Heuristics for Dynamically Adapting Propagation

Kostas Stergiou[1]

Abstract.

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 level of local consistency applied during search. A related problem that is receiving a lot of attention is the design of adaptive branching heuristics. The recently proposed adaptive variable ordering heuristics of Boussemart et al. use information derived from domain wipeouts to identify highly *active* constraints and focus search on hard parts of the problem resulting in important saves in search effort. In this paper we show how information about domain wipeouts and value deletions gathered during search can be exploited, not only to perform variable selection, but also to dynamically adapt the level of constraint propagation achieved on the constraints of the problem. First we demonstrate that when an adaptive heuristic is used, value deletions and domain wipeouts caused by individual constraints largely occur in clusters of consecutive or nearby constraint revisions. Based on this observation, we develop a number of simple heuristics that allow us to dynamically switch between enforcing a weak, and cheap local consistency, and a strong but more expensive one, depending on the activity of individual constraints. As a case study we experiment with binary problems using AC as the weak consistency and maxRPC as the strong one. Results from various domains demonstrate the usefulness of the proposed heuristics.

1 INTRODUCTION

Building adaptive constraint solvers is a major challenge in constraint programming. One aspect of this goal is concerned with ways to dynamically adapt the level of local consistency applied on constraints during search. Constraint solvers typically maintain (generalized) arc consistency (G)AC, or a weaker consistency property like bounds consistency, during search. Although many stronger local consistencies have been proposed, their practical usage is limited as they are mostly 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 consistencies 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 AC and weaker consistencies [8, 10, 17, 14]. Here we consider methods to selectively apply stronger local consistencies than AC during search.

[1] Department of Information & Communication Systems Engineering University of the Aegean, Greece (konsterg@aegean.gr).

Recently, Boussemart et al. proposed two adaptive *conflict-driven* variable ordering heuristics for CSPs called wdeg and dom/wdeg [2]. These heuristics use information derived from conflicts, in the form of domain wipeouts (DWOs), and stored as constraint weights to guide search. These heuristics, and especially dom/wdeg, are among the most efficient, if not *the* most efficient, general-purpose heuristics for CSPs. Grimes and Wallace proposed alternative conflict-driven heuristics that consider value deletions as the basic propagation events associated with constraint weights [11]. 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 [11].

It has been recognized, for example in [14], that in many problems only few of the constraint revisions that occur during search are fruitful (i.e. delete values) while, as an extreme case, some constraints do not cause any value deletions at all despite being revised many times. 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. Through weight recording, conflict-driven heuristics are able to identify highly active constraints and focus search on variables involved in such constraints. Given that highly active constraints usually reside in hard parts of the problem, can one take advantage of this information to adapt the level of constraint propagation accordingly?

In this paper we show how information about conflicts and value deletions can be exploited, not only to perform variable selection, but also to dynamically adapt the level of local consistency achieved on the constraints of the problem. First we demonstrate that when a conflict-driven heuristic is used on structured problems, 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. Based on this observation, we develop simple heuristics that allow us to dynamically switch between enforcing a weak, and cheap local consistency, and a strong but more expensive one. 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 DWO. Then it switches to a strong consistency and applies it on c for the next few revisions. If no further weight update occurs during these revisions, it switches back to a weaker consistency. As a case study we experiment with binary problems using AC as the weak consistency and maxRPC as the strong one. Experimental results from various domains demonstrate the usefulness of the proposed heuristics.

2 BACKGROUND

A *Constraint Satisfaction Problem* (CSP) is a tuple (X, D, C) where: X is a set of n variables, D is a set of domains, one for each variable, and C 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 $D(x_1)\text{x}\ldots\text{x}D(x_k)$. 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 constraints in C are AC. 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) [5]. 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.

The *revision* of a 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. 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. We will say that a constraint is *DWO-active* during a run of a search algorithm if it caused at least one DWO. 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.

3 CONSTRAINT ACTIVITY DURING SEARCH

In many, mainly structured, problems some constraints do not cause any DWOs or even are deletion-inactive during the run of a search algorithm. For example, when solving the scen11 RLFA problem with MAC+dom/wdeg, only 27 of the 4103 constraints in the problem were DWO-active while 2182 were deletion-active. 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 DWOs, especially if a heuristic like dom/wdeg that can identify such subproblems is used. 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 maintains maxRPC (MmaxRPC) + dom/wdeg, 29 constraints were DWO-active with 13 of these identified as DWO-active by both MAC and MmaxRPC.

