Focus sur les heuristiques basées sur la pondération de contraintes

Hugues Wattez[1*] **Frédéric Koriche**[1] **Christophe Lecoutre**[1]
Anastasia Paparrizou[1] **Sébastien Tabary**[1]

[1] CRIL, Univ. Artois & CNRS, 62300 Lens, France
{wattez,koriche,lecoutre,paparrizou,tabary}@cril.fr

Résumé

La recherche avec retour-arrière est une approche complète et traditionnellement utilisée pour la résolution de problèmes de satisfaction et d'optimisation de contraintes. Il est bien connu que l'espace exploré durant la recherche peut drastiquement fluctuer selon l'ordre dans lequel les variables sont instanciées. En considérant que le parfait ordonnancement des variables résulterait en une recherche sans retour-arrière, la recherche d'heuristiques de choix de variables suscite toujours autant l'intérêt des chercheurs. Depuis quinze ans, l'approche basée sur la pondération des contraintes s'est toujours montrée efficace pour guider la recherche. Dans cet article, nous montrons comment les heuristiques génériques de choix de variables `wdeg` et `dom/wdeg` ont été rendues plus robustes ces dernières années par un affinement et le vieillissement de l'information extraite à chaque conflit. Nos résultats expérimentaux avec `ACE`, le nouvel avatar d'`AbsCon`, montrent le réel intérêt de ces heuristiques revisitées.

Abstract

Backtracking search is a complete approach that is traditionally used to solve instances modeled as constraint satisfaction or optimization problems. The space explored during search depends dramatically on the order that variables are instantiated. Considering that a perfect variable ordering might result to a backtrack-free search, finding variable ordering heuristics has always attracted research interest. For fifteen years, constraint weighting has been shown to be a successful approach for guiding backtrack search. In this paper, we show how the popular generic variable ordering heuristics `dom/wdeg` and `wdeg` have been made more robust these two last years by refining and aging the extracted information at each conflict. Our experimental results with `ACE`, the new `AbsCon` avatar, show the practical interest of these revisited constraint weighting heuristics.

*Papier doctorant : Hugues Wattez[1] est auteur principal.

1 Introduction

Les solveurs de contraintes sont constitués de plusieurs composants, dédiés au filtrage de l'espace de recherche, à la conduite de l'exploration, ou encore à l'apprentissage de nogoods, dont la configuration et l'ajustement ont un impact réel sur les performances. Un point particulièrement important dans le choix d'une bonne configuration concerne la sélection de l'heuristique de choix de variables. Une heuristique soigneusement choisie permet de réduire significativement la taille de l'espace de recherche parcouru pour un problème donné. Dans cet article, nous nous attardons sur une famille d'heuristiques génériques de choix de variables ayant démontré leur grande robustesse au cours des ans : les heuristiques basées sur la pondération de contraintes dont `dom/wdeg` et `wdeg` [1] sont les précurseurs. Ces dernières années, de nouvelles avancées ont été faites sur la base de la pondération de contraintes. Tout d'abord, `chs` [4], qui utilise le principe de vieillissement des scores basés sur l'historique des conflits, est venu apporter une nouvelle dimension aux heuristiques d'origine. Ensuite, $\texttt{wdeg}^{\texttt{ca.cd}}$ [13], qui ajuste la pondération en fonction du contexte (arité courante des contraintes, taille des domaines courants), a également semblé démontrer un surcroît de robustesse.

Le but de notre étude est de proposer une uniformisation des différentes heuristiques que nous considérons appartenir à une même famille. La première étape consiste à identifier les principales fonctions de ces heuristiques afin d'établir une interface uniformisant les mécanismes de chacune d'entre elles. L'étape suivante décline chaque heuristique, dans le cadre de cette interface, en accord avec leur descriptif, mais aussi l'implantation effectuée dans le solveur `ACE`. Ainsi, nous espérons que le lecteur aura une meilleure appréhen-

sion des ressemblances et différences des mécanismes de ces heuristiques dont l'apparenté deviendra par la même occasion plus évidente. Pour finir, une évaluation expérimentale est présentée sur la base d'une sélection de trois heuristiques principales en diversifiant légèrement la configuration du solveur, et en utilisant les benchmarks des dernières compétitions XCSP [3, 2, 6], que ce soit pour la satisfaction ou l'optimisation de contraintes.

