Heuristiques de recherche : un bandit pour les gouverner toutes

Hugues Wattez, Frédéric Koriche, Christophe Lecoutre, Anastasia Paparrizou, Sébastien Tabary

▶ **To cite this version:**

Hugues Wattez, Frédéric Koriche, Christophe Lecoutre, Anastasia Paparrizou, Sébastien Tabary. Heuristiques de recherche : un bandit pour les gouverner toutes. 15es Journées Francophones de Programmation par Contraintes – JFPC 2019, Jun 2019, Albi, France. hal-02414288

HAL Id: hal-02414288
https://hal.science/hal-02414288

Submitted on 16 Dec 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Heuristiques de recherche : un bandit pour les gouverner toutes

Hugues Wattez[1,2*] **Frédéric Koriche**[1,2]
Christophe Lecoutre[1,2] **Anastasia Paparrizou**[1] **Sébastien Tabary**[1,2]
[1] CRIL, UMR CNRS 8188, Lens, France
[2] Université d'Artois
{wattez,koriche,lecoutre,paparrizou,tabary}@cril.fr

Résumé

L'automatisation de la configuration des solveurs a reçu une grande attention ces dernières années notamment pour la capacité à ajuster ses différents paramètres selon l'instance à résoudre. De plus, cette automatisation évite à l'utilisateur le choix d'une configuration appropriée à chaque instance et ainsi, évite une tâche requérant des qualités d'expert et rend plus accessible les outils de la programmation par contraintes. Les techniques basées sur les portfolios ont été grandement étudiées dans le but de configurer les solveurs de contraintes. Son principe est d'entraîner le solveur à résoudre plusieurs instances afin de trouver la meilleure configuration lors de la résolution d'une instance inconnue. Dans cette étude, nous portons notre attention sur le choix de la meilleure stratégie de recherche, à savoir la meilleure heuristique de choix de variables. Nous proposons une structure algorithmique générique qui optimise la sélection des différentes heuristiques de choix de variables grâce à l'apprentissage en ligne, c'est-à-dire sans aucune connaissance apriori. Dans ce but, nous utilisons une technique en apprentissage, appelée les bandits multi-bras, permettant d'automatiser la sélection de l'heuristique la plus appropriée et guidant au mieux la recherche. Nous essayons deux politiques différentes basées sur les bandits multi-bras gérant la proportion d'exploration-exploitation. Une évaluation expérimentale démontre que la machine proposée résulte en un solveur plus efficace et stable.

Abstract

Automating solver's configuration has received a great attention last years as it can adjust the various parameters of a solver to the instance to be solved. In addition, this automation releases the user from choosing the suitable configuration each time, task that requires a great expertise and thus, burdens the widespread of constraint programming technology and tools. Portfolio based techniques have been vastly used to configure constraint solvers. The principle of portfolios is to train themselves by solving many instances in advance , in order to find the right combination of parameters of the solver when solving an unknown instance. In this work, we turn our attention to the selection of the best search strategy, namely the best variable ordering. We propose a generic algorithmic framework that selects among several variable ordering heuristics using on-line learning, i.e. without any prior knowledge. For this purpose, we deploy a machine learning technique, called multi-armed bandits, that allows to automatically select the appropriate heuristic to guide the search. We tried two different policies for multi-armed bandits that manage the proportion of exploration-exploitation. We make a thorough experimental evaluation that demonstrates that the proposed framework results in a more efficient and stable solver.

1 Introduction

Les solveurs de contraintes sont équipés de plusieurs composants génériques dont le choix minutieux ou l'omission peut drastiquement améliorer leur efficacité. Un point important dans le choix d'une bonne configuration concerne la sélection de l'heuristique de choix de variables. Une heuristique savamment choisie permet de réduire significativement la taille de l'espace de recherche parcouru. Mis à part pour un expert, sélectionner une heuristique n'est pas une chose aisée. De plus, étant donnée leur nature, aucune heuristique ne peut être continuellement meilleure qu'une autre. Cela induit qu'utiliser une seule et même heuristique pour un large évantail de problèmes peut être désavantageux quant à l'efficacité de la recherche.

Les travaux que nous menons tendent à remplacer le

*Papier doctorant : Hugues Wattez[1,2] est auteur principal.

rôle de l'humain par un mécanisme intelligent permettant d'orienter automatiquement le solveur au cours de la recherche. A chaque *run*, le solveur décide quelle heuristique choisir. C'est un problème de décisions séquentielles et en tant que tel, il peut être modélisé comme un problème de bandit multi-bras (MAB). En apprentissage par renforcement, la proportion entre exploration et exploitation est spécifiée par différentes politiques. Nous nous concentrons sur epsilon-greedy [16] et upper confidence bound [1] (ucb). Le bandit utilise une fonction de récompense lui permettant de converger vers le meilleur choix. L'apprentissage se fait en ligne, sans l'usage de connaissances apriori. Il s'agit uniquement de la maximisation des récompenses décernées. Ces récompenses sont basées sur le nombre de solutions inférées fausses durant un *run*.

Les expérimentations menées sur de nombreuses séries d'instances montrent que l'utilisation du bandit surpasse l'heuristique la plus efficace.

2 Travaux connexes

La propagation de contraintes est l'un des procédés permettant de réduire efficacement la taille de l'arbre de recherche. En pratique, il est connu que changer de niveau de propagation au cours de la recherche produit de meilleurs résultats que dans le cas statique. Dans ces circonstances, l'application d'un bandit permet de dynamiser ce choix au cours de la recherche sans pour autant faire appel à un expert ou fixer de paramètres. Le procédé appliqué dans [2] correspond au positionnement de différents bandits à différentes profondeurs de l'arbre de recherche. Ainsi, chaque bandit choisit le niveau de propagation le plus adéquat en fonction de sa position dans l'arbre, à l'aide de l'implantation d'ucb. La fonction de récompense est basée sur le temps nécessaire au renforcement de la cohérence locale. Plus le solveur progresse dans la recherche d'une solution, plus les bandits gagnent en connaissances et plus fines deviennent les décisions dans le choix du niveau de propagation optimal. Les résultats de cette étude montrent que cette approche apporte une meilleure efficacité et rend le solveur plus stable.

