Using SAT Encodings to Derive CSP Value Ordering Heuristics

Christophe Lecoutre, Lakhdar Sais, and Julien Vion

CRIL-CNRS FRE 2499,
Université d'Artois
Lens, France
$\{lecoutre, sais, vion\}$@cril.univ-artois.fr

Abstract. In this paper, we address the issue of value ordering heuristics in the context of a backtracking search algorithm that exploits binary branching and the adaptive variable ordering heuristic $dom/wdeg$. Our initial experimentation on random instances shows that (in this context), contrary to general belief, following the $fail\text{-}first$ policy instead of the $promise$ policy is not really penalising. Furthermore, using SAT encodings of CSP instances, a new value ordering heuristic related to the $fail\text{-}first$ policy can be naturally derived from the well-known *Jeroslow-Wang* heuristic. This heuristic, called $min\text{-}inverse$, exploits the bi-directionality of constraint supports to give a more comprehensive picture in terms of domain reduction when a given value is assigned to (resp. removed from) a given variable. An extensive experimentation on a wide range of CSP instances shows that $min\text{-}inverse$ can outperform the other known value ordering heuristics.

1 Introduction

For solving instances of the Constraint Satisfaction Problem (CSP), backtracking search algorithms are commonly used. To limit their combinatorial explosion, various improvements have been proposed (e.g. ordering heuristics, filtering techniques and conflict analysis). It is well known that the ordering used to perform search decisions has a great impact on the size of the search tree. At each stage, one needs to decide the *value* to assign to a *variable*. So far, such decisions have been performed by choosing the variable in a first step (vertical selection) and the value to assign in a second step (horizontal selection).

Many works have been devoted to the first selection step. Variable ordering heuristics that have been proposed can be conveniently classified as static (e.g. deg), dynamic (e.g. dom [15], bz [6], $dom/ddeg$ [4]) and adaptive (e.g. $dom/wdeg$ [5]). The heuristic $dom/wdeg$ has been shown superior to the other ones [5, 21, 16, 28]. However, value ordering (the second step of the decision) has clearly been considered for a long time as potentially of marginal effect to search improvements. The arguments behind this can be related to the fact that selecting a given value is computationally more difficult than selecting a given variable, particularly when one considers dynamic selection. The second reason for considering value ordering as useless is that, when facing unsatisfiable instances or when searching all solutions, one needs to consider all values for each

variable. As clearly shown by Smith and Sturdy [26], these arguments hold when search is based on d-way branching but not on 2-way branching. d-way branching means that, at each node of the search tree, a variable x is selected and d branches are considered where d is the current size of the domain of x: the ith branch corresponds to $x = a_i$ where a_i denotes the ith value of the domain of x. On the other hand, with binary (or 2-way) branching, at each node of the search tree, a pair (x,a) is selected where x is an unassigned variable and a a value in the domain of x, and two branches are considered: the first one corresponds to the assignment $x = a$ and the second one to the refutation $x \neq a$. These two schemes are not equivalent as it has been shown that binary branching is more powerful than non-binary branching [17].

Traditionally, two principles are considered during search: at each step, select the variable which is the most constrained and select then the least constrained value (e.g. $min\text{-}conflicts$ [12]). These principles respectively correspond to two policies called $fail\text{-}first$ and $promise$, and one interesting issue is the adherence assessment of heuristics to both policies [2, 29]. In this paper, we focus on value ordering heuristics, and more precisely, we try to determine if value ordering heuristics should adhere in priority to the $promise$ policy. Of course, one can be surprised that we address this issue as it is commonly admitted that it should be the case. In particular, a lot of works support the idea that a value must be chosen by estimating the number of solutions or conflicts. One has then to prefer the value that maximizes the estimated number of solutions in the remaining network [8, 13, 24, 20] or minimizes the number of conflicts with variables in the neighbourhood [12, 23].

However, we noticed that most of the experimental results are given when d-way branching and/or non adaptive variable heuristics (such as dom, bz, $dom/ddeg$) are used. This is the reason why we decided to solve a wide range of random CSP instances using the MAC algorithm, i.e. the algorithm that maintains arc consistency during search [25]. We tested both branching schemes and both dynamic and adaptive variable ordering heuristics on 7 classes of binary instances situated at the phase transition of search. For each class $\langle n,d,e,t \rangle$, defined as usually, 50 instances have been generated. More precisely, the number of variables n has been set to 40, the domain size d between 8 and 180, the number of constraints e between 753 and 84 (and, so the density between 0.96 and 0.1) and the tightness t (which denotes here the probability that a pair of values is allowed by a relation) between 0.1 and 0.9. The first class $\langle 40,8,753,0.1 \rangle$ corresponds to dense instances involving constraints of low tightness whereas the seventh one $\langle 40,180,84,0.9 \rangle$ corresponds to sparse instances involving constraints of high tightness. What is interesting here is that a significant sampling of domain sizes, densities and tightnesses is considered.

In Tables 1 and 2, we can observe the results that we have obtained with the classical value ordering heuristic $min\text{-}conflicts$ and the "anti" heuristic $max\text{-}conflicts$[1] identified as $confts$. Here $min\text{-}conflicts$ corresponds to the heuristic called mc in [12] and involves selecting the value with the lowest number of conflicts with values in adjacent domains. Performances are given (on average) in terms of the cpu time (in seconds), the

[1] The value ordering heuristics considered here are static [23]. It means that the order of values is computed in a preprocessing step. In any case, we observed similar behaviours with dynamic versions.

