Conflict History Based Search for Constraint Satisfaction Problem*

Djamal Habet, Cyril Terrioux
Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France
{djamal.habet,cyril.terrioux}@lis-lab.fr

Abstract

Branching heuristic is an important module in algorithms dedicated to solve Constraint Satisfaction Problems (CSP). It impacts the efficiency of exploring the search space and exploiting the problem structure. In this paper, we propose Conflict-History Search (CHS), a dynamic and adaptive branching heuristic for CSP solving. It is based on the search failures and considers the temporality of these failures throughout the resolution process.

The exponential recency weighted average is used to estimate the evolution of the hardness of constraints throughout the search. The experimental evaluation on XCSP3 instances shows that integrating CHS to solvers based on MAC obtains competitive results and can improve those obtained by other heuristics of the state of the art, such as *dom/wdeg* and ABS.

Keywords. CSP, Branching Heuristic, Conflict Based Search, Search History, Exponential Recency Weighted Average

DOI. https://doi.org/10.1145/3297280.3297389

1 Introduction

The Constraint Satisfaction Problem (CSP) is a powerful framework to model and efficiently solve problems that occur in various fields, both academic and industrial [21]. A CSP instance is defined on a set of variables which must be assigned in their respective finite domains by satisfying a set of constraints which express restrictions between different assignments. A solution is an assignment of each variable which satisfies all constraints.

CSP solving, based on search tree algorithms, has made significant progress in recent years thanks to research on several aspects which receive considerable efforts such as global constraints, filtering techniques, learning and restarts. An important component in CSP solvers is the variable branching rule. Indeed, the corresponding heuristics define, statically or dynamically, the order in which the variables will be assigned and thus the way that the search space will be explored and the size of the search tree.

Many heuristics have been proposed (e.g. [1, 2, 3, 4, 6, 7, 18, 20, 9]) and aim to satisfy the famous *first-fail principle* [8] which advises "to succeed, try first where you are likely to fail". Nowadays, the most efficient heuristics are adaptive and dynamic [3, 6, 18, 20, 9]. Indeed, the order of branchings is defined according to the collected information since the beginning of the search. For instance, some heuristics consider the effect of filtering when decisions and propagations are applied [18, 20]. Defined since 2004, the *dom/wdeg* heuristic remains one of the simplest, the most popular and efficient one. It is based on the hardness of constraints to reflect how often a constraint fails. It uses a weighting process to focus on the variables appearing in constraints with high weights which are assumed to be hard to satisfy [3].

In this paper, we propose *Conflict-History Search (CHS)*, a new dynamic and adaptive branching heuristic for CSP solving. It is based on the history of search failures which happen as soon as a domain of a variable is emptied after constraint propagations. The goal is to reward the scores of constraints that have recently been involved in conflicts and therefore to favor the variables appearing in these constraints.

The scores of constraints are estimated on the basis of the exponential recency weighted average technique which comes from reinforcement learning [24]. It was also recently used in defining powerful branching heuristics for solving the satisfiability problem (SAT) [15, 16]. We have integrated CHS in solvers based on MAC (Maintaining Arc Consistency) [22] and BTD (Backtracking with Tree Decomposition) [12]. The empirical evaluation on XCSP3 instances (XCSP3, for XML-CSP version 3, is an XML-based format to represent instances of combinatorial constrained problems) shows that CHS is competitive and brings improvements to the heuristics of the state of the art.

*This work has been funded by the french Agence Nationale de la Recherche, reference ANR-16-C40-0028.

The paper is organized as follows. Section 2 includes some necessary definitions and notations. Section 3 describes related work on branching heuristics for CSP and SAT. Section 4 presents and details our contribution which is evaluated experimentally in Section 5. Finally, we conclude and give future work.

2 Preliminaries

We give some definitions including CSP and Exponential Recency Weighted Average (ERWA).

2.1 Constraint Satisfaction Problem

An instance of a Constraint Satisfaction Problem (CSP) is given by a triple (X, D, C), such that:

- $X = \{x_1, \cdots, x_n\}$ is a set of n variables,
- $D = \{D_1, ..., D_n\}$ is a set of finite domains, and
- $C = \{c_1, \cdots, c_e\}$ is a set of e constraints.