L'étude proposée dans [17] est une variante du sujet que nous traitons : l'usage des bandits pour le choix d'une heuristique de choix de variables. Dans cette étude, deux algorithmes issus des bandits stochastiques ont été implantés : ucb et Thomson sampling (ts). À chaque nœud de la recherche, une heuristique est choisie par l'algorithme et une récompense lui est ensuite décernée. Cette récompense est basée sur le nombre de nœuds parcourus dans le sous-arbre de recherche. Les conclusions de l'étude montrent que l'usage dynamique des différentes heuristiques d'un solveur le rend plus robuste et permet de meilleures performances que dans le cas d'un choix statique.

En résumé, l'usage des bandits dans le choix d'un niveau de propagation tout comme dans le choix d'une heuristique, évite de solliciter un expert pour fixer les valeurs des paramètres de contrôle du solveur, ce qui permet de garantir de meilleures performances.

3 Réseaux de contraintes

Un *réseau de contraintes* P est composé d'un ensemble fini de variables $\texttt{vars}(P)$, et d'un ensemble fini de contraintes $\texttt{ctrs}(P)$. Nous utilisons n pour définir le nombre de variables. 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. Le *degré* d'une variable x est le nombre de contraintes dans $\texttt{ctrs}(P)$ impliquant x.

Une *solution* de P correspond à l'affectation d'une valeur pour chaque variable de $\texttt{vars}(P)$ de façon à ce que toutes les contraintes dans $\texttt{ctrs}(P)$ 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 réseau de contraintes donné est cohérent ou non. Une procédure classique pour résoudre ce problème NP-complet est d'effectuer une recherche avec retours-arrières dans l'espace des solutions partielles, et d'établir une propriété appelée l'*arc-cohérence généralisée* [11] après chaque décision. Cette procédure, appelée le *maintien de l'arc-cohérence* (MAC pour *Maintaining Arc Consistency*) [14], 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, nous considérons deux cas : 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 la recherche en profondeur de l'espace de recherche 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 suit habituellement l'ordre lexicographique de $\texttt{dom}(x)$. 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 qui ont été adoptées par les solveurs de contraintes

modernes.

Résolution de contraintes Les algorithmes de recherche avec retours-arrières liés à des heuristiques de choix de variables déterministes exhibent un comportement à longue traine (*heavy-tailed behavior*) autant sur les instances aléatoires que les instances CSP du monde réel [6]. Ce problème peut être pallié par l'usage d'une politique naviguant efficacement entre les différentes heuristiques de choix de variables durant la recherche et l'usage d'une stratégie de *redémarrage*. Le redémarrage de la recherche se fait de façon itérative depuis la racine de l'arbre et associe à chaque *run* une heuristique de choix de variables. Conceptuellement, une stratégie de redémarrage est une application $\texttt{restart} : \mathbb{N} \to \mathbb{N}$, où $\texttt{restart}(t)$ est le nombre maximal d'« étapes » qui peuvent être réalisées par l'algorithme de recherche avec retours-arrières pendant un *run* t.

Un solveur de contraintes, équipé de la procédure MAC et d'une stratégie de redémarrage `restart`, construit une séquence d'arbres binaires de recherche $\langle \mathcal{T}^{(1)}, \mathcal{T}^{(2)}, \ldots \rangle$, où $\mathcal{T}^{(t)}$ est l'arbre de recherche exploré par MAC au *run* t. Lorsque le solveur redémarre depuis le début, celui-ci est capable de mémoriser les informations importantes de la séquence $\langle \mathcal{T}^{(1)}, \mathcal{T}^{(2)}, \ldots, \mathcal{T}^{(t-1)} \rangle$. Par exemple, le solveur peut maintenir en cache les *nogoods* qui apparaissent sur la branche courante de l'arbre construit pendant le *run* courant.

Le *cutoff*, qui correspond au nombre d'étapes autorisés, peut être défini par le nombre de nœuds, le nombre de mauvaises décisions [3], le nombre de secondes, ou toutes autre mesures adéquates. Pour une stratégie de redémarrage au *cutoff* fixe, le nombre T d'essais est fixé à l'avance, et $\texttt{restart}(t)$ est une constante pour chaque *run* t, exception faite pour le $T^{\text{ème}}$ *run* qui autorise un nombre illimité d'étapes (dans le but de maintenir un algorithme complet). Cette stratégie est reconnue pour son efficacité en pratique [7], toutefois une bonne valeur pour le *cutoff* doit être trouvée à travers différents essais et erreurs. Alternativement, pour une stratégie de redémarrage dont le *cutoff* est dynamique, le nombre T de *runs* est inconnu, mais `restart` croît géométriquement, ce qui garantit que l'espace entier des solutions partielles est exploré après $O(n)$ *runs*. Une stratégie de redémarrage communément utilisée ces dernières années est basée sur la suite de Luby [9]. Dans cette stratégie, la longueur du *run* t correspond à $u \times l_t$ où u est une constante servant d'unité pour le *run* et

$$l_t = \begin{cases} 2^{k-1}, & \text{if } t = 2^k - 1 \\ l_{t-2^{k-1}+1}, & \text{if } 2^{k-1} \leq t < 2^k - 1. \end{cases}$$

4 Heuristiques de choix de variables

Nous proposons dans cette section une vue d'ensemble des heuristiques de choix de variables. L'heuristique de choix de variables `dom` [8], qui sélectionne les variables en séquence selon l'ordre croissant de la taille de leur domaine, a longtemps été considérée comme la plus robuste des heuristiques. Cependant, il y a une quinzaine d'années, les heuristiques modernes *adaptatives* ont été introduites : elles prennent en compte l'information collectée à travers l'espace de recherche déjà explorée. Les deux premières heuristiques adaptatives génériques proposées ont été `impact` [13] et `wdeg` [4]. La première s'appuie sur une mesure de l'impact sur l'espace de recherche des différentes affectations, et la seconde associe un compteur à chaque contrainte (et indirectement, à chaque variable) indiquant combien de fois chaque contrainte a mené à un *domain wipeout*. Les heuristiques basées sur le comptage [12] et sur l'activité [10] sont aussi deux techniques adaptatives apparues plus récemment.

