Heuristics for Dynamically Adapting Propagation in Constraint Satisfaction Problems

Kostas Stergiou [a]

[a] *Department of Information & Communication Systems Engineering*
University of the Aegean, Greece
E-mail: konsterg@aegean.gr

Building adaptive constraint solvers is a major challenge in constraint programming. An important line of research towards this goal is concerned with ways to dynamically adapt the propagation method applied on the constraints of the problem during search. In this paper we present a heuristic approach to this problem based on the monitoring of propagation events like value deletions and domain wipeouts. We develop a number of heuristics that allow the constraint solver to dynamically switch between a weaker and cheap local consistency and a stronger, but more expensive one, when certain conditions are met. The success of this approach is based on the observation that propagation events for individual constraints in structured problems mostly occur in clusters of nearby revisions. Hence, parts of the search space where certain constraints are *highly active* can be identified and exploited paving the way for the informed use of constraint propagation techniques. In this paper we first give some experimental results displaying the clustering of propagation events in structured binary CSPs. Then we present simple heuristics that exploit this clustering to efficiently switch between different local consistencies on individual constraints during search. Finally, we make an experimental study on various binary CSPs demonstrating the effectiveness of the proposed heuristics.

Keywords: first keyword, second keyword, third keyword

1. Introduction

Constraint Programming (CP) is nowadays considered an established and successful paradigm for modelling and solving hard combinatorial problems from areas such as planning and scheduling, timetabling, resource allocation, bioinformatics, etc. However, the CP community still has many challenges to face in order to make CP technology even more widely known and used. One of the most important such challenges is "ease of use" since modelling real problems as constraint satisfaction or optimization problems is still largely dependent on specialized skills and expertise.

An important aspect of the modelling process is the choice of propagation method for the various constraints in the problem. This was not an issue in the early days of CP when (generalized) arc consistency or even lesser levels of local consistency like forward checking were the predominant propagation techniques used. However, these days many global constraints come with an array of propagators with different filtering power and cost [29]. Also, the number of generic local consistency methods that can be found in the literature for either binary or non-binary constraints has risen significantly [1,4]. Therefore, building constraint solvers that can efficiently exploit the wealth of available propagation techniques is a major challenge. One way to achieve this goal is by building solvers that are able to dynamically adapt the constraint propagation method applied on the constraints during search. Constraint solvers typically apply (generalized) arc consistency (G)AC, or a weaker consistency property like bounds consistency, during search. The choice of the appropriate propagation method for each constraint is typically left to the modeler who has to decide upon this based on the features of the problem and the available propagators for each constraint. Obviously, this requires a significant amount of expertise.

Although many propagation methods stronger than (G)AC have been proposed, their practical usage is limited as they are only applied during preprocessing, if at all. The main obstacle is the high time and in some cases space complexity of the algorithms that can achieve these consistencies. This, coupled with the implicit general assumption that constraints should be propagated with a predetermined local consistency throughout search, makes maintaining strong consis-

AI Communications
ISSN 0921-7126, IOS Press.

tencies an infeasible option, except for some specific CSPs. One way to overcome the high complexity of maintaining a strong consistency while retaining its benefits is to dynamically evoke it during search only when certain conditions are met. There have been some works along this line in the literature, mainly focusing on methods to switch between GAC and weaker consistencies [13,15,24,20].

In this paper we present a heuristic approach to the problem of dynamically adapting constraint propagation. We develop a number of heuristics that allow the constraint solver to dynamically switch between a weaker and cheap local consistency and a stronger, but more expensive one, when certain conditions are met. The proposed heuristics operate by monitoring and reacting to propagation events like value deletions and domain wipeouts. The success of our approach is based on the observation that propagation events caused by individual constraints in structured problems are highly clustered. That is, constraint activity during search is not uniformly distributed among the revisions of the constraints. On the contrary it is highly clustered as value deletions and domain wipeouts caused by individual constraints largely occur in clusters of nearby revisions. Hence, parts of the search space where certain constraints are *highly active* can be identified and exploited. This is a new and interesting observation that may pave the way to the informed clever use of constraint propagation techniques during search.

We start by giving some experimental results displaying the clustering of propagation events in structured binary CSPs. Then we present simple heuristics that exploit this clustering to efficiently switch between different local consistencies on individual constraints during search. The proposed heuristics achieve this by monitoring the activity of the constraints in the problem and triggering a switch between different propagation methods on individual constraints once certain conditions are met. For example, one of the heuristics works as follows. It applies a weak consistency on each constraint c until a revision of c results in a domain wipeout. Then it switches to a strong consistency and applies it on c for the next few revisions. If no further domain wipeout occurs during these revisions, it switches back to the weaker consistency. As a case study we experiment with binary problems using Arc Consistency as the weak consistency and max Restricted Path Consistency as the strong one. An experimental study on various binary CSPs demonstrates the effectiveness of the proposed heuristics. We show that the most efficient heuristics can be up to an order of magnitude faster than MAC, i.e. the standard search algorithm for binary CSPs, on hard instances.

Besides achieving faster problem solving, the work presented here contributes towards one of the most important goals of CP: ease of use. By allowing for the solver to dynamically change and readapt the way it propagates individual constraints, some of the burden of efficient modelling can be lifted from the shoulders of the user.

The paper is structured as follows. In Section 2 we give the necessary background and definitions. Section 3 makes an empirical investigation of the clustering of propagation events. Section 4 presents a number of heuristics for dynamically adapting propagation. In Section 5 we perform an experimental study of the proposed heuristics on binary CSPs. Section 6 discusses related work. Finally, in Section 7 we conclude and point out directions for future work.

2. Background

A *Constraint Satisfaction Problem* (CSP) is defined as a tuple (X, D, C) where: $X = \{x_1, \ldots, x_n\}$ is a set of n variables, $D = \{D(x_1), \ldots, D(x_n)\}$ is a set of domains, one for each variable, and $C = \{c_1, \ldots, c_e\}$ is a set of e constraints. Each constraint c is a pair $(var(c), rel(c))$, where $var(c) = \{x_1, \ldots, x_k\}$ is an ordered subset of X, and $rel(c)$ is a subset of the *Cartesian* product $D(x_1)\text{x}\ldots\text{x}D(x_k)$ that specifies the allowed combinations of values for the variables in $var(c)$. We denote the assignment of a value a_i to variable x_i by the pair (x_i, a_i). Each tuple $\tau \in rel(c_i)$ is an ordered list of values $(a_1, \ldots, a_k)$ such that $a_j \in D(x_j)$,$j = 1, \ldots, k$. A tuple $\tau \in rel(c_i)$ is *valid* iff none of the values in the tuple has been removed from the domain of the corresponding variable. The process which verifies whether a given tuple is allowed by a constraint c or not is called a *constraint check*. A constraint c can be either defined *extensionally* by explicitly giving $rel(c)$, or (usually) *intensionally* by implicitly specifying $rel(c)$ through a predicate or arithmetic function.

In a binary CSP, a directed constraint c, with $var(c) = \{x_i, x_j\}$, is *arc consistent* (AC) iff for every value $a_i \in D(x_i)$ there exists a value $a_j \in D(x_j)$ s.t. the 2-tuple $<(x_i, a_i), (x_j, a_j)>$ satisfies c. In this case (x_j, a_j) is called an AC-support of (x_i, a_i) on c. A problem is AC iff there is no empty domain in D and all the (directed) constraints in C are AC. A variable x_i is *singleton arc consistent* (SAC) iff for each value

$a_i \in D(x_i)$ after assigning a_i to x_i and applying AC in the problem there is no empty domain [10].

A directed constraint c, with $var(c) = \{x_i, x_j\}$, is *max restricted path consistent* (maxRPC) iff it is AC and for each value (x_i, a_i) there exists a value $a_j \in D(x_j)$ that is an AC-support of (x_i, a_i) s.t. the 2-tuple $<(x_i, a_i), (x_j, a_j)>$ is *path consistent* (PC) [10]. A tuple $<(x_i, a_i), (x_j, a_j)>$ is PC iff for any third variable x_m there exists a value $a_m \in D(x_m)$ s.t. (x_m, a_m) is an AC-support of both (x_i, a_i) and (x_j, a_j). In this case we say that (x_j, a_j) is a maxRPC-support of (x_i, a_i) on c.

Following [11], we call a consistency property A *stronger* than B iff in any problem in which A holds then B holds, and *strictly stronger* iff it is stronger and there is at least one problem in which B holds but A does not.

