Learning How to Propagate Using Random Probing

Efstathios Stamatatos and Kostas Stergiou

Department of Information and Communication Systems Engineering
University of the Aegean, Samos, Greece
{stamatatos,konsterg}@aegean.gr

Abstract. In constraint programming there are often many choices regarding the propagation method to be used on the constraints of a problem. However, simple constraint solvers usually only apply a standard method, typically (generalized) arc consistency, on all constraints throughout search. Advanced solvers additionally allow for the modeler to choose among an array of propagators for certain (global) constraints. Since complex interactions exist among constraints, deciding in the modelling phase which propagation method to use on given constraints can be a hard task that ideally we would like to free the user from. In this paper we propose a simple technique towards the automation of this task. Our approach exploits information gathered from a random probing preprocessing phase to automatically decide on the propagation method to be used on each constraint. As we demonstrate, data gathered though probing allows for the solver to accurately differentiate between constraints that offer little pruning as opposed to ones that achieve many domain reductions, and also to detect constraints and variables that are amenable to certain propagation methods. Experimental results from an initial evaluation of the proposed method on binary CSPs demonstrate the benefits of our approach.

1 Introduction

Constraint propagation is a crucial reason for the success of constraint programming in solving hard combinatorial problems. Hence, this topic has attracted considerable interest and numerous generic and specialized constraint propagation techniques have been developed. As a result, when modelling a CSP there are, quite often, many choices regarding the propagation method to be used on the constraints of the problem. For example, advanced constraint solvers offer efficient filtering algorithms for both bounds consistency and generalized arc consistency (GAC), also known as domain consistency, for certain global constraints (e.g. alldifferent). The former are typically faster but the latter are stronger. As another example, there are numerous choices for local consistencies that can be applied on binary constraints. Despite the wealth of choices for constraint propagation, simple constraint solvers usually only apply a standard method, typically (G)AC, on all constraints throughout search. For instance, arc consistency is almost exclusively used on binary constraints. Advanced solvers can also apply a

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 263–278, 2009.

predetermined propagation method but in addition they allow for the modeler to choose among an array of propagators for certain (global) constraints. Finally, some solvers employ mechanisms for dynamically determining the propagation method during search based on the event that triggered propagation. Typically this is done on particular types of constraints such as arithmetic constraints.

Since complex interactions exist among constraints, which may only be revealed during search, deciding in the modelling phase which propagation method to use on given constraints can be a hard task that we would like to free the user from. For the case of a binary constraint, for example, it is very difficult to know a priori if choosing to propagate it using a strong local consistency such as singleton arc consistency or path consistency will pay off. Ideally, we would like to avoid using a strong propagation method on a constraint that will never, or rarely, cause domain reductions during search as this would result in needless cpu effort. Also, it would be preferable to choose say a cheap bounds consistency propagator for a constraint if we knew that stronger propagators achieve little extra pruning. But again this is very difficult to predict prior to search.

Deciding on which propagator to use for certain constraints based on static features of the problem is part of the modelling process and has attracted considerable interest. However, most of these works are problem-specific and require specialized modelling skills. The dynamic selection of propagators during search has also been investigated before, but to a far lesser extent (for example [10,17,15,19,20]). In this paper we propose a simple novel technique towards automating the task of choosing the right propagation method for individual constraints prior to search. Our approach differs from previous works as it does not the require the modeler's involvement in the process. Furthermore, it can be easily combined with dynamic methods or in itself extended to operate dynamically during search.

The proposed approach, which we call LPP (**L**earning **P**ropagators through **P**robing) uses information gathered from a random probing preprocessing phase to automatically decide on the propagation method to be used on each constraint. A *random probe* is a single run of a search algorithm with random variable ordering, a fixed cut-off, and propagation turned on. Random probes provide a sample of diverse areas in the search space and in our case can provide useful information regarding the percentage of fruitful revisions for each constraint, the number of value deletions caused by certain propagation methods, etc. We show that by exploiting such data the solver is able to accurately differentiate between constraints that offer little pruning as opposed to ones that achieve many domain reductions. As a result, the solver may automatically choose to propagate the former constraints using a low-cost propagation method and the latter using a stronger, and more expensive, propagator. Further to this, LPP can detect constraints and variables that are amenable to certain propagation methods. As we explain, these are accomplished through the use of a clustering algorithm that partitions the constraints into clusters having different features.

