Fractionnement intelligent de domaine pour CSPs avec domaines ordonnés

Intelligent Domain Splitting for CSPs with Ordered Domains

M.-C. Silaghi D. Sam-Haroud B. Faltings

Département d'Informatique, Ecole Polytechnique Fédérale de Lausanne,
CH-1015 Lausanne, Suisse
{silaghi, djamila, faltings}@lia.di.epfl.ch

Résumé

Cet article présente une approche pour la recherche des solutions de problèmes de satisfaction de contraintes (CSPs) avec domaines ordonnés. L'approche proposée combine le fractionnement intelligent de domaines avec la recherche par retour-arrière (backtracking) dans le but de générer l'espace entier des solutions. Ces solutions sont produites sous une forme compressée ce qui simplifie substantiellement la représentation des espaces de solutions importants en taille. D'autre part, la notion de fractionnement intelligent réduit de façon significative l'effort de recherche. L'algorithme présenté s'applique aux CSPs avec représentation explicite des contraintes. Bien que conçu pour générer toutes les solutions, il s'avère également utile pour trouver la première solution, ainsi que le démontrent les expériences conduites.

Mots-Clés

Recherche et optimisation, satisfaction de contrainte.

Abstract

This paper presents an approach for searching in CSPs with ordered domains. It combines intelligent domain splitting with backtracking to find the whole solution space. The generated representation significantly helps in compressing large solution spaces. Performing intelligent splitting is another advantage that decreases the search effort. The algorithm can be applied to general systems of constraints with explicit representations. Although designed for generating all solutions, it also proves useful for finding the first solution of hard problems, as shown by the experiments.

Keywords

Search or optimization, constraint satisfaction

1 Introduction

La détermination de l'ensemble complet de solutions d'un problème de satisfaction de contrainte est importante dans beaucoup d'applications telles que le calcul de tableurs, la conception avec optimisation multicritères ou la négociation.
La tâche de caractériser toutes les solutions pour un CSP est plus difficile que celle de produire la première solution. Elle est confrontée à des problèmes de complexité non seulement pour extraire mais aussi pour représenter les solutions.
Une façon intuitive de contourner cette barrière de complexité est de structurer l'espace de recherche de sorte que l'algorithme d'exploration agisse sur des ensembles agrégés de données plutôt que sur différentes instantiations individuelles possibles.
Dans le cas discret, la représentation par produits cartésiens (cross-product representation CPR) proposée dans [8] est une technique efficace permettant de décrire de manière compacte l'espace complet des solutions d'un CSP. L'idée est d'agréger plusieurs solutions partielles en les représentant sous la forme d'une union de produits cartésiens, chaque produit représentant de façon compacte un certain nombre de combinaisons licites de valeurs. Dans le cas continu, les agrégations de valeurs sont typiquement représentées au moyen d'intervalles.
La méthode que nous proposons combine la puissance des deux approches d'agrégation pour des CSPs discrets ordonnés.
La représentation d'intervalle s'étend naturellement au cas discret pour des domaines ordonnés. Les techniques de recherche basée sur intervalles ainsi que les algorithmes de propagation locale provenant des domaines continus peuvent ainsi naturellement être transférées à des CSPs ordonnés discrets.

x/y	a	b	c	d
e	1	0	1	0
f	0	1	0	1
g	1	0	1	0
h	0	1	0	1

(1)

x/y	a	c	b	d
e	1	1	0	0
g	1	1	0	0
h	0	0	1	1
f	0	0	1	1

(2)

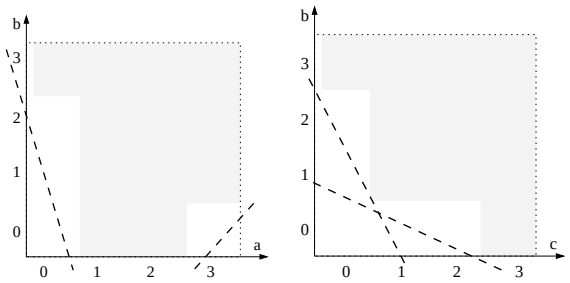
FIG. 1 – *Une contrainte discrète entre deux variables x et y. x prend des valeurs dans l'ensemble { e, f, g, h } et y en { a, b, c, d }. En ordonnant les valeurs selon (2), on induit des groupements plus importants des valeurs faisables dans la représentation de contrainte qu'en les ordonnant selon (1).*

Les techniques numériques de recherche basées sur intervalles sont essentiellement dichotomiques. Les variables sont instantiées par des intervalles. Quand la recherche atteint un intervalle qui ne contient que des solutions, elle s'arrête. Autrement l'intervalle est divisé de manière récursive en plus petits sous-intervalles jusqu'à une résolution établie. Ce processus est combiné avec la propagation locale de consistance pour réduire l'explosion combinatoire [9].

Dans la recherche numérique classique avec représentation intensionnelle des contraintes, il est plus pratique d'utiliser un mécanisme de bisection rigide. Dans notre cas cependant, la représentation explicite des contraintes permet la mise en place d'une technique de fractionnement flexible et plus intelligente des domaines de recherches.

Le nouvel algorithme de backtracking que nous présentons combine le fractionnement intelligent avec une technique locale de consistance inspirée de la hull-consistance [10, 2], un des algorithmes de propagation les plus puissants dans les domaines continus. L'espace de solutions est produit avec concision en utilisant l'approche CPR. Nous démontrons que l'utilisation de la notion d'intervalle simplifie considérablement la mise en place de CPR dans des domaines ordonnés.