The standard complete method for solving CSPs is through backtracking tree search. At each step of the search process, usually called a *choice point*, a variable is assigned to one of its available values and constraint propagation is triggered to propagate the effects of this assignment. For solvers that employ 2-way branching choice points may also correspond to the removal of value from a domain. Constraint propagation is typically implemented through a list which may hold constraints, variables, or propagators depending on the particular solver. If propagation results in the removal of all values from a variable's domain, we have a *domain wipeout* (DWO) in which case the search algorithm backtracks to the last choice point undoing the intermediate effects of propagation.

In this paper we assume the use of a constraint-oriented scheme for propagation where the propagation list, implemented as a queue or as a stack, handles constraints. In such a scheme a constraint is added to the list once a value in the domain of some variable involved in the constraint is deleted. Constraints are repeatedly removed from the list and are revised, while any further value deletions may result in new constraints being added to the list. This process terminates when the list empties or a DWO occurs.

The *revision* of a binary constraint c, with $var(c) = \{x_i, x_j\}$, using a local consistency A is the process of checking whether the values of x_i verify the property of A. For example, the revision of c using AC verifies if all values in $D(x_i)$ have AC-supports on c. We say that a revision is *fruitful* if it deletes at least one value, while it is *redundant* if it achieves no pruning. A DWO-*revision* is one that causes a DWO. That is, a revision that deletes the last remaining value(s) from a domain. In the following we will say that a constraint is *DWO-active* during a run of a search algorithm if at least one of its revisions was a DWO-revision during the search process. Accordingly, we will call a constraint *deletion-active* if it deleted at least one value from a domain and *deletion-inactive* if it caused no pruning at all.

A standard search algorithm for solving binary CSPs is MAC (maintaining arc consistency) [23,2]. This algorithm applies AC to all constraints in the problem throughout search. Algorithms such as MAC use variable (and to a lesser extent value) ordering heuristics to guide search [28]. One of the most efficient general purpose variable ordering heuristics that have been proposed is *dom/wdeg* [6]. This heuristic uses information derived from conflicts, in the form of DWOs, and stored as constraint weights to guide search. A weight is assigned to each constraint and it is initially set to one. Each time a constraint causes a conflict, its weight is incremented by one. Each variable is associated with a *weighted degree*, 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. This heuristic is among the most efficient, if not *the* most efficient, general-purpose heuristics for CSPs. Following the work of [6], Grimes and Wallace proposed alternative conflict-driven heuristics that consider value deletions as the basic propagation events associated with constraint weights [16]. That is, the weight of a constraint is incremented each time the constraint causes one or more value deletions. The efficiency of all the proposed conflict-directed heuristics is due to their ability to learn though conflicts encountered during search. As a result they can guide search towards hard parts of the problem and identify *contentious* constraints [16].

3. Constraint Activity during Search

It has been recognized, for example in [20], that in many, mainly structured, problems only few of the constraint revisions that occur during search are fruitful while some constraints do not cause any DWOs or even are deletion-inactive during the run of a search algorithm despite being revised many times. For example, when solving the scen11 radio links frequency assignment (RLFA) problem with MAC equipped with dom/wdeg, only 27 of the 4103 constraints in the problem were identified as weight-active, while 1921 constraints were deletion-inactive.

Hence, it would be desirable to apply a strong consistency only when it is likely that it will prune many values and avoid using such a consistency when the expected pruning is non-existent or very low. The problem is that estimating the likelihood of pruning in order to target strong propagation accordingly is very difficult. This is because the activity of the constraints in a problem depends on the structure of the problem since constraints in difficult local subproblems are more likely to cause deletions and domain wipeouts, especially if a heuristic like dom/wdeg that can identify such subproblems is used. On top of that and due to the complex interactions that may exist between constraints, the activity also depends on the search algorithm, the propagation method, the variable ordering heuristic, and on the order in which constraints are propagated. For example, when solving scen11 with an algorithm that applies maxRPC on each constraint and dom/wdeg for variable ordering, 29 constraints were weight-active with only 13 of these identified as weight-active by both this algorithm and MAC.

Importantly, many revisions of the constraints that are weight-active and deletion-active are redundant or achieve very little pruning. To investigate how the fruitful revisions of the constraints are distributed along the time-line of their revisions we run experiments on some structured and random binary problems and recorded the following information for each constraint c:

- The number of times c was revised. To record this we simply used a counter $revision(c)$ that was incremented by one each time c was revised.
- The value of $revision(c)$ for each fruitful revision of c. To record this information we used an, initially empty, list of Boolean variables. An element was added to the list each time c was revised. The Boolean variable of the element was set to 0 or 1 depending on whether the revision was redundant or fruitful respectively. After search terminated the list's length was equal to the total number of times c was revised and it offered insight on the "history" of c's fruitful revisions.
- The value of $revision(c)$ for each DWO-revision of c. This information was also recorded using a list in a similar way as above. After search terminated this list offered insight on the "history" of c's DWO-revisions.

An analysis and visualization of the results obtained revealed interesting patterns. The four plots in Figure 1 demonstrate how the number of DWOs (y-axis) caused by 4 sample constraints increases as constraint revisions (x-axis) occur throughout search. That is, a data point with coordinates (i, j) corresponds to the j-th DWO-revision of the constraint, which occurred at the i-th time it was revised. The algorithm used is MAC + dom/wdeg and the sample constraints are taken from three structured and one random problem. Considering that heuristic dom/wdeg was used, we can also view data point (i, j) as giving the weight of the constraint at the i-th time it was revised.

As we can see in Figure 1, DWO-revisions in the three structured problems form clusters of successive or very close calls to the revision procedure, with the exception of a few outliers. This implies that once a DWO-revision of a constraint c occurs, it is likely that c will again cause pruning and possibly even DWOs within its few subsequent revisions. In contrast to structured problems, DWO-revisions in the random instance are distributed in a much more uniform way along the line of revisions.

Similar patterns to the ones of Figure 1 occur with respect to value deletions. The two plots in Figure 2 demonstrate how the number of fruitful revisions (y-axis) caused by 2 sample constraints increases as constraint revisions (x-axis) occur throughout search. That is, a data point with coordinates (i, j) corresponds to the j-th fruitful revision of the constraint, which occurred at the i-th time it was revised. As with DWO-revisions, fruitful revisions appear clustered within consecutive or close revisions. Of course in this case the clusters are closer to one another since value deletions occur much more often than DWOs.

Similar results with respect to the clusterness of propagation events were obtained when an algorithm that applied maxRPC was used in place of MAC. Figure 3 shows the DWO-revisions and value deletions caused by this algorithm (with dom/wdeg) for one sample constraint taken from scen11. This constraint was revised 1020 times during search, but only 85 of these revisions were fruitful and 64 of those were DWO-revisions. Naturally maxRPC, being stronger, usually displays a higher percentage of fruitful revisions compared to AC. But redundant revisions for maxRPC heavily penalize the run times and such revisions still appear in large numbers.

Another interesting observation is that in the structured problems the percentage of DWO-revisions to total revisions is in general low and there are also many

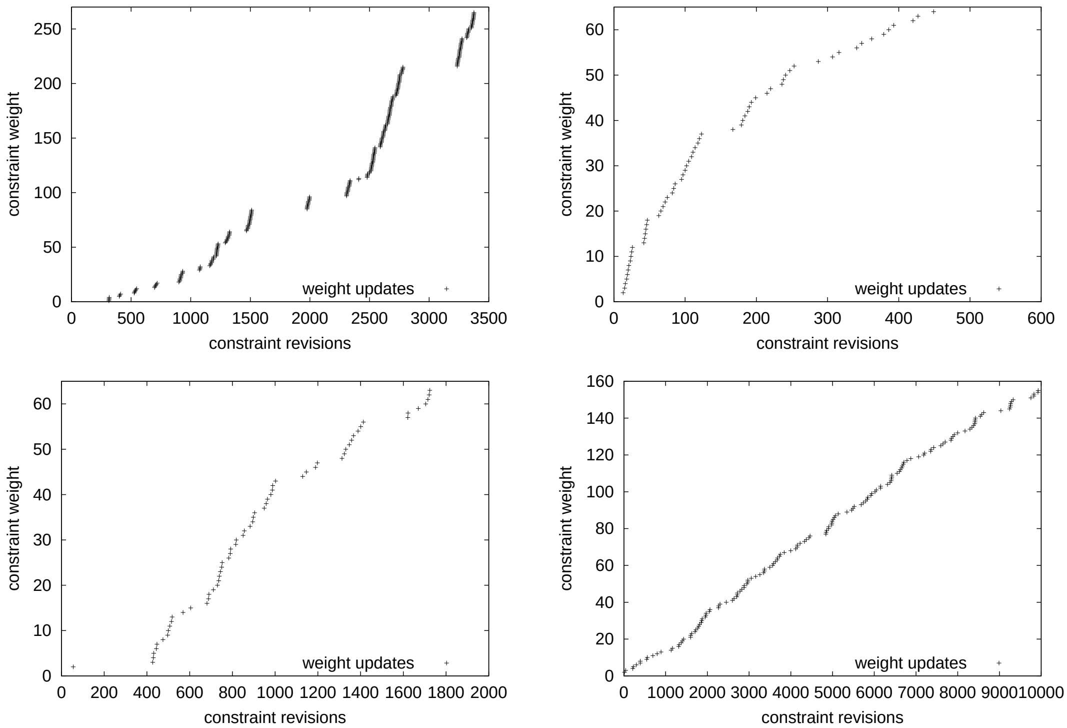

Fig. 1. DWO-revisions (or equivalently weight updates) for sample constraints from the RLFAP instance scen11 (top left), the driver instance driver-08c (top right), the quasigroup completion instance qcp15-120-0 (bottom left), and the forced random instance frb35-17-0 (bottom right).

