Artificial Intelligence Review (2021) 54:5055–5093
https://doi.org/10.1007/s10462-021-10012-4

# Adaptive constraint propagation in constraint satisfaction: review and evaluation

Kostas Stergiou[1]

Published online: 11 May 2021

**Abstract**
Several methods for dynamically adapting the local consistency property applied by a CP solver during search have been put forward in recent and older literature. We propose the classification of such methods in three categories depending on the level of granularity where decisions about which local consistency property to apply are taken: *node*, *variable*, and *value* oriented. We then present a detailed review of existing methods from each category, and evaluate them theoretically according to several criteria. Taking one recent representative method from each class, we then perform an experimental study. Results show that simple variable and value oriented methods are quite efficient when the older dom/ddeg heuristic is used for variable ordering, while a carefully tuned node oriented method does not seem to offer notable improvement compared to standard arc consistency propagation. In contrast, under the more realistic setting of dom/wdeg, the variable and value oriented methods cannot compete with standard propagation, while the node oriented method is very efficient. Finally, we obtain a new adaptive propagation method by integrating the variable and value oriented approaches and adding an amount of randomization The resulting method is simple, competitive, and almost parameter-free.

**Keywords** Constraint propagation · Adaptive propagation · Experimental evaluation

## 1 Introduction

Constraint propagation is a cornerstone of Constraint Programming (CP) and a major reason for its success in dealing with difficult combinatorial problems. CP solvers typically propagate constraints using algorithms that achieve (generalized) arc consistency (GAC). Properties stronger than GAC have received a fair bit of attention because their application can cut down the size of the search tree by orders of magnitude in many cases, but they are rarely used within solvers because they are considered too expensive to be maintained during search.

The classes of strong local consistencies that have been the most widely studied and evaluated are triangle based consistencies for binary constraints, e.g. path consistency and

✉ Kostas Stergiou
kstergiou@uowm.gr

[1] University of Western Macedonia, Kozani, Greece

its variants, relation based consistencies for non-binary constraints, such as pairwise consistency, relational k-consistency and their variants, and singleton consistencies for both binary and non-binary constraints.

The latter class has received the most attention, with singleton arc consistency (SAC) (Debruyne and Bessière 2001) being its prime member, because such local consistencies are independent of the constraints' arity and they are quite simple, both conceptually and in terms of implementation within CP solvers. Although several variants of SAC have been proposed that are either weaker (NSAC Wallace 2015; RNSAC Paparrizou and Stergiou 2017) or stronger (POAC Bennaceur and Affane 2001), but in any case, more efficient in practice than SAC, still none of these variants, or any other strong local consistency, has proved to be competitive with GAC over a wide range of problems. This is because the overhead of maintaining such properties throughout search usually outweighs the benefits of the reduction in the size of the search tree.

The main problem with the application of strong local consistencies during search is that their repeated, and costly, invocation is often fruitless, i.e. it does not achieve any extra pruning compared to standard GAC propagation, or it achieves little extra pruning. Because of this, an avenue of research that has attracted considerable interest, initially in the 90s and then in the last decade or so, is that of adaptive propagation. Such works have proposed various heuristic methods that exploit dynamic features of the problem at hand to selectively apply strong propagation algorithms during search, at points where it will hopefully pay off. This is implemented typically by switching between strong and standard propagation when certain conditions are met (Stergiou 2008; Balafrej et al. 2013, 2014, 2015; Woodward et al. 2018). In this way, the cost of maintaining a strong local consistency throughout search is avoided, while much of its pruning power is kept.

Although quite a few adaptive propagation methods have been proposed, the relevant literature is rather fragmented and there is considerable diversity in the ideas behind the various methods. In addition, there exists only one research work where an experimental evaluation of some adaptive propagation methods is presented (Woodward et al. 2018). In this paper we attempt to clarify and organize the state-of-the-art in adaptive propagation by proposing a classification of existing methods into three categories depending on the level of granularity where decisions about the local consistency to be applied are taken.

We identify a class of methods that make a decision on how to perform constraint propagation after each branching decision, i.e. at each node of the search tree. We call such methods *node oriented*. Another class of methods makes such decisions at a lower level of granularity. Namely, such methods use heuristics to decide which local consistency to apply on individual variables. We call such methods *variable oriented*. Finally, the third class of methods that we identify operates on a even lower level of granularity. Such methods check the consistency of specific values from the domains of some variables using a stronger than standard local consistency property. We call such methods *value oriented*.

We review the existing adapting propagation methods of all three categories and discuss their properties according to certain criteria, such as their generality, e.g. the extent of their applicability on constraints of different types and different solver settings, and their dependency on the fine tuning of parameters. Taking one outstanding representative from each class, we then perform an experimental evaluation on benchmark problems, albeit limited to binary constraints.

The results depend heavily on the variable ordering heuristic used. Under dom/ddeg, an older heuristic that does not interfere too much with propagation, the simple variable and value oriented methods of Stergiou (2008) and Balafrej et al. (2013) achieve a good balance between a standard solver that maintains arc consistency (MAC) and algorithms that maintain singleton consistencies throughout search. On the other hand, the performance of PrePeak, the carefully tuned node oriented method of Woodward et al. (2018), is very close to that of MAC, making it very inefficient on some problems. However, under the state-of-the-art dom/wdeg heuristic, results are quite different. The variable and value oriented methods offer an advantage over MAC on only a few problems, but are heavily outperformed on the majority, while PrePeak at least matches MAC on problems where MAC excels, but manages to notably improve on its performance on the few problems where MAC is less efficient.

As a final contribution, we propose a simple variant of the variable-oriented method of Stergiou (2008) that probabilistically decides whether to perform singleton tests or not during search. The main idea is to monitor, through very simple book-keeping, the effect that the revision of each variable has on its domain. But instead of relying on a user-defined threshold to decide whether to apply a strong propagator on a variable $x$ or not, as in Stergiou (2008), the decision is taken using a probability distribution that depends on the distance between the current revision of $x$ and the most recent singleton test on $x$ that resulted in a domain wipe-out.

Experimental results demonstrate that the randomized technique significantly improves the performance of the heavily parameter-dependant method it originates from, and when integrated with the value-oriented method of Balafrej et al. (2013), it gives a simple and nearly parameter-free adaptive technique that is very competitive with PrePeak.

The paper is organized as follows: Sect. 2 gives the necessary background. In Sect. 3 we describe the framework of a CP solver, highlighting where decisions about which local consistency to apply can be taken. Section 4 proposes the classification of adaptive propagation methods into three categories and presents a detailed review of the existing literature. In Sect. 5 we present the experimental evaluation of three selected adaptive methods, while in Sect. 6 we discuss our enhancement of an existing variable oriented method and evaluate it experimentally. Section 7 makes some generic observations based on our study and points to future work. Finally, in Sect. 8 we conclude.

## 2 Background

A finite domain *Constraint Satisfaction Problem* (CSP) $\mathcal{P}$ is defined as a triplet $\Leftarrow\mathcal{X}\Leftrightarrow\mathcal{D}\Leftrightarrow\mathcal{C}\Rightarrow$ where:

- $\mathcal{X} = \{x_1, \dots, x_n\}$ is a set of $n$ variables,
- $\mathcal{D} = \{D(x_1), \dots, D(x_n)\}$ is a set of finite domains, one for each variable in $\mathcal{X}$,
- $\mathcal{C} = \{c_1, \dots, c_e\}$ is a set of $e$ constraints. Each constraint $c_i \in \mathcal{C}$ is a pair $\langle sc(c_i), rel(c_i)\rangle$, where $sc(c_i) \subseteq X$ is the set of $k$ variables $\{x_{i_1}, \dots x_{i_k}\}$ involved in the constraint, known as the *scope* of the constraint, and $rel(c_i) \subset D(x_{i_1}) \times \cdots \times D(x_{i_k})$ is a relation giving the allowed tuples of the constraint.

The relation of a constraint can be defined by explicitly listing its allowed tuples, in which case it is called a *table constraint* or implicitly through a predicate or function, as is the case with *global constraints*. The cardinality $k$ of $sc(c_i)$ is the *arity* of the constraint.

A binary CSP $\mathcal{P}$ is typically depicted as a constraint graph $\mathcal{G}$, where variables correspond to nodes and constraints to edges. Sometimes, the constraint graph is considered to be directed, in which case an *arc* is a directed constraint. That is, each binary constraint between two variables $x_i$ and $x_j$ corresponds to two arcs in the graph. In the following, we will consider constraints to be undirected, unless otherwise stated, and a binary constraint between variables $x_i$ and $x_j$ will be denoted by $c_{ij}$.

A variable $x_j$ is a *neighbour* of a variable $x_i$ iff $c_{ij} \in \mathcal{C}$. Let $N_i \subset \mathcal{X}$ denote the set of variables that are neighbours of $x_i$. The *neighbourhood* $\mathrm{N}(x_i)$ of variable $x_i$ is the sub-graph of $\mathcal{G}$ that is induced by $N_i \cup \{x_i\}$. To put it in words, $\mathrm{N}(x_i)$ includes $x_i$, all neighbours of $x_i$, any constraint between $x_i$ and one of its neighbours, and any constraint between two neighbours of $x_i$.

Complete algorithms for CSPs are based on exhaustive backtracking depth-first search interleaved with constraint propagation, which typically involves enforcing a local consistency property on the constraints of the problems. Search is guided by variable and value ordering heuristics. After a branching decision (i.e. a variable assignment or a value removal from a domain) propagation kicks off. If this results in an empty domain, in which case we have a *domain wipe-out* (DWO), the search algorithm rejects the most recent branching decision and moves on to the next one.

A *solution* to a CSP is a *complete assignment*, i.e. an assignment involving all variables, that satisfies all constraints in $\mathcal{C}$. A *consistent partial assignment* is an assignment to a set $S \subseteq \mathcal{X}$ of variables that satisfies all constraints among the variables in $S$. The search process can be visualized by a traversal of a search tree where the root corresponds to the empty assignment (no variable has been assigned yet) and the rest of the nodes correspond to consistent partial assignments.

## 2.1 Local consistencies

The local consistency (LC) that is predominantly used during propagation is *arc consistency* Mackworth (1977). For binary problems, a value $a_i \in D(x_i)$ is *Arc Consistent* (AC) iff for every constraint $c_{ij}$ there exists a value $a_j \in D(x_j)$ s.t. the pair of values $(a_i, a_j)$ satisfies $c_{ij}$. In this case $a_j$ is a *support* for $a_i$ on $c_{ij}$. The extension of AC to non-binary constraints is known as *Generalized Arc Consistency* (GAC). A value $a \in D(x_i)$ is GAC iff for every constraint $c_j$ with arity $k$, s.t. $x_i \in sc(c_j)$, there exists a $k$-tuple in $rel(c_j)$ that includes the assignment of $a$ to $x_i$. Such a tuple is a support for $a$ on $c_j$. A variable is (G)AC iff all its values are (G)AC. A problem is (G)AC iff there is no empty domain in $\mathcal{D}$ and all the variables in $\mathcal{X}$ are (G)AC.

Forward checking (FC) is a restricted form of AC that was widely used in the past for constraint propagation during search (Haralick and Elliot 1980). After each assignment of a variable $x_i$, FC enforces AC only on constraints involving $x_i$.

A variable-oriented propagation scheme utilizes a queue of variables to perform propagation. When a variable $x_i$ is removed from the queue then for each constraint $c$ having $x_i$ in its scope, all the other variables in the scope of $c$ are *revised*, i.e. the values in their domains are checked for consistency. If (G)AC is the LC used then for any $x_j \in sc(c)$, this involves searching for a supporting tuple for the values of $x_j$ in $rel(c)$. If the constraint is binary then this essentially means searching for support in $D(x_i)$. In such a case, the first support for a value $a_j \in D(x_j)$ found in $D(x_i)$ will be denoted as $\sup(a_j, x_i)$. If no support is found then $a_j$ is removed from $D(x_j)$ and $x_j$ is inserted in the queue to further propagate the effects of the value deletions. If all values are removed from a domain, we have a DWO.

A constraint-oriented propagation scheme uses a queue of constraints instead. In this case, when a constraint is removed from the queue, we say that the constraint is *revised*, meaning that all of the variables in its scope are revised. Modern CP solvers use one or more queues which may hold variables, constraints, or propagators (i.e. filtering algorithms for specific constraints triggered by value removals).

## 2.2 Strong local consistencies

Local consistencies that achieve stronger domain pruning than (G)AC have also been considered. We call such consistencies *strong local consistencies* (SLCs) hereafter. One class of SLCs that has received attention in the case of binary constraints is the class of *triangle based* consistencies which consider triplets of variables when checking the consistency of a value, instead of pairs as AC does. Members of this class include the well known *Path Consistency* (PC) (Montanari 1974) and its variants: *Restricted Path Consistency* (RPC) (Berlandier 1995), *Path Inverse Consistency* (PIC) (Freuder and Elfe 1996) and *max Restricted Path Consistency* (maxRPC) (Debruyne and Bessière 1997).

In the case of non-binary constraints, SLCs have been mainly proposed for table constraints, either as extensions of binary SLCs such as RPC and maxRPC (Bessière et al. 2008; Stergiou (2008), or as variants of *pairwise* (Janssen et al. 1989) and *relational* consistency (Dechter and van Beek 1997) (e.g. Karakashian et al. 2010; Woodward et al. 2011; Lecoutre et al. 2012). Other notable works include a method that adds new (factor) variables to achieve SLC reasoning through GAC (Likitvivatanavong et al. 2014) and the introduction of SLCs based on bounds consistency (Bessiere et al. 2015).

