Descente Agressive de Borne en Optimisation sous Contraintes

Thibault Falque[1] *__Christophe Lecoutre__[2] **Bertrand Mazure**[2] **Hugues Wattez**[2]

[1] Exakis Nelite, Paris, France
[2] CRIL, Univ Artois & CNRS
thibault.falque@exakis-nelite.com {lecoutre,mazure,wattez}@cril.fr

Résumé

La recherche avec retour en arrière est une approche complète classique pour explorer l'espace de recherche d'un problème d'optimisation sous contraintes. Chaque fois qu'une nouvelle solution est trouvée pendant la recherche, la borne qui lui est associée est utilisée pour contraindre davantage le problème, et donc la recherche restante. Un scénario extrême (mauvais) est celui où les solutions sont trouvées en séquence avec de très petites différences entre les bornes successives. Dans cet article, nous proposons une approche ABD (*Aggressive Bound Descent*) pour remédier à ce problème : les nouvelles bornes sont modifiées de manière exponentielle tant que l'algorithme de recherche est efficace. Nous montrons que cette approche peut rendre le solveur plus robuste, surtout au début de la recherche. Nos expérimentations confirment ce comportement pour l'optimisation sous contraintes et les problèmes Pseudo-Booléens.

Abstract

Backtrack search is a classical complete approach for exploring the search space of a constraint optimization problem. Each time a new solution is found during search, its associated bound is used to constrain more the problem, and so the remaining search. An extreme (bad) scenario is when solutions are found in sequence with very small differences between successive bounds. In this paper, we propose an aggressive bound descent (ABD) approach to remedy this problem: new bounds are modified exponentially as long as the searching algorithm is successful. We show that this approach can render the solver more robust, especially at the beginning of search. Our experiments confirm this behavior for both constraint optimization and Pseudo-Boolean problems.

*Papier doctorant : Thibault Falque[1] est auteur principal.

1 Introduction

En programmation par contraintes (CP), même si de nombreuses extensions ont été proposées depuis les années 70, il est habituel de traiter soit des problèmes de satisfaction de contraintes (CSP), soit des problèmes d'optimisation sous contraintes (COP). En plus d'un ensemble de variables (entières) à assigner tout en satisfaisant un ensemble de contraintes, une instance COP implique une fonction objective à optimiser (c'est-à-dire une fonction de coût à minimiser ou une fonction de récompense à maximiser). La résolution d'une instance COP nécessite non seulement de prouver la satisfaisabilité (c'est-à-dire de trouver au moins une solution), mais aussi, idéalement, de prouver l'optimalité ou au moins de trouver des solutions d'assez bonne qualité (c'est-à-dire proches de l'optimalité).

La recherche avec retour en arrière est une approche complète classique pour explorer l'espace de recherche d'une instance COP. Cela revient à résoudre une série d'instances CSP. En supposant une fonction objective f à minimiser où initialement f est traitée sous la forme d'une contrainte $f < \infty$, et chaque fois qu'une nouvelle solution de coût B est trouvée, B est utilisé comme nouvelle borne pour la contrainte objective, de façon à devenir $f < B$ (formant ainsi une instance CSP plus contrainte). De cette façon, toute nouvelle solution trouvée est garantie d'être de meilleure qualité (coût inférieur) que la précédente, et l'optimalité peut être prouvée lorsque l'instance du problème devient insatisfaisable.

Il n'est pas rare que les problèmes d'optimisation issus de l'industrie soient faiblement contraints, ce qui signifie qu'un grand nombre de solutions existent avec des qualités diverses dans différentes parties de l'espace de recherche. Dans de telles situations, l'application

de la recherche avec retour en arrière peut être pénalisée parce que la descente des bornes (c'est-à-dire la séquence décroissante des bornes successivement trouvées) peut être très lente : la distance entre deux bornes successives peut être plutôt faible. Dans cet article, nous proposons une approche pour tenter de réduire ce phénomène en modifiant les bornes d'une manière plus agressive : tant qu'elle est réussie, de nouvelles bornes pour la contrainte objective sont calculées en suivant une croissance exponentielle. Bien sûr, dans le cas où la borne rend le problème insatisfaisable ou très difficile à résoudre, un mécanisme de redémarrage nous permet de reprendre la recherche sur des pistes plus certaines.

Même si d'autres techniques de recherche existent dans la littérature, comme par exemple la métaheuristique largement utilisée *Large Neighborhood Search* (LNS) [12], dans cet article, nous nous concentrons sur la recherche en profondeur avec retour en arrière, équipée du mécanisme de *solution(-based phase) saving* qui s'est avérée être très efficace [14, 5].

Cet article est organisé comme suit. La Section 2 présente le contexte technique de la programmation par contraintes. Ensuite, dans la Section 3, le principe de Descente Agressive de Borne (ABD pour *Aggressive Bound Descent*) est présenté, et avant de conclure, les résultats expérimentaux sont donnés dans la Section 4.

2 Contexte Technique

Un *réseau de contraintes* (CN pour *Constraint Network*) est constitué d'un ensemble fini de variables et d'un ensemble fini de contraintes. Chaque variable x peut prendre une valeur dans un ensemble fini appelé le *domaine* de x, noté $\texttt{dom}(x)$. Chaque contrainte c est spécifiée par une relation sur un ensemble de variables. Une *solution* d'un CN est l'affectation d'une valeur à toutes les variables de façon à satisfaire toutes les contraintes. Un CN est *satisfaisable* si il admet au moins une solution, et le *problème de satisfaction de contraintes* (CSP) correspondant consiste à déterminer si un CN donné est satisfaisable ou non.

Un *réseau de contraintes sous optimisation* (CNO pour *Constraint Network under Optimization*) est un réseau de contraintes associé à une fonction objectif $\texttt{obj}$ faisant correspondre toute solution à une valeur [1] dans $\mathbb{D}$. Sans aucune perte de généralité, nous considérerons que $\texttt{obj}$ doit être minimisée. Une solution S d'un CNO est une solution du CN sous-jacent ; S est *optimal* s'il n'existe aucune autre solution S' telle que $\texttt{obj}(S') < \texttt{obj}(S)$. La tâche habituelle du *problème d'optimisation sous contraintes* (COP) est de trouver une solution optimale pour un CNO donné. Notez que les CN et les CNO sont également appelés instances de CSP/COP.

La recherche avec retours en arrière est une procédure complète classique pour résoudre les instances de CSP/COP. Elle alterne les affectations de variables (et les réfutations) et un mécanisme appelé propagation de contraintes afin de filtrer l'espace de recherche. Typiquement, comme pour MAC (*Maintaining Arc Consistency*) [11] qui propage les contraintes en maintenant la propriété de cohérence d'arc, un arbre de recherche binaire $\mathcal{T}$ est construit : à chaque nœud interne de $\mathcal{T}$, (i) on sélectionne une paire (x, v) où x est une variable non-fixée et v est une valeur dans $\texttt{dom}(x)$, et (ii) deux cas (branches) sont considérés, correspondant à l'affectation $x = v$ et à la réfutation $x \neq v$.