Importantly, many revisions of the constraints that are DWO-active and deletion-active are redundant or achieve very little pruning. Figure 1 demonstrates how DWOs (y-axis) caused by 4 sample constraints are detected as constraint revisions (x-axis) occur throughout search. That is, each data point gives the weight of the constraint at its i-th DWO-revision. The algorithm used is MAC + dom/wdeg and the sample constraints are taken from three structured and one random problem. As we can see, DWOs in structured problems form clusters of successive or very close calls to the revision procedure, with the exception of a few outliers. The same pattern occurs with respect to value deletions (not shown due to lack of space). In contrast, DWOs in the random instance are distributed in a much more uniform way along the line of revisions. Similar results were

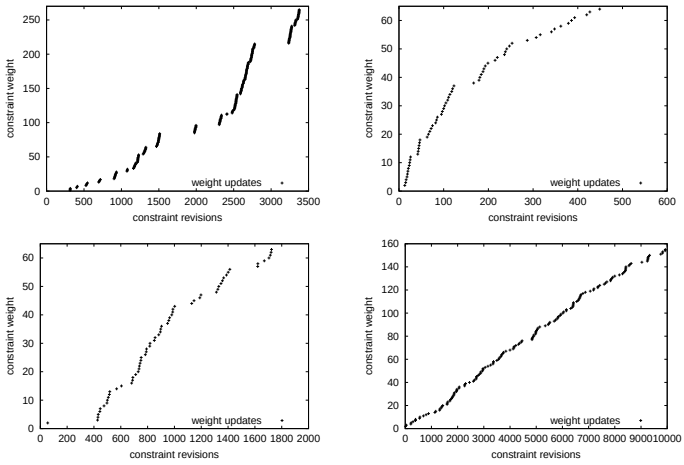

Figure 1. DWOs caused by sample constraints from the RLFAP instance scen11 (top left), the driver instance driver-08c (top right), the quasigroup completion instance qcp15-120-0 (bottom left), and the forced random instance frb35-17-0 (bottom right).

obtained when MmaxRPC was used in place of MAC. Note that in the structured problems the percentage of DWO-revisions to total revisions is low. There were also many redundant revisions. For example in the RLFAP instance the sample constraint, which was the most active one in terms of DWOs caused, was revised 3386 times during search, but only 407 of these revisions were fruitful, while only 265 were DWO-revisions.

To further investigate these observations we run the *Expectation Maximization* (EM) clustering algorithm [7] 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. The SD in a cluster is an important piece of information as it represents the average distance of 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. Clustering results from benchmark instances.

instance	#constraints	avg #clusters	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

Table 1 shows clustering results from the four benchmark instances of Figure 1. For each instance we report the ratio of DWO-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 DWO-active constraints from each problem. 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. These heuristics exploit information regarding domain reductions and wipeouts gathered during search. 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. The intuition behind the 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.

Importantly, any heuristic, be it for branching or for adapting the local consistency enforced, must be *lightweight*, i.e. cheap to compute. 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.

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 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.

In our experiments we have used AC and maxRPC as the weak and strong local consistency respectively. As proved in [5], 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 [6]. However, 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, where S is strictly stronger than W.

For each $c \in C$, the heuristics record the following information: 1) rev[c] is a counter holding the number of times c has been revised, incremented by one each time c is revised. 2) dwo[c] is an integer denoting the revision in which the most recent DWO caused by c occurred. 3) del[c] is a Boolean flag denoting whether the most recent revision of c resulted in at least one value deletion (del[c]=T) or not (del[c]=F). 4) del_S[c] is a Boolean flag denoting whether the most recent revision of c identified and deleted at least one value that is W but not S. The flag becomes T only if a value that is W but not S is deleted. Otherwise, it is set to F. 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).