However, the class of SLCs that has been predominantly studied is that of *singleton consistencies* (Prosser et al. 2000), which is also the most interesting class in the context of adaptive propagation, as we will shortly explain. The most well known singleton consistency is of course *Singleton Arc Consistency* (SAC) (Debruyne and Bessière 2001). For any variable $x_i$ and value $a_i \in D(x_i)$, let AC($\mathcal{P}_{x_i \leftarrow a_i}$) be the resulting problem after restricting $D(x_i)$ to $a_i$ and applying AC to $\mathcal{P}$ (in a process called a *singleton test*). Value $a_i$ is SAC iff AC($\mathcal{P}_{x_i \leftarrow a_i}$) has no empty domain. A problem is SAC iff all values in all domains are SAC. SAC is generalized to non-binary constraints in a straightforward way.

Several variants of SAC, that achieve either weaker or stronger pruning, have been proposed and studied since its introduction to CP. *Neighbourhood* SAC (NSAC) is a weaker variant of SAC which, after restricting $D(x_i)$ to $a_i$, applies AC only to N($x_i$) (Wallace 2015). *Restricted* NSAC (RNSAC) makes a singleton test on a value $a_i$ of $x_i$ (again applying AC to N($x_i$)) only if $a_i$ has a single support in the domain of at least one variable in N($x_i$) (Paparrizou and Stergiou 2017).

Whereas SAC can only remove values from the domain of the variable that is being singleton tested, a stronger variant of SAC, called Partition-One AC (POAC) can also remove values from other domains (Bennaceur and Affane 2001). A value $a_i \in D(x_i)$ is POAC iff it is SAC and $\forall x_j \in \mathcal{X}$, there exists $a_j \in D(x_j)$ s.t. $a_i$ belongs to a domain in AC($\mathcal{P}_{x_j \leftarrow a_j}$). Hence, if all the singleton tests on the values in a domain $D(x_j)$ result in the removal of a value $a_i \in D(x_i)$ then POAC will remove $a_i$ from $D(x_i)$. In the spirit of NSAC, a neighbourhood variant of POAC (called NPOAC) has also been defined (Woodward et al. 2017). And we can easily define *Restricted* NPOAC following the definition of RNSAC.

SAC and its variants have certain advantages and disadvantages. Their main advantages are that they are conceptually simple, they can be quite easily integrated into solvers, and that they can be applied on problems including constraints of any arity. On the other hand, despite the progress in SAC algorithms (Bessiere et al. 2011; Balafrej et al. 2014), they are typically very expensive to maintain during search. (R)NSAC and POAC are promising in that respect, as the experiments carried out so far demonstrate that they are more efficient than SAC and can potentially be successfully applied before or during search on some problems (Wallace 2015, 2016; Paparrizou and Stergiou 2017; Bennaceur and Affane 2001).

Importantly, singleton consistencies are particularly suitable for use within adaptive propagation methods. This is because they are easily implementable "on top" of the standard propagation technique of the solver, something that is not the case with triangle or relation based consistencies. Hence, as we will explain, many of the recent adaptive methods have been proposed specifically as means to harness the pruning effects of singleton consistencies without paying the cost of maintaining them throughout search.

Following Debruyne and Bessière (2001), a consistency property $A$ is *stronger* than $B$ iff in any problem in which $A$ holds then $B$ holds, and *strictly stronger* iff there is at least one problem in which $B$ holds but $A$ does not. A local consistency property $A$ is incomparable with $B$ iff $A$ is not stronger than $B$ nor vice versa. For example, according to these definitions, POAC is strictly stronger than SAC which in turn is strictly stronger than NSAC. On the other hand, NPOAC is incomparable to SAC.

## 2.3 Heuristics and branching schemes

Variable ordering heuristics play a very important role in CP. A variable ordering heuristic that was widely used in the past is *dom/ddeg* (i.e. domain size over dynamic degree) (Bessière and Régin 1996). The degree of a variable is the number of constraints it participates in. The dynamic degree is the number of constraints it participates in, such that for each constraint, at least one other variable involved in the constraint has not been assigned yet. The heuristic *dom/ddeg* chooses to assign the variable having minimum ratio of current domain size over dynamic degree.

A considerably more effective variable ordering heuristic is *dom/wdeg*, which is one of the most efficient general-purpose heuristics for CSPs (Boussemart et al. 2004). This heuristic assigns a weight to each constraint, initially set to one. Each time a constraint causes a DWO, its weight is incremented by one. Each variable is associated with a *weighted degree* (wdeg), which is the sum of the weights over all constraints involving the variable and at least another (unassigned) variable. The *dom/wdeg* heuristic chooses the variable with minimum ratio of current domain size to weighted degree.

Apart from the variable ordering heuristic, another factor that determines how the search tree is explored is the branching scheme of the solver[1]. The two most widely used branching schemes are *binary* (also called 2-way) and $d$-way branching. In binary branching, after a variable $x_i$ is chosen and a value $a_i \in D(x_i)$ is selected, two branches are created (Sabin and Freuder 1997). In the left branch $a_i$ is assigned to $x_i$ and then propagation is triggered. In the right branch $a_i$ is removed from $D(x_i)$ and again propagation is triggered. If the propagation of $a_i$'s assignment fails and the propagation of $a_i$'s removal succeeds then any variable can be selected next (not necessarily $x_i$). If both branches fail then the algorithm backtracks.

In $d$-way branching, after variable $x_i$ is selected, $d$ branches are built, each one corresponding to one of the $d$ possible value assignments of $x_i$. If the branch corresponding to the assignment of a value $a_i$ to $x_i$ fails, the next available value assignment to $x_i$ is tried (next branch), and so on. If all $d$ branches fail then the algorithm backtracks. Although

[1] Of course, value ordering also plays a part, but to a much lesser degree than variable ordering.

2-way branching is considered more efficient than *d*-way, this is not always true, especially in the case of binary problems.

## 3 Basic search and propagation framework

Before we start our review of the literature, we describe the framework of a basic CP solver, highlighting the points where decisions about which LC to apply can be taken. This allows for the identification of three categories of adaptive propagation methods, as we explain below.

We consider as *adaptive propagation method* any technique that can decide at certain points during the search and propagation process to apply a propagation technique different than the standard one, either on the entire problem or on parts of it. These decisions are taken by exploiting information that becomes available dynamically during search. And typically, such a decision involves the application of a SLC algorithm instead of (G)AC, but other approaches have also been considered, especially in early works.

We make certain assumptions regarding the operation of the solver:

- Our description follows a typical setting where constraint propagation involves running an (G)AC3-like algorithm that seeks a support for each value of a variable on a constraint. In the context of non-binary constraints, including global ones, propagation is implemented in this way only for specific cases, as most constraints come with their own specialized filtering algorithm. Such algorithms do not seek for supports for individual values but exploit the semantics of the constraint to filter inconsistent values "as they discover them". Below we will explain if and how the existing adaptive methods are applicable to the case of non-binary constraints with specific semantics.
- Our description of the solver's framework follows a standard d-way branching scheme. This allows for the easier identification and description of the components where adaptive propagation can be used. However, almost all of the adaptive methods in the literature are applicable independent of the branching scheme. If a method is tied to a specific branching scheme then this will be clarified in its description below.
- We assume that propagation is variable oriented. That is, the main data structure at the center of the propagation mechanism is a queue of variables. Once a variable has its domain pruned, it is inserted in the queue. Variables are iteratively removed from the queue and their domain pruning is propagated through the constraints involving them, possibly resulting in new domain reductions and queue insertions. This process continues until the queue becomes empty or a domain is wiped out.
- We assume that the standard local consistency used to propagate all the constraints is (G)AC. This is not a restrictive assumption as modern adaptive propagation methods typically alternate between the standard local consistency, which indeed is (G)AC in all solvers, and stronger local consistencies.

### 3.1 Search

Algorithm 1 gives the basic framework of a backtracking-based CP solver. In line 1 all constraints are propagated with the initial variable domains (a process known as preprocessing). If no DWO occurs, search commences by heuristically selecting the variable at depth 1. While there is no backtrack to level 0, signalling that the search space had been exhaustively explored and no solution has been found, the solver traverses the space of partial variable assignments until a complete one (a solution) has been found.

**Algorithm 1** *Search*($\mathcal{X}, \mathcal{D}, \mathcal{C}$)

```
1: if $Propagate(\mathcal{X}, \mathcal{D}, \mathcal{C}, null)$ = FALSE then
2:    return FALSE;
3: $depth \leftarrow 1$;
4: select an unassigned variable $x_{depth}$;
5: while $depth \geq 1$ do
6:    if all values in $D(x_{depth})$ have been tried then
7:       $depth \leftarrow depth-1$;
8:    else
9:       select a value $a \in D(x_{depth})$ that has not been tried;
10:      $D(x_{depth}) \leftarrow \{a\}$;
11:      if $Propagate(\mathcal{X}, \mathcal{D}, \mathcal{C}, x_{depth})$ = FALSE then
12:         restore domains;
13:      else
14:         $depth \leftarrow depth+1$;
15:         if $depth = n + 1$ then
16:            return TRUE;
17:         select an unassigned variable $x_{depth}$;
18: return FALSE;
```

At each iteration of the **while** loop a value $a$ is selected and assigned to the current variable $x_{depth}$ (lines 9–10). If all values in $D(x_{depth})$ have been tried then a chronological backtrack is triggered (lines 6–7). Otherwise, the assignment of $a$ is propagated, typically by applying (G)AC on the resulting problem. This is the first point where a decision about the local consistency to be used during propagation can be taken. That is, before calling function *Propagate*, an adaptive propagation method may apply some heuristic to decide whether the propagation of the assignment at this node of the search tree will be performed using the standard technique of the solver or some other (stronger) local consistency. We call such a method *node oriented.*

If propagation fails (i.e. a DWO occurs) then the affected domains are restored to their previous state and the solver moves on to try the next value of $x_{depth}$, if one is available, at the next iteration of the loop. Otherwise, search moves on to the next level of the search tree by increasing the depth, and a new variable is selected for instantiation, unless we have reached a complete assignment.

An advantage of node oriented adaptive propagation methods is that they are independent of the constraints' arity and type. This makes them, at least in principle, generic and applicable within a wide range of CP solvers. On the other hand, as the decision about how to propagate constraints is taken at a high level (i.e. after a branching decision), this decision is uniform, in the sense that all constraints will be propagated using the chosen LC, and therefore there is less flexibility. This will be elaborated further when we discuss the existing node oriented methods from the literature.

## 3.2 Propagation

Algorithm 2 gives the basic framework of variable oriented propagation. It is called with the currently assigned variable $x_i$ as parameter. Q denotes the queue where variables pending propagation are inserted. It is initialized by inserting $x_i$ into it, unless we are at the preprocessing phase, in which case all variables are inserted into Q.

**Algorithm 2** *Propagate*$(\mathcal{X}, \mathcal{D}, \mathcal{C}, x_i)$

```
1: if $x_i = null$ then
2:     Q ← $\mathcal{X}$;
3: else
4:     Q ← $\{x_i\}$;
5: while Q $\neq \emptyset$ do
6:     remove variable $x_j$ from Q;
7:     for each $c \in \mathcal{C}$, s.t. $x_j \in sc(c)$ do
8:         for each $x_k \in sc(c)$, with $k \neq j$ do
9:             deletion ← FALSE;
10:            if $Revise(x_k, c, deletion)$ = FALSE then
11:                return FALSE;
12:            if deletion = TRUE then
13:                Q ← Q ∪ $\{x_k\}$;
14: return TRUE;
```

The main loop picks a variable $x_j$ from the queue (line 6) and for each constraint $c$ that includes $x_j$ in its scope, it *revises* the domains of the other variables that are involved in the constraint by filtering values that are now inconsistent (i.e. have no support on $c$) (lines 7–13). This is the second place where a decision about the LC to be used during propagation can be taken. That is, an adaptive propagation method may apply some heuristic to decide whether the revisions of the constraints involving a variable $x_j$ picked from the queue will be performed using the standard technique of the solver or some other (stronger) local consistency.

Depending on the method, this decision may be taken for each constraint $c$ involving $x_j$, which means that it is taken after line 7 and before line 8 and it is uniform for all the variables in the scope of $c$, or it may be taken for each individual variable in the scope of $c$, which means that it is taken after line 8. For the case of binary constraints where each constraint $c$ involves only one variable apart from $x_j$, these two approaches are of course identical. We collectively call such methods *variable oriented*.

For each variable $x_k$ appearing in a constraint together with $x_j$, the revision of its domain is carried out by the *Revise* function. This function returns FALSE if the domain of $x_k$ is wiped out. Otherwise, if there is at least one deletion from the domain, it sets the flag *deletion* to TRUE to signal that $x_k$ must be inserted into Q to continue propagation.

Variable oriented adaptive propagation methods take decisions at a lower level of granularity in the search/propagation process than node oriented ones. This offers greater flexibility because in contrast to node oriented methods that propagate everything using the same local consistency at each node of search, variable oriented methods may choose to propagate constraints for any individual variable removed from Q using either the standard technique or a stronger one.

Variable oriented methods are also independent of the branching scheme and the arity of the constraints. However, they may require to be modified or they may not even be applicable within solvers that do not use a queue of variables to implement propagation. More on this when we discuss the individual methods below. Also, variable oriented methods may need alterations in their implementation depending on the SLC used as the alternative to GAC. Again, this is further explained below.

### 3.3 Revision

Algorithm 3 gives the basic framework of the revision process for a variable $x_k$ and a constraint $c$ involving this variable. It checks the consistency for all values in $D(x_k)$ and removes inconsistent ones. This is done through function *Consistent* which may be implemented in different ways depending on the type of constraint and the LC used to check the consistency. For example, for a binary constraint and AC, for any value $a \in D(x_k)$ it simply searches for a support in $D(x_j)$. This is the third point where a decision about the LC to be used during propagation can be taken. That is, an adaptive propagation method may apply some heuristic to decide whether the test for the consistency of a value $a \in D(x_k)$ will be performed using the standard technique of the solver or some other (stronger) local consistency. We call such a method *value oriented.*

**Algorithm 3** *Revise*$(x_k, c, deletion)$

```
1: for each $a \in D(x_k)$ do
2:     if $Consistent(x_k, a, c)$ = FALSE then
3:         $D(x_k) \leftarrow D(x_k) \setminus \{a\}$;
4:         $deletion \leftarrow$ TRUE;
5: if $D(x_k) = \emptyset$ then
6:     return FALSE;
7: return TRUE;
```

Value oriented adaptive propagation methods take decisions at an even lower level of granularity in the search/propagation process than variable oriented ones. This offers even greater flexibility because in contrast to variable oriented methods that use the same LC to check the consistency of all values in a domain, value oriented methods may choose to do this check using the standard method for some values or a SLC for other values. On the other hand, when a value oriented method chooses values to apply a SLC on, it needs to be quite precise so as to utilize the pruning strength of the SLC.