		$dom/ddeg$			$dom/wdeg$		
$Instances$		$min\text{-}conflts$	$max\text{-}confts$	$ratio$	$min\text{-}confts$	$max\text{-}confts$	$ratio$
⟨40-8-753-0.1⟩	cpu	42.0	51.4	1.22	34.5	41.6	1.20
	$ccks$	$22M$	$27M$	1.22	$20M$	$24M$	1.20
	$nodes$	43,269	55,558	1.28	38,104	48,158	1.59
⟨40-11-414-0.2⟩	cpu	30.9	35.0	1.13	29.4	32.7	1.11
	$ccks$	$26M$	$29M$	1.11	$26M$	$29M$	1.11
	$nodes$	58,955	70,007	1.18	58,055	67,905	1.17
⟨40-16-250-0.35⟩	cpu	22.1	28.9	1.30	21.0	26.6	1.26
	$ccks$	$30M$	$40M$	1.33	$30M$	$37M$	1.23
	$nodes$	59,669	83,445	1.39	56,036	75,025	1.33
⟨40-25-180-0.5⟩	cpu	33.1	37.1	1.12	28.6	30.0	1.04
	$ccks$	$62M$	$67M$	1.08	$55M$	$57M$	1.03
	$nodes$	85,122	98,519	1.15	69,805	78,005	1.11
⟨40-40-135-0.65⟩	cpu	25.9	34.6	1.33	20.0	25.1	1.25
	$ccks$	$68M$	$89M$	1.30	$53M$	$66M$	1.24
	$nodes$	52,622	74,592	1.41	36,571	49,211	1.34
⟨40-80-103-0.8⟩	cpu	25.8	52.8	2.04	15.3	36.3	2.37
	$ccks$	$98M$	$193M$	1.96	$59M$	$133M$	2.25
	$nodes$	29,989	72,841	2.42	16,163	45,177	2.79
⟨40-180-84-0.9⟩	cpu	113.1	121.3	1.07	40.6	44.6	1.09
	$ccks$	$554M$	$587M$	1.05	$217M$	$231M$	1.06
	$nodes$	76,788	85,482	1.11	20,077	22,557	1.12

Table 1. MAC with d-way branching, $dom/ddeg$ and $dom/wdeg$

		$dom/ddeg$			$dom/wdeg$		
$Instances$		$min\text{-}confts$	$max\text{-}confts$	$ratio$	$minconfts$	$max\text{-}confts$	$ratio$
⟨40-8-753-0.1⟩	cpu	29.3	35.8	1.22	28.9	28.4	0.98
	$ccks$	$22M$	$27M$	1.22	$24M$	$23M$	0.95
	$nodes$	43,268	55,557	1.28	45,650	46,645	1.02
⟨40-11-414-0.2⟩	cpu	23.0	25.9	1.12	26.1	27.3	1.04
	$ccks$	$26M$	$29M$	1.11	$32M$	$33M$	1.03
	$nodes$	59,002	70,026	1.18	69,111	76,941	1.11
⟨40-16-250-0.35⟩	cpu	18.5	24.5	1.32	23.0	24.4	1.06
	$ccks$	$30M$	$40M$	1.33	$39M$	$41M$	1.05
	$nodes$	59,773	83,531	1.18	72,555	82,459	1.13
⟨40-25-180-0.5⟩	cpu	28.8	31.9	1.33	28.5	30.7	1.07
	$ccks$	$62M$	$67M$	1.08	$65M$	$68M$	1.04
	$nodes$	85,187	98,548	1.15	80,017	91,464	1.14
⟨40-40-135-0.65⟩	cpu	21.4	28.6	1.33	19.8	19.6	0.98
	$ccks$	$68M$	$89M$	1.30	$65M$	$64M$	0.98
	$nodes$	52,569	74,544	1.41	44,120	46,573	1.05
⟨40-80-103-0.8⟩	cpu	20.4	42.3	2.07	12.6	18.6	1.47
	$ccks$	$98M$	$193M$	1.96	$64M$	$89M$	1.39
	$nodes$	29,931	72,747	1.41	16,168	28,087	1.73
⟨40-180-84-0.9⟩	cpu	85.0	92.0	1.08	26.4	27.1	1.02
	$ccks$	$553M$	$587M$	1.06	$192M$	$193M$	1.00
	$nodes$	76,489	85,255	1.11	15,835	16,566	1.04

Table 2. MAC with 2-way branching, $dom/ddeg$ and $dom/wdeg$

number of constraint checks (ccks) and the number of nodes of the explored search tree. What is interesting to note is that while the performance ratio between $max\text{-}conflicts$ and $min\text{-}conflicts$ usually lies between 1.1 and 1.3 when one uses d-way branching or a classical heuristic (here, $dom/ddeg$), it falls around 1 when one uses binary branching and $dom/wdeg$. A noticeable exception is for the class $\langle 40{,}80{,}103{,}0.8\rangle$. One can also remark that the proportion of constraint checks per visited node is weaker when $max\text{-}conflicts$ is used. This is natural since by selecting in priority conflicting values, the size of the search space is reduced faster. Finally, our observation suggests that an analysis as the one performed in [16] deserves to be considered for 2-way branching.

Considering the results of our experience about random instances and the fact that, for some types of constraints, good value ordering can significantly reduce the search effort [26], we decided to further investigate value ordering heuristics (assuming, of course, an underlying 2-way branching scheme). In particular, our attention was attracted by the fact that 2-way branching is the basic scheme in SAT solvers. We thought that this might be very helpful to map SAT heuristics to CSP ones. Indeed, considering any SAT encoding of a CSP instance, selecting a pair composed of a variable and a value corresponds to the selection of a literal in SAT.

In this paper, we propose a new value ordering heuristic that is derived from the well known *Jeroslow-Wang (JW)* heuristic [18]. The obtained heuristic, called $min\text{-}inverse$, exploits the bi-directionality of constraints to give a more comprehensive picture in terms of domain reduction when a given value is assigned to a given variable and *also* when a given value is removed from the domain of a given variable. Let us illustrate this with the following example.

Example 1. Let C be the binary constraint depicted by Figure 1. Note that any value in the domain of x_1 occurs in two allowed tuples and is in conflict with two values in the domain of x_2. Consequently, applying a classical value ordering heuristics such as $min\text{-}conflicts$ (or $max\text{-}conflicts$) do not discriminate between the different values of x_1 since all the values of x_1 have the same number of conflicts in x_2.