$H_1(l)$: *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* H_2 monitors revisions and value deletions. A constraint c is made S as long as del[c]=T. 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* 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]=T. Otherwise, it is made W. Once the constraint causes a DWO, del_S[c] is set to T and the monitoring of S's effects starts again. If this is not done then once del_S[c] is set to F 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* 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 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. For example, heuristic $H^{\vee}_{124}$ 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 to a greater or lesser extent respectively.

Figure 2 describes the implementation of functions `Revise` for applying a weak or a strong consistency using the proposed heuristics. They are based on corresponding functions of coarse-grained algorithms like AC-3. Once a constraint is selected for revision a function we call `Decide` (which is not shown for space reasons) is called to determine how it will be propagated. This function is parameterized by the adaptive propagation heuristic h and the data structures required for the computation of the heuristics. The appropriate function w.r.t. to h is called to compute the heuristic and decide on the local consistency to be applied. Thereafter, depending on the selected consistency, the appropriate version of function `Revise` is called to perform the propagation. The two versions of `Revise` shown, one for W and one for S, implement $H^{\vee}_{124}$ or $H^{\vee}_{134}$. As values are deleted and DWOs are detected, the data structures used by the heuristics are updated. Initially, i.e. before the first revision of c, del[c], del_W[c] and del_S[c] are set to F and rev[c], dwo[c] are set to 0.

5 EXPERIMENTS

We implemented and tested the heuristics described in Section 4 as well as a number of combined heuristics. We used d-way branching, dom/wdeg for variable ordering, and lexicographic value ordering. We experimented with the following classes of benchmarks taken from C. Lecoutre's web page, where details about them can be found: radio links frequency assignment (RLFAP), langford, black hole, driver, hanoi, quasigroup completion, quasigroup with holes,

```
function Revise(c,x_i,S)
   rev[c]++;
   for each a ∈ D(x_i)
      if a is not W-supported in c then
         delete a from D(x_i);
2:       del[c] ← T;
      else if a is not S-supported in c then
         delete a from D(x_i);
2:       del[c] ← T;
3:       del_S[c] ← T;
   if D(x_i) = ∅ then
      dwo[c] ← rev[c];
3:    del_S[c] ← T;
2:if no value is deleted then del[c] ← F;
3:if no value that is W is deleted by S then del_S[c] ← F;

function Revise(c,x_i,W)
   rev[c]++;
   for each a ∈ D(x_i)
      if a is not W-supported in c then
         delete a from D(x_i);
         del_W[c] ← T;
2:       del[c] ← T;
   if del_W=T then
      for each a ∈ D(x_i)
         if a is not S-supported in c then
            delete a from D(x_i);
3:          del_S[c] ← T;
   if D(x_i) = ∅ then
      dwo[c] ← rev[c];
3:    del_S[c] ← T;
2:if no value is deleted then del[c] ← F;
3:if no value that is W deleted by S then del_S[c] ← F;
```

Figure 2. The versions of Revise given can apply $H^{\vee}_{124}$ or $H^{\vee}_{134}$. Removing lines labelled with 3 (2) gives $H^{\vee}_{124}$ ($H^{\vee}_{134}$).

graph coloring, composed random, forced random, geometric random. Some classes and many specific instances are very easy (e.g. composed) or very hard (e.g black hole) for all methods. The results presented below demostrate that the heuristics retain the efficiency of maxRPC where it is better than AC and improve it where it is worse. Also, we need to point out that for many of the tested classes 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.

Table 2 shows results from some real-world RLFAP instances. We compare adaptive algorithms that use the heuristics of Section 4, where each algorithm is denoted by the corresponding heuristic, to MAC and MmaxRPC, simply denoted by AC and maxRPC respectively. For H_1, and any combined heuristic that includes H_1, 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[2].

In these problems maxRPC is too expensive to maintain compared to 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 significant way. Table 3 shows results, including only some of the heuristics, from instances belonging to the following classes of benchmarks: graph

[2] The fully automated version of H_2 is competitive but less robust.