L'ordre dans lequel les variables sont choisies pendant le parcours en profondeur de l'espace de recherche est décidé par une *heuristique de choix de variables* ; une heuristique générique classique étant `dom/wdeg` [3]. L'ordre dans lequel les valeurs sont choisies lors de l'affectation des variables est déterminé par une *heuristique de choix de valeurs* ; pour les COPs, il est fortement recommandé d'utiliser d'abord la valeur présente dans la dernière solution trouvée, ce qui est une technique connue sous le nom de *solution(-based phase) saving* [14, 5].

La recherche avec retours en arrière pour le COP repose sur la résolution de CSP : le principe consiste à ajouter au réseau de contraintes une *contrainte objectif* spéciale $\texttt{obj} < \infty$ (bien qu'elle soit initialement satisfaite), et de mettre à jour la borne de cette contrainte chaque fois qu'une nouvelle solution est trouvée. Cela signifie qu'à chaque nouvelle solution S trouvée avec un coût $B = \texttt{obj}(S)$, la contrainte objectif devient $\texttt{obj} < B$. Par conséquent, une séquence de solutions, dont la qualité est croissante, est générée (la SATisfiabilité est systématiquement prouvée par rapport à la borne actuelle de la contrainte objectif) jusqu'à ce qu'il n'en existe plus aucune (l'insatisfaisabilité (*UNSATisfiability*) est finalement prouvée par rapport à la borne imposée par la dernière solution trouvée), ce qui garantit que la dernière solution trouvée est optimale.

Les politiques de redémarrage jouent un rôle important dans les solveurs de contraintes modernes, car elles permettent d'aborder le phénomène du *heavy-tailed* des instances SAT (*Satisfiability Testing*) et CSP/COP [6]. En substance, une politique de redémarrage correspond à une fonction $\texttt{restart} : \mathbb{N}^+ \to \mathbb{N}^+$, qui indique le nombre maximal d'*étapes* autorisés pour l'algorithme de recherche lors d'une exécution, appelée *run*, $j \geq 1$. Cela signifie que la recherche avec retour en arrière piloté par un mécanisme de redémarrageconstruit une séquence d'arbres de recherche binaires $\langle \mathcal{T}_1, \mathcal{T}_2, \ldots \rangle$, où $\mathcal{T}_j$ est l'arbre de recherche exploré au run j.

1. Pour simplifier la présentation, nous considérons que les coûts sont donnés par des valeurs entières.

Notez que le *cutoff*, qui est le nombre maximum d'étapes autorisées pendant une exécution, peut correspondre au nombre de retours en arrière, au nombre de mauvaises décisions [1], ou toute autre mesure pertinente. Dans une stratégie de redémarrage au cutoff *fixé*, `restart`(j) est constant quelle que soit le run j. Dans une stratégie de redémarrage à cutoff *dynamique*, `restart` augmente géométriquement le cutoff [15], ce qui garantit que tout l'espace des solutions partielles sera exploré.

3 Descente Agressive de Borne

Lors de la résolution d'une instance COP, la *descente de borne* est définie comme la séquence $D = \langle B_1, B_2, \ldots \rangle$ de bornes successives identifiées par l'algorithme de recherche. À un extrême, cette séquence ne contient qu'une seule valeur, la borne optimale. A un autre extrême, elle contient une grande séquence de valeurs, chacune étant proche de la précédente : la descente de borne est dite *lente*. C'est le cas lorsque la valeur moyenne de la séquence dérivée des gains (ou écarts) de bornes $G = \langle B_1 - B_2, B_2 - B_3, \ldots \rangle$ est petite (proche de 1).

Il est certain qu'une descente lente de borne indique qu'il y a une certaine marge d'amélioration dans la façon dont la recherche avec retour arrière est menée. En effet, l'énumération d'un grand nombre de solutions proches les unes des autres avant d'atteindre l'optimalité implique la résolution d'un grand nombre d'instances de problèmes de satisfaction dérivés et cela peut être pénalisant. C'est pourquoi nous proposons une politique *agressive* de descente de bornes : la politique ABD. Au lieu de fixer la borne stricte de la contrainte objective à B lorsqu'une nouvelle solution de coût B est trouvée, nous proposons de la fixer à une valeur éventuellement inférieure B'.

Une première politique ABD simple pourrait consister à utiliser une différence statique entre B et B' : $B' = B - \Delta$ où Δ est une valeur entière positive fixe. Cependant, cette politique *statique* souffre clairement d'un manque d'adaptabilité, et de plus, fixer la bonne valeur pour Δ peut dépendre du problème et ne pas être très facile à réaliser.

C'est pourquoi nous proposons des politiques *dynamiques* pour ABD, inspirées des études concernant les suites utilisées par les politiques de redémarrage.

Pour définir des politiques ABD dynamiques, nous introduisons d'abord quelques suites classiques d'entiers strictement positifs, c'est-à-dire des fonctions `abd` : $\mathbb{N}^+ \rightarrow \mathbb{N}^+$. Bien que détaillé plus loin, le paramètre $i \geq 1$ de ces séquences correspond au nombre de mises à jour successives réussies de la borne, c'est-à-dire de mises à jour successives agressives de la borne de la contrainte objective tout en conservant la satisfaisabilité.

Plus précisément, quatre suites entières sont utilisées dans notre étude :

$$\texttt{exp}(i) = 2^{i-1} \tag{1}$$

$$\texttt{rexp}(i) = \begin{cases} 2^{k-1}, & \text{si } i = \frac{k(k+1)}{2} \\ 2^{i-\frac{k(k+1)}{2}-1}, & \text{si } \frac{k(k+1)}{2} < i < \frac{(k+1)(k+2)}{2} \end{cases} \tag{2}$$

$$\texttt{luby}(i) = \begin{cases} 2^{k-1}, & \text{si } i = 2^k - 1 \\ \texttt{luby}(i - 2^{k-1} + 1), & \text{si } 2^{k-1} \leq i < 2^k - 1 \end{cases} \tag{3}$$

$$\texttt{prev}(i) = \begin{cases} 1, & \text{si } i = 1 \\ G_{i-1} \times 2, & \text{sinon} \end{cases} \tag{4}$$

où, dans l'Équation 4, G_i est la ième valeur de la suite de gains, telle que définie précédemment.

L'Équation 1 correspond à la fonction exponentielle classique `exp` (en base 2). Dérivée de cette classique progression exponentielle, `rexp` dans l'Équation 2 correspond à une suite de `exp` régulièrement réinitialisée. Les premières valeurs de cette suite sont : $1, 1, 2, 1, 2, 4, \ldots$. En ne considérant que les nombres les plus élevés produits par le premier terme (condition) de l'équation, on obtient une progression légèrement plus lente que la précédente : $O(2^{\sqrt{i}})$. Une autre suite, couramment utilisée dans les politiques de redémarrage, est la suite de Luby [10], donnée par l'Équation 3. Les premières valeurs de la suite de Luby sont : $1, 1, 2, 1, 1, 2, 4, \ldots$. En considérant à nouveau les plus grands nombres produits par le premier terme, on constate que la progression est en $O(i)$. Enfin, la dernière suite, donnée par l'Équation 4, est basée sur la suite des gains, et suit également une progression exponentielle.