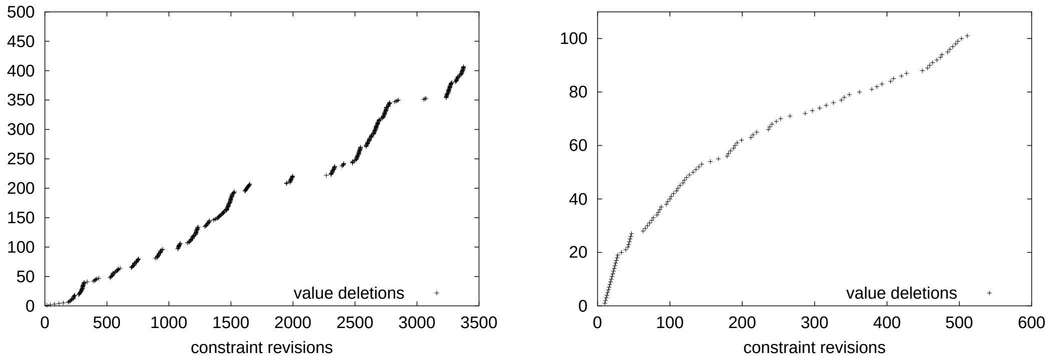

Fig. 2. Value deletions for two sample constraints from a RLFA (left) and a driver (right) problem.

redundant revisions. For example in the RLFAP of instance Figure 1 the sample constraint, which was the most active one in terms of DWO-revisions, was revised 3386 times during search, but only 407 of these revisions were fruitful, while only 265 were DWO-revisions. Similar results with respect to the percentage of fruitful revisions were obtained for constraints across a variety of structured problems.

To further investigate these observations we run the *Expectation Maximization* (EM) clustering algorithm [12] on the data of Figure 1 (top left). This revealed 20 clusters of DWO-revisions with average size of 13,25. The mean and median standard deviation (SD) for the DWO-revisions (x-axis) across the clusters was 21,67 and 7,41 respectively. Accordingly, EM revealed 16 clusters of DWO-revisions for the data of Figure 1 (bottom left) of average size 25,43. The mean and median SD for the DWO-revisions was 35,91 and 23,80 respectively. The SD in a cluster is an important piece of information as it represents the average distance of

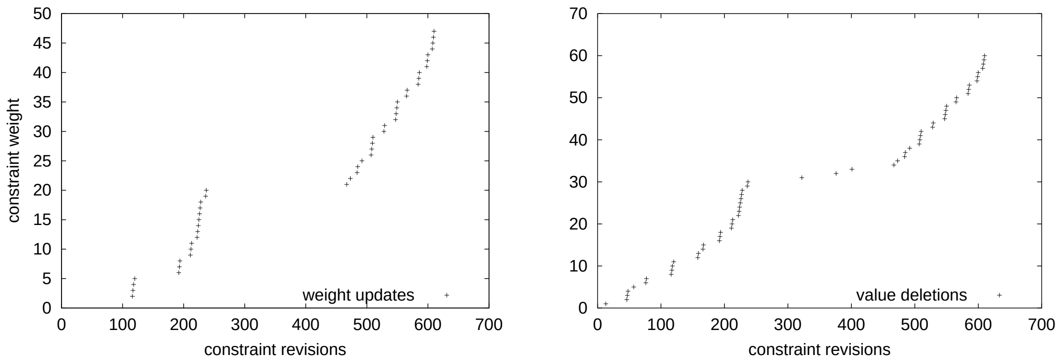

Fig. 3. DWO-revisions (left) and value deletions (right) for one sample constraint from scen11 when maxRPC is applied.

Table 1
Clustering results from benchmark instances.

instance	#cons	avg #cls	avg size	mean SD	median SD
scen11	27/4103	6,66	10,82	41,09	16,12
driver-08c	87/9321	2,44	12,62	38,50	25,11
qcp15-120-0	554/3150	12,87	15,26	226,12	129,28
frb35-17-0	233/262	7,20	19,38	1856,70	1649,05

any member of the cluster from the cluster's centroid. That is, it is a measure of the cluster's density. The median SD over the 20 clusters is quite low which indicates that DWO-revisions are closely grouped together. The mean is higher because it is affected by the presence of outliers. That is, some of the clusters formed by EM may include outliers which increase the cluster's SD.

Table 1 shows clustering results from the four benchmark instances of Figure 1. For each instance we report the ratio of weight-active constraints over the total number of constraints, the average number of clusters, the average cluster size, and the mean and median SD for the clusters of DWO-revisions. Averages are taken over 20 sample weight-active constraints from each problem on which EM was applied. The mean and median SD are much lower in structured problems compared to the random one verifying the observation that in the presence of structure DWO-revisions largely occur in clusters while in its absence they tend to be uniformly distributed. The question we try to answer in the following is whether we can take advantage of this to discover dead-ends sooner through strong propagation while keeping cpu times manageable.

4. Heuristically Adapting Propagation

We now present four simple heuristics that can be used to dynamically adapt the level of consistency enforced on individual constraints during search. These heuristics exploit information regarding domain reductions and wipeouts gathered during search. For the purposes of this paper we limit ourselves to the case where dynamic adaptation involves switching between a weak, and cheap, local consistency and a stronger but more expensive one. In general it may be desirable to utilize a suit of local consistencies with varying power and properties. However, adapting the heuristics presented here to handle more than two propagation methods is not straightforward, and therefore is left for future work.

The intuition behind the proposed heuristics is twofold. First to target the application of the strong consistency on areas in the search space where a constraint is highly active so that domain pruning is maximized and dead-ends are encountered faster. And second, to avoid using an expensive propagation method when pruning is unlikely. The first three heuristics try to take advantage of the clusterness that fruitful revisions display in structured problems, while the fourth heuristic simply reacts to any deletions caused by a constraint. To take advantage of the clusteness of propagation events the heuristics monitor these events while the constraints are revised throughout search.

The heuristics can be distinguished according to the propagation events they monitor (deletions or DWOs) and also according to the extent of user involvement in their tuning (fully and semi automated). Heuristics based on DWOs (value deletions) may change or maintain the level of local consistency employed on a given constraint by monitoring the DWOs (value deletions) caused by this constraint. There are also hybrid heuristics that may react to both types of propagation events. Fully automated heuristics do not require any tuning while semi-automated ones are parameterized by a user-defined bound. This bound specifies the desired number of revisions during which a strong consistency is enforced after a propagation event has been detected. The greater the bound the longer is the strong consistency applied.

Importantly, any heuristic, be it for branching or for adapting the local consistency enforced, must be *lightweight*, i.e. cheap to compute. Otherwise, it is possible that any benefits offered by the heuristic will be outweighed by the cost of its computation. As it will become clear, the heuristics proposed here are indeed lightweight as they affect the complexity of the propagation procedure only by a constant factor.

In our experiments we have used AC and maxRPC as the weak and strong local consistency respectively. As proved in [10], maxRPC is strictly stronger than AC. That is, it will always delete at least the same values as AC. Also, maxRPC displays a good cpu time to value deletions ratio compared to other strong local consistencies [11]. Of course other local consistencies can be used instead, and this is indeed an interesting direction for future work. Since our approach is generic, when describing the heuristics we will avoid naming specific consistencies and instead we will refer to switching between a weak (W) and a strong (S) local consistency.

