Compilation de CSP en Set-labeled Diagram

Alexandre Niveau * **Hélène Fargier** † **Cédric Pralet** ‡

* CRIL, Lens
† IRIT/RPDMP, Toulouse
‡ ONERA/DCSD, Toulouse
niveau@cril.fr fargier@irit.fr cpralet@onera.fr

Résumé

La plupart des requêtes associées aux CSPs sont NP-difficiles, et doivent pourtant parfois être exécutées en ligne et en temps limité. Dans ce cas, la résolution du CSP n'est pas assez efficace, voire impossible. Des structures ont été proposées, tels les MDDs, pour *compiler* les CSPs et rendre leur exploitation en ligne efficace. Les MDDs sont des DAGs orientés dont chaque nœud représente l'assignation d'une variable ; l'ensemble des solutions d'un CSP correspond à l'ensemble des chemins du MDD correspondant. Dans cet article, nous étudions la relaxation de deux restrictions usuellement imposées aux MDDs, l'ordonnancement et la non-répétition des variables, introduisant une structure de compilation nommée « set-labeled diagrams » (SDs). Un CSP peut être compilé en SD en suivant la trace de l'arbre de recherche exploré par un solveur de CSP. L'impact des restrictions susnommées peut être étudié en faisant varier les heuristiques utilisées par le solveur lors de la résolution.

1 Introduction

Les problèmes de satisfaction de contraintes (CSPs) fournissent un cadre puissant permettant de représenter une grande variété de problèmes, tels des problèmes de planification ou de configuration. On peut considérer divers types de requêtes sur un CSP, comme l'extraction d'une solution (la requête la plus classique), la cohérence forte des domaines, l'ajout ou le retrait de nouvelles contraintes (CSP dynamique), voire une combinaison de plusieurs de ces requêtes. Par exemple, la résolution interactive d'un problème de configuration revient à ajouter et retirer des contraintes (unaires) tout en maintenant la cohérence forte des domaines — c'est-à-dire que chaque valeur d'un domaine doit faire partie d'au moins une solution).

La plupart des requêtes sont NP-difficiles ; elles doivent cependant être parfois effectuées en ligne. Une façon possible de résoudre cette contradiction consiste à compiler l'ensemble des solutions d'un CSP en un *diagramme de décision multivalué* (MDD, *multivalued decision diagram*), c'est-à-dire un graphe dont les nœuds sont étiquetés par des variables et les arcs représentent des valeurs assignées à ces variables. Dans de tels diagrammes, chaque chemin de la racine au puits représente une solution du CSP. Un certain nombre d'opérations, notamment les requêtes précédemment citées, sont faisables sur un MDD en temps polynomial en sa taille. Théoriquement, cette taille peut être exponentiellement plus grande que celle du CSP original, mais en pratique elle reste raisonnable dans un certain nombre d'applications. En effet, leur nature de graphe permet aux MDDs de tirer parti de l'interchangeabilité des valeurs, en gagnant de la place grâce à la fusion de sous-problèmes équivalents. Les diagrammes de décision ont été utilisés dans divers contextes, notamment la configuration de produit [2], les systèmes de recommandation [8], ou encore, dans leur forme booléenne originale, dans la planification [10, 13] et le diagnostic [24].

À notre connaissance, ces travaux ne considèrent que des graphes *ordonnés*, c'est-à-dire que l'ordre dans lequel les variables sont rencontrées le long d'un chemin est fixé (x ne peut apparaître avant y dans un chemin, et après y dans un autre), et « *read-once* », c'est-à-dire que les variables ne peuvent être répétées le long d'un chemin. Ce n'est cependant pas une nécessité pour bien des applications ; dans le cas booléen, [5] ont montré que les OBDDs pouvaient être avantageusement remplacés par des FBDDs (*free BDDs*), qui sont « *read-once* » mais pas ordonnés. Dans cet article, nous utilisons le langage des « diagrammes à étiquettes ensemblistes » (SDs, *set-labeled diagrams*) [19], qui généralise les MDDs en relâchant ces deux

contraintes à la fois.

Il est important de noter que nous ne nous intéressons pas à l'utilisation de MDDs pour représenter une contrainte à la fois, avec l'objectif d'utiliser les formes compilées dans un solveur de CSP. Dans ce cas, les requêtes seraient principalement de la propagation. Notre objectif est différent : nous voulons compiler l'ensemble des solutions d'un CSP complet, de manière à pouvoir exécuter en ligne diverses requêtes sur la forme compilée.

Dans cet article, nous étudions une méthode de compilation en SDs, qui consiste à appliquer le concept de « DPLL *with a trace* » [14] à un solveur de CSP. Nous présentons un compilateur générique, permettant de construire des SDs tout en bénéficiant des techniques de programmation par contraintes disponibles dans les solveurs. Avec ce compilateur, relâcher les hypothèses d'ordonnancement statique ou de « *read-once* » dépend des heuristiques de branchement et de choix de variables utilisées pendant la recherche par le solveur. Nous étudions diverses heuristiques possibles et présentons des résultats expérimentaux, obtenus grâce à l'implantation de notre algorithme sur le solveur Choco [9].

Le papier est organisé comme suit : nous présentons tout d'abord dans la section 2 le cadre formel des SDs, qui généralisent les MDDs. Puis, dans la section 3, nous décrivons notre algorithme de compilation. La section 4 détaille les heuristiques que nous avons utilisées, et les résultats de nos expérimentations.

2 Set-labeled Diagrams

2.1 Structure et sémantique

Nous donnons tout d'abord une définition générale des *set-labeled diagrams*, puis nous nous restreignons à un cadre spécifique.

Définition 1 (Set-labeled diagram)**.** Soit $\mathcal{V}$ un ensemble de variables et $\mathcal{E}$ un ensemble d'ensembles. Un *diagramme à étiquettes ensemblistes* (*set-labeled diagram*, SD) est un graphe acyclique orienté ayant au plus une racine et au plus une feuille (appelée le *puits*). Chaque nœud (excepté le puits) est étiqueté par une variable de $\mathcal{V}$ ou par le symbole disjonctif $\veebar$. Chaque arc est étiqueté par un ensemble, élément de $\mathcal{E}$.

Cette définition implique peu de restrictions sur les étiquettes des nœuds et des arcs. Les SDs généralisent ainsi un certain nombre de structures représentant des ensembles de solutions, telles les BDDs [7] (*binary decision diagrams*, en prenant des variables booléennes et $\mathcal{E} = \{\bot, \top\}$), les MDDs [22, 25, 3, 4] (*multivalued decision diagrams*, en prenant des variables discrètes et $\mathcal{E}$ un ensemble de singletons), et les automates à intervalles [20] et *interval diagrams* [23] (en prenant $\mathcal{E}$ un ensemble d'intervalles).