Each constraint c_i is defined by $S(c_i)$ and $R(c_i)$, where $S(c_i) = \{x_{i_1}, \cdots, x_{i_k}\} \subseteq X$ defines the scope of c_i and $R(c_i) \subseteq D_{i_1} \times \cdots \times D_{i_k}$ is its compatibility relation. The constraint satisfaction problem asks for an assignment of a value from D_i to each variable x_i of X that satisfies each constraint in C. Checking whether a CSP instance has a solution (i.e. a consistent assignment of X) is NP-complete.

2.2 Exponential Recency Weighted Average

Given a time series of m numbers $y = (y_1, y_2, \cdots, y_m)$, the simple average of y is $\sum_{i=1}^{m} \frac{1}{m} y_i$ where each y_i has the same weight $\frac{1}{m}$. However, recent data may be more pertinent than the older ones to characterize the current situation. The Exponential Recency Weighted Average (ERWA) [24] takes into account such considerations by giving to the recent data higher weights than the older ones. In fact, the *exponential moving average* $\bar{y}_m$ is computed by: $\bar{y}_m = \sum_{i=1}^{m} \alpha.(1-\alpha)^{m-i}.y_i$, where $0 < \alpha < 1$ is a step-size parameter which controls the relative weights between recent and past data. The moving average can also be calculated incrementally by the formula:

$$\bar{y}_{m+1} = (1-\alpha).\bar{y}_m + \alpha.y_{m+1}.$$

ERWA was used to solve the bandit problem to estimate the expected reward of different actions in non-stationary environments [24]. In bandit problems, there is a set of actions and the agent must select the action to play in order to maximize its long term expected reward.

3 Related Work

We present the most efficient branching heuristics for CSP and SAT. The recalled heuristics share the same behavior. Indeed, the variables and/or constraints are weighted dynamically throughout the search by considering the collected information since the beginning of the search. Also, some heuristics smooth (or decay) these weights as it will be explained further.

3.1 Impact-Based Search (IBS)

This heuristic selects the variable which leads to the largest search space reduction [20]. This impact on the search space size is approximated as the reduction of the product of the variable domain sizes. Formally, the impact of assigning the variable x_i to the value $v_i \in D_i$ is defined by :

$$I(x_i = v_i) = 1 - \frac{P_{after}}{P_{before}}$$

P_{after} and P_{before} are respectively the products of the domain cardinalities after and before branching on $x_i = v_i$ and applying constraint propagations.

3.2 Conflict-Driven Heuristic

A popular branching heuristic for CSP solving is *dom/wdeg* [3]. It guides the search towards the variables appearing in the constraints which seem to be hard to satisfy. For each constraint c_j, the *dom/wdeg* heuristic maintains a weight $w(c_j)$ (initially set to 1) counting the number of times that c_j has led to a failure (i.e. the domain of a variable x_i in $S(c_j)$ is emptied during propagation from c_j). The weighted degree of a variable x_i is defined as:

$$wdeg(x_i) = \sum_{c_j \in C \mid x_i \in S(c_j) \wedge |Uvars(c_j)|>1} w(c_j)$$

with $Uvars(c_j)$ the set of unassigned variables in $S(c_j)$. The *dom/wdeg* heuristic selects the variable x_i to branch on with the smallest ratio $|D_i|/wdeg(x_i)$, such that D_i is the current domain of x_i (potentially, the size of D_i may be reduced by the propagation process in the current step of the search). The constraint weights are not smoothed in *dom/wdeg*. Variants of *dom/wdeg* were introduced (for example, see [9]).

3.3 Activity-Based Heuristic (ABS)

This heuristic is motivated by the prominent role of filtering techniques in CSP solving [18]. It exploits this filtering information and maintains measures of how often the variable domains are reduced during the search. Indeed, at each node of the search tree, constraint propagation may filter the domains of some variables after the decision has been made. Let X_f be the set of such variables. Accordingly, the activities $A(x_i)$ (initially set to 0) of the variables $x_i \in X$ are updated as follows:

- $A(x_i) = A(x_i) + 1$ if $x_i \in X_f$ and
- $A(x_i) = \gamma \times A(x_i)$ if $x_i \notin X_f$.

