Refining Constraint Weighting

Hugues Wattez*†, Christophe Lecoutre*†, Anastasia Paparrizou*, Sébastien Tabary*†
*CRIL, CNRS UMR 8188, Lens, France
†Université d'Artois, Lens, France
Email: {wattez,lecoutre,paparrizou,tabary}@cril.fr

***Abstract*—Backtracking search is a complete approach that is traditionally used to solve instances modeled as constraint satisfaction problems. The space explored during search depends dramatically on the order that variables are instantiated. Considering that a perfect variable ordering might result to a backtrack-free search (i.e., finding backdoors, cycle cutsets), finding heuristics for variable ordering has always attracted research interest. For fifteen years, constraint weighting has been shown to be a successful approach for guiding backtrack search. In this paper, we show how the popular generic variable ordering heuristic `dom/wdeg` can be made more robust by taking finer information at each conflict: the "current" arity of the failing constraint as well as the size of the current domains of the variables involved in that constraint. Our experimental results show the practical interest of this refined variant of constraint weighting.**

***Index Terms*—constraint satisfaction, search heuristics, constraint weighting**

I. Introduction

Backtrack search remains a classical approach for solving instances of the Constraint Satisfaction Problem (CSP). It is based on a depth-first exploration, which is conducted by instantiating variables in sequence and backtracking when dead-ends occur. For efficiently exploring the search space, a property (called local consistency) is enforced at each step of the search so as to filter the domains of the variables; typically most of the constraints guarantees the property known as (generalized) arc consistency.

The order in which variables are chosen during the depth-first traversal of the search space is decided by a *variable ordering heuristic* H. At each internal node of the search tree built by the backtrack search algorithm, the next variable x is selected by H, and a value is assigned to x according to a *value ordering heuristic*, which can simply be the lexicographic order over the domain of x. Choosing the right variable ordering heuristic for a given constraint network is a key issue in the design of constraint solvers, since different heuristics can lead to drastically different search trees.

For a long time, the most popular (variable ordering) heuristic was `dom` [8] that selects variables in sequence of increasing size of domain. However, fifteen years ago, modern *adaptive* heuristics were introduced: they take into account information collected along the part of the search space (tree) already explored. The two first proposed generic adaptive heuristics are `impact` [13] and `wdeg` [2]. The former relies on a measure of the effect of any assignment, and the latter associates a counter with each constraint (and indirectly, with each variable) indicating how many times any constraint led to a domain wipe-out. Counting-based heuristics [12] and activity-based search [10] are two more recent additional adaptive techniques for guiding search.

Currently, the constraint weighting variant `dom/wdeg` that additionally takes domain sizes into account, is considered as the most robust generic heuristic, as it is used by default in many constraint solvers (e.g., Choco). It certainly remains the state-of-the-art (as a generic heuristic) even if several attempts were made to further improve it. A first idea [5] was to learn weighting information during an initial phase in which variables are chosen at random and the search is repeatedly run to a fixed cutoff. This random probing method was intended to start the "real" search better informed after gathering information from different parts of the search space. Some other variants were also studied in [1]. By noting the constraint responsible of each value deletion (a kind of explanation), it is possible to implement different weighting strategies. For example, whenever there is a domain wipe-out on a variable x while propagating constraint c, the weight of every constraint responsible for the removal of a value of x is incremented. Another variant uses an aging mechanism, as in some SAT solvers, which periodically divides the value of all weights by a constant, thereby giving greater importance to conflicts discovered recently. Surprisingly, the "basic" `dom/wdeg` heuristic remained very competitive compared to such attractive variants.

A specific variant of constraint weighting was shown to be successful for job-shop scheduling problems [6]: by reasoning from the domain sizes associated with the variables denoting the starting times of tasks, the proposed weighting-based heuristic was shown to be better informed and to yield particularly strong performance for scheduling. Because it was observed that the efficiency of `dom/wdeg` may deteriorate when problem instances contain many constraints of large arity (because it loses its ability to discriminate variables), a possible approach [9] is to weight a conflict set rather than the entire scope of a failed constraint. Although this approach is stimulating, it is unfortunately not generic since one has to conceive a specific procedure for each type of (global) constraints.

