24th European Conference on Artificial Intelligence - ECAI 2020
Santiago de Compostela, Spain

Learning Variable Ordering Heuristics with Multi-Armed Bandits and Restarts

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

Abstract. In constraint-based applications, the user is often required to be an expert as, for a given problem instance, many parameters of the used solver must be manually tuned to improve its efficiency. Clearly, this background knowledge burdens the spread of constraint programming technology to non-expert users. In order to alleviate this issue, the idea of "autonomous" constraint solving is to adjust the solver parameters and to efficiently handle any problem instance without manual tuning. Notably, the choice of the variable ordering heuristic can lead to drastically different performances. A key question arises then: how can we find the best variable ordering heuristic for a problem instance, given a set of available heuristics provided by the solver? To answer this question, we propose an algorithmic framework that combines multi-armed bandits and restarts. Each candidate heuristic is viewed as an arm, and the framework learns to estimate the best heuristic using a multi-armed bandit algorithm. The common mechanism of restarts is used to provide feedback for reinforcing the bandit algorithm. Based on a thorough experimental evaluation, we demonstrate that this framework is able to find the best heuristic for most problem instances; notably, it outperforms the state-of-the-art in terms of time and solved instances.

1 INTRODUCTION

Constraint Programming (CP) has long been recognized as a powerful paradigm for modelling and solving combinatorial search problems, with numerous applications ranging from configuration, planning and scheduling to bio-informatics, network design and software validation. The key motto of Constraint Programming is to alleviate as much as possible the cognitive burden of combinatorial search by using general-purpose constraint solvers. Ideally, the user should only focus on modelling her task as a set of decision variables, together with a set of constraints specifying which relations hold among the variables. Based on this description, the *Constraint Satisfaction Problem* (CSP) is to find an assignment of all variables that satisfies all the constraints. Solving this problem should be entirely dedicated to the constraint solver, that interleaves search and inference in order to efficiently explore the space of possible assignments [12, 3, 34, 21].

The reality is however different. Modern constraint solvers are equipped with various components whose careful tuning can significantly increase their efficiency. Among those components, the *variable ordering heuristic* takes a central place in backtracking search by ordering the variables to be visited in the search tree. Choosing the right variable ordering heuristic for a given constraint network can drastically affect the solving time by significant speed-ups, since different heuristics can lead to entirely different search trees. For this purpose, several heuristics have been proposed and analyzed for variable ordering. Therefore, the question that arises is: given a CSP instance and a set of variable ordering heuristics available in the solver, which heuristic is the best one for solving the instance? Since no existing heuristic has proven to be optimal on all possible CSP instances, finding the right heuristic for the task at hand is not a straightforward exercise, except maybe for an expert.

This research is in line with *autonomous* constraint solvers, which adopt Machine Learning techniques for adjusting the solver parameters on a given problem instance without human intervention [18]. This work proposes to use *Multi-Armed Bandit* (MAB) techniques for estimating the best variable ordering heuristic on a CSP instance. Informally, a (basic) MAB problem is a sequential decision task in which the bandit algorithm has at its disposal a set of arms, and observes the reward for the chosen arm after each trial. In our setting, each arm is a variable ordering heuristic available in the constraint solver, and trials are realized by using a *restart* mechanism [15]. Namely, the constraint solver operates through a sequence of "runs", whose termination is determined by a cutoff function. During each run, the solver explores a search tree using the heuristic selected by the bandit algorithm. When reaching the cutoff, the solver abandons the current search and restarts, while the bandit algorithm receives a reward for the selected arm. The reward reflects the performance of the solver on the corresponding heuristic. The overall goal is to learn which heuristic is the best for a given CSP instance. This is done by interleaving exploration (trying out different heuristics for acquiring new information) and exploitation (selecting the optimal heuristic based on the available information) during runs.

This conceptually simple framework for autonomous constraint solving can be easily adopted by any CP solver. The framework is generic and can be instantiated using different cutoff functions, bandit policies, and reward functions. For the bandit policies, we have examined three well-studied algorithms: `EXP3` [5], `UCB1` [4], and Thompson Sampling (`TS`) [35]. Finally, we have considered several reward functions for capturing the performance of the heuristic on each run. The most promising metric that emerges from our empirical analysis is the *pruned tree size* which reflects the solver's ability to efficiently infer "early domain wipeouts" during backtracking search.

We conducted an extensive evaluation on various benchmark problems, involving several well-known heuristics. In a nutshell, the proposed framework is able to solve more instances in less time than the original solver equipped with any fixed heuristic. We also compare to a state-of-the-art framework (detailed in the following) showing that

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

our framework admits better performance.

2 RELATED WORK

Autonomous constraint solving has received increasing attention in the CP community, as evidenced by the diversity of work using Machine Learning (ML) techniques for automatically tuning some components of the solver. These approaches can roughly be divided into two categories, depending on the learning paradigm they rely on. The first category is using *supervised learning*: given a space $\mathcal{S}$ of configurations for some component of the solver, we start from a sample of CSP instances, each labeled with the best configuration in $\mathcal{S}$, and then learn a hypothesis mapping CSP instances into $\mathcal{S}$. In this category, Balcan et al. [7] have recently developed a supervised learning framework for estimating the best variable ordering heuristic ($\mathcal{S}$ is a simplex over scoring rules). In [13], a *new* strategy (e.g., a combination of search algorithms and heuristics) is synthetized through the use of ML. Supervised learning approaches for constraint solving are closely related to *portfolio techniques*, where $\mathcal{S}$ is a set of candidate solvers or algorithms. Portfolios have been proposed for SAT [42], parallel SAT [25], QBF [32], ASP [29, 19] and CP [31, 20, 2].

The second category relies on *reinforcement learning*: using again a space $\mathcal{S}$ of configurations for some component of the solver, the best configuration in $\mathcal{S}$ is estimated during search, by observing a feedback on each selected configuration. Our framework, together with other approaches using MAB for constraint solving [6, 14, 26, 41], belongs to this category. Gagliolo and Schmidhuber [14] apply the `EXP3` bandit algorithm for learning restart strategies, while Balafrej et al. [6] use `UCB1` for selecting different levels of propagation during search. In [26], multi-armed bandits are exploited to select which node of a *Monte Carlo Tree Search* must be extended. In SAT, the bandit algorithm ERWA (Exponential Recency Weighted Average) is used as a heuristic to choose the next variable to branch on [23, 24]. This principle is also applied for CSPs by the recent variable ordering heuristic called `CHS` [17].