La condition d'avoir des domaines ordonnés est remplie par la plupart des types de contraintes [3]. L'ordre peut être arbitraire, déterminé par des heuristiques ou naturel, comme dans des applications numériques. Chaque ordre des valeurs dans les domaines induit certains groupements des valeurs faisables dans les représentations explicites de contraintes (schéma 1). Nous caractérisons un ordre donné par une mesure appelée sa densité, calculée comme le rapport de nombre de voisinages faisables sur le nombre d'éléments faisables dans la représentation explicite des contraintes avec des matrices (0,1).



1	1	1	1
0	1	1	1
0	1	1	1
0	1	1	0

3)

1	1	1	1
0	1	1	1
0	1	1	1
0	0	0	1

4)

FIG. 2 – *Un CSP numérique sur des nombres entiers avec trois variables a, b, c de domaine { 0,1,2,3 } et quatre inégalités linéaires (lignes pointillées). 1) les contraintes entre a et b sont $b \geq 3 - 3a \wedge b > a - 3$; 2) les contraintes entre b et c sont $b \geq 3 - 2c \wedge 2b \geq 3 - c$; 3) la représentation de la première contrainte avec une matrice (0,1); 4) la représentation de la deuxième contrainte avec une matrice (0,1).*

Nous estimons que l'approche présentée est d'autant plus efficace que l'ordre donné a une densité naturellement élevée, comme cela est souvent le cas dans les applications numériques sous contraintes (avec domaines finis ou discretisés), les problèmes de planification, de coloration de graphes, etc..

2 Exemple

Nous commençons par illustrer la notion de fractionnement intelligent au moyen d'un petit exemple.

Considérons le CSP numérique sur des nombres entiers du schéma 2. Le problème a trois variables a, b et c, dont chacune prend ses valeurs dans l'ensemble ordonné {0,1,2,3}. Les variables a et b (respectivement b et c) sont contraintes par deux inégalités (schéma 2 (1) et (2)). Nous supposons que les contraintes sont représentées discrètement au moyen de matrices (0,1) et que les domaines sont ordonnés selon l'ordre naturel des nombres entiers.

Un algorithme standard de backtracking pour CSPs discrets exigerait 64 tests pour produire toutes les solutions: chacune des quatre valeurs possibles pour c doit être testées par rapport aux 12 paires faisables de valeurs pour a et b. La détermination des 12 paires faisables pour a et b exige un total de 16 tests.

Supposons maintenant que plutôt que d'assigner à une variable donnée une valeur individuelle à chaque étape,

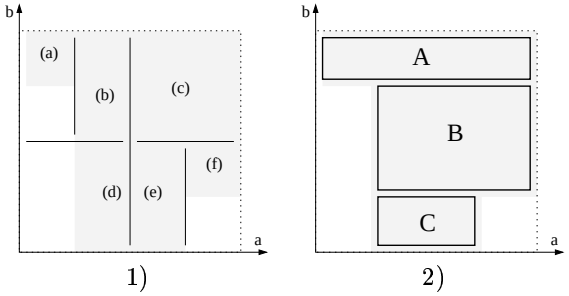


FIG. 3 – *Approches du fractionnement de domaine: 1) fractionnement binaire; 2) fractionnement intelligent.*

nous voulions assigner simultanément un groupe de valeurs.

Une première façon serait de procéder selon le schéma de recherche dichotomique, tel que décrit pour les CSPs continus. Les domaines étant ordonnés, une agrégation de valeurs peut en effet être représentée par un intervalle. En utilisant cette technique, la région de solutions déterminée par les contraintes entre a et b serait décomposée en six régions, comme représenté sur le schéma 3(1). Le nombre de tests exigés pour produire toutes les solutions est 56 dans ce cas-ci: 16 tests sont nécessaires pour décomposer les contraintes en 6 régions faisables. Le choix de a et b dans la région (a) ou (f) implique 4 tests par région pour c et dans la région (b), (c), (d) ou (e) 8 tests par région.

Le procédé de bisection agrège implicitement les valeurs qui peuvent être assignées simultanément. Ceci permet de générer une description plus compacte de l'espace de solutions avec un nombre inférieur de tests. Le mécanisme d'agrégation offert par la recherche dichotomique demeure cependant non informé puisque les descriptions de contraintes ne sont pas prises en considération pour exécuter le fractionnement. La bisection coupe donc avec une grande probabilité juste au milieu des régions faisables, induisant d'autres fractionnements inutiles.

L'idée du fractionnement intelligent consiste à tenter de réduire le nombre de régions faisables produites en choisissant des points de fractionnement appropriés. La région de solutions déterminée par les contraintes entre a et b pourrait par exemple être décomposée en trois régions (A), (B) et (C), comme représenté sur le schéma 3(2). Le nombre de tests exigés pour produire toutes les solutions se réduit à 32 en ce cas: 16 tests doivent être utilisés pour produire la décomposition. Le choix de a et b dans la région (A) ou (C) implique 4 tests par région pour c et dans la région (B), 8 tests. Dans la section 4.2, nous présentons un algorithme permettant de réaliser un tel fractionnement. L'algorithme établit les régions par expansion, selon des directions choisies en fonction d'heuristique d'ordonnancement. Bien que non optimale, la stratégie de fractionnement proposée réduit sensiblement l'effort de recherche, comme le démontrent les expériences.

3 Travaux antérieurs

Nous considérons le cas général de CSPs n-aires et utilisons leurs représentations classiques de graphes/hyper-graphes. Dans la représentation primale, les noeuds représentent les variables, et les arcs, les contraintes qui les relient. Dans le dual, un noeud représente une contrainte et les arcs sont des contraintes d'égalité entre les variables communes [1]. L'étiquette d'un noeud désigne l'ensemble des valeurs que ce noeud peut prendre.