2 Réseaux de contraintes

Un *réseau de contraintes* (CN pour *Constraint Network*) P est composé d'un ensemble fini de variables $\mathscr{X}$, et d'un ensemble fini de contraintes $\mathscr{C}$. Chaque variable x prend une valeur dans un domaine fini, noté $\texttt{dom}(x)$. Chaque contrainte c représente une relation mathématique $\texttt{rel}(c)$ associé à un ensemble de variables, appelé la portée (*scope*) de c, et noté $\texttt{scp}(c)$. L'*arité* d'une contrainte c est la taille de sa portée.

Une *solution* de P correspond à l'affectation d'une valeur à chaque variable de $\mathscr{X}$ de sorte que toutes les contraintes de $\mathscr{C}$ soient satisfaites. Un réseau de contraintes est *cohérent* lorsqu'il admet au moins une solution. Le *problème de satisfaction de contraintes* (CSP pour *Constraint Satisfaction Problem*) consiste à déterminer si un CN donné est cohérent ou non.

Un réseau de contraintes sous optimisation (CNO pour *Constraint Network under Optimization*) est un réseau de contraintes combiné à une fonction d'objectif $\texttt{obj}$ permettant d'associer à toute solution une valeur dans $\mathbb{R}$. Sans perte de généralité, nous considérons que $\texttt{obj}$ est à minimiser. Une solution S d'un CNO est une solution du CN sous-jacent ; S est *optimale* s'il n'existe pas d'autre solution S' tel que $\texttt{obj}(S') < \texttt{obj}(S)$. Le *problème d'optimisation sous contraintes* (COP pour *Constraint Optimisation Problem*) consiste à trouver une solution optimale à un CNO donné.

Une procédure classique pour résoudre ces problèmes est d'effectuer une recherche avec retour-arrière dans l'espace des solutions partielles, et de maintenir une propriété appelée l'*arc-cohérence généralisée* [8] après chaque décision. Cette procédure, appelée le *maintien de l'arc-cohérence* (MAC pour *Maintaining Arc Consistency*) [9], construit un arbre binaire de recherche $\mathcal{T}$: pour chaque nœud interne ν de $\mathcal{T}$, un couple (x, v) est sélectionné où x est une variable non affectée et v est une valeur dans $\texttt{dom}(x)$. Ainsi, deux cas sont considérés pour l'exploration : l'affectation $x = v$ (décision dite positive) et la réfutation $x \neq v$ (décision dite négative).

L'ordre dans lequel les variables sont choisies durant l'exploration est décidé par une *heuristique de choix de variables*, notée H. Autrement dit, à chaque nœud interne ν de l'arbre de recherche $\mathcal{T}$, la procédure MAC sélectionne une nouvelle variable x en utilisant H, et affecte à x une valeur v en accord avec l'heuristique de choix de valeurs, qui est par exemple tout simplement l'ordre lexicographique sur $\texttt{dom}(x)$. Choisir la bonne heuristique de choix de variables H pour un réseau de contraintes donné est une question clé. Dans la section suivante, nous présentons l'état de l'art des heuristiques de choix de variables basant l'évaluation relative des variables sur le concept de pondération de contraintes.

Pour finir, il est important de noter que les solveurs modernes sont tous équipés d'une politique de redémarrage, afin de permettre une meilleure diversification, et contrer de possibles mauvais choix (décisions) initiaux. C'est pourquoi, dans la section suivante, nous faisons référence à la notion de *run* qui est l'arbre partiel de recherche construit entre deux (re-)démarrages.

3 Heuristiques de choix de variables

Afin d'exprimer simplement, et de manière homogène, les différentes heuristiques qui vont suivre, nous utilisons une interface composée de quatre fonctions.

init() Avant que ne démarre la phase d'exploration par le solveur (c'est-à-dire que le solveur n'entame sa série de runs), l'heuristique de choix de variables initialise ses structures par le biais de cette fonction.

beforeRun() À chaque début de run, cette fonction permet la mise-à-jour des différents paramètres ou scores de l'heuristique.

atConflict(c) Pendant le run courant, le solveur peut rencontrer des conflits (lorsque le domaine d'une variable devient vide lors de l'appel d'un algorithme de filtrage sur l'une des contraintes). À chaque conflit, la fonction atConflict est appelée avec la contrainte c ayant causé le conflit.

score(x) Cette fonction est appelée (sur toutes les variables non déjà instanciées) lors de la sélection de la prochaine variable à instancier par l'heuristique. D'un point de vue global, le solveur cherche à identifier la variable dont le score est maximal : $\max_{x \in \mathscr{X}} \textsc{score}(x)$. En pratique, seules les variables libres (non-assignées par le solveur) du réseau de contraintes seront considérées.