## 4 Review of adaptive constraint propagation methods

The idea of adapting the level of local consistency applied during search was introduced in the 90s in various contexts. At that time, the focus of research was mainly on binary constraints, and AC was considered the strongest practical local consistency, as very few stronger properties had been introduced. Hence, adaptive methods from the 90s proposed heuristics for either switching propagation between AC and a restricted form of AC (like forward checking) or for limiting propagation to certain parts of the problem. In the past decade or so there is rekindled interest in adaptive propagation, following the emergence of various strong propagation methods for both binary and non-binary constraints. They focus on heuristics for switching propagation between the solver's standard scheme and SLCs such as SAC and POAC.

Having identified three different categories of adaptive propagation methods (*node*, *variable*, and *value* oriented), we now review the literature and apply this classification to the existing methods.

## 4.1 Evaluation criteria

In the following, besides reviewing the existing methods of each category, we will also evaluate them according to the following criteria:

*constraint type dependency* Is the method applicable on constraints of any arity or is it limited to binary constraints? Can the method be applied in the presence of global constraints or other constraints with specific structure?
*LC dependency* Is the method tied to a specific local consistency (e.g. a specific SLC) or can it be used, as is or modified, in tandem with any SLC and any algorithm for applying it?
*solver dependency* Is the method applicable within the context of a specific solver architecture or is it generic? For example, does it require a variable-oriented propagation scheme? Does it work only with a specific branching scheme?
*parameter dependency* Is the method fully automated? Is it parameter-free? If not, does it require parameter tuning from the user's part?

For each category, before we move to discuss the specific methods that belong to it, we will make general remarks regarding the criteria so as to avoid repeating them for each individual method.

## 4.2 Node oriented methods

As discussed above, node oriented methods choose to enforce a specific LC at certain search tree nodes according to some heuristic. A node may correspond to a variable assignment, as is always the case in *d*-way branching, or to a value removal (in binary branching), or to some other branching decision (e.g. adding a constraint to the problem). As we will explain, some node oriented methods are better suited to *d*-way branching, but others are indifferent to the specific branching decision, and therefore in principle they can be applied to solvers of different architectures. In general, as node oriented methods take the decision about how to propagate at a high level of the solving process, other decisions taken at this level (e.g. branching scheme or variable ordering) may place restrictions on the applicability of the adaptive propagation method. For example, certain methods may only fit *d*-way branching or may only be applicable in tandem with specific variable ordering heuristics.

On the other hand, node oriented methods are quite powerful, in the sense that they can exploit the pruning power of SLCs to a high degree, given that SLCs are employed during entire rounds of propagation at specific nodes. As a downside, this means that the propagation technique of choice must be implemented for all the constraints in the problem. This is not an issue with the standard method applied by the solver because typically (G)AC algorithms are available for all the constraints that are supported by a solver. Even if this is not the case for some constraint, for which only, say, a bounds consistency filtering algorithm is available, the choice to propagate "in the standard way" at some node will imply to propagate this specific constraint with its available algorithm.

However, it is not realistic to assume that any given SLC will be implemented for all the available constraints, given that, as we discussed, SLCs for non-binary constraints have almost exclusively been proposed for table constraints. But importantly, this is not a problem with singleton consistencies which operate on top of the standard propagation mechanism. Hence, node oriented methods are better suited for use in tandem with SAC or its variants.

### 4.2.1 Reduced exceptional behaviour algorithm

In one of the early works on adaptive constraint solving, Borrett et al. proposed a method called REBA (*Reduced Exceptional Behaviour Algorithm*) that can dynamically switch between different search algorithms, including FC and MAC (Borrett et al. 1996). Although REBA cannot be exactly cast as an adaptive propagation technique (and that was not its goal in the first place), we include it in our review because it is an early example of a method that can adapt its propagation scheme at different nodes of the search tree.

Specifically, REBA uses a predefined chain of algorithms, starting from a very simple one and advancing to FC and then MAC, and a prediction mechanism for the early detection of thrashing behaviour (i.e., the repeated exploration of similar subtrees in the search tree caused by decisions taken further up the search tree). Search commences with the first algorithm in the chain and when the predictor, which depends on a combination of problem features, such as domain sizes, number of variables, and backtracks, decides so, the next algorithm in the chain is invoked. The interesting bit in terms of adaptive propagation, is that on the harder instances if at a certain point during search it is deemed that the limited pruning of FC does not cut down the search space enough then a switch to MAC will automatically occur.

### 4.2.2 Discussion

REBA is suited to *d*-way branching because FC is a technique that better works within such a setting. The performance of REBA depends on the value of a user-defined threshold that is used by the predictor module, but its main disadvantage, when evaluated under today's standards, is that the switch from one local consistency to another (i.e. FC to AC) occurs only once during search. However, the ideas put forward had an influence on later works.

### 4.2.3 Learning propagation policies

Epstein et al. used the ACE (*Adaptive Constraint Learning*) mechanism (Epstein et al. 2002) to learn a "policy" for propagation in a given problem class (Epstein et al. 2005). A policy includes decisions on how to perform preprocessing, on whether to use FC or AC or intermediate methods, such as the ones proposed in Freuder and Wallace (1991) that are discussed later, and importantly from the point of view of adaptive propagation, on whether to switch between different propagation techniques during search. If switching between methods was part of a policy then the depths in the search tree where this was done were also learned.

The work of Epstein et al. has some similarities with the REBA approach to adaptive propagation, especially in the motivation behind the switch between FC and AC, but it is a more modern and enhanced approach. It relies on machine learning to identify the levels of the search tree where the switch should be best employed, instead of relying on user-defined thresholds. Also, it utilizes an array of different propagation techniques that include FC, AC and a number of intermediate methods.

#### 4.2.4 Discussion

Evaluated by today's standards it is rather limited as an adaptive technique because it does not allow for arbitrary switches between the different propagation methods during search. Also, it requires the training of ACE, which means that a substantial number of instances of the target problem class must be available and solvable in reasonable time. However, the use of machine learning to learn how to perform adaptive propagation is quite interesting and has not been explored much within CP. Also, the ideas put forward are not limited to AC and FC, but are independent of specific LCs and solver settings. Therefore, they could potentially be used in the context of switching between (G)AC and SLCs, as is the norm in more recent works on adaptive propagation.

#### 4.2.5 Adaptive POAC

Moving on to more recently proposed methods, *Adaptive POAC* (APOAC) is a node oriented technique that tries to harness the pruning power of POAC without paying the high CPU cost of fully maintaining it during search (Balafrej et al. 2014). The main idea is that singleton consistencies like POAC are worth applying while they achieve a high number of value deletions. As algorithms for such consistencies iteratively make singleton tests on all variables until reaching a fix point, it could be advantageous to somehow restrict this iteration once the value deletion rate of the algorithm drops.

To implement this idea, APOAC focuses on the number of times that all the values in the domain of a variable are singleton tested during one call to the POAC algorithm. A series of singleton tests on all values of a variable is performed through a procedure called *varPOAC*. APOAC tries to learn a value $k$ for the maximum times that varPOAC will be called for each variable, by alternating between a learning and an exploitation phase during search. During the learning phase POAC is run to completion at each node, and its effects in terms of pruning on individual variables are monitored. As a result, the value of a parameter, indicating after how many varPOAC calls the algorithm starts being ineffective, is learned. During the exploitation phase, the method applies a weaker version of POAC that calls procedure varPOAC on each variable only until the learned number of calls.

Starting with a learning phase, one of the two phases is executed on a sequence of nodes before switching to the other phase for another sequence of nodes. The total length of a pair of sequences learning + exploitation is fixed to a parameter $LE$. APOAC uses an approximation of the search space size, called the *volume* of the CSP, which is the $log_2$ of the Cartesian product of the domains, to measure the search space reduction achieved during the learning phases. The $i$th learning phase is applied to a sequence of $1/10 \times LE$ consecutive nodes. During that phase, a cut-off value $k_i$ is learned, which is the maximum number of calls to the procedure varPOAC that each node of the next ($i$th) exploitation phase will be allowed to perform.

The value of $k_i$ is learned based on the cut-off value $k_{i-1}$ of the previous learning phase and the reduction in the volume of the problem observed during propagation at this phase. To be more specific, for each node explored during the learning phase, a cut-off value $k_i(j)$, with $j = 1 \dots 1/10 \times LE$, is learned, and these values are then aggregated (e.g. by taking their mean) to produce $k_i$. Each $k_i(j)$ represents the number of calls to varPOAC during which the reduction in volume was deemed *sufficient* during propagation at that specific node. Different criteria as to what constitutes sufficient volume reduction were proposed (e.g. if there was at least one value deletion).

### 4.2.6 Discussion

APOAC was proposed as a specific technique tied to POAC, but its main idea could easily be used in tandem with other singleton consistencies. Being an adaptive technique for singleton consistencies, it is independent of the constraints' arity or type, and can be used with any branching scheme. However, APOAC is quite complex compared to other adaptive propagation methods, and crucially, it is heavily dependant on various parameters for determining, for instance, how the effort between learning and exploration will be divided, or what constitutes a significant reduction of the search space volume.

### 4.2.7 Multi-armed bandits

An adaptive propagation method based on the reinforcement learning technique known as *Multi-Armed Bandits* (MAB) was proposed by Balafrej et al. (2015). This is one of the few methods that use machine learning to guide the decisions about what propagation method to use, and it is interesting in several respects. The main idea is to attach a ML component, called a MAB selector, to each level of the search tree. The MAB selector at some level decides which LC to apply whenever propagation is triggered at that level. The selector is based on a model defined over:

1. A set of $k$ arms $\{LC_1, \dots, LC_k\}$. Each arm corresponds to an algorithm that enforces a specific LC.
2. A set of rewards $R_i(j)$, with $1 \leq i \leq k$, $j \geq 1$, where $R_i(j)$ is the reward delivered when an arm $LC_i$ has been chosen at time $j$.

The reward function can be any measure that reflects the performance or a criterion that indicates the appropriate arm. The implementation described used the CPU time as measure of performance (Balafrej et al. 2015). Specifically, if a LC $i$ is selected to be applied at some node at time $j$ then the reward $R_i(j)$ is computed based on the performance of LC $i$ at time $j$ compared to the performance of all consistencies at previous visits at this depth, as follows:

$$R_i(j) = 1 - \frac{T_i(j)}{max_{i=1\dots k, m=1\dots j} T_i(m)}$$

where $T_i(m)$ is the CPU time needed to enforce LC $i$ at the $m$th visit of a node at the given depth plus the time to explore the sub-tree rooted at that node.

The policy used to choose the LC to apply at any node takes into account the previous rewards obtained by the LCs, the number of times each LC was selected, and the current total number of calls to the available LCs. Specifically, the policy selects the LC $i$ that maximizes:

$$\rho(i) = \overline{R}_i + \sqrt{\frac{2lnm}{m_i}}$$

where $\overline{R}_i$ is the mean of the past rewards of the $i$ LC, $m_i$ is the number of times LC $i$ was called and $m$ is the current number of calls to any LC. The reward term $\overline{R}_i$ encourages the exploitation of successful LCs, i.e. ones with higher-rewards, while the term $\sqrt{\frac{2lnm}{m_i}}$ promotes the exploration of the less selected LCs so as to avoid constantly selecting the same LC.

The LCs used by the MAB approach for the experiments presented in Balafrej et al. (2015) were AC, maxRPC, and POAC. The results obtained demonstrated that MAB manages to select the "right" LC for most problems, achieving a very good overall performance and outperforming APOAC on hard instances. However, these results were obtained under the older dom/deg variable ordering heuristic, and may be considerably different under modern heuristics, as we detail in Sect. 5.

#### 4.2.8 Discussion

The MAB method has some strong points compared to other methods of any category. First, it is not limited to choosing the propagation method among two options (a weak and a strong one), as most methods typically do. The allocation of one bandit to one LC allows the MAB method to exploit an array of different LCs with different pruning power and cost. Second, the method is fully automated, as it does not require any parameter tuning. And although it is based on machine learning, it does not require training (as does the method of Epstein et al. 2005 for example). Another advantage of the method is that it can be modified to be used as a variable oriented adaptive technique. This is because MABs could be attached to individual variables instead of search tree levels, as noted in Balafrej et al. (2015).

However, the MAB method also has an important weakness. Given that each MAB operates at a fixed level of the search tree, its performance is largely affected by high level settings of the CP solver, such as the branching scheme and the variable ordering heuristic of choice. The MAB method is tailored to $d$-way branching where each level of the search tree corresponds to a variable and the entire set of its variable assignments, while it is not suitable (at least in its original form) to binary branching where each level corresponds to one variable assignment and its refutation. Also, the MAB method does not interact well with highly dynamic variable ordering heuristics (e.g. dom/wdeg) because such heuristics affect the effectiveness and stability of a bandit's learning.

#### 4.2.9 PrePeak

PrePeak is a node oriented method that, by monitoring the backtracks that occur during search, can activate a SLC algorithm once thrashing starts to be noticed. Then, depending on the effects of this algorithm, propagation sooner or later reverts to the standard choice, until thrashing is detected again, and so on Woodward et al. (2018).

