Learning Propagation Policies

Susan L. Epstein[1], Richard Wallace[2], Eugene Freuder[2], Xingjian Li[1]

[1] Department of Computer Science
Hunter College of The City University of New York
695 Park Avenue, New York, NY 10021 USA
susan.epstein@hunter.cuny.edu
http://www.cs.hunter.cuny.edu/~epstein
[2] Cork Constraint Computation Centre
rwallace/efreuder@4c.ucc.ie

Abstract. Propagation is intended to remove from consideration values that will not lead to a solution. A propagation policy includes preprocessing, selection of a propagation method, identification of relevant method parameters, and switching among methods. We show here the significant impact a propagation policy has on solution time, and that the choice of a good propagation policy varies with the problem class. We also demonstrate how a propagation policy can be learned automatically and can substantially improve performance.

1 Introduction

Since the earliest days of the modern study of backtracking (Golumb and Baumert 1965), we have faced the question of the best tradeoff between search and inference: how much constraint propagation is cost efficient to interleave with backtrack search choices? The answer is almost certainly "it depends" -- on the problem under consideration, as well as on the method of propagation. This answer, however, provides little comfort to the constraint programming practitioner. In this paper we extend the Adaptive Constraint Engine (*ACE*) (Epstein and Freuder 2001; Epstein, Freuder et al. 2002) to construct automatically an appropriate "customized propagation policy" when confronted with a class of problems.

The classic propagation choices are forward checking or maintaining arc consistency, embodied in the FC and MAC algorithms. Forward checking is the minimal lookahead one must do to assure consistency with previous choices; MAC restores full arc consistency after every choice. A variety of intermediate methods have been proposed, which do more propagation than FC but less than AC. We employ here the restricted propagation methods of (Freuder and Wallace 1991) and develop new variants. Specifically we develop an AC version of FC-based restricted propagation and add to restricted propagation the option of thresholds that are functions of search depth. We also introduce a limited "one-pass" form of AC preprocessing, and the "meta-method" of switching propagation methods at different search depths.

We show that our new intermediate methods excel in appropriate circumstances. As expected, however, they too are no panaceas. We would like to use the new and old methods together as “building blocks” to be chosen, tuned and combined to best effect for individual circumstances, but that presents the constraint programmer with a bewildering array of choices and combinations. This is where ACE comes in.

Specifically, ACE trains on a set of problems from a given class to automatically:

- decide which form of preprocessing to do
- decide whether to use FC, AC, or any of the intermediate propagation methods
- decide upon thresholds for intermediate methods
- decide whether to switch between methods, and determine switching point depths

We call such a set of decisions a *propagation policy*. The classical propagation policies are FC (with limited preprocessing) and MAC. We demonstrate that, for a fixed search method, the customized propagation policies constructed by ACE for various problem classes sometimes outperform both of the classical extremes and never underperforms them (cf. Chmeiss and Sais, 2004 on FC versus AC). One would expect that an appropriate propagation policy would depend not just on the problem class, but also on the *search method* employed, specifically the variable-ordering and value-ordering heuristics. We present preliminary evidence to show that ACE can choose propagation policies appropriate for different search methods as well.

We then provide detailed experiments to suggest that not only is ACE choosing good propagation policies, but most likely it is choosing essentially the best policies that can be constructed from the building blocks provided. Our experiments incorporate a representative sample of such building blocks, but additional variations, old or new, could naturally be accommodated. In fact, we have effectively demonstrated here, with the positive results obtained for some of our new methods, and the negative results obtained for others, that a constraint programmer can throw new ideas into the mix, and ACE will not be confused, but will sort the wheat from the chaff, using new ideas appropriate to the circumstances, and eschewing inappropriate ones.

Section 2 describes the building blocks, new and old, from which the propagation policies are constructed and carefully defines essential terminology. Section 3 describes how ACE learns a propagation policy. Section 4 presents the results of the learning experiments. Section 5 provides a more detailed study of various methods and combinations, which provides further evidence for the ability of some of our new methods to excel, and further support for the choices that ACE made. Section 6 discusses related and future work.

2 The building blocks

A constraint satisfaction problem (*CSP*) is a triple, <*X, D, C*>, where *X* is a set of variables, *D* is the set of domains for *X*, and *C* is a set of constraints on *X*. A *solution* for a CSP is a set of values, one for each variable, that satisfies *C*. In this paper, we restrict our discussion to binary constraints. A *partial assignment* is a set of values for some of *X* (the *past variables*) with the remainder (the *future variables*) described by their (possibly reduced) domains. A partial assignment is said to be *consistent* if it does not violate *C*. Search for a solution, then, can be represented as movement from an initial state where all variables are future variables to a consistent assignment where all variables are past variables. In the paradigm used here, search alternately selects a *current variable* and then assigns it a value. When a propagation method executes after each assignment during search, and removes any inconsistent values from the domains of future variables, the method is said to be *maintained.* We consider only maintained consistency here.

A binary CSP can also be represented as a labeled graph (a *constraint graph*), where each variable is a node, each constraint is an edge, nodes are labeled by their domains, and edges are labeled by their acceptable value pairs. A pair of nodes that share an edge are said to be *neighbors*. The *degree* of a node is the number of neighbors it has. Here, the *density d* of a CSP on *n* variables is the percentage of edges it includes beyond the *n*-1 necessary to connect the graph. The *tightness t* of a graph is the percentage of possible value pairs each edge excludes. With these parameters, we represent a class of random problems as <*n,m,d,t*>, where *m* is the maximum initial domain size. For fixed values of *n* and *m*, values of *d* and *t* that make the problems particularly difficult are said to lie at the *phase transition*.