En cas de meilleurs scores identiques, il existe un mécanisme subsidiaire permettant de discriminer les variables, appelé *tie-breaker*. Ce mécanisme, extérieur à l'interface précédemment déclarée, peut être considéré comme un biais $b(x)$ venant apporter une information supplémentaire sur la variable x. Ce biais peut avoir une

influence certaine sur l'heuristique de choix de variables car il permet notamment de pré-ordonner l'ensemble des variables au début du premier run. Ce biais peut être arbitraire ; basé sur un générateur aléatoire, sur l'ordre lexicographique du nom des variables, ou encore sur un ordre défini par une heuristique secondaire telle que `deg` (plus grand degré) par exemple.

Maintenant que l'interface est définie, la description de chacune des heuristiques auxquelles nous nous intéressons dans cet article peut être donnée. Chacune de ces heuristiques est basée sur une forme de pondération, et est donc notée `wdeg` (weighted degree) en suivant l'appellation choisie initialement [1]. Les différentes variantes de cette famille d'heuristiques sont alors identifiées par un terme placé en exposant, comme par exemple dans $\texttt{wdeg}^{\texttt{ca.cd}}$. Il est également possible d'utiliser `dom/wdeg` à la place de `wdeg` comme base de calcul ; cela sera défini un peu plus loin.

Algorithme 1 : $\texttt{wdeg}^{\texttt{unit-2004}}$

```
1  Méthode INIT() :
2    pour chaque c ∈ 𝒞 faire
3      w_c ← 0

4  Méthode BEFORERUN() :
     Aucune opération

5  Méthode ATCONFLICT(c) :
6    r ← 1
7    w_c ← w_c + r

8  Méthode SCORE(x) :
9    retourner Σ_{c∈𝒞 : x∈scp(c) ∧ |fut(c)|>1} w_c
```

$\texttt{wdeg}^{\texttt{unit-2004}}$ Il s'agit de l'heuristique originelle de la famille `wdeg`. Comme structure de données, cette heuristique ne nécessite en fait qu'une variable w_c par contrainte c pour enregistrer le score de celle-ci. À chaque conflit issu d'une contrainte c (lors du processus de filtrage par celle-ci), cette variable, qui sert simplement de compteur, est incrémentée de 1 (*unit* dans l'intitulé faisant référence à une incrémentation unitaire). Ainsi, le score d'une variable x, selon $\texttt{wdeg}^{\texttt{unit-2004}}$, correspond à la somme du poids des contraintes (ayant strictement plus d'une variable non-assignée ; $fut(c)$ désigne l'ensemble des variables *futures* de $scp(c)$, c'est-à-dire des variables non assignées par le solveur, impliquant x.

Algorithme 2 : $\texttt{wdeg}^{\texttt{unit}}$

```
1  Méthode INIT() :
2    pour chaque c ∈ 𝒞 faire
3      pour chaque x ∈ scp(c) faire
4        w_c^x ← 0

5  Méthode BEFORERUN() :
     Aucune opération

6  Méthode ATCONFLICT(c) :
7    pour chaque x ∈ fut(c) faire
8      r ← 1
9      w_c^x ← w_c^x + r

10 Méthode SCORE(x) :
11   retourner Σ_{c∈𝒞 : x∈scp(c) ∧ |fut(c)|>1} w_c^x
```

$\texttt{wdeg}^{\texttt{unit}}$ Une première amélioration, implantée assez rapidement dans `AbsCon`, consiste à affiner la façon dont le poids évolue : non plus globalement pour toutes les variables d'une contrainte, mais seulement pour celles qui sont futures. Il est alors nécessaire d'introduire un tableau pour chaque contrainte c afin d'enregistrer le score (poids) local w_c^x d'une variable x impliquée dans c. La particularité réside ainsi dans la manière d'interpréter un conflit : au lieu d'augmenter globalement le poids de la contrainte ayant causé le conflit, seules les variables futures (non encore assignées) dans la portée de la contrainte voient leur poids incrémenté.