Specifically, the number of times each level of the search tree was backtracked to is stored in a vector *btcounts*[] indexed by the corresponding level. When an entry in this vector reaches a threshold value $\theta$, then the corresponding tree level is considered as the "peak" depth of thrashing, and is stored in a global variable $peak_d$. Once search backtracks to a shallower level than $peak_d$, propagation switches from the standard method to the SLC and continues using the SLC as long as it is effective. Once it starts being deemed as ineffective, propagation switches back to the standard method, and all the counters in *btcounts*[] are reset to 0. The main idea is that through $peak_d$ a level of search where intensive thrashing (i.e. a peak) occurs is identified. Then after backtracking above this level, the use of the SLC as search moves forward again will hopefully help alleviate the thrashing at the peak level.

PrePeak commences search using the standard propagation method, and as soon as $n^2$ nodes have been counted (with $n$ being the number of variables) then $\theta$ is initialized to the maximum value of the entries in *btcounts*[]. Thereafter, the switching mechanism is activated. Whenever propagation is performed using the SLC, $\theta$ is updated to reflect the effects of propagation. This update is done by multiplying its value with a factor or $r_w$, $r_f$, or $r_n$, depending on whether propagation has resulted in a DWO, in the filtering of some values but no DWO, or in no filtering at all, respectively. The three factors are set to appropriate values ($1.2^{-1}$, $1.2^2$, $1.2^3$, respectively) to reflect that in the first case where a DWO is caused, the SLC should continue to be applied, while in the other two it is better to prevent it from triggering again too soon.

A variant of PrePeak, named PrePeak+, was also proposed. PrePeak+ uses a mechanism to enforce the early termination of the SLC when this is being applied, so as to keep most of its pruning power while reducing the cost of its application. This is similar to the idea behind APOAC but it is implemented in a much simpler way. The termination is controlled either by the size of the propagation queue or by the run time of the SLC algorithm. The former allows only a fraction of the propagation queue to be processed, while the latter imposes a bound on the duration of any call to the SLC.

Experimental results with PrePeak+, using POAC as the SLC, were very promising, showing that it is clearly superior to MAC under the dom/ddeg heuristic, and quite competitive under dom/wdeg, beating MAC on some problem classes. Also, in the only existing experimental comparison of different adaptive propagation methods, experiments showed that PrePeak+ is clearly superior to the other two recent node-oriented methods, APOAC and MAB (Woodward et al. 2018).

#### 4.2.10 Discussion

PrePeak is a sophisticated adaptive propagation method that tries to precisely focus the application of a SLC on parts of search where thrashing occurs, by monitoring the backtracks that take place. Given the experimental results achieved, it is clear that PrePeak is the state-of-the-art in node oriented methods.

As a downside, PrePeak is not fully automated, as it requires tuning the initialization and update policies of the threshold $\theta$ that determines when the SLC will be activated. Also, there are other switches that need to be set to control how long the SLC algorithm will be allowed to run if the PrePeak+ variant is used. Finally, but importantly, PrePeak is suited to $d$-way rather than 2-way branching because, like the MAB method, its methodology is tied to levels of the search tree that correspond to variables.

### 4.3 Variable oriented methods

Variable oriented adaptive propagation methods make a choice about which LC to enforce every time a variable is removed from the propagation queue. This requires the use of a solver that employs variable oriented propagation, which is often the case but not always. The branching scheme of the solver does not make any difference as the decision about propagation is taken at a lower level of granularity. The dependency on parameter settings varies from method to method as we detail below.

Again, the use of singleton consistencies as alternatives to standard propagation results in a method with wider applicability compared to other SLCs. This is because in the former case, once a variable under revision is chosen for propagation with the SLC, a singleton consistency algorithm can test the values in the domain of the variable by simply temporarily assigning them and calling the standard propagation mechanism of the solver. In contrast, a different SLC, e.g. a relation based one, needs to be implemented for constraints of many different types, given that a variable may be involved in many different constraints. But this is not feasible, especially in the case of global constraints.

### 4.3.1 Selective relaxation

The idea of adaptive propagation was first put forward by Freuder and Wallace (1991). They proposed a technique, called *selective relaxation* which can be used to restrict AC propagation on certain variables based on two local criteria, and they incorporated this technique in the standard arc consistency algorithm AC3 accordingly.

The first criterion is the distance in the constraint graph of any variable from the currently instantiated one where propagation is initiated. Hence, the so called *distance-bounded* relaxation confines propagation to variables within a fixed distance from the first variable processed. In the context of the framework described in the previous section, this can be implemented by calling function *Revise* only for variables whose distance from the first one is within the given bound. The second heuristic stops propagation along a path of the constraint graph as soon as the amount of value pruning from a variable's domain falls between a certain threshold. This is called *response-bounded* relaxation and can be implemented within our framework by not allowing the insertion of a variable into $Q$ if the percentage of values that were pruned from its domain during a call to *Revise* falls below the threshold.

### 4.3.2 Discussion

Although selective relaxation was proposed within the context of AC, it can be generalized to other LCs, especially for the case of binary constraints. The response-bounded variant is applicable to constraints of any arity, whereas the distance-bounded one may require modification because the notion of distance between variables could be misleading as a means to restrict propagation in a hyper-graph, because of the large differences in the propagation cost of different types of non-binary constraints. For instance, two variables may be in close distance, but the cost of propagation for the (non-binary) constraints that connect them could be quite high (as is the case for table constraints, for example). While in another case, two variables may be quite far apart, but the cost of propagating the connecting constraints could be low (e.g. if they are connected through a sequence of binary constraints). Selective relaxation is easy to implement but depends heavily on the parameters used for the distance or the response bound.

Although the idea of stopping propagation altogether to avoid potential overheads is rather outdated, a modification of selective relaxation so that different local consistencies are used for propagation according to the distance or the response may be worth exploring.

For instance, propagation could kick off with a SLC and switch to a weaker LC once the distance between the first variable inserted in the queue and the currently removed one goes beyond a certain bound. Similarly for the response-bounded case. As we explain below, these ideas have influenced certain techniques that were more recently proposed.

#### 4.3.3 Adaptive constraint propagation

In the work where the term "adaptive constraint propagation" was first introduced, El Sakkout et al. proposed a scheme aptly called *Adaptive Arc Propagation* (AAP) for dynamically deciding whether to process any individual constraint $c_{ij}$ using AC or FC-like propagation (El Sakkout et al. 1996). That is, whether to revise $x_j$ whenever there is a value removal from $D(x_i)$ (AC propagation) or only when $D(x_i)$ is reduced to a singleton (FC-like propagation).

To achieve this, for any constraint $c_{ij}$ and any value $a \in D(x_j)$ AAP calculates *AC-superiority*$(x_j, a)$, which represents the probability that AC revision will remove $a$, whereas FC will not. These probabilities are calculated by taking into account the number of supports that the values of $x_j$ have in $D(x_i)$, and can be then aggregated to indicate whether it is worth making a revision using AC or FC. The authors do not specify how the probabilities can be aggregated, except for the specific case of *anti-functional* constraints, but roughly speaking, if the values of $x_j$ have a small average number of supports in $D(x_i)$ then it is more likely that AC will prune extra values compared to FC.

The instantiation of the general schema to the case of anti-functional constraints is further analyzed and evaluated. In an anti-functional constraint $c_{ij}$, for any value in $D(x_j)$, all values in $D(x_i)$, except one, support it (e.g. $\neq$ is anti-functional). In this case, AC propagation can achieve no more pruning than FC-like propagation.

Although AAP best fits a constraint oriented propagation mechanism, it can be easily incorporated within a variable oriented one such as the one we describe in Sect. 3. Specifically, in function *Revise* (Algorithm 3), for each variable $x_k$ we can determine if the consistency of its values will be checked using the standard propagation method or a SLC based on the scheme of AAP, assuming that the aggregation of the *AC-superiority* probabilities has been defined.

#### 4.3.4 Discussion

AAP is rather outdated by today's standards, given that it is based on support counting, which is a prohibitively expensive procedure for non-binary constraints. Support counting also means that it is very specific to AC and weaker variants of AC, like FC. On the other hand, the idea of using some probabilistic procedure to determine how constraints will be propagated is very interesting and has not been explored much in CP.

#### 4.3.5 Monitoring the effects of revisions

A more recent method which rekindled interest on adaptive constraint propagation was proposed by Stergiou (2008, 2009). This method is a heuristic, coming in various flavours, which allows the solver to switch between a SLC and AC by monitoring the effect of constraint revisions ( the AAP method of ElStergiou 2008, 2009). Although it was originally proposed following a constraint oriented propagation scheme, it was later modified to fit variable oriented propagation (Paparrizou and Stergiou 2012).

The motivation behind the proposed method comes from observations based on experimental results. Namely, that in structured problems constraint revisions that cause domain reductions or even DWOs closely follow one another. That is, if the revision of a constraint $c$ at some point during search causes domain reductions then it is likely that the revisions of $c$ that immediately follow will also cause domain reductions. Hence, the main heuristic proposed by Stergiou works as follows: all constraints are revised using the standard propagation method, but once the revision of a constraint causes a DWO then this constraint is revised using the SLC for its $l$ following revisions, where $l$ is a parameter set by the user. Given that these revisions are likely to cause pruning, the use of the SLC may boost the pruning achieved, possibly resulting in early backtracks.

A number of variants of the basic heuristics were also proposed. For example, instead of monitoring DWOs and switching the propagation method based on their occurrence, an alternative heuristic monitors and reacts to value deletions. This results in more often invocations of the SLC. Disjunctive combinations of the DWO and deletion monitoring heuristics were also considered.

In a subsequent work, the heuristics were described in a variable-oriented scheme, where the revisions of variables instead of constraints are monitored (Paparrizou and Stergiou 2012). Also, a variant of the method that does not require parameter setting was proposed. However, this variant is too restrictive, in the sense that it only applies the strong propagator in the revision that immediately follows a DWO (or a deletion).

#### 4.3.6 Discussion

The revision monitoring approach is very simple, it is independent of the constraints' arity and the branching scheme, and can be incorporated in solvers with different architectures. Also, it can work with SLCs of different types, be it singleton consistencies [as in Paparrizou and Stergiou (2012), triangle based (Stergiou 2008) or relation based (Paparrizou and Stergiou 2012)]. However, a major drawback of this approach is that the best heuristics proposed rely on the parameter $l$, whose value needs to be set through experimentation.

### 4.4 Value oriented methods

As explained, these methods selectively apply a strong propagation technique on specific values when certain conditions are met. As in the other categories, and perhaps even more so, such methods are particularly suited to SAC-like SLCs because after the specific values have been selected, a singleton consistency will simply assign them temporarily and call the standard propagation mechanism to test consistency.

#### 4.4.1 Probabilistic arc consistency

*Probabilistic Arc Consistency* (PAC) is a method proposed by Mehta and van Dongen that bears resemblance to the AAP method of (El Sakkout et al. 1996) in the sense that it also uses probabilistic reasoning about supports (Mehta and van Dongen 2007). PAC tries to avoid checking the consistency of some values that are unlikely to be pruned by AC, using probabilities about the existence of supports. As in AAP, the scheme is based on information gathered by examining the supports of values in constraints. However, this costly operation is only performed once, before search commences. Also, the reasoning is first applied at the level of values (hence we cast it as a value oriented method), but can also be applied at the level of variables as with AAP.

Specifically, when a support for a value $a \in D(x_i)$ is sought in a domain $D(x_j)$, PAC computes a probability that support exists based on the number of supports that $a$ had in the initial domain of $x_j$, the current domain size of $x_j$, and the number of values that have been removed from $D(x_j)$. If this probability is greater or equal to a threshold then the search is not executed. This is called a *probabilistic support condition* and could be incorporated right before function *Consistent* is called in function *Revise* (Algorithm 3) of our solver framework from the previous section.

This can also be applied at the level of variables in what is called a *probabilistic revision condition* in Mehta and van Dongen (2007). Specifically, if for all values in $D(x_i)$, the probabilities that they have support in $D(x_j)$ are computed and the minimum value of these probabilities is greater or equal to the threshold then the revision of $x_i$ with respect to $x_j$ can be avoided, as it is likely that no pruning will be achieved. Mehta and van Dongen also proposed to use the reasoning behind PAC within a SAC algorithm, and specifically when AC is being enforced within singleton tests.

### 4.4.2 Discussion

Although support counting is only done once in a preprocessing step, PAC's reliance on such a process means that it is only applicable on binary constraints, or very specific non-binary ones. Also, it is difficult to generalize it to other LCs apart from AC because it is based on the notion of support for a value in a domain. However, the main idea of somehow computing the likelihood that a value is consistent and acting accordingly during propagation is certainly interesting.

### 4.4.3 (Quick) shaving

The selective application of SAC on certain values is known as *Shaving*. This technique originates from scheduling and numeric CSPs but has also been explored in the context of finite domain CSPs. Lhomme proposed a heuristic method, called *quick shaving* that can be used to select values to test for SAC based on the likelihood that they are ''shavable'', i.e. they will be removed by SAC (Lhomme 2005). Quick shaving makes a singleton check on a value $a \in D(x_i)$ at level $k$ of the search tree only if it was shavable at level $k$+1, and such decisions are taken only after a backtrack to level $k$ has occurred.

Initial information about shavable values at level $k + 1$ is cheap to obtain because it comes from tried and failed instantiations. That is, if search assigns $a$ to $x_i$ at level $k$+1 and propagation of the assignment fails then $a$ will be singleton tested when the solver backtracks to level $k$. Apart from the values that are selected for singleton testing, propagation runs in the standard way, and therefore quick shaving can be viewed as a value oriented adaptive propagation method.

Szymanek and Lecoutre studied ways to selectively apply shaving using the semantics of the global constraints alldifferent and sum to suggest the values that are most likely to be removed by a singleton test (Szymanek and Lecoutre 2008). According to the proposed heuristic, each global constraint ''proposes'' one value to singleton test. The choice of value is made in such a way so that the pruning power of the constraint is maximized if the value is deleted. Information about the success of failure of past singleton tests is also exploited to fine-tune the heuristic.

