Heuristiques exploitant la relaxation linéaire pour l'optimisation dans les réseaux de fonction de coût

Fulya Trösser[1*] **Simon de Givry**[1] **George Katsirelos**[2]

[1] MIAT, UR-875, INRA, F-31320 Castanet Tolosan, France
[2] UMR MIA-Paris, INRA, AgroParisTech, Université Paris-Saclay, 75005 Paris, France
{fulya.ural,simon.de-givry}@inra.fr gkatsi@gmail.com

Résumé

Les solveurs exacts pour le problème d'optimisation dans les modèles graphiques, comme par exemple les réseaux de fonctions de coût ou les champs aléatoires de Markov, utilisent généralement l'algorithme par séparation et évaluation. L'efficacité de la recherche dépend principalement de deux facteurs : la qualité de la borne calculée à chaque nœud de l'arbre de recherche et les heuristiques de branchement. Dans ce contexte, il y a un compromis à trouver entre la qualité de la borne et son coût de calcul. En particulier, l'algorithme de Cohérence d'Arc Virtuelle (*Virtual Arc Consistency or VAC*) produit des bornes de bonne qualité, mais avec un coût élevé, il est donc utilisé généralement pendant le prétraitement uniquement, au lieu de l'appliquer à chaque nœud de l'arbre de recherche.

Dans ce travail, nous identifions une faiblesse des solveurs par séparation et évaluation dans l'utilisation de VAC. C'est-à-dire qu'ils ignorent l'information sur la relaxation linéaire du problème que VAC fournit, à l'exception de la borne duale. Notamment, il se peut que l'heuristique de branchement prenne des décisions qui sont clairement inefficaces à la lumière de cette information. En éliminant ces décisions inefficaces, nous réduisons considérablement la taille de l'arbre de recherche. De plus, nous pouvons supposer de façon optimiste, que la relaxation linéaire est correcte en grande partie au niveau des affectations qu'elle effectue, ce qui aide à trouver des solutions de bonne qualité rapidement. La combinaison de ces méthodes montre une très bonne performance pour certaines familles d'instances et surpasse l'état de l'art.

Abstract

Exact solvers for optimization problems on graphical models, such as Cost Function Networks and Markov Random Fields, typically use branch-and-bound. Its efficiency relies mainly on two factors: the quality of the bound computed at each node of the branch-and-bound tree and the branching heuristics. In this respect, there is a trade-off between quality of the bound and computational cost. In particular, the Virtual Arc Consistency (VAC) algorithm computes high quality bounds but at a significant cost, so it is mostly used in preprocessing, rather than in every node of the search tree.

In this work, we identify a weakness in the use of VAC in branch-and-bound solvers, namely that they ignore the information that VAC produces on the linear relaxation of the problem, except for the dual bound. In particular, the branching heuristic may make decisions that are clearly ineffective in light of this information. By eliminating these ineffective decisions, we significantly reduce the size of the branch-and-bound tree. Moreover, we can optimistically assume that the relaxation is mostly correct in the assignments it makes, which helps find high quality solutions quickly. The combination of these methods shows great performance in some families of instances, outperforming the previous state of the art.

1 Introduction

Les modèles graphiques non dirigés, comme le problème de satisfaction de contraintes pondérées (*Weighted Constraint Satisfaction Problem or WCSP*) et les champs aléatoires de Markov (*Markov Random Field or MRF*), peuvent être utilisés afin de donner une représentation factorisée d'une fonction, dans lesquels les noeuds du graphe correspondent aux variables de la fonction et les (hyper-)arêtes correspondent aux facteurs. Les facteurs peuvent être, par exemple, des fonctions de coût, auquel cas le modèle graphique représente la factorisation d'une fonction de coût ; ou des tables de probabilités jointes, auquel cas le modèle représente une distribution globale de probabilité non normalisée.

*Papier doctorant : Fulya Trösser[1] est auteur principal.

Les deux modèles, WCSP et MRF, sont équivalents sous une transformation $-\log$, donc la tâche NP-difficile de minimisation de coût en WCSP est équivalente à la tâche d'affectation de probabilité maximum a posteriori en MRF. Ce problème d'optimisation a des applications diverses telles qu'en bioinformatique, en vision par ordinateur, *etc.*

Les méthodes de résolution exactes pour ce problème sont généralement basées sur l'algorithme par séparation et évaluation, sauf dans quelques exceptions notables. Par exemple, il est possible de modéliser l'optimisation d'un WCSP en tant qu'un problème linéaire en nombres entier (PLNE) et d'utiliser un solveur pour ce problème. Or, les solveurs PLNE doivent résoudre la relaxation linéaire de façon exacte afin d'obtenir un minorant à chaque noeud de l'arbre de recherche, ce qui est une opération très coûteuse en fonction de la taille des problèmes rencontrés dans de nombreuses applications. En revanche, les solveurs dédiés les plus performants utilisent des algorithmes qui résolvent la relaxation linéaire approximativement et donc potentiellement sous-optimalement. Spécifiquement, les algorithmes comme EDAC [9], VAC [7], TRWS [15] et d'autres, produisent des solutions faisables pour le dual de la relaxation linéaire du WCSP, qui peuvent être utilisées comme minorants. Ces algorithmes compensent la perte de précision en gagnant considérablement en efficacité de calcul. A l'inverse de la PLNE, non seulement la relaxation linéaire est approchée, mais c'est l'algorithme le plus faible, EDAC, qui est préféré durant la recherche, tandis que VAC ou TRWS sont principalement utilisés en prétraitement.

Parmi les exceptions à l'approche précédente, nous sommes intéressés par la méthode CombiLP [11], qui résout la relaxation linéaire et décompose le problème en deux parties : la partie "facile" correspond à l'ensemble des variables intégrales dans la relaxation et la partie combinatoire contient les variables restantes. Ceci est fait en identifiant une condition appelée cohérence d'arc stricte (*strict arc consistency or strictAC*) à partir de la solution duale produite par l'algorithme TRWS. La partie combinatoire est résolue de façon exacte par l'algorithme par séparation et évaluation, et si la solution peut être fusionnée avec la partie facile sans induire de coût supplémentaire, l'algorithme a prouvé l'optimalité. Sinon, il transfère certaines variables de la partie facile vers la partie combinatoire et réitère le processus.