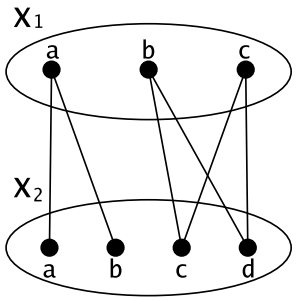

Fig. 1. A constraint between x_1 and x_2. Edges correspond to allowed tuples.

This example shows that it is not always sufficient to only consider the number of conflicts in order to choose the most (or least) promising value. However, considering a

binary branching scheme, when the value a is assigned to x_1 (decision corresponding to the first branch), two values are removed from dom(x_2), and when a is removed from dom(x_1) (decision corresponding to the second branch) two values are also removed from dom(x_2). On the other hand, when the value b or c is assigned to x_1, two values are removed from dom(x_2), and when b or c is removed from dom(x_1) no value is removed from dom(x_2). So, the value a is more constrained than b or c. This illustration shows that it can be important to consider the impact on both branches when evaluating values to be selected by an heuristic.

Our heuristic is then related to $max\text{-}conflicts$, but this last one only gives the estimation of the number of removed values when assigning a value to a given variable (the assignment labelling the first branch of a binary search). In fact, the estimation of the number of removed values when eliminating a given value from the domain of a given variable (the refutation labelling the second branch of a binary search) has not been (to our knowledge) considered so far when devising value ordering heuristics. Interestingly enough, our approach can be used to derive in more general way a suitable value ordering with respect to any type of constraint. We also show a direct correspondence between $min\text{-}conflicts$ (resp. $max\text{-}conflicts$) and the maximum number of literal occurrences in the SAT formula obtained using support (resp. direct) encoding of CSP instances.

The rest of the paper is organized as follows. After some technical background about CSP and SAT, SAT encodings of CSP instances are recalled. Our approach is then presented. Experimental results conducted on a wide range of CSP instances are described and discussed before concluding.

2 Technical Background

2.1 Constraint Satisfaction Problem

A (finite) Constraint Network (CN) P is a pair $(\mathscr{X}, \mathscr{C})$ where $\mathscr{X}$ is a finite set of variables and $\mathscr{C}$ a finite set of constraints. Each variable $x \in \mathscr{X}$ has an associated domain, denoted *dom*(x), which contains the set of values allowed for x. Each constraint $C \in \mathscr{C}$ involves a subset of variables of $\mathscr{X}$, called the scope and denoted *vars*(C), and has an associated relation, denoted *rel*(C), which contains the set of tuples allowed for the variables of its scope. From now on, to simplify and without any loss of generality, we will only consider binary networks, i.e. networks involving binary constraints.

A solution to a constraint network is an assignment of values to all the variables such that all the constraints are satisfied. A constraint network is said to be satisfiable iff it admits at least one solution. The Constraint Satisfaction Problem (CSP) is the NP-complete task of determining whether a given constraint network is satisfiable. A CSP instance is then defined by a constraint network, and solving it involves either finding one (or more) solution or determining its unsatisfiability. To solve a CSP instance, one can modify the constraint network by using inference or search methods. Usually, domains of variables are reduced by removing inconsistent values, i.e. values that can not occur in any solution. Indeed, it is possible to filter domains by considering some properties of constraint networks. Arc Consistency (AC), which remains the central one, guarantees the existence of a support for each value in each constraint.

Algorithm 1 MAC(P = $(\mathscr{X}, \mathscr{C})$: Constraint Network) : Boolean

1: **if** $\mathscr{X} = \emptyset$ **then return** true
2: $P' \leftarrow AC(P)$
3: **if** $P' = \bot$ **then return** false
4: select a pair (x,a) such that $x \in \mathscr{X}$ and $a \in dom(x)$
5: **if** MAC($P'|_{x=a} \backslash x$) **then return** true
6: **if** MAC($P'|_{x \neq a}$) **then return** true
7: **return** false

Definition 1. *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN, $C \in \mathscr{C}$ such that $vars(C) = \{x, y\}$ and $a \in dom(x)$.*

- *The set of supports of (x, a) in C, denoted $supports(C, x, a)$, corresponds to the set $\{b \in dom(y) \mid (a, b) \in rel(C)\}$.*
- *The set of conflicts of (x, a) in C, denoted $conflicts(C, x, a)$, corresponds to the the set $\{b \in dom(y) \mid (a, b) \notin rel(C)\}$.*

Definition 2. *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN. A pair (x,a), with $x \in \mathscr{X}$ and $a \in dom(x)$, is arc consistent (AC) iff $\forall C \in \mathscr{C} | x \in vars(C)$, $supports(C, x, a) \neq \emptyset$. P is AC iff $\forall x \in \mathscr{X}$, dom($x$) $\neq \emptyset$ and $\forall a \in dom(x)$, (x, a) is AC.*

Let us now, briefly describe the well known MAC algorithm [25]. This algorithm aims at solving a CSP instance and performs a depth-first search with backtracking while maintaining arc consistency. More precisely, at each step of the search, a variable assignment is performed followed by a filtering process called constraint propagation which corresponds to enforcing arc-consistency. Algorithm 1 corresponds to a recursive version of the MAC algorithm (using binary branching). It returns $true$ iff the given constraint network P is satisfiable. More precisely, if P can be made arc consistent (i.e. AC(P) $\neq \bot$) then the search for a solution begins. It involves selecting a pair (x,a) and trying first $x = a$ and then $x \neq a$ (if no solution has been found with $x = a$). After any consistent assignment, the assigned variable is eliminated from the network and search is continued (line 5)[2]. When the current constraint network has no more variables (line 1), it means that a solution has been found.

2.2 Encoding CSP into SAT

Propositional satisfiability (SAT) is the problem of deciding whether a Boolean formula in conjunctive normal form (CNF) is satisfiable. A *CNF formula* Σ is a set (interpreted as a conjunction) of *clauses*, where a clause is a set (interpreted as a disjunction) of *literals*. A literal is a positive or negated propositional variable. A *truth assignment* of a Boolean formula is an assignment of truth values $\{true, false\}$ to its variables.