Nous proposons un algorithme hybride de recherche avec backtracking qui utilise les deux représentations, primale et duale. Le graphe primal supporte une formulation compacte de l'espace de solutions. Le dual est employé pour exécuter le mécanisme de fractionnement intelligent et pour contrôler le procédé de retour-arrière. L'algorithme que nous présentons est une variante de MAC (maintenance de consistance d'arc) [12]. MAC impose la consistance d'arc sur les variables non instantiées restantes, après l'instantiation de chaque variable. Il s'est avéré être l'une des meilleures techniques pour des problèmes difficiles [4]. Dans notre approche, nous mettons à jour une forme plus faible de consistance d'arc empruntée des domaines continus, à savoir la hull-consistance [10, 2]. La hull-consistance impose la consistance d'arc sur les limites externes du domaine d'une variable. Elle produit une projection convexe de l'ensemble de valeurs possibles.

Quand une variable doit être choisie, les heuristiques que nous avons testées sont celles instantiant d'abord les variables avec les plus petits domaines (LD) [5] ainsi que les variables les plus contraintes. LD peut aussi être vue comme une heuristique pour le choix de la variable la plus contrainte. Des algorithmes voisins de ceux présentés ici se trouvent en [8, 6] et [13].

4 Algorithmes

Comme susmentionné, la technique de recherche que nous présentons utilise les deux représentations, primale et duale, des CSPs. Nous choisissons d'utiliser les termes "variable" et "contrainte" seulement pour les éléments de la représentation primale.

Dans la représentation primale, l'étiquette assignée à un noeud, et qui représente l'ensemble des valeurs qu'il peut prendre, est un intervalle discret.

Définition 1 (intervalle discret) *Soit I un ensem-*

ble de valeurs prises d'un domaine discret totalement ordonné D. I est un intervalle discret *si ses éléments sont successifs dans D.*

Un intervalle discret peut être représenté extensionnellement en énumérant tous ses éléments, ou intensionnellement en utilisant ses valeurs extrêmes.
Dans la représentation duale, le domaine pour chaque noeud est une contrainte. Les noeuds sont étiquetés en utilisant la notion des blocs multidimensionnels, appellés multi-blocs.

Définition 2 (multi-bloc) *Soit C une contrainte entre un ensemble $V = \{v_1, \ldots v_k\}$ de variables. Soit $\{I_1, \ldots I_k\}$ un ensemble d'intervalles discrets respectivement assignés aux éléments de V à partir de leurs domaines. Le produit cartésien $I_1 \times \ldots I_k$ est un multi-bloc de C ssi il satisfait C.*

La technique de recherche procède comme suit. Lorsqu'une contrainte est choisie pour instantiation, son domaine est ramené au produit cartésien des domaines des variables qu'elle implique. Un noeud dual c est choisi et instantié avec un multi-bloc dans son domaine actuel. Les étiquettes des noeuds primaux correspondants sont alors limitées aux valeurs permises par le multi-bloc. Cette restriction est propagée dans le graphe primal en utilisant un algorithme local de consistance. Si le domaine d'une variable est vidé, un autre multi-bloc est choisi pour c. Un backtracking est lancé si tous les multi-blocs de c ont été épuisés. Quand tous les noeuds duaux ont été instantiés, une région de solutions est fournie. Cette région de solutions est donnée par le produit cartésien des valeurs de leurs étiquettes (intervalles) dans le graphe primal. L'utilisation de la représentation duale est principalement destinée à permettre un mécanisme facile d'agrégation. Nous motivons ce choix plus en détail dans la Section 5.
Nous pouvons alors présenter notre algorithme. Nous commençons par décrire l'algorithme local de propagation utilisé pendant la recherche.

4.1 "Hull-Consistance"

En imposant la consistance d'arc sur les limites externes des domaines seulement, la "hull-consistance" épargne une quantité significative de travail et d'espace. Elle s'est avérée particulièrement utile dans le cas continu et est intégrée dans les techniques les plus efficaces pour les CSPs numériques. L'utilisation de cette notion a été également suggérée pour des problèmes dans des domaines finis [11]. On note par $\prod_{v_i} C$ la projection de la contrainte C sur la variable v_i.

Définition 3 (hull-consistance) *Soit $\mathcal{S}$ un système de contraintes discrètes avec n variables $v_1, ..., v_n$, $\mathcal{I}$ un produit cartésien de n* intervalles discrets $\langle I_1, ..., I_n \rangle$ *respectivement assignés aux variables $v_1, ..., v_n$, et soit $l_i = min(I_i)$ et $u_i = max(I_i)$ $(1 \leq i \leq n)$. Soit $C(\mathcal{I})$ l'ensemble des tuples du produit cartésien $\mathcal{I}$ permis par une contrainte donnée C. $\mathcal{S}$ est hull-consistant dans $\mathcal{I}$ par rapport à i si*

$$\{l_i, u_i\} \in \bigcap_{C \in \mathcal{S}} \prod_{v_i} C(\mathcal{I}).$$

$\mathcal{S}$ est hull-consistant dans $\mathcal{I}$ s'il est hull-consistant dans $\mathcal{I}$ par rapport à tous les i dans 1...n.