More recently, a new competitive heuristic [7], called `CHS`, has been proposed by exploiting the history of search failures. Techniques coming from reinforcement learning are used to

2375-0197/19/$31.00 ©2019 IEEE
DOI 10.1109/ICTAI.2019.00019

make an exponential recency weighted average in order to estimate the evolution of the hardness of constraints throughout the search. In brief, this heuristic gives a higher reward to constraints that fail regularly over short periods.

The paper is organized as follows. After some preliminaries, we introduce classical variable ordering heuristics. Next, we show how to refine constraint weighting, and demonstrate the practical interest of our approach. Finally, we conclude.

II. Preliminaries

A *constraint network* P is composed of a finite set of variables $\mathcal{X}$, and a finite set of constraints $\mathcal{C}$. Each variable x must be assigned a value from its current domain, denoted by $\mathtt{dom}(x)$; the initial domain of x is denoted by $\mathtt{dom}^{\mathtt{init}}(x)$. Each constraint c represents a mathematical relation over an ordered set of variables, called the *scope* of c, and denoted by $\mathtt{scp}(c)$. The *arity* of a constraint c is the size of its scope. The *degree* of a variable x is the number of constraints of $\mathcal{C}$ involving x.

A *solution* to P is the assignment of a value to each variable of $\mathcal{X}$ such that all constraints of $\mathcal{C}$ are satisfied. A constraint network is *satisfiable* iff it admits at least one solution. The *Constraint Satisfaction Problem (CSP)* is to determine whether a given constraint network is satisfiable, or not. A classical approach for solving this NP-complete problem is to perform a depth-first search with backtracking, while enforcing a property called *(generalized) arc consistency* [11] after each taken decision. This procedure, called *Maintaining Arc Consistency* (MAC) [14], builds a binary search tree $\mathcal{T}$: for each internal node ν of $\mathcal{T}$, a pair (x, v) is selected where x is a variable and v is a value in $\mathtt{dom}(x)$. Then, two cases are considered: the assignment $x = v$ (positive decision) and the refutation $x \neq v$ (negative decision). In this paper, we shall be interested in the *future* variables of a constraint c, denoted by $\mathtt{fut}(c)$, which are the variables at the current node of the search tree that have not been explicitly assigned by MAC.

Backtrack search algorithms that rely on deterministic variable ordering heuristics have been shown to exhibit heavy-tailed behavior on both random and structured CSP instances [4]. This issue can be alleviated using *randomization* and *restart* strategies, which respectively incorporate some random choices in the search process, and iteratively restart the computation from the beginning, with a different variable ordering.

III. Variable Ordering Heuristics

We provide in this section a quick overview of popular general-purpose search heuristics. The simple variable ordering heuristic `dom` [8], which selects variables in sequence of increasing size of domain, has long been considered as the most robust backtrack search heuristic. However, fifteen years ago, modern *adaptive* heuristics were introduced: they take into account information collected along the part of the search space (tree) already explored.

In this paper, we shall mainly focus our attention to the very popular heuristic `wdeg`, and its variant `dom/wdeg`. As a baseline, we shall also consider `impact` and `activity`, which are defined as follows:

- `impact`, or IBS (Impact-Based Search) selects in priority the variable with the highest impact. The impact of a variable x gives a measure about the importance of x in reducing the search space [13]. The size of the search space of P is the product of all current domain sizes:
$$\mathtt{size}(P) = \prod_{x \in \mathcal{X}} |\mathtt{dom}(x)|$$
The impact I of a variable assignment $x = a$ on P is computed as follows:
$$I(x = a) = 1 - \frac{\mathtt{size}(P')}{\mathtt{size}(P)}$$
where $P' = AC(P|_{x=a})$ denotes the CN obtained after assigning x to a and enforcing (generalized) arc consistency. Note that if P' leads to a failure, then $I(x = a) = 1$. It is easy to see that this heuristic can be used for value selection as well.
- `activity`, or ABS (Activity-Based Search) selects in priority the variable with the highest activity. The activity of a variable x is roughly measured by the number of times the domain of x is reduced during search [10]. This heuristic is motivated by the key role of propagation in constraint programming and relies on a decaying sum to forget the oldest statistics progressively. The activities are initialized by making random probing in the search space.
- `CHS` (Conflict-History Search), selects in priority variables appearing in recent failures. All failures are registered with a timestamp. More precisely, `CHS` maintains for each constraint c, a score $q(c)$ and updates it at every domain wipeout with an exponential recency weighted average:
$$q(c) = (1 - \alpha) \times q(c) + \alpha \times r(c)$$
where $\alpha = 0.4$ (decreasing as time goes by) and $r(c)$ is the reward gives when a domain wipeout occurred. Reward is higher when the constraint entered frequently in conflict :
$$r(c) = \frac{1}{\mathtt{\#Conflicts} - \mathtt{Conflict(c)} + 1}$$
`#Conflicts` is the total number of conflicts and `Conflict(c)` stores the last `#Conflicts` value where c led to a failure. The conflict history score (`chv`) of a variable x which will be used in selecting the branching variable is given by:
$$\mathtt{chv}(x) = \frac{\sum_{c \in \mathcal{C}\,:\, x \in \mathtt{scp}(c) \wedge |\mathtt{fut}(c)| > 1} q(c) + \delta}{|\mathtt{dom}(x)|}$$
where δ is a positive real number close to 0 that avoid random selection at the beginning of search. Thus, the branching will be oriented according to the degree of the variables.