Dans ce papier, nous portons principalement notre attention aux trois heuristiques adaptatives et génériques `wdeg`, `impact` et `activity`. Nous considérons aussi d'autres heuristiques de référence, les classiques `dom`, `cos` (conflict-ordering search [5]) qui réordonne progressivement les variables. Il est à noter que pour `ddeg` et `wdeg`, nous choisissons les variantes les plus efficaces `dom/ddeg` et `dom/wdeg` comme décrit ci-après :

- `dom/ddeg` sélectionne en priorité la variable avec le plus petit rapport « taille du domaine courant sur degré dynamique » [15]. Le degré dynamique d'une variable x est le nombre de contraintes impliquant x et au moins une autre variable non affectée.
- `dom/wdeg` sélectionne en priorité la variable avec le plus petit rapport « taille du domaine courant sur degré pondéré » [4]. Un poids est calculé pour chaque variable x comme étant la somme des poids de toutes les contraintes impliquant x et au moins une autre variable non affectée. Chaque contrainte c possède un poids, initialisé à 1, qui est incrémenté à chaque *domain wipeout* apparaissant lorsque c est sollicité dans le processus de propagation de contraintes.
- `impact`, ou IBS (*Impact-Based Search*) sélectionne en priorité la variable avec le plus fort impact. L'impact d'une variable x correspond à une mesure de l'importance qu'a x dans la réduction de l'espace de recherche [13]. La taille $\texttt{size}(P)$ de l'espace de recherche de P est le produit de la taille de tous les domaines courants :

$$\texttt{size}(P) = \prod_{x \in \texttt{vars}(P)} |\texttt{dom}(x)|$$

L'impact I de l'affectation d'une variable $x = a$ sur P est calculé comme suit :

$$I(x = a) = 1 - \frac{\texttt{size}(P')}{\texttt{size}(P)}$$

où $P' = AC(P|_{x=a})$ donne le réseau de contraintes après affectation de x à a et l'établissement de l'arc-cohérence. Il est à noter que si P' mène à un échec, alors $I(x = a) = 1$. Cette heuristique peut être aussi utilisée comme choix de valeur.

— `activity`, ou ABS (*Activity-Based Search*) sélectionne en priorité la variable avec la plus forte activité. L'activité d'une variable x est mesurée par le nombre de fois où le domaine de x est réduit durant la recherche [10]. Cette heuristique est basée sur le rôle clé de la propagation en programmation par contraintes et s'appuie sur un facteur de vieillissement pour affaiblir progressivement les précédentes statistiques. Les activités sont initialisées par une distribution aléatoire de probabilités sur l'espace de recherche.
Plus formellement, l'activité $A(x)$ d'une variable x est mise à jour à chaque nœud de l'arbre de recherche peu importe le résultat (succès ou échec) suivant ces deux règles :

i. $A^{P'}(x) = A^P(x) * \gamma$, où $0 \leq \gamma \leq 1$, $|\texttt{dom}^{P'}(x)| > 1$ et $\texttt{dom}^{P'}(x) = \texttt{dom}^P_x$

ii. $A^{P'}(x) = A^P(x) + 1$ si $\texttt{dom}^{P'}(x) \subset \texttt{dom}^P(x)$

— `cos` (*Conflict Ordering Search* [5]). L'idée est d'associer un *timestamp* à l'ensemble des variables participant aux derniers conflits. Quand le solveur doit sélectionner une nouvelle variable non fixée, celle possédant le plus récent *timestamp* est choisie. Au commencement de la recherche, quand aucune variable n'a de *timestamp*, l'heuristique de choix de variables du solveur est utilisée pour guider la recherche. `cos` n'oublie pas les variables conflictuelles et réordonne progressivement ses variables pour donner une priorité à celles apparaissant dans des conflits plus récents. Les résultats expérimentaux ont montré l'efficacité de cette approche, en particulier pour certains problèmes structurés connus comme les problèmes d'ordonnancement.

Lors d'une égalité, qui correspond au cas de deux variables (au moins) jugées équivalentes par l'heuristique, on peut utiliser un second critère (par exemple, le degré dynamique de chaque variable). Cependant, pour les heuristiques adaptatives, il est habituel d'utiliser `lexico` donnant la priorité à la première variable ayant le meilleur score.

5 Les modèles de bandits

Comme indiqué dans la Section 3, le processus de résolution de contraintes par l'application de la politique de redémarrage peut être vu comme une séquence $\langle 1, 2, \cdots, T \rangle$ de *runs*. Pour les politiques de redémarrage mentionnées précédemment, la séquence de *runs* est finie, mais l'horizon T n'est pas nécessairement connu à l'avance. Durant chaque *run* t, le solveur appelle l'algorithme MAC pour construire un arbre de recherche $\mathcal{T}_t$, dont la taille est déterminée par le *cutoff* de la politique de redémarrage. Si le solveur n'a accès qu'à une heuristique de choix de variables, dit H_1, il va exécuter MAC avec H_1 après chaque redémarrage. Toutefois, si le solveur a accès à d'autres heuristiques de choix de variables, il doit faire face à un choix fondamental à chaque *run* t : soit appeler MAC en exploitant l'heuristique H_i maximisant l'efficacité du solveur au *run* t ou appeler MAC en explorant une heuristique dont les résultats paraissent moins fructueux mais qui a tout de même ses chances de battre l'heuristique la plus efficace actuellement.