$\texttt{wdeg}^{\texttt{ca.cd}}$ Toujours dans l'optique d'affiner la pondération, $\texttt{wdeg}^{\texttt{ca.cd}}$ [13] propose une nouvelle amélioration. Après chaque conflit, l'incrémentation du score local des variables futures de la contrainte conflictuelle est construit sur l'arité courante (nombre de variables futures) et la taille des domaines courants.

$\texttt{wdeg}^{\texttt{chs}}$ En plus des variables w_c, comme pour $\texttt{wdeg}^{\texttt{unit-2004}}$, une variable t_c pour chaque contrainte c et correspondant au temps du dernier conflit impliquant la contrainte c est introduite pour l'heuristique $\texttt{wdeg}^{\texttt{chs}}$ [4] : le « temps » correspond ici à un compteur de conflits et non à une horloge effective. Par la suite, `time` est introduit et correspond au temps du dernier conflit du solveur : $\texttt{time} = \max_{c \in \mathscr{C}} t_c$. Les valeurs des variables t_c servent par la suite à différencier les contraintes entrant souvent en conflit. La fonction BEFORERUN est utilisée dans l'implantation de $\texttt{wdeg}^{\texttt{chs}}$

Algorithme 3 : `wdeg`$^{\texttt{ca.cd}}$

1 **Méthode** INIT() :
2   **pour chaque** $c \in \mathscr{C}$ **faire**
3   **pour chaque** $x \in \texttt{scp}(c)$ **faire**
4       $w_c^x \leftarrow 0$

5 **Méthode** BEFORERUN() :
  Aucune opération

6 **Méthode** ATCONFLICT(c) :
7   **pour chaque** $x \in \texttt{fut}(c)$ **faire**
8     $r \leftarrow \frac{1}{|\texttt{fut}(x)| \times \max(|\texttt{dom}(x)|, 1/2)}$
9     $w_c^x \leftarrow w_c^x + r$

10 **Méthode** SCORE(x) :
11   **retourner** $\sum_{c \in \mathscr{C} : x \in scp(c) \wedge |\texttt{fut}(c)|>1} w_c^x$

Algorithme 4 : `wdeg`$^{\texttt{chs}}$

1 **Méthode** INIT() :
2   $\texttt{time} \leftarrow 0$
3   **pour chaque** $c \in \mathscr{C}$ **faire**
4     $w_c \leftarrow 0$
5     $t_c \leftarrow 0$

6 **Méthode** BEFORERUN() :
7   **pour chaque** $c \in \mathscr{C}$ **faire**
8     $w_c \leftarrow w_c \times 0.995^{\texttt{time}-t_c}$
9   $\alpha \leftarrow 1/10$

10 **Méthode** ATCONFLICT(c) :
11   $r \leftarrow \frac{1}{\texttt{time}-t_c+1}$
12   $\alpha \leftarrow \max(6/100, \alpha - 10^{-6})$
13   $w_c \leftarrow (1-\alpha) \times w_c + \alpha \times r$
14   $\texttt{time} \leftarrow \texttt{time} + 1$
15   $t_c \leftarrow \texttt{time}$

16 **Méthode** SCORE(x) :
17   **retourner** $\sum_{c \in : x \in scp(c) \wedge |\texttt{fut}(c)|>1} w_c$

afin de vieillir la pondération des contraintes en fonction du temps depuis lequel elle n'est pas entrée en conflit : plus ce temps est important, plus l'affaiblissement du score l'est aussi. Une variable α initialisée à $1/10$ et évoluant au cours des conflits est introduite afin de donner plus ou moins d'importance au score donné à une variable lors d'un conflit. La fonction ATCONFLICT reprend ce principe de temps depuis lequel la contrainte a subi son dernier conflit pour calculer l'incrément : $r \leftarrow \frac{1}{\texttt{time}-t_c+1}$. La paramètre α est mis à jour à chaque conflit : $\alpha \leftarrow \max(6/100, \alpha - 10^{-6})$. Plus le nombre de conflits est important, plus α s'affaiblit et plus l'historique de la pondération de la contrainte c sera conservé : $w_c \leftarrow (1-\alpha) \times w_c + \alpha \times r$. Autrement dit, dans les premiers conflits du run, les incréments ont un impact plus grand sur la pondération des contraintes. Enfin, t_c est mis à jour en lui attribuant la valeur du dernier conflit. Les constantes de cette heuristique ont été calculées empiriquement dans l'étude [4] se basant sur des travaux d'une heuristique de branchement pour les solveurs SAT [7].