[2] $P|_{x=a}$ denotes the constraint network obtained from P by restricting the domain of x to the singleton $\{a\}$ whereas $P|_{x \neq a}$ denotes the constraint network obtained from P by removing the value a from the domain of x. $P \backslash x$ denotes the constraint network obtained from P by removing the variable x.

A *model* of a formula is a truth assignment that satisfies the formula. SAT is one of the most studied NP-Complete problems because of its theoretical and practical importance. Encouraged by the impressive progress in practical solving of SAT, various applications ranging from formal verification to planning are encoded and solved using SAT. CSP instances can also be reformulated as SAT instances.

In this paper, we consider the most commonly used encodings of CSP into SAT, namely, the direct encoding [7] and the support encoding [14]. In both SAT encodings of a constraint network $P = (\mathscr{X}, \mathscr{C})$, a propositional variable $x_{i,v}$ is associated to each pair (x_i,v) of P with $x_i \in \mathscr{X}$ and $v \in dom(x_i)$. The correspondence is the following: $x_{i,v}$ is true if x_i is assigned the value v (i.e. $x_i = v$) and $x_{i,v}$ is false if v is removed from dom(x_i) (i.e. $x_i \neq v$).

Direct Encoding The direct encoding [7] of a constraint network $P = (\mathscr{X}, \mathscr{C})$ involves two kinds of clauses:

- *At least one*: for each variable $x_i \in \mathscr{X}$ with dom(x_i) = $\{v_1, v_2, \ldots, v_d\}$, a clause of the form $x_{i,v_1} \vee x_{i,v_2} \cdots \vee x_{i,v_d}$ expresses the fact that the variable x_i must be assigned at least one value from its domain.
- *Conflict*: for each triplet (C,x_i,v) such that $C \in \mathscr{C}$, $x_i \in vars(C)$ and $v \in dom(x_i)$, we have, assuming that $vars(C) = \{x_i, x_j\}$ a clause $\neg x_{i,v} \vee \neg x_{j,w}$ for each value $w \in conflicts(C, x_i, v)$.

Support Encoding The idea of encoding support has been first introduced by Kasif in [19] and expanded on by Gent [14]. In support encoding, two kinds of clauses are introduced:

- *At most one*: for each variable $x_i \in \mathscr{X}$ and for any pair $\{v, w\} \subseteq dom(x_i)$, the following clause encodes the fact that the variable x_i must be assigned at most one value in $\{v, w\}$: $\neg x_{i,v} \vee \neg x_{i,w}$.
- *Support*: for each triplet (C,x_i,v) such that $C \in \mathscr{C}$, $x_i \in vars(C)$ and $v \in dom(x_i)$, we have, assuming that $vars(C) = \{x_i, x_j\}$ a clause $\neg x_{i,v} \vee x_{j,w_1} \vee x_{j,w_2} \vee \cdots \vee x_{j,w_k}$ where $supports(C, x_i, v) = \{w_1, w_2, \ldots, w_k\}$.

Note that the following set of clauses $\Sigma_D(C)$ is obtained from the constraint C of Example 1 using the direct encoding:

- *At least*: $(x_{1,a} \vee x_{1,b} \vee x_{1,c}) \wedge (x_{2,a} \vee x_{2,b} \vee x_{2,c} \vee x_{2,d})$
- *Conflict*: $(\neg x_{1,a} \vee \neg x_{2,c}) \wedge (\neg x_{1,a} \vee \neg x_{2,d}) \wedge (\neg x_{1,b} \vee \neg x_{2,a}) \wedge (\neg x_{1,b} \vee \neg x_{2,b}) \wedge (\neg x_{1,c} \vee \neg x_{2,a}) \wedge (\neg x_{1,c} \vee \neg x_{2,b})$

The following set of clauses $\Sigma_S(C)$ is obtained from C using the support encoding:

- *At most*: $(\neg x_{1,a} \vee \neg x_{1,b}) \wedge (\neg x_{1,a} \vee \neg x_{1,c}) \wedge (\neg x_{1,b} \vee \neg x_{1,c}) \wedge (\neg x_{2,a} \vee \neg x_{2,b}) \wedge (\neg x_{2,a} \vee \neg x_{2,c}) \wedge (\neg x_{2,a} \vee \neg x_{2,d}) \wedge (\neg x_{2,b} \vee \neg x_{2,c}) \wedge (\neg x_{2,b} \vee \neg x_{2,d}) \wedge (\neg x_{2,c} \vee \neg x_{2,d})$
- *Support*: $(\neg x_{1,a} \vee x_{2,a} \vee x_{2,b}) \wedge (\neg x_{1,b} \vee x_{2,c} \vee x_{2,d}) \wedge (\neg x_{1,c} \vee x_{2,c} \vee x_{2,d}) \wedge (\neg x_{2,a} \vee x_{1,a}) \wedge (\neg x_{2,b} \vee x_{1,a}) \wedge (\neg x_{2,c} \vee x_{1,b} \vee x_{1,c}) \wedge (\neg x_{2,d} \vee x_{1,b} \vee x_{1,c})$

Remark 1. Let us note that the *at most* (resp. *at least*) clauses are not required in direct (resp. support) encoding for checking satisfiability [30, 14]. One must add such clauses only if a mapping between SAT and CSP solutions is needed.

Support encoding admits interesting features. In [14], it is shown that encoding supports enables arc consistency in the original CSP instance to be established by unit propagation in the translated SAT instances. Last but not least, applying the well known DPLL algorithm to the obtained SAT instance behaves exactly like the MAC algorithm on the original CSP instance. We can also mention that support encoding has been extended to encode non binary constraints in SAT [3]. Interestingly enough, it has been proved in [9] that support clauses can be inferred from direct encoding using HyperBin resolution introduced by Bacchus [1]. These nice results open new interesting perspectives for establishing strong connections between SAT and CSP. The results that we present below on value ordering can be seen as a step in this direction.