5.1 Généralité et notations

Par essence, la tâche d'exploration et d'exploitation d'un solveur de contraintes avec redémarrages est de décider, à chaque *run* t, quelle heuristique H_i utiliser. En empruntant la terminologie de la théorie des jeux, une stratégie d'*exploration-exploitation* sélectionne à chaque *run* t l'une des actions disponibles dans $A = \{H_1, ..., H_K\}$ où K est le nombre d'actions disponibles. Le problème est d'équilibrer la part d'exploration par rapport à la part d'exploitation de la meilleure heuristique, afin d'estimer au mieux la probabilité qu'a chacune des heuristiques d'apporter le meilleur gain pour la recherche.

Cette succession de décisions peut donc être représentée par un problème de bandit multi-bras stochastique. Nous partons du principe que ce problème est stochastique et que donc, la distribution de probabilités de succès des différentes heuristiques est fixe. Le comportement d'un bandit stochastique peut être décrit par le cadre général suivant :

1. (choix) A chaque début de *run*, le bandit est appelé à sélectionner l'heuristique qui sera utilisée pour poursuivre la recherche.
2. (mise-à-jour) À chaque fin de *run*, le bras courant (représentant l'heuristique) est mis à jour avec la récompense qui lui est décernée.

Avant de présenter nos différents modèles de bandit, nous devons introduire ces notations :

— $\mu(a)$ est la moyenne (réelle) de la récompense $r(a)$;

— μ^* est la moyenne de la meilleure action (heuristique) ;
— $\Delta_a = \mu^* - \mu(a)$ est l'intervalle de sous-optimalité du choix a ;
— $n_t(a)$ est le nombre de fois où le joueur a sélectionné a durant les t premières étapes (*runs*) ;
— $\widehat{\mu}_t$ est la moyenne empirique de $r(a)$ sur les $n_t(a)$ jeux.

Enfin, la performance d'un bandit sur la totalité T de son exécution est défini comme la différence entre l'espérance de la récompense que l'on obtiendrait en utilisant T fois le meilleur bras et l'espérance de la récompense après T essais effectués par le bandit. Il s'agit du *pseudo-regret* :

$$\mathcal{R}_T = \max_{a \in A} \sum_{t=1}^{T} \mathbb{E}[r_t(a)] - \sum_{t=1}^{T} \mathbb{E}[r_t(a_t)]$$

Cette mesure indique la performance théorique qu'un bandit nous garantit en appliquant sa politique.

5.2 Fonction de récompense

Nous décrivons dans cette partie la fonction de récompense nous permettant de récompenser le plus fidèlement possible les différents bandits décrits ci-après. Notre fonction de récompense est définie par le nombre de solutions inférées fausses (`sif`) lors d'un *run*. Ce nombre de solutions est calculé lors de chaque *domain wipeout* par le produit de la taille des domaines de chaque variable non affectée du réseau de contraintes P' :

$$\texttt{sif}(P') = \prod_{x \in \texttt{FutVars}(P')} |\texttt{dom}(x)|$$

où $\texttt{FutVars}(P')$ est l'ensemble des variables non affectées. Ainsi, nous récompensons le bandit par la somme de ces solutions inférées fausses normalisée par le nombre total de solutions du problème :

$$r_t(a) = \frac{\log(\sum_{\nu \in \texttt{dwo}(T_t)} \texttt{sif}(P_\nu))}{\log(\texttt{size}(P))} \quad (1)$$

Une mise à l'échelle logarithmique est nécessaire afin d'obtenir une récompense plus équilibrée entre 0 et 1. Plus le nombre de solutions inférées fausses est important, meilleure est la qualité de la récompense attribuée à l'heuristique de choix de variables.

5.3 Le modèle d'epsilon-greedy

Epsilon-greedy [16] (Algorithme 1) correspond à la stratégie la plus simple et naïve afin de venir à bout du problème des bandits stochastiques. Il suffit de préciser une valeur $\varepsilon \in [0, 1]$, permettant de savoir dans quelle proportion nous souhaitons explorer ou exploiter.

Algorithme 1 : epsilon-greedy

pour chaque $t = 1$ *à* T **faire**
  Choix du paramètre $\varepsilon_t \in [0, 1]$
  Choix de $a_t \in \{H_1, ..., H_K\} =$
  $\begin{cases} \text{aléatoirement,} & \text{avec la probabilité } \varepsilon_t \\ \text{maximisant } \widehat{\mu}_{t-1}(a_t), & \text{sinon} \end{cases}$

En admettant que $d < \min_{j \neq i^*} \Delta_j$ et qu'on initialise $\varepsilon_t = \frac{K}{(d^2 t)}$, epsilon-greedy accomplit un *pseudo-regret* en $O(\frac{K \ln(T)}{d^2})$. Une borne inférieure d basée sur l'intervalle minimal doit être connue à l'avance.

En application à ce sujet, cela revient à dire qu'au bout de T *runs* à choisir parmi K heuristiques, epsilon-greedy garantit un pseudo-regret en $O(\frac{K \ln(T)}{d^2})$, et que donc, le regret cumulé est borné par cette précédente formule.

5.4 Le modèle d'ucb

Avec ucb (upper confidence bound) [1], il n'est plus nécessaire de préciser un ε afin de faire varier l'exploration et l'exploitation. En effet, la politique appliquée (Algorithme 2) lui permet de gérer la part d'exploration et d'exploitation sans l'usage d'une fonction aléatoire.

Algorithme 2 : ucb

pour chaque $t = 1$ *à* T **faire**
  Choix de $a_t \in \{H_1, ..., H_K\}$ maximisant
  $\widehat{\mu}_{t-1}(a_t) + \sqrt{\frac{8 \ln(t)}{n_{t-1}(a_t)}}$

$\widehat{\mu}_{t-1}(i)$ correspond à la partie exploitation de l'algorithme, tandis que $\sqrt{\frac{8 \ln(t)}{n_{t-1}(i)}}$ correspond à un poids supplémentaire donné au bras notamment quand celui-ci n'a pas été souvent sélectionné (partie exploration).

Ucb accomplit un *pseudo-regret* en $O(\sqrt{KT})$.