Although the method proposed is generic, we only present an initial evaluation on binary CSPs. To obtain the required data from random probing, we built

a *staged propagator* [18] for binary problems, i.e. a set of multiple propagators having varying cost and pruning power. This propagator progressively applies various local consistencies starting with bounds consistency and culminating in bounds singleton arc consistency. In a series of random probes where the propagator is applied after each variable assignment, we recorded the number of times each constraint was fruitfully revised, the local consistency that was responsible for each such revision, and the number of value deletions caused by each consistency. A comparison of these results to similar results obtained by running heuristically guided search to termination (using the same propagator) revealed interesting patterns. For instance, constraints that display a very low percentage of fruitful revisions can be accurately discovered through random probing.

Our methodology exploits the results of random probing to decide how to propagate each constraint during search using simple heuristic rules. Experimental results from various benchmarks demonstrate that LPP outperforms MAC, i.e. the standard search algorithm for binary problems, on hard instances, sometimes by a very large margin. Also, LPP is quite competitive with heuristics from [19] which dynamically switch between two local consistencies throughout search.

The rest of the paper is structured as follows. Section 2 gives some necessary background and introduces notation. In Section 3 we describe the staged propagator for binary constraints that we used in our experiments. Section 4 presents the LPP framework and gives experimental results demonstrating the accuracy of its predictions. In Section 5 we make an experimental evaluation of LPP on various binary problems. In Section 6 we discuss related work, and finally in Section 7 we conclude.

2 Background

A *Constraint Satisfaction Problem* (CSP) is 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 $D(x_1)\mathrm{x}\ldots\mathrm{x}D(x_k)$. 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. Maintaining arc consistency (MAC), which the most commonly used search algorithm for binary CSPs, applies AC to the problem after each variable assignment. 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 there is no empty domain [9]. A problem is SAC iff all variables are SAC.

Assuming finite integer domains for the variables, each domain $D(x_i)$ has a *minimum* and a *maximum* value, called the *bounds* of $D(x_i)$ and denoted by $min_{D(x_i)}$ and $max_{D(x_i)}$ respectively. A directed constraint c is *bounds consistent* (BC) iff both $min_{D(x_i)}$ and $max_{D(x_i)}$ have AC-supports on c. This definition of BC corresponds to BC(D) as defined in [4]. *Bounds SAC* (BSAC) is a

restricted version of SAC that only applies SAC on the bounds of the variables' domains [14].

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) [9]. 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).

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.

3 A Staged Propagator for Binary Constraints

Staged propagators were introduced by Schulte and Stuckey as a way to efficiently apply the different propagators that may be available for certain types of constraints [18]. A staged propagator for a constraint c is a set of propagators for c, having varying pruning power and cost, that are combined together. Each staged propagator has an internal state variable, called the state of the propagator, which determines the individual propagation method to be used once an event that triggers propagation for c occurs. For example, assuming that variable x_i appears in c, the removal of $min_{D(x_i)}$ may force the staged propagator to enter a state where a bounds consistency algorithm will be applied.

Here we describe a simple staged propagator for binary constraints that combines together four local consistencies: BC, AC, maxRPC, and BSAC. For simplicity, we will use the term *stage* to refer to one of the local consistencies that are combined together. For example, value deletions caused by the AC stage will refer to value deletion caused by the application of AC. We slightly abuse the definition of a staged propagator as we have implemented a variable-oriented propagation scheme where variables are the entities added to and removed from the propagation queue. Although in constraint solvers like Ilog Solver and Gecode the entities handled by the propagation queue are propagators, in the case of binary constraints variable-oriented propagation is more efficient. This has been previously demonstrated for arc consistency algorithms (e.g. [6,1]), but it is also true for higher level consistencies. To be precise, our experimental results showed a speed-up of up to three times in favor of variable-oriented propagation compared to its constraint-oriented counterpart[1].

Figure 1 gives an abstract high-level description of the staged propagator used. During preprocessing with random probing this propagator is applied as shown in Figure 1 after each variable assignment (*current_variable* denotes the currently assigned variable). The propagator removes a variable x_i from the queue and revises all constraints involving x_i. That is, it applies all four stages successively,

[1] These experimental results are omitted because of space restrictions.

function `Binary_Staged_Propagation`(X,D,C,*current_variable*)
1: add *current_variable* to Q
2: **while** $Q \neq \emptyset$
3: remove variable x_i from Q;
4: **for** any constraint c, with $var(c) = \{x_j, x_i\}$
5: successively apply BC,AC,maxRPC to c;
6: apply BSAC to x_j;
7: **if** $D(x_j) = \emptyset$ **then return** FAILURE;
8: **else if** $D(x_j)$ has been reduced **then** add x_j to Q;
9: **return** TRUE;

Fig. 1. A staged propagator for binary CSPs

as long as no domain wipeout (DWO) occurs. After the application of each stage the propagator records information concerning the pruning effects of the relevant constraint and the currently applied stage, as detailed in the next section (this is not shown in Figure 1 for simplicity). Once the process terminates, the data gathered is processed as will be explained below to select the propagation method to be applied on each constraint during search. Note that using the staged propagator in its full power throughout search is prohibitively expensive as it incurs many redundant revisions resulting in cpu times that can be orders of magnitude slower than MAC. Also, using SAC instead of BSAC results in more domain reductions albeit with a much higher cost.