4 Évaluation expérimentale

Cette section présente les résultats expérimentaux composés de différentes campagnes comparant les trois heuristiques implantées dans `ACE` telles que présentées dans la littérature : `dom/wdeg`$^{\texttt{unit}}$, `wdeg`$^{\texttt{ca.cd}}$ et `dom/wdeg`$^{\texttt{chs}}$. Par défaut, `ACE` utilise une suite géométrique de raison 1.1 pour le *cutoff* (point (d'arrêt) du mécanisme de redémarrage (le cutoff du premier run est fixé à 10 mauvaises décisions), `lexico` comme heuristique de choix de valeurs et *last-conflict* (lc) [5] comme une manière paresseuse de simuler des retours-arrières intelligents en retenant les k derniers conflits.

L'ensemble des campagnes a été exécuté dans les mêmes conditions [1] avec un temps maximal d'exécution, pour chaque instance, fixé à 1,200 secondes. Les différentes heuristiques ont été exécutées sur une large diversité de problèmes de contraintes venant de la distribution XCSP [3, 6]. Nous utilisons deux benchmarks. Un premier, $\mathcal{I}_{\texttt{CSP}}$, correspond à l'ensemble des instances CSP des trois dernières compétitions XCSP : XCSP'17, XCSP'18 et XCSP'19, pour un total de 83 familles de problèmes et de 810 instances. Le second benchmark, $\mathcal{I}_{\texttt{COP}}$, est composé de l'ensemble des instances COP des compétitions XCSP'18 et XCSP'21, pour un total de 51 familles de problèmes et 697 instances.

Sur la base des trois heuristiques sélectionnées, nous souhaitons observer le comportement de chacune d'elles sur trois configurations du solveur `ACE` :

1. CPU 3.3 GHz Intel XEON E5-2643 et 32 GB RAM

— $\texttt{ACE}_{b=lex}^{lc=2}$: avec le mécanisme *last-conflict* retenant les deux derniers conflits et l'ordre lexicographique comme *tie-breaker* ;
— $\texttt{ACE}_{b=lex}^{lc=0}$: sans le mécanisme *last-conflict* et l'ordre lexicographique comme *tie-breaker* ;
— $\texttt{ACE}_{b=deg}^{lc=2}$: avec le mécanisme *last-conflict* retenant les deux derniers conflits et les poids proposés par `deg` comme *tie-breaker*.

Le solveur `ACE` par défaut correspond à $\texttt{ACE}_{b=lex}^{lc=2}$. Il s'agit par la même occasion de vérifier la robustesse de la configuration par défaut de `ACE` en variant deux de ses paramètres : *lc* et *b*. Pour cela, nous expérimentons les trois heuristiques données sur les trois environnements du solveur à travers deux sections : la première met en concurrence les solveurs sur $\mathcal{I}_{\texttt{CSP}}$ et la deuxième sur $\mathcal{I}_{\texttt{COP}}$.

4.1 Campagne sur les CSPs

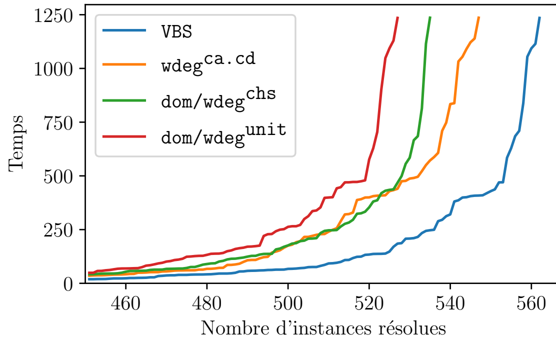

FIGURE 1 – Comparaison des heuristiques avec la configuration $\texttt{ACE}_{b=lex}^{lc=2}$ sur $\mathcal{I}_{\texttt{CSP}}$

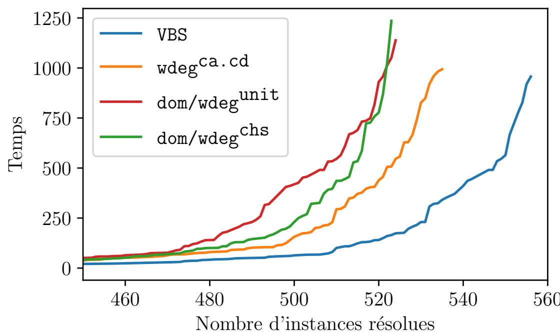

FIGURE 2 – Comparaison des heuristiques avec la configuration $\texttt{ACE}_{b=lex}^{lc=0}$ sur $\mathcal{I}_{\texttt{CSP}}$