Chaque suite dans $\Psi = \{\texttt{exp}, \texttt{rexp}, \texttt{luby}, \texttt{prev}\}$ nous permet de définir un éponyme de la politique ABD.

3.1 Politique ABD

Soit $\mathcal{T}$ l'arbre de recherche actuel construit par l'algorithme de recherche (c'est-à-dire pendant l'exécution actuelle). Soit D la descente de bornes produite depuis le début du run courant, et soit `abd` $\in \Psi$. L'exécution courante peut rencontrer trois situations distinctes :

1. le run en cours est arrêté car la valeur limite est atteinte,
2. le run en cours est arrêté parce que les algorithmes de recherche indiquent qu'il n'existe plus de solution,

3. une nouvelle solution S est trouvée.

Tout d'abord, nous discutons du cas le plus intéressant : le troisième. La politique ABD stipule que lorsqu'une nouvelle solution S de coût B est trouvée, B est ajouté à D, et la borne de la contrainte objective est fixée à $B + 1 - \texttt{abd}(i)$, où $i = |D|$. En d'autres termes, la contrainte objectif devient : $\texttt{obj} < B+1-\texttt{abd}(i)$; notez que 1 est ajouté à B car les fonctions `abd` ne retournent que des valeurs supérieures ou égales à 1. Maintenant, nous donnons une description générale précise (traitant en particulier les deux premières situations ci-dessus) de la façon dont une politique ABD peut être implémentée dans une recherche avec retour en arrière.

3.2 Simple Implémentation d'ABD

La fonction `solve`, Algorithme 1, tente de résoudre le CNO P grâce à l'usage de la politique de descente agressive de bornes `abd` spécifiée.

Algorithm 1: `solve`$(P, \texttt{abd})$

Output: $\underline{B_P}..\overline{B_P}$, runStatus

1 $\underline{B_P}..\overline{B_P} \leftarrow -\infty..+\infty$
2 **do**
3 $\quad P, \text{runStatus} \leftarrow \texttt{run}(P, \texttt{abd})$
4 **while** runStatus = CONTINUE
5 **return** $(\underline{B_P}..\overline{B_P}, \text{runStatus})$

Tout d'abord, les bornes inférieure et supérieure, désignées par $\underline{B_P}$ et $\overline{B_P}$, de la fonction objectif de P sont respectivement initialisées à $-\infty$ et $+\infty$ (ou toute autre valeur pertinente pouvant être pré-calculée). Pendant la recherche, ces bornes seront mises à jour (mais pour des raisons de simplicité, cela ne sera pas explicitement montré dans le pseudo-code). À la ligne 2, la séquence de runs (redémarrages) est lancée. Chaque fois qu'une nouvelle exécution est terminée, elle renvoie le réseau de contraintes (éventuellement mis à jour avec certaines contraintes ou certains *nogoods* qui ont été appris ; ceci sera discuté plus en détail plus tard) ainsi qu'un statut. Le statut prend l'une des valeurs suivantes : CONTINUE si le solveur est autorisé à poursuivre avec un nouveau run ; COMPLETE si le dernier run a exploré exhaustivement l'espace des solutions ; INCOMPLETE si le solveur a atteint la limite de temps sans avoir exploré entièrement l'espace de recherche. Enfin, la fonction renvoie les meilleures bornes trouvées (dans le cas où l'optimalité a été prouvée, nous avons $\underline{B_P} = \overline{B_P}$) et l'état final de la recherche.

La fonction `run`, Algorithme 2, effectue une recherche, en suivant les politiques de redémarrage et de `abd`. Avant d'aller plus loin, nous devons introduire la notion de *safe/unsafe* : lorsqu'on demande au solveur de diminuer agressivement sa borne objective, nous pouvons entrer dans une partie de l'espace de recherche qui est UNSAT. Si l'insatisfaisabilité est prouvée pendant l'exécution actuelle, cela peut être dû à notre approche agressive, et par conséquent, nous devons traiter ce problème. Ce point est abordé ci-dessous.

La fonction commence par initialiser un compteur i à 1. Il correspond au nombre de fois où l'on a essayé de trouver une nouvelle solution pendant l'exécution courante. À la ligne 2, Σ_i est la séquence de décisions prises le long de la branche la plus à droite du run actuel, juste avant de commencer la prochaine tentative de trouver une nouvelle solution ; de cette façon, nous pouvons continuer à chercher à partir du même endroit (en pratique, nous reprenons simplement la recherche après l'avoir arrêtée). Initialement, aucune décision n'est prise (et donc, Σ_1 est l'ensemble vide). D'un point de vue pratique, comme nous le verrons, seules les deux dernières séquences Σ_i et Σ_{i-1} seront utiles (pour traiter les cas de résolution sûre et non sûre).

Algorithm 2: `run`$(P, \texttt{abd})$

Output: (P, status)

1 $i \leftarrow 1$
2 $\Sigma_i \leftarrow \emptyset$
3 **do**
4 $\quad \Delta \leftarrow \texttt{abd}(i)$
5 $\quad i \leftarrow i + 1$
6 $\quad \Sigma_i, \text{status} \leftarrow \texttt{search\_next\_sol}(P, \Sigma_{i-1}, \Delta)$
7 **while** status = SAT
8 safe $\leftarrow \Delta = 1$
The global timeout of the resolution
9 **if** status = TIMEOUT **then**
10 $\quad$ **return** $(P, \text{INCOMPLETE})$
An unsat status with $\Delta = 1$
11 **if** status = UNSAT & safe **then**
12 $\quad$ **return** $(P, \text{COMPLETE})$
An unsat status with $\Delta > 1$
13 **if** status = UNSAT & ¬safe **then**
14 $\quad$ **return** $(P \oplus nld(\Sigma_{i-1}), \text{CONTINUE})$
The run cutoff is reached with $\Delta > 1$
15 **if** status = CUTOFF_REACHED & ¬safe **then**
16 $\quad$ **return** $(P \oplus nld(\Sigma_{i-1}), \text{CONTINUE})$
The run cutoff is reached with $\Delta = 1$
17 **if** status = CUTOFF_REACHED & safe **then**
18 $\quad$ **return** $(P \oplus nld(\Sigma_i), \text{CONTINUE})$