γ is a decay parameter, such that $0 \leq \gamma \leq 1$. The ABS heuristic selects the variable x_i with the highest ratio $A(x_i)/|D_i|$.

3.4 Branching Heuristics for SAT

In the context of the satisfiability problem (SAT), modern solvers based on Conflict-Driven Clause Learning (CDCL) [5, 17, 19] employ variable branching heuristics correlated to the ability of the variable to participate in producing learnt clauses when conflicts arise (a conflict is a clause falsification). The Variable State Independent Decaying Sum (VSIDS) heuristic [19] maintains an activity value for each Boolean variable. The activities are modified by two operations: the bump (increase the activity of variables appearing in the process of generating a new learnt clause when a conflict is analyzed) and the multiplicative decay of the activities (often applied at each conflict). VSIDS selects the variable with the highest activity to branch on.

Recently, a conflict history based branching heuristic (CHB) [15], based on the exponential recency weighted average, was introduced. It rewards the activities to favor the variables that were recently assigned by decision or propagation. The rewards are higher if a conflict is discovered. The Learning Rate Branching (LRB) heuristic [16] extends CHB by exploiting locality and introducing the learning rate of the variables.

Dedicated to constraint programming, Gecode solver implements CHB since version 5.1.0 released in April 2017 [23]. Indeed, in this version of Gecode, the variables are weighted in the same manner as in the SAT context following ERWA [15, 16].

As we will describe it in the next sections, the branching heuristic that we propose (CHS) uses ERWA to weight the constraints and not the variables. The constraint weights are then used in a next phase to calculate the variable scores which are used to select the branching variable.

4 Conflict-History Search for CSP

Inspired by the CHB heuristic for SAT, we define a new branching heuristic for CSP solving which we call *Conflict-History Search* (CHS). The central idea is to consider the history of constraint failures and favor the variables that often appear in recent failures.

So, the conflicts are dated and the constraints are weighted on the basis of the exponential recency weighted average. These weights are coupled to the variable domains to calculate the Conflict-History scores of the variables.

4.1 CHS Description

Formally, CHS maintains for each constraint c_j a score $q(c_j)$ which is initialized to 0 at the beginning of the search. If c_j leads to a failure during the search because the domain of a variable in $S(c_j)$ is emptied by propagation then $q(c_j)$ is updated by the formula below derived from ERWA:

$$q(c_j) = (1 - \alpha) \times q(c_j) + \alpha \times r(c_j)$$

The parameter $0 < \alpha < 1$ is the step-size and $r(c_j)$ is the reward value. It defines the importance given to the old value of q at the expense of the reward r. The value of α decreases over time as it is applied in ERWA [24].

Indeed, starting from its initial value α_0, α decreases by 10^{-6} at each constraint failure to a minimum of 0.06. Decreasing the α value amounts to giving more importance to the last value of q and considering that the values of q are more and more relevant as the search progresses.

The reward value $r(c_j)$ is based on how recently c_j occurred in conflicts. The goal is to give a higher reward to constraints that fail regularly over short periods of time during the search space exploration. The reward value is calculated according to the formula:

$$r(c_j) = \frac{1}{\#Conflicts - Conflict(c_j) + 1}$$

Initialized to 0, $\#Conflicts$ is the number of conflicts which have occurred since the beginning of the search. Also initialized to 0 for each constraint $c_j \in C$, $Conflict(c_j)$ stores the last $\#Conflicts$ value where c_j led to a failure. Once $r(c_i)$ and $q(c_i)$ are updated, $\#Conflicts$ is incremented by 1.

At this stage, we are able to define the Conflict-History score of the variables $x_i \in X$, which will be used in selecting the branching variable as follows:

$$chv(x_i) = \frac{\sum_{c_j \in C \mid x_i \in S(c_j) \wedge |Uvars(c_j)|>1} q(c_j)}{|D_i|}$$

CHS keeps the variable to branch on with the highest chv value. In this manner, CHS focuses branching on the variables with small size of domain while belonging to constraints which appear recently and repetitively in conflicts.