3 Value Ordering Heuristics from SAT to CSP

3.1 SAT Branching Heuristics

Many branching heuristics has been proposed in SAT. One can cite the most recent ones, namely the VSIDS and UP heuristics used in Zchaff [31] and Satz solvers [22]. The first one uses literal occurrences in the set of learned no-goods whereas the second one measures the effect of unit propagation on the formula when a literal is assigned a truth value. Previously, CSAT [10] and POSIT [11], among other solvers, used simpler heuristics. Most of them are variants of the well-known Jeroslow-Wang (JW) heuristic [18], and evaluate a given literal according to syntactical properties (e.g. occurrence number of literals, clause length).

In SAT branching heuristics, the score, denoted $H(\Sigma, x)$, of a variable x of a CNF Σ is generally defined as a function f of the weight w associated with its positive and negative literals, i.e. $H(\Sigma, x) = f(w(x), w(\neg x))$. The next variable to assign is then chosen among variables with the greatest score. For example, the two sided Jeroslow-Wang rule is defined as : $H_{JW}(\Sigma, x) = w(x) + w(\neg x)$ where $w(x) = \sum_{x \in c} w(c)$, with $c \in \Sigma$ and $w(c) = 2^{-|c|}$. JW can be seen as a refinement of the MOMS (Maximum Occurrences in clauses of Minimal Size) heuristic [10]. Another basic heuristic that we consider in this paper is $H_{OCC}(\Sigma, x)$ that can be obtained from F_{JW} by instantiating $w(c)$ to 1. With H_{OCC}, we select in priority a variable with the greatest number of occurrences in the formula.

3.2 Mapping SAT Heuristics to CSP

Using direct and support encodings, we present now the CSP value ordering heuristics respectively corresponding to H_{OCC} and H_{JW} SAT branching heuristics. We have to emphasize that, when a CSP solver is based on a 2-way branching scheme, in order to simulate SAT branching heuristics, one needs to evaluate the score of $n * d$ pairs composed of a variable and a value. This can be very time consuming, especially if such evaluation is done at each node of the search tree. Hence, in the following, we

investigate the mapping of SAT branching heuristics to CSP, under the hypothesis that the CSP solver performs at each step of the search, a vertical selection (the choice of a variable) followed by a horizontal selection (the choice of a value for the selected variable). As a consequence, the *at least* and *at most* clauses can be omitted in our analysis since all literals occur the same number of times in such clauses. It is important to remark that this hypothesis corresponds to the current practice of CSP solvers.

Mapping $\mathbf{H_{OCC}}$ Let us show how the basic H_{OCC} SAT branching heuristic on direct (resp. support) encoding corresponds to the CSP value ordering heuristic max-$conflicts$ (resp. min-$conflicts$).

Property 1. Let $P = (\mathscr{X}, \mathscr{C})$ be a constraint network, Σ_D (resp. Σ_S) the CNF formula obtained from P using direct (resp. support) encoding. For a given variable $x_i \in \mathscr{X}$ and value $v \in dom(x_i)$, we have,

- $H_{OCC}(\Sigma_D, x_{i,v}) = 1 + \sum_{C \in \mathscr{C} | x_i \in vars(C)} |conflicts(C, x_i, v)|$
- $H_{OCC}(\Sigma_S, x_{i,v}) = \begin{pmatrix} 2 \\ |dom(x_i)| \end{pmatrix} + \sum_{C \in \mathscr{C} | x_i \in vars(C)} (1 + |supports(C, x_i, v)|)$

Proof. Indeed, for Σ_D, the positive literal $x_{i,v}$ occurs exactly one time positively (*at least* clause) and the negative literal $\neg x_{i,v}$ occurs the same number of times as the number of forbidden tuples with v in all constraints involving x_i (*conflict* clauses). For Σ_S, the negative literal $\neg x_{i,v}$ occurs $\begin{pmatrix} 2 \\ |dom(x_i)| \end{pmatrix}$ times in *at most* clauses and exactly one time in each constraint involving x_i. The number of positive occurrences of $x_{i,v}$ corresponds to the number of tuples supporting the value v of x_i for each constraint involving x_i. □

We can remark that, if the choice is restricted to the values of a given variable (i.e. if we adopt the current model based on a vertical selection followed by an horizontal one), then H_{OCC} on direct (resp. support) encoding delivers the same ordering as max-$conflicts$ (resp. min-$conflicts$). If it is not the case (i.e. if we perform a global selection among all pairs of the form (x,v)), then, whereas H_{OCC} and max-$conflicts$ always correspond with respect to direct encoding, H_{OCC} and min-$conflicts$ may deliver, with respect to support encoding, different orderings since the number of occurrences of a literal in the *at most* clauses depends on the size of the domains.

Mapping $\mathbf{H_{JW}}$ First, for the direct encoding, as the conflict clauses are all binary, H_{JW} admits the same behavior as H_{OCC} when the choice is restricted to the values of a given variable. On the other hand, when considering the support encoding, the length of the clauses, which depends on the number of supports of a value with respect to a given constraint, becomes important. Consequently, considering H_{JW} on support encoding, we derive a new interesting value ordering that corresponds to maximize the function SI defined as follows.

Definition 3. *Let $P = (\mathscr{X}, \mathscr{C})$ be a constraint network. For any pair (x_i,a), with $x_i \in \mathscr{X}$ and $a \in dom(x_i)$, we define:*

$$SI(P, x_i, v_i) = \sum_{c \in \mathscr{C} | vars(c) = \{x_i, x_j\}} [W_\downarrow(C, x_i, v_i) + W_\uparrow(C, x_i, v_i)]$$

where,

$$W_\downarrow(C, x_i, v_i) = w(1 + |supports(C, x_i, v_i)|) \text{ and}$$
$$W_\uparrow(C, x_i, v_i) = \sum_{v_j \in supports(C, x_i, v_i))} w(1 + |supports(C, x_j, v_j)|)$$

Here, w denotes any weighting function.

The following property establishes the connection between the SAT JW heuristic and the new derived CSP value ordering heuristic. Note that the factor α which is introduced below is such that for any variable $x_i \in \mathscr{X}$ and any pair $\{a, b\} \subseteq dom(x_i)$, we have $\alpha_{x_i,a} = \alpha_{x_i,b}$. It simply means that this factor can be discarded when a value must be selected in the domain of a variable (and this is what is done by SI).