Dans la suite, nous nous restreignons à un cadre proche de celui des MDDs : les domaines des variables sont des parties finies de $\mathbb{Z}$. En revanche, contrairement aux MDDs, les arcs ne sont pas étiquetés par des singletons mais par des ensembles finis de $\mathbb{Z}$. Dans cet article, nous considérerons des ensembles explicitement énumérés — utiliser des ensembles ne fait pas gagner d'espace dans ce cas (l'intérêt est de pouvoir définir une nouvelle restriction structurelle, la convergence). Néanmoins, les SDs sont plus généraux que les MDDs, notamment car l'ordre des variables et leur répétition le long d'un chemin ne sont pas contraints, mais aussi car ils peuvent être *non-déterministes* : les ensembles étiquetant les arcs sortant d'un même nœud ne sont pas forcément deux à deux disjoints. Les SDs déterministes sont appelés dSDs.

Notons qu'un SD peut être vide (n'avoir aucun nœud) ou ne contenir qu'un seul nœud (qui est alors à la fois racine et puits). La figure 1 donne un exemple de SD.

Nous utilisons les notations suivantes : le domaine d'un variable $x \in \mathcal{V}$ est noté $\mathrm{Dom}(x)$. Par convention, on note $\mathrm{Dom}(\veebar) = \{0\}$. On suppose que $\mathcal{V}$ est totalement ordonné, et dans un ensemble noté $X = \{x_1, \ldots, x_k\} \subseteq \mathcal{V}$, les variables soient ordonnés dans l'ordre croissant de leur indice. On note alors $\mathrm{Dom}(X) = \mathrm{Dom}(x_1) \times \cdots \times \mathrm{Dom}(x_k)$, et $\vec{x}$ représente une X-instanciation des variables de X, c'est-à-dire que $\vec{x} \in \mathrm{Dom}(X)$. Enfin, la valeur assignée à x_i dans l'instanciation $\vec{x}$ est notée $\vec{x}_{|x_i}$ (par convention, $\vec{x}_{|\veebar} = 0$ pour tout X). Le cardinal d'un ensemble S est noté $|S|$.

Soit φ un SD, N un nœud et E un arc de φ. On note :

- $\mathrm{Var}(\varphi)$ l'ensemble des variables mentionnées dans φ ;
- $\mathrm{Root}(\varphi)$ la racine et $\mathrm{Sink}(\varphi)$ le puits de φ ;
- $\|\varphi\|$ la *taille* de φ, c'est-à-dire la somme des cardinalités des ensembles et des domaines des variables mentionnés dans φ ;
- $\mathrm{Out}(N)$ (resp. $\mathrm{In}(N)$) l'ensemble des arcs sortants (resp. entrants) du nœud N ;
- $\mathrm{Var}(N)$ la variable étiquetant N (par convention $\mathrm{Var}(\mathrm{Sink}(\varphi)) = \veebar$) ;
- $\mathrm{Src}(E)$ le nœud duquel sort l'arc E et $\mathrm{Dest}(E)$ le nœud dans lequel il entre ;
- $\mathrm{Lbl}(E)$ l'ensemble étiquetant l'arc E ;
- $\mathrm{Var}(E) = \mathrm{Var}(\mathrm{Src}(E))$ la variable relative à E.

Un SD est une représentation compacte d'une fonction booléenne sur des variables discrètes. Cette fonc-

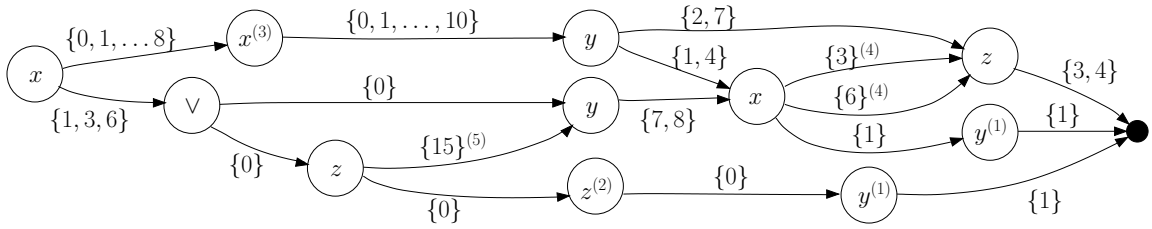

FIGURE 1 – Un exemple de SD. Les variables ont toutes pour domaine l'ensemble $\{0, 1, \ldots, 10\}$. Ce SD n'est pas réduit (voir section 2.2) : les deux nœuds marqués $^{(1)}$ sont isomorphes ; le nœud $^{(2)}$ est bégayant ; le nœud $^{(3)}$ est non-décisif ; les arcs marqués $^{(4)}$ sont contigus ; l'arc $^{(5)}$ est mort.

tion est appelée *interprétation* du SD.

Définition 2 (Sémantique d'un SD)**.** Soit φ un SD, et $X = \mathrm{Var}(\varphi)$. L'*interprétation* de φ est la fonction $[\![\varphi]\!]$, de $\mathrm{Dom}(X)$ vers $\{\top, \bot\}$, et définie comme suit : pour toute X-instanciation $\vec{x}$, $[\![\varphi]\!](\vec{x}) = \top$ si et seulement s'il existe un chemin p de la racine au puits de φ, tel que pour tout arc E de p, $\vec{x}_{|\,\mathrm{Var}(E)} \in \mathrm{Lbl}(E)$.

Une X-instanciation $\vec{x}$ est *modèle* de φ si et seulement si elle vérifie $[\![\varphi]\!](\vec{x}) = \top$. On note $\mathrm{Mod}(\varphi)$ l'ensemble des modèles de φ.

φ est *équivalent* à un autre SD ψ (ce que l'on note $\varphi \equiv \psi$) ssi $\mathrm{Mod}(\varphi) = \mathrm{Mod}(\psi)$.

L'interprétation du SD vide renvoie toujours $\bot$, puisqu'il ne contient aucun chemin de la racine au puits. Inversement, l'interprétation du SD-feuille renvoie toujours $\top$; en effet, le seul chemin de la racine au puits de ce SD ne contient aucun arc.

Définition 3 (Cohérence, validité, contexte)**.** Soit φ un SD, et $X = \mathrm{Var}(\varphi)$.

φ est dit *cohérent* (resp. *valide*) si et seulement si $\mathrm{Mod}(\varphi) \neq \varnothing$ (resp. $\mathrm{Mod}(\varphi) = \mathrm{Dom}(X)$).

Un entier $\omega \in \mathbb{Z}$ est dit *cohérent* pour une variable y dans φ si et seulement s'il existe un modèle $\vec{x}$ de φ tel que $\vec{x}_{|y} = \omega$.

L'ensemble des valeurs cohérentes de y dans φ est appelé *contexte* de y dans φ, et noté $\mathrm{Ctxt}_\varphi(y)$.

Contrairement au cas des MDDs, décider si un SD est cohérent ou non est un problème difficile. Une des raisons est que les différents ensembles qui restreignent le domaine d'une même variable le long d'un chemin peuvent être disjoints, ce qui implique que ledit chemin ne correspond à aucun modèle. Il faudrait donc potentiellement parcourir tous les chemins d'un SD pour décider de sa cohérence. Pour éviter cela, une possibilité est de considérer des SDs dont les ensembles relatifs à une même variable ne peuvent que rétrécir le long d'un chemin. Nous appelons cette propriété *convergence* ; elle est illustrée figure 2. La convergence généralise la propriété de « *read-once* » définie dans le contexte des « free BDDs » ; en effet, si aucune variable n'est répétée, le SD est trivialement convergent.