Ici, nous faisons plusieurs contributions. Premièrement, nous améliorons la condition qu'utilise CombiLP afin de détecter l'intégralité. Nous montrons en Section 3 que cette condition est trop stricte et nous donnons une condition relâchée qui admet un ensemble plus grand de variables intégrales. Deuxièmement, nous montrons que l'on peut détecter un ensemble de variables intégrales de la relaxation à partir d'une solution duale réalisable sans avoir à favoriser la condition d'intégralité. Sur le plan pratique, nous introduisons deux heuristiques simples qui exploitent cette propriété au sein d'un algorithme par séparation et évaluation. La première heuristique présentée en Section 4 modifie l'heuristique de choix de branchement pour éviter de brancher sur les variables strictAC. La deuxième heuristique, présentée en Section 5, est une variante de l'heuristique RINS [8] connue en PLNE, qui fait l'hypothèse que les variables strictAC sont affectées à leur valeur intégrale dans la solution optimale et effectue une exploration arborescente dans le sous-problème des variables restantes dans le but de trouver un bon majorant. En Section 6, nous montrons que l'intégration de ces techniques dans le solveur toulbar2 [7] surpasse considérablement l'état de l'art pour certaines familles d'instances.

2 Préliminaires

Definition 1. Un *Problème de Satisfaction de Contraintes (CSP)* [6] est un triplet $\langle X, D, C\rangle$. $X = \{1, \ldots, n\}$ est un ensemble de n variables. Chaque variable $i \in X$ a un domaine de valeurs $D_i \in D$ et peut être affectée à n'importe quelle valeur $a \in D_i$, également notée (i, a). C est un ensemble de contraintes.

Chaque contrainte $c_S \in C$ est définie sur un ensemble de variables $S \subseteq X$ par un sous-ensemble du produit Cartésien $\ell(S) = \prod_{i \in S} D_i$ qui définit tous les tuples de valeurs cohérents vis à vis de la contrainte. Nous supposons, sans perte de généralité, qu'au plus une contrainte est définie sur un ensemble de variables donné. La contrainte unaire sur la variable i sera notée c_i, et les contraintes binaires c_{ij}. Par souci de simplification et en raison de limitations techniques de mise en oeuvre, nous nous focalisons sur des CSP binaires, c'est à dire pour lesquels chaque contrainte porte sur au plus deux variables. Le problème est de trouver une solution qui satisfait toutes les contraintes dans C. C'est un problème NP-complet.

Definition 2. Un CSP est *arc-cohérent* (AC) si $\forall c_{ij} \in C$, $\forall a \in D_i$, $\exists b \in D_j$ avec $\{(i, a), (j, b)\} \in c_{ij}$.

Definition 3. Un *Problème de Satisfaction de Contraintes Pondérées (WCSP)* [7] est un quadruplet $\langle X, D, C, k\rangle$ où X est un ensemble de n variables, chaque variable $i \in X$ a un domaine de valeurs possibles $D_i \in D$, comme pour un CSP. C est un ensemble de fonctions de coût et k est un entier positif ou infini qui sert de majorant.

Chaque fonction de coût $c_S \in C$ est définie sur un ensemble de variables $S \subseteq X$ et la fonction fait

correspondre à chaque affectation des variables de S un coût entier positif ou nul.

Nous supposons que tous les WCSPs contiennent une fonction de coût unaire pour chaque variable et une fonction de coût $c_\emptyset$, qui représente une constante dans la fonction objectif. Puisque tous les coûts sont positifs, cette constante est un minorant du coût des affectations admises par le WCSP.

Nous nous restreignons à des WCSP binaires, bien que toutes les définitions et algorithmes présentés par la suite puissent se généraliser à des fonctions d'arité supérieure.

Un WCSP binaire peut être représenté de façon graphique comme illustré en tête de la Figure 1. Chaque variable $i \in X$ correspond à une cellule. Chaque valeur $a \in D_i$ correspond à un point dans la cellule. Un coût unaire $c_i(a)$ est écrit à côté du point s'il est non nul. S'il existe une fonction de coût binaire c_{ij} ayant un coût non nul entre (i, a) et (j, b), alors une arête est mise entre les points qui correspondent à ces affectations.

Le problème est de trouver une solution t qui minimise la somme de toutes les fonctions de coût dans C, notée $c_P(t) = c_\emptyset + \sum_{i \in X} c_i(t[i]) + \sum_{c_{ij} \in C} c_{ij}(t[i], t[j])$, et qui soit de coût total inférieur à k. C'est un problème NP-difficile.

Étant donné deux WCSPs $P = (X, D, C, k)$ et $P' = (X, D, C', k)$ tels que $\forall c_S \in C, \exists c'_S \in C'$ et vice versa, on dit qu'ils ont la même structure. Si $c_P(t) = c_{P'}(t)\ \forall t \in \ell(X)$, alors P et P' sont équivalents et ils sont des reparamétrisations de l'un à l'autre. Il a été montré dans [7] que la reparamétrisation optimale qui maximise le facteur constant $c_\emptyset$ correspond au dual du programme linéaire ci-dessous, appelé le polytope local du WCSP binaire :

$$\min \sum_{i \in X, a \in D_i} c_i(a)\, x_{i,a} + \sum_{c_{ij} \in C, a \in D_i, b \in D_j} c_{ij}(a,b)\, y_{i,a,j,b}$$

s.t.

$$\sum_{a \in D_i} x_{i,a} = 1 \qquad i \in X$$

$$x_{i,a} = \sum_{b \in D_j} y_{i,a,j,b} \qquad i \in X, a \in D_i, c_{ij} \in C$$

$$0 \le x_{i,a} \le 1 \qquad i \in X, a \in D_i$$

$$0 \le y_{i,a,j,b} \le 1 \qquad c_{ij} \in C, a \in D_i, b \in D_j \quad (1)$$

La reparamétrisation est extraite à partir des coûts réduits des variables $x_{i,a}$ et $y_{i,a,j,b}$ dans la solution optimale du PL ci-dessus et en ajoutant à $c_\emptyset$ l'optimum de ce PL.