4 Learning through Random Probing

In this section we first show that results gathered through random probing, concerning the pruning effects of the constraints, often reflect similar results gathered by running search to completion. Then we explain how LPP exploits this to decide on the propagator for individual constraints prior to search.

4.1 Accuracy of Learning

The LPP methodology utilizes the staged propagator described previously to gather data concerning the filtering power of the various propagation stages on individual constraints. For each constraint c we record the following information:

1. the number of times c was revised,
2. the ratio of fruitful revisions over the total number of revisions,
3. the ratio of fruitful revisions over the total number of revisions for each of the propagator's stages,
4. the total number of value deletions caused by c,
5. the ratio of value deletions over the total number of deletions caused by each stage separately.

The third item above is computed by simply recording the stage that is responsible for each value deletion during a fruitful revision of a constraint. Table 1

Table 1. Sample data gathered by random probing in a frequency assignment problem

recorded data	c_1	c_2	c_3	...
#revisions	60	63	69	
fr-ratio	0.28	0.47	0.05	
fr_{BC}-ratio	0.08	0.27	0.01	
fr_{AC}-ratio	0.03	0.35	0.01	
fr_{maxRPC}-ratio	0.28	0.00	0.04	
fr_{BSAC}-ratio	0.00	0.00	0.00	
#deletions	51	136	8	
del_{BC}-ratio	0.06	0.60	0.25	
del_{AC}-ratio	0.03	0.40	0.25	
del_{maxRPC}-ratio	0.91	0.00	0.50	
del_{BSAC}-ratio	0.00	0.00	0.00	

depicts part of the data gathered by random probing in tabular form. There is one column for each constraint, and each row corresponds to a piece of information concerning the pruning achieved by the constraints. The sample data shown is taken from a frequency assignment problem where we run 20 random probes each being cut off once 100 nodes (i.e. variable assignments) have been counted.

As one can see, constraint c_1 displayed a relatively high ratio of fruitful to total revisions (28% in row 2), the maxRPC stage contributed at least one value deletion in each of the constraint's fruitful revisions (the number in row 5 is the same as in row 2), and most of the value deletions it caused were due to the maxRPC stage (91% in row 10). Constraint c_2 displayed an even higher ratio of fruitful revisions but this time all value deletions were contributed by the BC and AC stages. Finally, constraint c_3 had a low ratio of fruitful revisions (only 5%) and the 8 value deletions it caused were due to either BC, AC, or maxRPC. BSAC did not contribute any value deletions for these three constraints.

As the data in Table 1 demonstrates, the various constraints can display different behavior with respect to their revisions and the pruning they cause. The interesting question is whether this behavior observed during random probing is relevant to the corresponding behavior of the constraints during heuristically guided search. Or in other words, whether we can "predict" how each constraint will behave based on the random probing results. First of all, to get a better picture of the distribution of the constraints into different patterns of behavior, we run the clustering algorithm fuzzy c-means on the data gathered by random probing. The following paragraph briefly discusses fuzzy c-means and then we present some clustering results.

Fuzzy c-means (FCM) is one of the most frequently used clustering algorithms. FCM allows one piece of data to belong to more than one clusters [5]. To this end, data are bound to each cluster by means of a membership function. Given a predefined number of clusters, FCM iteratively optimizes an objective function that is based on the distance of each data point from the cluster centers and the degree of membership in each cluster. In comparison to k-means, another well-known clustering algorithm, the FCM objective function differs in taking into account the degrees of membership in each cluster as well as an additional parameter (the fuzzifier) that determines the level of cluster fuzziness. A large fuzzifier results in fuzzier clusters whereas a fuzzifier equal to 1 results in crisp