Zanarini and Pesant performed an experimental evaluation of quick shaving across several benchmarks and in tandem with various variable ordering heuristics (Zanarini and Pesant 2009). The obtained results allowed them to identify a threshold in the success ratio of shaving, i.e. the number of times it achieves pruning over the number of singleton tests, below which quick shaving becomes an overhead. Based on this they proposed a simple adaptive method that dynamically switches off quick shaving once its success ratio falls below the threshold. This is reminiscent of the *response-bounded* relaxation heuristic from Freuder and Wallace (1991).

### 4.4.4 Discussion

Quick shaving is easy to incorporate into solvers, it works with constraints of any arity or type and is largely independent of the internal mechanics of solvers (e.g. it works with any branching scheme). In the context of our framework, we can incorporate such a method by simply testing the consistency of the selected values through a singleton test within function *Revise*. As a downside, adaptive methods based on shaving are specific to SAC-like SLCs. Also, quick shaving is narrow in its application of singleton tests as it only applies them on specific values, only after a backtrack. But importantly, it does not depend on parameters that the user needs to set, except for the variant of quick shaving given in Zanarini and Pesant (2009).

### 4.4.5 Adaptive parameterized consistency

*Adaptive parameterized consistency* is a value oriented adaptive method that decides whether to enforce AC or a stronger consistency on a value, based on its *stability*, which is an estimate of the difficulty to find support for this value (Balafrej et al. 2013). Two variants of the method are described. The first one uses a parameter that needs to be set through experimentation or experience, while the second one, on which we focus, does not require any parameter setting, and is thus fully automated.

Assuming ordered domains, for any variable $x_i$ and value $a \in D(x_i)$, the *distance to end* of $a$ in $D(x_i)$, denoted $\Delta(x_i, a)$, is the normalized distance between $a$ and the end of $D(x_i)$, defined as follows:

$$\Delta(x_i, a) = (|D_0(x_i)| - rank(a, D_0(x_i)))/|D_0(x_i)|$$

where $D_0(x_i)$ is the initial domain of $x_i$ and $rank(a, D_0(x_i))$ is the position of value $a$ in the ordered set of values $D_0(x_i)$.

Given a value $a_i \in D(x_i)$ and a binary constraint $c_{ij}$, $a_i$ is *stable* for AC on $c_{ij}$ if it has a support $\sup(a_i, x_j)$ in $D(x_j)$ s.t. the distance to end of $\sup(a_i, x_j)$ in $D(x_j)$ is greater or equal to a parameter $p(x_i)$. A value $a_i \in D(x_i)$ is stable if it is stable on all constraints where $x_i$ participates. If the stability of value $a_i$ falls under the threshold $p(x_i)$ on a constraint $c_{ij}$ then, assuming that the value of $p(x_i)$ is suitably small, it is unlikely that $a_i$ will have a lot of supports (if any) in $D(x_j)$, apart from $\sup(a_i, x_j)$. Therefore, it is likely that the application of a SLC on $a_i$ will result in its pruning. Hence, the main idea is that if it is difficult to find a support for a value then a SLC may prune this value.

Adaptive parameterized consistency has two important properties. First, the threshold $p(x_i)$ for each variable can be adjusted automatically during search using the weighted degrees of the variables, as we will explain in the next section. Second, there is no need to compute all the supports for any value, as is the case for other methods that reason about supports such as AAP and PAC.

Adaptive parameterized consistency was extended to non-binary table constraints utilizing Simple Tabular Reduction algorithms (Ullmann 2007; Lecoutre 2011) that offer information about the number of supporting tuples that a value has on a constraint (Woodward et al. 2014). This means that the decision about which LC to apply is taken using the exact number of supporting tuples that a value has on a constraint at any time, instead of an approximation of support, as does the original method for binary constraints. The proposed method switches between GAC and pairwise consistency, selecting the latter when the number of supports for some value is below a threshold.

#### 4.4.6 Discussion

An important advantage of adaptive parameterized consistency is that it is fully automated, as the setting of its parameters does not require any user involvement. Also, it can be combined with various SLCs, be it triangle-based [as in Balafrej et al. (2013)], relation-based [as in Woodward et al. (2014)], or SAC-like (as we demonstrate in the next section). On the other hand, the applicability of adaptive parameterized consistency is essentially limited to binary constraints and to non-binary table constraints because of the way it reasons about supports. This reasoning is difficult, and in most cases impractical, to generalize to global constraints.

### 4.5 Summary

Table 1 summarizes our critical evaluation of the existing adaptive propagation methods based on the criteria given in Sect. 4.1. Let us explain the information given in the columns of the table:

*Method* The name and main reference for each method is given.
*Cons Type (eval)* We specify whether a method is applicable to constraints of any arity or just to binary ones. If a method that works with constraint of arbitrary arity was only evaluated on binary problems in the paper where it was proposed, we write “bin” in brackets.
*SLC type - #SLCs* “SLC type” specifies the type of SLC that a method can work with. FC/AC means that the corresponding method was proposed as a heuristic for switching between FC and AC. For node-oriented methods we specify that the chosen SLC is applied uniformly on all variables/constraints of the problem. #SLCs specifies whether a method can switch between standard propagation and only one specific SLC or if it can choose among more than one SLCs.
*switches* This piece of information refers to the switching between the standard propagation and the SLC(s) during search. Most methods allow for multiple switches at heuristically chosen points in time. But some place restrictions (Quick Shaving switches only after a backtrack, while ACE allows switching at certain levels of the search tree).
*solver* Here we highlight the dependence of a method on solver settings. Specifically, we refer to possible dependence on the branching scheme (e.g. d-way means that the method is tailored for d-way branching), on the variable ordering heuristic (voh), and the propagation scheme (variable or constraint oriented). Also, some methods may be

**Table 1** Summary of the critical evaluation of adaptive propagation methods

| Method | Cons type (eval) | SLC type - #SLCs | Switches | Solver | Parameters |
|---|---|---|---|---|---|
| *Node-oriented* | | | | | |
| REBA Borrett et al. (1996) | any (bin) | FC/AC - 1 | once | d-way | one (user) |
| ACE Epstein et al. (2005) | any (bin) | any (uniform) - >1 | multiple (level) | independent | some (train.) |
| APOAC Balafrej et al. (2014) | any | SAC-based - 1 | multiple | independent | some (user) |
| MAB Balafrej et al. (2015) | any (bin) | any (uniform) - >1 | multiple | d-way/voh | some (autom.) |
| PrePeak Woodward et al. (2018) | any | any (uniform) - 1 | multiple | d-way | some (user) |
| *Variable-oriented* | | | | | |
| Sel. relaxation Freuder and Wallace (1991) | any (bin) | any - 1 | multiple | independent | one (user) |
| AAP El Sakkout et al. (1996) | bin | FC/AC - 1 | multiple | support search | one (user) |
| Rev. monitoring Stergiou (2008) | any (bin) | any - 1 | multiple | cons-oriented | one (user) |
| Rev. monitoring Paparrizou and Stergiou (2012) | any | any - 1 | multiple | var-oriented | none |
| *Value-oriented* | | | | | |
| PAC Mehta and van Dongen (2007) | bin | AC/SAC - 1 | multiple | support search | one (user) |
| Quick Shaving Lhomme (2005) | any | SAC-based - 1 | multiple (bt) | independent | none |
| Ad. Param. Cons. Balafrej et al. (2013) | bin+table | many - 1 | multiple | support search | one (autom.) |

applicable only in tandem with propagation algorithms that operate by searching for support for values in domains.

*parameters* This refers to the dependence of a method on parameter settings. A method may use only one parameter that is either set by the user or automatically. Alternatively, it may require the setting of several parameters, which is done by the user or automatically or through training on CSP instances. There are also cases where no parameters are required.

## 5 Experimental evaluation

As mentioned above, the only existing experimental comparison of adaptive propagation methods was carried out between the three recently proposed node oriented techniques (APOAC, MAB, PrePeak) (Woodward et al. 2018). But no comparison between methods operating at different levels of granularity has been made. We address this by running experiments on three representative methods, one from each class. Specifically, we compare PrePeak, one of the best heuristics from Stergiou (2008) (called $H_1$ in Stergiou 2008 and VarAdapt hereafter), and the adaptive parameterized consistency method of Balafrej et al. (2013) (called ValAdapt hereafter).

In the following, we will refer to an algorithm that maintains a LC, by the name of that LC. Hence, MAC will be referred to as AC. Following our analysis in the previous sections, we concluded that singleton consistencies are better suited for use within adaptive propagation methods than other SLCs. Hence, we focused on recently proposed low-cost singleton consistencies like NSAC, NPOAC, RNPOAC and RNSAC.

To decide which is the best choice of SLC among them, we first experimented with algorithms that maintain such singleton consistencies during search. The results revealed that NSAC is the worst option being clearly less efficient than the other three. Among RNPOAC and NPOAC, the latter is usually slightly faster, while there were negligible differences between RNSAC and RNPOAC. Hence, we decided to focus on RNSAC and NPOAC which, in terms of pruning, are the weakest and strongest SLCs among the four considered. We now discuss the implementation of the three adaptive methods.

### 5.1 PrePeak

PrePeak was implemented as described in Woodward et al. (2018) regarding the main part of the method, including the initialization and updates of the triggering threshold. However, the mechanism to control the early termination of the SLC algorithm was not implemented. The reason for this is twofold:

- For a fair comparison we would have to implement the same mechanism in the methods that are compared to PrePeak, but this would introduce extra parameters to these methods.
- The SLCs considered in this paper are neighbourhood variants of SAC and POAC, whose algorithms typically have much shorter runs than SAC and POAC algorithms, especially on sparse networks. This is of course because at any singleton test of a value $a_i \in D(x_i)$, the neighbourhood variants only propagate the assignment of $a_i$ to $x_i$ in $\mathrm{N}(x_i)$

instead of the whole network. As a result, forcing early termination is largely unnecessary.

POAC is a better option than SAC when used within node oriented methods such as PrePeak Balafrej et al. (2014), Woodward et al. (2018). But in our experiments we noticed very small, essentially negligible, differences between PrePeak variants that use any of the four neighbourhood singleton consistencies that we considered. This is because, as we detail below, PrePeak makes very few singleton tests compared to the other adaptive methods. Hence, the difference in pruning power between the four SLCs has a very small impact.

## 5.2 VarAdapt

Function *Revise-VarAdapt* (Algorithm 4) gives a detailed description of the *Revise* function from Sect. 3 (Algorithm 3), as we have implemented it within method VarAdapt. It differs from the corresponding description in Stergiou (2008) in two features:

1. We use here a variable-oriented propagation scheme, whereas a constraint-oriented propagation scheme was used Stergiou (2008).
2. The original description was specifically tailored for heuristics that use maxRPC as the SLC. Here, the description fits the use of a singleton consistency as the SLC.

As in Algorithm 3, *Revise-ValAdapt* takes as arguments the variable to be revised, the constraint that it will be revised against, and the flag *deletion*. But in addition, it takes a parameter *l* that controls the amount of times that the SLC will be applied.

**Algorithm 4** Revise-VarAdapt($x_k$, $c$,deletion,$l$)

```
1:  $revision[k] \leftarrow revision[k]+1$;
2:  for each $a \in D(x_k)$ do
3:      $consistent \leftarrow$ Consistent$(x_k, a, c)$;
4:      if $consistent =$ TRUE then
5:          if $revision[k]$ - $dwo[k] \leq l$ then
6:              $D(x_k) \leftarrow \{a\}$;
7:              $consistent \leftarrow$ SLC-Consistent$(\mathcal{P})$;
8:      if $consistent =$ FALSE then
9:          deletion $\leftarrow$ TRUE;
10:         $D(x_k) \leftarrow D(x_k) \setminus \{a\}$;
11: if $D(x_k) = \emptyset$ then
12:     $dwo[k] \leftarrow revision[k]$;
13:     return FALSE;
14: return TRUE;
```

Assuming that a variable $x_k$ is to be revised against a constraint $c$, *Revise-VarAdapt* is called to enforce a local consistency on the values of $x_k$. VarAdapt uses two global vectors *revision*[] and *dwo*[] of size $n$, which are initialized to 0 for every variable. At each revision of a variable $x_k$, *revision*[$k$] is incremented by one (line 1), meaning that it counts the revisions of $x_k$. If the revision of $x_k$ results in a DWO, *dwo*[$k$] is set to *revision*[$k$] (line 12), meaning that at any point, and for every variable $x_k$, *revision*[$k$] records the most recent revision that resulted in the variable's DWO.

We first check the consistency of each value $a \in D(x_k)$ using the standard *Consistent* function of the solver. For binary constraints, where AC is used for propagation, this function simply seeks support for $a$ on $c$ (i.e. in $D(x_j)$, where $x_j$ is the other variable involved in $c$ - see Function *Propagate* in Sect. 3). If $a$ is consistent then, depending on a condition, it may be checked for consistency again, using the SLC this time. The consistency of a value is first checked using the standard LC to quickly locate inconsistencies that are provable through the weaker LC. This is typical in SAC and SAC-like algorithms where AC is enforced before applying the singleton consistency.

The condition we consider here concerns the DWOs suffered by $x_k$. If the distance, in terms of consecutive revisions, between the current revision of $x_k$ and the most recent revision that resulted in a DWO, is less or equal to $l$ then all values of $D(x_k)$ are tested using the given SLC. The idea is that revisions that cause DWOs frequently appear in clusters, so focusing a SLC on them may result in early backtracks (Stergiou 2008).

A crucial factor that affects the performance of VarAdapt is the value of the parameter $l$. Unfortunately, there is no "optimal" value for all problem classes and it is quite likely that the value needs to be readjusted to maximize performance, not only when considering different problem classes but also when considering different SLCs. In our experiments we set $l$ to 100, which has been shown to achieve a good overall performance (Stergiou 2008).