For clarity in our work, we make the following distinctions. *Neighborhood consistency* (*NC*) guarantees that, for each variable *x,* each value in the domains of *x's* neighbors in the constraint graph is consistent with some value in the domain of *x*. *Forward checking* (*FC*) is an algorithm that combines search with NC propagation after each choice; it considers those neighbors of the just-assigned variable that are future variables, compares the neighbors' domains with the newly-assigned value, and removes from them any value inconsistent with the new value. Thus FC guarantees only that any consistent assignment to one variable can be extended to a consistent partial solution on two variables. *Arc consistency* guarantees that for every value *v* in the domain of each variable *x*, and for every constraint $c \in C$ between *x* and another variable *y*, there is a value *w* in the domain of *y* such that (*v w*) satisfies *c*. MAC is an algorithm that combines search with AC propagation after each choice. Each test that a value is supported by another value in a neighboring domain is called a *constraint check*. One would expect a higher level of consistency to improve search, but such consistency demands more computation. *Initialization* is a propagation pass prior to search. Let *one-pass AC initialization* be a process that, before search, examines each edge once, in both directions, to remove unsupported values. We investigate both one-pass AC initialization and (full) AC initialization here.

Research results on constraint propagation during search initially favored FC's simple one-step lookahead (Haralick and Elliott 1980). Later work indicated that for hard problems the constraint propagation method of choice was often AC (Sabin and Freuder 1994). This in turn drove research on clever data structures (Bessière and Régin 1996; Bessière, Freuder et al. 1999; Bessière and Régin 2001) and elaborate AC queue management to speed AC's computation (Lecoutre, Boussemart et al. 2003; Mehta and van Dongen 2005). As a result, maintained arc consistency (*MAC*) has become the most popular propagation method. There are many implementations of MAC. Here we use *MAC-3*, where each iteration processes a queue of edges, confirming for each edge from *x* to *y* that the domain values of *y* are supported by the domain values of *x*. Whenever such confirmation reduces the domain of *y*, edges (*y z*) are added to the queue, where *z* is a future variable and a neighbor of *y*. Before search, MAC-3 does a full AC, with an initial queue that includes every edge in the graph.

During search, immediately after variable *v* is assigned a value, MAC-3 begins with a queue that includes all the edges from *v* to future variables that are its neighbors. Our implementation has no special queue management and no special treatment for variables whose domain is reduced to a single value.

Many search methods depend upon the efficacy of a propagation method because they consider *dynamic domain size* (the number of values consistent with the current partial solution) when selecting the next variable. Prominent among these are *Min Domain* (which selects as the next variable the future variable with minimum dynamic domain size), and *Min Domain/Degree* (which minimizes the ratio of dynamic domain size to static degree when selecting a variable). This work assumes, for each problem class, a known, efficient search method which references dynamic domain size. Unless otherwise stated, the search method used here is Min Domain/Degree. Lexical order is used to break ties and in choosing values.

2.1 Problem classes

Intuitively, the degree and the nature of connectivity in the constraint graph can influence the potential impact of constraint propagation. In the experiments described here, we therefore consider a variety of random, same-size problems: *sparse* <30, 8, 0.05, 0.5>, *simple* <30, 8, 0.1, 0.5>, *medium* <30, 8, 0.12, 0.5>, and *hard* <30, 8, 0.26, 0.34>, the latter so named because they are at a phase transition. Random problems, however, have arbitrary constraints and lack reliable structure; they may obscure some interesting properties of propagation. Therefore, we also consider three additional problem classes, to explore the impact of propagation further:

• A *coloring problem* is a CSP with constraints that prohibit assigning the same value to certain pairs of variables. The coloring problems we use here have 30 variables, domain size 8, and density 0.58. too hight density! #colors?

• A *geometric* CSP is formed from a random set of points in the Cartesian plane — each point becomes a variable in the problem; constraints are formed among any pair of variables within a specified distance of each other, with additional constraints added to connect the underlying constraint graph (Johnson, Aragon et al. 1989). The result is a constraint graph ridden with clusters (not necessarily cliques) of vertices which can prove particularly difficult for traditional solvers. The geometric problems we use here have 50 variables, domain size 10, and tightness 0.18. Density is determined by the distance parameter (here, 0.4) and the spacing of the points in the unit square; for a sample of 20 of these problems the average density was 0.32.

• An *n X n quasigroup* is a Latin square of size *n:* each of n^2 variables participates in $2n–2$ binary constraints. *Quasigroups with holes* specifies values for some variables (the unspecified variables are the *holes*). The phase transition for quasigroups with holes is about 33% non-holes (Achlioptas, Gomes et al. 2000). The problems we use here are 10 X 10 quasigroups with 60 holes and are balanced (i.e., the holes are evenly distributed across the square). For quasigroups with balanced holes, we use Min Domain, which selects the same variables as Min Domain/Degree.

All problems have at least one solution, but some geometric and quasigroup problems are so difficult that some in our training set were never solved within 1000 seconds.