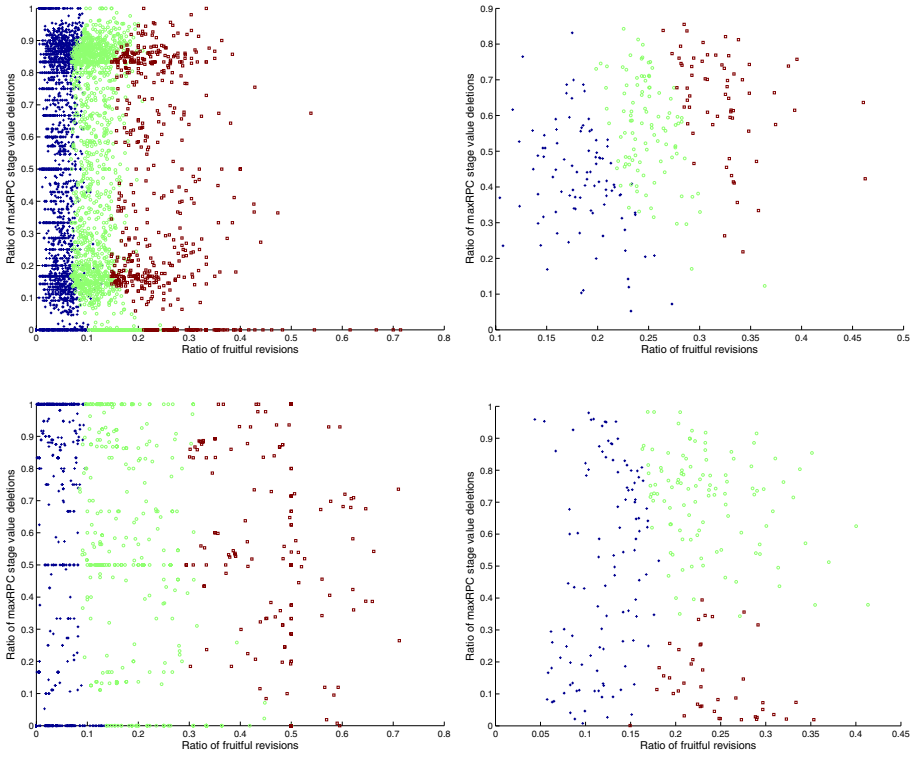

Fig. 2. Clustering results from a frequency assignment (left plots) and a random problem (right plots). The x-axis gives the ratio of fruitful revisions while the y-axis gives the ratio of value deletions due to the maxRPC stage. The top plots show clusters formed from the random probing results while the bottom ones show clusters formed from search results.

partitioning. The iteration stops when the degrees of membership of data in each cluster are not significantly modified in successive iteration steps. Similarly to k-means, FCM tends to group data spatially according to their distance from the cluster centers. However, this spatial partitioning is more flexible due to the fuzzifier parameter. In this paper, we used 3 clusters and set the fuzzifier to 2 based on preliminary experiments.

The top plots in Figures 2 and 3 show how constraints are clustered after running FCM on the data gathered by random probing for four different problems. The input parameters for FCM were the ratio of fruitful revisions and the corresponding ratios for the propagation stages. The horizontal axis in the figures gives the ratio of fruitful revisions while the vertical axis gives the ratio of value deletions caused by the maxRPC stage. As is evident, in all four problems the three clusters created partition the constraints mainly according to the ratio of fruitful revisions. Going from left to right the three clusters include constraints with increasing ratio. Apart from this differentiation additional useful