Arguably, the closest work to ours is of Xia and Yap [41], which applies existing bandit algorithms (`UCB1` and `TS`) for estimating the best variable ordering heuristic, given a predefined set of candidate heuristics (arms). In their framework, a single search tree is explored (there are no restarts), and the bandit algorithm is called at each node of the tree. Specifically, for each unexplored node, the algorithm selects an arm and uses the corresponding heuristic to instantiate a variable at that node. A mean reward for the arm bound to this node is recursively computed according to the rewards observed on each child. From a conceptual viewpoint, the key difference between this approach and ours lies in the characterization of "trials" during the sequential learning process. In this approach, trials are associated with explored subtrees, while in our approach, trials are mapped to runs using a restart mechanism. In the evaluation part, we have compared both approaches.

3 BACKGROUND

A *Constraint Network* P consists of 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 must takes a value from a finite domain, denoted by $\texttt{dom}(x)$. Each constraint c is specified by a relation $\texttt{rel}(c)$ over a set of variables, called the *scope* of c, and denoted by $\texttt{scp}(c)$. The *arity* of a constraint c is the size of its scope. The *degree* of a variable x is the number of constraints in $\texttt{ctrs}(P)$ involving x in its scope. 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 or not a given constraint network is satisfiable.

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 GAC (*Generalized Arc Consistency*) [28] after each decision. This procedure, called *Maintaining Arc Consistency* (MAC) [37], builds a binary search tree $\mathcal{T}$: for each internal node ν of $\mathcal{T}$, a pair (x, v) is selected where x is an unfixed variable and v is a value in $\texttt{dom}(x)$. An unfixed variable is a variable that has not already been selected for building $\mathcal{T}$. Then, two cases are considered: the assignment $x = v$ (positive decision) and the refutation $x \neq v$ (negative decision). The order in which variables are chosen during the depth-first traversal of the search space is decided by a *variable ordering heuristic*, denoted here H. Namely, at each internal node ν of the search tree $\mathcal{T}$, the MAC procedure selects the next variable x using H, and assigns to x a value v according to its value ordering heuristic, which is usually the lexicographic order over $\texttt{dom}(x)$.

Choosing the right variable ordering heuristic H for a given CSP is a key issue in constraint solving, as one choice from another might provoke differences of orders of magnitude in solving time. To this end, several heuristics have been proposed in the literature including, among others, `activity` [30], `impact` [33], `dom/ddeg` [38], `CHS` [17] and $\texttt{wdeg}^{\texttt{ca.cd}}$ [40] (refinement of `dom/wdeg` [9]). For example, `dom/ddeg` gives priority to the variable with the smallest ratio "current domain size to dynamic degree", where the dynamic degree of a variable x is the number of constraints involving x and at least another unfixed variable.

Finally, modern constraint solvers are equipped with a *restart* function for addressing the heavy-tailed behavior on both random and real-world instances [15]. In essence, this function is a mapping $\texttt{cutoff}^{\texttt{rs}} : \mathbb{N} \rightarrow \mathbb{N}$, where $\texttt{cutoff}^{\texttt{rs}}(t)$ is the maximal number of "steps" that can be performed by the backtracking search algorithm at run t according to a restart strategy `rs`. A constraint solver, equipped with the MAC procedure and a restart strategy `rs`, builds a sequence of binary 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. Importantly, even if the solver restarts from the beginning, it can memorize some relevant information about the sequence $\langle \mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{t-1} \rangle$. For example, the solver may extract the nogoods from the last branches in order to avoid exploring later the same parts of the search tree, or can maintain in a cache the nogoods which have appeared frequently in the search trees explored so far [22].

The *cutoff*, which is the number of allowed steps in a run, may be defined by the number of backtracks, the number of wrong decisions [8], or any other relevant measure. In a *fixed* cutoff restart strategy, the number T of runs is fixed in advance, and $\texttt{cutoff}^{\texttt{rs}}(t)$ is constant for each run t, except for the T^{th} run which allows an unlimited number of steps (in order to have a complete algorithm). This strategy is known to be effective in practice [16], but a good cutoff value $\texttt{cutoff}^{\texttt{rs}}(t)$ has to be found by trial and error. Alternatively, in a *dynamic* cutoff restart strategy, the number T of runs is unknown, but `rs` increases the cutoff geometrically, which guarantees that the whole space of partial solutions is explored after $O(n)$ runs. A commonly used cutoff function is the Luby sequence [27]: the number of

steps per run is given by $u \times l_t$, where u is a constant and

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

4 THE MULTI-ARMED BANDIT FRAMEWORK

Based on the above technical background, we are now in the position to formulate our task: given a constraint network P and a set of variable ordering heuristics $H_1, \cdots, H_K$, which is the best one for solving P? This task is cast as a K-armed bandit problem, where each arm is a candidate heuristic. To this point, recall that a MAB problem is a sequential decision-making process in which the bandit algorithm interacts with its environment. During each trial t, the algorithm selects an arm i_t in $\{1, \cdots, K\}$ and receives a reward $r_t(i_t)$ for this arm. The goal is to minimize the expected regret over T trials, which is defined as the expectation of the difference between the total reward obtained by the best arm and the total reward obtained by the bandit algorithm. Importantly, the algorithm observes the reward for the chosen arm after each trial, but not for the other arms that could have been selected. Therefore, the minimization of regret is achieved by balancing *exploration* (acquiring new information) and *exploitation* (using acquired information).

In order to further advance in the specification of our MAB framework, we need to clarify the notions of "trials" and "rewards", and of course, we need to choose appropriate bandit policies for the task of selecting the best heuristic during constraint solving. The key idea of this paper is to consider trials as *runs*, by exploiting a restart mechanism that uses the Luby sequence as cutoff function. So, during each run t, the bandit algorithm selects an arm i_t, and the MAC algorithm is run using the corresponding heuristic H_{i_t}. When the cutoff $\texttt{cutoff}^{\texttt{rs}}(t)$ is reached, the bandit algorithm receives a reward $r_t(i_t)$ that captures the performance of the solver. Notably, this feedback can exploit some information about the binary search tree $\mathcal{T}_t$ explored by the MAC algorithm at run t.