2.2 Locality and response in propagation

The propagation methods detailed in this section (some of which were first described in Freuder and Wallace, 1991) seek a balance between AC and FC. Each of them potentially does more work than FC but less than AC. The intuition behind these methods is that propagation may only be effective in the neighborhood of the current variable (*locality*) or that it is only effective if it reduces the domains of the neighbors of the current variable substantially (*response*).

We address locality with two approaches: one extends FC's reach beyond the current variable and its neighbors; the other limits AC to the vicinity of the current variable. More formally, let the p-neighborhood of a variable be the set of all future variables within distance p of it in the dynamic constraint graph.

• *FC-spread* first forward checks and then permits propagation to extend beyond the current variable's immediate neighbors within its p-neighborhood. FC-spread can be thought of as a kind of spreading activation, which processes each edge at most once, and considers only future variables within the p-neighborhood of the current variable. FC-spread with $p = 1$ is equivalent to FC.

• *AC-bound* first forward checks and then performs AC with a queue restricted to edges within the p-neighborhood of the current variable. AC-bound with $p = n$-1 is equivalent to AC. (A similar method was examined recently by Chmeiss and Sais, 2004.)

We address response with approaches whose names include R for "response":

• *FCR* first forward checks the neighbors of the current variable and then continues to check edges only from neighbors whose domain sizes have been reduced by at least r%. No edge is visited more than once.

• *ACR* is like FCR, but it permits edges from variables with sufficiently-reduced domains to re-enter the queue.

It may be the case that the appropriate response varies with the search depth, that is, that r is not uniform during search. We address this with two approaches whose names include D for "depth":

• *FCRD* first forward checks and then performs AC with a queue that includes edges only from neighbors whose domain sizes have been reduced by at least r%, where r is a function of search depth. No edge is visited more than once.

• *ACRD* is like FCRD, but it permits edges from variables with sufficiently-reduced domains to re-enter the queue.

2.3 Switching and initialization in propagation methods

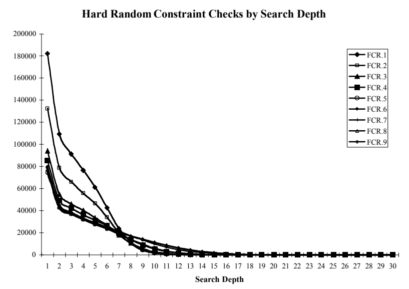
At some point during search a problem may become so easy that FC is sufficient. The solver may have already instantiated a backdoor (Ruan, Horvitz et al. 2004) so that the remainder of the problem is relatively easy. Indeed, the constraint graph may have become acyclic, in which case, after a single AC pass, it can be solved backtrack-free with a pre-computed (width-one) ordering of the variables and random value selection (Freuder 1982). This is documented in Figure 1(a) which plots the number of constraint checks calculated with Min Domain/Degree and FCR with different r values against search depth for the hard random problems. Initially the FCR methods do far less work than AC itself, and do substantially less work (as does AC) after some point, here when about 9 variables have been bound. We tested FCR for r = .1, .2,…,.9 on a set of 100 problems. We therefore investigate propagation methods of the form x-FC with *terminal switch* s_t, where x is itself a successful method. While no more than s_t variables are bound, x-FC uses x to propagate; afterwards it uses FC.

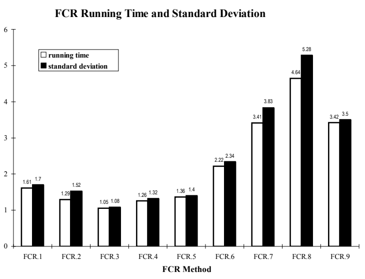
Problems also differ in the number of values removed by AC immediately after the first few value assignments. We therefore investigate propagation methods of the form FC-x with *initial switch* s_i, where x is itself a successful method. While no more than s_i variables are bound, FC-x uses FC to propagate; afterwards it uses x. Finally we investigated propagation methods of the form FC-x-FC with both initial and terminal switches between FC and a successful method x.

Because most search methods (including Min Domain/Degree) depend in part on dynamic domain size to select variables for assignment, a solver may derive some clues on its initial selection of a variable with AC initialization. This is common CSP practice, as well as part of MAC-3. Nonetheless, we solved 100 problems from each problem set twice, once with one-pass AC initialization and the second time with AC initialization, using Min Domain/Degree to search and AC to propagate after the initialization. There was no statistically significant difference at the 95% confidence level, in initialization time, in solution time, or in total time between AC initialization and one-pass AC initialization in any problem class. We therefore chose to make either one-pass AC or AC initialization our final building block.