One can observe that at the beginning of the search, all the variables have the same score equal to 0. To avoid random selection of the branching variable, we reformulate the calculation of chv as given below, where δ is a positive real number close to 0.

$$chv(x_i) = \frac{\sum_{c_j \in C \mid x_i \in S(c_j) \wedge |Uvars(c_j)|>1} (q(c_j) + \delta)}{|D_i|}$$

Thus, at the beginning of the search, the branching will be oriented according to the degree of the variables without having a negative influence on the ERWA-based calculation later in the search.

4.2 CHS and Restarts

Nowadays, restart techniques are important for the efficiency of solving algorithms (see for example [13]). Restarts may allow to reduce the impact of irrelevant choices done during search according to heuristics such as variable selection.

As it will be detailed in the next section, CHS is integrated into CSP solving algorithms which include restarts. In the corresponding implementations, the $Conflict(c_j)$ value of each constraint c_j is not reinitialized when a restart occurs. It is the same for $q(c_j)$ (however, a *smoothing* may be applied and will be explained later). Keeping this information unchanged reinforces learning from the search history.

Concerning the step-size α, which defines the importance given to the old value of $q(c_j)$ at the expense of the reward $r(c_j)$, CHS reinitializes the step-size value α to α_0 at each restart. This may guide the search through different parts of the search space.

4.3 CHS and Smoothing

At each conflict and as in the $dom/wdeg$ heuristic, CHS updates the chv score of one constraint at a time: the constraint c_j which is used to wipe out the domain of a variable in $S(c_j)$. As long as they do not appear in new conflicts, some constraints can have their weights unchanged for several search steps. These constraints may have high scores while their importance does not seem high for the current part of the search. To avoid this situation, we propose to smooth the scores $q(c_j)$ of all the constraints $c_j \in C$ at each restart by the following formula:

$$q(c_j) = q(c_j) \times 0.995^{\#Conflicts - Conflict(c_j)}$$

Hence, the scores of constraints are decayed according to the date of their last appearances in conflicts. Decaying is also used in other heuristics such as ABS [18] for CSP and VSIDS [19] for SAT. However, it is applied to the score of the variables and not that of the constraints (or clauses).

5 Experimental Evaluation

5.1 Experimental Protocol

We consider 10,785 instances from the XCSP3 repository[1], including notably structured instances and discarding fully random instances. This latter restriction is quite natural since adaptive heuristics aim to exploit the underlying structure of the instances to solve.

Regarding the solving step, we exploit MAC with restarts [14] before assessing the impact of our approach on a structural solving method, namely BTD-MAC+RST+Merge [10]. MAC uses a geometric restart strategy based on the number of backtracks with an initial cutoff set to 100 and an increasing factor set to 1.1. In order to make the comparison fair, the lexicographic ordering is used for the choice of the next value to assign.

We have written our own code to implement all the compared branching heuristics in this section ($dom/wdeg$, ABS, CHS, CHB, ABS and $dom/wdeg$), as well as the solvers that exploit them (MAC and BTD). All the algorithms are written in C++.

The experiments are performed on Dell PowerEdge M610 blade servers with Intel Xeon E5620 processors under Ubuntu 18.04. Each solving process is allocated a slot of 30 minutes and at most 12 GB of memory per instance. In the following tables, #solv denotes the number of solved instances by a given solver and time is the cumulative runtime.

5.2 Impact of CHS Settings

In this part, we assess the sensitivity of CHS with respect to the chosen values for α or δ. First, we fix δ to 10^{-4} (to start the search by considering the variable degrees then quickly exploit ERWA-based computation) and vary the value of α_0 between 0.1 and 0.9 with a step of 0.1. Figure 1 presents the number of instances solved