6 Évaluation expérimentale

Nous menons une évaluation sur la base d'une grande diversité de classes de problèmes (37) provenant de la compétition XCSP3 (`http://xcsp.org`), incluant : AllInterval, Bibd, Blackhole, CarSequencing, ColouredQueens, CostasArray, CoveringArray, Crossword, DiamondFree, Dubois, Fischer, frb, gp10, GracefulGraph, Haystacks, KnightTour, Langford, MagicHexagon, MagicSquare, MarketSplit, MultiKnapsack, NumberPartitioning, Ortholatin, Primes,

		vbs	epsilon	ucb	rand	dom/wdeg	impact	activity	dom/ddeg	cos	dom
All Interval	**commun (32)**	107	117	118	117	2977	108	108	173	110	117
	total (#res)	**107 (32)**	117 (32)	118 (32)	117 (32)	2977 (32)	**108 (32)**	**108 (32)**	173 (32)	110 (32)	117 (32)
Bibd	**commun (2)**	6	6	6	6	6	6	6	6	6	6
	total (#res)	**3737 (7)**	1508 (6)	1758 (6)	**3978 (7)**	3506 (3)	309 (4)	2180 (5)	6 (2)	93 (3)	1297 (5)
Blackhole	**commun (0)**	0	0	0	0	0	0	0	0	0	0
	total (#res)	**891 (13)**	2130 (11)	64 (10)	4775 (11)	1471 (2)	**892 (13)**	68 (7)	0 (0)	11 (5)	717 (5)
Car Sequencing	**commun (1)**	2	3	3	3	3	2	2	3	2	3
	total (#res)	**2592 (14)**	5250 (10)	**9479 (14)**	9508 (11)	3124 (13)	5348 (8)	1971 (6)	3 (1)	11 (2)	596 (2)
Coloured Queens	**commun (5)**	10	14	15	14	15	10	11	15	11	15
	total (#res)	25 (6)	**299 (7)**	42 (6)	**314 (7)**	15 (5)	10 (5)	11 (5)	15 (5)	11 (5)	29 (6)
Costas Array	**commun (4)**	9	13	13	13	12	10	10	12	35	1197
	total (#res)	**702 (9)**	1767 (9)	3662 (9)	1356 (9)	1793 (9)	1331 (8)	770 (8)	**719 (9)**	2420 (7)	1197 (4)
Covering Array	**commun (2)**	4	5	6	6	6	4	4	6	4	6
	total (#res)	41 (6)	216 (6)	57 (6)	**31 (6)**	3500 (6)	41 (6)	4 (2)	6 (2)	74 (3)	1202 (3)
Crossword	**commun (6)**	16	21	21	22	21	47	16	21	1365	536
	total (#res)	**2382 (12)**	378 (9)	2880 (10)	1150 (10)	**3921 (12)**	3148 (10)	1460 (9)	2570 (11)	1833 (7)	536 (6)
Diamond Free	**commun (0)**	0	0	0	0	0	0	0	0	0	0
	total (#res)	**128 (6)**	256 (6)	847 (6)	578 (6)	2017 (6)	2146 (6)	339 (6)	1372 (4)	0 (0)	**128 (6)**
Dubois	**commun (2)**	74	52	186	78	1286	207	74	430	88	89
	total (#res)	2028 (5)	**1708 (6)**	3248 (5)	689 (4)	2353 (3)	3968 (4)	1097 (4)	2677 (3)	3769 (5)	1624 (4)
Fischer	**commun (0)**	0	0	0	0	0	0	0	0	0	0
	total (#res)	2251 (8)	**9192 (9)**	**9156 (9)**	**9681 (9)**	8235 (7)	2869 (8)	136 (6)	7967 (7)	774 (1)	0 (0)
frb	**commun (0)**	0	0	0	0	0	0	0	0	0	0
	total (#res)	**5096 (7)**	2039 (4)	19 (3)	2255 (4)	2097 (6)	34 (3)	17 (3)	**4280 (6)**	236 (1)	0 (0)
gp10	**commun (0)**	0	0	0	0	0	0	0	0	0	0
	total (#res)	**4269 (5)**	**7461 (4)**	5242 (3)	5755 (3)	1724 (1)	3714 (3)	2685 (3)	0 (0)	0 (0)	0 (0)
Graceful Graph	**commun (11)**	24	33	33	33	33	24	50	33	132	33
	total (#res)	**943 (20)**	4192 (19)	2086 (19)	3107 (19)	253 (18)	3337 (20)	50 (11)	2576 (20)	460 (13)	**971 (20)**
Haystacks	**commun (1)**	2	2	3	3	3	2	2	3	2	3
	total (#res)	**42 (4)**	50 (4)	11 (3)	127 (4)	8 (2)	**42 (4)**	437 (4)	473 (4)	33 (3)	3 (1)
Knight Tour	**commun (3)**	16	26	97	25	23	16	20	777	153	446
	total (#res)	**110 (5)**	2317 (5)	984 (5)	1457 (5)	1757 (4)	**110 (5)**	20 (3)	777 (3)	153 (3)	446 (3)
Langford	**commun (10)**	152	409	486	365	215	519	482	159	3225	3845
	total (#res)	**219 (13)**	1285 (13)	601 (13)	1701 (13)	**283 (13)**	748 (13)	499 (12)	179 (12)	3225 (10)	3845 (10)
Magic Hexagon	**commun (0)**	0	0	0	0	0	0	0	0	0	0
	total (#res)	**293 (5)**	**715 (5)**	766 (5)	810 (5)	924 (5)	1290 (4)	1018 (4)	1357 (5)	2221 (1)	0 (0)