3 The learning algorithm

Tweaking parameters empirically is tedious and inexact. Ideally, a solver should learn which propagation policy to use. We have enhanced ACE to learn a good propagation policy for a fixed search method and problem class as follows. (A high-level synopsis appears in Figure 2.) The program first solves a set of problems (here, 100) with FC and gathers statistics on the *response* (percentage reduction in domain size) that it





(a) (b)

Figure 1: (a) The number of constraint checks with Min Domain/Degree on 100 random problems in <30, 8., 0.26, 0.34> for FCR propagation with r = .1, .2…, .9. (b) Running time average and standard deviation for these runs. Note that the best time appears to be for r = 0.3.

```
For initialization method m in {one-pass AC, AC}
   f ← fastest method among {FC, FCR, FCRD}
   a ← fastest method among {AC, ACR, ACRD}
   Accelerate f and a by late FC
   Accelerate f and a by early FC
   Select b(m), the faster of FC-f-FC and FC-a-FC
Select the faster of b(one-pass AC) and b(AC)
```

*Figure 2:*ACE's high-level algorithm for learning a propagation policy.

accomplishes as it does so. This data is stored by search depth. The same problems are all reused at every stage in the process described here. One method was judged superior to another if it solved more problems (occasional geometric and quasigroup problems went unsolved in the 1000-second time limit under some propagation methods), or if it had an initialization plus search time that was statistically significantly better, or if it had a lower median time, or if it had a lower average time, in that order. (Because poor propagation policies often produced highly skewed distributions of performance, we emphasize, and report, median times here.) In any tie, the method simpler to compute was preferred.

ACE tests FCR on the problems and attempts to accelerate it. A higher r value results in less propagation from FCR or ACR. ACE begins with $r = 1/m$ and increases r by $1/m$ and retests on the 100 problems as long as there is no statistically significant increase in time to solution. (Recall that m is the maximum domain size.) ACE also tests FCRD using the data by search depth already collected. (Parameters are not changed for the D methods.) The best among FC, FCR with the best observed r, and FCRD becomes the foundation method f for propagation. Then the entire process is repeated, beginning this time with AC, and resulting in a second foundation method a. (To make ACR's queue more selective than AC's, however, r begins at $2/m$ instead of $1/m$.) For example, in a learning run on 100 simple random problems, ACE found f = FCR with $r = 0.25$ and a = AC.

Then ACE reruns f and a with the increased overhead of monitoring for the point at which the graphs become acyclic. ACE then turns off the acyclic computation and tests f-FC and a-FC. (The intuition here is that a late-enough terminal switch to FC should be relatively safe, even without the width-one order.) First ACE tests a terminal switch s_t that is the minimum of the greatest search depth at which any domain reduction occurred and the greatest search depth at which any problem became acyclic in the 100 problems. As long as there is no statistically significant increase in time to solution, ACE continues to reduce s_t by 1. At this point the foundation methods are of the form f-FC and a-FC (unless late switching reduced performance or a base method was FC already). In our example, the foundation methods were now f = FCR with $r = 0.25$ and a = AC-FC with $s_t = 25$.

Next, unless a base method is FC, ACE fixes any terminal switch s_t and tests FC-f-FC and FC-a-FC, beginning with $s_i = 1$ and increasing the initial switch until there is a statistically significant increase in time to solution. (If the two switches for FC-x-FC converge to the same value, ACE reverts to method x.) In our example, f became FC-FCR with $r = 0.25$ and $s_i = 2$, while a became FC-AC-FC with $s_i = 2$ and $s_t = 25$. Then

ACE compares the times for *f* and *a*, and chooses the more effective propagation method, in this case *f*.

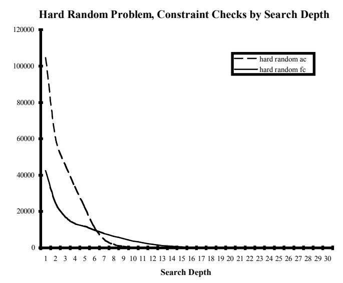
ACE runs this entire procedure, from foundation methods on, twice: once with one-pass AC initialization and again with AC initialization. It thereby learns a propagation policy with an initialization. The example above was for one-pass AC initialization on the simple problems; with AC initialization, ACE found f = FC-FCR with $r = 0.25$ and $s_i = 2$, and a = FC-AC-FC with $s_i = 2$ and $s_t = 25$.. Ultimately, ACE preferred the latter.

4 Results with learning

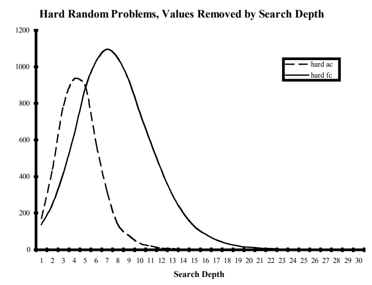
We had ACE learn to propagate for each of the classes described in Section 2. The results appear in Table 1. Note that the propagation policy learned does indeed vary by class. Of course, it is necessary to confirm these results on a separate set of data.

*Table 1:*Best propagation policies learned by ACE on 100 problems, based on initialization plus search time. Integer parameters are switch points; decimal parameters are *r* values. Times in seconds (mean μ, median md, and std deviation σ) are shown for a second set of problems in the same class: for FC (with one-pass AC initialization), for AC (with AC initialization), and for ACE's learned policy. Improvement is time reduction by ACE over each of FC and AC.