Definition 4 ([7])**.** Soit le WCSP binaire $P = \langle X, D, C, k\rangle$. On note le CSP $Bool(P) = \langle X, \overline{D}, \overline{C}\rangle$ (resp. $Bool_\theta(P) = \langle X, \overline{D_\theta}, \overline{C_\theta}\rangle$) tel que pour tout $i \in X$, $a \in \overline{D}$ (resp. $\overline{D_\theta}$) si et seulement si $\exists a \in D_i, c_i(a) = 0$ (resp. $c_i(a) < \theta$) et pour tout $i, j \in X^2$, $r_{ij} \in \overline{C}$ (resp. $\overline{C_\theta}$) si et seulement si $\exists c_{ij} \in C$, avec r_{ij} la relation $\forall a \in D_i, b \in D_j, \{(i,a),(j,b)\} \in r_{ij} \Leftrightarrow c_{ij}(a,b) = 0$ (resp. $c_{ij}(a,b) < \theta$).

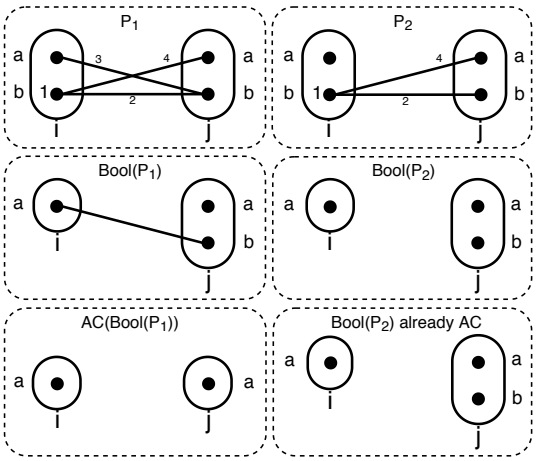


FIGURE 1 – Deux instances WCSP P_1, P_2, leurs transformations correspondantes en CSP par $Bool(P)$ et leurs fermetures arc-cohérentes de $Bool(P)$.

Un exemple du $Bool(P)$ est illustré dans la Figure 1. Ici une arête indique un tuple interdit.

Definition 5. Un WCSP P est *virtuellement arc-cohérent* (VAC) si la fermeture arc-cohérente de $Bool(P)$ n'est pas vide [7].

Dans l'exemple, les deux fermetures sont arc-cohérentes, avec la valeur b supprimée du domaine de $\overline{D_j}$ dans la fermeture de $Bool(P_1)$.

L'algorithme VAC construit itérativement $Bool(P)$ et si sa fermeture par cohérence d'arc est vide, il extrait une reparamétrisation améliorant le minorant [1]. Il termine lorsque la fermeture par cohérence d'arc n'est pas vide. Il converge ainsi vers un point-fixe qui n'est pas unique et qui peut être inférieur à l'optimum du PL 1.

Par la suite, nous supposons que $\forall c_S \in C$, $S \neq \emptyset$, $\min_{t \in \ell(S)} c_S(t) = 0$, ou sinon l'instance peut être triviallement reparamétrée pour augmenter le minorant.

3 Cohérence d'arc stricte

La cohérence d'arc stricte (*strictAC*) [18] est une façon de partitionner un WCSP en une partie "facile", qui

1. Le fait de garder des coûts entiers peut conduire à ne pas améliorer le minorant, posant un problème de terminaison de l'algorithme. Ceci est réglé en augmentant localement à une variable ou à une fonction de coût le seuil θ [7].

est résolue de façon exacte par un solveur de programmation linéaire et une partie combinatoire "difficile" résolue par un solveur d'optimisation combinatoire.

Definition 6 (Cohérence d'arc stricte [18]). *Une variable $i \in X$ est arc-cohérente stricte s'il existe une valeur unique $a \in D_i$ telle que $c_i(a) = 0$ et $\forall c_{ij} \in C$, il existe un tuple unique $\{(i,a),(j,b)\}$ qui satisfait $c_{ij}(a,b) = 0$ parmi tous les tuples possibles de la fonction de coût. La valeur a est appelée la valeur* strict AC *de i.*

Étant donné un WCSP P et un sous-ensemble S de ses variables tel que toutes les variables dans S sont strict AC, nous pouvons résoudre P restreint à S de manière exacte en affectant la valeur strict AC à chaque variable. Cette affectation partielle aura un coût nul. Cette propriété donne une partition naturelle d'un WCSP en l'ensemble des variables strict AC et le reste. Cette bi-partition a été utilisée dans [18] et dans un algorithme raffiné qui a été introduit plus tard [11]. Ces algorithmes exploitent la solvabilité du sous-ensemble des variables strict AC et doivent résoudre seulement la partie plus petite composée des variables non strict AC avec un solveur combinatoire. Une affectation complète est obtenue en combinant les solutions trouvées pour les deux parties. Si les coûts entre les deux parties sont nuls alors l'optimalité est prouvée, sinon les variables strict AC impliquées dans ces coûts non nuls sont ajoutées à la partie combinatoire qui est résolue à nouveau jusqu'à obtenir une preuve d'optimalité du problème global.

Notre première contribution ici est de noter que la propriété strict AC peut être plus forte que nécessaire. En particulier, nous pouvons modifier la deuxième condition comme ci-dessous :

Definition 7 (Cohérence d'arc stricte revisitée strict* AC). *Une variable $i \in X$ est strict* AC s'il existe une valeur unique $a \in D_i$ telle que $c_i(a) = 0$ et $\forall c_{ij} \in C$, il existe au moins un tuple $\{(i,a),(j,b)\}$ qui satisfait $c_{ij}(a,b) + c_j(b) = 0$. La valeur a est appelée la valeur strict* AC de i.*

La différence entre strict* AC et strict AC est que pour strict* AC, la seconde condition nécessite que la valeur témoin apparaisse dans au moins un tuple de coût nul et non dans un unique tuple pour chaque fonction de coût incidente. En contre partie, nous imposons que le coût unaire du support sur l'autre variable doit aussi être nul ($c_j(b) = 0$). Cela permet de reconstruire une affectation partielle de coût nul pour les variables strict* AC en affectant simplement ces variables à leur valeur strict* AC, à l'instar de la propriété strict AC.