Les Figures 1, 2 et 3 montrent les résultats des trois heuristiques et leur `VBS` pour les trois environnements du solveur `ACE`. Un `VBS` (pour *Virtual Best Solver*) correspond à un solveur virtuel simulant les meilleurs

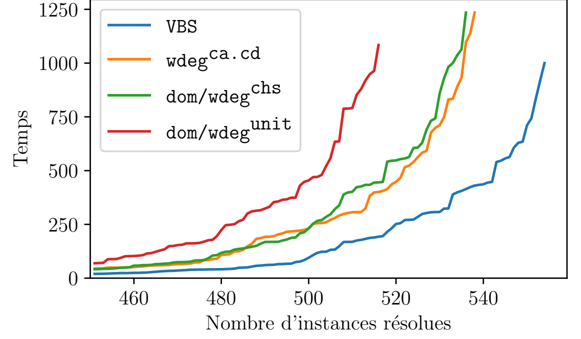

FIGURE 3 – Comparaison des heuristiques avec la configuration $\texttt{ACE}_{b=deg}^{lc=2}$ sur $\mathcal{I}_{\texttt{CSP}}$

résultats des solveurs réels : pour une instance donnée, le `VBS` prend le résultat de la meilleure des heuristiques. Dans notre cas, le `VBS` correspond toujours aux heuristiques courantes de la figure présentée. Un cactus-plot permet de mettre en évidence l'efficacité de chacun des solveurs : chaque solveur est représenté par une courbe représentant le temps cpu de chacune des instances résolues par celui-ci et ceci, dans l'ordre croissant de ce temps de résolution (représenté en y). Ainsi, pour un y donné, plus la courbe d'un solveur tend vers la droite, plus le solveur a résolu un grand nombre d'instances en ce temps donné. Pour simplifier la comparaison entre les cactus-plots, une heuristique est toujours associée à la même couleur. La légende donne le nom des solveurs dans l'ordre décroissant du nombre d'instances résolues.

De manière générale, nous remarquons que l'heuristique $\texttt{dom/wdeg}^{\texttt{unit}}$ est bel et bien dominée par ses successeurs $\texttt{dom/wdeg}^{\texttt{chs}}$ et $\texttt{wdeg}^{\texttt{ca.cd}}$ (bien que la configuration $\texttt{ACE}_{b=lex}^{lc=0}$ fasse légèrement défaut à $\texttt{dom/wdeg}^{\texttt{chs}}$ à partir de $y = 1,000$ secondes). Excepté pour la configuration $\texttt{ACE}_{b=deg}^{lc=2}$ où $\texttt{dom/wdeg}^{\texttt{chs}}$ et $\texttt{wdeg}^{\texttt{ca.cd}}$ semblent sensiblement égales, $\texttt{wdeg}^{\texttt{ca.cd}}$ obtient toujours les meilleurs résultats en terme du nombre d'instances résolues. L'étude des `VBS` apporte une information intéressante : il semblerait que ces trois heuristiques ont un comportement suffisamment différents pour créer un décalage d'une vingtaine à une trentaine d'instances par rapport au meilleur solveur réel. En effet, une analyse complémentaire montre que sur la configuration $\texttt{ACE}_{b=lex}^{lc=2}$, les heuristiques $\texttt{dom/wdeg}^{\texttt{unit}}$, $\texttt{wdeg}^{\texttt{ca.cd}}$ et $\texttt{dom/wdeg}^{\texttt{chs}}$ ont respectivement résolu 4, 17 et 6 instances que les autres heuristiques n'ont pas été en mesure de résoudre.