L'algorithme 1 réalise la hull-consistance. La mise en place utilisée est basée sur AC-6 [3]. Elle exige d'enregistrer pour chaque borne de chaque contrainte testée le tuple dans la contrainte qui supporte la borne. Ceci exige un espace de $O(NC * MA^2)$ où MA représente le nombre maximum de variables d'une contrainte et NC le nombre de contraintes.

```
function DHullConsistency(Queue)
    while c=ExtractConstraint(Queue) do
        if !CheckBounds(Queue,c) then
            return 0
        end
    end
    return 1
end.
```

Algorithme 1: *Hull-Consistance Discrète*

Au commencement toutes les contraintes sont insérées dans la file d'attente. Une contrainte est réinsérée quand le domaine d'une de ses variables change.
Une étape de révision vérifie que les valeurs supportant les bornes de la contrainte actuelle sont encore valides. Pour n'importe quel support tombant en dehors des intervalles actuels des variables (lignes 2.1 et 2.2), la contrainte est balayée de sorte qu'un nouveau support puisse-t-être identifié. Si le balayage est exécuté selon une ordre statique des domaines et des variables impliquées dans les contraintes, alors il est nécessaire de scanner au delà de l'ancien support.

Théorème 1 *L'algorithme de hull-consistance se termine en moins des $NC * DS * NV$ étapes de révision où le DS et les NV représentent la taille maximum de domaine, respectivement le nombre des variables.*

Preuve. Après chaque réduction d'un domaine il y aura au maximum $NC - 1$ contraintes testées sans réduction ultérieure de domaine. Si, après contrôle de toutes les $NC-1$ contraintes, il n'y a aucune réduction de domaine, la consistance est atteinte et l'algorithme

```
function CheckBounds(Q,c)
    for all VariablesInConstraint(v,c) do
        changed←0
2.1     if OutOfBound(UpSupport(v,c)) then
            Bloc←ScanForUpper(v,c)
            changed← 1
        end
2.2     if OutOfBound(LowSupport(v,c)) then
            Bloc←ScanForLower(v,c)
            changed← 1
        end
        if Empty(Bloc,v) then
            return 0
        end
        if changed then
            Insert(Q,AffectedConstraints({v}))
2.3         UpdateVariableRange(Bloc,v)
        end
    end
    return 1
end.
```

Algorithme 2: *Check Bounds*

s'arrête. Il y a au maximum $DS * NV$ réductions de domaine possibles. Par conséquent il y aura au maximum $NC * DS * NV$ étapes avant que la contradiction ou la consistance soit atteinte. □

Théorème 2 *La complexité de l'algorithme de Hull-Consistance est* $O(NC * DS * (NV * MA + DS^{MA-1}))$.

Preuve. Les domaines des contraintes sont balayés seulement une fois, ceci a pour conséquence un nombre de $O(NC * DS^{MA})$ tests. Les seules valeurs qui doivent être reconsidérées en tant que supports sont celles des variables changées. Cependant, à chaque étape de révision, tous les supports des bornes d'une contrainte (éventuellement un de moins[1]) doivent être testés, ayant pour résultat une complexité de $O(NC * DS * NV * MA)$. □

Une différence entre la hull-consistance et un algorithme standard de AC, est que la modification d'un domaine D_k exige que les contraintes soient testées également pour la réduction des limites déjà réduites de D_k. Une telle situation n'est pas possible avec le AC classique où les valeurs correspondantes dans le D_k réduit ont déjà été testées dans les étapes précédentes.

1. Si seule une borne d'une variable donnée est réduite, alors le support de l'autre borne ne doit pas être testé.

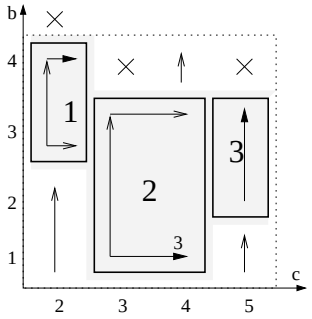


FIG. 4 – *Exemple d'exécution d'une procédure dynamique d'agrégation. Les flèches montrent la direction du balayage. Les croix montrent les points de départ. Les rectangles sont les multi-blocs résultants et les grands nombres à l'intérieur montrent l'ordre dans lequel ils sont retournés. Les plus petits nombres dans les rectangles montrent le déplacement enregistré par le procédé de balayage.*

4.2 Implémentation de l'agrégation

Nous présentons maintenant le procédé dynamique d'agrégation que nous proposons pour calculer les multi-blocs. Ce procédé balaye une contrainte selon un ordre de variables. Cet ordre indique des priorités sur les directions de l'agrégation. On utilise une structure enregistrant le progrès qui mémorise le point de départ pour l'agrégation à chaque étape. Etant donné un ordre des variables, à chaque demande d'un multi-bloc, nous balayons systématiquement la contrainte jusqu'à un tuple faisable. Le point de départ est hérité de l'appel précédent. Une fois qu'un tuple faisable est trouvé, nous essayons de l'augmenter systématiquement le long de toutes les variables selon l'ordre fourni par l'heuristique décrite dans la Section 6. Les tuples du bloc obtenu sont marqués comme déjà couverts, de sorte que les futures lectures et agrégations les évitent. Le point de départ pour la prochaine demande est placé à droite du bloc, dans la direction avec la priorité la plus élevée. Pour permettre à la procédure de balayage d'éviter les grands blocs déjà couverts nous assignons aux tuples les plus à gauche, la largeur du bloc couvert (déplacement). L'heuristique d'ordonnancement utilisée est basée sur la probabilité qu'un bloc puisse-t-être agrégé le plus sur sa direction prioritaire.