For each $c \in C$, the heuristics make use of the following data structures:

1. rev[c] is an integer counter holding the number of times c has been revised, incremented by one each time c is revised.
2. dwo[c] is an integer counter denoting the revision in which the most recent DWO caused by c occurred.
3. del[c] is an integer counter denoting the most recent revision of c which resulted in at least one value deletion.
4. del_S[c] is an integer counter denoting the most recent revision of c in which at least one value that was W but not S was identified and deleted. This means that the deletion of a W-inconsistent value does not trigger a change in del_S[c]. The counter is incremented only if a value that is W but not S is deleted.
5. del_W[c] is a Boolean flag denoting whether the current revision of c resulted in at least one value deletion (del_W[c]=T) or not (del_W[c]=F).

We now describe the four heuristics, which we simply call H_1-H_4, specifying what type of propagation events they monitor and the extent of user involvement in their tuning.

$H_1(l)$: semi automated - DWO monitoring Heuristic H_1 monitors and counts the revisions and DWOs of the constraints in the problem. A constraint c is made S if the number of calls to `Revise`(c) since the last time it caused a DWO is less or equal to a (user defined) threshold l. That is, if rev[c]-dwo[c] $\leq l$. Otherwise, it is made W.

H_2: fully or semi automated - deletion monitoring Heuristic H_2 monitors revisions and value deletions. A constraint c is made S if the last call to `Revise`(c) caused at least one value deletion. That is, c is made S as long as del[c]=rev[c]. Otherwise, it is made W. H_2 can be semi automated in a similar way to H_1 by allowing for a (user defined) number l of redundant revisions after the last fruitful revision. If l is set to 0 we get the fully automated version of H_2.

H_3: fully or semi automated - hybrid Heuristic H_3 is a refinement of H_2. It monitors revisions, value deletions, and DWOs. A constraint c is made S as long as del_S[c]=rev[c]. Otherwise, it is made W. Once the constraint causes a DWO, del_S[c] is set to rev[c] and the monitoring of S's effects starts again. If this is not done then once rev[c] becomes greater than del_S[c] the constraint will thereafter be propagated using W. H_3 can be semi automated in a similar way to H_1 and H_2 by allowing for a (user defined) number l of revisions that only delete W-inconsistent values or no values at all after the last revision that deleted values that were W but not S.

H_4: fully or semi automated - deletion monitoring Heuristic H_4 monitors value deletions. For any constraint c, H_4 applies W until del_W[c] becomes T. In this case c is made S. In other words, if at least one value is deleted from the domain of a variable $x \in var(c)$ by W then S is applied on the remaining available values in $D(x)$. H_4 can be semi automated by insisting that S is applied only if a (user defined) proportion p of x's available values have been deleted by W during the current revision of c. With high values of p S will be applied only when it is likely that it will cause a DWO.

Importantly, the heuristics defined above can be combined either disjunctively or conjunctively in various ways to give rise to new heuristics. For example, heuristic $\text{H}_{124}^{\vee}$ applies S on a constraint whenever the condition specified by either H_1, H_2, or H_4 holds. Heuristic $H^{\wedge}_{24}$ applies S when both the conditions of H_2 and H_4 hold. We can choose a disjunctive or conjunctive combination depending on whether we want S applied more or less frequently respectively. Experimental results in the following section demonstrate that certain disjunctive combinations are more robust than the individual heuristics displaying good performance over a range of problems.

4.1. *Implementing the Heuristics*

The proposed heuristics can be easily crafted into any solver that performs constraint-oriented propagation. But this does not preclude their use within solvers where propagation is implemented differently, i.e. variable-oriented or propagator-oriented. As mentioned in Section 2, solvers implementing constraint-oriented propagation utilize a list that holds the constraint to be revised.

function `Propagate`(X,C,L,h)
while $L \neq \emptyset$
  remove constraint c, with $var(c) = \{x_i, x_j\}$, from L;
  prop ← `Decide`(c,h);
  if prop = S **then** `Revise`(c,x_i,S);
  else `Revise`(c,x_i,W);
  if $D(x_i)$ has been reduced **then**
    if $D(x_i) = \emptyset$ **then** return FAILURE;
    else add to L any constraint c', with $var(c') = \{x_k, x_i\}$;
return SUCCESS;

Fig. 4. The main constraint propagation function for adaptive propagation.

Figures 4 and 5 describe the implementation of the heuristics in a constraint-oriented solver using function `Propagate` and procedure `Revise`. These are based on corresponding functions for coarse-grained AC algorithms like AC-3 [19] and AC2001/3.1 [3]. Function `Propagate` takes as input the variables and the constraints of the problem, an initialized list L of constraints to be propagated, and the heuristic h to be used. Once a constraint c is removed from L, function `Decide` is called to determine how it will be propagated. This function is parameterized by the adaptive propagation heuristic h and uses the data structures required for the computation of the heuristics, which for simplicity we assume to be globally defined. `Decide` simply applies the heuristic and decides on the local consistency (W or S) to be used for its revision. For example, if h is $H_1(l)$ then `Decide` simply checks whether

function `Revise`(c,x_i,S)
  rev[c]++;
  for each $a \in D(x_i)$
    if a is not W-supported on c **then**
      delete a from $D(x_i)$;
2:      del[c] ← rev[c];
    else if a is not S-supported on c **then**
      delete a from $D(x_i)$;
2:      del[c] ← rev[c];
3:      del_S[c] ← rev[c];
  if $D(x_i) = \emptyset$ **then**
    dwo[c] ← rev[c];
3:    del_S[c] ← rev[c];

function `Revise`(c,x_i,W)
  rev[c]++;
  del_W[c] ← F;
  for each $a \in D(x_i)$
    if a is not W-supported on c **then**
      delete a from $D(x_i)$;
      del_W[c] ← T;
2:      del[c] ← rev[c];
  if del_W=T **then**
    for each $a \in D(x_i)$
      if a is not S-supported on c **then**
        delete a from $D(x_i)$;
3:        del_S[c] ← rev[c];
  if $D(x_i) = \emptyset$ **then**
    dwo[c] ← rev[c];
3:    del_S[c] ← rev[c];

Fig. 5. Constraint revision functions for adaptive propagation. The versions of `Revise` given can apply $H^{\vee}_{124}$ or $H^{\vee}_{134}$. Removing lines labelled with 3 (2) gives $H^{\vee}_{124}$ ($H^{\vee}_{134}$).

rev[c]-dwo[c] $\leq l$ holds or not. Thereafter, depending on the selected consistency, the appropriate version of procedure `Revise` (Figure 5) is called to perform the propagation. If the revision causes a DWO, `Propagate` returns to denote failure.

The two versions of `Revise` shown, one for W and one for S, implement either the combined heuristic $H^{\vee}_{124}$ or $H^{\vee}_{134}$. We now briefly describe the implementation of these heuristics as sketched in Figure 5. The rest of the heuristics, either individual or combined, can be implemented in a similar way. Initially, i.e. before the first revision of any constraint c, rev[c], dwo[c], del[c], and del_W[c] are set to 0, while del_S[c] is set to F.

If the computation of the heuristic function in `Decide` results in the application of S then `Revise`(c,x_i,S) is called. First rev[c], the counter of c's revisions, is incremented. Then each value a of x_i is checked for support on c. This is done by first checking if a is W-supported on c. If it is not then it is removed from $D(x_i)$ and if $\mathrm{H}^{\vee}_{124}$ is used counter del[c] is set to rev[c] to denote that the most recent revision of c resulted in a value deletion. If a is W-supported then it is checked for S-support. If it does not have an S-support it is deleted and counters del[c] (for $\mathrm{H}^{\vee}_{124}$) and del_S[c] (for $\mathrm{H}^{\vee}_{134}$) are set to rev[c] to denote that the most recent revision of c resulted in the deletion of a value that was W but not S. If the domain of x_i is wiped out after checking all its values then counter dwo[c] is set to rev[c] to denote that the most recent revision of c resulted in a DWO. Also, in the case of $\mathrm{H}^{\vee}_{134}$ counter del_S[c] is set to rev[c] to restart the monitoring of S's effects. This is necessary in case all the remaining values of $D(x_i)$ were removed by W.