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	$H^{\vee}_{14}$	$H^{\vee}_{124}$
s11	n	2864	1334	1175	1842	1432	1678	1358	1360
	t	6.9	24.2	**3.7**	6.7	5.5	6.0	4.9	4.9
s11-f9	n	108184	37663	35102	47552	39312	53338	38202	37743
	t	539.6	3478.3	**170.4**	335.4	183.3	274.8	205.2	212.7
s11-f10	n	8576	2098	2197	2675	1938	3849	2462	2467
	t	30.2	93.8	11.6	18.8	**10.2**	13.9	11.4	11.3
s11-f12	n	6678	1923	1750	2804	1763	3095	1953	1921
	t	19.7	101.7	**8.6**	14.5	9.4	14.7	11.0	10.6
s02-f25	n	11998	5262	3114	10802	2938	12961	4367	4922
	t	9.3	65.1	5.6	16.0	**5.5**	15.2	9.3	10.3
s03-f11	n	8314	880	1047	4830	2762	4518	2068	1489
	t	26.4	24.7	**5.6**	20.2	11.8	17.2	12.5	9.5
g08-f10	n	11948	6342	6650	6423	9540	4863	4474	4119
	t	34.5	147.1	21.9	19.4	26.8	**13.9**	16.3	16.2
g08-f11	n	9996	629	753	960	748	713	608	619
	t	35.9	18.7	4.3	4.5	4.8	**3.6**	**3.6**	**3.6**
g14-f27	n	11602	926	10759	2237	9698	2877	2750	2750
	t	13.0	**2.5**	15.3	3.1	17.2	3.3	3.1	3.1

coloring (1st,2nd), driver (3rd,4th), quasigroup completion (5th-7th), quasigroups with holes (8th,9th). In some of these problems maxRPC is much more efficient than AC. The heuristics, except H_4, can further improve on the performance of maxRPC making the adaptive algorithms considerably more efficient than MAC.

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

instance		AC	maxRPC	H_2	H_4	$H^{\vee}_{24}$	$H^{\vee}_{124}$
queen8-8-8	n	-	1458	2807	-	5863	4244
	t	>1h	3.15	2.9	>1h	5.1	**2.7**
games120-9	n	3208852	1392922	5511126	2265133	1604133	1452449
	t	403.7	432.3	834.3	293.7	216.1	**195.9**
driverlogw-08	n	3814	785	1003	3417	855	903
	t	13.2	25.5	6.9	9.2	**6.1**	6.2
driverlogw-09	n	14786	8342	10802	10627	8859	8895
	t	239.2	265.8	152.9	167.1	**137.8**	141.2
qcp-15-120-0	n	108336	21926	35394	101901	29990	27167
	t	98.4	43.3	39.9	83.9	33.4	**28.3**
qcp-15-120-5	n	387742	80424	84193	370461	81269	112290
	t	422.0	201.0	118.2	369.4	**117.7**	147.0
qcp-15-120-10	n	1136801	52112	58325	152497	76399	68046
	t	1178.0	113.6	**65.1**	145.1	88.6	71.2
qwh-20-166-0	n	104288	20236	15550	62993	15591	24725
	t	269.1	86.9	**42.3**	140.0	46.0	78.2
qwh-20-166-1	n	132842	22688	29681	66775	25147	39435
	t	355.4	111.4	88.2	151.1	**78.5**	116.7

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. Comparing the heuristics, H_2 and the combined ones that include H_2 display good performance on a variety of problems. It has to be noted that $H^{\vee}_{24}$ and $H^{\vee}_{124}$ were faster than AC in all instances we tried from the classes mentioned at the start of this section, except for some easy instances where they were slightly slower. H_1 and H_3 are effective on RLFAPs 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, 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.

Finally, Table 4 gives result from forced and geometric random problems. As is clear, in such problems that lack structure the heuristics do not reduce the node visits in a significant way and are outperformed by AC. The best heuristic is by far H_4. This is because 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 4. Nodes (n) and cpu times (t) in seconds from random instances.

instance		AC	maxRPC	H_2	H_4	$H^{\vee}_{24}$	$H^{\vee}_{124}$
frb35-17	n	23782	14920	15022	21182	15064	14642
	t	**13.5**	107.5	47.5	16.1	48.4	46.8
frb40-19	n	40058	20073	24446	32393	19722	22752
	t	**24.9**	151.6	76.8	27.9	63.4	76.1
geo50-20-75	n	227535	112785	148853	221211	142416	141726
	t	**218.9**	2089.4	765.7	247.1	748.3	750.1

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 costraints, which in turn can guide dom/wdeg to different variable selections, and hence different parts of the search space.

