Perturbation des heuristiques de branchement dans la résolution de contraintes[*]

Hugues Wattez[†] **Frederic Koriche** **Anastasia Paparrizou**

CRIL, Université d'Artois & CNRS,
UFR des Sciences Jean Perrin, Rue Jean Souvraz SP 18, F-62307 Lens, France
{wattez,koriche,paparrizou}@cril.fr

Résumé

L'heuristique de choix de variables est l'un des mécanismes clés d'un solveur de contraintes. Au cours des deux dernières décennies, des heuristiques efficaces ont été proposées, adaptant l'ordre des variables au fur et à mesure que la recherche progresse. Dans le même temps, des méthodes de redémarrage et de randomisation ont été conçues pour rendre les solveurs plus robustes. Alors que les méthodes de redémarrage sont maintenant bien comprises, choisir *comment* et *quand* randomiser une heuristique donnée reste un problème ouvert. Dans cet article, nous présentons plusieurs stratégies de perturbation conceptuellement simples pour incorporer des choix aléatoires dans la résolution de contraintes avec redémarrages. La quantité de perturbation est contrôlée et apprise par des bandits sous diverses politiques d'exploration (stationnaire ou non stationnaire). L'évaluation expérimentale montre une amélioration significative des performances pour les heuristiques perturbées par rapport à leurs homologues d'origine.

1 Introduction

Un solveur de contraintes est généralement composé d'un algorithme de recherche avec retour-arrière avec une heuristique de branchement pour guider la recherche, une procédure de filtrage pour réduire l'espace de recherche et un cache mémorisant les branches infructueuses, les "*nogoods*". L'heuristique de branchement est notamment en charge de l'ordonnancement des variables dans l'arbre de recherche. C'est ce qu'on appelle l'heuristique de choix de variables. La sélection de l'heuristique de choix de variables pour résoudre un réseau de contraintes donné est essentiel car le temps de résolution peut varier de façon exponentielle selon l'heuristique. Les heuristiques proposées ces dernières années ont vu cet effet s'atténuer.

Les algorithmes de recherche sont vulnérables à un tel comportement en raison de la nature combinatoire des problèmes. Au début des années 2000, les différences exponentielles des temps ont été beaucoup étudiées avec la découverte des phénomènes de transition de phase [6] et des distributions à queue longue [4]. Ces différences ont été réduites en introduisant des politiques de redémarrage, la randomisation et l'enregistrement de *nogoods* pendant la recherche. Alors que les redémarrages et les *nogoods* sont bien compris dans les solveurs de contraintes, l'usage de la randomisation reste floue. Les solveurs de contraintes et les solveurs SAT (Satisfiability Testing) utilisent des techniques ad hoc qui se sont avérées empiriquement efficaces en pratique.

Dans cet article, nous revisitons la randomisation pour la résolution de contraintes. Même si la randomisation est déjà connue en résolution de contraintes, par exemple dans les algorithmes incomplets, elle n'est pas autant étudiée dans les solveurs complets. La randomisation la plus utilisée dans les solveurs complets est utilisée en tant que *tie-breaker* lors d'ex-aequo pendant l'utilisation d'une heuristique de choix de variables. Un ex-aequo est un ensemble de variables qui reçoivent le même classement par l'heuristique. Il est habituel de sélectionner une de ces variables aléatoirement [5]. Notre approche étudie plus en profondeur l'application de la randomisation. Notre approche provoque une perturbation du processus de sélection des variables en introduisant des choix aléatoires. Ces perturbations ne sont pas arbitraires mais visent à maintenir un bon équilibre entre exploitation et exploration. Nous présentons plusieurs stratégies de perturbation, conceptuellement simples, pour incorporer des choix aléatoires dans la résolution de contraintes avec redémarrages.

[*] Article publié à CP2020

[†] Papier doctorant : Hugues Wattez est auteur principal.

La quantité de perturbation est contrôlée et apprise par des bandits lors de redémarrages successifs sous diverses politiques d'exploration stationnaire et non stationnaire. Notre évaluation expérimentale a montré des améliorations significatives des performances des heuristiques perturbées par rapport à leur homologue classique, établissant ainsi la nécessité d'incorporer la perturbation dans les solveurs de contraintes modernes.

2 Contribution

Nous avons introduit des stratégies de perturbation conceptuellement simples pour incorporer des choix aléatoires dans la résolution de contraintes avec redémarrages. La plupart de nos stratégies sont adaptatives, c'est-à-dire que la quantité de perturbation est apprise lors des redémarrages successifs. Nous exploitons le mécanisme de redémarrage qui existe dans tous les solveurs modernes pour contrôler l'application de choix aléatoires. Nous avons déployé une technique d'apprentissage par renforcement qui détermine à chaque *run* (c'est-à-dire au début du redémarrage), si elle appliquera l'heuristique standard, intégrée dans le solveur, ou une procédure qui fait des choix de branchement aléatoires. Il s'agit d'un problème de décision séquentielle et en tant que tel, il peut être modélisé comme un problème de bandit multi-armé (MAB) [2], plus précisément, comme un problème à double bras.

Dans l'apprentissage par renforcement, la proportion entre l'exploration et l'exploitation est commandée par diverses politiques. Nous en avons essayé plusieurs, telles que *Epsilon-greedy* [7], *EXP3* [3], *Thompson Sampling* [8], *UCB1* [3] et *MOSS* [1]. L'apprentissage provient du retour d'expérience après chaque exécution. Il reflète l'efficacité de l'exécution pour un choix donné, appelée fonction de récompense. Nous proposons également une stratégie statique qui perturbe une heuristique donnée avec une probabilité fixe trouvée empiriquement. Nous avons évalué les stratégies statiques et adaptatives pour plusieurs heuristiques bien connues, montrant des améliorations significatives des performances en faveur du solveur perturbé, indépendamment de l'heuristique utilisée. Nous avons également mené des expériences permettant l'utilisation de *nogoods*, un composant intégrant les solveurs modernes, montrant que les perturbations dominent toujours le solveur standard, car les *nogoods* obtenus lors des exécutions aléatoires ne désorientent pas la recherche.

De nombreuses observations ont été tirées de cette étude :

- Nos techniques de perturbation améliorent constamment les performances du solveur indépendamment de l'heuristique utilisée comme base dans le solveur. La stratégie perturbée surpasse toujours son homologue de base.
- Un solveur perturbé peut compenser un mauvais choix d'heuristique de l'utilisateur, car il améliore automatiquement ses performances en visitant des parties inconnues de l'espace de recherche. Cela est dû aux exécutions aléatoires, au cours desquelles l'heuristique acquiert des connaissances supplémentaires que l'heuristique seule ne peut pas obtenir.
- Les stratégies adaptatives sont toujours plus efficaces que les stratégies statiques car, grâce à leur fonctions de récompense, elles peuvent ajuster leur comportement à l'heuristique et à l'instance à résoudre.
- La présence de *nogoods* n'affecte pas l'efficacité du solveur perturbé. Les *nogoods* produits restent utiles.
- Les résultats expérimentaux ont montré l'avantage des perturbations pour améliorer les performances des solveurs de contraintes, quel que soit leur réglage par défaut.

Références

[1] J-Y. Audibert et S. Bubeck : Minimax policies for adversarial and stochastic bandits. *In COLT*, pages 217–226, Montreal, Canada, 2009.

[2] P. Auer, N. Cesa-Bianchi et P. Fischer : Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2):235–256, May 2002.