Algorithm 1: MAB Framework

Input: constraint network P, heuristics $H_1, \cdots, H_K$, bandit policy B

Initialize the bandit policy
1 INITARMS$_B(K)$

Trials
2 **for** *each run* $t = 1, \cdots, T$ **do**
Select an arm according to the bandit policy
3 $i_t \leftarrow$ SELECTARM$_B(K)$
Run the solver and observe a reward
4 $r_t(i_t) \leftarrow \text{MAC}(H_{i_t})$
Update the bandit policy
5 UPDATEARMS$_B(r_t)$

With these notions in hand, our algorithmic framework is given by Algorithm 1. The framework takes as input a constraint network P, a set of variable ordering heuristics $\{H_1, \cdots, H_K\}$, and a bandit policy B. The three main procedures of this policy are INITARMS$_B$ for initializing the parameters of the bandit policy, SELECTARM$_B$ for choosing the arm (heuristic) that will be used to guide the search all along the run, and UPDATEARMS$_B$ for updating the parameters of the bandit policy according to the observed reward at the end of the run. The rest of this section is devoted to the specification of the reward function and the bandit policies.

4.1 Reward function

The tricky part of our framework is to define an appropriate reward function that maps, at each run, the performance of the MAC algorithm to a numeric feedback. For example, the reward function can be defined as the CPU time required to complete a run, the average size of nogoods inferred by the MAC algorithm, the number of values removed through propagation, the number of "domain wipeouts" which are empty domains inferred during search, or the number of "wrong decisions" which are variable assignments leading to a full exploration of the subtree without finding a solution.

We performed an empirical study on such possible metrics, and the reward function that emerges from our analysis is the *pruned tree sizes* (pts). Namely, pts solved 3 to 10 more instances in a slightly better time, compared to the aforementioned measures.

Intuitively, this measure is given by the sum of the sizes of the subtrees rooted at a node that has yielded a domain wipeout. Since any such node is a dead-end, pts captures the ability of the solver to quickly prune large portions of its search space. In formal terms, given a binary search tree $\mathcal{T}$ generated by the MAC algorithm, let $\texttt{dwo}(\mathcal{T})$ be the set of nodes for which at least one variable has an empty domain, and let $\texttt{fut}(\nu)$ be the set of variables that are left unfixed at node ν. Then,

$$\texttt{pts}(\mathcal{T}_t) = \sum_{\nu \in \texttt{dwo}(\mathcal{T}_t)} \prod_{x \in \texttt{fut}(\nu)} |\texttt{dom}(x)| \quad (2)$$

Based on this metric, the reward for the arm i_t selected by the bandit policy at run t is given by:

$$r_t(i_t) = \frac{\log(\texttt{pts}(\mathcal{T}_t))}{\log\left(\prod_{x \in \texttt{vars}(P)} |\texttt{dom}(x)|\right)} \quad (3)$$

A logarithmic scaling is needed to obtain a more balanced reward between 0 et 1. Bigger is the cumulative size of pruned trees, better is the reward given to the bandit.

4.2 Bandit policies

As indicated in Algorithm 1, our algorithmic framework is conceptually simple: based on a restart mechanism, the bandit algorithm performs each run by first selecting an arm (heuristic), next observing a reward for that arm, and then updating its policy according to the observed reward. This simplicity allows us to use easily implementable and computationally efficient bandit policies. From this perspective, we have opted for three well-studied policies for the K-arm bandit problem, which are summarized below.

EXP3 policy. When very little is known about the reward functions $r_1, \cdots, r_t$, the exponentially weighted forecaster for exploration and exploitation [5] is arguably the prime candidate for the K-armed bandit problem. Namely, EXP3 can operate in non-stochastic environments, for which no statistical assumption is made about the reward functions. Using only the fact that the range of each r_t is bounded, EXP3 is guaranteed to converge towards the best arm (heuristic) by achieving an expected regret in $O(\sqrt{T})$. Several variants of EXP3 have been proposed in the literature, but we use here the simplest version defined in [11].

EXP3 maintains a probability distribution π_t over $\{1, \cdots, K\}$. Specifically, the procedure INITARMS$_{\text{EXP3}}$ sets the initial vector π_1 to the uniform distribution $(1/K, \cdots, 1/K)$. During each trial t, the procedure SELECTARM$_{\text{EXP3}}$ simply draws an arm i_t according to the distribution π_t. Based on the observed reward $r_t(i_t)$, the procedure UPDATEARMS$_{\text{EXP3}}$ updates the distribution π_t according to the multiplicative weight-update rule:

$$\pi_{t+1}(i) = \frac{\exp(\eta_t R_t(i))}{\sum_{j=1}^{K} \exp(\eta_t R_t(j))} \tag{4}$$

where

$$R_t(i) = \sum_{s=1}^{t} \frac{r_s(i)}{\pi_s(i)} \mathbb{1}_{i \sim \pi_s} \tag{5}$$

and $\mathbb{1}_{i \sim \pi_s}$ indicates whether a was the arm picked at trial s, or not. Using the fact that, in our framework, the range of all rewards functions is in $[0, 1]$, the step-size parameter η_t can be set to $\sqrt{\ln K / tK}$ in order to obtain an $O(\sqrt{TK \ln K})$ regret bound.

UCB policy. Upper Confidence Bound policies are commonly used in stochastic environments [5]. Here it is assumed that the reward value $r_t(i)$ of each arm i is drawn according to a fixed, but unknown, probability distribution. In the setting of our framework, $r_t(i)$ is determined by $\texttt{pts}(\mathcal{T}_t)$ according to (3), which in turn is determined by the binary search tree $\mathcal{T}_t$ generated by the MAC algorithm at run t, according to (2). For a fixed heuristic H_i and a fixed cutoff value $\texttt{cutoff}^{\texttt{rs}}(t)$, MAC is expected to build the same tree $\mathcal{T}_t$. Thus, it is not unreasonable to assume that $r_t(i)$ is drawn according to a fixed (yet hidden) distribution over $[0, 1]$.

Based on these considerations, we use UCB1 which is the simplest algorithm in the Upper Confidence Bound family. This algorithm maintains two K-dimensional vectors, namely, $n_t(i)$ is the number of times the policy has selected arm i on the first t runs, and $\hat{r}_t(i)$ is the empirical mean of $r_t(i)$ during the $n_t(i)$ steps. INITARMS$_{\text{UCB1}}$ sets both vectors to zero and, at each run t, SELECTARM$_{\text{UCB1}}$ selects the arm that maximizes:

$$\hat{r}_t(i_t) + \sqrt{\frac{8 \ln(t)}{n_t(i_t)}} \tag{6}$$

Finally, UPDATEARMS$_{\text{UCB1}}$ updates the vectors n_t and $\hat{r}_t$ according to i_t and $r_t(i_t)$, respectively. Under the assumption that, for each arm i, $r_t(i)$ is drawn (independently at random) according to a fixed distribution over $[0, 1]$, UCB1 also achieves an expected regret in $O(\sqrt{TK \ln K})$.