If the computation of the heuristic function in `Decide` results in the application of W then `Revise`(c,x_i,W) is called. First, rev[c] is incremented and flag del_W[c] is initialized to F. Then each value a of x_i is checked for W-support on c. If a value is not W-supported it is deleted, flag del_W[c] is set to T to denote that the current revision of c caused a value deletion (this will be used later in this revision), and in the case of $\mathrm{H}^{\vee}_{124}$ counter del[c] is set to rev[c] to denote that the most recent revision of c resulted in a value deletion (this will be used in subsequent revisions). After checking all values in $D(x_i)$, if del_W[c] is T then S is applied on the remaining values. Any value that is not S-supported is deleted and if $\mathrm{H}^{\vee}_{134}$ is used counter del_S[c] is set to rev[c] to denote that the most recent revision of c resulted in the deletion of a value that was W but not S. Finally, if the domain of x_i is wiped out then counters dwo[c] and del_S[c] (for $\mathrm{H}^{\vee}_{134}$) are set to rev[c].

5. Experiments

In this section we make a detailed experimental evaluation of the proposed heuristics on various binary CSPs. We first compare the heuristics and some of their disjunctive and conjunctive combinations against MAC and an algorithm that propagates all constraints using maxRPC throughout search. These two algorithms are simply denoted by AC and maxRPC hereafter. As results demonstrate, disjunctively combined heuristics are more efficient and robust compared to the single versions, and outperform both AC and maxRPC. We then study the effect of random restarts on the performance of the heuristics. Results show that the use of random restarts can result in significant speed-ups in favor of one of the best adaptive heuristics compared to AC. We also study the impact that the user defined bounds l have on the efficiency of semi-automated heuristics.

Our solver uses d-way branching, lexicographic value ordering, and can employ restarts. The code was written in C. In the experiments presented the heuristic used for variable ordering was dom/wdeg. The constraint revision list was implemented as a FIFO queue, which is known to be more efficient than a LIFO implementation[1]. We experimented with the following classes of benchmarks taken from C. Lecoutre's web page (*http://www.cril.univ-artois.fr/ lecoutre/*), where details about them can be found: radio links frequency assignment (RLFA), black hole, driver, hanoi, quasigroup completion, quasigroup with holes, graph coloring, composed random, forced random, geometric quasi-random. Some classes and many specific instances we tried are very easy or very hard (e.g black hole) for all methods.

We need to point out that for many of the tested classes, with graph coloring being a notable example, there exist specialized methods that can solve the specific problems much faster than the generic methods we use. Our aim is only to demonstrate the efficiency of the proposed heuristics in dynamically switching between different local consistencies and not to outperform state-of-the-art methods for specific problems.

5.1. Evaluation of the Heuristics

In this section we compare adaptive algorithms that use the heuristics of Section 4, where each algorithm is denoted by the corresponding heuristic, to algorithms that apply AC and maxRPC. All algorithms were run to completion in a single run (i.e. there were no restarts). We include results from all four heuristics as well as from three disjunctive and one conjunctive combination of the heuristics. For H_1, and any combined heuristic that includes H_1, the value of l was set to 100 while for H_2 l was set to 10. These values were chosen empirically and display a good performance across a number of instances. However, the value of these parameters is an important factor that can affect the performance of the heuristics. This topic is further discussed in Section 5.3. Results given for heuristic H_3 are also from a semi automated version where the bound l was set to 100.

[1]The experimental results given in [26] were obtained using a LIFO implementation of the revision list.

Table 2

Nodes (n) and cpu times (t) in seconds from RFLAP instances. The s and g prefixes stand for scen and graph respectively. The best cpu time for each instance is highlighted with bold.

instance		AC	maxRPC	H_1	H_2	H_3	H_4	$H^{\vee}_{12}$	$H^{\vee}_{124}$	$H^{\vee}_{134}$	$H^{\wedge}_{12}$
s11	n	2,790	1,308	1,399	1,384	1,372	2,213	1,372	1,324	1,335	1,352
	t	5.3	12.5	4.7	5.9	4.5	5.8	**4.0**	4.5	5.8	4.4
s11-f12	n	7,349	1,703	2,531	1,744	2,462	6,030	1,687	1,812	1,744	2,754
	t	18.4	30.1	9.9	10.0	9.7	12.3	9.4	**8.2**	12.6	9.4
s11-f10	n	9,601	2,028	2,932	2,275	2,637	8,154	2,314	2,242	2,371	2,752
	t	24.8	42.6	11.6	13.0	10.4	17.3	10.8	**9.6**	17.2	10.8
s11-f9	n	101,525	33,577	37,722	35,401	34,364	78,487	34,140	36,643	34,310	37,722
	t	360.2	973.5	151.7	167.4	153.2	224.2	146.8	153.3	295.5	**146.0**
s02-f25	n	12,688	5,548	3,878	4,968	2,759	6,673	2,759	3,542	3,143	2,958
	t	13.6	56.2	5.0	17.4	4.1	8.7	**4.0**	6.0	10.3	4.1
s03-f10	n	1,507	700	912	738	922	1,025	922	831	787	921
	t	**1.8**	4.8	2.3	2.8	2.5	2.3	2.5	2.3	2.7	2.4
s03-f11	n	9,486	2,370	3,504	2,294	3,337	5,917	3,311	2,504	2,461	3,570
	t	16.8	32.4	9.1	11.2	9.4	9.3	8.2	**8.0**	14.3	8.8
g08-f10	n	19,590	8,808	9,301	5,651	10,747	10,795	4,242	4,718	6,733	9,301
	t	38.3	36.0	19.6	14.4	21.8	16.5	12.2	**9.1**	20.6	19.4
g08-f11	n	4,439	638	2,059	512	1,976	678	525	565	527	2,170
	t	10.3	4.7	5.3	2.9	5.3	2.4	3.0	**2.3**	2.4	5.6
g14-f27	n	13,833	926	11,140	3,095	10,496	5,697	2,680	3,319	3,457	11,126
	t	12.0	4.6	11.0	3.8	10.8	4.8	**3.2**	3.5	4.1	11.4
g14-f28	n	8,405	1,668	4,511	2,313	4,803	3,014	2,179	4,581	2,051	5,407
	t	13.7	7.3	8.0	4.6	7.7	3.9	**3.6**	6.1	4.1	8.3

Table 2 displays results from some selected real-world RLFAP instances taken from the *scen* and *graph* classes of RLFAPs. Originally these are optimization problems but for the purposes of CSP benchmarking some have been turned into satisfaction problems [7]. First of all we can note that in most of these problems maxRPC can be too expensive to maintain compared to AC. However, in some cases it reduces the size of the search tree significantly and is faster than AC. The adaptive heuristics cut down the size of the explored search space and reduce the run times in most cases. This is more visible in problems where maxRPC visits considerably less nodes than AC (e.g. graph08-f11). Importantly, in easy problems or in problems where maxRPC does not have a considerable effect compared to AC the heuristics do not slow the search process in a notable way.

Comparing the various heuristics it seems that all four individual heuristics are competitive, with H_1 and H_3 being better on the scen instances and H_3 and H_4 on the graph instances. Disjunctive heuristics $H^{\vee}_{12}$ and $H^{\vee}_{124}$ are the most robust as they display good performance over all instances, while $H^{\vee}_{134}$ and the conjunctive heuristic $H^{\wedge}_{12}$ are less robust. For example, $H^{\vee}_{134}$ is twice as slow as $H^{\vee}_{12}$ and $H^{\vee}_{124}$ on the scen11-f9 instance, while $H^{\wedge}_{12}$ is very competitive on the scen instances but less so on the graph ones.

Table 3 displays results from instances belonging to a collection of the following classes of benchmarks: graph coloring (1st-5th), driver (6th,7th), quasigroup completion (8th-11th), quasigroups with holes (12th,13th). In some of these problems, especially quasigroup ones, maxRPC is much more efficient than AC. The heuristics, except H_1 and H_4, can further improve on the performance of maxRPC making the adaptive algorithms considerably more efficient than MAC.