6 RELATED WORK

Building adaptive constraint solvers is a topic that has attracted considerable interest in the literature (see for example [1, 15, 9, 12]). 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 [8]. Freuder and Wallace proposed a technique, called *selective relaxation* which can be used to restrict AC propagation based on two criteria; the distance in the constraint graph of any variable from the currently instantiated one, and the proportion of values deleted [10]. 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 [4]. Schulte and Stuckey proposed a technique for selecting which propagator to apply to a given constraint, among an array of available constraint propagators, using priorities that are dynamically updated [17]. Similar ideas are also implemented in constraint solvers such as Choco [13]. *Probabilistic arc consistency* is a scheme that can help avoid some consistency checks and constraint revisions that are unlikely to cause any domain pruning [14]. As in [8], the scheme is based on information gathered by examining the supports of values in constraints which can be very expensive for non-binary constraints.

Our work is more closely related to [8] as the aim is to dynamically adapt the level of local consistency achieved on individual constraints. However, neither [8] or any of other works use information about failures captured in the form of constraint weights 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 CONCLUSION

We have proposed a number of simple heuristics for dynamically switching between different local consistencies 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 conflict-driven search heuristics. As we demonstrated, DWOs and value deletions in structured problems mostly occur in clusters of consecutive or 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 heuristics.

The work presented here is only a first step towards designing heuristics for adaptive constraint propagation using information gathered during search. There are several directions for future work. First of all we need to further evaluate the heuristics including their conjunctive combinations. We can also investigate different local consistencies for binary and non-binary problems, try to devise more sophisticated heuristics, and integrate with existing related works (e.g. [14]). 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 [16] and the explanation-based heuristics of [3].

REFERENCES

[1] J. Borrett, E Tsang, and N. Walsh, 'Adaptive Constraint Satisfaction: The Quickest First Principle', in *ECAI-96*, pp. 160–164, (1996).

[2] F. Boussemart, F. Heremy, C. Lecoutre, and L. Sais, 'Boosting systematic search by weighting constraints', in *ECAI-2004*, pp. 482–486, (2004).

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

[4] A. Chmeiss and L. Sais, 'Constraint Satisfaction Problems: Backtrack Search Revisited', in *ICTAI-2004*, pp. 252–257, (2004).

[5] R. Debruyne and C. Bessière, 'From restricted path consistency to max-restricted path consistency', in *CP-97*, pp. 312–326, (1997).

[6] R. Debruyne and C. Bessière, 'Domain Filtering Consistencies', *Journal of Artificial Intelligence Research*, **14**, 205–230, (2001).

[7] 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).

[8] H. El Sakkout, M. Wallace, and B. Richards, 'An Instance of Adaptive Constraint Propagation', in *CP-96*, pp. 164–178, (1996).

[9] S. Epstein, E. Freuder, R. Wallace, A Morozov, and Samuels. B., 'The Adaptive Constraint Engine', in *CP-2002*, pp. 525–540, (2002).

[10] E. Freuder and R.J. Wallace, 'Selective relaxation for constraint satisfaction problems', in *ICTAI-96*, (1996).

[11] D. Grimes and R.J. Wallace, 'Sampling Strategies and Variable Selection in Weighted Degree Heuristics', in *CP-2007*, pp. 831–838, (2007).

[12] *1st International Workshop on Autonomous Search (in conjunction with CP-07)*, eds., Y. Hamadi, E. Monfroy, and F. Saubion, 2007.

[13] F. Laburthe and Ocre, 'Choco : implementation du noyau d'un systeme de contraintes', in *JNPC-00*, pp. 151–165, (2000).

[14] D. Mehta and M.R.C. van Dongen, 'Probabilistic Consistency Boosts MAC and SAC', in *IJCAI-2007*, pp. 143–148, (2007).

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

[16] P. Refalo, 'Impact-based search strategies for constraint programming', in *CP-2004*, pp. 556–571, (2004).

[17] C. Schulte and P.J. Stuckey, 'Speeding Up Constraint Propagation', in *CP-2004*, pp. 619–633, (2004).