TS policy. The Thompson Sampling algorithm is another well-known policy used in stochastic environments [1, 39, 36]. In essence, the TS algorithm maintains a beta distribution for the rewards of each arm. INITARMS$_{\text{TS}}$ sets $\alpha_1(i)$ and $\beta_1(i)$ to 1 for $i \in \{1, \cdots, K\}$. On each run t, SELECTARM$_{\text{TS}}$ selects the arm i_t that maximizes $\text{Beta}(\alpha_t(i), \beta_t(i))$, and UPDATEARMS$_{\text{TS}}$ uses $r_t(i_t)$ to update the beta distribution as follows:

$$\alpha_{t+1}(i) = \alpha_t(i) + \mathbb{1}_{i=i_t} r_t(i_t) \tag{7}$$

$$\beta_{t+1}(i) = \beta_t(i) + \mathbb{1}_{i=i_t}(1 - r_t(i_t)) \tag{8}$$

As for UCB1, the convergence of TS relies on the assumption that rewards are drawn (independently at random) according to a fixed distribution. Under this assumption, the TS algorithm achieves an expected regret which is again in $O(\sqrt{TK \ln K})$ [1].

5 EXPERIMENTAL EVALUATION

We have conducted some experiments on two benchmarks in XCSP3 format [10], so as to demonstrate the practical interest of the proposed framework. The first benchmark, called [XCSP17], includes all CSP instances (612 in total) from the 2017 XCSP3 competition[2] coming from 60 different problems. The second one, called [XCSP-ALL], is a larger set containing all CSP instances (more than $16,000$) available on the XCSP3 website[3] in order to give us an overall performance view. Experiments have been launched on a cluster of 2.66 GHz Intel Xeon with 32 GB RAM nodes. We have used the solver AbsCon[4] where we integrated our bandit model as well as the one proposed in [41]. The restart policy is the Luby progression (where the cutoff is based on the number of visited nodes) and the timeout has been set to 1 hour. Concerning restarts, we also tried CPU time, number of failures and value deletions as cutoff metrics, but results were either non deterministic or less effective. We have tested both bandit models under the same variable ordering heuristics: activity [30], impact [33], dom/ddeg [38], CHS [17] and wdeg$^{\texttt{ca.cd}}$ [40]. These heuristics are efficient ones, competitive to each other, which makes the task of learning even tougher for discriminating the best arm. We have also run them separately for a baseline comparison. In the following, we will refer to our bandit model as RestartsMAB and to the bandit model of [41] as NodesMAB. From now on, any execution of AbsCon with a particular way of selecting variables (either with a bandit policy or a classical ordering heuristic) will be referred to as a (solving) *method*.

As the model NodesMAB is not based on restarts, a straightforward comparison with the original NodesMAB would not be fair. Therefore, we run NodesMAB with a restart policy (and learning activated at each run), as in our approach. This results in much better performance, as displayed in Table 1. The comparison is given by the number (#inst) of instances solved within $3,600$ seconds, and the cumulative CPU time (c.time) computed from the 332 instances solved by the two alternatives (restarts and no-restart). Numbers highlighted in bold correspond to the best obtained results.

Table 1: Comparison of NodesMAB when restarts are activated (restarts) or not activated (no-restart) on the [XCSP17] benchmark.

	restarts	no-restart
#inst	**371**	338
c.time (332)	**18,824**	19,720

Table 2: Comparison of classical heuristics and bandit variants on the [XCSP17] benchmark.

	#inst	c.time (296)	By1	By2	By10
dom/ddeg	324	17,727	662,521	1,296,120	6,364,920
NodesMAB$^{\texttt{EXP3}}$	352	10,882	553,867	1,086,670	5,349,070
Activity	357	8,541	540,879	1,055,680	5,174,080
NodesMAB$^{\texttt{TS}}$	361	8,652	522,838	1,023,240	5,026,440
Impact	365	21,348	529,180	1,015,180	4,903,180
NodesMAB$^{\texttt{UCB}}$	371	10,075	498,394	962,794	4,677,990
RestartsMAB$^{\texttt{EXP3}}$	375	**5,103**	470,749	920,749	4,520,750
CHS	380	6,668	458,638	890,638	4,346,640
RestartsMAB$^{\texttt{TS}}$	380	5,965	462,243	894,243	4,350,240
wdeg$^{\texttt{ca.cd}}$	381	8,282	460,039	888,439	4,315,640
RestartsMAB$^{\texttt{UCB}}$	**386**	5,189	**440,956**	**851,356**	**4,134,560**

[2] See http://www.cril.univ-artois.fr/XCSP17
[3] See http://www.xcsp.org
[4] See http://www.cril.fr/~lecoutre/#/softwares

In Table 2, the models RestartsMAB and NodesMAB are compared under all bandit policies (UCB, TS, EXP3); as indicated earlier, classical heuristics are used as arms. Results of classical heuristics are also given in the table. For the comparison, we use some additional time metrics, denoted by By1, By2, By10, which are cumulative CPU times, computed from all instances by considering for each unsolved instance a solving time equal to $x \times 3,600$ seconds for $x = 1$, $x = 2$ and $x = 10$, respectively. We observe that all variants (i.e., specific uses of a bandit policy) of NodesMAB are outperformed by all the variants of RestartsMAB. The variant $\texttt{RestartsMAB}^{\texttt{UCB}}$ has respectively solved 15 and 34 more instances than the worst and best variant of NodesMAB. In addition, on the 296 *common* instances (i.e., solved by all methods), it was twice as fast. Note that the variants of RestartsMAB outperform most of the classical heuristics, and that the efficient heuristics CHS and $\texttt{wdeg}^{\texttt{ca.cd}}$ are both dominated by $\texttt{RestartsMAB}^{\texttt{UCB}}$. In terms of solving (cumulated) times, all variants of RestartsMAB display close performances.

Figure 1 shows similar results by means of a cactus plot, in which the virtually best solver (VBS) is also displayed; VBS is an oracle solver that knows the optimal heuristic for a given instance. The more a solver is close to the VBS the better the solver is. The cactus plot provides a global view of performance, where we can see the number of instances solved per method (x-axis) while time increases (y-axis). $\texttt{RestartsMAB}^{\texttt{UCB}}$ is, as shown previously, the most efficient solving method; its curve is closer to VBS than any other curve.

