www.nytimes.com/library/cyber/week/1210math.html.

[33] T. Kropf, et J. Ruf. Using MTBDDs for discrete timed symbolic model checking. Dans *Proceedings of the 1997 European conference on Design and Test*, page 182. IEEE Computer Society, 1997.

[34] M. Z. Kwiatkowska, G. Norman, D. Parker, et R. Segala. "Symbolic Model Checking of Probabilistic Processes Using MTBDDs and the Kronecker representation". Dans *Proceedings of TACAS 2000*, volume 1587 de *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[35] Y.-T. Lai, et S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. Dans *Proceedings of the 29th ACM/IEEE conference on Design automation conference*, pages 608–613. IEEE Computer Society Press, 1992.

[36] K. Larsen, et A. Skou. Bisimulation through probabilistic testing (preliminary report). Dans *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 344–352. ACM Press, 1989.

[37] C. Lee. Representation of switching circuits by binary-decision programs. Dans *Bell System Technical Journal*, volume 38, pages 985–999. Springer-Verlag, 1959.

[38] D. J. Lehmann, et S. Shelah. Reasoning with time and chance (extended abstract). Dans *Proceedings of the 10th Colloquium on Automata, Languages and Programming*, pages 445–457. Springer-Verlag, 1983.

[39] M. Fujita, E. Clarke, et X. Zhao. Applications of multi-terminal binary decision diagrams. Rapport technique CMU-CS-95-160, Pittsburgh, PA 15213, 1995.

[40] K. L. McMillan. Model checking @cmu. En ligne, page consultée le 16 mars 2005.
http://www-2.cs.cmu.edu/~modelcheck/smv.html.

[41] C. Meinel, et T. Theobald. *Algorithms and Data Structures in VLSI Design*. Springer-Verlag, 1998.

[42] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. Dans *Proceedings of the 30th international on Design automation conference*, pages 272–277. ACM Press, 1993.

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

[44] A. Narayan, J. Jain, M. Fujita, et A. Sangiovanni-Vincentelli. 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*, pages 547–554. IEEE Computer Society, 1996.

[45] M. Nobilitas. Martin's java notes. En ligne, page consultée le 16 mars 2005. http://martin.nobilitas.com/java/sizeof.html.

[46] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. Thèse de doctorat, University of Birmingham, 2002.

[47] M. L. Puterman. *Markov Decision Processes : Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., 1994.

[48] N. Richard. La vérification formelle de systèmes probabilistes continus. Mémoire de maîtrise, Université Laval, septembre 2003.

[49] Y. Saad. *Iterative methods for sparse linear systems*. PWS Publishing Company, 1st edition, 1996. http://www-users.cs.umn.edu/~saad/books.html.

[50] T. Sasao, éditeur. *Representations of Discrete Functions*. Springer-Verlag, 1996.

[51] M. Siegle. Compositional representation and reduction of stochastic labelled transition systems based on decision node BDDs. Dans *Messung, Modellierung und Bewertung von*, pages 173–185, 1999.

[52] D. Sieling. The nonapproximability of OBDD minimization. *Inf. Comput.*, 172(2) :103–138, 2002.

[53] S. S. Skiena. Bandwidth reduction. En ligne, page consultée le 16 mars 2005. http://www.cs.sunysb.edu/~algorith/files/bandwidth.shtml.

[54] F. Somenzi. CUDD. En ligne, page consultée le 16 mars 2005. http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html.

[55] G. Sutcliffe. Automated theorem proving. En ligne, page consultée le 16 mars 2005. http://www.cs.miami.edu/~tptp/OverviewOfATP.html.

[56] A. Vahidi. JBDD, a java interface to CUDD and BuDDY. En ligne, page consultée le 16 mars 2005. http://javaddlib.sourceforge.net/jbdd/index.html.

[57] A. Valmari. A stubborn attack on state explosion. Dans *CAV '90 : Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 156–165. Springer-Verlag, 1991.

[58] S. Wilson, et J. Kesselman. Javatm platform performance, strategies and tactics. En ligne, page consultée le 16 mars 2005. java.sun.com/docs/books/performance/1st_edition/html/JPTitle.fm.html.