[3] P. Auer, N. Cesa-Bianchi, Y. Freund et R. Schapire : The nonstochastic multiarmed bandit problem. *SIAM Journal on Computing*, 32(1):48–77, 2002.

[4] 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.

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

[6] T. Hogg, B. A. Huberman et C. P. Williams : Phase transitions and the search problem. *Artificial Intelligence*, 81(1):1 – 15, 1996.

[7] R.S. Sutton et A.G. Barto : Reinforcement learning : An introduction. 9:1054, 02 1998.

[8] W. R. Thompson : On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3-4):285–294, 12 1933.

Perturbing Branching Heuristics in Constraint Solving

Anastasia Paparrizou and Hugues Wattez

CRIL, University of Artois & CNRS, France
{paparrizou,wattez}@cril.fr

Abstract. Variable ordering heuristics are one of the key settings for an efficient constraint solver. During the last two decades, a considerable effort has been spent for designing dynamic heuristics that iteratively change the order of variables as search progresses. At the same time, restart and randomization methods have been devised for alleviating heavy-tailed phenomena that typically arise in backtrack search. Despite restart methods are now well-understood, choosing *how* and *when* to randomize a given heuristic remains an open issue in the design of modern solvers. In this paper, we present several conceptually simple perturbation strategies for incorporating random choices in constraint solving with restarts. The amount of perturbation is controlled and learned in a bandit-driven framework under various stationary and non-stationary exploration policies, during successive restarts. Our experimental evaluation shows significant performance improvements for the perturbed heuristics compared to their classic counterpart, establishing the need for incorporating perturbation in modern constraint solvers.

1 Introduction

For decades now, researchers in Constraint Programming (CP) have put a tremendous effort in designing constraint solvers and advancing their internal components. Many mechanisms have been combined, leading to a technology that is now widely used to solve combinatorial problems. A constraint solver is typically composed of a backtracking search algorithm, a branching heuristic for guiding search, a filtering procedure for pruning the search space, and no-good recording.

Since the very beginning, the order in which variables are selected (assigned) by the branching heuristic holds a central place. It is referred to as the variable ordering heuristic. Choosing the appropriate variable ordering heuristic for solving a given constraint satisfaction problem is quite important since the solving time may vary by orders of magnitude from one heuristic to the other. Recent heuristics are more stable [15,36].

Backtrack search is also vulnerable to unstable behavior because of its inherent combinatorial nature. In the early '00s, the exponential time differences have been investigated under the phase-transition phenomena [19] and the heavy-tailed distributions of solving time [12]. We can decrease such undesired differences by introducing restart policies, randomization [6] and no-goods recording during search. While restarts and no-goods are well established in CP solvers [11,18,24,27], randomization remains limited to ad-hoc techniques that have been found to work well in practice.

In this work, we use randomization to perturb the variable selection process. These perturbations are designed to keep a good and controlled balance between exploitation and exploration. We introduce conceptually simple *perturbation strategies* for incorporating random choices in constraint solving with restarts. Most of strategies that we present are adaptive, meaning that the amount of perturbation is learned during successive restarts. We exploit the restart mechanism that exists in all modern solvers to control the application of random choices. We deploy a reinforcement learning technique that determines at each run (i.e., at the beginning of the restart), if it will apply the standard heuristic, embedded in the constraint solver, or a procedure that makes random branching choices. This is a sequential decision problem and as such, it can be modeled as a multi-armed bandit problem (MAB) [3], precisely, as a double-armed. In reinforcement learning, the proportion between exploration and exploitation is specified by various policies. We tried several of them, such as `Epsilon-greedy` [31], `EXP3` [4], `Thompson Sampling` [32], the upper confidence bound `UCB1` [4] and `MOSS` [2]. The learning comes from the feedback taken after each run that reflects the efficiency of the run under a given choice, referred to as a reward function. We also propose a static strategy that perturbs a given heuristic with a fixed probability, found empirically. We evaluate the static and adaptive strategies for several well known heuristics, showing significant performance improvements in favor of the perturbed solver independently of the underlying heuristic used. A perturbed strategy always outperforms its baseline counterpart both in time and number of solved instances. We have also run experiments allowing the use of no-goods, an integral component of solvers nowadays, showing that perturbations still dominate the standard setting, as the no-goods obtained during random runs do not disorientate search.

Many useful observations are derived from this study. The more inefficient a heuristic is, the more effective the perturbation. A perturbed solver can compensate for a potentially bad heuristic choice done by the user, as it permits to automatically improve its performance by visiting unknown parts of the search space. This is due to the random runs, during which the heuristic acquires extra knowledge, other than what obtained when running alone. We show that adaptive strategies always outperform the static ones, as they can adjust their behavior to the instance to be solved and to the heuristic setting. Overall, the results show the benefits of establishing perturbation in CP solvers for improving their overall performance whatever their default setting is.

2 Related Work

Introducing a touch of randomization for better diversifying the search of local and complete procedures has been shown to be quite effective for both SAT (Satisfiability Testing) and CP (Constraint Programming). A stochastic local search requires the right setting of the "noise" parameter so as to optimize its performance. This parameter determines the likelihood of escaping from local minima by making non-optimal moves. In GSAT [30], it is referred as the random walk parameter and in walkSAT [29], simply as the "noise". Large Neighborhood Search uses randomization to perform jumps in the search space while freezing a fragment of the best solution obtained so far [26].

In complete CP solvers, the first evidence that diversification can boost search dates back to Harvey and Ginsberg research [17]. Harvey and Ginsberg proposed a deterministic backtracking search algorithm that differs from the heuristic path by a small number of decision points, or "discrepancies". Then, Gomes et al. [13,12] showed that a controlled form of randomization eliminates the phenomenon of "heavy-tailed cost distribution" (i.e., a non-negligible probability that a problem instance requires exponentially more time to be solved than any previously solved instances). Randomization was applied as a tie-breaking step: if several choices are ranked equally, choose among them at random. However, if the heuristic function is powerful, it rarely assigns more than one choice the highest score. Hence, the authors introduced a "heuristic equivalence" parameter in order to expand the choice set for random tie-breaking.

More recently, Grimes and Wallace [14] proposed a way to improve the classical `dom/wdeg` heuristic (based on constraint weighting) by using random probing, namely a pre-processing sampling procedure. The main idea is to generate the weights of the variables with numerous but very short runs (i.e., restarts) prior search, in order to make better branching decisions at the beginning of the search. Once the weights are initialized, a complete search is performed during which weights either remain frozen or continue updating.

3 Preliminaries

A *Constraint Network* P consists in a finite set of variables $\texttt{vars}(P)$, and a finite set of constraints $\texttt{ctrs}(P)$. We use n to denote the number of variables. Each variable x takes values from a finite domain, denoted by $\texttt{dom}(x)$. Each constraint c represents a mathematical relation over a set of variables, called the *scope* of c. A *solution* to P is the assignment of a value to each variable in $\texttt{vars}(P)$ such that all constraints in $\texttt{ctrs}(P)$ are satisfied. A constraint network is *satisfiable* if it admits at least one solution, and the corresponding *Constraint Satisfaction Problem (CSP)* is to determine whether a given constraint network is satisfiable, or not. A classical procedure for solving this NP-complete problem is to perform a backtrack search on the space of partial solutions, and to enforce a property called *generalized arc consistency* [23] on each decision node, called *Maintaining Arc Consistency* (MAC) [28]. The MAC procedure selects the next variable to assign according to a *variable ordering heuristic*, denoted H. Then, the selected variable is assigned to a value according to its value ordering heuristic, which is usually the lexicographic order over $\texttt{dom}(x)$.