À l'échelle des configurations de `ACE`, celle apportant les meilleures performances pour les différentes heuristiques semble correspondre à la configuration par défaut, soit $\texttt{ACE}_{b=lex}^{lc=2}$. Cette observation est corroborée

	$\texttt{ACE}^{lc=2}_{b=lex}$	$\texttt{ACE}^{lc=2}_{b=deg}$	$\texttt{ACE}^{lc=0}_{b=lex}$
$\texttt{dom/wdeg}^{\texttt{unit}}$	527 $(75,519s)$	516 $(90,499s)$	524 $(82,409s)$
$\texttt{wdeg}^{\texttt{ca.cd}}$	**547** $(57,394s)$	**538** $(66,927s)$	**535** $(63,877s)$
$\texttt{dom/wdeg}^{\texttt{chs}}$	535 $(63,182s)$	536 $(69,583s)$	523 $(76,540s)$

TABLE 1 – Comparaison des heuristiques avec différentes configurations du solveur ACE sur $\mathcal{I}_{\texttt{CSP}}$

par les résultats donnés par le tableau 1. Notons que les configurations supprimant le mécanisme *last-conflict* ou utilisant le biais basé sur le degré des variables semblent détériorer les résultats (en terme d'instances résolues et/ou de temps).

4.2 Campagne sur les COPs

		$\mathcal{S}_{\texttt{domi}}$	$\mathcal{S}_{\texttt{opti}}$	$\mathcal{S}_{\texttt{agg}}$	$\mathcal{S}_{\texttt{borda}}$
$\texttt{ACE}^{lc=2}_{b=lex}$	$\texttt{dom/wdeg}^{\texttt{unit}}$	338	167	505	5,880.44
	$\texttt{wdeg}^{\texttt{ca.cd}}$	395	181	576	**6,178.23**
	$\texttt{dom/wdeg}^{\texttt{chs}}$	334	176	510	5,785.58
$\texttt{ACE}^{lc=2}_{b=deg}$	$\texttt{dom/wdeg}^{\texttt{unit}}$	366	170	536	5,954.80
	$\texttt{wdeg}^{\texttt{ca.cd}}$	**402**	**184**	**586**	6,143.00
	$\texttt{dom/wdeg}^{\texttt{chs}}$	365	179	544	5,862.42
$\texttt{ACE}^{lc=0}_{b=lex}$	$\texttt{dom/wdeg}^{\texttt{unit}}$	321	158	479	5,743.46
	$\texttt{wdeg}^{\texttt{ca.cd}}$	356	172	528	6,026.50
	$\texttt{dom/wdeg}^{\texttt{chs}}$	300	163	463	5,652.59

TABLE 2 – Comparaison des heuristiques avec différentes configurations du solveur ACE sur $\mathcal{I}_{\texttt{COP}}$

Pour la campagne COP, nous définissons quelques métriques afin d'interpréter les résultats et mener à bien la comparaison des heuristiques avec les différentes configurations de ACE. Pour cela, nous proposons deux métriques de base, que nous agrégeons ensuite en une troisième et présentons une quatrième métrique utilisée pour les compétitions MiniZinc [2] :

- $\mathcal{S}_{\texttt{domi}}$, pour un solveur donné, correspond au nombre d'instances pour lesquelles la meilleure borne trouvée (à l'échelle d'une instance) est au moins aussi bonne que la meilleure borne trouvée par l'ensemble des solveurs de la campagne ;
- $\mathcal{S}_{\texttt{opti}}$, pour un solveur donné, correspond au nombre d'instances dont l'optimalité a été prouvée ;
- $\mathcal{S}_{\texttt{agg}}$ correspond à la somme de $\mathcal{S}_{\texttt{domi}}$ et de $\mathcal{S}_{\texttt{opti}}$;
- $\mathcal{S}_{\texttt{borda}}$ correspond à une agrégation des résultats basée sur la méthode de vote Borda et prenant en compte la qualité de la borne et le temps d'exécution. De façon plus détaillée, chaque instance est considérée comme un votant donnant un rang à chaque solveur en fonction de sa capacité à résoudre efficacement l'instance donnée (comprenant la qualité de la borne et le temps). Ainsi, le score d'un solveur correspond à la somme du nombre de solveurs qu'il bat pour chaque instance.