Le procédé d'agrégation est illustré par l'exemple du schéma 4. Le domaine actuel de la contrainte (c, b) est associé au produit cartésien $[2, 5] \times [1, 4]$. L'itérateur est placé dans la position (2,1). Considérons que l'heuristique d'ordonnancement nous donne l'ordre c, b. Nous balayons la contrainte comme montré par la

flèche commençant à (2,1) et nous découvrons un tuple faisable à (2,3). Nous l'étendons d'abord pour c obtenant le premier bloc $[2,2]\times[3,4]$. Alors nous essayons de l'étendre pour b, opération qui échoue en raison du tuple infaisable (3,4). Le point suivant est placé à (2,5), et le multi-bloc 1 est retourné.

Quand une nouvelle requête pour un bloc est reçue, le point de départ est lu et le balayage continue depuis (3,1), où il trouve également un tuple faisable et s'arrête. Le bloc obtenu est étiré d'abord pour c jusqu'au bloc $[3,3]\times[1,3]$. Puis, il est augmenté pour b et il devient $[3,4]\times[1,3]$. Nous n'essayons pas de l'augmenter dans la direction décroissante d'une variable. Le nouveau point de départ est placé à (3,4) et le déplacement de saut pour le balayage dans (4,1) à 3. Alors le multi-bloc 2 est retourné. De la même manière, à la troisième demande, le balayage commence depuis (3,4), teste (4,1) et saute à (4,4). Il continue avec (5,1) et découvre un tuple faisable à (5,2). Le bloc est augmenté pour c et le multi-bloc 3 est livré. A la quatrième demande, le point de départ (5,4) est testé et le signal de contrainte vide est donné.

La complexité de l'algorithme pour le balayage de toute la contrainte est $O(MA * DS^{MA})$. La méthode d'agrégation proposée dans [6] a une complexité moindre de $O(DS^{MA})$ mais demande la gestion de plus de structures.

4.3 MHC

Nous décrivons dans cette section notre algorithme de maintenance de hull-consistance appelé MHC.

Au début, les étiquettes assignées aux variables sont leurs domaines initiaux. La hull-consistance est appelée avec toutes les contraintes dans la file d'attente, ce qui permet d'obtenir des étiquettes hull-consistantes.

La recherche est commencée en appelant MHC(0, nil). A l'étape i, la contrainte i est instantiée par un multi-bloc complètement faisable, dans les limites permises par les étiquettes courantes des variables. Chaque fois qu'une nouvelle contrainte est abordée, nous estimons dynamiquement ses priorités d'agrégation, c'est à dire l'ordre des variables à considérer pour l'agrégation (ligne 3.5).

L'instantiation est donnée par un itérateur initialisé à la ligne 3.4 et qui construit les multi-blocs par agrégation 4.2. L'algorithme 4 exécute l'agrégation le long de toutes les directions selon la priorité calculée par une heuristique dédiée (section 6) et il est appelé à la ligne 3.6. Entre deux appels, le dernier tuple testé dans l'ordre de lecture qui ne laisse aucun tuple non testé derrière, est enregistré comme point de départ (ligne 4.3). Si une telle instantiation est réussie, nous intersectons les étiquettes actuelles des variables impliquées dans la contrainte avec les intervalles couverts par le multi-bloc choisi (ligne 3.8). Si une étape mentionnée modifie les domaines, alors les domaines courants doivent être rechargés depuis la pile (ligne 3.7).

```
     function MHC (i, Stack)
3.1  |  if LastConstraint(i) then
3.2  |  |  Report(SolutionCrossProduct)
     |  |  return
     |  end
3.3  |  PushCurrentDomains(Stack)
3.4  |  ResetIterator
3.5  |  ReOrderVariables(i)
3.6  |  while GetNextHull(i) do
3.7  |  |  LoadDomains(Stack)
3.8  |  |  UpdateDomains(ChosenHull)
     |  |  Q←AffectedConstraints(i)
3.9  |  |  if !DHullConsistency(Q) then
     |  |  |  continue
     |  |  end
3.10 |  |  ReOrderConstraints(i)
3.11 |  |  MHC(i + 1, Stack)
     |  end
3.12 |  PopCurrentDomains(Stack)
     |  return
     end.
```

Algorithme 3: *MHC*

La pile de domaines est chargée et déchargée chaque fois qu'une contrainte est abordée respectivement abandonnée (ligne 3.3 respectivement ligne 3.12). L'instantiation d'une contrainte correspond à un fractionnement de la contrainte et de plusieurs fractionnements des variables. Nous exécutons un backtracking basé sur de telles étapes. Toutes les fois que toutes les contraintes sont instantiées (ligne 3.1), nous produisons des solutions sous la forme de produits cartésiens des étiquettes actuelles des variables (ligne 3.2).

La ligne 3.9 appelle l'algorithme réalisant la hull-consistance. Nous savons que les contraintes instantiées ont des supports pour toutes leurs limites et seules les contraintes futures sont révisées. Si la fonction échoue, une nouvelle étape commence par la recherche d'un nouveau multi-bloc. Lors de cette étape et à chaque backtracking, on restaure les supports pour les limites employés par la hull-consistance discrète. Si la propagation locale n'a pas comme conséquence l'élimination de domaines, les contraintes futures sont réordonnées (ligne 3.10) selon l'heuris-

tique LD et MHC est alors récursivement appelé à la ligne 3.11.

```
    function GetNextHull
4.1 |  if ! ScanFurtherForNewTuple then
    |  |  return 0;
    |  end
    |  repeat
    |  |  if ExpandableOnADirection then
4.2 |  |  |  ExpandOnFirstExpandableDirection
    |  |  end
    |  until ! SuccesfullyExpanded
4.3 |  StoreProgressInfo
    |  return 1
    end.
```