Magic Square	**commun (0)**	0	0	0	0	0	0	0	0	0	0
	total (#res)	**2843 (24)**	3362 (23)	542 (21)	1386 (22)	**2471 (23)**	12502 (17)	2173 (18)	1325 (12)	0 (0)	0 (0)
Market Split	**commun (10)**	109	929	386	1084	5374	913	221	948	621	953
	total (#res)	**109 (10)**	929 (10)	386 (10)	1084 (10)	5374 (10)	913 (10)	**221 (10)**	948 (10)	621 (10)	953 (10)
Multi Knapsack	**commun (24)**	63	83	81	84	1116	130	63	111	84	91
	total (#res)	**93 (27)**	170 (27)	**149 (27)**	156 (27)	1116 (24)	1042 (25)	157 (27)	126 (26)	3520 (27)	3911 (26)
Number Partitioning	**commun (6)**	20	21	22	22	20	20	41	22	92	880
	total (#res)	**148 (19)**	497 (19)	**224 (19)**	722 (19)	626 (19)	589 (19)	4681 (17)	6031 (16)	3101 (9)	880 (6)
Ortholatin	**commun (1)**	6	7	16	13	8	17	90	185	17	6
	total (#res)	3959 (4)	**152 (5)**	199 (5)	183 (5)	830 (2)	1744 (2)	3194 (2)	185 (1)	17 (1)	1318 (3)
Primes	**commun (15)**	32	44	44	45	44	33	33	45	33	45
	total (#res)	**55 (23)**	74 (23)	76 (23)	76 (23)	**71 (23)**	86 (23)	99 (23)	821 (23)	33 (15)	45 (15)
Quasi Group	**commun (3)**	6	9	9	9	9	6	6	9	6	9
	total (#res)	**213 (4)**	384 (4)	425 (4)	769 (4)	474 (4)	6 (3)	**213 (4)**	1153 (4)	6 (3)	9 (3)
Queens Knights	**commun (6)**	19	22	21	21	21	19	19	4027	19	86
	total (#res)	**1838 (18)**	8174 (18)	5619 (18)	4685 (18)	6618 (17)	**1840 (18)**	681 (12)	4027 (6)	2102 (18)	86 (6)
qwh	**commun (5)**	12	17	16	16	15	22	12	38	79	2290
	total (#res)	**14722 (53)**	25982 (47)	31605 (49)	29480 (49)	**24042 (52)**	14544 (20)	21269 (48)	18924 (37)	96 (6)	2302 (6)
Radar Surveillance	**commun (9)**	28	29	29	29	28	29	29	29	29	29
	total (#res)	1662 (39)	**7528 (42)**	10642 (41)	11413 (41)	1664 (39)	54 (13)	142 (14)	417 (12)	2759 (19)	29 (9)
Rlfap	**commun (0)**	0	0	0	0	0	0	0	0	0	0
	total (#res)	**3524 (12)**	4725 (11)	3813 (11)	1697 (10)	**3524 (12)**	3353 (10)	2451 (6)	0 (0)	2483 (3)	0 (0)
RoomMate	**commun (9)**	2586	2647	2705	2680	2774	2651	2637	3020	2602	2644
	total (#res)	**2586 (9)**	**2647 (9)**	2705 (9)	2680 (9)	2774 (9)	2651 (9)	2637 (9)	3020 (9)	2602 (9)	2644 (9)
Social Golfers	**commun (22)**	59	77	77	77	78	60	60	81	625	804
	total (#res)	575 (27)	612 (27)	**453 (27)**	467 (27)	912 (27)	2494 (26)	851 (27)	1242 (25)	625 (22)	809 (23)
Sports Scheduling	**commun (5)**	13	29	19	51	37	26	13	28	781	593
	total (#res)	**87 (6)**	372 (6)	155 (6)	526 (6)	350 (6)	715 (6)	**87 (6)**	2452 (6)	781 (5)	593 (5)
Steiner3	**commun (3)**	6	9	9	9	9	7	6	9	6	9
	total (#res)	**10 (4)**	13 (4)	13 (4)	13 (4)	13 (4)	14 (4)	**10 (4)**	13 (4)	11 (4)	9 (3)
Strip Packing	**commun (0)**	0	0	0	0	0	0	0	0	0	0
	total (#res)	384 (4)	1499 (5)	2872 (5)	**1460 (5)**	600 (4)	225 (2)	8 (2)	275 (1)	38 (1)	0 (0)
Subiso-morphism	**commun (1)**	4	11	5	7	5	141	4	4	2297	30
	total (#res)	1126 (8)	2799 (8)	**5098 (9)**	2902 (8)	3736 (7)	2649 (5)	8 (2)	1294 (8)	2312 (2)	30 (1)
Travelling Salesman	**commun (43)**	218	632	686	557	298	1852	613	422	624	4079
	total (#res)	**347 (45)**	742 (45)	910 (45)	641 (45)	**442 (45)**	3099 (45)	819 (45)	557 (45)	1020 (45)	4079 (43)
Global	**commun (241)**	3616	5281	5123	5400	14447	6894	4643	10630	13061	18856
	total (#res)	**60153 (513)**	**101562 (498)**	**106923 (497)**	**107748 (497)**	95613 (474)	77982 (413)	52585 (406)	67977 (371)	37575 (300)	30418 (275)
		vbs	epsilon	ucb	rand	dom/wdeg	impact	activity	dom/ddeg	cos	dom