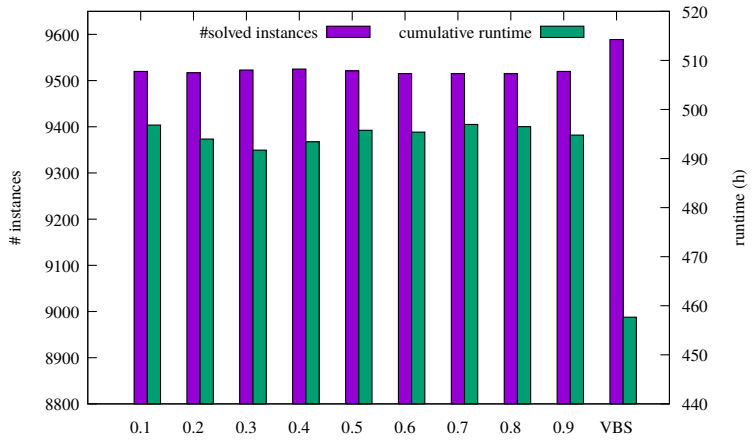


Figure 1: Number of instances solved by MAC+CHS depending on the initial value of α and cumulative runtime in hours for all the instances.

by MAC depending on the value of α and the corresponding cumulative runtime. We also provide the results of the Virtual Best Solver (VBS) which counts the number of the instances solved (and the cumulative running time) at least one time when varying the value of α.

We observe that the value $\alpha_0 = 0.4$ allows MAC to solve more instances (9,525 solved instances with a cumulative solving time of 493 hours) than the other considered values. The worst case is $\alpha_0 = 0.7$ with 9,515 solved instances in 496 hours. This shows the robustness of CHS w.r.t. the α parameter.

Regarding the Virtual Best Solver (VBS), we note that it can solve 64 additional instances than MAC+CHS when $\alpha_0 = 0.4$. Among these instances, some of them seem to be hard. Indeed, often, only one of the checked values of α allows MAC to solve them and the required runtime generally exceeds several minutes. Therefore, a finer adjustment of the value of α_0 or its adaptation to the treated instance would allow MAC+CHS to perform even better.

Now, we set α_0 to 0.4 and evaluate different values of δ. From Table 1, the observations are similar to those presented previously, showing the robustness of CHS regarding δ.

Also, it is interesting to highlight that MAC+CHS with $\delta = 0$ solves 9,517 instances while it solves 9,525 instances if $\delta = 10^{-4}$. This illustrates the relevance of introducing δ in CHS while it allows to solve 8 more

[1] http://www.xcsp.org/series

Table 1: Impact of δ value on MAC+CHS regarding the number of solves instances and the cumulative runtime in hours

δ	#solv.	time (h)
0	9,517	498.17
10^{-5}	9,520	494.07
10^{-4}	**9,525**	**493.41**
10^{-3}	9,524	493.91

instances with this last setting.

Finally, Table 2 gives the results of MAC+CHS ($\alpha_0 = 0.4$, $\delta = 10^{-4}$) with smoothing (+s) or not (-s) the constraint scores and/or with resetting (+r) or not (-r) the value of α to 0.4 at each new restart.

Table 2: Number of instances solved by MAC with CHS with/without smoothing and reset of α and cumulative runtime in hours

Solver	#solv.	time (h)
MAC+CHS (+s+r)	**9,525**	**493.41**
MAC+CHS+s-r	9,509	498.80
MAC+CHS-s-r	9,482	514.73
MAC+CHS-s+r	9,478	518.05

The observed behaviors clearly support the importance of smoothing and restarts for CHS. For example, MAC+CHS+s-r solves 16 less instances than MAC+CHS, while MAC+CHS-s+r solves 47 instances less.

5.3 CHS vs. Other Search Heuristics

Now, we compare CHS (the settings are : $\alpha_0 = 0.4$, $\delta = 10^{-4}$) to other search strategies: *dom/wdeg*, ABS and CHB as implemented in Gecode. For ABS, we fix the decay parameter γ to 0.999 as in [18]. For CHB, we use the value parameters as given in [23]. We add a variant *dom/wdeg*+s which is *dom/wdeg* but the weights of constraints are smoothed exactly as in CHS.