Il est intéressant de noter que l'existence d'un tuple unique $\{(i,a),(j,b)\}$ de coût nul pour la propriété strict AC implique que les coûts unaires associés $c_i(a), c_j(b)$ sont nuls également dans le cas où les deux variables i et j sont strict AC. Si toutes les variables voisines de i et j sont aussi strict AC alors les variables i et j sont également strict* AC. L'inverse n'est pas forcément vrai. Par contre l'ensemble des variables à la frontière, *i.e.,* incidentes à des variables de la partie combinatoire, est incomparable d'une propriété à l'autre.

Dans l'exemple du WCSP P_1 en Figure 1, la variable i est strict AC et strict* AC, avec pour valeur témoin la valeur a. Dans le cas de P_2, la variable i est strict* AC mais pas strict AC.

Une difficulté présente pour les deux propriétés, strict AC et strict* AC, est qu'un même minorant $c_\emptyset$ peut être produit par différentes reparamétrisations issues de plusieurs solutions duales du polytope local n'induisant pas le même ensemble de variables strict* AC. Une façon d'y remédier est d'inciter le solveur de programmation linéaire vers des solutions qui maximisent l'ensemble strict* AC [11]. Ici nous proposons une autre méthode, à partir de l'observation suivante.

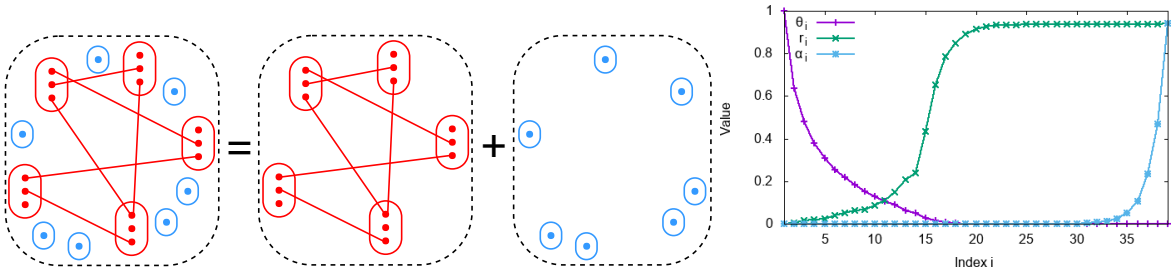
Proposition 1. *Soit une instance WCSP P qui est VAC et une variable i. S'il existe une unique valeur $a \in \overline{D}_i$ dans $Bool(P)$, alors la variable i est strict* AC avec pour valeur témoin a.*

D'après la définition de VAC et $Bool(P)$, nous savons que $Bool(P)$ est arc-cohérent et si une seule valeur reste dans le domaine de i dans $Bool(P)$, elle a nécessairement un coût nul dans P et un support de coût nul dans toutes les variables voisines. A contrario, si une valeur est supprimée dans $Bool(P)$, soit elle a un coût non nul dans P, soit il est possible de déplacer un coût non nul vers cette valeur [7]. Voir l'exemple en Figure 1 où nous montrons la fermeture par cohérence d'arc de $Bool(P)$ pour deux instances. Ici, les variables i et j sont strict* AC dans la fermeture $Bool(P_1)$.

Cette observation nous permet de construire un ensemble de variables strict* AC plus grand que celui obtenu en vérifiant la propriété pour une reparamétrisation particulière. Ainsi nous n'avons pas à modifier le solveur de programmation linéaire pour favoriser des solutions duales spécifiques et c'est plus facile à utiliser au sein de l'algorithme VAC qui maintient la fermeture arc-cohérente de $Bool(P)$ explicitement.

Complexité. Il est naturel de se demander si la présence d'un grand ensemble de variables strict* AC rend le problème plus facile à résoudre au sens de la complexité paramétrée [10]. Malheureusement, ce n'est pas le cas. Soit Almost-S-AC-WCSP la classe des WCSPs qui sont VAC avec $n-1$ strict* AC variables.

Theorème 1. Almost-S-AC-WCSP *est NP-complet.*



(a) Partition des variables en deux groupes : difficile et facile.

(b) Évolution de θ_i, r_i et α_i en fonction du nombre d'itérations.

FIGURE 2 – strict* AC et l'heuristique fondée sur RINS.

Démonstration. L'appartenance à NP est triviale puisque c'est une sous-classe de WCSP. Pour la complétude, nous effectuons une réduction à partir d'un WCSP binaire. Soit $P = \langle X, D, C, k\rangle$ une instance WCSP arbitraire. Nous construisons une instance $P'\langle X \cup X' \cup \{q\}, D, C', k\rangle$ de ALMOST-S-AC-WCSP, où X' est une copie des variables de X, avec les mêmes coûts unaires, et q est une variable supplémentaire de domaine $\{a, b\}$ avec $c_q(a) = c_q(b) = 0$. Pour chaque fonction de coût $c_{ij} \in C$, P' a deux fonctions de coût ternaires portant sur $\{i, j, q\}$ et sur $\{i', j', q\}$.

Chaque variable de P a au moins une valeur de coût unaire nul puisque P est VAC. Sans perte de généralité, supposons que cette valeur est a pour toutes les variables. Nous définissons les fonctions de coût ternaires sur les variables de X par $c_{ijq}(a, a, a) = 0$, $c_{ijq}(u, v, a) = k$ quand $u \neq a$ ou $v \neq a$, $c_{ijq}(u, v, b) = c_{ij}(u, v) + 1\forall u, v$. De même, $c_{i'j'q}(a, a, b) = 0$, $c_{i'j'q}(u, v, b) = k$ quand $u \neq a$ ou $v \neq a$, $c_{i'j'q}(u, v, a) = c_{ij}(u, v) + 1\forall u, v$.

P' est une instance ALMOST-S-AC-WCSP avec q la variable non-strict* AC. En effet, $c_i(a) = 0$ pour toutes les variables $i \in X \cup X' \cup \{q\}$ et chaque variable $i \in X$ (resp. $i \in X'$) est supportée par l'unique tuple de coût nul (a, a, a) (resp. (a, a, b)) dans chaque fonction de coût ternaire. Toutes les autres valeurs appartiennent à des tuples de fonctions ternaires ayant un coût non nul et seront supprimées dans la fermeture par cohérence d'arc de $Bool(P')$.