As mentioned in Section 2, backtrack search algorithms that rely on deterministic variable ordering heuristics have been shown to exhibit heavy-tailed behavior on both random and real-world CSP instances [12]. This issue can be alleviated using *randomization* and *restart* strategies, which incorporate some random choices in the search process, and iteratively restart the computation from the beginning, with a different variable ordering [6]. Since our randomization method will be discussed in Section 4, we focus here on restart strategies.

Conceptually, a restart strategy is a mapping $\texttt{res} : \mathbb{N} \rightarrow \mathbb{N}$, where $\texttt{res}(t)$ is the maximal number of "steps" which can be performed by the backtracking search algorithm at run, or *trial*, t. A constraint solver, equipped with the MAC procedure and a restart strategy res, builds a sequence of search trees $\langle \mathcal{T}^{(1)}, \mathcal{T}^{(2)}, \ldots \rangle$, where $\mathcal{T}^{(t)}$ is the search tree explored by MAC at run t. After each run, the solver can memorize some relevant information about the sequence $\langle \mathcal{T}^{(1)}, \mathcal{T}^{(2)}, \ldots, \mathcal{T}^{(t-1)} \rangle$, like the number of constraint checks in the previous runs, the no-goods that have appeared frequently in the search trees explored so far [20]. The *cutoff*, $\texttt{res}(t)$, which is the number of allowed steps, may be defined by the number of nodes, the number of wrong decisions [8], the number of seconds, or any other relevant measure. In a *fixed* cutoff restart strategy, the number T of trials is fixed in advance, and $\texttt{res}(t)$ is constant for each trial t, excepted for the Tth trial which allows an unlimited number of steps (in order to maintain a complete algorithm). This strategy is known to be effective in practice [13], but a good cutoff value $\texttt{res}(t)$ has to be found by trial and error. Alternatively, in a *dynamic* cutoff restart strategy, the number T of trials is unknown, but res increases geometrically, which guarantees that the whole space of partial solutions is explored after $O(n)$ runs [33]. A commonly used cutoff strategy is driven by the Luby sequence [22].

4 Perturbation strategies

As indicated in Section 3, the process of constraint solving with a restart policy may be viewed as a sequence $\langle 1, 2, \cdots, T \rangle$ of *runs*. For the aforementioned restart functions, the sequence of runs is finite, but the horizon T is not necessarily known in advance. During each run t, the solver calls the MAC algorithm for building a search tree $\mathcal{T}_t$, whose size is determined by the cutoff of the restart policy. If the solver has only access to a single variable ordering heuristic, say H, it will run MAC with H after each restart. Yet, if the solver is also allowed to *randomize* its variable orderings, it is faced with a fundamental choice at each run t: either call MAC with the heuristic H in order to "exploit" previous computations made with this heuristic, or call MAC with a random variable ordering U so as to "explore" new search trees, and potentially better variable orderings. Here, U is any variable ordering drawn at random according to a uniform distribution over the permutation group of $\texttt{vars}(P)$. We need to highlight here, that the intermediate random runs of U perturb the involved classic heuristic H by updating its parameters, which ultimately affects the behavior/performance of H. In other words, the subsequent heuristic runs, will not produce the same orderings as in the traditional solving process, allowing thus the solver to (potentially) tackle instances that neither H nor U would solve stand-alone.

Algorithm 1: Bandit-Driven Perturbations

```
Input: constraint network $P$, heuristic $H$, policy $B$
initArms$_B(H, U)$          // Initialize the arms and the bandit policy
for each run $t = 1, \cdots, T$ do
    $a_t \leftarrow$ selectArm$_B()$       // Select an arm according to the bandit policy
    $r_t(a_t) \leftarrow \mathrm{MAC}(P, a_t)$     // Execute the solver and compute the reward
    updateArms$_B\big(r_t(a_t)\big)$   // Update the bandit policy
```

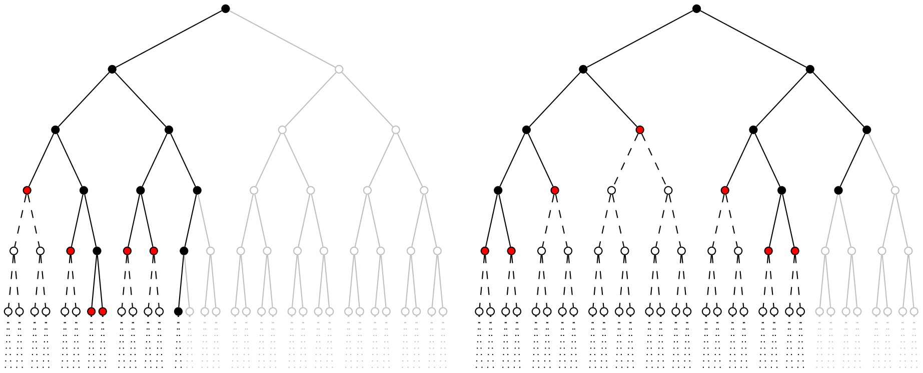
The task of incorporating perturbations into constraint solving with restarts can be viewed as a *double-armed* bandit problem: during each run t, we have to decide whether the MAC algorithm should be called using H (exploitation arm) or U (exploration arm). Once MAC has built a search tree $\mathcal{T}_t$, the performance of the chosen arm can be assessed using a reward function defined according to $\mathcal{T}_t$. The overall goal is to find a policy mapping each run t to a probability distribution over $\{H, U\}$ so as to maximize cumulative rewards.

Multi-armed bandit algorithms have recently been exploited in CP in different contexts, i.e. for guiding search [21], for learning the right level of propagation [5] or the right variable ordering heuristic [34,35,37]. In the framework of Xia and Yap [37], a single search tree is explored (i.e., no restarts), and the bandit algorithm is called at each node of the tree to decide which heuristic to select. The trial is associated with explored subtrees, while in our approach, trials are mapped to runs using a restart mechanism. Our framework makes use of restarts in the same way as the ones of [34,35], as it was shown in [35] that such a framework offers greater improvements compared to the one of [37]. In our case, we utilise a double-armed framework in order to construct our bandit-driven perturbation given by Algorithm 1. The algorithm, takes as input a constraint network P, a variable ordering heuristic H, and a bandit policy B. As indicated above, the bandit policy has access to two arms, H and U, where U is the random variable ordering generated on the fly, during execution. The three main procedures used by the bandit policy are initArms$_B$ for initializing the distribution over $\{H, U\}$ according to policy B, selectArm$_B$ for choosing the arm $a_t \in \{H, U\}$ that will be used to guide the search all along the tth run, and updateArms$_B$ for updating the distribution over $\{H, U\}$ according to the observed reward $r_t(a_t)$ at the end of the tth run.

4.1 Reward function

The feedback $r_t(a_t)$ supplied at the end of each run captures the performance of the MAC algorithm, when it is called using $a_t \in \{H, U\}$. To this end, the reward function r_t maps the search tree $\mathcal{T}_t$ built by $\mathrm{MAC}(a_t)$ to a numeric value in $[0, 1]$ that reflects the quality of backtracking search when guided by a_t.