Algorithme 4: *Procédure d'Agrégation*

Théorème 3 *L'algorithme MHC est correct et complet.*

Preuve. Chaque tuple de solutions a au moins un sous-tuple faisable correspondant dans chaque contrainte. Quand une solution est générée comme le produit cartésien des étiquettes de variables, nous savons que toutes les contraintes ont été prises en considération et instantiées au moyen de multi-blocs (produits cartésiens). Il est garanti par construction que la projection de n'importe quel multi-bloc sur une variable donnée est un intervalle qui contient l'étiquette de cette variable. Ceci garantit que l'algorithme est correct. L'algorithme d'agrégation balaye exhaustivement l'espace entier de la contrainte compatible avec l'instantiation des contraintes passées. La hull-consistance est correcte et complète. Par conséquent le MHC est complet. □

5 CPR dans des domaines ordonnés

Nous comparons maintenant l'approche initiale de CPR avec l'implantation basée sur dual que nous proposons. Cette implantation est désignée sous le nom de DCPR dans le reste du papier. Nous rappelons d'abord comment le CPR basé sur le primal fonctionne. Supposons qu'à une étape donnée, $k-1$ variables $x_1, \ldots x_{k-1}$ ont déjà été instantiées et que nous voulions assigner des valeurs agrégées à une nouvelle variable x_k . L'instantiation actuelle pour $x_1, \ldots x_{k-1}$ est structurée comme produit cartésien b_{k-1}. Chaque valeur possible v_i pour x_k doit être d'abord testée par rapport à b_{k-1}, exigeant la recherche et la représentation des produits cartésiens du type $b^i_{k-1} \times \{v_i\}$.

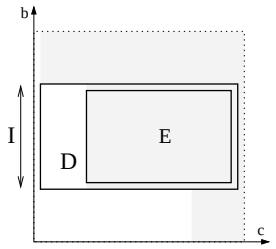


FIG. 5 – *Si la variable b était instantiée par un intervalle I, le domaine D où se fait la construction de E par agrégation est donné par l'intersection entre I et la contrainte entre b et c.*

La technique de CPR exige que ces représentations soient restructurées par le fusionnement de ces produits cartésiens.

Dans notre cas, exécuter l'instantiation en dual fournit directement les produits cartésiens. Ces produits cartésiens en fait sont générés l'un après l'autre et l'étape de fusionnement est implicite. Les domaines où le fusionnement implicite est exécuté sont déterminés par simple intersection d'intervalles, comme représenté sur le schéma 5. Par ailleurs, la gestion des données pour enregistrer explicitement tous les produits cartésiens partiels n'est plus nécessaire. L'exécution de DCPR dans des domaines ordonnés fournit également une représentation concise pour un produit cartésien, qui prend la forme d'un ensemble d'intervalles.

Pour récapituler, l'implantation basée sur le primal réordonne les valeurs dans les domaines pour exécuter les agrégations. Tant que nous traitons des domaines ordonnés, il n'est pas nécessaire de réordonner dans DCPR. MHC combine DCPR avec la maintenance de la Hull-consistance. Il peut être démontré que l'aggrégation produite par MHC est plus puissante que celle effectuées dans le primal [13]. Ceci est du au fait que MHC combine davantage les résultats de la maintenance de consistance locale avec la procédure d'agrégation.

DCPR offre la possibilité de réordonner les contraintes pendant la recherche permettant plus de flexibilité. Une représentation duale peut incorporer n'importe quel ordre des valeurs et n'importe quel ordre des contraintes tandis que dans le primal l'ordre des contraintes est limité par celui des variables. Notons finalement que DCPR est une approche purement "en profondeur d'abord" (depth-first) et peut par conséquent être plus facilement étendue aux techniques avec "nogoods" ou dites de "backjumping".

Nous présentons maintenant le gain apporté par DCPR. Dans l'approche duale, nous notons par $C_i(b)$

et $V(b)$ le volume (le nombre des tuples instantiées) de la projection de la solution partielle représenté par le produit cartésien b sur les variables instantiées de la contrainte i, respectivement le volume de b. Le volume d'un ensemble vide est 1 par convention. Le gain correspondant de DCPR est:

Théorème 4 *A chaque instantiation d'une nouvelle contrainte i, DCPR réduit le nombre maximum des tests pour une solution partielle courante représentée avec le produit cartésien $cprps$ de:*

$$V(cprps)/C_i(cprps) \quad (1)$$

à 1 pour chaque tuple actif (pas encore éliminé par des nogoods) dans la contrainte i.

Preuve. Avec DCPR chaque tuple est testé une fois. Dans les approches standards, procédant tuple par tuple, il aurait été testé une fois pour chaque combinaison des valeurs dans les étiquettes courantes indépendantes, c'est-à-dire dans les étiquettes de toutes les variables instantiées non impliquées dans la contrainte courante.
Ici nous avons considéré en $V(cprps)$ seulement les variables qui ont été contraintes par les contraintes passées. □

De par la définition de $C_i(b)$, nous voyons que $V(cprps)/C_i(cprps)$ est toujours plus grand que 1.