Définition 4 (SD convergent et *read-once*)**.** Un arc E d'un SD φ est *convergent* si et seulement si tout arc E' sur un chemin de la racine de φ à $\mathrm{Src}(E)$ tel que $\mathrm{Var}(E) = \mathrm{Var}(E')$ vérifie $\mathrm{Lbl}(E) \subseteq \mathrm{Lbl}(E')$.

Un SD *convergent* (*focusing SD*, FSD) est un SD ne contenant que des arcs convergents.

Un SD *read-once* (RSD) est un SD dans lequel il n'existe aucun chemin contenant deux nœuds étiquetés par une même variable.

On note dFSD (resp. dRSD) un SD à la fois déterministe et convergent (resp. *read-once*). Enfin, il est possible d'imposer un ordre sur les variables, et retrouver les MDDs dans leur définition usuelle [1] [22, 25, 3, 4].

Définition 5 (SD ordonné)**.** Soit X un ensemble de variables et $<$ un ordre total sur X. Un SD est *ordonné selon* $<$ si et seulement si tout couple de nœuds (N, M) tel que N est ancêtre de M vérifie $\mathrm{Var}(N) < \mathrm{Var}(M)$.

Un SD déterministe et ordonné selon $<$ est appelé un $\mathrm{MDD}_<$. Le langage [2] `MDD` est l'union des langages $\mathtt{MDD}_<$ pour tout $<$.

Nous montrons à présent comment réduire un SD, de façon à le rendre aussi compact que possible, et gagner de l'espace mémoire.

2.2 Réduction

Comme un BDD, un SD peut être réduit en taille sans que sa sémantique ne change. La réduction est basée sur plusieurs opérations ; certaines sont des généralisations directes de celles introduites dans le contexte des BDDs [7], telles la fusion des nœuds isomorphes

1. La définition originale des MDDs n'exige ni déterminisme, ni ordonnancement. Néanmoins, les travaux utilisant cette structure se restreignant systématiquement à des graphes ordonnés et déterministes, nous décidons arbitrairement de nommer « MDDs » les SDs ordonnés et déterministes.

2. Un *langage* est un ensemble de graphes muni d'une fonction d'interprétation. On note `SD` le langage des SDs, `FSD` le langage des FSDs, etc.

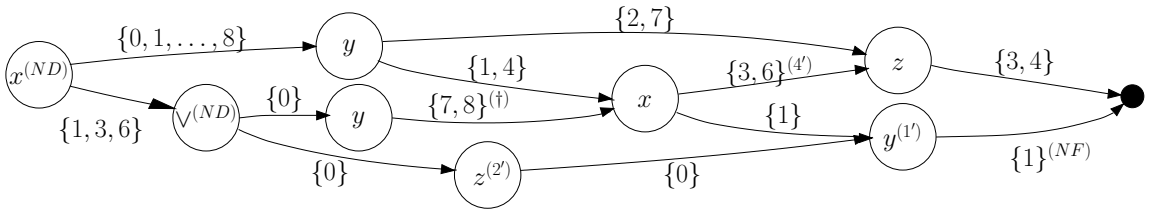

FIGURE 2 – Dans ce SD, tous les arcs sont convergents, excepté celui marqué $^{(NF)}$ (son étiquette n'est pas incluse dans celle de l'arc marqué $^{(\dagger)}$), et tous les nœuds sont déterministes excepté ceux marqués $^{(ND)}$. Ce SD est la forme réduite de celui présenté figure 1 : les nœuds isomorphes, marqués $^{(1)}$, ont été fusionnés en un seul nœud $^{(1')}$; le nœud bégayant $^{(2)}$ a fusionné avec le nœud $^{(2')}$; les arcs contigus, marqués $^{(4)}$, ont fusionné en un seul arc $^{(4')}$; et le nœud non-décisif $^{(3)}$ et l'arc mort $^{(5)}$ ont été supprimés.

(qui sont équivalents) et non-décisifs (qui sont automatiquement franchis), tandis que d'autres sont spécifiques aux SDs, comme la suppression des arcs morts (qui ne sont jamais franchis), et la fusion des arcs contigus (qui ont la même source et la même destination) et des nœuds bégayants (des décisions successives relatives à une même variable). Toutes ces notions sont illustrées sur le SD de la figure 1, et la forme réduite de ce SD est présentée figure 2.

Définition 6.

- Deux arcs E_1 et E_2 sont *contigus* ssi $\mathrm{Src}(E_1) = \mathrm{Src}(E_2)$ et $\mathrm{Dest}(E_1) = \mathrm{Dest}(E_2)$.
- Deux nœuds N_1 et N_2 sont *isomorphes* ssi $\mathrm{Var}(N_1) = \mathrm{Var}(N_2)$ et il existe une bijection σ de $\mathrm{Out}(N_1)$ vers $\mathrm{Out}(N_2)$ vérifiant $\forall E \in \mathrm{Out}(N_1), \mathrm{Lbl}(E) = \mathrm{Lbl}(\sigma(E))$ et $\mathrm{Dest}(E) = \mathrm{Dest}(\sigma(E))$.
- Un arc E est *mort* ssi $\mathrm{Lbl}(E) \cap \mathrm{Dom}(\mathrm{Var}(E)) = \varnothing$.
- Un nœud N est *non-décisif* ssi $|\mathrm{Out}(N)| = 1$ et $E \in \mathrm{Out}(N)$ vérifie $\mathrm{Dom}(\mathrm{Var}(E)) \subseteq \mathrm{Lbl}(E)$.
- Un nœud non-racine N est *bégayant* ssi tous ses parents sont étiquetés par $\mathrm{Var}(N)$, et soit $\sum_{E \in \mathrm{Out}(N)} |E| = 1$, soit $\sum_{E \in \mathrm{In}(N)} |E| = 1$.

Définition 7 (Forme réduite)**.** Un SD φ est dit *réduit* ssi aucun nœud de φ n'est isomorphe à un autre, bégayant ou non-décisif, et aucun arc de φ n'est mort ou contigu à un autre.

La réduction peut être effectuée en temps polynomial en la taille du graphe.

Proposition 8 (Réduction)**.** *Soit* L *un sous-langage de* SD *parmi* $\{\mathtt{SD}, \mathtt{FSD}, \mathtt{dSD}, \mathtt{dFSD}, \mathtt{MDD}, \mathtt{MDD}_<\}$*. Il existe un algorithme polynomial qui associe tout* φ *de* L *à un SD réduit* φ' *de* L *équivalent à* φ *et vérifiant* $|\varphi'| \leq |\varphi|$*.*

3 Algorithme de compilation : « Choco with a Trace »

3.1 État de l'art

La compilation de connaissances est un domaine principalement étudié du point de vue théorique. Quelques compilateurs ont été implantés, principalement dans le cadre booléen — ils permettent de compiler des fonctions à valeur booléennes sur des variables booléennes — notamment les paquetages pour manipuler des OBDDs (tels Buddy [17] et CUDD [21]) et le plus récent « DPLL with a trace » proposé par Huang et Darwiche [14]. La première catégorie de paquetages compile des formules élémentaires (ou des contraintes élémentaires) en diagramme de décision (binaire), et combine incrémentalement les résultats obtenus en utilisant des opérations de conjonction (approche *bottom-up*). Il est toujours possible d'utiliser le cadre booléen pour représenter des domaines multivalués, en utilisant un encodage booléens des domaines du CSP [15] ; néanmoins, il a été montré expérimentalement [11] que les MDDs sont souvent plus compacts que les BDDs pour des problèmes réels (notamment de configuration de produit).