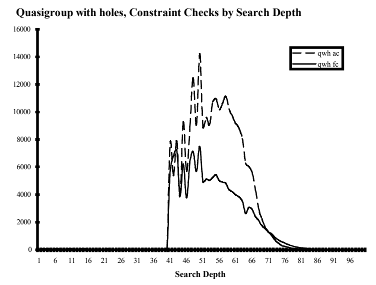
			Times			**Improvement**	
Class	**ACE learns**		**FC**	**AC**	**ACE**	**FC**	**AC**
Sparse	ACR-FC 0.75,23	μ	0.05	0.05	0.05	Same	Same
random	AC initialization	md	0.05	0.05	0.05	Same	
		σ	0.01	0.00	0.02		
Simple	ACR-FC 0.25, 23	μ	0.25	0.12	0.10	60%	17%
random	AC initialization	md	0.12	0.09	0.08	33%	11%
		σ	0.50	0.06	0.09		
Medium	ACR 0.25	μ	0.32	0.17	0.14	56%	18%
random	one-pass AC	md	0.24	0.14	0.12	50%	14%
	initialization	σ	0.28	0.27	0.06		
Hard	ACR 0.25	μ	1.52	0.85	0.70	54%	18%
random	one-pass AC	md	1.08	0.63	0.53	51%	16%
	initialization	σ	1.29	0.65	0.55		
Coloring	FCR-FC 0.5, 28	μ	0.45	0.43	0.47	-4%	–9%
		md	0.25	0.34	0.27	-8%	21%
		σ	0.77	0.24	1.00		
Geometric	ACR-FC 0.4 45	μ	6.41	6.69	6.38	0.4%	5%
	AC initialization	md	0.74	0.76	0.76	–3%	Same
		σ	22.47	28.81	22.40		
Quasigroups	AC-FC 93	μ	16.61	6.65	6.55	60%	2%
with holes	AC initialization	md	1.45	0.99	0.94	32%	5%
		σ	54.80	21.49	20.21		



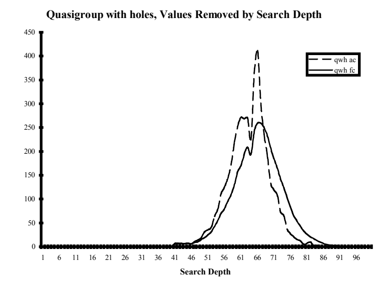
(a)



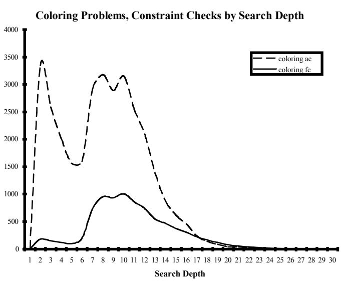
(b)



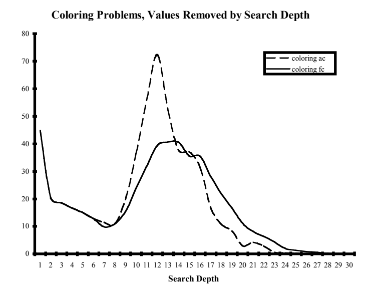
(c)



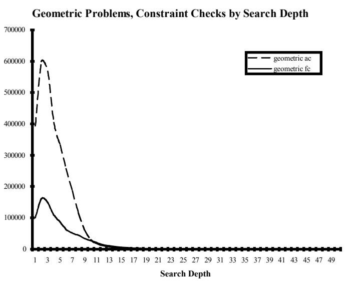
(d)



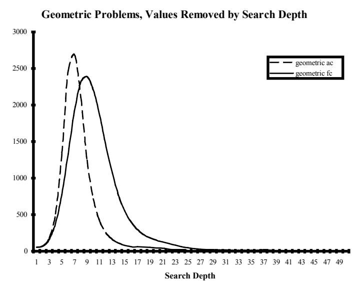
(e)



(f)



(g)



(h)

Figure 3: Constraint checks performed at each search depth (a, c, e, g) and values removed (b, d, f, h) by a traditional solver on 100 solvable random hard problems, quasigroups with holes, coloring problems, and geometric problems, respectively. An AC initialization pass was performed on each problem.

We ran Min Domain/Degree three times on a second, fresh set of 100 problems in each class: with ACE's learned propagation policy, with FC and one-pass AC initialization, and with AC and AC initialization. For every class of problems, our learned policies were at least as good as the others; for all random problem classes but sparse, ACE was statistically significantly better than FC at the 95% confidence level.

5 Further assessment of learning

In the previous section we have shown that the propagation policy ACE learns improves search performance on several different classes of CSPs. It is reasonable to ask whether this was the best propagation policy learnable from these building blocks, and whether most any policy would have sufficed. This section addresses those questions with additional data. We begin with three examples that compare the propagation activity and performance time of FC and AC.

• On random problems in <20, 30, 0.444, 0.5>, Min Domain/Degree averaged 29.21 seconds under FC to find a solution, but 82.14 seconds under AC.

• On hard random problems, the solver under FC does considerably less work (as measured in constraint checks) and removes more values (during compensation for its errors) somewhat later in search than under AC, as shown in Figures 3(a) and (b). Nonetheless, solution under FC is actually slower on these problems, presumably because FC leaves more unsupportable values which the solver cannot readily avoid.

• On quasigroups of order 10 with 60 holes, under FC the solver does less work and removes fewer values than AC. See Figures 3(c) and (d).

Table 2 confirms the differences between FC and AC on our problem classes, and that comparing them is well worth the effort. (In this table only, initialization time is excluded, to focus on work done after it; times are means, to show statistical significance.) If problems are easy because most initial value selections are consistent with some solution, then they probably do not require the intense scrutiny of AC, particularly if we seek only one solution. AC indeed does more work (as measured by constraint checks), but can significantly speed solution, depending on the problem class.