As a reward function, we introduce the measure of the *explored sub-tree* denoted as `esb`. `esb` is given by the number of visited nodes during a run, divided



(a) $\texttt{esb}(\mathcal{T}_{\text{left}}) = \frac{\log(16)}{\log(2^6)} \approx 0.67$ (b) $\texttt{esb}(\mathcal{T}_{\text{right}}) = \frac{\log(16)}{\log(2^5)} \approx 0.80$

Fig. 1: Comparison of two runs with a restart cutoff fixed to 16 nodes.

by the size of the complete sub-tree defined over the variables selected during this run. The later is simply the size of the Cartesian product of the domains of the variables selected during the run. As selected variables, we consider those ones that have been chosen at least once by the arm in question. `esb` represents the search space covered by the solver, under a certain setting of a run, compared to the total possible space on the selected variables. The intuition is that an exploration that discovers failures deeply in the tree (meaning that, many variables are instantiated) will be penalized (due to the big denominator) against an exploration that discovers failures at the top branches.

In formal terms, given a search tree $\mathcal{T}$ generated by the MAC algorithm, let $\texttt{vars}(\mathcal{T})$ be the set of variables that have been selected at least once during exploration of $\mathcal{T}$ and $\texttt{nodes}(\mathcal{T})$ the number of visited nodes. Then,

$$r_t(a_t) = \texttt{esb}(\mathcal{T}_t) = \frac{\log(\texttt{nodes}(\mathcal{T}_t))}{\log\left(\prod_{x \in \texttt{vars}(\mathcal{T}_t)} |\texttt{dom}(x)|\right)}$$

A logarithmic scaling is needed to obtain a better discrimination between the arms as the numerator is usually significantly smaller than the denominator of the fraction. The reward values belong to [0, 1]. The higher the ratio/reward is, the better the performance of a_t is.

Figure 1, shows a motivating example for the reward function. It displays an example of tree explorations done by two different runs. For simplicity, domains are binary and at each level the variable to be instantiated is fixed per run (namely left and right branches are on the same variable). Empty nodes represent non-visited or pruned nodes, solid black nodes are the visited ones, solid red ones denote failures. Below red nodes, there are the pruned sub-trees in dashed style while the non-visited sub-trees are slightly transparent. For both runs we consider the same number of node visits, i.e. 16. At the left run (Figure 1a), the solver goes until level 6 (selecting 6 variables) while the solver at the right run (Figure 1b) goes until level 5. The left run will take the score of 0.67 and the right one 0.80. Our bandit will prefer the arm that produced the right tree, namely a search that goes faster at the right branches than deeper in the tree (which implies that more search is required). Early failures is a desired effect that many heuristics consider explicitly on implicitly in order to explore smaller search spaces until the solution or unsatisfiability.

4.2 Perturbation rate

As indicated in Algorithm 1, the perturbation framework is conceptually simple: based on a restart mechanism, the solver performs each run by first selecting an arm in $\{H, U\}$, next, observing a reward for that arm, and then, updating its bandit policy according to the observed reward. This simple framework, allows us to use a variety of computationally efficient bandit policies to adapt/control the amount of perturbation applied during search. From this perspective, we have opted for five well-studied "adaptive" policies for the double-armed bandit problem, and one "static" (or stationary) policy which serves as reference for our perturbation methods.

ϵ-Greedy. Arguably, this is the simplest adaptive bandit policy that interleaves exploitation and exploration using a parameter ϵ. The policy maintains the empirical means $\hat{r}$ of observed rewards for H and U. Initially, both $\hat{r}_1(H)$ and $\hat{r}_1(U)$ are set to 0. On each run t, the function $\textsc{selectArm}_{\epsilon G}$ returns with probability $(1-\epsilon)$ the arm a_t that maximizes $\hat{r}_t(a_t)$, and returns with probability ϵ any arm a_t drawn uniformly at random. Finally, based on the observed reward $r_t(a_t)$, the procedure $\textsc{updateArms}_{\epsilon G}$ updates the empirical mean of a_t according to

$$\hat{r}_{t+1}(a_t) = \frac{t}{t+1}\hat{r}_t(a_t) + \frac{1}{t+1}r_t(a_t)$$

EXP3. The EXPonentially weighted forecaster for EXPloration and EXPloitation (EXP3) is the baseline bandit policy operating in "non-stochastic" environments, for which no statistical assumption is made about the reward functions [4]. Several variants of EXP3 have been proposed in the literature, but we use here the simplest version defined in [10]. Here, the procedure $\textsc{initArms}_{\texttt{EXP3}}$ sets the initial distribution π_1 of arms to the uniform distribution $(1/2, 1/2)$. During each trial t, the procedure $\textsc{selectArm}_{\texttt{EXP3}}$ simply draws an arm a_t according to the distribution π_t. Based on the observed reward $r_t(a_t)$, the procedure $\textsc{updateArms}_{\texttt{EXP3}}$ updates the distribution π_t according to the multiplicative weight-update rule:

$$\pi_{t+1}(a) = \frac{\exp(\eta_t R_t(a))}{\exp(\eta_t R_t(H)) + \exp(\eta_t R_t(U))}$$

η_t corresponds to the learning rate (usually set to $\frac{1}{\sqrt{t}}$),

$$R_t(a) = \sum_{s=1}^{t} \frac{r_s(a)}{\pi_s(a)} \mathbb{1}_{a \sim \pi_s}$$

and $\mathbb{1}_{a \sim \pi_s}$ indicates whether a was the arm picked at trial s, or not.

UCB1. Upper Confidence Bound (UCB) policies are commonly used in "stochastic" environments, where it is assumed that the reward value $r_t(a)$ of each arm a is drawn according to a fixed, but unknown, probability distribution. `UCB1` is the simplest policy in the Upper Confidence Bound family [3]. In the setting of our framework, this algorithm maintains two 2-dimensional vectors, namely, $n_t(a)$ is the number of times the policy has selected arm a on the first t runs, and $\hat{r}_t(a)$ is the empirical mean of $r_t(a)$ during the $n_t(a)$ steps. initArms$_{\texttt{UCB1}}$ sets both vectors to zero and, at each run t, selectArm$_{\texttt{UCB1}}$ selects the arm a_t that maximizes

$$\hat{r}_t(a) + \sqrt{\frac{2\ln(t)}{n_t(a)}}$$

Finally, updateArms$_{\texttt{UCB1}}$ updates the vectors n_t and $\hat{r}_t$ according to a_t and $r_t(a_t)$, respectively.

MOSS. The Minimax Optimal Strategy in the Stochastic case (MOSS) algorithm is an instance of the UCB family. The only difference with `UCB1` lies in the confidence level which not only takes into account the number of plays of individual arms, but also the number of arms (2) and the number of runs (t). Specifically, selectArm$_{\texttt{MOSS}}$ chooses the arm a_t that maximizes

$$\hat{r}_t(a) + \sqrt{\frac{4}{n_t(a)} \ln^{+}\left(\frac{t}{2n_t(a)}\right)}$$

where $\ln^{+}(x) = \ln\max\{1, x\}$.