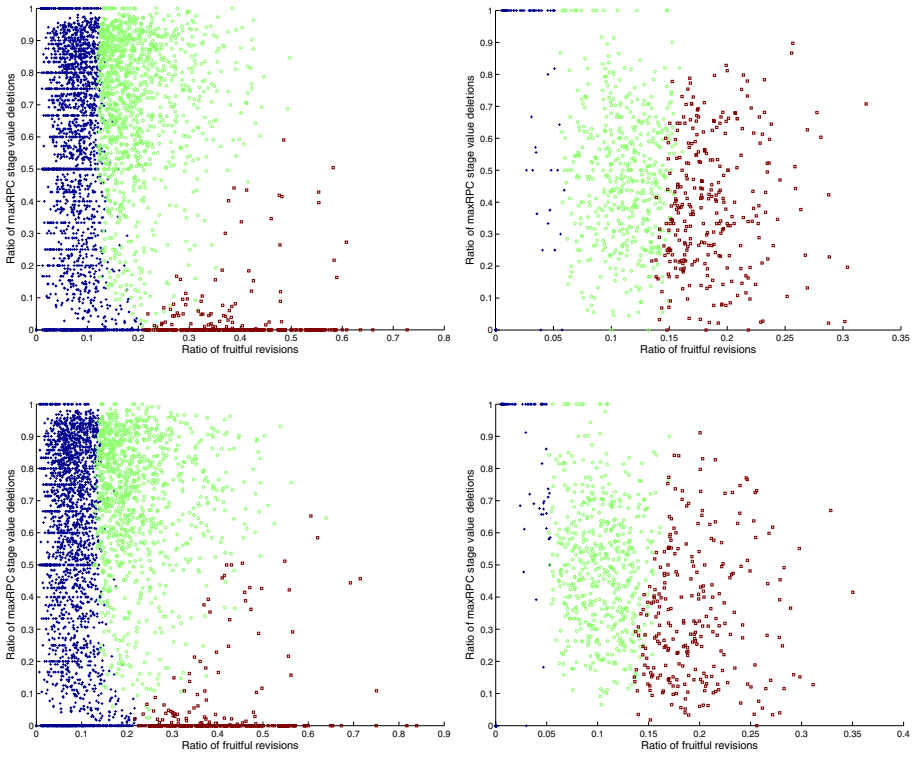

Fig. 3. Clustering results from a frequency assignment (left plots) and a quasigroup completion problem (right plots). The top plots show clusters formed from the random probing results while the bottom ones show clusters formed from accumulated probing and search results.

information can also be extracted. For example, the rightmost cluster in the top left plot of Figure 3 mostly includes constraints with low ratio of value deletions by maxRPC. Hence, it seems that for these constraints the maxRPC stage has little effect.

To answer the question posted above on whether the pruning behavior of the constraints during random probing is relevant to their behavior during search, we run a search algorithm that applied the staged propagator after each variable assignment. We also recorded the same information regarding revisions and value deletions as during random probing. The bottom plots in Figure 2 show how constraints are clustered after running FCM on this data for the same two problems as in the top plots. Figure 3 displays similar results but in these two cases random probing was applied prior to search. That is, the data is accumulated from both preprocessing and search. In the three structured problems (the left problem in Figure 2 and both problems in Figure 3) the distribution of the constraints in the three clusters resembles the corresponding distribution from the random probing results, especially in Figure 3. In contrast, the clusters in

the random problem (right plots in Figure 2) are quite different compared to the corresponding clusters from random probing. This indicates that in the absence of structure it is difficult to predict the behavior of the constraints using random probing.

Note that the bottom left plot in Figure 2 includes fewer data points (i.e. constraints) than the top left one. This is because heuristically guided search focuses on certain parts of the search space and as a result many constraints are not revised at all, which means that the corresponding data points have (0,0) coordinates on the plot. Also, in the bottom plot the three clusters are shifted to the right compared to the top one. However, the membership of constraints to clusters remains similar. That is, most constraints that belong to a particular cluster in the top plot, say the middle one, belong to the corresponding cluster in the bottom plot as well.

Table 2. Accuracy of clusters for various problems. The second column gives the number of constraints in each problem.

instance	e	**% accuracy**	% left cluster accuracy
scen11-f6	4102	70	96
scen11-f7	4102	79	97
driver9	17446	83	89
qcp15-120-9	3149	92	99
qwh20-166-1	7599	95	99
qwh20-166-8	7599	95	99
3-fullins-5-5	33750	58	94
myciel7-4	2359	65	72
frb40-19-0	320	59	73

Table 2 gives further evidence concerning the similarity of the clusters created using the random probing results compared to the clusters created using results from search. Each row in the table gives results from a benchmark problem. These problems are all structured (either real or academic) apart from the last one which was randomly generated (see Section 5 for more details). The third column gives the percentage of constraints that were assigned to corresponding clusters in both the random probing and the final clusterings. The fourth column gives the percentage of constraints that were assigned to the leftmost cluster in the preprocessing clustering and remained assigned to the leftmost cluster in the final clustering. This is particularly useful as it demonstrates the accuracy in identifying constraints that have a low ratio of fruitful revisions. For most problems the percentage is very high, getting close to 100%. As expected, the similarity between the clusterings is lower in the case of the random problem.

4.2 Exploiting Learning to Determine Propagators

Having shown that some important aspects of the pruning behavior that the constraints display can be predicted through random probing, the question that

naturally arises is how to exploit this in order to make informed automatic decisions about the propagators to use on individual constraints during search. A naive answer would be to simply look at the results gathered (e.g. Table 1) and select the propagation stage that caused the highest number of deletions for a constraint c as the propagator to be used on c. Although this is not entirely useless (experiments showed it outperforms MAC!), it suffers from certain drawbacks. Most notably it ignores the ratio of fruitful revisions which is a crucial piece of information. Choosing a strong propagator for constraints that have a low ratio is not cost-effective. For example, following this naive approach the solver would select to propagate constraint c_3 of Table 1 using maxRPC. This can result in many redundant revisions of high cost.

LPP answers the above question by exploiting the results provided by the FCM clustering algorithm and making the decisions using simple (heuristic) rules which basically constitute a decision tree. Note that it is easy to identify the three clusters by looking at the clusters' centers. The cluster whose center has the lowest value of fruitful revisions ratio is the one which includes constraints with low ratio of fruitful revisions. Accordingly, we can differentiate the other two clusters through their centers. The rules we have used are as follows.