To introduce `wdeg` and `dom/wdeg`, we need to describe the way constraint propagation is run each time a decision is taken by the backtrack search algorithm. Algorithm 1 describes

Algorithm 1: propagate($P = (\mathcal{X}, \mathcal{C})$: CN): Boolean

```
1  Q ← C
2  while Q ≠ ∅ do
3      pick and delete c from Q
4      X_evt ← filter(c)   // X_evt is the subset of scp(c) with reduced domains
5      if ∃x ∈ X_evt | dom(x) = ∅ then
6          incrementWeight^VER(c)
7          return false // global inconsistency
8      foreach c' ∈ C | c' ≠ c and X_evt ∩ scp(c') ≠ ∅ do
9          Q ← Q ∪ {c'}
10 return true
```

a basic propagation scheme based on the use of a queue of constraints. Other schemes exists in the literature, but this is not an important issue for introducing constraint weighting. This algorithm is then applied at the beginning of the search and systematically each time a decision is taken. Initially the queue Q contains the whole set of constraints of the constraint network. Then, each constraint c in Q is picked in turn and a filtering process is applied from c: typically, this is for enforcing arc-consistency by calling Function `filter(c)` at Line 4. The call to this function returns a subset of variables of the scope of c, denoted by $X_{\texttt{evt}}$, whose domains have been modified (i.e., such that at least one value has been removed from these domains). By means of $X_{\texttt{evt}}$, we can update Q so as to ensure constraint propagation is run until a fixed point is reached. If ever the domain of one variable of $X_{\texttt{evt}}$ becomes empty, it simply means that a conflict occurred (a dead-end has been identified) and so, a backtrack is required. This is triggered by the returned Boolean value `false`, after having called the function $\texttt{incrementWeight}^{\texttt{VER}}$ with the culprit constraint (responsible of the domain wipeout) passed as a parameter. In the initial paper [2], the principle of constraint weighing is very simple: the weight of the culprit constraint c, denoted by $c.\texttt{weight}$, is incremented by 1, as shown in Algorithm 2 (here, `VER` written as a superscript at Line 6 of Algorithm 1 corresponds to 2004). To summarize, each constraint c admits a weight, initially set to 1, which is incremented whenever a domain wipeout occurs while filtering c.

Algorithm 2: incrementWeight^{2004}(c: Constraint)

```
c.weight ← c.weight + 1
```

Algorithm 3: incrementWeight$^{\texttt{AbsCon}}$(c: Constraint)

```
foreach x ∈ fut(c) do
    c.weights[x] ← c.weights[x] + 1
```

Algorithm 4: incrementWeight$^{\texttt{refined}}$(c: Constraint)

```
foreach x ∈ fut(c) do
    switch VARIANT do
        case ia do
            increment ← 1/|scp(c)|
        case ca do
            increment ← 1/|fut(c)|
        case id do
            increment ← 1/|dom^init(x)|
        case cd do
            increment ← 1/(1+|dom(x)|)
        case ca.cd do
            increment ← 1/(|fut(c)|×(1+|dom(x)|))
    c.weights[x] ← c.weights[x] + increment
```

The heuristics `wdeg` and `dom/wdeg` are defined as follows:

- `wdeg` selects in priority the future variable with the highest 'weighted degree'. Each variable x is given a weighted degree, which is the sum of the weights over all constraints involving x and at least another future variable. For each future variable x, the score of x according to `wdeg` is:
$$\sum_{c \in \mathcal{C} \,:\, x \in \texttt{scp}(c) \wedge |\texttt{fut}(c)|>1} c.\texttt{weight}$$
- `dom/wdeg` selects in priority the future variable with the smallest ratio 'current domain size to weighted degree'. For each future variable x, the score of x according to `dom/wdeg` is:
$$\frac{|\texttt{dom}(x)|}{\sum_{c \in \mathcal{C} \,:\, x \in \texttt{scp}(c) \wedge |\texttt{fut}(c)|>1} c.\texttt{weight}}$$

To break ties, which correspond to sets of variables that are considered as equivalent by the heuristic, one can use a second criterion (e.g., the dynamic degree of each variable). However, for adaptive heuristics, it is usual to use `lexico`, meaning that the first encountered variable with the best score is selected.

IV. Refining Weighted degrees

The heuristic `dom/wdeg` is very simple to be implemented while being quite robust. However, this is not exactly the version that is implemented in the constraint solver `AbsCon`. Indeed, it was observed experimentally that it was more effective to consider only the future variables involved in a culprit constraint. Technically, instead of associating a global weight $c.\texttt{weight}$ with each constraint c, one can introduce a local weight $c.\texttt{weight}[x]$ to be associated with each variable x in $\texttt{scp}(c)$. Hence, when a conflict occurs, instead of incrementing the weight $c.\texttt{weight}$ of the culprit constraint, one can decide to increment the local weight $c.\texttt{weight}[x]$ of each future variable involved in $\texttt{scp}(c)$. Because each variable has now its specific weight in each constraint, the score of a future variable x becomes:

$$\sum_{c \in C\,:\, x \in \mathtt{scp}(c) \wedge |\mathtt{fut}(c)|>1} c.\mathtt{weight}[\mathtt{x}]$$

for `wdeg` and:

$$\frac{|\mathtt{dom}(x)|}{\sum_{c \in C\,:\, x \in \mathtt{scp}(c) \wedge |\mathtt{fut}(c)|>1} c.\mathtt{weight}[\mathtt{x}]}$$

for `dom/wdeg`.

Constraint weighting is now given by Algorithm 3 that describes the function called at Line 6 of Algorithm 1. To distinguish between the 2004 version and the `AbsCon` version, we shall refer to the heuristics of the 2004 initial paper with `wdeg`^{2004} and `dom/wdeg`^{2004}.

Even if `dom/wdeg` slightly outperforms `dom/wdeg`^{2004} (this is shown in Section V), one may regret that constraint weighting remains very simplistic and does not differentiate between constraints. For instance, characteristics like the arity of the constraints and the state of the domains of the participant variables are totally ignored as the increment is static (i.e., 1). This is why we propose to refine constraint weighting by exploiting such information. More specifically, we introduce four variants in Algorithm 4 as follows:

- `ia` is the variant in which the *'initial' arity* of the constraints is used.
- `ca` is the variant in which the *'current' arity* (i.e., the number of future variables) of the constraints is used.
- `id` is the variant in which the (size of the) *initial domains* of the future variables is used.
- `cd` is the variant in which the (size of the) *current domains* of the future variables is used.
- `ca.cd` combines both current arity and current domains.

These different variants are described by Algorithm 4.

V. Experimental Results

We have conducted a first experiment with all available CSP series (82) from the XCSP3 [3] archive (`http://xcsp.org`), which contains 9,243 CSP instances (referred to as ALL). We have also conducted two additional experiments by considering the instances from the main CSP track at the 2017 XCSP3 competition (COMP-17 composed of aim, AllInterval, bdd, Bibd, Blackhole, bmc, bqwh, Cabinet, CarSequencing, ColouredQueens, composed, CostasArray, CoveringArray, Crossword, CryptoPuzzle, DeBruijnSequence, DiamondFree, Domino, driverlogw, Dubois, ehi, Fischer, geometric, gp10, GracefulGraph, Hanoi, Haystacks, jnh, Kakuro, Knights, KnightTour, Langford, LangfordBin, lard, MagicHexagon, MagicSequence, MagicSquare, MarketSplit, mdd, MultiKnapsack, Nonogram, NumberPartitioning, Ortholatin, Pb, pigeonsPlus, Primes, PropStress, QuasiGroup, QueenAttacking, Queens, QueensKnights, qwh, RadarSurveillance, rand, RectPacking, reg, Renault, RenaultMod, Rlfap, RoomMate, Sadeh, Sat, SchurrLemma, SocialGolfers, SportsScheduling, Steiner3, StripPacking, Subisomorphism, Sudoku, SuperQueens, SuperSadeh, SuperTaillard, TravellingSalesman, Wwtpp) and 2018 XCSP3 competition (COMP-18 composed of Bibd, CarSequencing, ColouredQueens, Crossword, Dubois, Eternity, frb, GracefulGraph, Haystacks, LangfordBin, MagicHexagon, MisteryShopper, Pb, QuasiGroup, Rlfap, SocialGolfers, SportsScheduling, StripPacking, Subisomorphism). These instances have been launched on a cluster equipped with 2.66 GHz Intel Xeon and 32 GB RAM nodes. The constraint solver used for our experiments is AbsCon where our new constraint weighting variants and `CHS` have been implemented. The timeout was set to 20 minutes.