Table 3: Number of instances solved by MAC with **dom/wdeg**, **dom/wdeg + s**, ABS, CHB and CHS and cumulative runtime in hours

Solver	#solv.	time (h)
MAC+CHS	**9,525**	**493.41**
MAC+*dom/wdeg*	9,501	507.17
MAC+*dom/wdeg*+s	9,500	505.13
MAC+ABS	9,476	515.17
MAC+CHB	9,458	525.38

¿From Table 3, it is clear that MAC with CHS performs better than with the other heuristics. Indeed, it solves 24 instances more than MAC+*dom/wdeg*, 49 instances more than MAC+ABS and 67 instances more than MAC+CHB. Interestingly, whatever the value of α_0, MAC with CHS remains better than all its competitors. Indeed, the worst case is when $\alpha_0 = 0.7$ where MAC+CHS solves 9,515 instances. Moreover, the results obtained by MAC+CHB show that the calculation of weights by ERWA on the constraints (as done in CHS) is more relevant than its calculation on the variables (as done in CHB). Furthermore, the smoothing phase introduced in *dom/wdeg* allows MAC+*dom/wdeg*+s to reduce slightly the computation time when compared to MAC+*dom/wdeg*, while solving one less instance.

dom/wdeg and ABS are two powerful and popular branching heuristics. In particular, *dom/wdeg* is widely used in the literature and integrated in many solvers. A careful reading of the results of Table 3 shows that *dom/wdeg* solves 25 instances more than ABS. This remark is to highlight the improvement brought by our heuristic CHS. Indeed, it allows MAC to solve 24 additional instances compared to *dom/wdeg*.

Finally, Table 4 provides the results of MAC variants on some instance families chosen from a representative panel of the used benchmark in order to show the different trends we observed. First, we note that no heuristic is

Table 4: Number of instances solved by MAC with **dom/wdeg**, ABS, CHB and CHS and cumulative runtime in seconds for some instance families.

Family			*dom/wdeg*		ABS		CHB		CHS	
Origin	Name	#inst.	#solv.	time (s)	#solv.	time (s)	#solv.	time (s)	#solv.	time (s)
Academic	AllInterval-m1-s1	32	25	15,406	**32**	9	**32**	9	**32**	9
	Blackhole-xcsp2-s04	10	10	5.51	10	4.82	10	5.15	10	5.46
	Dubois-m1-s1	30	10	38,249	**16**	28,639	10	38,111	11	37,234
	GracefulGraph-m1-s1	104	17	160,007	16	160,090	16	160,355	**18**	155,667
	Kakuro-sumdiff-hard	187	**187**	285	185	4,216	180	15,044	**187**	811
	Nonogram-table-s1	176	167	1,991	**168**	34.56	**168**	35.19	**168**	331.67
	PigeonsPlus-m1-s1	38	**37**	4,860	29	20,120	**37**	5,192	**37**	4,878
	Sat-xcsp2-bmc	24	**24**	1,816	**24**	518	20	51	**24**	4,708
	Subisomorphism-m1-LV	1,176	1,100	151,661	1,108	136,868	1,101	147,664	**1,109**	134,787
	SuperSolutions-Taillard-os05	30	23	14,102	19	20,036	**26**	11,125	21	16,386
	TravellingSalesman-xcsp2-s20a4	15	15	64.29	15	60.77	15	311.45	15	116.59
Real-world	OpenStacks	76	40	4,663	40	6,342	**41**	5,757	**41**	5,007
	RenaultMod-m1-s1	50	50	1.70	50	0.61	50	0.99	50	0.52
	SocialGolfers-xcsp2-s1	12	4	14,576	4	16,300	5	14,030	**6**	11,404

always better than the others. However, if we sort the heuristics with respect to the number of solved instances per family, CHS is ranked at the first place for 88% of the 141 considered families, by performing better or similarly than the two other heuristics. This percentage exceeds respectively 93% and 99% if we consider the first two places or the first three places. Hence, CHS is clearly competitive.

Also, one might think that *dom/wdeg* performs worse than ABS and CHB. This impression is explained by the fact that, when MAC+*dom/wdeg* is better on a given family, it solves only few additional instances. In contrast, when it is outperformed, this is done by several additional solved instances. Finally, if we compare the results on the instances labeled real-world in the XCSP3 repository, we observe that MAC with CHS solves more instances and performs faster, between 10% and 30%, than any other combination.