The results given in Tables 2 and 3 show that individual heuristics can display considerable variance in their performance from instance to instance. On the contrary, combined heuristics are quite robust. A comparison between the heuristics shows that H_2 and the combined ones that include H_2 display good performance on a wide variety of problems. It has to be

Table 3
Nodes (n) and cpu times (t) in seconds from structured instances.

instance		AC	maxRPC	H_1	H_2	H_3	H_4	$\mathrm{H}_{12}^{\vee}$	$\mathrm{H}_{124}^{\vee}$	$\mathrm{H}_{134}^{\vee}$	$\mathrm{H}_{12}^{\wedge}$
anna-8	n	69,321	29,260	29,572	29,262	29,578	69,881	29,042	29,248	29,518	29,572
	t	22.9	48.4	8.7	19.6	8.7	24.5	**8.5**	16.0	15.5	8.5
homer-8	n	69,280	28,994	29,368	28,975	29,368	69,321	28,986	29,039	29,237	29,368
	t	126.6	225.7	**36.4**	90.9	37.0	74.3	43.7	62.5	54.6	36.6
games120-8	n	3,208,978	1,375,255	1,371,027	1,371,882	1,371,496	3,207,187	1,366,902	1,371,027	1,379,423	1,371,027
	t	374.1	325.2	**153.1**	187.9	155.4	280.7	169.4	172.4	168.4	152.0
2-fullins-5-4	n	1,052	634	706	609	703	847	703	694	585	706
	t	26.8	18.5	12.9	15.7	**10.1**	13.3	10.3	10.4	14.8	13.4
4-fullins-4-6	n	31,507	15,436	26,777	15,507	23,385	28,720	24,147	22,074	16,944	26,777
	t	236.5	228.6	186.6	154.4	190.2	171.4	153.6	144.5	**138.7**	183.5
driverlogw-08	n	3,872	848	3,314	992	3,693	2,922	942	2,384	1,609	3,314
	t	9.8	15.4	8.0	5.5	8.9	6.4	6.6	6.0	**4.6**	7.8
driverlogw-09	n	14,129	9,814	15,707	9,673	14,683	12,475	10,589	12,333	11,587	15,707
	t	155.9	194.8	167.4	**114.1**	156.8	122.5	130.3	120.5	120.9	167.6
qcp-15-120-0	n	102,136	19,496	79,108	29,064	45,381	99,455	32,403	30,206	29,434	79,108
	t	86.8	**26.4**	63.9	28.4	40.0	68.5	30.6	26.5	27.5	62.6
qcp-15-120-5	n	536,056	62,682	404,003	63,984	135,914	405,187	52,163	73,628	53,311	404,003
	t	559.6	103.0	388.4	76.2	138.7	327.6	**64.8**	82.1	66.1	378.4
qcp-15-120-9	n	851,950	129,526	565,663	162,243	372,360	792,334	128,974	140,900	118,719	565,663
	t	841.0	193.8	486.2	162.1	346.0	564.4	130.0	140.2	**125.0**	481.6
qcp-15-120-10	n	1,058,477	54,622	142,637	62,124	97,483	236,131	64,686	59,252	333,339	142,637
	t	950.0	76.3	117.9	59.8	79.2	166.6	60.9	**55.0**	369.4	116.1
qwh-20-166-0	n	94,013	8,975	18,179	15,891	34,409	73,655	21,153	28,508	28,757	18,179
	t	238.1	**28.4**	49.0	41.6	87.7	149.4	58.5	72.3	78.8	48.4
qwh-20-166-1	n	48,892	14,556	46,970	18,285	32,905	80,360	17,414	18,100	26,178	46,970
	t	135.4	48.8	120.1	51.9	88.8	169.7	50.3	**35.8**	77.0	118.5

noted that $\mathrm{H}_{24}^{\vee}$ and $\mathrm{H}_{124}^{\vee}$ were faster than AC in all instances we tried, except for some easy instances where they were slightly slower. H_1 and H_3 are effective on RLAFPs but worse than H_2 on quasigroup problems. The fully automated version of H_4 displays the worst performance among the individual heuristics. But we have not yet tried semi automated versions of H_4. Overall the heuristics offer a good balance between AC and maxRPC. In problems where maxRPC offers significant savings in nodes compared to AC, they retain this advantage and translate it into considerable savings in run times. In problems where maxRPC offers moderate savings in nodes, the heuristics significantly reduce the run times of maxRPC and are competitive, and often faster, than AC.

Table 4 gives result from forced random and geometric quasi-random problems. As is clear, in such problems the heuristics are generally outperformed by AC in run times, especially on the forced random problems that lack structure. On geometric problems, which display some structure, the heuristics fare better. The best heuristics are by far H_4 and $\mathrm{H}_{12}^{\wedge}$ which are in fact competitive to AC. The disjunctive heuristics which apply maxRPC more frequently are the worse. A probable explanation is that the clusterness of propagation events is absent in these types of problems which means that many of the revisions following a value deletion or a DWO are quite often redundant. Therefore targeting the application of maxRPC on such revisions may only increase cpu times without offering much pruning. On the other hand, H_4 does not target clusters of activity to apply maxRPC but reacts to value deletions wherever they occur. Hence, it is not significantly handicapped by the absence of clusters.

Table 5 summarizes the above results concerning the eight tested heuristics and their comparison with AC and maxRPC. For each heuristic an entry of the form x/y in the first column gives the number x of instances where the heuristic is faster than AC, and the number y of instances where it is slower than AC. The second

Table 4

Nodes (n) and cpu times (t) in seconds from random instances.

instance		AC	maxRPC	H_1	H_2	H_3	H_4	$H_{12}^{\vee}$	$H_{124}^{\vee}$	$H_{134}^{\vee}$	$H_{12}^{\wedge}$
frb35-17	n	26,729	9,372	19,032	9,460	13,005	20,585	9,520	9,732	9,321	19,032
	t	**10.4**	28.9	13.1	21.2	21.1	11.1	20.2	20.8	27.6	12.9
frb40-19	n	45,296	16,608	33,381	19,001	29,427	35,457	19,037	17,503	17,070	33,381
	t	**20.1**	60.6	25.2	46.7	60.2	21.9	46.9	40.1	60.0	24.8
frb45-21	n	1,207,920	538,317	925,945	561,800	690,290	1,032,873	551,669	555,943	537,722	925,945
	t	**654.0**	1957.6	779.5	1450.8	1707.8	741.7	1441.9	1476.0	1970.2	765.4
geo50-20-75-1	n	195,270	67,914	138,774	82,092	113,247	161,792	102,452	79,129	69,094	138,774
	t	135.4	429.2	135.4	283.9	261.2	135.9	227.8	262.9	390.2	**132.6**
geo50-20-75-2	n	199,608	32,750	157,509	39,001	78,973	174,720	52,481	37,052	32,690	157,509
	t	**112.8**	209.4	133.7	140.7	177.9	133.4	115.4	120.9	196.0	131.5
geo50-20-75-8	n	119,180	32,443	83,498	46,300	66,816	98,383	48,285	45,646	40,755	83,498
	t	73.4	164.2	71.7	133.6	141.9	73.4	86.2	122.4	185.2	**70.8**

Table 5

Summary of results for the eight tested heuristics.

	AC			maxRPC		
	$\times 1$	$\times 2$	$\times 5$	$\times 1$	$\times 2$	$\times 5$
H_1	23(1)/6(4)	9/0	1/0	21(6)/9	15(5)/5	4/0
H_2	21/9(6)	11/4(4)	4/0	27(6)/3	10/0	1/0
H_3	22/8(6)	12/3(3)	1/0	21(6)/9	9/2	4/0
H_4	20/9(5)	6/0	1/0	23(6)/7	15(5)/6	1/1
$H_{12}^{\vee}$	23/7(6)	18/2(2)	3/0	27(6)/3	11/1	4/0
$H_{124}^{\vee}$	23/7(6)	18/3(3)	3/0	28(6)/2	12/1	2/0
$H_{134}^{\vee}$	21/9(6)	9/5(5)	2/0	24(4)/6(2)	9/2	1/1
$H_{12}^{\wedge}$	24(2)/6(4)	9/0	1/0	21(6)/9	15(5)/6	4/0

and third columns give similar data for the number of instances where the heuristic is two times faster (resp. slower), and five times faster (resp. slower) than AC. The following three columns give similar information concerning the comparison between the heuristics and maxRPC. A number z in brackets, e.g. $x/y(z)$, is the number of random instances, i.e. forced random or geometric, out of the y instances.

As is clear, all heuristics are on average better than both AC and maxRPC but $H_{12}^{\vee}$ and $H_{124}^{\vee}$ are the ones achieving the most robust performance. For example, looking at the "twice as fast" columns it is clear that they dominate AC and maxRPC while other heuristics are not as dominant over both AC and maxRPC.

A final interesting observation is that sometimes the heuristics result in fewer node visits than maxRPC or in more than AC. This is explained by the interaction between constraint propagation and the variable ordering heuristic. Different propagation methods can lead to different weight increases for the constraints, which in turn can guide dom/wdeg to different variable selections, and hence different parts of the search space.