TS. The Thompson Sampling algorithm is another well-known policy used in stochastic environments [1,32]. In essence, the TS algorithm maintains a beta distribution for the rewards of each arm. initArms$_{\texttt{TS}}$ sets $\alpha_1(a)$ and $\beta_1(a)$ to 1 for $a \in \{H, U\}$. On each run t, selectArm$_{\texttt{TS}}$ selects the arm a_t that maximizes $\text{Beta}(\alpha_t(a), \beta_t(a))$, and updateArms$_{\texttt{TS}}$ uses $r_t(a_t)$ to update the beta distribution as follows:

$$\begin{aligned}\alpha_{t+1}(a) &= \alpha_t(a) + \mathbb{1}_{a=a_t} r_t(a_t)\\ \beta_{t+1}(a) &= \beta_t(a) + \mathbb{1}_{a=a_t}(1 - r_t(a_t))\end{aligned}$$

SP. Finally, in addition to the aforementioned adaptive bandit policies which learn a distribution on $\{H, U\}$ according to observed rewards, we shall consider the following Static Policy (SP): on each round t, selectArm$_{\texttt{SP}}$ chooses H with probability $(1-\epsilon)$, and U with probability ϵ. Although this policy shares some similarities with the ϵ-greedy algorithm, there is one important difference: the distribution over $\{H, U\}$ is fixed in advance, and hence, SP does not take into account the empirical means of observed rewards. In other words, updateArms$_{\texttt{SP}}$ is a dummy procedure that always returns $(1-\epsilon, \epsilon)$. This stationary policy will serve as reference for the adaptive policies in the experiments.

5 Experimental Evaluation

We have conducted experiments on a large dataset to demonstrate the performance of the proposed perturbations. The set includes all instances (612 in total) from the 2017's XCSP3 competition[1] coming from 60 different problem classes. The experiments have been launched on an 2.66 GHz Intel Xeon and 32 GB RAM nodes. We have used the `AbsCon`[2] solver in which we integrated our perturbation strategies and the strategies of [13] and [14]. We used 2-way branching, generalized arc consistency as the level of consistency, Luby progression based on node visits as restart policy (the constant is fixed to 100 in `AbsCon`) and the timeout set to $1,200$ seconds. We have chosen a big variety of variable ordering heuristics, including recent, efficient and state-of-the-art ones: `dom` [16], `dom/ddeg` [7], `activity` [25], `dom/wdeg` [9], `CHS` [15], $\texttt{wdeg}^{\texttt{ca.cd}}$ [36] and finally `rand` which chooses uniformly randomly a variable order. Among these, `dom` and `dom/ddeg` do not record/learn anything between two runs (`dom` and `ddeg` are re-initialized at the root), while all the others learn during each (random) run and maintain this knowledge all along the solving process, which might change/improve their behavior. We have run all these original heuristics separately for a baseline comparison. Note that in our first experiments no-goods recording are switched off in the solver to avoid biasing the results of heuristics and strategies.

Regarding our perturbation strategies, we denote by `SP` the static perturbation and by `e-greedy`, `UCB1`, `MOSS`, `TS` and `EXP3` the various adaptive perturbation (`AP`) strategies. Epsilon of `SP` and `e-greedy` are fixed to 0.1. This value has been fixed offline after a linear search of the best value. Apart from comparing to the default solver settings (i.e., `original` heuristics), we compare to three other perturbation strategies from the bibliography. The one is the `sampling` algorithm of [14] that corresponds to the sampling pre-processing step which is fixed to 40 restarts with a cutoff of n nodes corresponding to the number of variables of each instance. When the probing phase finishes, we continue updating the variable scoring as it produces better results . The second is the `equiv-30` that corresponds to the criterion of equivalence of [13]. This equivalence parameter is set to 30% as authors proposed. Last, we compare to the standard tie-breaking denoted `equiv-0`, where a random choice is done among the top ranked variables scored equally by the underlying heuristic. Note that there are no ties, `equiv-0` has no effect on the heuristic, as opposed to `equiv-30`.

Table 1 displays the results of the aforementioned settings and strategies on the XCSP'17 competition dataset. The comparison is given on the number of solved instances (*#inst*), within $1,200$ seconds, the cumulative CPU time (*time*) computed from instances solved by at least one method and the percentage of perturbation (*%perturbation*) which is the mean perturbation of the solved instances, computed by the number of runs with the arm U divided by the total of runs. Each time a setting has not solved an instance that another setting solved, it is penalized by the timeout time. Numbers in bold indicate that a strategy

[1] See http://www.cril.univ-artois.fr/XCSP17

[2] See http://www.cril.fr/~lecoutre/#/softwares

Table 1: Comparison of `original`, `sampling`, `equiv-30` and the proposed perturbed strategies for the XCSP'17 dataset.

							AP				
		`original`	`sampling`	`equiv-0`	`equiv-30`	`SP`	`e-greedy`	`UCB1`	`MOSS`	`TS`	`EXP3`
	#inst	287	**321**	**312**	**308**	**315**	**314**	**323**	**322**	**318**	**323**
`dom`	time (359)	101,589	**58,496**	**74,064**	**75,992**	**72,460**	**75,474**	**59,527**	**61,171**	**63,856**	**61,842**
	%perturb.	0	-	-	-	10	7.7	32.8	21.0	24.9	44.1
	#inst	307	**321**	**319**	**324**	**343**	**337**	**345**	**346**	**342**	**343**
`dom/ddeg`	time (365)	85,131	**61,071**	**67,537**	**66,005**	**46,179**	**51,371**	**41,404**	**43,280**	**44,981**	**40,260**
	%perturb.	0	-	-	-	10	8.1	34.3	21.4	26.5	45.3
	#inst	342	311	334	329	**356**	**356**	**352**	**353**	**350**	**351**
`activity`	time (372)	52,989	82,041	60,304	64,619	**36,688**	**36,125**	**39,552**	**37,463**	**40,306**	**39,371**
	%perturb.	0	-	-	-	10	7.6	33.5	20.7	26.0	44.4
	#inst	347	342	**349**	**349**	**358**	**346**	**358**	**354**	**356**	**363**
`dom/wdeg`	time (381)	55,599	56,038	**54,276**	**53,263**	**45,615**	58,888	**43,726**	**49,052**	**46,896**	**39,277**
	%perturb.	0	-	-	-	10	11.6	34.3	23.9	28.6	45.2
	#inst	366	354	**368**	361	**370**	**368**	**371**	**372**	**367**	**369**
`wdeg`$^{ca.cd}$	time (389)	42,565	52,344	43,944	49,717	**38,966**	**39,292**	**41,745**	**40,433**	44,941	42,661
	%perturb.	0	-	-	-	10	7.8	32.1	19.9	24.8	42.9
	#inst	370	343	**371**	361	**371**	**372**	367	**373**	367	367
`CHS`	time (389)	41,462	64,779	42,025	56,639	**37,699**	**38,158**	43,869	**38,190**	46,597	42,444
	%perturb.	0	-	-	-	10	7.3	32.6	19.9	25.7	44.5
	#inst	291									
`rand`	time (291)	12,921									
	%perturb.	100									

outperformed her corresponding default setting of the solver (i.e., `original`). Underlined numbers show the winning strategy. `dom` and `dom/ddeg`, the unaffected heuristics, appear at the top of the table as they cannot be perturbed by randomized runs. After each run their parameters are reinitialized and not accumulated as for the rest of the heuristics (perturbed ones). Hence, for `dom` and `dom/ddeg`, any additional instance that perturbation strategies are able to solve comes from an intervening run of U.