2. https://www.minizinc.org/challenge2020/rules2020.html

Le tableau 2 présente l'ensemble des métriques proposées sur l'ensemble des heuristiques et configurations de solveurs. Les résultats mettent en valeur l'heuristique $\texttt{wdeg}^{\texttt{ca.cd}}$. En ce qui concerne les configurations du solveur, les résultats sont tranchés : les métriques basiques $\mathcal{S}_{\texttt{domi}}$, $\mathcal{S}_{\texttt{opti}}$ et $\mathcal{S}_{\texttt{agg}}$ ont tendance à mettre en valeur la configuration utilisant le *tie-breaker* basé sur le degré des variables, là où $\mathcal{S}_{\texttt{borda}}$ attribue un score plus important à la configuration par défaut de ACE. Si nous essayons de classer l'ensemble des heuristiques et configurations en fonction de $\mathcal{S}_{\texttt{agg}}$ et $\mathcal{S}_{\texttt{borda}}$, ces deux meilleurs solveurs se retrouvent premier et deuxième, puis deuxième et premier : il est donc assez compliqué de trancher entre les deux.

5 Conclusion

Sur la base de l'heuristique originelle wdeg, et des différentes optimisations (et affinements) apportées à celle-ci, nous proposons dans cet article une vision uniforme des heuristiques basées sur la pondération de contraintes ; ceci dans l'optique de mieux appréhender la compréhension des différents mécanismes en jeu pour chacun des variantes. Les expérimentations menées sur ACE (notamment la configuration par défaut) démontrent une certaine robustesse de $\texttt{wdeg}^{\texttt{ca.cd}}$ autant en satisfaction de contraintes qu'en optimisation. Toutefois, nous sommes conscients que l'implantation (et le solveur sous-jacent) peuvent jouer un rôle important. Il ne semble pas toujours facile d'obtenir des résultats totalement concordants sur la base d'autres solveurs. Nous faisons notamment référence à Choco [10], et aux discussions que nous avons menées avec C. Prud'Homme.

Perspective En lien direct avec certains travaux cherchant à combiner plusieurs heuristiques tels que la diversification de l'heuristique $\texttt{dom/wdeg}^{\texttt{chs}}$ [4], ou encore la diversification de familles d'heuristiques [12], la recherche d'une méthode permettant de récupérer les meilleures instances des heuristiques de la présente famille pourrait aussi être intéressante. La diversification des méthodes pour pondérer les contraintes semblent non-seulement intéressantes quant aux capacités du VBS présenté dans cet article, mais aussi à travers le VBS d'une étude annexe montrant que la diversification du *tie-breaker* permet aussi de larges bénéfices.

Références

[1] F. Boussemart, F. Hemery, C. Lecoutre et L. Sais : Boosting systematic search by weighting constraints. *In Proceedings of ECAI'04*, pages 146–150, 2004.

[2] F. Boussemart, C. Lecoutre, G. Audemard et C. Piette : XCSP3-core : A format for representing constraint satisfaction/optimization problems. *CoRR*, abs/2009.00514, 2020.

[3] F. Boussemart, C. Lecoutre et G. Audemard C. Piette : XCSP3 : an integrated format for benchmarking combinatorial constrained problems. *CoRR*, abs/1611.03398, 2016.

[4] M. S. Cherif, D. Habet et C. Terrioux : On the Refinement of Conflict History Search Through Multi-Armed Bandit. *In Proceedings of ICTAI'20*, pages 264–271. IEEE, 2020.

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

[6] C. Lecoutre et N. Szczepanski : PYCSP3 : modeling combinatorial constrained problems in python. *CoRR*, abs/2009.00326, 2020.

[7] J. Liang, Vijay Ganesh, P. Poupart et K. Czarnecki : Exponential recency weighted average branching heuristic for sat solvers. *In Proceedings of AAAI'16*, 2016.

[8] U. Montanari : Network of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7:95–132, 1974.

[9] D. Sabin et E.C. Freuder : Contradicting conventional wisdom in constraint satisfaction. *In Proceedings of CP'94*, pages 10–20, 1994.

[10] The Choco Team : Choco : an open source Java constraint programming library. *In [11]*, pages 8–14, 2008.

[11] M.R.C. van Dongen, C. Lecoutre et O. Roussel, éditeurs. *Proceedings of the third constraint solver competition.* http ://www.cril.univ-artois.fr/CPAI08/Competition-08.pdf, 2008.

[12] H. Wattez, F. Koriche, C. Lecoutre, A. Paparrizou et S. Tabary : Learning Variable Ordering Heuristics with Multi-Armed Bandits and Restarts. *In Proceedings of ECAI'20*, 2020.

[13] H. Wattez, C. Lecoutre, A. Paparrizou et S. Tabary : Refining constraint weighting. *In Proceedings of ICTAI'19*, pages 71–77, 2019.