5.4 CHS and Tree-Decomposition

We now assess the behavior of CHS when the search is guided by a tree-decomposition. Studying this question is quite natural since CHS aims at exploiting the structure of the instance, but in a way different from what the tree-decomposition does. With this aim in view, we consider BTD-MAC+RST+Merge [10]. The parameters of BTD-MAC+RST+Merge are set like in [11] except the variable heuristic which can be one of the two best heuristics considered previously, namely *dom/wdeg* or CHS.

Like for MAC, the solving is more efficient with CHS than with *dom/wdeg*. Indeed, BTD-MAC+RST+Merge with CHS solves 9,525 instances (in 485 h) against 9,495 instances (in 501 h) for *dom/wdeg*: 30 additional instances are solved. This observation shows that exploiting both CHS and tree-decomposition may be of interest and that these two strategies can be complementary.

6 Conclusion

We have proposed CHS, a new branching heuristic for CSP based on the search history and designed following techniques coming from reinforcement learning. The experimental results confirm the relevance of CHS which is competitive with the powerful heuristics *dom/wdeg* and ABS, when implemented in solvers based on MAC or tree-decomposition exploitation.

The experimental study suggests that the α parameter value could be refined. We will explore the possibility of defining its value depending on the instance to be solved. For example, we will look for probing techniques to fix the appropriate value of α. Furthermore, similarly to the ABS heuristic, we will also consider to include information provided by filtering operations in CHS. Finally, we will extend CHS to deal with constraint optimization problems.

References

[1] Christian Bessière, Assef Chmeiss, and Lakhdar Saïs. Neighborhood-based variable ordering heuristics for the constraint satisfaction problem. In *CP*, pages 565–569, 2001.

[2] Christian Bessière and Jean-Charles Régin. MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems. In *CP*, pages 61–75, 1996.

[3] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Saïs. Boosting systematic search by weighting constraints. In *ECAI*, pages 146–150, 2004.

[4] Daniel Brélaz. New Methods to Color Vertices of a Graph. *Communications of the ACM*, 22(4):251–256, 1979.

[5] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *SAT*, pages 502–518, 2003.

[6] Pieter A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *ECAI*, pages 31–35, 1992.

[7] Solomon W. Golomb and Leonard D. Baumert. Backtrack programming. *Journal of the ACM*, 12:516–524, 1965.

[8] Robert M. Haralick and Gordon L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *AIJ*, 14:263–313, 1980.

[9] Emmanuel Hebrard and Mohamed Siala. Explanation-based weighted degree. In *CPAIOR*, pages 167–175, 2017.

[10] Philippe Jégou, Hanan Kanso, and Cyril Terrioux. Towards a Dynamic Decomposition of CSPs with Separators of Bounded Size. In *CP*, pages 298–315, 2016.

[11] Philippe Jégou, Hanan Kanso, and Cyril Terrioux. BTD and miniBTD. In *XCSP3 Competition*, 2017.

[12] Philippe Jégou and Cyril Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *AIJ*, 146:43–75, 2003.

[13] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Nogood recording from restarts. In *IJCAI*, pages 131–136, 2007.

[14] Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal. Recording and Minimizing Nogoods from Restarts. *JSAT*, 1(3-4):147–167, 2007.

[15] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Exponential Recency Weighted Average Branching Heuristic for SAT Solvers. In *AAAI*, pages 3434–3440, 2016.

[16] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning Rate Based Branching Heuristic for SAT Solvers. In *SAT*, pages 123–140, 2016.

[17] João Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, August 1999.

[18] Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *CPAIOR*, pages 228–243, 2012.

[19] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, pages 530–535, 2001.

[20] Philippe Refalo. Impact-based search strategies for constraint programming. In *CP*, pages 557–571, 2004.

[21] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[22] Daniel Sabin and Eugene C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *ECAI*, pages 125–129, 1994.

[23] Christian Schulte. Programming branchers. In Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist, editors, *Modeling and Programming with Gecode*. 2018. Corresponds to Gecode 6.0.1.

[24] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.