L'inconvénient de l'approche *bottom-up* est qu'elle peut générer des graphes intermédiaires, qui utilisent de l'espace et peuvent être exponentiellement plus grands que le graphe final. L'algorithme « DPLL with a trace » [14] utilise une approche différente pour compiler des formules propositionnelles, qui s'avère efficace en pratique : celle de construire des diagrammes de décision (ainsi que des d-DNNFs) en exploitant l'arbre de recherche d'une exécution de l'algorithme DPLL, qui énumère toutes les solutions d'une CNF.

Cette idée a été adaptée [12] pour construire des MDDs approchés (dont l'ensemble de modèles est une approximation de l'ensemble de solutions du CSP d'entrée) en exploitant la trace d'un algorithme en profondeur d'abord. Une technique similaire est utilisée [18] pour construire des AOMDDs (MDDs avec des nœuds

ET) à partir de la trace d'une recherche AND/OR.

Toutes les approches précédentes se basent sur un ordre des variables prédéterminé (dans le cas des AOMDDs, c'est un ordre arborescent). Nous relâchons ici cette hypothèse : le choix de la prochaine variable sur laquelle brancher peut être *dynamique*, en fonction d'une heuristique. Nous présentons ici une description générale de notre algorithme, que nous avons implanté au-dessus du solveur de CSP Choco [9].

3.2 Description générale

Soit $\mathcal{C} = \langle X, C \rangle$ un CSP défini par un ensemble de contraintes C sur un ensemble de variables X.

En étendant les principes de « DPLL with a trace » des domaines booléens aux domaines entiers, nous introduisons des mécanismes permettant de construire des dFSDs en exploitant l'arbre de recherche d'un solveur de CSP. Cette approche est présentée dans l'algorithme 1. Implanté au-dessus du solveur Choco, nous obtenons le compilateur « *Choco with a trace* ». La fonction principale, `SD_builder`$(\mathcal{C}, X_a)$ prend en entrée un CSP $\mathcal{C}$ et l'ensemble courant X_a des variables instanciées dans $\mathcal{C}$. Elle retourne une représentation compilée de l'ensemble des solutions de $\mathcal{C}$ sous forme de dFSD. L'appel initial du compilateur est `SD_builder`$(\mathcal{C}_0, \varnothing)$, avec $\mathcal{C}_0$ le CSP initial que l'on veut compiler.

Algorithm 1 `SD_builder`$(\mathcal{C}, X_a)$: renvoie un SD représentant l'ensemble des solutions du CSP $\mathcal{C}$. X_a est l'ensemble courant des variables assignées.

```
1:  Propagate(C)
2:  k := Compute_key(C, X_a)
3:  if le cache contient une entrée pour la clé k then
4:      return le SD correspondant à k dans le cache
5:  if C est prouvé incohérent then
6:      return le SD vide
7:  if X_a = X (toutes les variables de C sont assignées) then
8:      return le SD-feuille
9:  Ψ := ∅
10: x := Choose_var(C)
11: R := Split(C, x)
12: for all r ∈ R do
13:     X'_a := X_a
14:     if r est réduit à un singleton then
15:         X'_a := X'_a ∪ {x}
16:     soit ψ_r := SD_builder(C_| Dom(x)←r, X'_a)
17:     Ψ := Ψ ∪ {ψ_r}
18: soit N le nœud N := Get_node(x, Ψ)
19: soit φ le graphe de racine N
20: ranger φ à la clé k dans le cache
21: return φ
```

3.2.1 Recherche standard

La partie classique de la fonction `SD_builder`$(\mathcal{C}, X_a)$ est une recherche générique en profondeur d'abord, énumérant l'ensemble des solutions du CSP $\mathcal{C}$. Elle fonctionne comme suit : tout d'abord, la fonction `Propagate` applique des techniques de propagation au CSP d'entrée, de manière à retirer des domaines certaines valeurs incohérentes ; si le CSP obtenu est localement cohérent (aucun domaine n'est vide), alors (i) la fonction `Choose_var` sélectionne une variable x non encore assignée ($x \notin X_a$), (ii) la fonction `Split` partitionne le domaine courant de x en plusieurs sous-ensembles disjoints, et (iii) la fonction principale est appelée successivement sur chacun des sous-problèmes induits par la partition. À ce niveau, l'implantation des fonctions `Propagate`, `Choose_var` and `Split` n'a pas d'importance ; n'importe quelles techniques et heuristiques peuvent être utilisées.

3.2.2 Ajouts pour la compilation

Pour construire un dFSD représentant l'ensemble des solutions du CSP initial, certains ajouts sont nécessaires ; ils sont encadrés dans l'algorithme 1. L'idée de base est de construire une trace de l'arbre de recherche par le biais de divers mécanismes :

- Soit n le nœud courant de l'arbre de recherche ; on note $\mathcal{C}(n)$ le CSP courant associé à ce nœud. Soit x la variable non-assignée choisie au nœud n, et R la partition de $\mathrm{Dom}(x)$ sur laquelle l'algorithme a décidé de brancher (une branche par élément dans la partition). L'idée est que l'exploration associée à chaque sous-domaine $r \in R$ génère un SD ψ_r ; ce SD représente l'ensemble des solutions du sous-problème $\mathcal{C}(n)_{|\,\mathrm{Dom}(x)\leftarrow r}$, obtenu en réduisant le domaine de x à r. Alors, l'ensemble des solutions du CSP $\mathcal{C}(n)$ sur $X \setminus X_a$ est un SD $\varphi(n)$ dont la racine est étiquetée par x et qui contient, pour chaque $r \in R$ tel que ψ_r n'est pas le SD vide, un arc de la racine à ψ_r.
  Dans la fonction `SD_builder`$(\mathcal{C}, X_a)$, le graphe $\varphi(n)$ est obtenu par l'appel `Get_node`$(x, \{\, \psi_r \mid r \in R \,\})$. En particulier, la fonction `Get_node` vérifie si le graphe $\varphi(n)$ existe déjà dans la *table de nœuds uniques*, qui contient tous les nœuds de SD créés au cours de la recherche. Si le nœud demandé est isomorphe à un nœud contenu dans la table, le nœud déjà existant est retourné ; sinon, le nœud est créé et

ajouté à la table.
- Si n est une feuille de l'arbre de recherche, elle correspond soit à une solution soit à un cul-de-sac. Dans le premier cas, l'algorithme renvoie le SD-feuille (l. 8) : toute instanciation est solution du problème courant. Dans le deuxième cas, il retourne le SD vide (l. 6).
- Les deux points précédents suffisent à compiler un dFSD représentant l'ensemble des solutions du CSP initial. Pour diverses raisons expliquées dans la suite, on maintient lors de la recherche un *cache*, permettant d'éviter à des sous-problèmes équivalents d'être ré-explorés. Plus précisément, chaque fois qu'un sous-problème $\mathcal{C}$ est résolu, une clé $k(\mathcal{C})$ est générée, qui dépend des domaines courants des variables. Cette clé est enregistrée avec le SD représentant ce sous-problème (l. 20). Par la suite, avant de résoudre un nouveau sous-problème $\mathcal{C}'$, sa clé $k(\mathcal{C}')$ est calculée. Si cette clé est déjà présente dans le cache (l. 2), on renvoie directement le SD associé (l. 4).