Table 2: Mean propagation time, checks and nodes expanded, exclusive of AC initialization, to solve with FC or AC and Min Domain/Degree on different problem classes. Results are averaged over 100 problems. Figures in bold represent a statistically significantly difference at the 95% confidence level.

	Time		**Checks**		**Nodes**	
Class	**FC**	**AC**	**FC**	**AC**	**FC**	**AC**
Sparse	0.04	0.05	**347.68**	1804.20	34.99	**30.18**
Simple	0.20	**0.12**	**1647.85**	5425.75	254.21	**38.43**
Medium	0.36	**0.17**	**3401.41**	9070.71	477.22	**55.14**
Hard	1.70	**0.85**	20416.20	51598.53	1879.87	**189.43**
Coloring	0.32	0.43	**2686.37**	17941.30	169.88	**73.36**
Geometric	7.37	6.69	74428.00	293088.37	2223.95	**385.46**
Quasigroups	16.61	6.65	26123.90	35392.58	3376.16	1196.96

Among our problem classes, FC appears to be a viable alternative only on sparse and coloring problems. Otherwise, FC's fewer checks come at the expense of visiting more nodes. Moreover, in these experiments AC initialization rarely removed any more values than one-pass AC initialization — at most two values in 100 problems.

Finding a good propagation policy by hand is not trivial. We tested FC, AC, and the propagation methods of Section 3 on the hard random problems, using AC initialization and Min Domain/Degree. We tested FC-spread and AC-bound for $p = 2, 3, \ldots, n/2$, and FCR and ACR for $r = .1, .2, \ldots, .9$. For x-FC methods we tested terminal switch $s_t = 5, 10, \ldots, n-5$. We observed that immediately upon any initial switch, there is a pronounced spike in the number of constraint checks, often well beyond what AC would have done, as the first AC pass catches up. We therefore tested FC-x methods only for initial switches $s_i = 1, 2, \ldots, 5$. Often a single parameter change (e.g., from $r = 0.5$ to 0.4) made it impossible to solve some problems that had been solved under the previous setting. Under many parameter settings, the solver spent hours on a single problem and we terminated the run.

ACE is learning in a space of methods that have the potential to perform quite poorly. Nonetheless, ACE found a very good propagation policy for each class. We tested these "best observed" parameter settings on the testing problems from Table 1, to see how they compared with ACE. Inspection indicates that ACE's learning is consistent with these results. For example, ACE learns $r = 0.25$ for the hard random problems for both FCR and ACR, as close as it can get to 0.3 with its algorithm.

Finally, as observed earlier, random problems lack reliable structure, which real-world problems generally have. Figure 3 suggests that a good propagation method might vary with problem class. We tested coloring, geometric, and quasigroup with holes problems empirically, using AC initialization and various propagation methods described above, beginning with parameters for the hard random problems and then choosing a few new values to test based on those results. The best observed parameters that produced them appear in Table 3, retested on the problems of Table 1. ACE

Table 3: Observed median solution time in seconds on 100 problems for three problem classes, along with the parameter values that produced them. The classes and the range of parameter values tested are detailed in the text. Min Domain/Degree and AC initialization were used.

	Hard		**Geometric**		**Coloring**		**Quasigroup**	
Propagation	**Time**	**Pars.**	**Time**	**Pars.**	**Time**	**Pars.**	**Time**	**Pars.**
FC	1.20	—	0.78	—	0.26	—	1.45	—
FC-spread	0.63	5	0.72	40	0.36	5	2.28	40
AC-bound	0.62	10	0.99	15	0.42	5	1.42	50
FCR	0.54	.3	0.67	.3	0.27	.5	2.21	.5
ACR	0.51	.3	0.66	.3	0.24	.5	1.42	.5
FC-AC	0.65	5	0.69	30	0.36	10	2.11	40
AC-FC	0.62	20	0.83	20	0.33	20	2.25	90
ACR-FC	1.09	5, .3	0.68	.3, 20	0.28	.2, 10	2.22	.5, 60
FC-AC-FC	0.65	5, 25	0.74	5, 30	0.28	5, 20	2.37	20,80
FC-ACR-FC	0.67	5,.3,25	0.75	3,.3,15	0.29	5,.2,25	1.34	40,.3,80
AC	0.92	—	0.76	—	0.34	—	0.99	—
ACE learned	0.53	ACR .25	0.76	ACR-FC .4, 45	0.27	FCR-FC .5, 28	0.94	AC-FC 93

came close to the best time for the hard problems and the coloring problems. Note that ACR and FCR consistently match or outperform the more traditional FC and AC.

6 Discussion and related work

It is noteworthy that AC initialization often, but not always, leads to improved performance. In some cases, of course, the nature of the problem class makes any reduction by AC unlikely (e.g., coloring). Otherwise, we surmise that much of the difficulty a search method experiences with a problem has to do with where to begin (once again, the backdoor), and that an initialization pass of either kind may offer a useful clue based on initially reduced domain size.

AC's automatic reconsideration of edges may be overkill. Propagation is effective only when it can quickly remove values that will not lead from the current instantiation to a solution. If the crucial potential inconsistencies lie nearby the current variable, then propagation need not explore every constraint. In Table 1, ACE learned ACR or FCR for every class, which suggests that the impact of propagation, as measured by the response *r*, may be a better indication of when to reconsider them.