The existing perturbation techniques, `sampling`, `equiv-30` and `equiv-0`, solve more instances than their respective baseline heuristic for the case of `dom` and `dom/ddeg`. However, this never happens for `sampling` and `equiv-30` on the more sophisticated heuristics (except from `dom/wdeg`), where we see that the perturbation they apply disorientates totally the search, solving constantly less instances than the original heuristics (e.g., `sampling` missed 31 instances for `activity`). `equiv-0` just marginally outperforms pure heuristics by one or two instances while it is far inferior to `AP`s (e.g., it missed 22 instances compared to `e-greedy` on `activity`). `equiv-30` is superior to `sampling` on the perturbed heuristics but still far inferior to all proposed strategies. Among the proposed perturbation strategies, we observe that both `SP` and `AP` strategies constantly outperform the default setting of the solver, while for many heuristics they have close performance (e.g., `activity`). `MOSS` is the best strategy in terms of solved instances and time results, except for `activity` and `dom/wdeg` heuristics, where

SP (e-greedy too) and EXP3 dominate respectively. UCB1 and MOSS are the best strategies for dom and dom/ddeg, with only one instance of difference, showing that a perturbation rate between 20% and 30% is the best choice. On the other side, TS with a similar rate seems not to select that well the right arm for all runs. Similarly for equiv-30, which also applies a randomization of 30% on the top ranked variables, it seems that the way it is applied (i.e., at every decision) is not that efficient. SP and e-greedy, despite winning their original counterparts are less competitive due to their low perturbation rate. EXP3 is also a good candidate policy for dom and dom/ddeg.

rand solved the less instances in total, i.e. 291, in $12,921$ seconds, which means that many of them correspond to quite easy instances. As rand is the "bad" arm, a small participation of 10% in SP is just enough to be beneficial for the heuristics that are by themselves efficient (e.g., $\texttt{wdeg}^{\texttt{ca.cd}}$, CHS), but as SP is not adaptive, it is rarely better than the AP strategies; it cannot adjust its behavior to heuristics that require more perturbation to improve (e.g., dom). AP strategies, being adaptive, allow usually much more exploration of the U arm that makes them win several instances. An exception is the CHS heuristic, where a perturbation over 20% might be harmful (i.e., in UCB1, TS and EXP3 policies). Each bandit policy follows a general trend that can vary between heuristics (e.g., UCB1 is around 30%, MOSS around 20% and e-greedy with EPX3 represent the two extremities). EXP3 sets the initial distribution π_1 of arms to $(1/2, 1/2)$ such that both arms have equal chances at the beginning. As it is a non-stochastic bandit it needs more exploration than stochastic ones need to converge. Although one would expect that this could deteriorate the solver, it seems that the bandit utilizes the right arm at each run since it is efficient both in time and solved instances. The high perturbation rates come from the mean, that smoothens the high variance between instances. Also, many of the instances are solved fast and EXP3 favorizes a lot U, being the best arm at the early (short) runs, while the long runs at the end are done by H, which explains its good overall performance. TS despite it is better than original heuristics and existing perturbation methods, it is usually inferior to SP by small differences.

We distinguish the MOSS policy, which apart from being the best policy for many heuristics, it never deteriorates the solver for any heuristic (as happens for some strategies on CHS). It applies the exploration when and where needed (more at the early runs) and converges faster to the best heuristic. Regarding the time performance, all perturbation strategies are faster than their baseline heuristic, even for the most efficient heuristics (i.e., CHS and $\texttt{wdeg}^{\texttt{ca.cd}}$).

Figure 2, visualizes in a cactus plot the performance of the best AP strategy, namely MOSS, for all heuristics compared to their corresponding original heuristic. On x-axis we see the instances solved as time progresses. y-axis displays the allowed time given to the solver. Dashed lines display the performance of original heuristics and solid lines the perturbed solver. The closer a line is to the right bottom corner the more instances has solved in less time. In general, MOSS policy perturbations appear always at the right side of their respective default heuristic. We observe that for the less efficient heuristics, as dom and dom/ddeg, the performance gap between the original and the respective perturbed version is big even for easy instance and increases significantly as time passes, corresponding to more difficult instances. It worths noticing that, dom/ddeg after being perturbed, becomes better than activity (solved 4 more instances), that originally was much more efficient than dom/ddeg. dom$_{\texttt{MOSS}}$ solved in total 35 (resp. 39) more instances than dom (resp. dom/ddeg). For activity and dom/wdeg, the curves are closer, though the gap is still significant especially for harder instances. Even for the most efficient heuristics proposed the last two years, namely CHS and wdeg$^{\texttt{ca.cd}}$, MOSS is almost all the time the best setting for the solver.

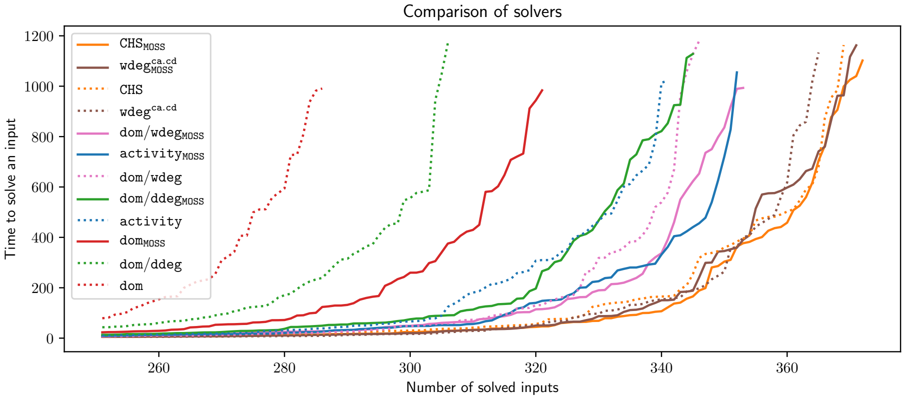


Fig. 2: Comparison of original and MOSS strategies while increasing the allowed time.

In the following, we do not present results for all adaptive strategies, but only for the most stable and efficient ones. We omit presenting results for EXP3 because it failed for CHS, which is one of the two best heuristics in CP, and for TS, because it is never the winning strategy.

Table 2 gives complementary information derived from Table 1 for the best proposed strategies. We have calculated the number of instances solved exclusively by rand (i.e., instances that the heuristic alone could not solve), denoted by *#random*, and the number of instances, *#perturb*, that the solver solved due to the perturbation rand caused to the corresponding heuristic. Instances in *#perturb* are those that are solved neither by the original heuristic nor the rand heuristic, making thus the perturbed solver outperform even its corresponding virtual best solver. As expected, dom and dom/ddeg do not win instances due to perturbation, but only due to a good ordering during a random run (more than 30 instances for each policy). For other heuristics, we see a more balanced distribution of instances in *#random* and *#perturb*. We observe that the most robust heuristics (CHS and wdeg$^{\texttt{ca.cd}}$) solve extra instances mainly by perturbation (for

Table 2: Won instances by perturbation or random runs for `SP`, `e-greedy`, `UCB1` and `MOSS` with nodes as cutoff for the XCSP'17 dataset

		`dom`	`dom/ddeg`	`activity`	`dom/wdeg`	`wdeg`$^{ca.cd}$	`CHS`
`SP`	#perturb.	0	0	8	8	5	4
	#random	28	36	12	10	3	3
`e-greedy`	#perturb.	0	0	8	5	5	6
	#random	29	31	10	6	3	2
`UCB1`	#perturb.	0	0	5	7	8	7
	#random	38	41	13	10	3	2
`MOSS`	#perturb.	0	0	6	7	7	10
	#random	37	41	12	8	3	3

`MOSS`: 10 and 7 respectively) rather than by calling `rand` (3 and 3). Indeed, both are so efficient that the majority of solved instances by `rand` are also solved by them, which explains why their perturbations 'gain' fewer instances than the other heuristics do. In contrast, `activity` and `dom/wdeg`, that are less efficient than `CHS` and `wdeg`$^{ca.cd}$, gain more instances in total, most of which are gained by `rand`.