Property 2. Let $P = (\mathscr{X}, \mathscr{C})$ be a constraint network and Σ_S the CNF formula obtained from P using the support encoding. For any pair (x_i,a), with $x_i \in \mathscr{X}$ and $a \in dom(x_i)$, we have:

$$H_{JW}(\Sigma_S, x_{i,v}) = SI(P, x_i, v) + \alpha_{x_i,v}$$

where $\alpha_{x_i,v}$ corresponds to the score of H_{JW} applied on *at most* clauses of Σ_S involving $\neg x_{i,v}$.

Proof. We have to show that $H_{JW}(\Sigma_S, x_{i,v}) - \alpha_{x_i,v} = SI(P, x_i, v)$. It simply means that we do not have to take into consideration the *at most* binary clauses involving $\neg x_{i,v}$. Consequently, we only need to consider the *support* clauses. The proof is a direct consequence of the following fact: for each constraint $C \in \mathscr{C} | vars(C) = \{x_i, x_j\}$, the negative literal $\neg x_{i,v}$ occurs exactly in one *support* clause of size $1 + |supports(C, x_i, v_i)|$, and the positive literal $x_{i,v}$ occurs in $|supports(C, x_i, v_i))|$ *support* clauses of different length (i.e. for each $v_j \in supports(C, x_i, v_i)$ the length of the clause is $1 + |supports(C, x_j, v_j)|$). Under the same weighting function $w(c) = 2^{-|c|}$, the two heuristics are equivalent, i.e. compute the same value for a given literal. □

As a summary, while the direct encoding naturally leads to $max\text{-}conflicts$ that adheres to the $fail\text{-}first$ policy, the support encoding also leads to adhere to this policy. Indeed, we have a correspondence between H_{JW} and SI (maximizing the score of H_{JW} is equivalent to maximizing SI). In practice (for our experimentation), we will use a simplified version of SI, denoted SI_s, obtained by substituting $w(1 + |supports(C, x_i, v_i)|)$ (resp. $w(1 + |supports(C, x_j, v_j)|)$) by $|supports(C, x_i, v_i)|$ (resp. $|supports(C, x_j, v_j)|$) in Definition 3. As a result, the new heuristic that we propose, called $min\text{-}inverse$, corresponds to minimize the value of SI_s since instead of using $w(c) = 2^{-|c|}$, we use $w(c) = |c|$.

To illustrate this, let us consider again Example 1. Applying the SI_s function on the variable x_1, we obtain $SI_s(C, x_1, a) = 4$, $SI_s(C, x_1, b) = 6$ and $SI_s(C, x_1, c) = 6$.

The best value according $min\text{-}inverse$ is then a. It is justified as follows. All values in dom(x_1), when assigned to x_1, lead to the removal of 2 (arc inconsistent) values from the domain of x_2. The main difference is that, while removing a from the domain of x_1 leads to the removal of 2 values from the domain of x_2, removing b or c from the domain of x_1 does not lead to any inconsistent value in the domain of x_2. More generally, given a constraint C such that $vars(C) = \{x_i, x_j\}$ and $v_i \in dom(x_i)$, minimizing $W_{\downarrow}(C, x_i, v_i)$ (resp. $W_{\uparrow}(C, x_i, v_i)$) increases the potential number of values of dom(x_j) that can be made arc inconsistent when considering $x_i = v_i$ (resp. $x_i \neq v_i$).

4 Experiments

To prove the practical interest of adhering to the $fail\text{-}first$ policy for value ordering, we have implemented the different heuristics described in the previous sections in our platform $Abscon$ and conducted an experimentation with respect to some scheduling instances and the full set of 1064 instances used as benchmarks of the first CSP Competition [28]. The search algorithm that has been employed is MAC equipped with $dom/wdeg$. All value ordering heuristics are implemented statically: an ordering is established prior to search and remains unchanged during the whole search process[23].

First, we searched to establish a comparison between all heuristics with respect to series of 100 open-shop scheduling instances randomly generated using Taillard's model [27] by fixing 5 jobs and 5 machines. For each instance, the optimal makespan OPT have been computed. We have considered different sets of instances by setting different time windows around the optimal makespan. For $x < OPT$, instances are unsatisfiable whereas for $x \geq OPT$, instances are satisfiable. Note that the hardest instances are those that are unsatisfiable and such that x is close to OPT.

Figure 2 shows the proportion of instances that have been solved (the higher, the better) in a limited amount of time (300 seconds). One can observe that $min\text{-}inverse$ is the most efficient heuristic on the hardest unsatisfiable instances, whereas the classical $min\text{-}conflicts$ is only able to solve a small number of these instances. Note that this statement is true even on hard satisfiable instances (for $1 \leq x \leq 1.01$). On easier satisfiable instances, following the $promise$ policy seems better.

Then, we tested the different heuristics on the 1064 instances of the first CSP Competition. Figure 3 shows the percentage of unsolved instances (the lower, the better) against search time. Although $min\text{-}inverse$ seems better than the other heuristics (in particular, between 200 and 350 allowed seconds), the difference is quite small.