Ensuite, l'algorithme effectue des parcours itératifs tant que de nouvelles solutions peuvent être trouvées. À chaque tour de boucle, la prochaine limite d'écart Δ est calculée en sollicitant la politique `abd` et i est incrémenté (lignes $4et5$). Pour effectuer une partie de la recherche, la fonction `search_next_sol` est appelée, tout en considérant la séquence spécifiée de décisions de départ, et l'écart de borne spécifié. L'écart Δ est utilisé par `search_next_sol` pour calculer une borne supérieure temporaire B' qui remplace la borne supérieure courante $\overline{B_P}$: on a $B' = \overline{B_P} - \Delta + 1$, forçant alors la contrainte objectif à être $f < B'$ pendant cet appel à `search_next_sol`. Si une nouvelle solution de coût B (nécessairement, $B < B'$) est trouvée par `search_next_sol`, l'appel est arrêté, et la contrainte objectif est mise à jour pour devenir sans risque $f < B$. Sinon, l'appel est arrêté (parce que les limites de cutoff ou de temps sont atteintes), et la contrainte objectif est mise à jour pour devenir $f < \overline{B_P}$ (en récupérant la précédente borne supérieure). Pour résumer, cette fonction met implicitement à jour les bornes d'optimisation avant de retourner la nouvelle séquence de décisions (l'endroit exact où la recherche s'est arrêtée) et un statut (local). Le statut local est soit SAT, auquel cas l'exécution peut se poursuivre avec une nouvelle itération de la boucle, soit une valeur parmi UNSAT, CUTOFF_REACHED et TIMEOUT.

Le run actuel exécute nécessairement certaines vérifications à partir de la ligne 8. À cette ligne, un booléen est défini, nous informant si l'exécution actuelle était sûre ou non, en ce qui concerne la dernière valeur Δ calculée. Lorsque la limite de temps globale est atteinte, le réseau de contraintes et l'état INCOMPLETE sont renvoyés (lignes 9 et 10). Lorsque le statut local notifie UNSAT, nous avons deux cas à considérer. Si la recherche en cours a été effectuée de façon *safe*, COMPLETE peut être retourné car l'espace de recherche est garanti d'avoir été entièrement exploré. Sinon, CONTINUE est retourné avec le réseau de contraintes P intégrant éventuellement quelques nouvelles contraintes (nogoods). La notation $P \oplus nld(\Sigma_{i-1})$ indique que tous les nogoods qui peuvent être extraits de l'avant-dernière séquence de décisions (voir [8]) sont ajoutés à P ; ceci est valable car cette séquence était celle correspondant à la dernière solution trouvée. Lorsque l'état local notifie CUTOFF_REACHED, on peut aussi continuer, tout en considérant l'ajout de quelques nogoods de redémarrage, à partir de Σ_i ou Σ_{i-1}.

Nous concluons cette section par deux remarques. Premièrement, l'algorithme est introduit dans le contexte d'un schéma d'enregistrement de nogoods légers (seuls sont considérés les nogoods qui peuvent être extraits de la branche la plus à droite, lorsque la recherche est temporairement arrêtée). Cependant, il est possible de l'adapter à d'autres schémas d'apprentissage, en gardant la trace du moment exact où un nogood (clause) est inféré ; ceci est purement technique. Deuxièmement, il y a un cas spécifique concernant l'insatisfaisabilité : si jamais nous rencontrons une situation où $B' \leq \underline{B_P}$ en essayant de fixer une nouvelle borne supérieure temporaire B' pendant l'exécution actuelle, la suite est réinitialisée en forçant le retour de $i = 1$ et B' est recalculé.

3.3 Travail Connexe

Comme approche connexe, nous pouvons mentionner la division de domaine (par exemple [13]) dont le rôle est de partitionner le domaine de la variable sélectionnée par l'heuristique de choix de variables, et de brancher sur les sous-domaines résultants. Lorsque la fonction objectif est simplement représentée par une variable autonome (dont la valeur doit être minimisée ou maximisée), une approche de division de domaine peut être réglée pour simuler une descente de borne agressive. Cependant, aucun contrôle n'est possible car aucun mécanisme ne permet d'abandonner des choix trop optimistes, contrairement à ABD qui est bien intégrée dans une politique de redémarrage. De plus, lorsque la fonction objectif a une forme plus générale qu'une variable (comme par exemple une somme ou une valeur minimum/maximum à calculer), il n'existe pas de correspondance directe (et introduire systématiquement une variable auxiliaire pour représenter l'objectif peut être très intrusif, voire source d'inefficacité, pour le solveur).

Enfin, notons que la question d'éviter les longues séquences de solutions qui s'améliorent lentement a été abordée dans les MIP par l'introduction d'heuristiques primaires, qui visent à trouver et à améliorer les solutions réalisables au début du processus de résolution.

4 Résultats expérimentaux

Cette section présente quelques résultats expérimentaux concernant l'approche ABD sur un large éventail de problèmes d'optimisation. Pour réaliser les expériences, nous avons utilisé le solveur de contraintes `ACE`, qui est le nouvel avatar de AbsCon [2]. Les options par défaut du solveur ont été utilisées : $\texttt{wdeg}^{\texttt{ca.cd}}$ [17] comme heuristique de choix de variables, `lexico` comme heuristique de choix de valeurs, *last-conflict* (lc) [8] comme méthode paresseuse pour simuler un retour en arrière intelligent, `solution-saving` [14, 5] pour simuler une forme de recherche de voisinage, et une politique de redémarrage géométrique [16] avec

2. `https://www.cril.univ-artois.fr/~lecoutre/#/softwares`

un cutoff de base fixé à 10 mauvaises décisions et une raison fixée à 1.1. Nous étudions notamment l'impact de certains facteurs sur les suite les plus prometteuses parmi Ψ. Nous avons également réalisé des expériences similaires avec le solveur Pseudo-Booléen (PB) `Sat4j` [7]. Tous les solveurs ont été lancés sur un processeur Intel Xeon de 2.66 GHz, avec 32 Go de RAM, tandis que le timeout était fixé à $1,200$ secondes.

4.1 A propose des scores

Tout d'abord, nous présentons les méthodes de notation que nous avons utilisées pour évaluer les résultats expérimentaux. Pour des raisons de simplicité, nous continuons à considérer que nous n'avons que des problèmes de minimisation.

Étant donné un ensemble $\mathcal{I}$ d'instances et un ensemble $\mathcal{S}$ de solveurs, $b^t_{i,s}$ correspond à la meilleure borne (c'est-à-dire la plus basse) obtenue par le solveur $s \in \mathcal{S}$ sur l'instance $i \in \mathcal{I}$ au temps t, où $t \in [0, ..., T]$ et T est le timeout. Nous avons également un booléen $e^t_{i,\mathcal{S}}$ dont la valeur est *vrai* lorsqu'une solution a été trouvée au temps t par au moins un solveur de $\mathcal{S}$. Nous pouvons maintenant définir trois valeurs spécifiques :

$$\texttt{min}^t_i = \begin{cases} \min_{s \in S} b^t_{i,s}, & \text{si } e^t_{i,\mathcal{S}} \\ 0, & \text{sinon} \end{cases} \tag{5}$$

$$\texttt{max}^t_i = \begin{cases} \max_{s \in S} b^t_{i,s}, & \text{si } e^t_{i,\mathcal{S}} \\ 0, & \text{sinon} \end{cases} \tag{6}$$

$$\texttt{def}^t_i = b^t_{i,\texttt{def}} \tag{7}$$

où $b^t_{i,s}$ est égale à $\texttt{min}^t_i$ si $e^t_{i,\{s\}}$ est faux.