TABLE I
COMPARISON OF HEURISTICS IN TERMS OF NUMBER OF SOLVED INSTANCES (#INST.) AND SEVERAL TIME METRICS [COMP-18]

		2004	AbsCon	refined				
				ia	ca	id	cd	ca.cd
wdeg	#inst.	92	96	113	109	107	111	**119**
	c. time	1,937	2,026	1,489	1,606	1,602	**900**	1,117
	by1	62,813	56,995	39,862	45,242	47,304	40,813	**32,207**
	by2	116,813	106,195	68,662	78,842	83,304	72,013	**53,807**
	by10	548,813	499,795	299,062	347,642	371,304	321,613	**226,607**
dom/wdeg	#inst.	91	101	117	110	106	112	**119**
	c. time	1,070	2,131	1,857	**844**	2,129	1,607	924
	by1	66,446	55,773	35,839	40,935	49,120	41,916	**34,304**
	by2	121,646	98,973	59,839	73,335	86,320	71,916	**55,904**
	by10	563,246	444,573	251,839	332,535	383,920	311,916	**228,704**

In Table I, the new constraint weighting variants proposed in that paper are compared with the classical `wdeg` and `dom/wdeg` heuristics (`2004` and `AbsCon` versions). The comparison is given by the number of solved instances (within 1,200 seconds) as well as by several time metrics: the cumulated CPU time (c. time) computed from instances solved by all methods, and the cumulated CPU times (by1, by2, by10) computed from all instance by considering for unsolved instances a 'solving' time equal to $x \times 1,200$ for $x = 1$, $x = 2$ and $x = 10$, respectively. Numbers given in bold face correspond to the best obtained results. In Table I, we can observe that classical heuristics are outperformed by the new variants. Notably, the variant *ca.cd* is clearly the best one as it allows us to solve 18% more instances than the best classical heuristic `dom/wdeg`$^{\mathtt{AbsCon}}$ (119 vs 101). Such results are confirmed by Table II on the CSP 2017 competition instances.

TABLE II
COMPARISON OF HEURISTICS IN TERMS OF NUMBER OF SOLVED INSTANCES (#INST.) AND SEVERAL TIME METRICS [COMP-17]

		2004	AbsCon	refined				
				ia	ca	id	cd	ca.cd
wdeg	#inst.	347	359	363	368	359	367	**369**
	c. time	**4,792**	5,385	5,337	5,931	5,940	6,108	6,085
	by1	58,625	46,404	40,316	38,122	46,543	37,887	**35,156**
	by2	100,625	76,404	65,516	57,322	76,543	58,287	**53,156**
	by10	436,625	316,404	267,116	210,922	316,543	221,487	**197,156**
dom/wdeg	#inst.	345	360	362	360	348	362	**366**
	c. time	**3,573**	4,657	4,549	4,871	5,100	4,499	4,240
	by1	57,203	45,244	**36,867**	43,929	57,077	41,221	38,449
	by2	104,003	74,044	60,867	72,729	99,077	67,621	**60,049**
	by10	478,403	304,444	252,867	303,129	435,077	278,821	**232,849**

In Table III, we provide some details about specific series. Due to lack of space, we decided to only keep c. time as time metric because we find it to be the most relevant one. For the lack of clarity, some series have been discarded from this table because we obtained rather similar results whatever the heuristic is used. However, note that these series are taken into account when displaying the total number of solved instances