So, let us zoom on the results obtained for a limited set of hard representative instances (Table 3). Here, the time-out was set to $1,000$ seconds. On random instances, there is no clear winner. In fact, on these instances, there is no structure concerning supports, and so, heuristics based on these are not very relevant. On academic instances, results are more spectacular, although sometimes chaotic. On *allIntervalSeries*, *pigeons* and *queen-knights* instances, $min\text{-}inverse$ is clearly better while $min\text{-}conflicts$ badly behaves. However, on *GolombRuler* or *BQWH* instances, $min\text{-}conflicts$ is more efficient (note that these are mainly satisfiable instances, so, the *promise* strategy may be more of practical interest in this case). On some real world instances such as FAPP and RLFAP problems, the impact of value ordering heuristics seems negligible. Finally,

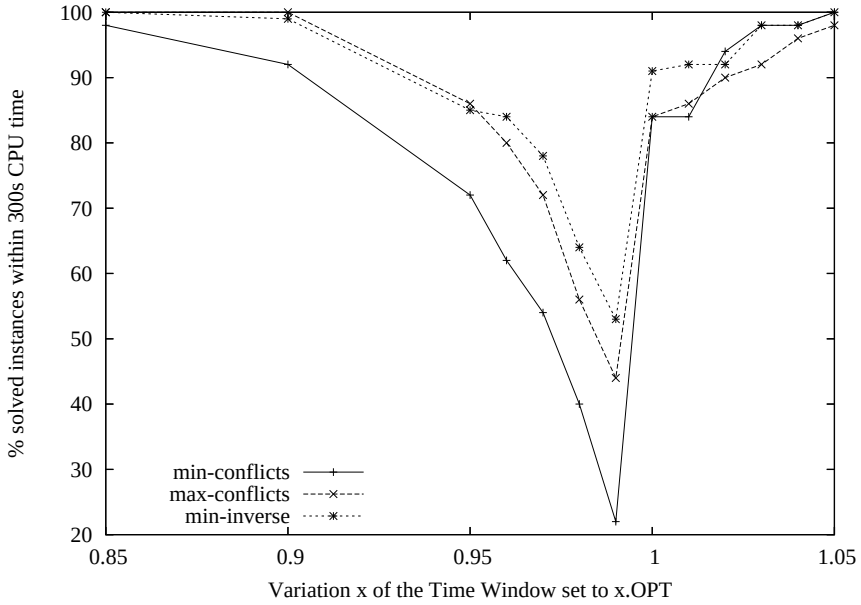

Fig. 2. Performance of value heuristics on 5x5 open-shop instances

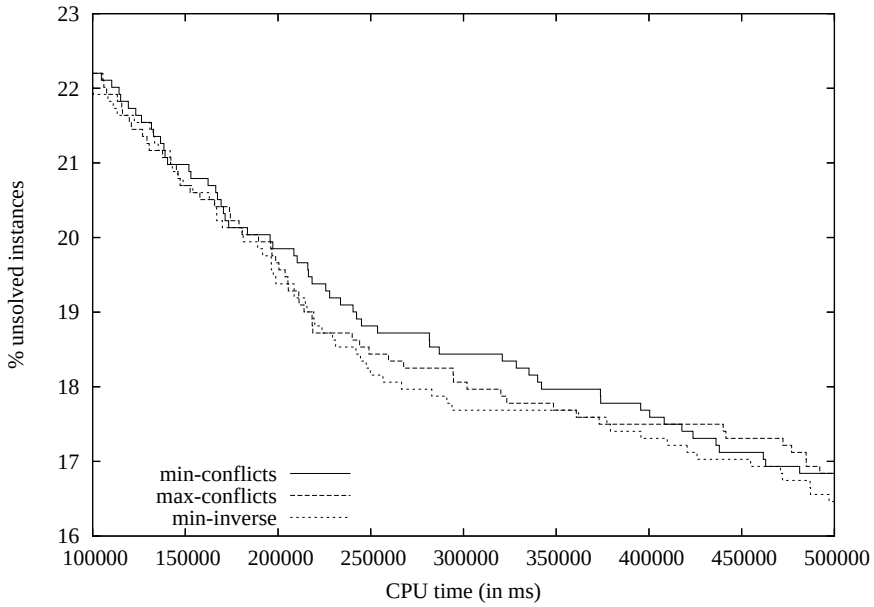

Fig. 3. Performance of value heuristics on competition's instances

$Instances$		$min\text{-}conflicts$	$max\text{-}conflicts$	$min\text{-}inverse$
Random Instances				
frb45-21-3	cpu	665	$timeout$	389
(sat)	$nodes$	896,278		511,101
frb45-21-5	cpu	$timeout$	243	258
(sat)	$nodes$		313,356	321,895
frb50-23-4	cpu	244	$timeout$	$timeout$
(sat)	$nodes$	276,441		
random-2-40-19-443-230-9	cpu	641	441	438
$(unsat)$	$nodes$	904,470	704,712	706,123
random-3-24-24-76-632-8	cpu	$timeout$	884	931
(sat)	$nodes$		1,537,568	1,558,190
Academic Instances				
series-15	cpu	849	59	3
(sat)	$nodes$	2,970,402	213,266	13,972
series-16	cpu	$timeout$	356	15
(sat)	$nodes$		1,074,057	59,314
bqwh-18-141-73	cpu	29	32	26
(sat)	$nodes$	115,916	136,821	106,254
bqwh-18-141-84	cpu	40	166	165
(sat)	$nodes$	157,398	685,424	685,424
gr-44-10	cpu	78	489	492
$(unsat)$	$nodes$	13,213	93,343	92,454
gr-55-10	cpu	25	$timeout$	$timeout$
(sat)	$nodes$	5,334		
pigeons-40	cpu	482	486	167
$(unsat)$	$nodes$	773,274	773,277	280,490
pigeons-45	cpu	255	258	98
$(unsat)$	$nodes$	304,584	304,588	119,820
qk-20-20-5-add	cpu	$timeout$	149	40
$(unsat)$	$nodes$		85,264	23,698
qk-20-20-5-mul	cpu	$timeout$	163	62
$(unsat)$	$nodes$		81,393	31,990
qa-5	cpu	4	2	3
(sat)	$nodes$	17,998	4,956	11,778
qa-6	cpu	245	61	197
(sat)	$nodes$	503,750	81,764	331,212
QCP-20-30	cpu	119	120	$timeout$
(sat)	$nodes$	237,759	237,759	
QCPp-20-14	cpu	$timeout$	$timeout$	6
(sat)	$nodes$			13,028
Real-world Instances				
scen11-f6	cpu	299	545	354
$(unsat)$	$nodes$	216,648	395,008	262,696
scen11-f7	cpu	163	344	220
$(unsat)$	$nodes$	112,952	251,098	164,593
163-TSP-25	cpu	282	530	280
(sat)	$nodes$	121,004	234,098	110,680
e0ddr1-10-by-5-10	cpu	19	643	$timeout$
(sat)	$nodes$	51,715	1,524,492	
e0ddr1-10-by-5-5	cpu	304	0	13
(sat)	$nodes$	1,244,563	50	36,888
enddr1-10-by-5-3	cpu	$timeout$	$timeout$	2
(sat)	$nodes$			1,757

Table 3. MAC with 2-way branching, using different value ordering heuristics

some job-shop instances from the competition show very chaotic results. In fact, we expect these problems to present an heavy-tailed behaviour.