Les deux premières expressions correspondent respectivement à la plus petite (meilleure) et à la plus grande (pire) borne obtenue par un solveur sur une instance donnée i au temps t. La troisième expression est simplement la borne obtenue par le solveur par défaut sur i au temps t. Le *solveur par défaut* est le solveur original utilisant son comportement par défaut (et donc, n'utilisant aucune politique ABD).

Ensuite, nous pouvons calculer deux récompenses différentes pour une paire (i, s) :

$$r^t_{i,s} = \begin{cases} 0, & \text{si } \neg e^t_{i,\{s\}} \\ 1 - \frac{b^t_{i,s} - \texttt{min}^t_i}{\texttt{max}^t_i - \texttt{min}^t_i}, & \text{si } \texttt{max}^t_i \neq \texttt{min}^t_i \\ 1, & \text{sinon} \end{cases} \tag{8}$$

$$r'^t_{i,s} = \begin{cases} -\mathbb{1}_{e^t_{i,\{\texttt{def}\}}}, & \text{si } \neg e^t_{i,\{s\}} \\ -\frac{b^t_{i,s} - \texttt{def}^t_i}{\texttt{max}^t_i - \texttt{min}^t_i}, & \text{si } \texttt{max}^t_i \neq \texttt{min}^t_i \\ \mathbb{1}_{\neg e^t_{i,\{\texttt{def}\}}}, & \text{sinon} \end{cases} \tag{9}$$

où $\mathbb{1}_\alpha$ retourne 1 si α est vrai, sinon 0.

La première fonction de récompense (Équation 8) correspond à une normalisation classique *min-max* avec des valeurs manquantes possibles. Dans le cas où un solveur n'a trouvé aucune solution, sa récompense est de 0, et dans le cas où les bornes, la plus petite et la plus haute, sont égales (ce qui signifie que le solveur a trouvé l'unique solution connue au temps t), sa récompense est de 1. Sinon, on utilise *min-max* en prenant le complément à 1 du résultat (car la *borne inférieure* est la meilleure).

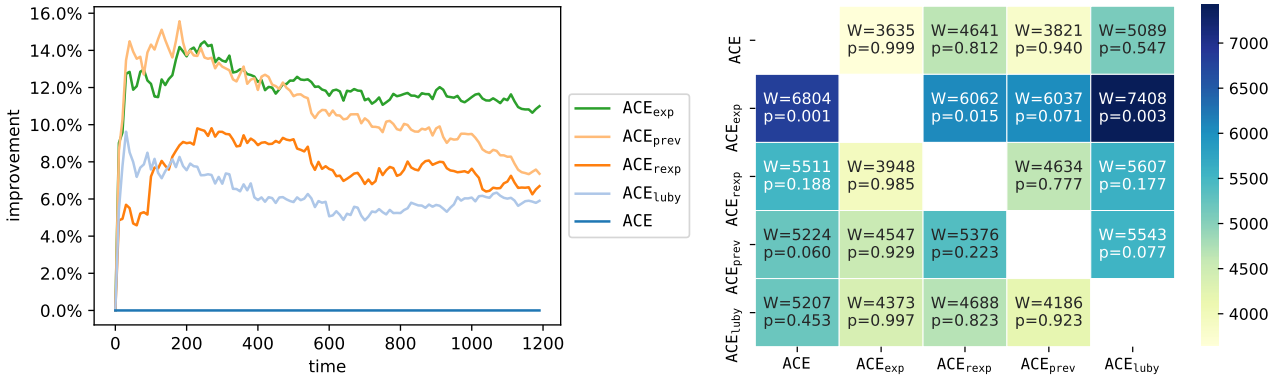
La deuxième fonction de récompense (Équation 9) est dérivée de la première. Au lieu de soustraire $\texttt{min}^t_i$ de $b^t_{i,s}$, nous soustrayons $\texttt{def}^t_i$: cela permet une comparaison plus fine et plus facile avec le solveur de base *default*. En effet, cette récompense montre la capacité de chaque solveur à être respectivement meilleur ou pire que le solveur par défaut avec une récompense positive ou négative. Le solveur par défaut reçoit toujours 0 (considéré comme une valeur neutre) comme récompense. Si un solveur n'a pas trouvé de solution, sa récompense est soit -1 si le solveur par défaut a trouvé une solution, soit 0. Dans le cas où la plus petite et la plus grande borne trouvée sont différentes, nous appliquons le calcul *min-max* où $\texttt{min}^t_i$ est remplacé par $\texttt{def}^t_i$ au numérateur : le résultat étant compris entre -1 et 1, la valeur opposée est considérée comme cohérente avec le but de minimiser la fonction objectif. La dernière condition dépend de la valeur de $e^t_{i,\{\texttt{def}\}}$: si le solveur par défaut a trouvé l'unique borne trouvée, il reçoit 0. Ce n'est pas un problème car nous savons que, dans notre implémentation, la même première solution est toujours trouvée (avec ou sans l'utilisation d'une politique ABD) ; et donc lorsqu'une solution unique a été trouvée, tous les solveurs reçoivent systématiquement 0.

Enfin, la récompense moyenne (basée sur r') de chaque solveur s au temps t est définie comme suit :

$$R^t_s = \frac{1}{|\mathcal{I}|} \times \sum_{i \in \mathcal{I}} r'^t_{i,s} \tag{10}$$

Cette dernière équation sera utile pour dessiner des graphiques (par exemple, Figure 1a) représentant la progression moyenne dans le temps de tout solveur par rapport au solveur par défaut. En raison de la manière dont il est défini, le solveur par défaut correspond à $y = 0$. Ainsi, les solveurs dont la courbe est située au-dessus de $y = 0$ peuvent être considérés comme meilleurs que le solveur par défaut, contrairement à ceux dont la courbe est sous à $y = 0$.

D'autre part, le test du rang signé de Wilcoxon [18] est une autre façon de comparer les solveurs. Le test de Wilcoxon est un test d'hypothèse statistique non paramétrique, ce qui signifie que nous n'avons pas besoin



(a) Impact des politiques ABD sur `ACE`

(b) Carte chaude de Wilcoxon

FIGURE 1 – Comparaison des politiques ABD Ψ

de comparer des échantillons respectant une distribution normale. Ce test est une métrique permettant de mesurer la différence, et la signification de la différence, entre n'importe quelle paire de solveurs. Il teste l'hypothèse nulle selon laquelle deux échantillons proviennent de la même distribution.