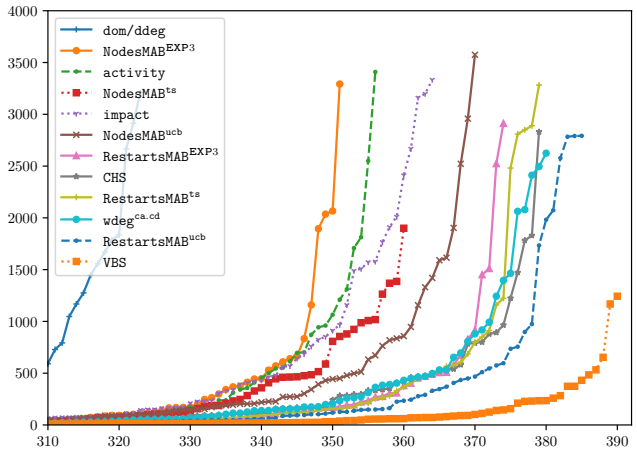
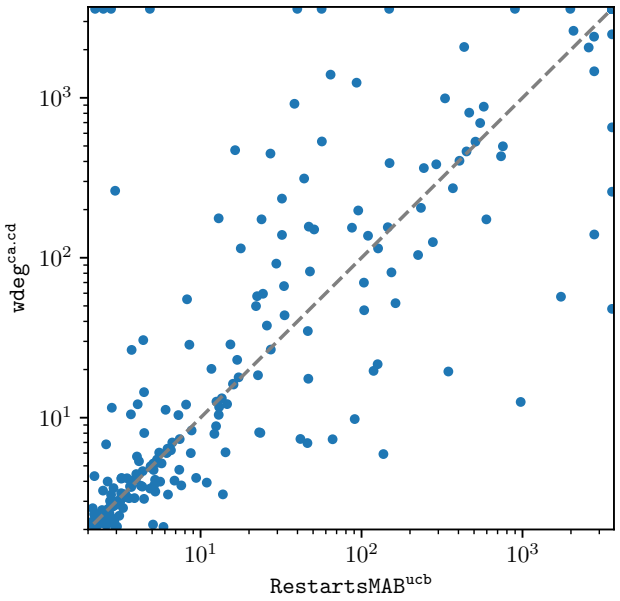
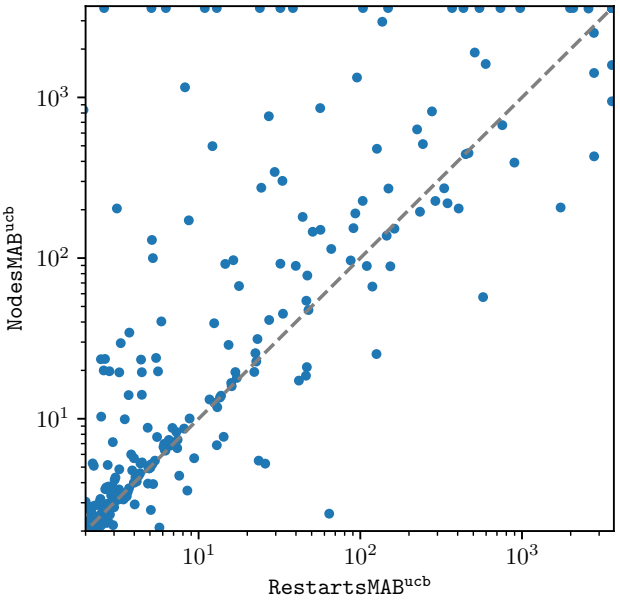


Figure 1: Comparison of classical heuristics, bandit variants and VBS on the [XCSP17] benchmark.

Figure 2 provides a pairwise comparison of the best variant of our model, $\texttt{RestartsMAB}^{\texttt{UCB}}$, against the best classical heuristic, $\texttt{wdeg}^{\texttt{ca.cd}}$ and the best variant of the first introduced bandit model $\texttt{NodesMAB}^{\texttt{UCB}}$. Each dot represents an instance and each coordinate of this dot on an axis (displayed in logarithmic scale) is the time taken to solve it by the method labelling the axis. Dots over the diagonal correspond to instances solved faster by $\texttt{RestartsMAB}^{\texttt{UCB}}$. In Figure 2a, the instances that are concentrated around the diagonal are solved with similar efficiency by both $\texttt{wdeg}^{\texttt{ca.cd}}$ and RestartsMAB, while a few instances are solved faster by $\texttt{RestartsMAB}^{\texttt{UCB}}$. Interestingly, there are some instances in the top left corner, meaning that these instances are quickly solved by $\texttt{RestartsMAB}^{\texttt{UCB}}$ whereas $\texttt{wdeg}^{\texttt{ca.cd}}$ reaches the timeout. Figure 2b clearly shows that $\texttt{RestartsMAB}^{\texttt{UCB}}$ significantly outperforms $\texttt{NodesMAB}^{\texttt{UCB}}$.



(a) $\texttt{RestartsMAB}^{\texttt{UCB}}$ vs $\texttt{wdeg}^{\texttt{ca.cd}}$



(b) $\texttt{RestartsMAB}^{\texttt{UCB}}$ vs $\texttt{NodesMAB}^{\texttt{UCB}}$

Figure 2: Pairwise comparisons of solving methods on the [XCSP17] benchmark.

Regarding the second benchmark, [XCSP-ALL], Table 3 shows the results obtained with the two best variable ordering heuristics, namely CHS and $\texttt{wdeg}^{\texttt{ca.cd}}$, as well as with the best variants of NodesMAB and RestartsMAB (i.e., with UCB as bandit policy). Here, $\texttt{NodesMAB}^{\texttt{UCB}}$ is the worst solving method, as it respectively solves 189 and 216 less instances than CHS and $\texttt{wdeg}^{\texttt{ca.cd}}$, whereas $\texttt{RestartsMAB}^{\texttt{UCB}}$ is the best solving method as it respectively solves 95 and 68 more instances than CHS and $\texttt{wdeg}^{\texttt{ca.cd}}$. One might think that 68 is a small number compared to the whole set of thousand instances, but these instances actually correspond to hard in-

stances that are not solved by any other method within $3,600$ seconds. In fact, #inst is saturated by easy instances. By employing the By1 measure, we find that RestartsMAB will finish the task in at least 65 less hours than any other method.

Table 3: Comparison of best classical heuristics and bandit variants (with UCB) on the [XCSP-ALL] benchmark.