- Any constraint belonging to the cluster whose center has the lowest ratio of fruitful revisions (the leftmost cluster in the plots) is propagated with AC or BC, depending on which one has the highest ratio of deletions.
- Any constraint belonging to the cluster whose center has the highest ratio of fruitful revisions (the rightmost cluster in the plots) is propagated with maxRPC if 1) the cluster center's ratio of fruitful revisions by maxRPC (fr_{maxRPC}-ratio in Table 1) is the highest among the three clusters and 2) maxRPC has the highest ratio of deletions (del_{maxRPC}-ratio in Table 1) compared to the other stages for this constraint. Otherwise, it is propagated using heuristic $\mathrm{H}_{12}^{\vee}$ from [19]. This heuristic switches between AC and maxRPC during search according to certain conditions explained below.
- Any constraint belonging to the remaining cluster (the middle cluster) is propagated using heuristic $\mathrm{H}_{12}^{\vee}$ except if: 1) the cluster center's ratio of fruitful revisions by maxRPC is the highest among the three clusters in which case it is propagated with maxRPC, or 2) the maxRPC stage does not cause any deletions at all, in which case it is propagated with AC.
- BSAC is applied on any variable whose ratio of fruitful calls to BSAC over the total number of calls is more that 0.5. That is, line 6 in Figure 1 is only executed for these variables.

Heuristic $\mathrm{H}_{12}^{\vee}$ monitors and counts revisions, DWOs and value deletions for the constraints in the problem. It uses two (user defined) thresholds l_1 and l_2, set to 100 and 10 in this paper, to switch between a weak but cheap local consistency W and a stronger but more expensive one S. A constraint c is made S if the number of times it was revised since the last time it caused a DWO is less or equal to l_1, or if the number of times it was revised since the last time it caused a

value deletion is less or equal to l_2. If none of these conditions holds, c it is made W. In this paper W and S were set to AC and maxRPC respectively. Setting S to BSAC or SAC resulted in a very cost-inefficient method.

A drawback of our method is that the heuristic rules described above were predetermined based on intuition and preliminary experiments, and hence required expertise. The intuition is simple: we select a low-cost propagator for constraints that displayed many redundant revisions during preprocessing, and a high-cost but more efficient one for constraints that displayed many fruitful revisions most of which were due to the high-cost propagator. Automatic generation of heuristic rules is an important topic that requires further research.

5 Experimental Results

In this section we present an initial evaluation of LPP on binary CSPs. We compare the method to the widely used MAC algorithm and also to heuristic $H_{12}^{\vee}$ applied to all constraints of the problem as proposed in [19]. The solver we used applies d-way branching, lexicographic value ordering, the dom/wdeg variable ordering heuristic [7], and restarts. Concerning the restart policy, 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. We set the number of random probes to 20 and the cut-off limit for each probe to 100 nodes. We noticed little variance in the results when these settings changed, but finding the "optimal" settings for each problem is an issue that requires further research. Another topic for future work is the use of random probes with random value ordering which may result in even more diverse sampling of the search space. To keep preprocessing times manageable BSAC, which can be quite time consuming, was only applied in 1/5th of the nodes (randomly selected).

We experimented with the following classes of problems: radio links frequency assignment (RLFAPs), graph coloring (GC), haystacks (H), quasigroup completion (QCP), quasigroups with holes (QWH), forced random problems (R). All apart from the last class are structured binary CSPs. Tables 3 and 4 give indicative experimental results. The specific benchmark instances taken from C. Lecoutre's web page. The first table gives results from insoluble problems while the second from soluble ones. For LPP we give both the total cpu time and the time required for random probing and clustering. Note that the time required for clustering is negligible compared to that for random probing.

As results demonstrate, LPP can be considerably more efficient than MAC on the majority of the problems, and especially on the hard insoluble ones. The random probing preprocessing phase consumes a significant portion of the execution time for easier instances, but in most cases this becomes negligible as the problems become harder. Comparing heuristic $H_{12}^{\vee}$ to LPP we can see that the methods are competitive with LPP often being faster despite the time spent on preprocessing. LPP, as well as $H_{12}^{\vee}$, is not competitive with MAC on random

Table 3. Nodes (n) and cpu times (t) in seconds from insoluble problems. The LPP column gives the total cpu time of preprocessing + search and in brackets the time required for preprocessing (i.e. random probing and clustering). A time out limit of 2 hours was set.