Table 3: Comparison of `original`, `SP`, `e-greedy` and `MOSS` strategies on a subset of families from the XCSP'17 dataset.

		`original`	`SP`	`e-greedy`	`MOSS`
CoveringArray	`dom/ddeg`	2 (4,803s, 0%)	**3** (**4,533**s, **10%**)	2 (4,803s, 3%)	**4** (**2,952**s, **17%**)
	`dom/wdeg`	4 (2,669s, 0%)	4 (3,450s, 10%)	3 (3,605s, 31%)	**5** (**2,156**s, **34%**)
	`CHS`	4 (2,483s, 0%)	**4** (**2,424**s, **10%**)	**5** (**1,705**s, **3%**)	**6** (**1,491**s, **17%**)
SuperSolutions	`dom`	2 (8,404s, 0%)	**4** (**6,972**s, **10%**)	**6** (**7,471**s, **6%**)	**6** (**3,912**s, **24%**)
	`wdeg`$^{ca.cd}$	5 (4,968s, 0%)	**7** (**3,101**s, **10%**)	**6** (**4,467**s, **9%**)	**7** (**3,821**s, **23%**)
	`CHS`	7 (3,507s, 0%)	6 (3,788s, 10%)	6 (3,812s, 6%)	**8** (**1,689**s, **20%**)
KnightTour	`dom/ddeg`	5 (7,220s, 0%)	**9** (**3,276**s, **10%**)	**9** (**2,623**s, **34%**)	**9** (**3,092**s, **37%**)
	`activity`	7 (4,891s, 0%)	**9** (**3,530**s, **10%**)	**9** (**2,823**s, **26%**)	**9** (**2,513**s, **30%**)
	`dom/wdeg`	5 (7,218s, 0%)	**9** (**4,302**s, **10%**)	**7** (**5,509**s, **25%**)	**9** (**3,286**s, **39%**)
Blackhole	`dom`	6 (2,609s, 0%)	**6** (**2,489**s, **10%**)	**6** (**2,503**s, **4%**)	**6** (**2,423**s, **16%**)
	`dom/ddeg`	8 (327s, 0%)	**8** (**44**s, **10%**)	**8** (**55**s, **4%**)	**8** (**25**s, **20%**)
	`activity`	8 (897s, 0%)	7 (1,618s, 10%)	8 (908s, 6%)	**8** (**816**s, **17%**)
LatinSquare	`dom`	8 (5,841s, 0%)	**8** (**5,840**s, **10%**)	8 (5,928s, 2%)	**8** (**5,813**s, **13%**)
	`dom/wdeg`	10 (2,481s, 0%)	**12** (**1,143**s, **10%**)	10 (3,262s, 44%)	10 (3,351s, 40%)
	`CHS`	11 (3,031s, 0%)	9 (3,947s, 10%)	9 (4,469s, 3%)	8 (5,866s, 13%)

As the results in Table 1 are quite condensed, in Table 3 we show how strategies operate and adjust for certain problem classes. For each class and each heuristic, we present the number of solved instances, the total time and the perturbation rate. For *CoveringArray*, `MOSS` is the best strategy to apply the ap-

Table 4: Comparison of `original`, `sampling`, `equiv-30`, `SP`, `e-greedy` and `MOSS` with nodes as cutoff and no-goods activated.

		original	sampling	equiv-0	equiv-30	SP		AP	
							e-greedy	UCB1	MOSS
	#inst	309	**321**	306	**316**	**338**	**334**	**342**	**340**
dom	time (359)	73,367	**57,279**	74,364	**68,446**	**43,327**	**51,168**	**42,219**	**44,027**
	%perturb.	0	-	-	-	10	8.6	33.9	21.4
	#inst	316	**322**	**320**	**321**	**347**	**346**	**352**	**347**
dom/ddeg	time (367)	71,191	**62,375**	**68,703**	**72,550**	**40,042**	**41,530**	**30,770**	**38,995**
	%perturb.	0	-	-	-	10	8.8	34.9	22.5
	#inst	352	319	349	346	**356**	**357**	**356**	**355**
activity	time (373)	40,601	74,680	45,590	46,245	**32,328**	**33,161**	**34,228**	**35,854**
	%perturb.	0	-	-	-	10	7.8	34.1	20.7
	#inst	352	340	349	344	**359**	**356**	**364**	**364**
dom/wdeg c.	time (377)	41,882	53,238	48,073	53,841	**34,253**	**40,100**	**29,559**	**30,995**
	%perturb.	0	-	-	-	10	11.9	34.7	23.9
	#inst	373	356	373	361	373	**375**	371	**377**
wdeg$^{ca.cd}$	time (388)	33,887	50,009	33,053	53,587	**33,412**	**32,359**	33,772	**28,614**
	%perturb.	0	-	-	-	10	7.8	32.8	19.8
	#inst	375	348	372	366	373	373	**375**	**376**
CHS	time (387)	30,990	58,121	36,829	47,159	**30,658**	31,547	31,020	**30,585**
	%perturb.	0	-	-	-	10	7.1	33.0	19.8
	#inst	293							
rand	time (293)	16,992							
	%perturb.	100							

propriate perturbation rate, independently of the heuristic chosen, compared to `SP` and `e-greedy` whose rate is too low. For *SuperSolutions* with `dom` as heuristic method, we see that `e-greedy` and `MOSS` are both winners despite their totally different rates. Though, it is notable that `MOSS` is twice faster than `e-greedy` on the same instances. `SP` with a rate close to `e-greedy` wins the half instances compared to it (and `MOSS`). Such observations are clear evidences, that not only the amount of randomization counts but also the when it appears. Policies as `MOSS` learn to discriminate on which run to apply H or U. Recall that, compared to other policies, `MOSS` considers in selectArm$_{\texttt{MOSS}}$ more parameters, as the number or arms and the number of runs. In *KnightTour*, all strategies are efficient, but `AP`s are always better than `SP` in terms of time. For *Blackhole* and *LatinSquare*, perturbation is not fruitful and thus, both `AP`s converge to very low rates even when instances are easy (just few seconds per instance). Notice that, in general `dom/wdeg` is helped a lot by perturbation, as in all classes rates are higher compared to other heuristics (double percentages). Also, the percentage of perturbation varies a lot depending on the heuristic and the problem class, which is the reason of the success of `AP`s.

As modern solvers exploit no-goods to improve their overall performance, we repeated our experiments by activating no-goods in order to examine the robustness of the proposed strategies and the interaction between no-goods and perturbation. Table 4 displays the results `sampling`, `equiv-0`, `equiv-30`, `SP` and the best AP strategies, namely e-greedy, UCB1 and MOSS. As seen in Table 1, sampling and equiv-30 make some improvements on the less efficient heuristics as dom and dom/ddeg, but are still inferior to SP and MOSS, while they are inefficient on all other heuristics. Surprisingly, equiv-0, despite being still more efficient than sampling and equiv-30, seems to interact badly with the presence of no-goods, as it can no longer improve the solver for any heuristic (just marginally dom/ddeg). SP, despite being static, it remains efficient apart from the case of CHS. Regarding the AP strategies, e-greedy (resp. UCB1) wins almost always the underlying heuristic except from the case of CHS (resp. $\texttt{wdeg}^{\texttt{ca.cd}}$). MOSS is again the most stable strategy, being able to improve all heuristics it perturbed. Note that the presence of no-goods has improved the performance of both the default and the perturbed solver. Therefore, there are slightly smaller differences between them compared to Table 1. The proposed perturbation strategies are robust to this fundamental parameter for solvers compared to existing strategies and adapt their behavior, especially MOSS.