As search deepens, dynamic domains become progressively smaller, so that eventually few values remain, and even AC removes few of them. In Figure 2(b), for example, this happens after assigning about one third of the values with AC, and after about two thirds with FC. A search method that prefers maximum degree will focus first on highly-connected variables; eventually the future variables will be connected to few others, and again are likely to have little impact beyond their immediate neighbors. This would argue for propagation methods that address response when the search method includes minimizing domain size, and explains to some extent our success here with R methods. Because Min Domain/Degree is responsive both to dynamic domain size and to degree, it supported our new methods particularly well.

ACE's algorithm to learn a propagation policy performs as well as any manually selected settings. Differences arise when the crucial *r* values tested by ACE (in increments of $1/m$) do not match those tested empirically (in increments of 0.1), or when its switch values (tested in increments of 1) step more gradually than those tested empirically (in increments of 5). Inspection indicates that despite its host of building blocks, ACE learns $r = 0.25$ for FCR on the hard random problems, as close as it can get to the 0.3 that performed best on those problems in Figure 2. (Ultimately, however, ACE judged one-pass AC initialization and ACR $r = 0.25$ to be better.) The learning algorithm eliminates much tedious lengthy testing (and automates the rest).

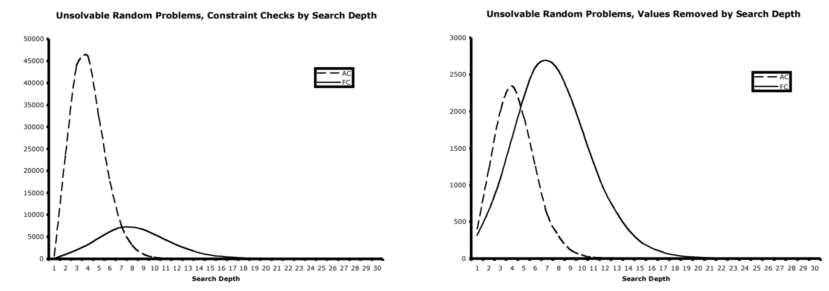
In the construction of this algorithm we explored and then eliminated many possible approaches. Based on observations of monotonicity during the extensive testing that led to Tables 3 and 4, we assumed that performance associated with response *r* has a single minimum. Based on their lackluster performance during initial testing, FC-spread and AC-bound were excluded from the process. (Nonetheless, a real-world problem could in principle be most affected by variables in the immediate vicinity of the current variable, and we expect to investigate these variants further.) One might also argue that value removals ought to be compared with tightness. Since the probability that a pair of values is unacceptable on an edge is roughly the square root of the tightness, a static approach should therefore be commensurate with $t^{1/2}$. While it is unlikely that a method will achieve such reductions consistently, in a state with f future variables and an average dynamic domain size g, one could hope for $t^{1/2}fg$ removals and continue to propagate with AC as long as r% of that gauge was removed. This method, however, is equivalent to AC-bound with an appropriately-scaled parameter. (See, however, (Mehta and van Dongen 2005).) One might also monitor removals per check, a sort of utility heuristic, that would select the method that removes the most values for the work it performs. By this standard, however, the best of the FCR methods on the hard problems would have been FC, which we know to have been unacceptably slow there. Finally, we coded and observed an algorithm that cycled between AC and FC at various intervals. The spikes we noted for the early FC switch reappeared and proved too costly, however.

Some propagation methods appear rarely if at all in Table 1. Inspection indicates that the D (adjust by search depth) methods performed relatively well. In most problem classes total domain size drops by 16-23% after the first assignment. This is not true of the random hard problems, however, and only geometric problems have another significant drop after the second assignment. Further work on the D methods is planned. It also appears that switching is not helpful with R methods. This suggests that a substantial reduction in domain size remains important throughout search. When a terminal switch was constructive, it led to some reduction in median time: during learning, about 12% on geometric and quasigroup problems. An initial switch, although it may have improved AC in Table 3, was never part of a best observed or learned propagation policy.

Because the focus of this work is propagation, we used equivalent search methods throughout. It is reasonable to expect, however, that the performance of the search method and the propagation policy are intertwined. We therefore had ACE learn a propagation policy for coloring problems under two other variable-ordering heuristics: Min Domain and the *Brélaz heuristic* (minimize the dynamic domain size and break

Table 4: Propagation policies ACE learned for coloring problems. Decimal parameters are r value. Times in seconds (mean μ, median md, and standard deviation σ) are shown for a second set of problems in the same class: for FC (with one-pass AC initialization), for AC (with AC initialization), and for ACE's learned policy. Improvement is for ACE over each of FC and AC.