TABLE 1 – Temps cpu (cumulé en secondes) et nombre d'instances résolues (#res) pour différentes stratégies de recherche.

QuasiGroup, QueensKnights, qwh, RadarSurveillance, RectPacking, Rlfap, RoomMate, SocialGolfers, SportsScheduling, Steiner3, StripPacking, Subisomorphism and TravellingSalesman, totalisant 746

instances. Celles-ci sont exécutées sur des nœuds possédant un processeur Intel Xeon 2,66 GHz et une mémoire de 32 GB. Nous utilisons le solveur AbsCon (`http://www.cril.fr/~lecoutre/#/softwares`) dans lequel nous avons intégré les différents bandits présentés précédemment. AbsCon est configuré pour utiliser AC comme niveau de cohérence, pour suivre le suite de Luby comme politique de redémarrage basée sur le nombre de nœuds visités et enfin, le temps limite d'exécution est fixé à une heure.

Pour mener à bien cette évaluation, nous testons l'ensemble des heuristiques de choix variables suivantes : `dom/wdeg`, `dom/ddeg`, `dom`, `activity`, `impact` et `cos`. Nous testons nos bandits `epsilon` et `ucb` dont les bras sont composés à partir des différentes heuristiques citées précédemment. La valeur ε du bandit `epsilon` a été fixée à 0.9 par recherche linéaire. Les bandits sont récompensés par la fonction présentée dans la Section 5.2. De nombreuses fonctions de récompenses ont été testées, mais seule celle-ci a retenu notre attention. En plus de ces bandits, nous expérimentons une politique basée sur les bandits mais dont le choix des différents bras se fait de façon aléatoire, que nous appelons `rand`. Enfin, nous agrémentons ces tests de la génération d'un `vbs` (virtual best solver) : il se compose lui aussi des six précédentes heuristiques, mais ne garde que les meilleurs résultats de chacune d'entre elles. À travers ces candidats témoins, nous voyons la capacité qu'ont nos bandits à être sensiblement plus efficaces que la politique de `rand`, mais aussi à quel point ceux-ci se rapprochent de l'optimalité du `vbs`.

Il est à noter que, pour le cas d'`epsilon` et `rand`, les expérimentations sont menées trois fois avec trois graines différentes pour leur fonction de randomisation. Les expérimentations sont agrégées en prenant la médiane des temps pour chaque instance. Cette mesure nous semble la plus juste dans le sens où :

— quand une instance n'est résolue qu'une seule fois sur les trois, celle-ci est considérée comme irrésolue ;
— dans le cas où deux graines ont réussi à résoudre l'instance, celle possédant le temps le plus élevé est conservée ;
— enfin, dans le cas où pour chaque graines l'instance est résolue, le temps médian semble tout aussi appropriée.

Il est aussi important de préciser que, dans les stratégies hybrides, les heuristiques sont isolées les une des autres afin de ne pas se perturber entre elles. Il s'agit d'un point important car, dans de précédentes expérimentations, ce partage que nous pensions collaboratif s'est avéré être une perturbation négative et néfaste à côté des résultats que nous décrirons dans la suite de cette section.

Le Tableau 1 montre (de gauche à droite), le `vbs`, nos bandits (`epsilon` et `ucb`), la stratégie choisissant aléatoirement ses bras `rand` et enfin les six heuristiques simples. Les colonnes sont ordonnées de façon décroissante par rapport à leurs résultats globaux. Pour chaque variante, nous présentons le nombre d'instances qu'elle a résolues et le temps CPU requis pour les résoudre, mais aussi le temps CPU requis pour résoudre les instances résolues par toutes les méthodes. Nous observons de façon globale que, les stratégies hybrides (`epsilon`, `ucb` et `rand`) ont la plupart du temps résolu plus d'instances que les stratégies simples. Nous remarquons que les bandits ne dépassent que de peu la stratégie aléatoire. Ce phénomène s'explique par le fait que, contrairement à `rand` qui donne arbitrairement et équitablement sa chance à chaque bras, les bandits nécessitent un temps d'apprentissage afin d'équilibrer les parts d'exploration et d'exploitation. De plus, malgré l'étude de différentes fonctions de récompense et le choix de la plus prometteuse, il est compliqué d'en construire une qui puisse s'adapter à chacune des instances.

De façon plus détaillée, nous remarquons que pour certains problèmes le `vbs` est surclassé par les bandits : ColouredQueens, Dubois, Ortholatin, RadarSurveillance, StripPacking et Subisomorphism. Pour la plupart des autres cas, les bandits égalent ou surclassent la meilleure des heuristiques (`dom/wdeg`), excepté pour frb, Rlfap, qwh où ils perdent quelques instances. Les résultats globaux montrent que les bandits conservent la stabilité de `dom/wdeg` et dépassent même cette heuristique par la résolution de 24 instances supplémentaires et la réduction du temps commun de 67%.

Pour les instances "faciles" (c'est-à-dire résolues par toutes les méthodes), les stratégies hybrides s'en sortent avec un temps proche de la meilleure des heuristiques (`impact`).

La Figure 1 permet de visualiser les résultats du Tableau 1 à travers un *cactus plot* qui montre le nombre d'instances résolues par chaque stratégie par temps de résolution croissant. Sur ce genre de graphique, l'important est de projeter l'extrémité de la courbe représentant une stratégie de recherche sur l'axe des abscisses, renseignant ainsi le nombre d'instances résolues. La projection sur l'axe des ordonnées n'est que peu informative puisqu'elle nous informe du temps qu'a pris la plus longue instance de la stratégie correspondante.

Connu de la littérature, `dom/wdeg` est la plus efficace des heuristiques et cette particularité est mise en évidence à travers cette figure. En effet, nous voyons que `dom/wdeg` est loin devant ses cinq homologues. La courbe que nous observons la plus à droite correspond au témoin `vbs`, dominant l'ensemble des stratégies. Entre `dom/wdeg` et le `vbs` se situent les bandits et la