TABLE III

COMPARISON OF HEURISTICS IN TERMS OF NUMBER OF SOLVED INSTANCES (#INST.) AND CUMULATED CPU TIME (C. TIME) [ALL]

	$\texttt{wdeg}^{2004}$		$\texttt{wdeg}^{\texttt{AbsCon}}$		$\texttt{dom/wdeg}^{2004}$		$\texttt{dom/wdeg}^{\texttt{AbsCon}}$		$\texttt{wdeg}^{\texttt{ca.cd}}$	
	#inst	c. time	#inst	c. time	#inst	c. time	#inst	c. time	#inst	c. time
AllInterval	5	1,094	14	397	**15**	**20**	15	845	**15**	**20**
bdd	48	**584**	48	926	48	**583**	48	1,028	48	952
Bibd	30	1,827	82	623	27	1,019	**88**	967	84	209
Blackhole	1	0	**20**	0	0	0	14	**0**	**20**	0
bmc	12	3,716	16	195	12	4,022	16	250	**16**	**180**
Cabinet	20	131	20	138	20	177	20	283	20	**115**
CarSequencing	31	248	37	383	19	872	30	565	**48**	446
ColouredQueens	0	0	**1**	0	0	0	0	0	**1**	0
CostasArray	3	374	**5**	89	4	186	4	825	4	169
CoveringArray	1	207	2	73	**4**	4	**4**	3	3	3
Crossword	169	11,852	157	18,902	**190**	3,631	185	4,260	176	11,628
DeBruijnSequence	5	516	6	495	**6**	**358**	6	396	5	805
DiamondFree	17	2,205	**22**	349	18	3,242	**22**	493	**22**	**208**
DistinctVectors	3	891	3	**32**	3	1,057	3	96	3	34
Dubois	8	81	19	36	8	120	6	435	**20**	60
frb	19	2,417	20	2,355	28	2,672	**29**	2,661	19	2,547
GracefulGraph	4	281	7	59	8	204	7	50	**8**	**28**
Knights	7	438	7	520	**7**	**309**	7	426	5	683
KnightTour	17	20	16	25	4	1,299	10	37	**18**	16
Langford	11	950	**12**	**704**	11	1,082	12	1,035	12	898
LangfordBin	1	6	1	36	1	5	1	13	**10**	3
MagicHexagon	3	89	9	67	9	7	**18**	10	16	**7**
MagicSequence	14	**819**	14	2,542	14	1,780	14	1,660	14	2,137
MagicSquare	11	626	21	407	32	163	**43**	58	41	150
MarketSplit	**10**	**396**	7	1,212	10	449	9	620	9	624
mdd	**33**	3,476	29	4,922	32	3,298	27	3,604	29	4,984
MultiKnapsack	11	64	9	416	**11**	**52**	9	1,066	10	86
NumberPartitioning	**38**	**258**	38	588	15	494	29	3,926	38	501
Ortholatin	4	28	**4**	**16**	2	5	2	8	3	29
Pb	3	299	4	705	3	117	4	105	**6**	146
pigeonsPlus	13	3,012	14	2,178	15	1,583	**15**	**1,466**	15	1,489
Primes	8	60	15	80	13	209	**18**	94	16	239
qcp	11	808	11	334	11	622	**13**	**480**	13	578
QuasiGroup	4	327	4	476	5	125	**6**	210	5	248
QueensKnights	**9**	**147**	**9**	150	5	500	6	387	8	206
qwh	43	8,303	43	4,291	51	3,487	47	5,178	**52**	3,876
RadarSurveillance	40	1,798	40	1,688	40	2,483	40	2,348	**41**	1,922
Rlfap	6	652	**7**	533	6	518	**7**	**397**	7	584
RoomMate	13	5,457	14	5,239	11	5,740	14	5,577	**14**	**5,167**
SocialGolfers	59	1,744	56	2,105	44	3,916	51	1,299	**61**	1,207
StripPacking	2	0	5	0	0	0	3	0	**7**	0
Subisomorphism	206	2,241	162	18,193	204	4,616	200	2,380	**211**	2,099
SuperQueens	1	657	1	739	1	292	1	**282**	1	892
SuperSadeh	10	163	**11**	245	9	225	9	612	9	182
SuperTaillard	39	2,258	37	1,960	**41**	3,847	37	2,570	37	2,386
TravellingSalesman	18	988	18	1,157	18	**281**	18	339	18	1,262
Wwtpp	229	5,562	**244**	2,119	243	9,380	242	15,269	240	2,049
Total	1,465	75,375	1,559	84,436	1,486	74,116	1,617	72,547	**1,694**	**60,482**

(last line of the table). For the remaining series, we also discarded 'easy' instances, which are CSP instances solved by all heuristics by less than 10 seconds. In each row, the highest number of solved instances is written in bold, except when all heuristics solve the same instances, in which case the c.time is given in bold. Once again, we can observe that $\texttt{wdeg}^{ca.cd}$ is the best variant.