3.2.3 Structure des SDs obtenus

Les SDs renvoyés par l'algorithme 1 satisfont toujours la propriété de convergence. En effet, les domaines des variables sont systématiquement réduits, soit par le partitionnement de leur domaine, soit par la propagation. Les SDs obtenus sont également déterministes, puisque la fonction `Split` renvoie une *partition* — les sous-ensembles du domaines courants sont donc disjoints.

Cependant, en fonction des heuristiques de branchement et de choix de variable utilisées par `Split` et `Choose_var`, les dFSDs obtenus peuvent varier :
- On obtient des dRSDs si `Split` partage le domaine en singletons, c'est-à-dire si l'algorithme énumère toutes les valeurs possibles des variables durant la recherche. Au contraire, une recherche dichotomique (qui partage les domaines en deux) renvoie des dFSDs non « *read-once* ».
- On obtient des MDDs si `Choose_var` suit un ordre statique ; mais utiliser des heuristiques pour guider le choix des variables (par exemple **MinDomain**) résulte le plus souvent en des dFSDs non-ordonnés.

3.2.4 Caching

Soit $\mathcal{C}$ le CSP courant à un nœud de recherche donné n, et X_a l'ensemble des variables assignées dans $\mathcal{C}$. L'exploration au nœud n retourne un SD ne mentionnant que des variables de $X \setminus X_a$, qui représente l'ensemble des solutions sur $X \setminus X_a$ quand les variables de X_a sont assignées à leur seule valeur possible. L'objectif est de définir des clés de hachage telles que chaque clé regroupe le plus possible de paires $(\mathcal{C}, X_a)$ « équivalentes » selon l'ensemble de leurs solutions sur $X \setminus X_a$.

Intuitivement, le sous-problème courant peut être représenté par une clé listant les domaines courants de toutes les variables. Il est possible, pour réduire la taille de la clé, d'utiliser la technique présentée par Lecoutre et al. [16], qui consiste à retirer de la clé les domaines des variables x qui sont assignées ($x \in X_a$) et utilisées uniquement dans des contraintes *nécessairement satisfaites* dans le sous-problème courant (parfois appelées contraintes *universelles*). Il est prouvé [16] que ce mécanisme conserve l'ensemble de solutions.

Soulignons que le cache est utile pour deux raisons :
- si le sous-problème courant est incohérent, l'enregistrer en cache peut permettre d'éviter des calculs redondants, si la recherche mène à un sous-problème équivalent ultérieurement. C'est l'utilisation qu'en font Lecoutre et al. [16], qui appliquent cette idée à des algorithmes dont l'objectif est d'exhiber une solution (et non toutes les solutions).
- si le sous-problème courant est cohérent, en plus d'éviter des calculs, enregistrer le SD associé en cache peut permettre de réduire la taille du SD final, si les heuristiques de sélection de variable et de partage des domaines sont dynamiques. En effet, sans le cache, des CSPs équivalents seraient explorés indépendamment, avec des ordres de variables et des partages de domaines différents, surtout si des heuristiques aléatoires sont utilisées. Cela conduirait à des SDs non-isomorphes bien qu'ayant la même interprétation, ce que les opérations de réduction ne détectent pas. En utilisant le cache, on maximise les chances que les SDs équivalents soient toujours isomorphes, ce qui réduit de fait la taille du SD final.

3.2.5 Minimisation du cache

Enregistrer tous les sous-problèmes rencontrées peut conduire à un cache de très grande taille. Pour garder un cache de taille raisonnable (qui puisse être conservé dans la mémoire vive), nous avons choisi d'utiliser un mécanisme de « *restart* » : la taille du cache est limitée à une valeur arbitraire, et à chaque fois qu'elle est atteinte, une partie des entrées sont supprimées — l'objectif étant de garder les entrées les plus intéressantes.

Le critère de sélection des entrées à supprimer que nous avons choisi est le suivant : celles qui ont été le moins utilisées d'abord, puis les plus anciennes, puis les plus longues (qui correspondent aux plus petits sous-problèmes).

Ce compromis permet à l'algorithme d'être plus rapide et d'utiliser moins de mémoire, et ainsi de compiler des problèmes plus gros. En revanche, il n'améliore pas le SD résultant ; cela peut même être le contraire (voir sous-section précédente).

3.3 Heuristiques

Nous détaillons ici les différentes alternatives que nous avons considérées pour la fonction `Choose_var`.

3.3.1 Heuristiques classiques

Nous avons utilisé des heuristiques standard de sélection de variables pour la résolution de CSPs :

- **MinDomain** La variable choisie est celle qui a le plus petit domaine.
- **Dom/WDeg** La variable choisie est celle qui minimise le ratio $|\mathrm{Dom}(x)|/\deg(x)$, où $\deg(x)$ est le nombre de contraintes sur x (les contraintes étant pondérées en fonction des conflits) [6].
- **Random :** La variable est choisie aléatoirement.

3.3.2 Heuristiques basées sur le graphe de contraintes

Nous avons utilisé des heuristiques basées sur le graphe de contraintes (graphe dont les variables sont des nœuds, qui sont liés par un arc si et seulement s'il existe une contrainte liant les deux variables). Pour une variable x, on note $\mathrm{N}(x)$ l'ensemble des variables liées à x dans le graphe.

Algorithm 2 `Next_var`(O) choisit la variable suivante, en fonction d'un ordre courant $O = \{o_1, \ldots, o_k\}$.

1: **if** $O = \varnothing$ **then**
2: **return** $\mathrm{Argmax}_{x \in X} |\mathrm{N}(x)|$
3: soit $x := \mathrm{Argmax}_{x \in X \setminus O} \mathcal{H}_S(x)$
4: ajouter x à la fin de l'ordre O
5: **return** x

Les heuristiques suivantes sont basées sur l'algorithme 2, et font varier le critère $\mathcal{H}_S(x)$. Elles ont été introduites par Amilhastre [1].