type	**instance**		MAC	$H_{12}^{\vee}$	LPP
RLFAP	scen11-f6	n	74,879	7,895	4,871
		t	58	14	66 (50)
RLFAP	scen11-f5	n	321,435	52,750	12,501
		t	254	94	99 (53)
RLFAP	scen11-f4	n	1,110,401	167,786	22,112
		t	856	266	110 (61)
RLFAP	scen11-f3	n	4,995,046	167,596	23,334
		t	3917	274	111 (62)
GC	homer-8	n	228,495	11,770	201,023
		t	102	7	118 (3)
GC	myciel5-5	n	22,640,358	22,640,358	22,640,358
		t	638	2021	842 (1)
GC	myciel6-5	n	6,915,618	6,915,618	6,915,618
		t	654	2815	896 (5)
GC	miles-500-10	n	19,996,866	9,693	18,048
		t	3596	3	15 (9)
H	haystacks-5	n	1,203,768	3,256	942
		t	13	0.5	0.2 (0.1)
H	haystacks-6	n	-	23,328	25,732
		t	>2h.	2	2 (0.2)
QCP	qcp15-120-10	n	8,580,800	2,747,682	113,487
		t	1860	1340	50 (5)
QCP	qcp15-120-13	n	1,007,089	155,971	230,591
		t	235	71	108 (4)

problems, which gives further evidence that the absence of structure hinders the accuracy of the learning process.

Interestingly, on the myciel graph coloring problems maxRPC and BSAC do not offer any more pruning than AC. LPP discovers this during preprocessing and does not select these two consistencies for any constraint. Hence the same node visits but reduced cpu times compared to $H_{12}^{\vee}$ which "blindly" switches between AC and maxRPC during search. However, the second rule of Section 4.2 selects to propagate some constraints using $H_{12}^{\vee}$ which accounts for the increased times compared to MAC. On a negative note, problem homer-8 is an example where LPP fails to interpret the random probing results in an efficient way. Although the leftmost cluster created includes constraints with low ratio of fruitful revisions, this is the only cluster that includes constraints where the maxRPC stage makes value deletions. Despite this, all constraints in this cluster are selected to be propagated with AC or BC which accounts for the significant difference in node visits and cpu time compared to $H_{12}^{\vee}$.

Table 4. Nodes (n) and cpu times (t) in seconds from various soluble problems

type	**instance**		MAC	$H_{12}^{\vee}$	LPP
QCP	qcp15-120-9	n	135,267	29,812	29,383
		t	30	12	31 (4)
QCP	qcp20-187-1	n	189,942	344,418	172,574
		t	102	262	149 (10)
QWH	qwh20-166-7	n	88,429	10,945	22,023
		t	206	19	49 (9)
QWH	qwh20-166-8	n	70,945	12,565	29,199
		t	160	23	72 (10)
GC	homer-10	n	-	3,505	2,994
		t	>2h.	3	6 (4)
R	frb35-17-0	n	59,910	10,155	4,320
		t	14	13	20 (10)
R	frb40-19-0	n	170,345	46,596	94,722
		t	45	80	238 (10)
R	frb45-21-0	n	1,028,028	767,550	1,205,280
		t	320	1862	1844 (10)

6 Related Work

Random probing has been used in constraint programming before, albeit in different contexts. Grimes and Wallace have used probing to initialize the scores of the dom/wdeg heuristic and in this way make it more informed at the initial stages of search [12][2]. Ruml proposed an adaptive probing scheme that iteratively adapts the search guiding heuristic in subsequent searches [16]. Beck used probing in the context of multi-point constructive search [2]. Probes are used to initialize a set of "elite" partial solutions some of which are thereafter used as starting points for subsequent searches. Finally, probing has been to measure the *promise* of variable ordering heuristics [3].

There have been several efforts, which are mainly related to the modelling of specific CSPs, on deciding which propagator to apply on certain constraints based on static features of the problem. As most of these works are not general but rather problem-specific, we will not review them in detail. Instead, we will focus on approaches that try to select the propagation method using dynamic features of the problem.

Adaptive constraint propagation has attracted interest in the past. The most common manifestation of adaptive propagation 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.

[2] Note that we did not do this in our experiments to avoid adding bias to the results.

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 [10]. 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 [11]. 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 [8].

Schulte and Stuckey proposed techniques for dynamically selecting which propagator to apply to a given constraint using priorities and staged propagators [17]. 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 [17]. 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. 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 [15]. As in [10], 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 [20]. Finally, Stergiou proposed heuristics for dynamically switching between two propagators on individual constraints during search [19]. These heuristics take advantage of the fact that in structured problems propagation events usually occur in clusters, but it is difficult to see how they can be generalized to work with more than two propagators.