Figure 1 shows a scatterplot allowing us to compare the overall respective behavior of $\texttt{dom/wdeg}^{\texttt{AbsCon}}$ and $\texttt{wdeg}^{\texttt{ca.cd}}$. For our comparison, we used the set COMP-17+18 containing instances coming from both the 2017 and 2018 XCSP instances (main CSP track). Each dot in this figure represents a CSP instance. The coordinates of this dot are defined by: on the horizontal axis, the CPU time (in seconds) required to solve the instance with $\texttt{dom/wdeg}^{\texttt{AbsCon}}$ and on the vertical axis, the CPU time required to solve the instance with $\texttt{wdeg}^{\texttt{ca.cd}}$. One can observe that more instances are located at the bottom-right side of the figure, meaning that $\texttt{wdeg}^{\texttt{ca.cd}}$ is usually more efficient. Also, note the presence of many dots along the right border, indicating that these instances have not been solved (within $1,200$ seconds) by the classical heuristic $\texttt{dom/wdeg}^{\texttt{AbsCon}}$. The same trend can be observed in Figure 2, when comparing CHS and $\texttt{wdeg}^{\texttt{ca.cd}}$, even if results are closer. When comparing these two heuristics

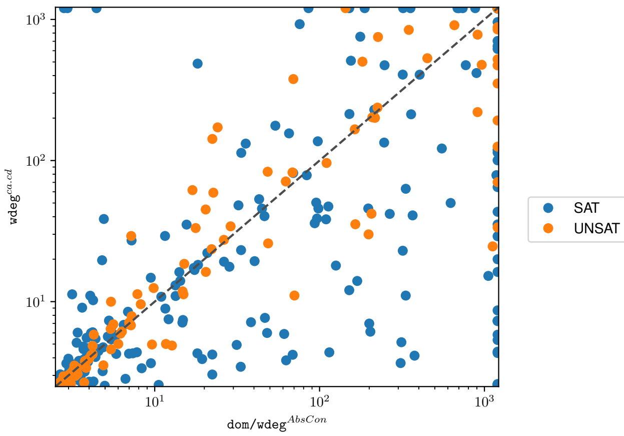

Fig. 1. Comparing dom/wdeg$^{\text{AbsCon}}$ and wdeg$^{ca.cd}$ [COMP-17+18]

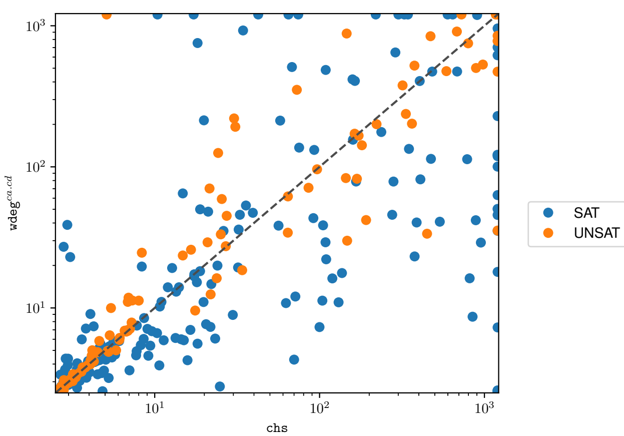

Fig. 2. Comparing CHS and wdeg$^{\text{ca.cd}}$ [COMP-17+18]