P a une solution optimale de coût c si et seulement si P' a une solution optimale de coût $c + |C|$. En effet, quand nous affectons q à a ou b, le problème se décompose en deux WCSPs binaires indépendants sur X et sur X'. L'un deux admet une solution que de a de coût nul et l'autre est identique à P avec un coût supplémentaire de 1 par fonction de coût. □

Bien que cette construction utilise des fonctions de coût ternaires, nous pouvons convertir celles-ci en fonctions binaires en utilisant l'encodage caché [3] qui préserve la cohérence d'arc et donc aussi VAC, démontrant ainsi le théorème dans le cas de WCSPs binaires.

4 Heuristique de choix de variable

Dans un algorithme de recherche par séparation et évaluation, l'ordre dans lequel les variables sont affectées a un impact crucial sur les performances. En général, une décision de branchement doit aider le solveur à élaguer rapidement des sous-arbres qui ne contiennent pas de solutions meilleures, en créant des sous-problèmes ayant un minorant augmenté dans toutes les branches [1].

A partir de cette observation et de la connexion entre strict* AC et l'intégralité expliquée en Section 3, nous observons que brancher sur une variable strict* AC i créera une branche qui contiendra la valeur strict* AC a de i. Puisque a est la seule valeur du domaine de i dans $Bool(P)$ et que son coût unaire ne change pas lors du branchement, $Bool(P\ |_{i=a})$ est identique à $Bool(P)$, ainsi le minorant ne sera pas augmenté dans cette branche. Il semble donc préférable de ne pas brancher sur les variables strict* AC.

Pour implanter ceci, nous partitionnons les variables suivant la taille de leur domaine dans la fermeture arc-cohérente de $Bool(P)$: celles ayant un domaine singleton correspondent aux variables faciles en bleu sur la Figure 2a et le reste correspond aux variables difficiles en rouge.

Nous donnons alors la priorité aux variables ayant plus d'une valeur, *i.e.,* les variables en rouge, pour le branchement. Le choix parmi ces variables est fait en utilisant l'heuristique par défaut dans le solveur. Dans le cas de TOULBAR2, utilisé dans nos expérimentations, il s'agit de l'heuristique DOM+WDEG [4] combinée avec l'heuristique last conflict [16].

5 Heuristique de production de majorants

Une approche appelée *recherche dans le voisinage induit par la relaxation* (Relaxation Induced Neighborhood Search or RINS) [8] consiste à construire un

sous-problème prometteur en utilisant l'information contenue dans la relaxation continue d'un modèle de programmation linéaire à variables entières (PLNE). L'exploration du sous-problème est formulée à nouveau comme un modèle PLNE et résolu à son tour avec une limite de temps [2]. Pour cela, le sous-problème est construit en affectant les variables qui ont la même valeur dans la meilleure solution trouvée et dans la relaxation courante.

En suivant cette approche, nous proposons une heuristique de production de majorants qui s'exécute en prétraitement. Dans ce cas, nous n'avons pas encore trouvé de meilleure solution et nous affectons seulement les variables qui sont intégrales dans la relaxation : affectation des variables strict* AC à leur valeur témoin et suppression des valeurs dans le reste des variables qui ont été supprimées dans la fermeture par cohérence d'arc de $Bool(P)$. De plus, comme nous allons l'expliquer ensuite, nous exploitons la fermeture par cohérence d'arc construite par des itérations intermédiaires de VAC, en examinant $Bool_\theta(P)$ pour une valeur appropriée de θ.

Cette heuristique RINS peut également s'appliquer durant la recherche à une fréquence donnée, comme par exemple tous les f noeuds de l'arbre de recherche [3]. Dans ce cas, les variables strict* AC sont affectées à leur valeur témoin si et seulement si leur valeur strict* AC est la même que dans la meilleure solution trouvée, sinon nous conservons ces deux valeurs dans le domaine de la variable, à l'instar de l'approche RINS [8].

De la même façon, nous rajoutons dans le domaine des variables non-strict* AC la valeur présente dans la meilleure solution trouvée après avoir réalisé le filtrage des domaines par maintien de cohérence d'arc sur $Bool_\theta(P)$.

Nous avons observé que la qualité des majorants trouvés par cette heuristique dépend fortement de la valeur du seuil θ dans la construction de $Bool_\theta(P)$. Rappelons d'abord la façon dont est implanté l'algorithme VAC_θ dans le solveur TOULBAR2 [7]. Cette implémentation ne traite que les WCSP binaires. Premièrement, tous les coûts binaires non nuls $c_{ij}(a,b)$ sont triés par ordre décroissant et repartis dans un nombre fixe de g sous-ensembles de coûts. Les coûts minimums θ_i de chaque sous-ensemble $i \in \{1,\ldots,g\}$ définissent une séquence de seuils $(\theta_1,\ldots,\theta_g)$. L'algorithme VAC_θ commence avec le seuil le plus grand θ_1 et itère à ce seuil fixe ses étapes de reparamétrisation jusqu'à ce que le problème $Bool_{\theta_1}(P)$ soit arc-cohérent. Puis il recommence avec le seuil suivant θ_{i+1}. Après avoir terminé avec le seuil θ_g, l'algorithme suit une suite géométrique décroissante avec $\theta_{i+1} = \frac{\theta_i}{2}$ jusqu'à atteindre $\theta_i = 1$.

Plus θ_i est petit, plus $Bool_{\theta_i}(P)$ est restreint, *i.e.,* les domaines sont réduits. Cela conduit globalement à un ensemble de variables strict* AC globalement croissant suivant les itérations i de VAC_θ. L'observation que nous faisons de manière informelle est que cet ensemble arrête de croître et semble saturer à partir d'un certain seuil, souvent avant d'atteindre $\theta_i = 1$. Cependant même après ce seuil de saturation, l'algorithme continue à réduire les domaines des variables non-strict* AC. Notre idée est de choisir un seuil θ où le nombre de variables strict* AC est maximisé et où les domaines des variables non-strict* AC sont suffisamment réduits mais pas trop. De cette manière, nous augmentons la taille de la partie facile et diminuons celle de la partie difficile, mais en même temps, nous ne restreignons pas trop les domaines des variables non-strict* AC par rapport au problème original.