As discussed, our work makes a static selection of propagator for individual constraints, but it can be combined with most dynamic approaches as we demonstrated for [19]. Combining with such approaches is an interesting direction for future work. Also, we can extend LPP to a dynamic version where constraint propagation data acquired during search is taken into account to perhaps readjust the initial static propagator choices if necessary.

7 Conclusions

Choosing the right propagator for specific constraints prior to search is a difficult task for CP modelers. We have presented LPP, a simple approach toward automating this task. Our approach is based on gathering data concerning the pruning behavior of the constraints in a random probing preprocessing phase. A case study on binary constraints was presented, and as experimental results demonstrated, decisions taken using the random probing results can be quite accurate in many cases, resulting in improved cpu times during search. In addition, we believe that our work emphasizes the largely untapped potential of

using machine learning techniques, such as clustering, to boost the performance of CP systems.

A drawback of LPP is that the preprocessing phase can be too expensive on very large problems with many variables and constraints. To overcome this we may lift the requirement that all stages of the propagator used are applied at each node and for each constraint. In the future we plan to extend the work presented here to include a wider range of local consistencies for binary as well as non-binary constraints. Also, we would like to investigate the use of machine learning techniques to automatically build the decision tree which exploiting random probing results will be able to propose propagators for the constraints.

References

1. Balafoutis, T., Stergiou, K.: Exploiting constraint weights for revision ordering in Arc Consistency Algorithms. In: ECAI 2008 Workshop on Modeling and Solving Problems with Constraints (2008)
2. Beck, C.: Solution-Guided Multi-Point Constructive Search for Job Shop Scheduling. JAIR 29, 49–77 (2007)
3. Beck, C., Prosser, P., Wallace, R.: Variable Ordering Heuristics Show Promise. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 711–715. Springer, Heidelberg (2004)
4. Bessiere, C.: Constraint propagation. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, ch. 3. Elsevier, Amsterdam (2006)
5. Bezdek, J.C. (ed.): Pattern Recognition with Fuzzy Objective Function Algorithms. Kluwer Academic Publishers, Norwell (1981)
6. Boussemart, F., Hemery, F., Lecoutre, C.: Revision ordering heuristics for the Constraint Satisfaction Problem. In: CP 2004 Workshop on Constraint Propagation and Implementation (2004)
7. Boussemart, F., Heremy, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: ECAI 2004, pp. 482–486 (2004)
8. Chmeiss, A., Sais, L.: Constraint Satisfaction Problems: Backtrack Search Revisited. In: ICTAI 2004, pp. 252–257 (2004)
9. Debruyne, R., Bessière, C.: From restricted path consistency to max-restricted path consistency. In: Smolka, G. (ed.) CP 1997. LNCS, vol. 1330, pp. 312–326. Springer, Heidelberg (1997)
10. El Sakkout, H., Wallace, M., Richards, B.: An Instance of Adaptive Constraint Propagation. In: Freuder, E.C. (ed.) CP 1996. LNCS, vol. 1118, pp. 164–178. Springer, Heidelberg (1996)
11. Freuder, E., Wallace, R.J.: Selective relaxation for constraint satisfaction problems. In: ICTAI 1996 (1996)
12. Grimes, D., Wallace, R.J.: Sampling Strategies and Variable Selection in Weighted Degree Heuristics. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 831–838. Springer, Heidelberg (2007)
13. Laburthe, F., Ocre: Choco: implementation du noyau d'un systeme de contraintes. In: JNPC 2000, pp. 151–165 (2000)
14. Lecoutre, C., Prosser, P.: Maintaining Singleton Arc Consistency. In: 3rd International Workshop on Constraint Propagation And Implementation (CPAI 2006), pp. 47–61 (2006)

15. Mehta, D., van Dongen, M.R.C.: Probabilistic Consistency Boosts MAC and SAC. In: IJCAI 2007, pp. 143–148 (2007)
16. Ruml, W.: Incomplete Tree Search using Adaptive Probing. In: IJCAI 2001, pp. 235–241 (2001)
17. Schulte, C., Stuckey, P.J.: Speeding Up Constraint Propagation. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 619–633. Springer, Heidelberg (2004)
18. Schulte, C., Stuckey, P.J.: Efficient Constraint Propagation Engines. ACM Trans. Program. Lang. Syst. 31(1), 1–43 (2008)
19. Stergiou, K.: Heuristics for Dynamically Adapting Propagation. In: ECAI 2008, pp. 485–489 (2008)
20. Szymanek, R., Lecoutre, C.: Constraint-Level Advice for Shaving. In: ICLP 2008, pp. 636–650 (2008)