- **HBW** $\mathcal{H}_S(x) = \max_{1 \leq i \leq |O|, o_i \in \mathrm{N}(x)} |O| - i$ (choisit en priorité une voisine de o_1, puis de o_2, etc).
- **HSBW** $\mathcal{H}_S(x) = \sum_{1 \leq i \leq |O|, o_i \in \mathrm{N}(x)} |O| - i$ (choisit une voisine des variables les plus anciennement choisies).
- **MCSInv** $\mathcal{H}_S(x) = |\mathrm{N}(x) \cap O|$ (choisit la variable la plus liée aux variables déjà choisies).

3.3.3 Heuristique exploitant le cache

Nous avons implanté une heuristique visant à maximiser l'utilisation du cache. L'idée est qu'un choix de variable menant à un sous-problème déjà traité limite le nombre de nouveaux nœuds. L'heuristique consiste donc à compter, pour toute variable non-assignée x, le nombre $n_{\mathrm{new}}(x)$ de branchements sur x pour lesquels il sera nécessaire d'ouvrir un nouveau nœud de recherche (autrement dit, ce branchement ne mène ni à un nœud du cache, ni à un problème incohérent). On choisit alors la variable x minimisant $n_{\mathrm{new}}(x)$ (en se rabattant sur **HBW** pour casser les ex-æquo). Nous appelons cette heuristique **MaxHashUse**.

3.3.4 Versions statique et dynamique

Les heuristiques **MinDomain**, **Dom/WDeg** et **MaxHashUse** sont toujours dynamiques : elles génèrent le plus souvent des dFSDs non-ordonnés. En ce qui concerne **HBW**, **HSBW**, **MCSInv** et **Random**, il est possible de choisir un ordre statique avant le début de la recherche, ce qui permet d'obtenir des MDDs, ou de choisir la variable dynamiquement. On appelle **DynHBW**, **DynHSBW**, **DynMCSInv** et **DynRandom** les heuristiques dynamiques correspondantes.

4 Expérimentations

Nous avons considéré les problèmes suivants lors de nos expérimentations avec le compilateur « Choco with a trace » :

- *ObsToMem* est un problème de reconfiguration. Il représente un contrôleur gérant les connexions entre l'outil d'observation et la mémoire de masse d'un satellite.
- *Drone* est un problème de planification, dans lequel un drone doit remplir certains objectifs dans un certain nombre de zones en temps limité.
- *NQueens* est le problème des « n reines ».
- *Star* est le problème consistant à colorier un graphe en étoile (une variable centrale liée à des variables indépendantes les unes des autres).

4.1 Heuristiques de choix de variable

Nous comparons tout d'abord les différentes heuristiques de choix de variable, en fixant `Split` à un partage en singletons — nous obtenons donc des dRSDs. Comme certaines de ces heuristiques n'ont pas de version statique, nous ne comparons que les versions dynamiques. Les résultats sont présentés figure 3. **Random** n'est pas incluse ici pour améliorer la lisibilité des graphiques, car elle donne de trop mauvais résultats ; on peut les trouver dans la sous-section suivante.

Il est intéressant de remarquer que **MinDomain** semble la meilleure heuristique pour *ObsToMem* et *NQueens*, alors qu'elle est bien pire que les autres pour

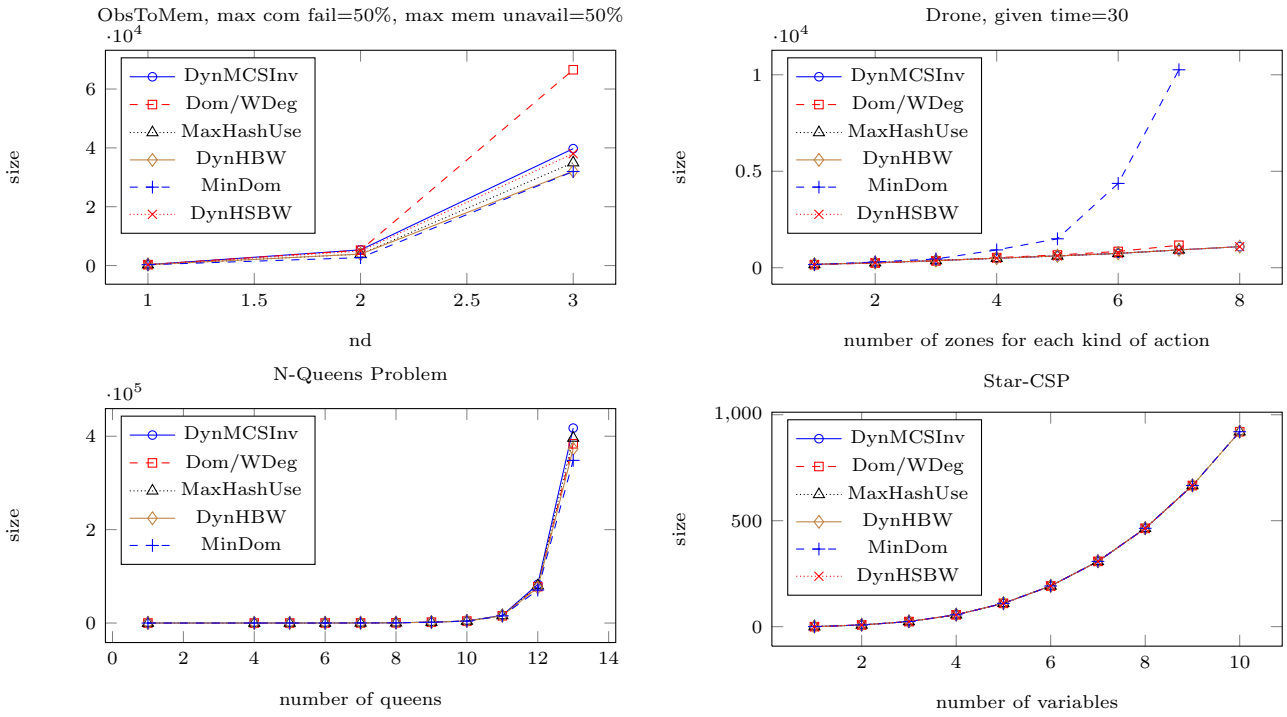

FIGURE 3 – Comparaison entre les heuristiques dynamiques pour les problèmes *ObsToMem*, *Drone*, *NQueens* et *Star*