6 Heuristique

Le procédé d'agrégation utilise des heuristiques pour assigner des priorités aux directions de l'agrégation.
L'heuristique que nous proposons n'est pas conçue pour produire les multi-blocs dotés des plus grands volumes. Elle essaie plutôt de produire des multi-blocs qui promettent le plus grand gain futur dans le nombre de tests. Les ordres de direction correspondent aux ordres des variables.
Puisque l'efficacité dépend des contraintes futures, nous pouvons employer le résultat du théorème 4 pour estimer les gains de chaque direction de l'agrégation. Nous pouvons donc essayer de maximiser la taille des étiquettes pour la variable qui promet le plus.
Nous estimons que le nombre des valeurs d'une variable v qui peuvent être agrégées avec succés le long de l'axe v est proportionnel à la taille de l'étiquette de cette variable $Size(v)$.
En raison de l'équation 1 nous pouvons estimer que pour n'importe quelle contrainte future C_i, pour une variable v quelconque qui n'est pas impliquée en C_i, nous gagnerons avec un facteur de $Size(v)$. C'est le facteur avec lequel $V(cprps)$ de l'équation 1 augmente. Si v est contraint par la contrainte C_i, alors il n'y a aucun gain au niveau de C_i. Dans notre calcul nous disons que pour la variable v nous avons une *perte* de facteur $Size(v)$.
On peut donc estimer que le *gain* d'augmenter la contrainte actuelle C_{crt} le long de la direction v est proportionnel avec

$$\sum_{i>crt, v\notin C_i} \frac{Size(v)V(cprps)}{C_i(cprps)}.$$

Nous avons utilisé une évaluation simplifiée de la *perte* pour augmenter la variable v comme:

$$Size(v)(\sum_{i>crt, v\in C_i} 1). \quad (2)$$

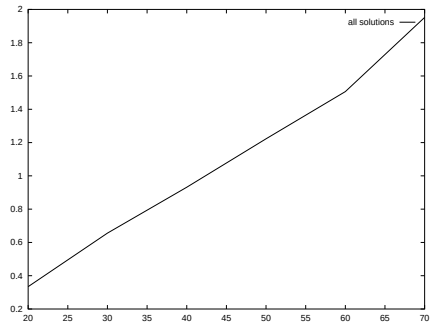


FIG. 6 – *Maintenance de la Hull-Consistance: le rapport moyen entre le temps requis par DCPR et MHC en fonction du nombre de variables. L'échelle est logarithmique. À 70 variables, MHC est en moyenne environ 90 fois plus rapide pour trouver toutes les solutions.*

7 Expériences

Nous considérons le problème de colorer les n noeuds d'un graphe non-orienté avec C couleurs, de sorte qu'aucune paire de noeuds avec la même couleur ne soient joints par un arc. Quand toutes les couleurs possibles pour les noeuds sont ordonnées, les contraintes d'un problème de coloration de graphe sont des matrices remplies de 1, excepté les éléments de la diagonale principale. Cette particularité les rend attrayantes pour des méthodes basées sur l'agrégation. La densité de l'ordre, comme définie dans l'introduction est affectée par le nombre de couleurs dans un problème de coloration de graphe. La densité minimale est 0 pour les problèmes bicolores, la densité maximale est 2 puisqu'elle ne peut pas excéder le nombre maximal de variables du graphe. Un problème tricolore a une densité de 0,66 tandis qu'un problème tétracolore a

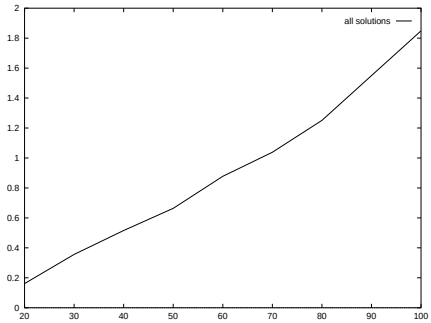


FIG. 7 – *Recherche de toutes les solutions: le rapport entre le temps requis par un backtracking classique avec maintenance de hull-consistance et le MHC en fonction des nombres de variables. L'échelle est logarithmique. Par exemple à 100 variables, MHC était en moyenne environ 80 fois plus rapide pour trouver toutes les solutions.*

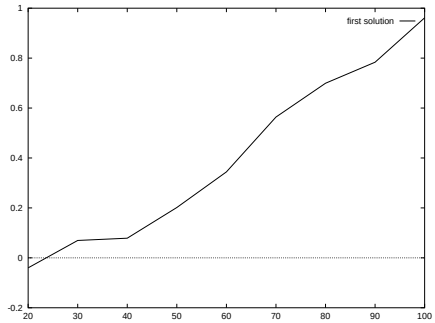


FIG. 8 – *Recherche de la première solution: le rapport moyen entre le temps requis par un backtracking classique avec la maintenance de hull-consistance et le MHC, en fonction des nombres de variables. L'échelle est logarithmique. Par exemple à 100 variables, MHC était environ 10 fois plus rapide pour trouver la première solution.*

une densité de 1. Les expériences ont été principalement exécutées sur les problèmes tricolores.

Les problèmes sont produits de telle manière qu'ils aient au moins une solution et de sorte qu'ils tombent dans le pic de difficulté, comme décrit dans [7]. Avec plus de précision, les graphes produits sont des problèmes de coloration dits *padded* avec *coloration préspécifiée* et une moyenne de 4,9 arcs par noeud [7]. Nous avons également pris en considération le fait que pour des problèmes de coloration de graphe, toutes les solutions obtenues par n'importe quelle permutation des couleurs dans une solution donnée est également une solution. Dans le cas de 3 couleurs, en modifiant un arc au hasard pour permettre seulement une paire de couleurs, nous nous assurons qu'aucune permutation redondante des solutions n'est produite.

Le paramètre que nous avons utilisé pour nos comparaisons est le rapport entre le temps de fonctionnement des algorithmes sur les mêmes problèmes, sur les mêmes machines et avec la même heuristique. Nous avons exécuté environ 400.000 essais sur des problèmes avec des tailles entre 20 et 100 variables. Les résultats pour chaque taille de problème sont présentés comme une moyenne pour environ 10.000 exemples.