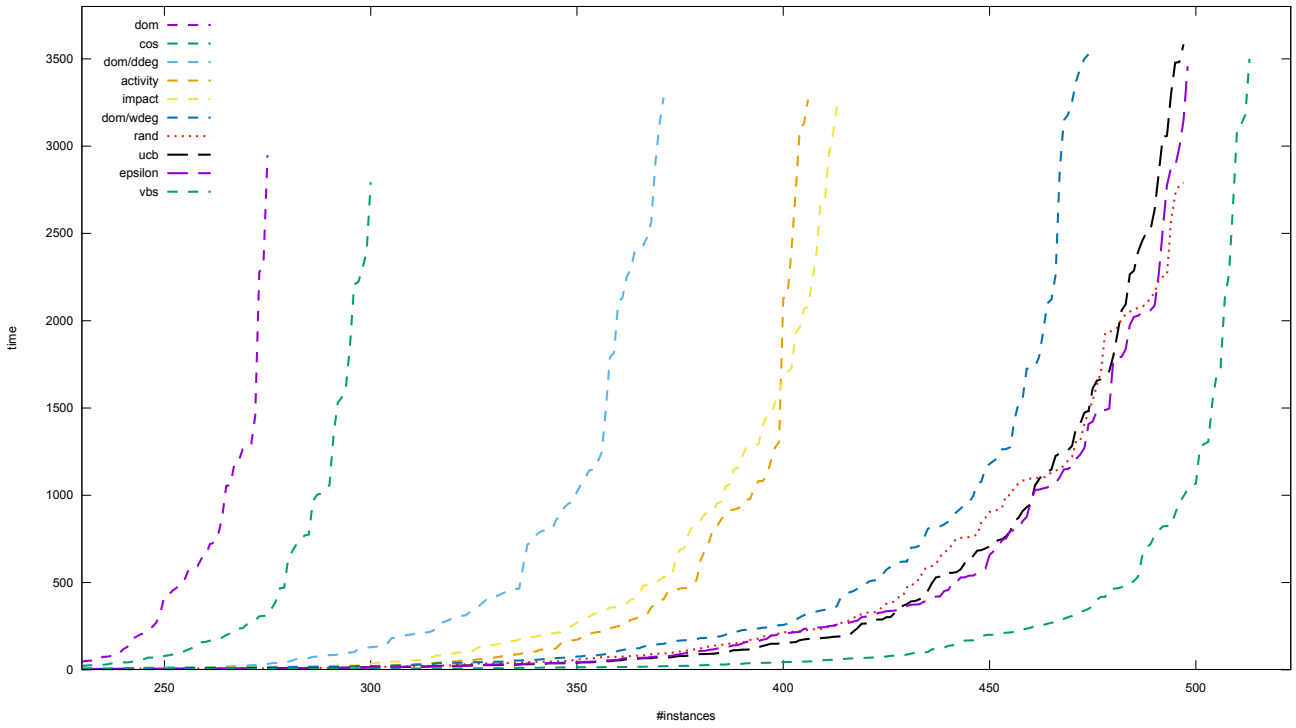

FIGURE 1 – Comparaison des heuristiques simples, de `rand`, des bandits et de `vbs`

stratégie aléatoire : tout comme dans le Tableau 1, nous constatons des résultats assez semblables entre ces stratégies hybrides. Il est tout de même rassurant de voir que la composition des heuristiques permette des résultats notables, visibles à la fois sur la Figure 1 et dans le Tableau 1.

7 Conclusion

Dans notre approche, nous tirons profit du mécanisme de redémarrage afin d'y placer une stratégie issue de l'apprentissage par renforcement, capable de sélectionner l'heuristique de choix de variables la plus adaptée à chaque *run*. Ainsi, par un jeu d'exploration et d'exploitation dont la politique des bandits multi-bras sait trouver un équilibre, nous observons une amélioration de l'efficacité du solveur. Il se montre ainsi plus efficace, dans le sens où il est capable de résoudre une plus grande diversité d'instances. Bien que la différence entre la stratégie basée sur la randomisation du choix des heuristiques et celle des bandits donnent des résultats très proches, le déterminisme de la politique d'`ucb` nous permet d'assurer ces résultats, contrairement à la variance des résultats pour les stratégies basées sur une fonction aléatoire.

Cette faible différence s'explique par le fait qu'il est compliqué de trouver une fonction de récompense suffisamment générique pour un bandit dans le cas d'un solveur de contraintes. En effet, au même titre qu'une heuristique ne peut pas être systématiquement la meilleure sur l'ensemble des instances, il s'agit ici de la même problématique pour les fonctions de récompense. Il est difficile pour une telle fonction d'être capable de discriminer de façon certaine les résultats préliminaires d'une heuristique, sans en avoir vu l'exécution complète. Celle-ci peut très bien être inefficace en début de recherche, puis trouver une certaine stabilité au bout d'un certain nombre de redémarrages, là où le bandit aurait écarté la piste de cette heuristique.

Peut-être serait-il plus judicieux de réduire cet ensemble d'heuristiques, par exemple à un couple d'heuristiques complémentaires, afin de récompenser plus finement le bandit ?

Remerciements. Les auteurs remercient les relecteurs anonymes pour leurs remarques constructives ayant permis d'améliorer la qualité de ce papier.

Références

[1] Peter AUER, Nicolò CESA-BIANCHI et Paul FISCHER : Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2):235–256, May 2002.

[2] A. BALAFREJ, C. BESSIERE et A. PAPARRIZOU : Multi-armed bandits for adaptive constraint propa-

gation. *In Proceedings of IJCAI'15*, pages 290–296, 2015.

[3] 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.

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

[5] S. Gay, R. Hartert, C. Lecoutre et P. Schauss : Conflict ordering search for scheduling problems. *In Proceedings of CP'15*, pages 140–148, 2015.

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

[7] C. Gomes, B. Selman et H.A. Kautz : Boosting combinatorial search through randomization. *In Proceedings of AAAI'98*, pages 431–437, 1998.

[8] R. Haralick et G. Elliott : Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[9] Michael Luby, Alistair Sinclair et David Zuckerman : Optimal speedup of las vegas algorithms. *Inf. Process. Lett.*, 47(4):173–180, septembre 1993.

[10] L. Michel et P. Van Hentenryck : Activity-based search for black-box constraint programming solvers. *In Proceedings of CPAIOR'12*, pages 228–243, 2012.

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

[12] G. Pesant, C.-G. Quimper et A. Zanarini : Counting-based search : Branching heuristics for constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 43:173–210, 2012.

[13] P. Refalo : Impact-based search strategies for constraint programming. *In Proceedings of CP'04*, pages 557–571, 2004.

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

[15] B. Smith et S. Grant : Trying harder to fail first. *In Proceedings of ECAI'98*, pages 249–253, Brighton, UK, 1998.

[16] Richard Sutton et Andrew G. Barto : Reinforcement learning : An introduction. 9:1054, 02 1998.

[17] W. Xia et R. H. C. Yap : Learning robust search strategies using a bandit-based approach. pages 6657–6665, 2018.