Pour appliquer le test de Wilcoxon sur une paire distincte de solveurs (s, s'), nous procédons comme suit :

1. pour chaque instance $i \in \mathcal{I}$, nous calculons la distance entre les récompenses des solveurs s et s' : $d_i = |r^t_{i,s} - r^t_{i,s'}|$
2. avec w_i désignant le rang de chaque instance ordonnée par les valeurs de d_i, nous appliquons le test statistique suivant :

$$\mathcal{W}_{s,s'} = \sum_{i \in \mathcal{I}} \Big[\mathtt{sgn}(r^t_{i,s} - r^t_{i,s'}) \times w_i \Big] \qquad (11)$$

où t est le temps fixé à $1,200$ secondes (le timeout) et `sgn` est la fonction signée qui extrait le signe d'un nombre réel : 1 si le nombre est positif, sinon -1.

Nous avons utilisé la fonction de Wilcoxon de Python [3] incluant le calcul de la *p-value* (critère de signification).

La p-value doit être aussi proche que possible de 0 afin d'écarter l'hypothèse nulle. Dans notre cas, nous voulons vérifier l'hypothèse que s est meilleur que s', et donc, l'hypothèse nulle est : « s n'est pas meilleur que s' ».

Nous présentons les résultats du test de Wilcoxon pour chaque paire de solveurs au moyen d'une carte chaude (par exemple, la Figure 1b). Pour lire la carte chaude, nous devons considérer un solveur s le long de l'axe des ordonnées, un solveur s' le long de l'axe des abscisses. Si la valeur p ($p=$) dans la cellule pour (s, s') est proche de 0, nous pouvons rejeter l'hypothèse nulle, et donc admettre que s est significativement meilleur que s'. Plus le score $\mathcal{W}$ est élevé (Équation 11), plus la différence entre les deux solveurs est importante.

4.2 Optimisation sous contraintes

Notre approche a été évaluée sur un large éventail de problèmes d'optimisation provenant de la distribution XCSP [4, 9]. Nous avons utilisé deux benchmarks : un premier, appelé $\mathcal{I}_{\mathtt{light}}$, correspondant à l'ensemble des instances COP de la compétition XCSP18, résultant en 22 problèmes et 362 instances, et un second, appelé $\mathcal{I}_{\mathtt{full}}$, étendant $\mathcal{I}_{\mathtt{light}}$ avec 27 problèmes supplémentaires et 331 instances provenant des compétitions XCSP17 et XCSP19 (et intégrant quelques nouveaux problèmes PyCSP[3]).

Notre campagne expérimentale est composée de deux parties concernant $\mathcal{I}_{\mathtt{light}}$ et d'une troisième partie concernant $\mathcal{I}_{\mathtt{full}}$. Nous montrons d'abord sur $\mathcal{I}_{\mathtt{light}}$ les résultats expérimentaux concernant notre ensemble primaire de politiques ABD Ψ, ainsi que quelques variantes de Ψ (définies dans un sous-ensemble Ψ'). Le comportement des politiques les plus efficaces est ensuite présenté sur $\mathcal{I}_{\mathtt{full}}$.

4.2.1 Étude de Ψ.

La Figure 1 nous permet de comparer le comportement respectif de `ACE` sans et avec les politiques ABD (de Ψ). En lien avec la Section 4.1, la Figure 1a montre, en fonction du temps, une comparaison directe des politiques ABD par rapport au solveur par défaut (rappelons que `ACE` par défaut est représentée par $y = 0$). Nous pouvons tout d'abord observer un

3. La fonction utilisée est `scipy.stats.wilcoxon` avec le paramètre `alternative = 'greater'` de `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.wilcoxon.html`

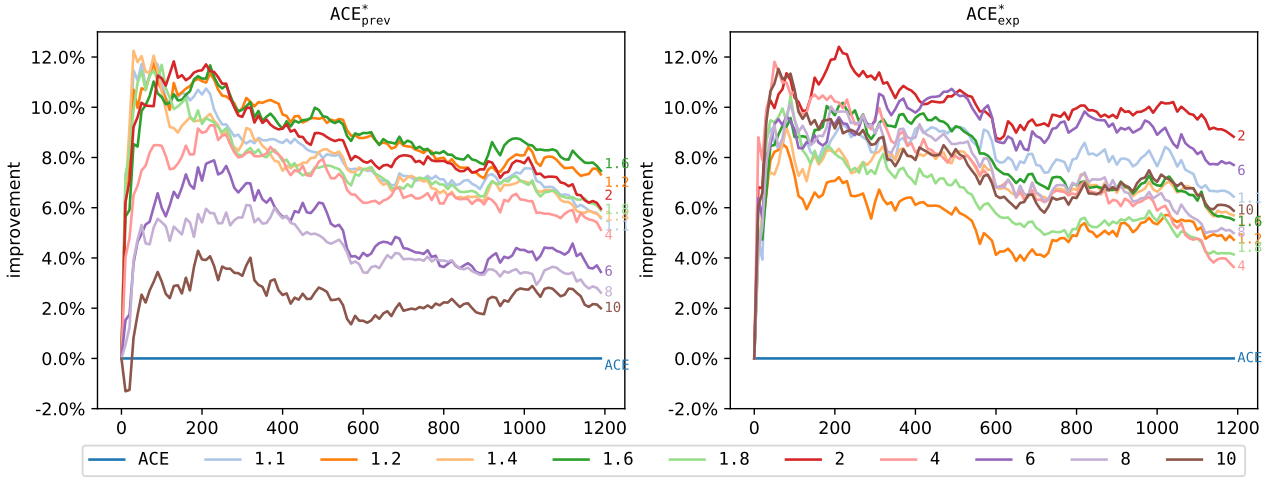


FIGURE 2 – Comparaison des politiques ABD sur Ψ'

intérêt net à utiliser les politiques ABD pendant les 200 premières secondes : une amélioration de 8% (pour les pires politiques) à 14 - 16% (pour $\texttt{ACE}_{\texttt{exp}}$ et $\texttt{ACE}_{\texttt{prev}}$). Nous pouvons également constater que $\texttt{ACE}_{\texttt{exp}}$ reste à (un niveau d'amélioration de) environ 12%, tandis que $\texttt{ACE}_{\texttt{prev}}$ diminue à 8%.

La Figure 1b montre la carte chaude construite à partir du test de Wilcoxon. Rappelons que chaque cellule correspond à un test de Wilcoxon entre une paire de solveurs. Prenons la cellule supérieure la plus à gauche avec $p = 0.1\%$: le test de Wilcoxon renvoie un score $\mathcal{W}_{\texttt{ACE}_{\texttt{exp}},\texttt{ACE}} = 6,804$ avec une p-value très proche de 0. En d'autres termes, le score actuel $\mathcal{W}$ est fortement soutenu par la réfutation de l'hypothèse nulle qui est « $\texttt{ACE}_{\texttt{exp}}$ n'est pas meilleur que $\texttt{ACE}$ ». Notez que $\texttt{ACE}_{\texttt{exp}}$ surpasse les autres politiques lorsque nous les comparons directement par la p-value (voir la ligne $\texttt{ACE}_{\texttt{exp}}$). Comme deuxième choix, on pourrait envisager de sélectionner $\texttt{ACE}_{\texttt{rexp}}$ car son score est plutôt bon (par rapport à $\texttt{ACE}$), mais sa p-value est trop élevée pour rejeter l'hypothèse nulle. Par conséquent, comme le confirme la Figure 1a, $\texttt{ACE}_{\texttt{prev}}$ semble être un meilleur second choix car il présente une p-value raisonnable.