	CHS	wdeg$^{ca.cd}$	RestartsMAB^{UCB}	NodesMAB^{UCB}
#inst	14,157	14,184	**14,252**	13,968
c. time (13,368)	271,387	234,687	**224,534**	559,794
By1	9,464,830	9,239,700	**9,003,260**	10,134,300
By2	18,234,400	17,912,100	**17,430,900**	19,584,300
By10	88,391,200	87,291,300	**84,851,700**	95,184,300

To establish the aforementioned statement that RestartsMAB is even better on hard instances, we present Table 4, where the set of instances of the initial benchmark is split by considering less and less easy instances in the following way: at each row $> \alpha$s, we select the instances that have been solved by the best method in more than α seconds. As mentioned above, the majority of the instances were easy (for all or some heuristics); this is why the numbers in the table never exceeds 600. In each cell of Table 4, next to the number of solved instances, we have put between brackets the percentage of additional instances that have been solved compared to the worst method in the same row. As we can see, RestartsMAB^{UCB} is the most efficient method on these difficult instances: for example, RestartsMAB^{UCB} solves between 60% and 105% more instances than NodesMAB^{UCB}. Note that RestartsMAB^{UCB} tends to break away as the instances become harder and harder. The harder the instance is, the more the multi-armed bandit has time to learn and improve.

Table 4: Number of hard instances being solved by four different solving methods, on subsets of instances of increasing difficulty from the [XCSP-ALL] benchmark.

	CHS	wdeg$^{ca.cd}$	RestartsMAB^{UCB}	NodesMAB^{UCB}
$>$ 100s	582 (+55%)	563 (+50%)	**600** (+**60**%)	375 (+0%)
$>$ 250s	363 (+73%)	341 (+62%)	**384** (+**83**%)	210 (+0%)
$>$ 500s	239 (+87%)	212 (+66%)	**262** (+**105**%)	128 (+0%)
$>$ 1000s	132 (+57%)	117 (+39%)	**162** (+**93**%)	84 (+0%)

Finally, Figure 3 shows for three different structured problems the behaviour of all methods (classical heuristics and bandit variants); the distribution of arm calls is also given for all bandit variants. Inside each bar-diagram, each color represents a different heuristic, and thus, the MAB variants are colorful, displaying the proportion of use of each heuristic; the height of the bar corresponds to the time needed to solve the full series of the problem. Figure 3a shows that, for Problem SocialGolfers, the heuristics CHS and wdeg$^{ca.cd}$ are very efficient. Hence, they ideally should be chosen in priority by the learning algorithms. This is the case for variants of RestartsMAB, and especially RestartsMAB^{UCB}, contrary to variants of NodesMAB. For Problem LatinSquare, one can see in Figure 3b that the best monolithic heuristic is wdeg$^{ca.cd}$. Interestingly, all variants of RestartsMAB outperform all other methods, and even the best arm wdeg$^{ca.cd}$, despite that less efficient arms are also used during exploration. What is hidden in the diagrams is the moment each arm is called. Actually, the best arm is mainly called at the last run, where the bandit has learned it. Remember that last runs are longer, which justifies the good obtained time performance. For this problem, the variants NodesMAB^{TS} and NodesMAB^{EXP3} are totally misled. In Figure 3c, for Problem KnightTour, we observe an unusual behavior of all bandit variants. Although CHS is clearly the best heuristic, this is not reflected in the bar of any variant. Neither the reward function of RestartsMAB nor the reward function of NodesMAB was able to discriminate the best choice and pass it to the learning algorithm.

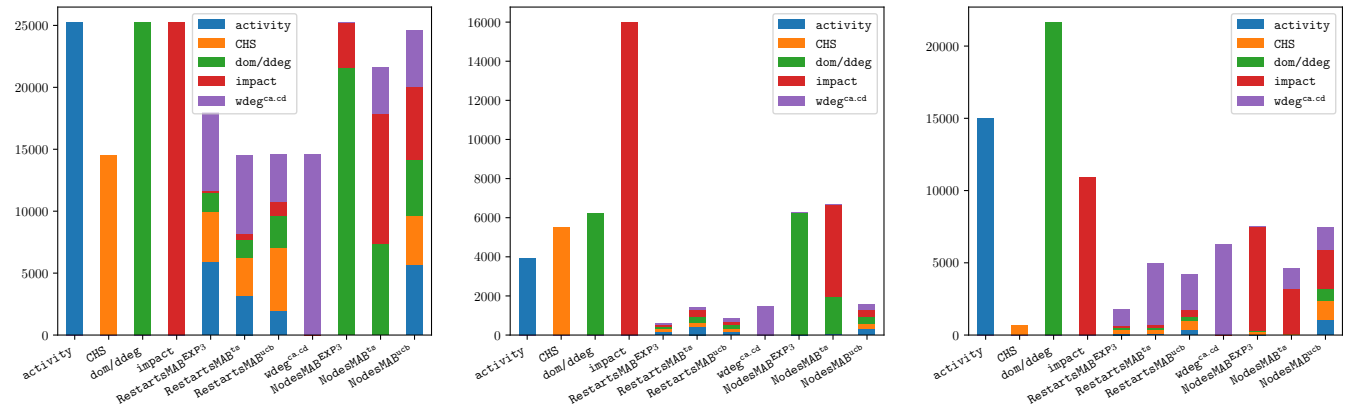
6 DISCUSSION

As already mentioned, the model proposed in [41] also addresses the issue of learning the right variable ordering heuristic to be used for solving a given instance. The method NodesMAB calls a multi-armed bandit algorithm at each node ν, and the reward for the arm used at ν is based on the number of visited nodes in the sub-tree rooted at ν. Two bandit policies, UCB and TS, are tested, and it is shown experimentally that they contribute to make the search more robust than stand-alone classical heuristics. In this section, we highlight some differences with respect to our approach.

Model. In our opinion, with NodesMAB, a first question arises from the fact that the bandit selects a new heuristic at each search node. So, the meaning of variable ordering is unclear as a single heuristic is not used to determine an order over several consecutive variables, but instead, at each level a variable is possibly chosen by a different heuristic. As, in the beginning, MAB mainly performs exploration, NodesMAB will not allow a single heuristic, even if it is the best, to be run on a substantial sequence of nodes. Hence, NodesMAB can only discover this best heuristic after many calls to the MAB. On the contrary, RestartsMAB lets the heuristic act during a whole run. A related issue, with NodesMAB, is the way the rewards are computed. Each time an arm is rewarded for its performance at a given node ν, this reward depends on the choices made by all heuristics used in the sub-tree rooted at ν and not only on the pure performance of the selected arm.