## 5.3 ValAdapt

Function *Revise-ValAdapt* (Algorithm 5) gives a detailed description of the *Revise* function from Sect. 3 (Algorithm 3), as we have implemented it within method ValAdapt. As in the case of VarAdapt, our description differs slightly from the original one which was focused on constraint-oriented propagation and the use of maxRPC as the SLC, while we focus on variable-oriented propagation and singleton consistencies.

**Algorithm 5** Revise-ValAdapt($x_k, c_{kj}$,deletion)

```
1: $p(x_k) \leftarrow$ (wdeg($x_k$) - min$_{x \in \mathcal{X}}$(wdeg($x$))) / (max$_{x \in \mathcal{X}}$(wdeg($x$)) - min$_{x \in \mathcal{X}}$(wdeg($x$)));
2: for each $a \in D(x_k)$ do
3:     $consistent \leftarrow$ Consistent($x_k, a, c_{kj}$);
4:     if $consistent$ = TRUE then
5:         if $\Delta$($x_j$,sup($a, x_j$)) $< p(x_k)$ then
6:             $D(x_k) \leftarrow \{a\}$;
7:             $consistent \leftarrow$ SLC-Consistent($\mathcal{P}$);
8:     if $consistent$ = FALSE then
9:         deletion $\leftarrow$ TRUE;
10:        $D(x_k) \leftarrow D(x_k) \setminus \{a\}$;
11: if $D(x_k) = \emptyset$ then
12:     return FALSE;
13: return TRUE;
```

As in Algorithm 3, *Revise-VarAdapt* takes as arguments the variable to be revised, the constraint that it will revised against, and the flag *deletion*. As this method is specific to binary constraints, we specify that the constraint involves two variables $x_k$ and $x_j$.

A strong advantage of ValAdapt is that it is fully automated. The value of the threshold $p(x_k)$ is automatically computed for each variable $x_k$ based on the weighted degree of $x_k$ and the maximum and minimum weighted degrees of all variables (line 1). The idea is that variables with high weighted degree are associated with very active constraints and therefore applying a SLC on their values may cause extra pruning. Information about weighted degrees is available if dom/wdeg is used for variable ordering, which is a reasonable assumption given that dom/wdeg is probably the most efficient heuristic for binary CSPs. But even if some other heuristic is used, it is easy and cheap to maintain information about weighted degrees.

As we are dealing with binary constraints, AC will naturally be the standard propagation method. After a value $a \in D(x_k)$ is checked for AC, if it is verified that the normalized distance $\Delta(x_j,\mathrm{sup}(a,x_j))$ between its located support $\mathrm{sup}(a,x_j)$ and the end of $D(x_j)$, is less than $p(x_k)$, then $a$ is checked for consistency using the SLC.

Note that ValAdapt cannot be combined with (N)POAC. This is because when a variable $x_k$ is revised, some of its values may not be singleton tested because the condition in line 5 may prevent it. But in order for (N)POAC to delete a value $a'$ from the domain of some other variable, $a'$ must not be present in any AC($\mathcal{P}_{x_k \leftarrow a}$), i.e. all the singleton tests of the values in $D(x_k)$ must remove it. Hence if some values in $D(x_k)$ are not singleton tested then $a'$ cannot be deleted. Given this, we used RNSAC as the SLC within ValAdapt.

### 5.4 Experimental results

We experimented with 59 benchmarks from 17 classes that include 1398 instances in total, available from C. Lecoutre's website[2]. For a fair comparison between ValAdapt and VarAdapt, we used RNSAC as the SLC in both methods. For the case of PrePeak we report results from its use with NPOAC as the SLC, but as mentioned, the results are very similar when RNSAC is used. We also report results from AC, RNSAC, and NPOAC as baseline methods. We first ran experiments with dom/ddeg to evaluate the methods under a variable ordering heuristic that does not interfere too much with propagation. Then we experimented with dom/wdeg, which interacts heavily with the propagation mechanism because of constraint weighting, but is much more efficient in practice.

#### 5.4.1 Dom/ddeg

Table 2 presents results from various benchmarks comparing the methods under the dom/ddeg variable ordering heuristic. The benchmarks are selected to demonstrate cases where AC is clearly worse or clearly better than the SLCs, so that the performance of the adaptive methods is highlighted. For each method we give the sum of the CPU times for all the instances in the benchmark, and the number of solved instances. A cut-off limit of 3600 s was set. If a method X was cut off on an instance that was solved by at least one method then we count 3600 s towards the total CPU time of method X. Hence, if there is at least one cut-off for a method on instances solved by other methods, the sum gives a lower bound on it's total run time.

Confirming what is already known, results given in Table 2 show that AC is generally more efficient than stronger methods but there exist classes of problems (e.g. *ehi-85* and *bqwh-18*) where it is exponentially slower than SLCs. Comparing RNSAC to NPOAC, the former is usually better on classes where the extra pruning of SLCs does not pay off and AC excels (with the exception of *driver*), while the stronger pruning of the latter makes it a better option on problems where AC fails.

[2] http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/benchmarks.html.

**Table 2** Experimental results with dom/ddeg

| Benchmark | | AC | RNSAC | NPOAC | PrePeak | VarAdapt | ValAdapt |
|---|---|---|---|---|---|---|---|
| qwh-20 | ∑cpu | > 14,098 | 1196 | **826** | > 13,764 | 6058 | 3851 |
| 10 | Solved | 9 | 10 | 10 | 9 | 10 | 10 |
| bqwh-18 | ∑cpu | 4758 | 232 | **164** | 4409 | 818 | 1120 |
| 50 | Solved | 50 | 50 | 50 | 50 | 50 | 50 |
| Pigeons | ∑cpu | 1294 | > 3908 | > 4258 | **1234** | 2577 | > 3855 |
| 19 | Solved | 9 | 8 | 8 | 9 | 9 | 8 |
| qcp-10,15 | ∑cpu | > 16,799 | 2884 | **2666** | > 16,447 | 4990 | 4387 |
| 30 | Solved | 27 | 30 | 30 | 27 | 30 | 30 |
| Driver | ∑cpu | **33** | 1719 | 113 | 59 | 132 | 107 |
| 7 | Solved | 7 | 7 | 7 | 7 | 7 | 7 |
| comp25-10 | ∑cpu | > 18,047 | > 3601 | **2** | > 7514 | > 3683 | > 3627 |
| 10 | Solved | 5 | 9 | 10 | 9 | 9 | 8 |
| rand-23 | ∑cpu | **277** | 1289 | 5575 | 283 | 1,295 | 886 |
| 10 | Solved | 10 | 10 | 10 | 10 | 10 | 10 |
| Geometric | ∑cpu | **501** | 2819 | 8003 | **501** | 2423 | 1.461 |
| 20 | Solved | 20 | 20 | 20 | 20 | 20 | 20 |
| sgb | ∑cpu | 1249 | 5052 | > 8991 | **1,180** | 1517 | 1542 |
| 50 | solved | 23 | 23 | 21 | 23 | 23 | 23 |
| ehi-85 | ∑cpu | 1805 | **4** | 7 | 1742 | 8 | 355 |
| 20 | Solved | 20 | 20 | 20 | 20 | 20 | 20 |

Below each benchmark's name we give the number of instances. Cpu times are in secs and the best is highlighted with bold

VarAdapt and ValAdapt achieve a balance between the two extremes, being faster than AC on problems where it fails, and faster than the SLCs on problems where AC dominates. Comparing the two of them, it is interesting that their relative performance is quite diverse, as there can be large differences between them, even on an instance to instance basis within the same benchmark. However, there is no clear overall winner, with ValAdapt being more robust, except for the *ehi-85* class where the propagation technique of VarAdapt makes the instances trivially solvable.

On the other hand, PrePeak is not able to notably improve upon the performance of AC, except for the *composed* class. This does not seem to agree with results given in Woodward et al. (2018) where notable differences in favour of PrePeak were observed. This may be due to the different sets of benchmarks [we ran a subset of the instances run in Woodward et al. (2018)] and to the different SLC used [NPOAC used here and the truncated variant of POAC used in Woodward et al. (2018) are related but still different].

**Table 3** Experimental results with dom/wdeg

| Benchmark | | AC | RNSAC | NPOAC | PrePeak | VarAdapt | ValAdapt |
|---|---|---|---|---|---|---|---|
| qwh-20 | ∑cpu | 414 | 327 | 291 | 261 | **243** | 301 |
| 10 | Solved | 10 | 10 | 10 | 10 | 10 | 10 |
| rlfap | ∑cpu | **915** | > 10,344 | > 10,063 | **915** | > 8806 | > 8751 |
| 24 | Solved | 19 | 18 | 18 | 19 | 18 | 18 |
| Pigeons | ∑cpu | **1360** | > 3929 | > 4130 | 1443 | 3149 | > 3893 |
| 19 | Solved | 9 | 8 | 8 | 9 | 9 | 8 |
| qcp-15,20 | ∑cpu | 1942 | 1521 | **1187** | 2205 | 1569 | 3888 |
| 30 | Solved | 23 | 23 | 23 | 23 | 23 | 23 |
| Driver | ∑cpu | **4** | 433 | 106 | **4** | 67 | 78 |
| 7 | Solved | 7 | 7 | 7 | 7 | 7 | 7 |
| rand-23 | ∑cpu | **274** | 1442 | 6061 | 294 | 1856 | 2074 |
| 10 | Solved | 10 | 10 | 10 | 10 | 10 | 10 |
| Geometric | ∑cpu | 358 | 1965 | 5684 | **353** | 1890 | 1309 |
| 20 | Solved | 20 | 20 | 20 | 20 | 20 | 20 |
| sgb | ∑cpu | 1669 | > 9230 | > 13,081 | 1653 | **1458** | 2287 |
| 50 | Solved | 31 | 29 | 28 | 31 | 31 | 31 |
| Insertion | ∑cpu | 356 | > 4091 | > 4955 | **353** | 363 | 1400 |
| 24 | Solved | 20 | 19 | 19 | 20 | 20 | 20 |

Cpu times are in secs and the best is highlighted with bold

### 5.4.2 Dom/wdeg

Table 3 presents results from various benchmarks comparing the methods under the dom/wdeg variable ordering heuristic. The set of benchmarks is not the same as in Table 2 because, as it is known, dom/wdeg makes some problems very easy while it allows solving some problems that are out of reach for dom/ddeg. Examples of the former are *bqwh-18*, *comp25-10* and *ehi-18*. That is, the advantage of strong propagation over AC on these problems, displayed in Table 3, is wiped out by dom/wdeg.

Also, on problems that are still hard and the SLCs were dominant under dom/ddeg, the differences between them and AC are much smaller. In contrast, on the majority of problems AC is much more efficient than the SLCs. Hence, it is clear that maintaining a SLC is not a viable option when dom/wdeg is used. The same conjecture can be made about the adaptive methods VarAdapt and ValAdapt. On the few benchmarks where they outperform AC (e.g. *qcp*[3], *qwh*), the difference in efficiency is not important enough to outweigh their poor performance on the rest of the problems (especially for ValAdapt). Hence, there is no convincing evidence to support their use with dom/wdeg.

In contrast, PrePeak fares much better when coupled with dom/wdeg. It manages to at least match AC on most problems, but in contrast to its use with dom/ddeg, it is able to notably improve on the performance of AC on benchmarks where the SLCs are efficient. Note that, as in the case of ValAdapt, PrePeak is quite faster than AC on all the solved

[3] ValAdapt displays high total cpu time on *qcp* because of one instance where the interaction between propagation and dom/wdeg seems to mislead search. It is faster than AC on the rest of the instances.

**Table 4** The number and success ratio of singleton tests on 4 instances

| Instance | | AC | PrePeak | VarAdapt | ValAdapt | RVarAdapt | RVarVal |
|---|---|---|---|---|---|---|---|
| qcp15-120-9 | ST | | 515 | 952 K | 1.1 M | 572 K | 363 K |
| (qcp) | sr | | 0.24 | 0.06 | 0.10 | 0.22 | 0.33 |
| | Nodes | 503 K | 339 K | 44 K | 24 K | 236 K | 272 K |
| | cpu | 42 | 16 | 10 | 11 | 14 | 14 |
| qwh20-166-5 | ST | | 334 | 1.2 M | 1.5 M | 405 K | 256 K |
| (qwh) | sr | | 0.07 | 0.05 | 0.09 | 0.19 | 0.30 |
| | Nodes | 371 K | 309 K | 51 K | 20 K | 162 K | 230 K |
| | cpu | 41 | 21 | 21 | 23 | 17 | 19 |
| miles750-10 | ST | | 2.1 K | 91 M | 171 M | 12 M | 6 M |
| (sgb) | sr | | 0.08 | 0.04 | 0.03 | 0.27 | 0.47 |
| | Nodes | 6.2 M | 6.2 M | 2.6 M | 2.6 M | 3.3 M | 3.6 M |
| | cpu | 727 | 706 | 725 | 1,263 | 504 | 484 |
| geo50-20-75-1 | ST | | 7.4 K | 7 M | 4.8 M | 1 M | 593 K |
| (geometric) | sr | | 0.17 | 0.06 | 0.13 | 0.29 | 0.42 |
| | Nodes | 180 K | 179 K | 103 K | 71 K | 118 K | 133 K |
| | cpu | 22 | 23 | 99 | 88 | 31 | 25 |

instances of *qcp*, except for one hard instance, which is the culprit for the high total cpu time. If this instance is excluded then the CPU times for AC and PrePeak are 1074 and 677 s mber and success ratio of sin respectively.

Hence, PrePeak is by far the most efficient adaptive method under dom/wdeg, at least among the ones evaluated here. But as detailed, PrePeak is a carefully tuned method that relies on the setting of parameters. We have used the same parameter settings as in Woodward et al. (2018) and obtained good results, but not as good as in Woodward et al. (2018), despite the close relationship between the SLC used here (NPOAC) and the one in Woodward et al. (2018) (truncated POAC). It is not unlikely that different parameter settings may be required to obtain optimal performance if a quite different SLC is used.