En résumé, les deux meilleures politiques ABD sont $\texttt{ACE}_{\texttt{exp}}$ et $\texttt{ACE}_{\texttt{prev}}$, dont l'agressivité durant les (200) premières secondes est payante (tout en restant robuste après ces trois premières minutes d'exécution).

4.2.2 Étude de Ψ'.

Certaines variantes de ces deux meilleures politiques, $\texttt{ACE}_{\texttt{exp}}$ et $\texttt{ACE}_{\texttt{prev}}$, sont maintenant considérées.

Commençons par généraliser, avec l'introduction d'un paramètre r, les suites originales sous-jacentes de ces deux politiques (jusqu'à présent, r a été fixé à la valeur 2). Nous pouvons alors définir un nouvel ensemble Ψ' de politiques ABD :

$$\texttt{exp}(i, r) = r^{i-1} \tag{12}$$

$$\texttt{prev}(i, r) = \begin{cases} 1, & \text{si } i = 1 \\ \lceil G_{i-1} \times r \rceil, & \text{sinon} \end{cases} \tag{13}$$

$$\Psi' = \left\{ \texttt{abd}_s^r \,\middle|\, (s, r) \in \{\texttt{exp}, \texttt{prev}\} \times \{1.1, 1.2, 1.4, 1.6, 1.8, 2, 4, 6, 8, 10\} \right\} \tag{14}$$

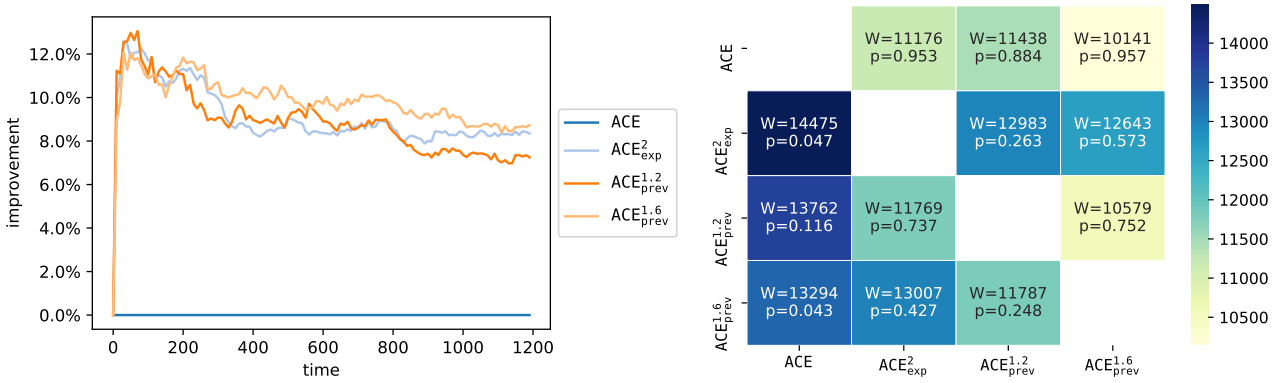
$\texttt{ACE}_s^r$ est le solveur $\texttt{ACE}$ utilisant la suite $\texttt{abd}_s^r$ pour définir sa politique ABD.

La Figure 2 montre le comportement de certaines variantes de $\texttt{ACE}_{\texttt{prev}}$ (figure de gauche) et de $\texttt{ACE}_{\texttt{exp}}$ (figure de droite). Les paramètres (r) utilisés pour les suites sont indiqués sous les figures. On peut observer que l'utilisation de n'importe quelle variante de Ψ' rend le solveur plus robuste, comparé au $\texttt{ACE}$ par défaut (sauf pour les premières secondes de $\texttt{ACE}_{\texttt{prev}}^{10}$). Concernant $\texttt{ACE}_{\texttt{exp}}$, la meilleure variante correspond à $r = 2$, notre valeur initiale. Concernant $\texttt{ACE}_{\texttt{prev}}$, les meilleures variantes sont $\texttt{ACE}_{\texttt{prev}}^{1.2}$ et $\texttt{ACE}_{\texttt{prev}}^{1.6}$ qui dominent nettement pendant $1,200$ secondes. Les variantes les plus agressives $\texttt{ACE}_{\texttt{prev}}^{6}$, $\texttt{ACE}_{\texttt{prev}}^{8}$ et $\texttt{ACE}_{\texttt{prev}}^{10}$ sont moins efficaces.

Après cette nouvelle expérience, nous proposons de conserver trois principales politiques, car elles restent plutôt robustes dans le temps : $\texttt{ACE}_{\texttt{exp}}^{2}$, $\texttt{ACE}_{\texttt{prev}}^{1.2}$ et $\texttt{ACE}_{\texttt{prev}}^{1.6}$.

4.2.3 Étude de Ψ''.

Nous considérons maintenant le benchmark $\mathcal{I}_{\texttt{full}}$, et la sélection Ψ'' :



(a) Impact des politiques ABD sur `ACE`

(b) Carte chaude de Wilcoxon

FIGURE 3 – Comparaison des politiques ABD Ψ''

$$\Psi'' = \left\{ \texttt{abd}^{1.2}_{\texttt{prev}}, \texttt{abd}^{1.6}_{\texttt{prev}}, \texttt{abd}^{2}_{\texttt{exp}} \right\} \tag{15}$$

La Figure 3 nous permet de comparer, sur $\mathcal{I}_{\texttt{full}}$, le comportement de `ACE` sans et avec les meilleures politiques ABD (de Ψ'').

La tendance observée sur $\mathcal{I}_{\texttt{light}}$ reste la même sur $\mathcal{I}_{\texttt{full}}$, mais avec des performances légèrement inférieures. En effet, les trois politiques bénéficient de leur agressivité avec une amélioration de 12 à 13 pour-cent au début, avant une stagnation située autour de 8 à 10 pour-cent ; voir Figure 3a. Bien que $\texttt{ACE}^2_{\texttt{exp}}$ soit légèrement meilleur au tout début, $\texttt{ACE}^{1.6}_{\texttt{exp}}$ obtient les meilleures performances parmi les trois politiques ABD.