5.2. Random Restarts

To investigate the impact of random restarts on the performance of the heuristics we implemented the following restart policy within our solver. The initial number of allowed backtracks for the first run has been set to 10 and at each new run the number of allowed backtracks increases by a factor of 1.5. Such policies, where the backtrack bound increases geometrically after each restart, have been shown to be quite efficient [30]. In general the use of restarts enables the solver to tackle much harder problems that are beyond its reach without restarts, but in some cases restarts slow down the solver considerably.

Table 6 gives indicative experimental results comparing the disjunctive heuristic $H_{12}^{\vee}$, which displayed good overall performance, to AC and maxRPC. The table displays results from RLFAPs, graph coloring, quasigroups, and random problems. We include results of some instances used in Section 5.1 as well as from additional instances, some of which are not solvable in reasonable time without restarts.

Results confirm that the adaptive heuristic displays a better overall performance compared to AC and maxRPC. $H_{12}^{\vee}$ is fastest on 12 instances (plus one tied with AC), AC is fastest on 4 instances (plus one tied with $H_{12}^{\vee}$), and maxRPC is fastest on 5 instances. It is interesting that maxRPC is the best method on all instances of the "quasigroup with holes" problem, except qwh20-166-8 where $H_{12}^{\vee}$ is better. AC performs poorly on these instances but on the other hand it outperforms

Table 6

Nodes (n) and cpu times (t) in seconds from RLFA, graph coloring problems (left columns) and quasigroup, random problems (right columns). The best cpu time for each instance is highlighted with bold. A time out limit of 2 hours was set.

instance		AC	maxRPC	$H_{12}^{\vee}$	**instance**	AC	maxRPC	$H_{12}^{\vee}$
scen11-f9	n	1,632	670	866	qcp15-120-13	1,007,089	150,480	151,743
	t	**5.0**	15.5	**5.0**		1,001.9	345,1	**184.1**
scen11-f8	n	2,769	858	824	qcp20-187-0	1,406,618	122,718	120,008
	t	9.6	22.6	**5.4**		3,664.8	566.5	**332.6**
scen11-f7	n	32,104	3,658	3,758	qcp20-187-1	189,942	357,657	272,574
	t	86.2	78.1	**13.9**		**429.8**	1,667.7	732.4
scen11-f6	n	74,879	5,220	6,292	qwh20-166-3	227,422	15,480	63,229
	t	194.2	148.4	**22.3**		571.1	**38.0**	176.5
scen11-f5	n	321,435	44,043	70,677	qwh20-166-4	34,507	6,535	11,308
	t	821.4	920.0	**233.1**		75.8	**11.4**	19.7
scen11-f4	n	1,110,401	251,304	219,795	qwh20-166-5	776,067	13,650	56,072
	t	2,714.7	4,739.7	**708.1**		1,882.7	**30.9**	137.2
games120-8	n	228,529	93,940	93,908	qwh20-166-6	-	45,947	154,332
	t	28.1	26.9	**11.9**		>2h.	**182.5**	442.8
anna-8	n	228,497	93,894	93,858	qwh20-166-7	88,429	9,026	10,945
	t	112.6	294.5	**31.2**		205.4	**19.5**	20.8
homer-8	n	228,495	93,906	93,872	qwh20-166-8	70,945	27,281	12,565
	t	509.0	996,4	**110,4**		158.7	84.0	**21.9**
myciel5-5	n	22,640,358	22,640,358	22,640,358	frb40-19	170,345	46,238	45,266
	t	**1,394.6**	2,534.4	2,276.4		**74.1**	162.8	112.3
4-fullins-4-6	n	9,354	4,070	5,985	geo50-20-75-1	548,208	164,036	230,374
	t	76.7	134.3	**60.1**		**389.3**	1,116.4	852.6

maxRPC on RLFA, graph coloring, and random problems. $H_{12}^{\vee}$ manages to combine the strengths of the two methods as it can be significantly faster than both AC and maxRPC on several instances while it rarely performs considerably worse than any of them. Note that the "quasigroup with holes" instances where maxRPC can outperform $H_{12}^{\vee}$ by a large margin are all soluble. Hence, two different algorithms may discover different solutions. As we explain in the next section, another reason for the dominance of maxRPC over $H_{12}^{\vee}$ on these problems may be related to the value of the user-defined bounds for H_1 and H_2 (100 and 10 may be too low).

5.3. *Semi-automated Heuristics*

In this section we study the effect that the user defined bounds have on the performance of semi-automated heuristics. Figure 6 displays search effort in visited nodes and cpu time against the value of l when heuristics H_1 (top plots) and H_2 (bottom plots) are used to solve four sample instances belonging to different problem classes. Each instance was solved in a single run of search, i.e. there were no restarts.

As the value of l increases, the number of visited nodes decreases in all four instances for both H_1 and H_2. This is to be expected as larger values of l imply the application of maxRPC in more constraint revisions. It is notable however that the number of nodes more or less stabilizes after a certain value of l, which may vary from instance to instance. In the case of H_2 this value is low (around 10 for all four instances) suggesting that allowing for a only a few redundant revisions after a fruitful one is enough to maximize the efficiency of adaptive propagation through H_2. In the case of H_1 the number of nodes seems to stabilize at around $l = 100$ except for the quasigroup problem where it continues to fall even for $l > 1000$.

Regarding cpu times, some interesting observations can be made. As expected, for low values of l cpu times are closer to those of AC which means that they are worse for structured problems but better for ran-

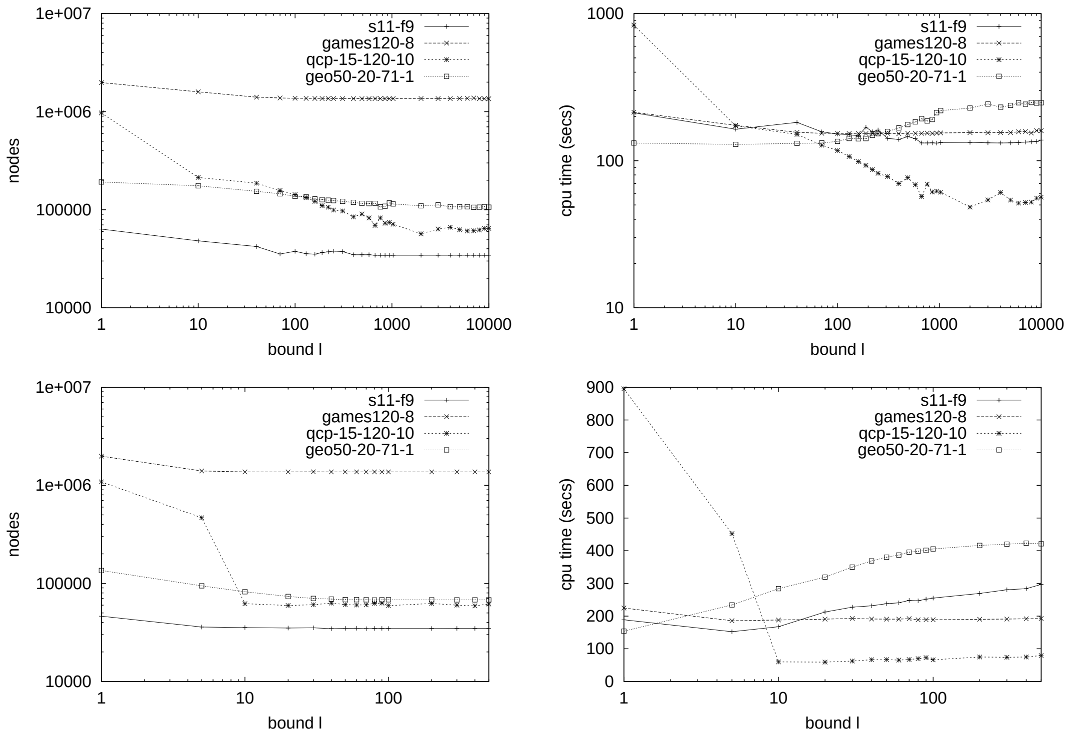

Fig. 6. Node visits (left) and cpu times (right) for H_1 (top) and H_2 (bottom).

dom ones. It is interesting that on structured problems heuristic H_2 can notably improve on the performance of AC even for values of l as low as 1. This is the case with the RLFAP and the graph coloring problem. After a relatively small value of l (again around 10) there is no notable improvement in the performance of H_2. On the contrary, as l increases beyond 10 the performance is in some cases degraded (e.g. RLFAP).