6 Conclusion

We presented several strategies that significantly improve the performance and robustness of the solver by perturbing the default branching heuristic. It is the first time an approach tries to learn how and when to apply randomization in an on-line and parameter-free fashion. Controlled random search runs help variable ordering heuristics to acquire extra knowledge from parts of the search space that they were not dedicated to explore. We summarize the benefits of our approach which are manifold:

- Our perturbation techniques constantly improve the performance of the solver independently of the heuristic used as baseline in the solver. A perturbed strategy always outperforms its baseline counterpart both in time and solved instances.
- The presence of no-goods does not impact the efficacy of the perturbed solver. The produced no-goods are still fruitful.
- Perturbed heuristics can compensate for a wrong heuristic choice done by the user. Thanks to perturbation, the performance of the solver with a bad initial heuristic can reach or even outperform the performance of the solver with a better baseline heuristic. This is a step towards autonomous and adaptive solving, where the solver learns and adjusts its behavior to the instance being solved.
- Our approach is generic and easy to embed in any solver that exploits restarts.

Acknowledgments. The authors would like to thank Frederic Koriche for his valuable advices on the Machine Learning aspects of the paper as well as the anonymous reviewers for their constructive remarks. This work has been partially supported by the project *Emergence* 2020 BAUTOM of INS2I and the project CPER Data from the region “Hauts-de-France”.

References

1. Agrawal, S., Goyal, N.: Near-optimal regret bounds for Thompson Sampling. J. ACM **64**(5), 30:1–30:24 (2017)
2. Audibert, J.Y., Bubeck, S.: Minimax policies for adversarial and stochastic bandits. In: COLT. pp. 217–226. Montreal, Canada (2009)
3. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine Learning **47**(2), 235–256 (May 2002)
4. Auer, P., Cesa-Bianchi, N., Freund, Y., Schapire, R.: The nonstochastic multiarmed bandit problem. SIAM Journal on Computing **32**(1), 48–77 (2002)
5. Balafrej, A., Bessiere, C., Paparrizou, A.: Multi-armed bandits for adaptive constraint propagation. In: Proceedings of IJCAI'15. pp. 290–296 (2015)
6. van Beek, P.: Backtracking search algorithms. In: Handbook of Constraint Programming, chap. 4, pp. 85–134. Elsevier (2006)
7. Bessiere, C., Régin, J.: MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In: Proceedings of CP'96. pp. 61–75 (1996)
8. Bessiere, C., Zanuttini, B., Fernandez, C.: Measuring search trees. In: Proceedings of ECAI'04 workshop on Modelling and Solving Problems with Constraints. pp. 31–40 (2004)
9. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: Proceedings of ECAI'04. pp. 146–150 (2004)
10. Bubeck, S., Cesa-Bianchi, N.: Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems. Foundations and Trends in Machine Learning, Now Publishers (2012)
11. Gecode Team: Gecode: Generic constraint development environment (2006), available from `http://www.gecode.org`
12. Gomes, C., Selman, B., Crato, N., Kautz, H.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. Journal of Automated Reasoning **24**(1), 67–100 (2000)
13. Gomes, C.P., Selman, B., Kautz, H.: Boosting combinatorial search through randomization. In: Proceedings of AAAI '98. pp. 431–437 (1998)
14. Grimes, D., Wallace, R.: Sampling strategies and variable selection in weighted degree heuristics. In: Proceedings of CP'07. pp. 831–838 (2007)
15. Habet, D., Terrioux, C.: Conflict History Based Branching Heuristic for CSP Solving. In: Proceedings of the 8th International Workshop on Combinations of Intelligent Methods and Applications (CIMA). Volos, Greece (Nov 2018)
16. Haralick, R., Elliott, G.: Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence **14**, 263–313 (1980)
17. Harvey, W.D., Ginsberg, M.L.: Limited discrepancy search. In: Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1. p. 607–613. IJCAI'95 (1995)
18. Hebrard, E.: Mistral, a constraint satisfaction library. Proceedings of the Third International CSP Solver Competition **3**(3), 31–39
19. Hogg, T., Huberman, B.A., Williams, C.P.: Phase transitions and the search problem. Artificial Intelligence **81**(1), 1 – 15 (1996)
20. Lecoutre, C., Sais, L., Tabary, S., Vidal, V.: Recording and minimizing nogoods from restarts. Journal on Satisfiability, Boolean Modeling and Computation (JSAT) **1**, 147–167 (2007)
21. Loth, M., Sebag, M., Hamadi, Y., Schoenauer, M.: Bandit-based search for constraint programming. In: Proceedings of CP'13. pp. 464–480 (2013)

22. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of las vegas algorithms. Information Processing Letters **47**(4), 173–180 (1993)
23. Mackworth, A.: On reading sketch maps. In: Proceedings of IJCAI'77. pp. 598–606 (1977)
24. Merchez, S., Lecoutre, C., Boussemart, F.: Abscon: a prototype to solve CSPs with abstraction. In: Proceedings of CP'01. pp. 730–744 (2001)
25. Michel, L., Hentenryck, P.V.: Activity-based search for black-box constraint programming solvers. In: Proceedings of CPAIOR'12. pp. 228–243 (2012)
26. Pisinger, D., Ropke, S.: Large neighborhood search. In: Handbook of metaheuristics. pp. 399–419. Springer (2010)
27. Prud'homme, C., Fages, J.G., Lorca, X.: Choco Solver Documentation. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. (2016), http://www.choco-solver.org
28. Sabin, D., Freuder, E.: Contradicting conventional wisdom in constraint satisfaction. In: Proceedings of CP'94. pp. 10–20 (1994)
29. Selman, B., Kautz, H.A., Cohen, B.: Noise strategies for improving local search. In: Proceedings of AAAI '94. pp. 337–343 (1994)
30. Selman, B., Levesque, H., Mitchell, D.: A new method for solving hard satisfiability problems. In: Proceedings of the Tenth National Conference on Artificial Intelligence. pp. 440–446. AAAI'92, AAAI Press (1992), http://dl.acm.org/citation.cfm?id=1867135.1867203
31. Sutton, R., G. Barto, A.: Reinforcement learning: An introduction **9**, 1054 (02 1998)
32. Thompson, W.R.: On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. Biometrika **25**(3-4), 285–294 (12 1933)
33. Walsh, T.: Search in a small world. In: Proceedings of IJCAI'99. pp. 1172–1177 (1999)
34. Wattez, H., Koriche, F., Lecoutre, C., Paparrizou, A., Tabary, S.: Heuristiques de recherche : un bandit pour les gouverner toutes. In: 15es Journées Francophones de Programmation par Contraintes – JFPC 2019 (2019), https://hal.archives-ouvertes.fr/hal-02414288
35. Wattez, H., Koriche, F., Lecoutre, C., Paparrizou, A., Tabary, S.: Learning variable ordering heuristics with multi-armed bandits and restarts. In: Proceedings of ECAI'20 (to appear)
36. Wattez, H., Lecoutre, C., Paparrizou, A., Tabary, S.: Refining constraint weighting. In: Proceedings of ICTAI'19. pp. 71–77 (2019)
37. Xia, W., Yap, R.H.C.: Learning robust search strategies using a bandit-based approach. In: Proceedings of AAAI'18. pp. 6657–6665 (2018)