Another drawback it that the UCB variant of NodesMAB normalizes the number of visited nodes by the maximum number of visited nodes for any node (considered as a root) since the beginning: the smaller the value, the better the reward. The problem with such a normalization is that the nodes close to the top of the search tree will generally have worse rewards than the nodes close to the bottom of the search tree. Hence, the reward function will not be sufficiently informative so as to discriminate the good arms. If we consider the case where some good heuristics make some good choices during the first decisions and a bad heuristic makes the next decisions, then the bad heuristic will receive an undeserved reward. In general, any heuristic used at a deep level will receive an improved reward. If depth was taken into account, maybe this problem could be eliminated (e.g., the model of [6] uses a MAB per level instead of a single one to overcome this drawback).

Some difficulty of convergence to the best heuristics by model NodesMAB is demonstrated in Table 5 that shows the distribution (in percentages) of arm calls for RestartsMAB^{UCB} and NodesMAB^{UCB} for subsets of instances of increasing difficulty. Again, RestartsMAB^{UCB} identifies the best arms, CHS and wdeg$^{ca.cd}$, in a rather satisfactory way. Selection of these arms tends to increase when the instances become harder. We must bear in mind that we have selected only competitive heuristics, not a mixture of good and bad ones that would make the discrimination easier. On the other hand, NodesMAB^{UCB} is totally disarranged and even for the hardest instances, where learning should be more effective, it does not converge to the best arms.



(a) Solving times for 12 `SocialGolfers` instances (b) Solving times for 12 `LatinSquare` instances (c) Solving times for 6 `KnightTour` instances

Figure 3: Solving efficiency (and distribution of arm calls) of classical heuristics and bandit variants on three structured problems.

Table 5: Distribution (percentage) of arm calls for bandit variants `RestartsMAB`$^{\text{UCB}}$ and `NodesMAB`$^{\text{UCB}}$ on (subsets of instances of increasing difficulty on) the [XCSP-ALL] benchmark.

		activity	impact	dom/ddeg	CHS	wdeg$^{\text{ca.cd}}$
> 0s	RestartsMAB$^{\text{UCB}}$	20.00%	18.95%	16.07%	21.58%	23.40%
	NodesMAB$^{\text{UCB}}$	20.08%	33.16%	18.44%	13.90%	14.41%
> 100s	RestartsMAB$^{\text{UCB}}$	19.63%	19.03%	15.53%	21.58%	24.22%
	NodesMAB$^{\text{UCB}}$	21.10%	31.79%	18.18%	13.91%	15.02%
> 250s	RestartsMAB$^{\text{UCB}}$	19.56%	19.12%	15.16%	21.74%	24.41%
	NodesMAB$^{\text{UCB}}$	21.77%	30.21%	18.39%	14.32%	15.31%
> 500s	RestartsMAB$^{\text{UCB}}$	19.98%	19.59%	15.02%	21.04%	24.37%
	NodesMAB$^{\text{UCB}}$	22.91%	27.81%	20.02%	13.41%	15.85%
> 1000s	RestartsMAB$^{\text{UCB}}$	19.99%	19.66%	14.50%	21.13%	24.72%
	NodesMAB$^{\text{UCB}}$	23.26%	26.52%	20.20%	13.81%	16.21%

Benchmarks. The benchmark used in [41] comes from an old competition that looks to be slightly outdated (XCSP'09). If we observe the experimental results, we can see that each heuristic, used as an arm, solves more or less the same number of instances. So, one can mainly focus on solving times and not on the number of solved instances.

Arms. Another difference between the two bandit models comes from the pool of heuristics. It is known that there is large gap between some heuristics of the literature, which contributes to a vast disparity in solver efficiency [17, 40]. In our work, we have chosen the most efficient adaptive heuristics while discarding old static and dynamic ones (e.g., `dom/ddeg`). In Figure 4, we can visualize the solving timeline per heuristic. Most of the instances are solved during the first few seconds. When time increases, there are few instances left. We can see the efficiency of the methods (here, classical heuristics) as a logarithmic distribution over time. This means that even a multi-armed bandit associated with a uniform choice of heuristics (i.e., no learning) will easily solve most of the instances. In addition, if the easy instances are not the same for each heuristic, the uniform bandit will perform better than monolithic heuristics. The reason is that when an instance is easy for heuristic A and hard for heuristic B, the uniform bandit will solve it when calling A. If another instance is easy for B and not for A, it will solve it when calling B. This is why for certain problems, a MAB algorithm can be more efficient than any heuristic alone (e.g., `RestartsMAB`$^{\text{UCB}}$ for `LatinSquare` in Figure 3b).

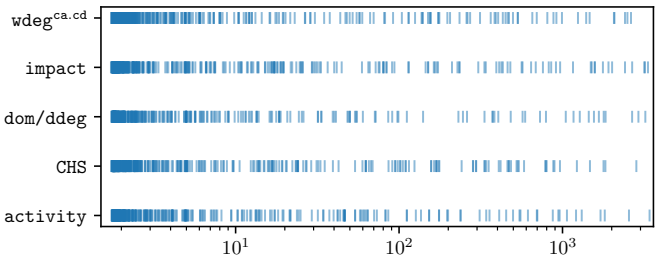


Figure 4: Distribution of solved instances over time (x-axis) for classical heuristics on the [XCSP17] benchmark.

7 CONCLUSION

In this work. we have proposed a new framework for automatically setting the variable ordering heuristic option of a CSP solver. Our framework uses a multi-armed bandit algorithm in order to learn the best heuristic to be used for any given instance to be solved. We propose an original and easy to embed MAB model, which exploits the restart mechanism of the solver in order to provide some feedback to the learning algorithm. We have conducted an experimental study showing that the proposed framework is more efficient than the state-of-the-art bandit model, but also more efficient than any single heuristic tried alone. In the future, we aim at extending our framework in order to make it, apart from autonomous, adaptive during each run.

ACKNOWLEDGEMENTS

This work has been supported by the project CPER Data from the region "Hauts-de-France".

REFERENCES

[1] S. Agrawal and N. Goyal, 'Near-optimal regret bounds for Thompson Sampling', *J. ACM*, **64**(5), 30:1–30:24, (2017).
[2] R. Amadini, M. Gabbrielli, and J. Mauro, 'Portfolio approaches for constraint optimization problems', *Ann. Math. Artif. Intell.*, **76**(1-2), 229–246, (2016).
[3] K.R. Apt, *Principles of Constraint Programming*, Cambridge University Press, 2003.