Cpu times for heuristic H_1 follow a similar pattern to node visits in the case of the three structured instances. That is, they tend to stabilize for the RLFAP and the graph coloring problem at around $l = 100$ but continue to fall as l increases for the quasigroup problem. Relating the performance of H_1 on the quasigroup problem to the data of Table 1, we can derive a possible explanation for this. The standard deviation in the clusters of propagation events in quasigroup problems is quite high compared to RLFAPs and graph coloring problems. This suggests that clusters are not as dense which means that DWO-revisions are not as close to one another. Hence, a higher value of l is required to achieve better performance. Finally, in the case of the quasi-random geometric problem the cpu time increases as l increases.

More experiments are required in the future to better understand the effect that the values of the l parameters have on the performance of semi-automated heuristics. This is important as the semi-automated versions of the proposed heuristics seem to be considerably better than the fully-automated ones. Another possibility that is worth exploring is the automatic adjustment of l during search considering that small values of l imply performance closer to the weak consistency W while higher values imply more regular use of the strong consistency S. Hence, it is probable that we would prefer a high value for l in certain parts of the search space (e.g. near the top of the search tree) while a low value of l might be preferable in other parts.

6. Related Work

Building adaptive constraint solvers is a topic that has attracted considerable interest in the literature (see for example [5,21,14,17]). Part of this interest has been directed to the dynamic adaptation of constraint propagation during search. The most common manifestation of this idea is the use of different propagators for different types of domain reductions in arithmetic constraints. When handling arithmetic constraints most solvers differentiate between events such as removing a value from the middle of a domain, or from a bound of a domain, or reducing a domain to a singleton, and apply suitable propagators accordingly. Works on adaptive propagation for general constraints include the following.

El Sakkout et al. proposed a scheme called *adaptive arc propagation* for dynamically deciding whether to process individual constraints using AC or forward checking [13]. *Anti-functional reduction* is an instantiation of this scheme that achieves the same level of consistency as AC but avoids some redundant overheads. Freuder and Wallace proposed a technique, called *selective relaxation* which can be used to restrict AC propagation based on two local criteria; the distance in the constraint graph of any variable from the currently instantiated one, and the proportion of values deleted [15]. Chmeiss and Sais presented a backtrack search algorithm, MAC (dist k), that also uses a distance parameter k as a bound to maintain a partial form of AC [9].

Schulte and Stuckey proposed techniques for dynamically selecting which propagator to apply to a given constraint using priorities and staged propagators [24,25]. Their proposed methods either select a single propagator from a given set or propagators or choose the order in which the propagator stages will be applied [25]. These methods are based on interpreting the event that triggers propagation for a constraint at any point in time, such as the reduction of a domain to a singleton or the removal of a value from a bound of a domain. On the other hand, our approach is based on monitoring the history of constraint propagation starting from preprocessing and continuing throughout search. Similar ideas to the ones of [25] are also implemented in constraint solvers such as Choco [18].

Probabilistic arc consistency is a scheme that can help avoid some consistency checks and constraint revisions that are unlikely to cause any domain pruning [20]. As in [13], the scheme is based on information gathered by examining the supports of values in constraints which can be very expensive for non-binary constraints.

Szymanek and Lecoutre studied ways to select values on which to apply “shaving” (i.e. make the values SAC) using the semantics of global constraints (e.g. alldifferent) to suggest values that are most likely to be removed by shaving [27].

Our work is more closely related to the work of [13] as the aim is to dynamically adapt the level of local consistency achieved on individual constraints during search. However, neither [13] or any of other works mentioned use information about failures captured in the form of DWOs to achieve this. Besides, to the best of our knowledge, although many levels of consistency stronger than AC have been proposed, they have not been studied in this context before (i.e evoking them dynamically).

7. Conclusions and Future Work

We have proposed a number of simple lightweight heuristics for dynamically switching between different constraint propagation methods applied on individual constraints during search. These heuristics monitor propagation events like DWOs and value deletions caused by the constraints and react by changing the propagation method when certain conditions are met. The inspiration behind the development of the heuristics was based on observing the activity of the constraints when using a conflict-driven search heuristic. As we demonstrated, DWOs and value deletions in structured problems mostly occur in clusters of consecutive of nearby revisions. This can be taken advantage of to increase or decrease the level of consistency applied when a constraint is highly active or inactive respectively. Experimental results from various domains displayed the usefulness of the proposed heuristics.

The work presented here is only a first step towards designing adaptive constraint propagation heuristics that can efficiently switch between different levels of local consistency using information gathered during search. There are several directions for future work. For example, we can investigate different local consistencies for binary and non-binary problems, try to devise more sophisticated heuristics, and integrate with existing related works (e.g. [20]). Also, it would be interesting to study the interaction of adaptive propagation with other adaptive branching heuristics apart from dom/wdeg. For example, the impact-based heuristics of [22] and the explanation-based heuristics of [8].

References

[1] C. Bessiere. Constraint Propagation. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 3. Elsevier, 2006.

[2] C. Bessière and J.C. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?). In *Proceedings of CP-1996*, pages 61–75, Cambridge MA, 1996.

[3] C. Bessière, J.C. Régin, R. Yap, and Y. Zhang. An Optimal Coarse-grained Arc Consistency Algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

[4] C. Bessiere, K. Stergiou, and T. Walsh. Domain filtering consistencies for non-binary constraints. *Artificial Intelligence*, 172(6-7):800–822, 2008.

[5] J. Borrett, E Tsang, and N. Walsh. Adaptive Constraint Satisfaction: The Quickest First Principle. In *ECAI-96*, pages 160–164, 1996.

[6] F. Boussemart, F. Heremy, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *ECAI-2004*, pages 482–486, 2004.

[7] B. Cabon, S. De Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio Link Frequency Assignment. *Constraints*, 4:79–89, 1999.

[8] H. Cambazard and N. Jussien. Identifying and Exploiting Problem Structures Using Explanation-based Constraint Programming. *Constraints*, 11:295–313, 2006.

[9] A. Chmeiss and L. Sais. Constraint Satisfaction Problems: Backtrack Search Revisited. In *ICTAI-2004*, pages 252–257, 2004.

[10] R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *CP-97*, pages 312–326, 1997.

[11] R. Debruyne and C. Bessière. Domain Filtering Consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.

[12] A. Dempster, N. Laird, and D. Rubin. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society*, 39(1):1–38, 1977.

[13] H. El Sakkout, M. Wallace, and B. Richards. An Instance of Adaptive Constraint Propagation. In *CP-96*, pages 164–178, 1996.

[14] S. Epstein, E. Freuder, R. Wallace, A Morozov, and Samuels. B. The Adaptive Constraint Engine. In *CP-2002*, pages 525–540, 2002.

[15] E. Freuder and R.J. Wallace. Selective relaxation for constraint satisfaction problems. In *ICTAI-96*, 1996.

[16] D. Grimes and R.J. Wallace. Sampling Strategies and Variable Selection in Weighted Degree Heuristics. In *CP-2007*, pages 831–838, 2007.

[17] Y. Hamadi, E. Monfroy, and F. Saubion, editors. *1st International Workshop on Autonomous Search (in conjunction with CP-07)*. 2007.

[18] F. Laburthe and Ocre. Choco : implementation du noyau d'un systeme de contraintes. In *JNPC-00*, pages 151–165, 2000.

[19] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[20] D. Mehta and M.R.C. van Dongen. Probabilistic Consistency Boosts MAC and SAC. In *IJCAI-2007*, pages 143–148, 2007.

[21] S. Minton. Automatically Configuring Constraint Satisfaction Programs: A Case Study. *Constraints*, 1(1/2):7–43, 1996.

[22] P. Refalo. Impact-based search strategies for constraint programming. In *CP-2004*, pages 556–571, 2004.

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

[24] C. Schulte and P.J. Stuckey. Speeding Up Constraint Propagation. In *CP-2004*, pages 619–633, 2004.

[25] C. Schulte and P.J. Stuckey. Efficient Constraint Propagation Engines. *ACM Trans. Program. Lang. Syst.*, 31(1):1–43, 2008.

[26] K. Stergiou. Heuristics for Dynamically Adapting Propagation. In *ECAI-2008*, pages 485–489, 2008.

[27] R. Szymanek and C. Lecoutre. Constraint-Level Advice for Shaving. In *ICLP-2008*, pages 636–650, 2008.

[28] P. van Beek. Backtracking Search Algorithms. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 4. Elsevier, 2006.

[29] W. van Hoeve and I. Katriel. Global Constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 5. Elsevier, 2006.

[30] R. Wallace and D. Grimes. Experimental studies of variable selection strategies based on constraint weights. *Journal of Algorithms*, 63(1-3):114–129, 2008.