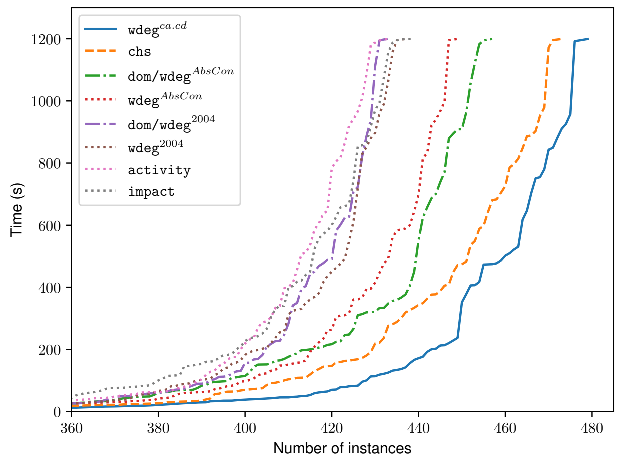

Fig. 3. Comparing popular heuristics and wdeg$^{\text{ca.cd}}$ [COMP-17+18]

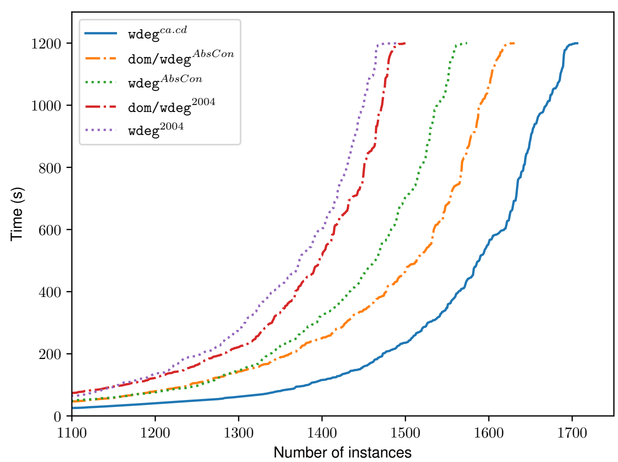

Fig. 4. Comparing popular heuristics and wdeg$^{\text{ca.cd}}$ [ALL]

on the overall set of instances (ALL), the c. time of wdeg$^{\text{ca.cd}}$ is decreased by 32% and 59 additional instances are solved.

The cactus plot in Figure 3 shows the performance of all popular generic heuristics and wdeg$^{\text{ca.cd}}$ on COMP-17+18. It displays the number of solved instances (on the horizontal axis) per unit of time (on the vertical axis). On the left of the figure, we can find the least effective heuristics, namely, impact, activity, wdeg^{2004} and dom/wdeg^{2004} that behave rather similarly. In the middle of the figure, we have wdeg$^{\text{AbsCon}}$ and dom/wdeg$^{\text{AbsCon}}$, as implemented (and used by default) in AbsCon. Finally, CHS and wdeg$^{\text{ca.cd}}$ are clearly the most efficient heuristics since they are situated on the right.

Figure 4 focuses on constraint weighting variants (comparing very classical heuristics with our new best variant ca.cd). Clearly, wdeg$^{ca.cd}$ appears to be the most robust heuristic based on constraint weighting.

VI. Conclusion

In this paper, we have revisited constraint weighting that is known to be a robust generic approach to guide backtrack search. By refining the way weights of constraints (and variables) are updated by taking into account both constraint arities and sizes of variable domains, we show how the popular heuristic dom/wdeg can be improved. We think that dom/wdeg$^{ca.cd}$ is the most robust generic (variable ordering) heuristic to be used for solving instances of constraint satisfaction problems.

Acknowledgment

The authors are grateful to the anonymous reviewers for their comments. This work has been supported by the project CPER Data from the region "Hauts-de-France".

References

[1] T. Balafoutis and K. Stergiou. On conflict-driven variable ordering heuristics. In *Proceedings of CSCLP'08*, 2008.

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

[3] F. Boussemart, C. Lecoutre, and G. Audemard C. Piette. XCSP3: an integrated format for benchmarking combinatorial constrained problems. *CoRR*, abs/1611.03398, 2016.

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

[5] D. Grimes and R.J. Wallace. Learning to identify global bottlenecks in constraint satisfaction search. In *Proceedings of FLAIRS'07*, 2007.

[6] Diarmuid Grimes and Emmanuel Hebrard. Solving variants of the job shop scheduling problem through conflict-directed search. *INFORMS Journal on Computing*, 27:268–284, 04 2015.

[7] Djamal Habet and Cyril Terrioux. Conflict history based search for constraint satisfaction problem. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, pages 1117–1122, New York, NY, USA, 2019. ACM.

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

[9] Emmanuel Hebrard and Mohamed Siala. Explanation-based weighted degree. pages 167–175, 05 2017.

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

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

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

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

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