Table 4 takes a closer look at the operation of the adaptive methods, revealing the reasons for the failure of VarAdapt and ValAdapt, and the success of PrePeak. We focus on four instances, each from a different benchmark. The first two are instances where all adaptive methods are successful, while the last two are instances where VarAdapt and ValAdapt fail to compete with AC, while PrePeak succeeds. Let us ignore the last two columns for now. For each adaptive method we give the number of singleton tests performed (ST) and the *success ratio* (sr) of the singleton tests. That is, the number of tests that resulted in domain reduction over ST. We also give the number of search tree nodes and cpu time in secs.

There is a huge difference in the number of singleton tests performed by VarAdapt and ValAdapt compared to PrePeak. In the first two instances the former methods succeed in cutting down the number of node visits made by AC by an order of magnitude, and as a result manage to speed up search, after making a million or so singleton tests with a low success ratio. On the other hand, PrePeak achieves similar speed-ups with only a few hundreds of tests, which are not all successful but evidently are very precisely targeted to cut down thrashing by detecting failures early.

In the third instance, VarAdapt makes many millions of tests with low success ratio to achieve a moderate reduction in the number of nodes compared to AC, resulting in a similar run time. ValAdapt slows down search by making even more tests with a lower success ratio, while PrePeak is the winner with only a couple of thousand tests. In the last instance, despite the millions of tests made by VarAdapt and ValAdapt, the search tree is not pruned by much compared to AC, and as a result, the overhead of the tests is reflected on cpu times. On the other hand, the much fewer tests made by PrePeak do not manage to cut down the search tree, but nevertheless avoid slowing down search.

To the defence of SLCs and the adaptive methods, it is important to note that dom/wdeg is designed for use with AC-like propagation, and it is not well suited to singleton-based SLCs. When AC results in a DWO, the constraint that resulted in the last value removal and will have its weight increased is easy to locate. In contrast, when a singleton consistency results in the DWO of a variable $x_i$, focusing on the constraint responsible is not as straightforward. Assuming that $a_i$ was the last value removed from $D(x_i)$, we have used a naive scheme that during the singleton test of $a_i$ finds the constraint that resulted in an empty domain by removing the last value from some domain in $\mathcal{P}_{x_i \leftarrow a_i}$, and only increases the weight of this constraint. It is quite likely that alternative schemes may give better results.

## 6 Randomization in adaptive propagation

Aiming at obtaining a simple adaptive propagation technique that is competitive when dom/wdeg is used, we now explore the use of randomization in the context of adaptive propagation. It is well known that the search process can greatly benefit from stochastic choices, that are usually associated with the variable and value ordering heuristics (Gomes et al. 1998). For example, when considering which variable to assign next, ties in the heuristic score of the variables are very often broken randomly. In contrast, the exploitation of randomness in the context of constraint propagation has received very little attention (Katriel and Van Hentenryck 2006; Mehta and van Dongen 2007).

We argue that incorporating an amount of randomness in the propagation process may actually be quite useful, especially when non-standard (strong) propagation is considered. Hence, we now describe one way to incorporate randomness in the VarAdapt method.

### 6.1 Incorporating randomness in VarAdapt

We propose a simple scheme that probabilistically decides whether to perform singleton tests or not during search by modifying the decision making process of the VarAdapt method. Instead of relying on the user-defined threshold $l$ to decide whether to apply a SLC on the value of a variable $x_i$ or not, the decision is taken based on a probability distribution that depends on the distance, in terms of consecutive revisions, between the current revision of $x_i$ and its most recent revision that resulted in a DWO.

Specifically, to obtain the randomized variant of VarAdapt, which we call RVarAdapt hereafter, we replace the condition in line 5 of Algorithm 4. We now execute lines 6,7 (i.e. call the SLC algorithm) with probability $\frac{1}{revision[i]-dwo[i]}$. Hence, RVarAdapt assigns a higher probability to invoke the SLC algorithm to revisions that are "close" to the most recent revision that caused the DWO of $D(x_i)$, and this probability fades for more distant revisions. The reasoning is that close revisions are more likely to cause a DWO, or at least

**Table 5** Experimental results of randomized methods with dom/wdeg

| Benchmar K | | AC | PrePeak | RVarAdapt | (SD) | RVarVal | (SD) (%) |
|---|---|---|---|---|---|---|---|
| qwh-20 | ∑cpu | 414 | 261 | 230 | 86% | **172** | 52 |
| 10 | solved | 10 | 10 | 10 | | 10 | |
| rlfap | ∑cpu | **915** | **915** | 1873 | 7% | 1019 | 5 |
| 24 | solved | 19 | 19 | 19 | | 19 | |
| Pigeons | ∑cpu | **1360** | 1443 | 2048 | 1.5% | 1510 | 1 |
| 19 | solved | 9 | 9 | 9 | | 9 | |
| qcp-15,20 | ∑cpu | 1941 | 2205 | > 4712 | – | **897** | 83 |
| 30 | solved | 23 | 23 | 23 | | 23 | |
| Driver | ∑cpu | **4** | **4** | 9 | 5% | **4** | 3 |
| 7 | solved | 7 | 7 | 7 | | 7 | |
| rand-23 | ∑cpu | **274** | 294 | 499 | 1% | 361 | 1 |
| 10 | solved | 10 | 10 | 10 | | 10 | |
| Geometric | ∑cpu | 358 | **353** | 473 | 1% | 392 | 1 |
| 20 | solved | 20 | 20 | 20 | | 20 | |
| sgb | ∑cpu | 1669 | 1653 | 1134 | 2% | **1061** | 1 |
| 50 | solved | 31 | 31 | 31 | | 31 | |
| Insertion | ∑cpu | 356 | 353 | **271** | 2% | 335 | 1 |
| 24 | solved | 20 | 20 | 20 | | 20 | |

value deletions, compared to distant ones, under the assumption that, especially in structured problems, successful revisions are often clustered (Stergiou 2008).

Note that although the probability distribution considered is quite "natural", alternative distributions could also be considered. For example, the probability that the SLC will be invoked could fade faster (or slower) as the distance between *revision*[*i*] and *dwo*[*i*] increases. Hence, the probability distribution constitutes a kind of parameter for RVarAdapt, albeit one that does not require detailed tuning from the user's part. This is why we refer to this method as "almost" parameter-free.

Table 5 presents results from various benchmarks showing the performance of RVarAdapt under the dom/wdeg variable ordering heuristic. We include results from AC and PrePeak as reference. The CPU times of RVarAdapt are computed by taking the mean over 50 runs for each instance. Let us ignore the (SD) columns for the time being.

As can be seen, randomization not only lifts the requirement for parameter setting from the user's part, but also significantly improves on the performance of VarAdapt on almost all benchmarks, and especially the ones where the SLCs are not competitive (e.g. *driver*). The only exception among all the tested benchmarks is *qcp*, and the reason for this is explained shortly. As can be seen in the corresponding column in Table 4, RVarAdapt not only cuts down the number of singleton tests, as expected, but achieves a much higher success ratio than the other adaptive methods, and this largely explains its good performance. But despite this, there are two problems:

- RVarAdapt is still notably inferior to PrePeak and to AC on some problems.
- There exist a few instances, belonging to *qcp* and to a lesser extent to *qwh*, where different runs of RVarAdapt display very large variance.

**Table 6** The variance of randomized adaptive propagation on 5 instances

| Instance | | AC | | RVarAdapt | | RVarVal | |
|---|---|---|---|---|---|---|---|
| | | Nodes | cpu | Nodes | cpu | Nodes | cpu |
| scen11-f9 | Min | 101 K | 23.0 | 15 K | 27.0 | 29 K | 21.1 |
| (rlfap) | Max | | | 17 K | 33.0 | 35 K | 25.6 |
| | Mean | | | 16 K | 30.6 | 31 K | 22.6 |
| | SD | | | 1,036 | 2.1 | 1,851 | 1.2 |
| games120-8 | Min | 3.2 M | 29.3 | 1.7 M | 19.5 | 1.8 M | 20.5 |
| (sgb) | Max | | | 1.7 M | 20.6 | 1.8 M | 21.5 |
| | Mean | | | 1.7 M | 20.0 | 1.8 M | 21.0 |
| | SD | | | 3.4 K | 0.3 | 855 | 0.2 |
| geo50-20-75-1 | Min | 180 K | 23.0 | 118 K | 30.4 | 133 K | 26.1 |
| (geometric) | Max | | | 119 K | 31.9 | 135 K | 27.2 |
| | Mean | | | 118 K | 31.0 | 134 K | 26.5 |
| | SD | | | 564 | 0.4 | 546 | 0.3 |
| qcp20-187-8 | Min | 1.2 M | 190 | 337 K | 43 | 404 K | 40 |
| (qcp) | Max | | | 14 M | 1590 | 4.8 M | 451 |
| | Mean | | | 2.3 M | 291 | 1.2 M | 118 |
| | SD | | | 4.1 M | 467 | 1.3 M | 124 |
| qwh20-166-3 | Min | 437 K | 50.1 | 39 K | 4.1 | 64 K | 5.6 |
| (qwh) | Max | | | 413 K | 41.1 | 317 K | 28.7 |
| | Mean | | | 176 K | 17.9 | 191 K | 16.6 |
| | SD | | | 155 K | 15.4 | 101 K | 8.6 |

Regarding the second problem, Table 6 gives the mean, min, max, and standard deviation of the CPU time and the number of nodes from 50 runs of RVarAdapt on representative instances of five classes. We include results from AC as a reference point. In the first three instances there is small variance, as displayed by the small difference between the minimum and maximum values and the value of the standard deviation. This stable behaviour of RVarAdapt is characteristic across all instances of these classes, and is the norm in all the tried classes, except for *qcp* (mainly) and *qwh*.

This can be verified by observing the SD column for RVarAdapt in Table 5, where we give the mean standard deviation over the instances of each class, as a percentage of the mean run time. That is, for each instance in a class we compute the standard deviation as a percentage of the mean cpu time (taken over the 50 runs for this particular instance), and then we compute and report the average of these percentages over all the instances of the class. Note that there is no value for the mean standard deviation in the *qcp-15,20* class. This is due to time-outs that occured on one specific instance, as we shortly explain.

Going back to Table 6, we can observe that on *qcp-20-187-8* we have a very large variance, as the shortest among the 50 runs took 43 secs and the longest 1590, while the standard deviation is also very high. A similar, but milder, picture is given by the instance from the *qwh* class. However, the most extreme case was *qcp-20-187-11* which was solved by all other methods in less than 10 sbut two out of the 50 runs of RVarAdapt reached the cut-off limit, while the rest took less than 1 s. This accounts for the high cpu time total of RVarAdapt in *qcp* given in Table 5 (we counted 3600 secs for this instance), while on average

**Table 7** Cpu run times (in s) from specific instances with dom/wdeg

| Instance | AC | PrePeak | RVarAdapt | RVarVal |
|---|---|---|---|---|
| qwh-20-2 | 98 | 64 | 28 | **20** |
| qwh-20-3 | 50 | 29 | 17 | **15** |
| rlfap-11-f8 | **40** | **40** | 59 | 44 |
| rlfap-11-f7 | 383 | **381** | 801 | 430 |
| pigeons-12 | **102** | 109 | 139 | 115 |
| pigeons-13 | **1248** | 1324 | 1895 | 1383 |
| qcp-15-120-10 | 107 | 40 | **7** | **7** |
| qcp-20-187-0 | 280 | 95 | 72 | **34** |
| qcp-20-187-9 | **237** | 310 | 245 | 451 |
| rand-23-47 | **35** | **35** | 58 | 45 |
| rand-23-51 | **36** | 39 | 62 | 48 |
| geometric-50-20-12 | **52** | **52** | 69 | 57 |
| geometric-50-20-94 | 40 | **38** | 50 | 42 |
| homer-10 | 585 | 586 | 335 | **315** |
| miles750-10 | 727 | 706 | 504 | **484** |
| queen11-11-8 | 10 | 10 | 6 | **6** |

Cpu times are in secs and the best is highlighted with bold

RVarAdapt is considerably faster than AC and VarAdapt in this class of problems. This is also the reason for the empty entry in the SD column.

## 6.2 A combined method

As the results obtained so far suggest that in order for an adaptive method to be successful it it necessary that the number of singleton tests is kept as low as possible, the success ratio of the tests as high as possible, and crucially, the tests are performed "at the right moment", we consider the integration of methods RVarAdapt and ValAdapt. This method, called RVarVal, applies the SLC on a value $a_i \in D(x_i)$ at line 7 of Algorithm 4 when both the probability $\frac{1}{revision[i]-dwo[i]}$ is verified and the condition of ValAdapt regarding the distance of $a_i$'s supports from the domain ends holds. In this way we hope to avoid singleton checks that are unlikely to result in value removals.

Tables 4 and 5 show that indeed RVarVal is successful in its goal. As the last column in Table 4 demonstrates, RVarVal manages to cut down the number of singleton tests even more compared to RVarAdapt, and importantly, the success ratio gets close to 50% in some cases, which is significantly higher than the other adaptive methods. In Table 5 we can see that this is reflected on the cpu times, as RVarVal is very competitive with both AC and PrePeak in mean run times, and it not only beats them on problems where the SLCs perform well (*qcp* and *qwh*), but also on some problems where the SLCs are heavily outperformed (*sgb* and *insertion*). But equally importantly, RVarVal is not significantly slower than AC or PrePeak on problems where they excel, including random ones such as *rand-23*.

As Tables 5 and 6demonstrate, the large variance displayed by RVarAdapt on *qcp* and *qwh* is alleviated to a certain degree, but not completely resolved. As in the case of RVarAdapt, these are the only classes where significant variance among different runs appears, but this perhaps is an inherent drawback of randomization techniques that cannot be avoided.