Annexe A

Rappels de la théorie de la mesure

Définition A.0.1 *Une σ-algèbre sur un ensemble X est un ensemble $\Sigma \subseteq \mathcal{P}(X)$ tel que*

1. $X \in \Sigma$
2. $\forall A \in \Sigma,\ A^c \in \Sigma$
3. $\forall (A_i)_{i \in \mathbf{N}} \in \Sigma,\ \bigcup_{i \in \mathbf{N}} A_i \in \Sigma.$

On note ici A^c le complément de l'ensemble A par rapport à l'ensemble X, c'est-à-dire $A^c = X \setminus A$. Les éléments de Σ sont appelés des ensembles mesurables et la paire (X, Σ) est nommée espace mesurable. Par exemple, la plus petite σ-algèbre sur X, pour n'importe quel ensemble X, est $\Sigma = \{\emptyset, X\}$. Si X est munie d'une topologie, on nommera la σ-algèbre engendrée par les ouverts la σ-algèbre de Borel et on la notera $B(X)$.

Définition A.0.2 *Une mesure sur un espace mesurable (X, Σ) est une application $\mu : \Sigma \to [0, 1]$ telle que*

1. $\mu(\emptyset) = 0$
2. $\mu(\cup A_i) = \sum_i \mu(A_i)$*, où $\cup A_i$ est une union dénombrable d'ensembles disjoints.*

Par exemple, la mesure de Lebesgue est l'unique mesure μ sur $(\mathbb{R}, B(\mathbb{R}))$ telle que $\mu([a, b]) = b - a$. C'est la mesure habituelle que l'on utilise pour la longueur d'un

intervalle. On remarque que la mesure de Lebesgue d'un singleton est nulle et qu'ainsi, par les propriétés des mesures, la mesure de Lebesgue d'un ensemble dénombrable est aussi nulle.

Définition A.0.3 *Une mesure de sous-probabilité sur un espace mesurable* (X, Σ) *est une mesure telle que* $0 \leq \mu(X) \leq 1$. *Le triplet* (X, Σ, μ) *est nommé un espace mesuré.*

Annexe B

Code pour estimer la taille d'un objet JAVA

Nous présentons ici le code pour estimer la taille d'un objet de la classe *String*. L'objet en question est la chaîne de caractères « `a` ». Pour estimer la taille d'un objet d'une autre classe, il suffit d'implémenter la classe *ClasseObjetFactory* pour cette classe. Le code pour évaluer la mémoire est emprunté à la compagnie *Sun* [58] et l'utilisation de l'interface *ObjectFactory* est inspirée de Nobilatas [45].

```
import java.util.*;

public class SizeOf
{
    public static void main(String[] argv)
    {
        System.out.println("SizeOf: (bytes)");
        reportSize(new StringFactory());
    }


    private static void reportSize(ObjectFactory f)
    {
        System.out.println(f.getClassName() + ": " + estimateSize(f));
    }
```

```
public static long estimateSize(ObjectFactory f)
{
    final int n = f.makeHowMany();
    Object[] array = new Object[n];
    Object o = f.makeObject(0);

    long before = getUsedMemory();

    for (int i = n - 1; i >= 0; i--)
    {
        array[i] = f.makeObject(i);
    }

    long after = getUsedMemory();

    return Math.round((double) (after - before) / (double) n);
}


private static void gc()
{
    try
    {
        System.gc();
        Thread.currentThread().sleep(100);
        System.runFinalization();
        Thread.currentThread().sleep(100);
        System.gc();
        Thread.currentThread().sleep(100);
        System.runFinalization();
        Thread.currentThread().sleep(100);
    }catch (Exception e)
    {
        e.printStackTrace();
    }
}


private static long getUsedMemory()
{
```

```
        gc();
        long totalMemory = Runtime.getRuntime().totalMemory();
        gc();
        long freeMemory = Runtime.getRuntime().freeMemory();
        long usedMemory = totalMemory - freeMemory;
        return usedMemory;
   }

}//SizeOf
```

```
/**
 * An interface for a factory that makes objects to be measured
 * by SizeOf.
 */
public interface ObjectFactory
{
    /**
    * Return the name of the class being tested.
    */

    public String getClassName();

    /**
    * Return a new instance of the class being tested.
    */
    public Object makeObject(int i);

    /**
    * Return a number of object that is large
    */
    public int makeHowMany();

}//ObjectFactory
```

```
import java.lang.*;

public  class  StringFactory implements ObjectFactory
{
    public String getClassName()
    {
        return "java.lang.String";
    }

    public Object makeObject(int i)
    {
        return new String("a");

    }

    public int makeHowMany()
    {
        return  100000;
    }

}//StringFactory
```