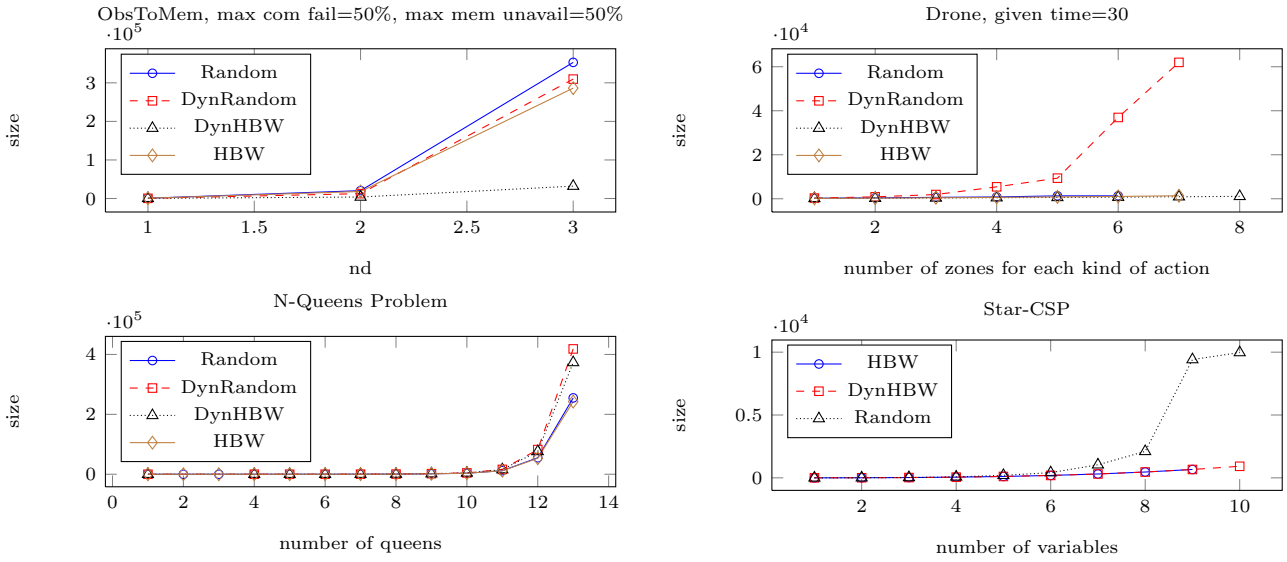

FIGURE 4 – Comparaison entre les versions statique et dynamique de **HBW** et **Random**. Les résultats pour **DynRandom** sur le problème *Star* ne sont pas montrés, étant bien plus mauvais que les autres (la taille du dRSD résultant dépasse les 100 000 pour 7 variables.

Drone. **Dom/WDeg** n'est vraiment efficace dans aucun des trois problèmes. Parmi les heuristiques basées sur le graphe de contraintes, la meilleure est **HBW**. **MaxHashUse** n'est pas mauvaise, mais ne surclasse pas **HBW** ; il semble que regarder seulement une étape en avant ne soit pas suffisant pour maximiser l'utilisation du cache (on choisit la meilleures variable pour le nœud suivant, ouvrant ainsi moins de nouveaux sous-graphes, mais ces sous-graphes sont plus gros). En ce qui concerne *Star*, le fait que toutes les courbes coïncident s'explique en remarquant que toutes les heuristiques choisissent la variable centrale en premier.

4.2 Comparaison entre ordres statique et dynamique

Nous comparons à présent les résultats obtenus en utilisant les versions statique et dynamique d'une même heuristique, avec la même fonction de partage des domaines que dans la sous-section précédente : on obtient respectivement des MDDs et des dRSDs. Les résultats pour **HBW** (meilleure heuristique de la section précédente) et **Random** sont montrés figure 4. On voit que **DynRandom** est beaucoup plus mauvaise que sa version statique ; en effet, le fait d'utiliser un ordre statique augmente fortement la probabilité de tomber sur des nœuds isomorphes. Les résultats pour **DynHBW** sont meilleurs que pour **HBW** statique pour les problèmes réels, mais pas pour les problèmes plus petits (pour *NQueens*, les MDDs obtenus sont même plus petits que les dRSDs). Cependant, on ne peut tirer de conclusions générales sur les qualités relatives des versions statique et dynamique, qui dépendent fortement du problème et de l'heuristique considérés ; sur la figure 5, on voit que pour le problème *Drone*, alors que **DynHBW** et **DynHSBW** coïncident, leurs versions statiques sont tantôt meilleures (**HSBW**) tantôt moins bonnes (**HBW**).

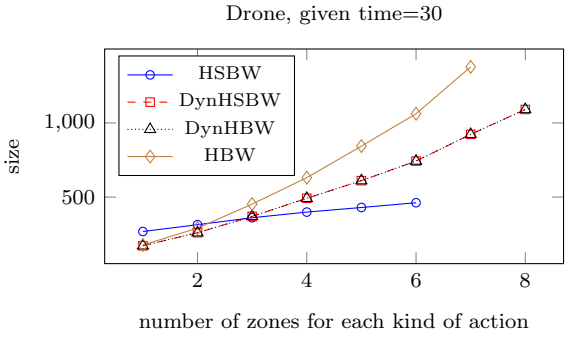

FIGURE 5 – Comparaison entre les versions statique et dynamique de **HBW** et **HSBW** pour le problème *Drone*.

4.3 Influence de la fonction de partage des domaines

Nous étudions à présent comment le choix de la fonction de partage des domaines (`Split`) affecte le SD résultant. Utiliser un partage énumératif (qui divise le domaine en singletons) permet d'obtenir des dRSDs, tandis qu'un partage dichotomique (qui divise le domaine en deux) permet d'obtenir des dFSDs. Les résultats sont présentés figure 6. Ils sont meilleurs dans le premier cas, le nombre de nœuds créés étant bien moindre. Néanmoins, cela n'implique pas qu'un dRSD est toujours plus petit qu'un dFSD équivalent — seulement que notre algorithme n'est apparemment pas efficace pour construire des dFSDs tirant parti du relâchement de la contrainte « *read-once* ».

5 Conclusion

Dans cet article, nous avons introduit les diagrammes à étiquettes ensemblistes, qui généralisent les MDDs en relâchant les propriétés d'ordonnancement, de « *read-once* » et de déterminisme. Nous avons présenté un algorithme permettant de compiler des CSPs discrets en diverses sortes de SDs déterministes (dFSDs, dRSDs et MDDs), en étendant l'approche « DPLL with a trace » sur un solveur de CSP. Nous avons montré comment le choix de certains paramètres de recherche (choix de la variable de branchement et du partage des domaines) affectent la structure du dSD résultant. En utilisant notre implantation de ce compilateur (basée sur le solveur de CSP Choco), sur deux problèmes réels et deux CSPs classiques, nous avons présenté des résultats expérimentaux à propos de l'influence de ces paramètres sur la taille des formes compilées. Ces résultats montrent qu'aucune des heuristiques de choix de variable que nous avons utilisées ne surclasse les autres pour tous les problèmes ; la recherche d'une heuristique globalement efficace reste ouverte. Ils montrent également que le compilateur que nous avons développé semble être utilisable pour compiler des SDs « *read-once* », mais pas purement convergents.

Notre travail s'oriente vers une étude plus approfondie des heuristiques de choix de variable, en particulier **MaxHashUse**. Plus généralement, un de nos objectifs est de compiler efficacement des SDs purement convergents, ainsi que des SDs non-déterministes. Il serait également intéressant d'ajouter des nœuds ET aux SDs, obtenant ainsi un langage proche des AOMDDs ; cela permettrait de comparer la compilation de CSPs en AOMDDs avec des ordres arborescents statiques et dynamiques.

Références

[1] J. Amilhastre. *Représentation par automate d'ensemble de solutions de problèmes de satisfaction de contraintes*. PhD thesis, Université Montpellier II, 1999.