Finally, in Table 7 we give CPU times for the compared methods on specific instances from various problems classes. We include this table to demonstrate that there can be substantial differences in favour of RVarVal (in most cases), because cumulative CPU times sometimes tend to flatten things out, especially when there exist many instance on which the methods perform similarly, as is the case here. Table 7 shows that RVarVal can be many times faster than both AC and PrePeak (from 5 up to 15 times on qcp and qwh instances). At the same time, it was never slower than AC by more than 25% on the whole range of instances that we tried, except for *qcp-20-187-9* where AC is around two times fastery[4].

## 7 Discussion and future work

We can make two general observations based on the analysis and the experimental results presented in this paper:

1. Although the choice of SLC is important, it is not the most crucial factor that determines the performance of adaptive methods, provided that a reasonable choice is made (i.e. a relatively low-cost SLC is used). Based on the correlation between the success ratio of calls to the SLC and the performance of the methods, as demonstrated by the results obtained by RVarVal, and also by the success of PrePeak with very few singleton tests, we conjecture that the most important factor is the precise targeting of the SLC's invocation. Hence, the goal of an adaptive method should be to keep the number of invocations low, but to execute them "at the right time". Towards this, it would be interesting to investigate the possibility of developing a method that integrates variable and value oriented approaches with node oriented ones.
2. For the case of binary constraints, where this paper focuses in terms of the experimental evaluation, the obtained results (from this paper and the existing literature) indicate that very few adaptive methods are really competitive with an optimized implementation of MAC with dom/wdeg. Specifically, PrePeak and the method proposed in this paper. Cases where these two methods are heavily ouperformed by a standard solver seem very rare, while there exist quite a few instances where they can be an order of magnitude faster. However, there do not seem to be entire classes of problems where the adaptive methods (even the best ones) heavily outperform the standard approach. But further experimentation is required to validate this, especially with non-binary problems.

The experimental evaluation that we have presented is limited to binary problems, as is our final proposed method RVarVal. However, method RVarAdapt, which incorporates randomization into VarAdapt, can be applied to non-binary problems. To extend RVarVal to table constraints, one can integrate RVarAdapt with the technique described in Woodward et al. (2014). But for the case of global constraints things are not as straightforward. One idea could be to exploit solution counting algorithms, for cases of constraints where such algorithms exist (Pesant et al. 2012).

In any case, it is necessary to evaluate existing methods, as well as the ones proposed here, on non-binary problems, so that a clearer picture of the usefulness of adaptive

[4] This is due to the interplay between the propagation mechanism and dom/wdeg. RVarVal takes 4.7 million nodes to solve this instance while AC takes 2.7 million.

propagation is obtained and the conjectures outlined above are validated. Apart from this obvious line of work, other avenues can also be explored in the future:

- There is potential to integrate randomization within node and value oriented methods. In fact, we have tried one such technique for the ValAdapt method. Specifically, we experimented with a method where the application of the SLC, in relation to the distance of the first support discovered from the end of the domain, was not determined simply by checking if the threshold was exceeded or not. But it depended on a probability distribution. However, the results were not good. This method was worse than the standard ValAdapt method. So, we did not include any reference to this experiment in the paper. However, there are other relevant avenues to explore, and we intend to do this in the future.
- As detailed in our review of the literature on adaptive propagation, existing methods focus on switching between standard (G)AC propagation and a non-standard SLC like SAC. In all cases, the LCs considered are applied on all values in the variables' domains. However, (G)AC is not always the standard propagation method for all types of constraints in CP solvers. The weaker property of bounds consistency is also widely used on some constraints (e.g. arithmetic constraints). We are not aware of any adaptive propagation method that considers bounds consistency as an option when deciding to switch between different levels of consistency during search. This would be interesting to explore in the future for at least two reasons:

    1. Bounds consistency can offer a third option that is cheaper than (G)AC when considering which LC to apply. In order for this idea to work, we would have to use an adaptive technique that supports switching between more than two LCs, e.g. the MAB method, or to modify methods such as PrePeak and VarAdapt that have been proposed specifically for switching between two LCs.
    2. The SLC that is chosen by the given adaptive method during search could itself be based on bounds consistency [i.e. an extension of bounds consistency that achieves stronger pruning, such as the method of Bessiere et al. (2015)]. The advantage of such an approach is that bounds consistency, and therefore also its extensions to stronger consistencies, are typically quite cheap compared to LCs that reason over the entire domain, such as GAC and its extensions. Hence, the cost of applying the stronger property would be even lower.

- Another line of work that needs to be followed concerns the development of general-purpose variable ordering heuristics that are better suited to SLCs. A relevant study on dom/wdeg can be found in Woodward and Choueiry (2017), but different heuristics can also be considered. One promising candidate is the *Activity* heuristic of Michel and Van Hentenryck (2012) which like VarAdapt monitors the activity of the variables during search, and makes branching decisions based on this activity.
- Finally, the potential of machine learning in guiding the selection of propagation techniques has not been explored enough. Specific reinforcement learning techniques have been used to build node-oriented adaptive methods [learning of propagation policies Epstein et al. (2005) and MAB Balafrej et al. (2015)], but no ML methods have been exploited at other levels of granularity. Also, alternative reinforcement learning methods (e.g. deep reinforcement learning), as well as unsupervised learning techniques, have not been tried at all.

## 8 Conclusion

We have proposed a classification of adaptive constraint propagation methods according to the level of granularity where the decisions about which local consistency to apply are taken. Having identified node, variable, and value oriented methods, we classified the existing approaches from the literature, reviewed and evaluated them using several criteria.

We also presented an experimental evaluation that includes one representative method from each category. Results showed that although simple variable and value oriented techniques can be quite useful when the older dom/ddeg variable ordering heuristic is used, their advantages are diminished when the state-of-the-art dom/wdeg is used. In contrast, the more sophisticated node oriented technique PrePeak is quite efficient under dom/wdeg. However, an integration of the variable and value oriented methods, boosted by a simple randomization scheme, results in an adaptive propagation technique that is very competitive, simple, and (almost) parameter-free.

## References

Balafrej A, Bessiere C, Bouyakh E, Trombettoni G (2014) Adaptive singleton-based consistencies. In: Proceedings of the twenty-eighth AAAI conference on artificial intelligence, pp 2601–2607

Balafrej A, Bessiere C, Coletta R, Bouyakh E (2013) Adaptive parameterized consistency. In: Proceedings of the CP-2013, pp 143–158

Balafrej A, Bessiere C, Paparrizou A (2015) Multi-armed bandits for adaptive constraint propagation. In: Proceedings of the twenty-fourth international joint conference on artificial intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25–31, 2015, pp 290–296

Bennaceur H, Affane M (2001) Partition-k-AC: an efficient filtering technique combining domain partition and arc consistency. In: Principles and practice of constraint programming—CP 2001, 7th international conference, CP 2001, Paphos, Cyprus, November 26–December 1, 2001, Proceedings, pp 560–564

Berlandier P (1995) Improving domain filtering using restricted path consistency. In: Proceedings of IEEE CAIA'95, pp 32–37

Bessiere C, Cardon S, Debruyne R, Lecoutre C (2011) Efficient algorithms for singleton ARC consistency. Constraints 16:25–53

Bessière C, Régin J (1996) MAC and combined heuristics: two reasons to forsake FC (and CBJ?). In: Proceedings of CP'96. Cambridge MA, pp 61–75

Bessière C, Stergiou K, Walsh T (2008) Domain filtering consistencies for non-binary constraints. Artif Intell 172(6–7):800–822

Bessiere C, Paparrizou A, Stergiou K (2015) Strong bounds consistencies and their application to linear constraints. Proc AAAI 2015:3717–3723

Borrett J, Tsang E, Walsh N (1996) Adaptive constraint satisfaction: the quickest first principle. In: Proceedings of ECAI'96, pp 160–164

Boussemart F, Heremy F, Lecoutre C, Sais L (2004) Boosting systematic search by weighting constraints. In: Proceedings of ECAI'04, pp 482–486

Debruyne R, Bessière C (1997) From restricted path consistency to max-restricted path consistency. In: Proceedings of CP-97, pp 312–326

Debruyne R, Bessière C (2001) Domain filtering consistencies. JAIR 14:205–230

Dechter R, van Beek P (1997) Local and global relational consistency. Theoret Comput Sci 173:283–308

El Sakkout H, Wallace M, Richards B (1996) An instance of adaptive constraint propagation. In: Proceedings of CP'96, pp 164–178

Epstein S, Freuder EC, Wallace R, Morozov A, Samuels B (2002) The adaptive constraint engine. In: Proceedings of CP-2002, pp 525–540

Epstein S, Freuder E, Wallace R, Li X (2005) Learning propagation policies. In: Proceedings of the international workshop on constraint propagation and implementation, pp 1–15

Freuder E, Elfe C (1996) Neighborhood inverse consistency preprocessing. In: Proceedings of AAAI'96, pp 202–208

Freuder E, Wallace R (1991) Selective relaxation for constraint satisfaction problems. In: Proceedings of ICTAI'91

Gomes C, Selman B, Kautz H (1998) Boosting combinatorial search through randomization. In: Proceedings of AAAI-98, pp 431–437

Haralick R, Elliot G (1980) Increasing tree search efficiency for constraint satisfaction problems. Artif Intell 14:263–313

Janssen P, Jégou P, Nouguier B, Vilarem M (1989) A filtering process for general constraint satisfaction problems: achieving pairwise consistency using an associated binary representation. In: Proceedings of the IEEE workshop on tools for artificial intelligence, pp 420–427

Karakashian S, Woodward R, Reeson C, Choueiry B, Bessière C (2010) A first practical algorithm for high levels of relational consistency. In: Proceedings of AAAI'10, pp 101–107

Katriel I, Van Hentenryck P (2006) Randomized filtering algorithms. Technical report CS-06-09, Brown University

Lecoutre C (2011) Str2: optimized simple tabular reduction for table constraints. Constraints 16(4):341–371

Lecoutre C, Paparrizou A, Stergiou K (2012) Extending STR to a higher-order consistency. In: Proceedings of AAAI'13

Lhomme O (2005) Quick shaving. In: Proceedings of AAAI'05, pp 411–415

Likitvivatanavong C, Wei X, Yap RHC (2014) Higher-order consistencies through GAC on factor variables. In: Principles and practice of constraint programming—20th international conference, CP 2014. Proceedings, pp 497–513

Mehta D, van Dongen M (2007) Probabilistic consistency boosts MAC and SAC. In: Proceedings of IJCAI'07, pp 143–148

Michel L, Van Hentenryck P (2012) Activity-based search for black-box constraint programming solvers. In: Integration of AI and OR techniques in contraint programming for combinatorial optimzation problems—9th international conference, CPAIOR 2012, Nantes, France, May 28–June 1, 2012. Proceedings, pp 228–243

Mackworth AK (1977) Consistency in networks of relations. Artif Intell 8(1):99–118

Montanari U (1974) Network of constraints: fundamental properties and applications to picture processing. Inf Sci 7:95–132

Paparrizou A, Stergiou K (2012) Evaluating simple fully automated heuristics for adaptive constraint propagation. In: IEEE 24th international conference on tools with artificial intelligence, ICTAI 2012, Athens, Greece, November 7–9, 2012, pp 880–885

Paparrizou A, Stergiou K (2017) On neighborhood singleton consistencies. In: Proceedings of the twenty-sixth international joint conference on artificial intelligence, IJCAI 2017, Melbourne, Australia, August 19–25, 2017, pp 736–742

Pesant G, Quimper C, Zanarini A (2012) Counting-based search: branching heuristics for constraint satisfaction problems. J Artif Intell Res 43:173–210

Prosser P, Stergiou K, Walsh T (2000) Singleton consistencies. In: Proceedings of CP-2000, Melbourne, pp 353–368

Sabin K,Freuder EC (1997) Understanding and improving the MAC algorithm. In: Proceedings of CP-1997, pp 167–181

Stergiou K (2008) Heuristics for dynamically adapting propagation. In: Proceedings of ECAI'08, pp 485–489

Stergiou K (2008) Strong domain filtering consistencies for non-binary constraint satisfaction problems. Int J Artif Intell Tools 17(5):781–802

Stergiou K (2009) Heuristics for dynamically adapting propagation in constraint satisfaction problems. AI Commun 22(3):125–141

Szymanek R, Lecoutre C (2008) Constraint-level advice for shaving. In: Proceedings of ICLP'08, pp 636–650

Ullmann JR (2007) Partition search for non-binary constraint satisfaction. Inf Sci 177(18):3639–3678

Wallace R (2015) SAC and neighbourhood SAC. AI Commun 28(2):345–364

Wallace R (2016) Neighbourhood SAC: extensions and new algorithms. AI Commun 29(2):249–268

Woodward R, Choueiry B (2017) Weight-based variable ordering in the context of high-level consistencies. CoRR arXiv:abs/1711.00909

Woodward R, Choueiry B, Bessiere C (2017) Cycle-based singleton local consistencies. In: Proceedings of the Thirty-First AAAI conference on artificial intelligence, February 4–9, 2017, San Francisco, California, USA, pp 5005–5006

Woodward R, Choueiry B, Bessiere C (2018) A reactive strategy for high-level consistency during search. In: Proceedings of the twenty-seventh international joint conference on artificial intelligence, IJCAI 2018, July 13–19, 2018, Stockholm, Sweden, pp 1390–1397

Woodward R, Schneider A, Choueiry B, Bessiere B (2014) Adaptive parameterized consistency for non-binary CSPS by counting supports. In: Principles and practice of constraint programming—20th international conference, CP 2014, Lyon, France, September 8–12, 2014. Proceedings, pp 755–764

Woodward RJ, Karakashian S, Choueiry BY, Bessière C (2011) Solving difficult CSPS with relational neighborhood inverse consistency. In: Proceedings of AAAI, pp 112–119

Zanarini A, Pesant G (2009) Where can I get a quick shave?. In: Proceedings of the CP-09 workshop on constraint modelling and reformulation, pp 186–200