En Figure 2b, nous montrons l'évolution du seuil normalisé $\frac{\theta_i}{\theta_1}$, le pourcentage r_i de variables strict* AC et le rapport de ces deux courbes $\alpha_i = \frac{r_i \theta_1}{\theta_i}$ en fonction des itérations allant de $i = 1$ à $i = 39$ pour l'instance cnd1threeL1_1228061 du benchmark Worms. Ici, r_i varie de 0 pour $\theta_1 = 705.867.889$ à 0.94 pour $\theta_{39} = 1$.

La courbe montrant l'évolution du rapport α en Figure 2b est utilisée pour fixer le seuil θ dans notre heuristique RINS. L'idée est que lorsque le nombre de variables strict* AC sature, le seuil θ_i normalisé continue à diminuer et donc le rapport α augmente de manière plus rapide, c'est à ce moment que nous identifions le bon seuil au point d'inflexion. En pratique, nous sélectionnons le seuil θ_i lorsque l'angle de la courbe α atteint 10 degrés. Sur notre exemple, cela survient à l'itération 32, avec $\theta_{32} = 627$.

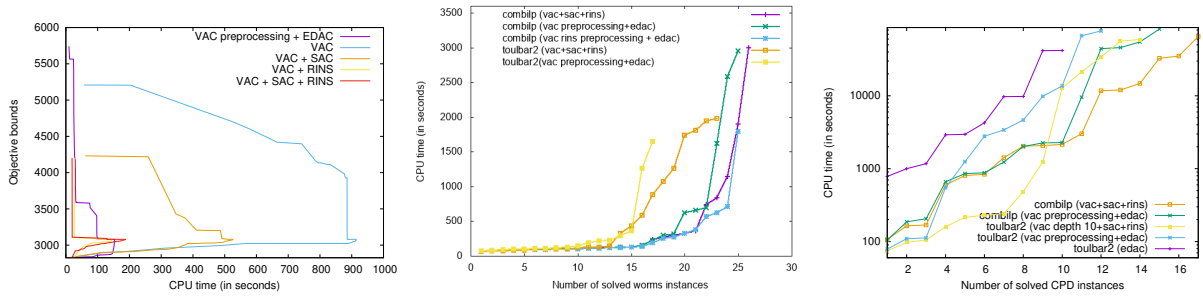
6 Résultats expérimentaux

Nous avons implantés les heuristiques strict* AC et RINS dans TOULBAR2 (github.com/toulbar2/toulbar2 version 1.0.1 avec les options par défaut), un solveur WCSP open-source écrit en C++ et ayant remporté plusieurs compétitions sur les modèles graphiques [4]. Les calculs ont été réalisés sur un seul coeur d'un processeur Intel Xeon à 2.1 GHz et 8 Go de mémoire vive avec un temps limite d'une heure (sauf pour les instances CPD, temps limite fixé à une journée).

6.1 Description des benchmarks

Les expérimentations ont porté sur des benchmarks de modèles graphiques probabilistes et déterministes issus de différentes communautés. D'abord, nous avons

2. Dans les expérimentations, nous limitons la recherche à 1000 retours-arrières.
3. Dans les expérimentations, $f = 100$.
4. www.inra.fr/mia/T/toulbar2



(a) CELAR graph11 instance. (b) Worms instances. (c) CPD instances.

FIGURE 3 – Comportement des bornes au cours du temps et nombre d'instances entièrement résolues (optimum et preuve d'optimalité) pour un temps de calcul donné sur les familles de problèmes CELAR, Worms et CPD.

considéré une large collection de 3016 instances [12] [5] pour lesquelles nous avons sélectionné les instances binaires. Cela inclut 251 instances MRF des compétitions *UAI 2008 Evaluation* et *Probabilistic Inference Challenge* (PIC 2011) [6], 853 instances en analyse d'images (Computer Vision and Pattern Recognition or CVPR) du benchmark OpenGM2 [7] [14], 503 instances MaxCSP de la compétition *CSP 2008 Competition* [8] et 251 instances d'une librairie de WCSPs (aka, Cost Function Network or CFN) [9].

Nous avons aussi collecté 30 instances Worms [13] [10] où COMBILP est l'état de l'art [11] et 21 instances de conception de protéines (Computational Protein Design or CPD) où TOULBAR2 est l'état de l'art [17] [11].

6.2 Analyse des résultats

Dans la Table 1, nous présentons les résultats par famille d'instance en nombre d'instances résolues et en temps CPU moyenné sur les instances résolues. Nous comparons différentes versions de TOULBAR2 avec :

- VAC en prétraitement et EDAC durant la recherche avec l'heuristique de choix de variable par défaut DOM+WDEG (VACpreprocessing+EDAC)
- VAC et RINS en prétraitement et EDAC durant la recherche avec l'heuristique de choix de variable par défaut (VAC+RINSpreprocessing+EDAC)
- VAC en prétraitement et durant la recherche avec l'heuristique de choix de variable par défaut (VAC)
- VAC en prétraitement et durant la recherche avec l'heuristique de choix de variables exploitant strict* AC (VAC+sAC)
- VAC et RINS en prétraitement et VAC seul durant la recherche avec l'heuristique de choix de variables exploitant strict* AC (RINSpreprocessing+VAC+sAC)
- VAC et RINS en prétraitement et durant la recherche avec l'heuristique de choix de variables exploitant strict* AC (VAC+sAC+RINS)

Bien que l'approche VAC en prétraitement et EDAC durant la recherche soit souvent la meilleure option, nous observons des gains de performance avec nos heuristiques sur les familles Worms, CPD, EHI et QCP. L'écart de performance entre VAC+RINS et/ou strict* AC et EDAC vient principalement du temps mis pour maintenir VAC durant la recherche. Si l'on se compare à VAC durant la recherche, alors l'apport de nos heuristiques est le plus souvent positif. En particulier le nombre de noeuds de l'arbre de recherche de VAC+sAC ou VAC+sAC+RINS comparé à VAC durant la recherche est en moyenne réduit sur toutes les familles sauf CFN/Auction, MaxCSP/BlackHole, Composed, Geometric, QCP et MRF/DBN (résultats non communiqués par faute de place). Cependant, dans ces cas défavorables, l'augmentation du nombre de noeuds est inférieur à 40%. Pour le reste des benchmarks, ce nombre est réduit de quelque pourcent à plusieurs ordres de magnitude (CVPR/ColorSeg, MRF/ImageAlignment, ProteinFolding, Segmentation et Worms).