To summarize, although the results must be tempered, we think that our experimentation allows to demonstrate that, with binary branching and an adaptive variable heuristic such as $dom/wdeg$, using a value ordering heuristic such as $max\text{-}conflict$ or $min\text{-}inverse$ that adheres to the $fail\text{-}first$ policy, is not penalizing and can even outperform the classical $min\text{-}conflicts$.

5 Conclusion

It is well known that the right approach to select values during search is to follow the $promise$ policy [2, 29] - the objective is to follow a path that maximizes the likelihood of finding a solution. However, we noticed that most of the experimental studies supporting this intuition have been based on d-way branching or/and non adaptive variable ordering heuristics.

In this paper, we first show that, on random instances, the anti-promise heuristic $max\text{-}conflicts$ was often as efficient as the standard promise $min\text{-}conflicts$ when 2-way branching and the heuristic $dom/wdeg$ were used. Our understanding of this phenomenon is that, as $dom/wdeg$ is able to efficiently refute unsatisfiable sub-trees, the overhead of refuting more unsatisfiable sub-trees (as, more often than not, we guide search toward unsatisfiable sub-trees) is compensated by the benefit of rapidly reducing the search space.

Then, after studying mappings from CSP to SAT, we devise a new heuristic, denoted $min\text{-}inverse$, that adheres to the $fail\text{-}first$ policy and that corresponds to the SAT Jeroslow-Wang branching heuristic. This correspondence supports our intuition that following the $fail\text{-}first$ policy for value ordering, in addition to variable ordering, can pay off. The results that we have obtained when experimenting a large sampling of problems indicate that $min\text{-}inverse$ and $max\text{-}conflicts$ can outperform $min\text{-}conflicts$.

As a perspective of this work, we project to study heuristics to perform global selection of pairs of the form (X,a) during search as it corresponds to the basic mechanism of 2-way branching. It means that we do not have anymore to distinguish between vertical and horizontal selection. The challenge is then to make such selections both cheap to realize and efficient in practice.

References

1. F. Bacchus. Enhancing Davis Putnam with extended binary clause reasoning. In *Proceedings of AAAI'02*, pages 613–619, 2002.
2. J.C. Beck, P. Prosser, and R.J. Wallace. Variable ordering heuristics show promise. In *Proceedings of CP'04*, pages 711–715, 2004.
3. C. Bessiere, E. Hebrard, and T. Walsh. Local consistencies in SAT. In *Selected revised papers from SAT'03*, pages 299–314, 2003.
4. C. Bessiere and J. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of CP'96*, pages 61–75, 1996.
5. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.

6. D. Brelaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.
7. J. de Kleer. A comparison of ATMS and CSP techniques. In *Proceedings of IJCAI'89*, pages 290–296, 1989.
8. R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.
9. L. Drake, A. Frisch, I. Gent, and T. Walsh. Automatically reformulating SAT-encoded CSPs. In *Proceedings of the RCSP'02 workshop held with CP'02*, 2002.
10. O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. Sat versus unsat. In *Second DIMACS Challenge*, pages 299–314, 1993.
11. J.W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.
12. D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of IJCAI'95*, pages 572–578, 1995.
13. P.A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of ECAI'92*, pages 31–35, 1992.
14. I.P. Gent. Arc consistency in SAT. In *Proceedings of ECAI'02*, pages 121–125, 2002.
15. R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
16. T. Hulubei and B. O'Sullivan. Search heuristics and heavy-tailed behaviour. In *Proceedings of CP'05*, pages 328–342, 2005.
17. J. Hwang and D.G. Mitchell. 2-way vs d-way branching for CSP. In *Proceedings of CP'05*, pages 343–357, 2005.
18. R.G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.
19. S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45:275–286, 1990.
20. K. Kask, R. Dechter, and V. Gogate. Counting-based look-ahead schemes for constraint satisfaction. In *Proceedings of CP'04*, pages 317–331, 2004.
21. C. Lecoutre, F. Boussemart, and F. Hemery. Backjump-based techniques versus conflict-directed heuristics. In *Proceedings of ICTAI'04*, pages 549–557, 2004.
22. C.M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of IJCAI'97*, pages 366–371, 1997.
23. D. Meetah and M.R.C. van Dongen. Static value ordering heuristics for constraint satisfaction problems. In *Proceedings of CPAI'05 workshop held with CP'05*, pages 49–62, 2005.
24. N. Prcovic and B. Neveu. Progressive focusing search. In *Proceedings of ECAI'02*, pages 126–130, 2002.
25. D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP'94*, pages 10–20, 1994.
26. B.M. Smith and P. Sturdy. Value ordering for finding all solutions. In *Proceedings of IJCAI'05*, pages 311–316, 2005.
27. E. Taillard. Benchmarks for basic scheduling problems. *European journal of operations research*, 64:278–295, 1993.
28. M.R.C. van Dongen, editor. *Proceedings of CPAI'05 workshop held with CP'05*, volume II, 2005.
29. R.J. Wallace. Heuristic policy analysis and efficiency assessment in constraint satisfaction search. In *Proceedings of CPAI'05 workshop held with CP'05*, pages 79–91, 2005.
30. T. Walsh. SAT v CSP. In *Proceedings of CP'00*, pages 441–456, 2000.
31. L. Zhang, C.F. Madigan, M.W. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of ICCAD'01*, pages 279–285, 2001.