[4] P. Auer, N. Cesa-Bianchi, and P. Fischer, 'Finite-time analysis of the multiarmed bandit problem', *Machine Learning*, **47**(2), 235–256, (May 2002).
[5] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. Schapire, 'The nonstochastic multiarmed bandit problem', *SIAM Journal on Computing*, **32**(1), 48–77, (2002).
[6] A. Balafrej, C. Bessiere, and A. Paparrizou, 'Multi-armed bandits for adaptive constraint propagation', in *Proceedings of IJCAI'15*, pp. 290–296, (2015).
[7] M-F. Balcan, T. Dick, T. Sandholm, and E. Vitercik, 'Learning to branch', in *Proceedings of ICML'18*, pp. 353–362, (2018).
[8] C. Bessiere, B. Zanuttini, and C. Fernandez, 'Measuring search trees', in *Proceedings of ECAI'04 workshop on Modelling and Solving Problems with Constraints*, pp. 31–40, (2004).
[9] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, 'Boosting systematic search by weighting constraints', in *Proceedings of ECAI'04*, pp. 146–150, (2004).
[10] F. Boussemart, C. Lecoutre, G. Audemard, and C. Piette, 'XCSP3: an integrated format for benchmarking combinatorial constrained problems', *CoRR*, **abs/1611.03398**, (2016).
[11] S. Bubeck and N. Cesa-Bianchi, *Regret analysis of stochastic and nonstochastic multi-armed bandit problems*, Foundations and Trends in Machine Learning, Now Publishers, 2012.
[12] R. Dechter, *Constraint processing*, Morgan Kaufmann, 2003.
[13] S. Epstein and S. Petrovic, 'Learning to solve constraint problems', in *ICAPS-07 Workshop on Planning and Learning*, (2007).
[14] M. Gagliolo and J. Schmidhuber, 'Learning restart strategies', in *Proceedings of IJCAI'07*, pp. 792–797, (2007).
[15] C. Gomes, B. Selman, N. Crato, and H. Kautz, 'Heavy-tailed phenomena in satisfiability and constraint satisfaction problems', *Journal of Automated Reasoning*, **24**(1), 67–100, (2000).
[16] C. Gomes, B. Selman, and H.A. Kautz, 'Boosting combinatorial search through randomization.', in *Proceedings of AAAI'98*, pp. 431–437, (1998).
[17] D. Habet and C. Terrioux, 'Conflict history based branching heuristic for CSP solving', in *Proceedings of CIMA@ICTAI*, pp. 70–80, (2018).
[18] Y. Hamadi, E. Monfroy, and F. Saubion, *Autonomous Search*, Springer, 2014.
[19] H. H. Hoos, R. Kaminski, M. T. Lindauer, and T. Schaub, 'aspeed: Solver scheduling via answer set programming', *TPLP*, **15**, 117–142, (2015).
[20] H. Hurley, L. Kotthoff, Y. Malitsky, and B. O'Sullivan, 'Proteus: A hierarchical portfolio of solvers and transformations', in *Proceedings of CPAIOR'2014*, pp. 301–317, (2014).
[21] C. Lecoutre, *Constraint networks: techniques and algorithms*, ISTE/Wiley, 2009.
[22] C. Lecoutre, L. Sais, S. Tabary, and V. Vidal, 'Recording and minimizing nogoods from restarts', *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, **1**, 147–167, (2007).
[23] J. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, 'Learning rate based branching heuristic for SAT solvers', in *Proceedings of SAT*, pp. 13–140, (2016).
[24] J. Liang, H. Govind, P. Poupart, K. Czarnecki, and V. Ganesh, 'An empirical study of branching heuristics through the lens of global learning rate', in *Proceedings of SAT*, pp. 119–135, (2017).
[25] M. Lindauer, H. H. Hoos, K. Leyton-Brown, and T. Schaub, 'Automatic construction of parallel portfolios via algorithm configuration', *Artif. Intell.*, **244**, 272–290, (2017).
[26] M. Loth, M. Sebag, Y. Hamadi, and M. Schoenauer, 'Bandit-based search for constraint programming', in *Proceedings of CP'13*, pp. 464–480, (2013).
[27] M. Luby, A. Sinclair, and D. Zuckerman, 'Optimal speedup of Las Vegas algorithms', *Information Processing Letters*, **47**(4), 173–180, (1993).
[28] A. K. Mackworth, 'Consistency in networks of relations', *Artif. Intell.*, **8**(1), 99–118, (February 1977).
[29] M. Maratea, L. Pulina, and F. Ricca, 'Multi-engine ASP solving with policy adaptation', *J. Log. Comput.*, **25**, 1285–1306, (2015).
[30] L. Michel and P. Van Hentenryck, 'Activity-based search for black-box constraint programming solvers', in *Proceedings of CPAIOR'12*, pp. 228–243, (2012).
[31] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan, 'Using case-based reasoning in an algorithm portfolio for constraint solving', in *Proceedings of AICS'10*, (2008).
[32] L. Pulina and A. Tacchella, 'A multi-engine solver for quantified boolean formulas', in *Proceedings of CP'07*, pp. 574–589, (2007).
[33] P. Refalo, 'Impact-based search strategies for constraint programming', in *Proceedings of CP'04*, pp. 557–571, (2004).
[34] *Handbook of Constraint Programming*, eds., F. Rossi, P. van Beek, and T. Walsh, Elsevier, 2006.
[35] D. Russo and B. Van Roy, 'An information-theoretic analysis of Thompson Sampling', *J. Mach. Learn. Res.*, **17**, 68:1–68:30, (2016).
[36] D. J. Russo, B. Van Roy, A. Kazerouni, I. Osband, and Z. Wen, 'A tutorial on thompson sampling', *Found. Trends Mach. Learn.*, **11**(1), 1–96, (July 2018).
[37] D. Sabin and E.C. Freuder, 'Contradicting conventional wisdom in constraint satisfaction', in *Proceedings of CP'94*, pp. 10–20, (1994).
[38] B. Smith and S. Grant, 'Trying harder to fail first', in *Proceedings of ECAI'98*, pp. 249–253, Brighton, UK, (1998).
[39] 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).
[40] H. Wattez, F. Koriche, C. Lecoutre, A. Paparrizou, and S. Tabary, 'Refining constraint weighting', in *Proceedings of ICTAI'19*, pp. 71–77, (2019).
[41] W. Xia and R. H. C. Yap, 'Learning robust search strategies using a bandit-based approach', in *Proceedings of AAAI'18*, pp. 6657–6665, (2018).
[42] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, 'Satzilla: Portfolio-based algorithm selection for SAT', *J. Artif. Intell. Res. (JAIR)*, **32**, 565–606, (2008).