Pour les instances CELAR et ProteinDesign, quand VAC est utilisé en prétraitement, l'ajout de RINS améliore considérablement les temps de résolution. Si nous ajoutons strict* AC et RINS avec VAC durant la recherche, nous pouvons surpasser VAC en prétraitement + EDAC. Voir Figure 3a, l'évolution des bornes au cours du temps pour l'instance CELAR graph11 [5].

Nous avons également comparé TOULBAR2 et COMBILP (qui utilise la même version de TOULBAR2 pour son solveur PLNE interne) avec différentes cohérences

5. genoweb.toulouse.inra.fr/~degivry/evalgm
6. graphmod.ics.uci.edu/uai08/Evaluation/Report/Benchmarks, www.cs.huji.ac.il/project/PASCAL
7. hci.iwr.uni-heidelberg.de/opengm2
8. www.cril.univ-artois.fr/CPAI08
9. forgemia.inra.fr/thomas.schiex/cost-function-library
10. genoweb.toulouse.inra.fr/~degivry/evalgm/CVPR/Worms_uai.tar.xz
11. genoweb.toulouse.inra.fr/~degivry/evalgm/CFN/ProteinDesignUAI2017_wcsp_xz.zip

locales, montrant les améliorations apportées par l'exploitation des heuristiques strict* AC et RINS.

En Figure 3b, nous donnons les résultats pour le benchmark Worms qui est composé de 30 instances. CombiLP résout 25 instances d'après [11] en 1 heure de temps CPU. Ici, nous comparons CombiLP avec les paramètres pris dans [11] (VAC en prétraitement et EDAC durant la recherche) et utilisant notre version de toulbar2. De plus, nous testons l'apport des heuristiques strict* AC et RINS. toulbar2 seul résout 23 instances. Cependant, en mettant notre version de toulbar2 dans CombiLP, nous avons résolu 26 instances, soit une de plus que dans [11]. Bien qu'il soit coûteux de maintenir VAC durant la recherche, c'est la meilleure approche obtenue pour ce problème. En effet, la réduction de la taille de l'arbre de recherche surpasse le coût de maintenir VAC.

La même amélioration est observée pour le benchmark CPD. En Figure 3c, CombiLP utilisant VAC durant la recherche avec strict* AC et RINS résout 17 instances parmi les 21, au lieu de 15 seulement pour CombiLP utilisant VAC en prétraitement + EDAC. Indépendemment de CombiLP, toulbar2 par défaut avec VAC en prétraitement en résout seulement 12. Pour pouvoir exploiter VAC durant la recherche en dehors de CombiLP, sans avoir un coût excessif en temps de calcul pour maintenir VAC, nous avons dû limiter l'application de VAC uniquement lors de l'expansion des noeuds ouverts dans l'algorithme Hybrid-Best First Search utilisé par défaut dans toulbar2 [2] et seulement à une profondeur de recherche inférieure à 10. Dans ce cas, toulbar2 exploitant partiellement VAC avec l'ajout des heuristiques strict* AC et RINS résout 14 instances. Plusieurs instances de taille intermédiaire sont résolues avec un gain en temps d'un ordre de grandeur par rapport à toulbar2 par défaut ou à CombiLP. Pour comparaisons, dans [17], toulbar2 n'utilisant ni VAC ni nos heuristiques n'a résolu que 10 instances avec un temps limite de quatre jours.

7 Conclusions

Nous avons revisité la propriété Strict AC de telle façon qu'elle soit plus simple à utiliser au sein d'un algorithme par séparation et évaluation en conjonction avec l'algorithme VAC. Cette nouvelle propriété intègre des informations calculées par VAC sur la relaxation du problème pour l'utiliser dans des heuristiques.

Nous avons présenté deux heuristiques qui exploitent cette information, l'une pour le choix de la prochaine variable à brancher et l'autre pour trouver des bons majorants. Une évaluation expérimentale montre que ces heuristiques peuvent améliorer l'état de l'art sur certaines familles d'instances.

Remerciements.

Ce travail a été en partie financé par l'Agence nationale de la Recherche, référence ANR-16-C40-0028. Nous sommes reconnaissant à la plateforme bioinformatique Genotoul de Toulouse pour l'accès au cluster.

Références

[1] T Achterberg : Scip : solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[2] D Allouche, S de Givry, G Katsirelos, T Schiex et M Zytnicki : Anytime Hybrid Best-First Search with Tree Decomposition for Weighted CSP. *In Proc. of CP-15*, pages 12–28, Cork, Ireland, 2015.

[3] Fahiem Bacchus, Xinguang Chen, Peter van Beek et Toby Walsh : Binary vs. non-binary constraints. *Artif. Intell.*, 140(1/2):1–37, 2002.

[4] F Boussemart, F Hemery, C Lecoutre et L Sais : Boosting systematic search by weighting constraints. *In Proc. of ECAI-04*, pages 146–150, Valencia, Spain, 2004.

[5] B Cabon, S de Givry, L Lobjois, T Schiex et JP Warners : Radio Link Frequency Assignment. *Constraints*, 4(1):79–89, 1999.

[6] M Cooper et T Schiex : Arc consistency for soft constraints. *Artificial Intelligence*, 154(1-2):199–227, 2004.

[7] M C Cooper, S De Givry, M Sánchez, T Schiex, M Zytnicki et T Werner : Soft arc consistency revisited. *Artificial Intelligence*, 174(7-8):449–478, 2010.

[8] E Danna, E Rothberg et C Le Pape : Exploring relaxation induced neighborhoods to improve mip solutions. *Mathematical Programming*, 102(1):71–90, 2005.

[9] S de Givry, M Zytnicki, F Heras et J Larrosa : Existential arc consistency : getting closer to full arc consistency in weighted CSPs. *In Proc. of IJCAI-05*, pages 84–89, Edinburgh, Scotland, 2005.

[10] R G. Downey et M R. Fellows : *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013.