[2] J. Amilhastre, H. Fargier, and P. Marquis. Consistency Restoration and Explanations in Dynamic CSPs — Application to Configuration. *AIJ*, 135(1–2) :199–234, 2002.

[3] J. Amilhastre, P. Vilarem, and M.-C. Vilarem. FA Minimisation Heuristics for a Class of Finite Languages. In *WIA*, pages 1–12, 1999.

[4] H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. A Constraint Store Based on Multivalued Decision Diagrams. In *CP*, pages 118–132, 2007.

[5] J. Bern, J. Gergov, C. Meinel, and A. Slobodová. Boolean manipulation with free bdd's. first experimental results. In *EDAC-ETC-EUROASIC*, pages 200–207, 1994.

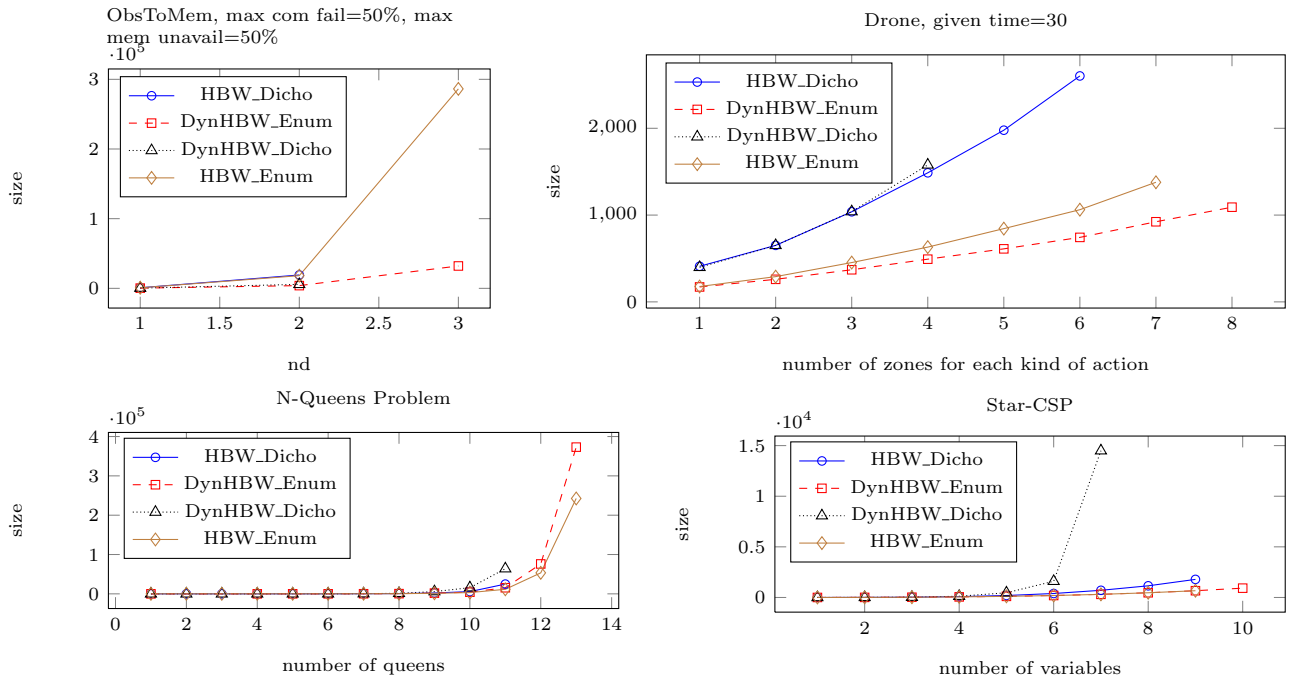

FIGURE 6 – Comparaison entre fonctions de partage de domaines dichotomique et énumérative. La compilation des instances d'ObsToMem non représentées sur le graphique a été interrompue en raison d'un trop grand nombre de nœuds (plus de 700 000).

[6] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *ECAI*, pages 146–150, 2004.

[7] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8) :677–691, 1986.

[8] H. Cambazard, T. Hadzic, and B. O'Sullivan. Knowledge Compilation for Itemset Mining. In *ECAI*, pages 1109–1110, 2010.

[9] choco Team. choco : an open source java constraint programming library. Research report 10-02-INFO, Ecole des Mines de Nantes, 2010.

[10] F. Giunchiglia and P. Traverso. Planning as Model Checking. In *ECP*, pages 1–20, 1999.

[11] T. Hadzic, E. Hansen, and B. B. O'Sullivan. On Automata, MDDs and BDDs in Constraint Satisfaction. In *ECAI Workshop on Inference methods based on Graphical Structures of Knowledge (WIGSK)*, 2008.

[12] T. Hadzic, J. N. Hooker, B. O'Sullivan, and P. Tiedemann. Approximate compilation of constraints into multivalued decision diagrams. In *CP*, pages 448–462, 2008.

[13] J. Hoey, R. St-Aubin, A. J. Hu, and C. Boutilier. SPUDD : Stochastic Planning Using Decision Diagrams. In *UAI*, pages 279–288, 1999.

[14] J. Huang and A. Darwiche. DPLL with a Trace : From SAT to Knowledge Compilation. In *IJCAI*, pages 156–162, 2005.

[15] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued Decision Diagrams : Theory and Applications. *Multiple-Valued Logic*, 4(1–2) :9–62, 1998.

[16] C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Transposition tables for constraint satisfaction. In *AAAI*, pages 243–248, 2007.

[17] J. Lind-Nielsen. BuDDy : Binary Decision Diagrams Library Package, release 2.4, 2002. `http://sourceforge.net/projects/buddy/`.

[18] R. Mateescu, R. Dechter, and R. Marinescu. AND/OR multi-valued decision diagrams (AOMDDs) for graphical models. *J. Artif. Intell. Res. (JAIR)*, 33 :465–519, 2008.

[19] A. Niveau, H. Fargier, and C. Pralet. Representing CSPs with set-labeled diagrams : A compilation map. In *Proc. of the 2nd International Workshop on Graph Structures for Knowledge Representation and Reasoning (GKR)*, 2011.

[20] A. Niveau, H. Fargier, C. Pralet, and G. Verfaillie. Knowledge compilation using interval automata and applications to planning. In *ECAI*, pages 459–464, 2010.

[21] F. Somenzi. CUDD : Colorado University Decision Diagram package, release 2.4.1, 2005. `http://vlsi.colorado.edu/~fabio/CUDD/`.

[22] A. Srinivasan, T. Ham, S. Malik, and R. Brayton. Algorithms for discrete function manipulation. In *ICCAD-90*, pages 92 –95, Nov. 1990.

[23] K. Strehl and L. Thiele. Symbolic model checking of process networks using interval diagram techniques. In *Proc. of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 686–692, 1998.

[24] P. Torasso and G. Torta. Model-Based Diagnosis Through OBDD Compilation : A Complexity Analysis. In *Reasoning, Action and Interaction in AI Theories and Systems*, pages 287–305, 2006.

[25] N. R. Vempaty. Solving Constraint Satisfaction Problems Using Finite State Automata. In *AAAI*, pages 453–458, 1992.