Le gain apporté par la maintenance de hull-consistance est montré dans le schéma 6. MHC apporte les améliorations les meilleures pour des problèmes de grandes tailles comme montré sur le schéma 7. Nous avons observé cependant un petit pourcentage ($0-5\%$) de problèmes, généralement très faciles avec peu de variables, qui étaient plus rapides sans agrégation. Cela est dû au fait que le procédé d'agrégation effectue toujours un certain calcul supplémentaire en raison des réordonnement des directions et des agrégations multidimensionnelles. Les différences négatives obtenues en ce cas étaient inférieures à la seconde. Notons également que quelques exemples difficiles ayant nécessité des jours pour être résolus sans agrégation ont pu être résolus en quelques minutes avec agrégations.

Le comportement de MHC a aussi été testé pour trouver la première solution. Comme décrit par le schéma 8, MHC est plus rapide que la maintenance de hull-consistance sans agrégation pour des problèmes difficiles. Il a été dépassé seulement dans quelques cas simples. Le gain augmente avec la difficulté et avec la taille du problème. Nous pouvons donc avec précaution inférer que MHC pourrait être recommandé pour trouver la première solution. Ce résultat était inattendu puisque MHC est susceptible de calculer les solutions inutiles par l'agrégation.

Il était intéressant d'étudier l'efficacité du compactage offert par MHC pour représenter les solutions (schéma 9). Comme prévu, l'efficacité de compactage augmente avec la taille du problème.

Nous avons également exécuté quelques essais pour des problèmes de coloration avec 30 variables et 4 couleurs. Le rapport moyen pour toutes les solutions était environ 10 fois meilleur en temps et compactage que pour 3 couleurs.

8 Conclusions

Ce travail a proposé une technique de recherche pour produire toutes les solutions de CSPs discrets ordonnés. L'idée centrale consiste à utiliser conjointement des intervalles et des produits cartésiens pour représenter l'agrégation des valeurs. Ceci a permis d'u-

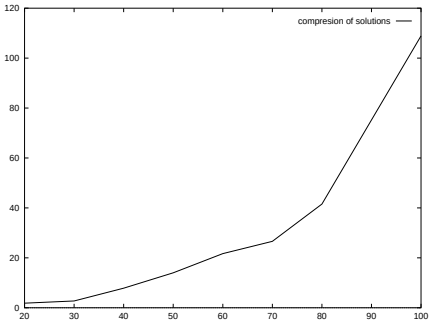


FIG. 9 – *Compactage de l'espace de solution: le rapport entre l'espace nécessaire pour représenter toutes les solutions par un algorithme de recherche avec la maintenance de hull-consistance et le MHC, en fonction du nombre de variables.*

tiliser à profit des techniques locales de propagation empruntées des domaines continus.
Ces techniques ont été mises en valeur par la représentation explicite des contraintes que nous avons utilisée. Cette représentation a servi de base pour concevoir des mécanismes de fractionnements intelligents, pouvant s'avérer plus efficaces que le fractionnement binaire classique. L'espace de solutions est représenté de manière compacte en utilisant CPR. Nous avons montré que l'implémentation de CPR utilisant le dual offre beaucoup d'avantages tels que permettre des approches purement depth-first et qu'elle épargne la gestion de structures superflues.
Les résultats encourageants obtenus par l'évaluation expérimentale des problèmes de coloration de graphes suggèrent l'utilité d'investiguer l'apport de cette technique dans d'autres contextes.
Dans les travaux futurs, nous voudrions étudier l'utilité de l'approche dans les applications avec des domaines naturellement ordonnés tels qu'en planification, dans les bases de données ou pour les CSPs hybrides continu-discrets.

9 Remerciements

Ce travail a été exécuté au laboratoire d'intelligence artificielle de l'Ecole Polytechnique Fédérale de Lausanne et a été commandité par le Fond National Suisse sous le numéro de projet 21-52462.97.

Références

[1] F. Bacchus and P. van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *Proceedings of the 15th National Conference on Artificial Intelligence*, Madison, Wisconsin, July 98.

[2] F. Benhamou. Interval constraint logic programming. In *Constraint Programming: Basics and Trends*, volume 910, pages 1–21. Springer-Verlag, 95.

[3] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1):179–190, 94.

[4] C. Bessière and J.-C. Régin. MAC and combined heuristics: Two reasons to forsake FC(and CBJ?) on hard problems. In *CP96*, pages 61–75. CP, 96.

[5] P.A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *ECAI*, 92.

[6] A. Haselböck. Exploiting interchangeabilities in constraint satisfaction problems. In *Proceedings of IJCAI'93*, pages 282–287, 93.

[7] T. Hogg. Refining the phase transition in combinatorial search. *Artificial Intelligence*, 81(1-2):127–154, 96.

[8] P. D. Hubbe and E. C. Freuder. An efficient cross product representation of the constraint satisfaction problem search space. In *Proc. of AAAI-92*, pages 421–427. AAAI, July 92.

[9] N. Jussien and O. Lhomme. Dynamic domain splitting for numeric CSP. In *European Conference on Artificial Intelligence*, pages 224–228, 98.

[10] O. Lhomme. Consistency techniques for numeric CSPs. In *Proceedings of IJCAI'93*, pages 232–238, 93.

[11] O. Lhomme and M. Rueher. Application des techniques CSP au raisonnement sur les intervalles. *Revue d'intelligence artificielle*, 11(3):283–311, 97. Dunod.

[12] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings ECAI-94*, pages 125–129, 94.

[13] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Ways of maintaining arc consistency in search using Cartesian representation. In *Proc. of ERCIM'99*, 99.