[11] S Haller, P Swoboda et B Savchynskyy : Exact map-inference by confining combinatorial search with LP relaxation. *In Proc. of AAAI-18*, pages 6581–6588, New Orleans, Louisiana, USA, 2018.

[12] B Hurley, B O'Sullivan, D Allouche, G Katsirelos, T Schiex, M Zytnicki et S de Givry : Multi-Language Evaluation of Exact Solvers in Graphical Model Discrete Optimization. *Constraints*, 21(3):413–434, 2016.

[13] D Kainmueller, F Jug, C Rother et G Myers : Active graph matching for automatic joint segmentation and annotation of c. elegans. *In Medical Image Computing and Computer-Assisted Intervention (MICCAI-14)*, pages 81–88, Boston, USA, 2014.

[14] J Kappes, B Andres, F Hamprecht, C Schnörr, S Nowozin, D Batra, S Kim, B Kausler, T Kröger, J Lellmann, N Komodakis, B Savchynskyy et C Rother : A comparative study of modern inference techniques for structured discrete energy minimization problems. *Int. J. of Computer Vision*, 115(2):155–184, 2015.

[15] V Kolmogorov : Convergent tree-reweighted message passing for energy minimization. *IEEE transactions on pattern analysis and machine intelligence*, 28(10):1568–1583, 2006.

[16] C Lecoutre, L Sais, S Tabary et V Vidal : Reasoning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.

[17] A Ouali, D Allouche, S de Givry, S Loudni, Y Lebbah, F Eckhardt et L Loukil : Iterative Decomposition Guided Variable Neighborhood Search for Graphical Model Energy Minimization. *In Proc. of UAI-17*, pages 550–559, Sydney, Australia, 2017.

[18] B Savchynskyy, JH Kappes, P Swoboda et C Schnörr : Global map-optimality by shrinking the combinatorial search area with convex relaxation. *In Proc. of NIPS-13*, pages 1950–1958, Lake Tahoe, Nevada, USA, 2013.

Problem/s/d	VAC preprocessing + EDAC	VAC and RINS preprocessing + EDAC	VAC	VAC + sAC	RINS preprocessing + VAC + sAC	VAC + sAC + RINS
RFC/251/300	248 (92.40)	246 (97.79)	242 (100.94)	243 (116.87)	244 (140.95)	242 (126.01)
Auction/170/2	170 (34.77)	170 (34.78)	170 (50.04)	170 (68.45)	170 (68.00)	170 (68.98)
CELAR/16/44	14 (345.35)	13 (72.85)	13 (524.02)	14 (416.11)	13 (276.99)	13 (469.92)
ProteinDesign/10/198	9 (4.90)	9 (1.89)	9 (12.21)	9 (9.08)	9 (2.59)	9 (4.50)
Warehouse/55/300	55 (220.45)	54 (318.16)	50 (179.95)	50 (217.09)	52 (369.37)	50 (252.37)
CVPR/853/20	742 (42.25)	742 (35.45)	724 (6.35)	743 (40.40)	742 (34.59)	742 (38.03)
ColorSeg/21/12	17 (1322.43)	17 (1086.87)	3 (1198.70)	17 (1244.88)	17 (1093.99)	17 1142.04
InPainting/4/4	2 (1083.39)	2 (983.25)	2 (419.48)	3 (952.89)	2 (520.26)	2 (1226.70)
Matching/4/20	4 (3.46)	4 (4.50)	4 (28.21)	4 (32.66)	4 (42.54)	4 (39.72)
ObjectSeg/5/8	4 (1658.67)	4 (1447.96)	0 (-)	4 (1454.08)	4 (1452.63)	4 (1533.74)
SceneDecomp/715/8	715 (0.07)	715 (0.07)	715 (0.07)	715 (0.07)	715 (0.07)	715 (0.07)
MaxCSP/503/50	419 (139.75)	419 (138.91)	422 (217.74)	421 (196.67)	421 (196.54)	419 (190.64)
BlackHole/37/50	10 (0.08)	10 (0.08)	10 (0.06)	10 (0.13)	10 (0.13)	10 (0.15)
Coloring/22/6	17 (7.07)	17 (6.80)	17 (11.45)	17 (11.06)	17 (10.90)	17 (11.78)
Composed/80/10	80 (0.12)	80 (0.12)	80 (0.83)	80 (1.44)	80 (1.42)	80 (1.44)
EHI/200/7	198 (231.73)	198 (229.34)	200 (344.84)	200 (296.08)	200 (298.64)	200 (297.25)
Geometric/100/20	96 (84.64)	96 (85.24)	91 (115.43)	91 (143.44)	91 (139.96)	91 (183.05)
Langford/4/29	2 (0.36)	2 (0.35)	2 (0.47)	2 (1.29)	2 (1.12)	2 (1.15)
QCP/60/9	16 (275.93)	16 (280.32)	22 (552.39)	21 (486.64)	21 (474.97)	19 (181.72)
MRF/297/503	213 (37.32)	213 (38.26)	209 (56.55)	208 (36.60)	209 (49.79)	209 (48.29)
DBN/108/2	82 (90.22)	82 (93.86)	78 (130.98)	77 (87.97)	78 (126.87)	78 (123.00)
ImageAlignment/10/93	10 (2.46)	10 (2.28)	10 (4.28)	10 (2.39)	10 (2.39)	10 (2.28)
ProteinFolding/21/503	21 (23.91)	21 (19.29)	21 (72.62)	21 (37.65)	21 (21.93)	21 (21.52)
Segmentation/100/21	100 (0.24)	100 (0.25)	100 (0.35)	100 (0.24)	100 (0.25)	100 (0.25)
Worms/30/128	17 (308.14)	19 (472.92)	14 (606.06)	22 (668.48)	23 (579.13)	23 (585.56)

TABLE 1 – Nombre d'instances WCSP binaires résolues en moins d'une heure. En parenthèses, temps CPU en secondes pour les instances résolues. La colonne 1 contient le nom des familles d'instances suivi du nombre d'instances par famille (s) et de la taille du plus grand domaine (d). Les familles d'instances CVPR/ChineseChars/100/2, MatchingStereo/2/20, PhotoMontage/2/7, MRF/Grid/21/2 et ObjectDetection/37/21 pour lesquelles aucune instance n'est résolue quelque soit la méthode ont été retirées de cette table.