La Figure 3b confirme les précédentes observations. En effet, lorsque nous comparons chaque politique ABD au solveur par défaut de `ACE`, il apparaît que $\texttt{ACE}^{1.6}_{\texttt{prev}}$ a la meilleure p-value confirmant le rejet de l'hypothèse nulle. Avec une p-value légèrement supérieure et un meilleur score de Wilcoxon, $\texttt{ACE}^2_{\texttt{exp}}$ présente les meilleures performances. Enfin, bien que $\texttt{ACE}^{1.2}_{\texttt{prev}}$ montre de bonnes performances, sa p-value élevée ($> 10\%$) nous empêche de tirer des conclusions claires sur son intérêt.

En résumé, notre préférence va à $\texttt{ACE}^{1.6}_{\texttt{prev}}$ et à $\texttt{ACE}^2_{\texttt{exp}}$: $\texttt{ACE}^{1.6}_{\texttt{prev}}$ a de bonnes performances (et notamment, une p-value cohérente liée au test de Wilcoxon), et $\texttt{ACE}^2_{\texttt{exp}}$ est assez simple et non intrusif (sans contexte, car il n'y a pas besoin de gérer la séquence des gains liés).

4.2.4 Analyse par famille

Par manque d'espace, les figures ne sont pas affichées et sont brièvement décrites. Cette analyse consiste en une étude par famille de problème de la dernière campagne Ψ'' sur $\mathcal{I}_{\texttt{full}}$. Celle-ci met en valeur de très larges améliorations sur le problème *Fapp*, où une amélioration de 60 à 80 pour-cent est observée pour chacune des deux politiques $\texttt{ACE}_{\texttt{prev}}$. Seuls deux cas montrent que les politiques ABD produisent de moins bonnes performances que le solveur par défaut : *QueenAttacking* et *SteelMillSlab*. Pour les problèmes restants, l'usage d'ABD améliore la performance du solveur avec une amélioration approchant parfois les 40%. Aussi, nous remarquons toujours pour chaque problème, qu'ABD montre de plus claires performances dans les premiers temps d'exécution.

4.3 Optimisation Pseudo-Booléenne

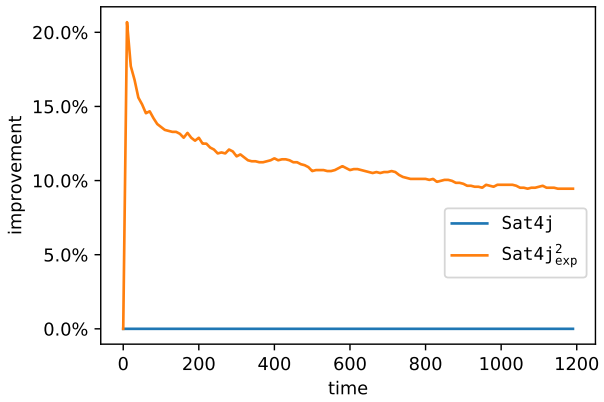


FIGURE 4 – Impact d'ABD sur `Sat4j`

Une dernière expérimentation montre les performances de l'implantation d'ABD sur le solveur Pseudo-Booléen `Sat4j`. Nous résumons, ici aussi, très brièvement les expérimentations et omettons la description du problème d'optimisation Pseudo-Booléen [2]. La Figure 4 montre le résultat de notre expérience avec le solveur par défaut `Sat4j` et le solveur $\texttt{Sat4j}^2_{\texttt{exp}}$, où $\texttt{Sat4j}^2_{\texttt{exp}}$

correspond à l'intégration d'une politique ABD basée sur la suite $\texttt{abd}^2_{\texttt{exp}}$. La Figure 4 affiche la même tendance que celle que nous avons déjà observée pour les solveurs COP : une forte croissance sur les 100 premières secondes (environ 20% d'amélioration), avant un gain plus modeste (environ 10% d'amélioration) à l'approche de la limite de temps.

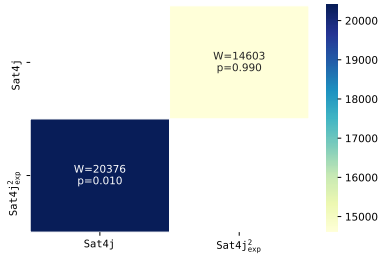


FIGURE 5 – Carte chaude de Wilcoxon

Ce résultat est confirmé par le test de Wilcoxon (Figure 5) où $\texttt{Sat4j}^2_{\texttt{exp}}$ a une p-value proche de 0 et un score plus élevé que le solveur par défaut.

5 Conclusion

Dans cet article, nous avons introduit ABD (*Aggressive Bound Descent*) qui est une technique modifiant de manière agressive la borne de la contrainte objectif. En prenant le risque d'exécuter périodiquement le solveur dans des parties non sûres de l'espace de recherche, nous montrons que des performances expérimentales intéressantes peuvent être obtenues sur des problèmes d'optimisation à la fois sous contraintes et Pseudo-Booléens. Cette politique permet d'obtenir plus rapidement de meilleures bornes que l'approche par défaut de recherche avec retour en arrière sur certains problèmes fortement structurés. Néanmoins, nous pensons que certains raffinements d'ABD pourraient être étudiés, comme par exemple, exploiter davantage l'historique des gains de bornes, ou identifier les suite pertinentes d'écarts de bornes à utiliser en fonction de la structure des problèmes.

Références

[1] C. Bessiere, B. Zanuttini et C. Fernandez : Measuring search trees. *In Proceedings of ECAI'04 workshop on Modelling and Solving Problems with Constraints*, pages 31–40, 2004.

[2] Endre Boros et Peter L. Hammer : Pseudo-boolean optimization. *Discrete Applied Mathematics*, 123(1):155 – 225, 2002.

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

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

[5] E. Demirovic, G. Chu et P. Stuckey : Solution-based phase saving for CP : A value-selection heuristic to simulate local search behavior in complete solvers. *In Proceedings of CP'18*, pages 99–108, 2018.

[6] C. Gomes, B. Selman, N. Crato et H. Kautz : Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1):67–100, 2000.

[7] Daniel Le Berre et Anne Parrain : The sat4j library, release 2.2. *JSAT*, 7:59–6, 01 2010.

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

[9] C. Lecoutre et N. Szczepanski : PyCSP^3 : Modeling combinatorial constrained problems in Python. Rapport technique, CRIL, 2020. Available from `https://github.com/xcsp3team/pycsp3`.

[10] M. Luby, A. Sinclair et D. Zuckerman : Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

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

[12] P. Shaw : Using constraint programming and local search methods to solve vehicle routing problems. *In Proceedings of CP'98*, pages 417–431, 1998.

[13] W.J. van Hoeve et M. Milano : Postponing branching decisions. *In Proceedings of ECAI'04*, pages 1105–1106, 2004.

[14] J. Vion et S. Piechowiak : Une simple heuristique pour rapprocher DFS et LNS pour les COP. *In Proceedings of JFPC'17*, pages 39–45, 2017.

[15] T. Walsh : Search in a small world. *In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, IJCAI '99, page 1172–1177, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[16] T. Walsh : Search in a small world. *In Proceedings of IJCAI'99*, pages 1172–1177, 1999.

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

[18] F. Wilcoxon : Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.