Search			**Times**			**Improvement**	
method	**ACE learns**		**FC**	**AC**	**ACE**	**FC**	**AC**
Min	FCR-FC 0.5, 28	μ	0.45	0.43	0.47	-4%	–9%
Domain/Degree		md	0.25	0.34	0.27	-8%	21%
		σ	0.77	0.24	1.00		
Brélaz	ACR-FC 0.5, 25	μ	0.45	0.44	0.43	4%	4%
		md	0.27	0.35	0.27	0%	23%
		σ	0.56	0.33	0.57		
Min Domain	ACR 0.625	μ	1.40	1.38	0.75	46%	46%
		md	0.50	0.48	0.32	36%	33%
		σ	3.48	3.45	1.43		



(a) (b)

Figure 4: (a) Constraint checks performed at each search depth and (b) values removed by a traditional solver on 100 unsolvable problems with 30 variables, domain size 8, density 0.26, and tightness 0.34.

ties with maximum forward degree) (Brélaz 1979). Both select values lexically. Min Domain is an inferior search method for these problems, and Brélaz is known to be superior to Min Domain/Degree on coloring problems. Table 4 compares the results. ACE learned a different propagation policy for each search method method. For Min Domain the learned policy was able to compensate, to some degree, for the poor search method, cutting search time nearly by half.

Learning a propagation policy is now part of ACE's framework for learning to solve CSPs, but several intriguing research issues remain. We have not yet addressed whether the propagation policy ACE learns to find the first solution is equally good when seeking all solutions or when working with unsolvable problems. We generated a separate set of unsolvable problems in <30, 8, 0.26, 0.34> and redrew diagrams like those of Figure 2(a) and (b) for them in Figure 4. Comparing them, the values removed curves are similar, but the constraint checks are not. Learning a propagation policy is not limited to binary constraints; it should be of value with any specialized propagation methods (e.g., all-diff or rank sum). Additional speedup should be available through queue management. The impact of a value-selection heuristic on this process is also unknown. An algorithm to learn a propagation policy might be based upon checks and/or nodes as well as time. Finally, one might wonder to what extent our results are dependent upon ACE, rather than upon the problems themselves. To this we reply that every implementation has aspects that are done more or less efficiently. This paper demonstrates that AC may be more work than is necessary, that response (rather than locality) seems to be key, and that early and late FC are often useful as well. We therefore encourage others to have their solvers learn their own, possibly implementation-dependent balance between AC and FC, confident that learning such a propagation policy offers clear benefits within a problem class.

Acknowledgments

We thank Barbara Smith for her thoughtful questions and ideas. This work was supported in part by NSF IIS-0328743, by PSC-CUNY, by Enterprise Ireland under Grant No. SC/2002/0137, and is based upon works supported in part by Science Foundation Ireland under Grant 00/PI.1/C075.

References

Achlioptas, D., C. Gomes, H. Kautz and B. Selman (2000). Generating Satisfiable Problem Instances. *AAAI-00*.

Bessière, C., E. C. Freuder and J.-C. Régin (1999). "Using constraint metaknowledge to reduce arc consistency computation." *Artificial Intelligence* **107**(125-148).

Bessière, C. and J.-C. Régin (1996). MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems. *Principles and Practice of Constraint Programming - CP96*, Springer-Verlag.

Bessière, C. and J.-C. Régin (2001). "Refining the basic constraint propagation algorithm." *JFPLC*: 1-13.

Brélaz, D. (1979). "New Methods to Color the Vertices of a Graph." *CACM* **22**: 251-256.

Chmeiss, A. and L. Sais (2004). Constraint satisfaction problems: Backtrack search revisited. Sixteenth International Conference on Tools with Artificial Intelligence (ICTAI'04). IEEE

Epstein, S. L. and E. C. Freuder (2001). Collaborative Learning for Constraint Solving. *Principles and Practice of Constraint Programming - CP 2001*, Springer-Verlag.

Epstein, S. L., E. C. Freuder, R. Wallace, A. Morozov and B. Samuels (2002). The Adaptive Constraint Engine. *Principles and Practice of Constraint Programming -- CP2002*. P. Van Hentenryck. Berlin, Springer Verlag. **LNCS 2470:** 525-540.

Freuder, E. C. (1982). "A Sufficient Condition for Backtrack-Free Search." *JACM* **29**(1): 24-32.

Freuder, E. C. and R. J. Wallace (1991). Selective relaxation for constraint satisfaction problems. *Third International Conference on Tools for Artificial Intelligence (TAI'91)*, San Diego, CA.

Golumb, S. and L. Baumert (1965). "Backtrack programming." *Journal of the ACM* **12**: 516-524.

Haralick, R. M. and G. L. Elliott (1980). "Increasing tree search efficiency for constraint satisfaction problems." *Artificial Intelligence* **14**: 263-314.

Johnson, D. B., C. R. Aragon, L. A. McGeooh and C. Schevon (1989). "Optimization by Simulated Annealing: An experimental evaluation; Part 1, Graph partitioning." *Operations Research* **37**(865-892).

Lecoutre, C., F. Boussemart and F. Hemery (2003). Exploiting multidirectionality in coarse-grained arc consistency algorithms. *Principles and Practice of Constraint Programming - CP2003, LNCS 2833*, Springer Verlag.

Mehta, D. and M. R. C. van Dongen (2005). Reducing Checks and Revisions in Coarse-grained MAC Algorithms. *IJCAI-05*.

Ruan, Y., E. Horvitz and H. Kautz (2004). The Backdoor Key: A Path to Understanding Problem Hardness. *AAAI-2004*, San Jose, CA, AAAI Press.

Sabin, D. and E. C. Freuder (1994). Contradicting Conventional Wisdom in Constraint Satisfaction. *Eleventh European Conference on Artificial Intelligence*, Amsterdam, John Wiley & Sons.