M.R.C. van Dongen (Ed.)

Proceedings of the Second International Workshop on

Constraint Propagation And Implementation, Volume I

Contributed Papers

Sitges, Spain, October 2005

Held in conjunction with the
Eleventh International Conference on
Principles and Practice of
Constraint Programming (CP 2005)

Organisation

CPAI'2005 Organising Committee

Marc van Dongen	Cork Constraint Computation Centre, Ireland
Christophe Lecoutre	Université d'Artois, France
Rick Wallace	Cork Constraint Computation Centre, Ireland
Yuanlin Zhang	Texas Tech University, USA

CPAI'2005 Programme Committee

Frédéric Boussemart	Université d'Artois, France
Fred Hemery	Université d'Artois, France
Christophe Lecoutre	Université d'Artois, France
Peter van Beek	University of Waterloo, Canada
Marc van Dongen	Cork Constraint Computation Centre, Ireland
Pascal Van Hentenryck	Brown University, USA
Willem-Jan van Hoeve	Cornell University, USA
Rick Wallace	Cork Constraint Computation Centre, Ireland
Roland Yap	National University of Singapore, Singapore
Yuanlin Zhang	Texas Tech University, USA

CPAI'2005 Competition Organising Committee

Frédéric Boussemart	Université d'Artois, France
Fred Hemery	Université d'Artois, France
Mark Hennessy	Cork Constraint Computation Centre, Ireland
Deepak Mehta	Cork Constraint Computation Centre, Ireland
Christophe Lecoutre	Université d'Artois, France
Radoslaw Szymanek	Cork Constraint Computation Centre, Ireland
Marc van Dongen	Cork Constraint Computation Centre, Ireland
Rick Wallace	Cork Constraint Computation Centre, Ireland
Yuanlin Zhang	Texas Tech University, USA

Preface

Constraint Propagation is an essential part of many constraint programming systems. Sitting at the heart of a constraint solver, it consumes a significant portion of the time that is required for problem solving.

The *Second International Conference on Constraint Propagation and Implementation* (CPAI'2005) was convened to study the design and analysis of new propagation algorithms as well as practical issues in and the evaluation of implementing existing and new constraint propagation algorithms in settings ranging from special purpose solvers to programming language systems.

The CPAI'2005 workshop proceedings are divided into two volumes. This is Volume I of the proceedings. It is dedicated to the first part of the workshop: a "regular-style" workshop. It includes eight contributed papers. Volume II is dedicated to the second part of the workshop: the First International CSP Solver Competition.

The organisors wish to thank all authors for submitting their work, all participants of the solver competition for entering their solver, the invited speakers, Christian Schulte and Laurent Simon, the CP'2005 Workshop/Tutorial Chairs, Alan Frish and Ian Miguel, the members of the CPAI'2005 Programme Committee, and the members of the CPAI'2005 Competition Organising Committee. They wish to express their gratitude to Gene Freuder for providing support to the competition in the form of computing power and system administrator's time. Finally, they wish to thank Peter MacHale for his technical support of the solver competition.

Marc van Dongen
Christophe Lecoutre
Rick Wallace
Yuanlin Zhang

September 2005

Table of Contents

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, $\langle X, D, C \rangle$, 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.

• 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 $p = n\text{-}1$ 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

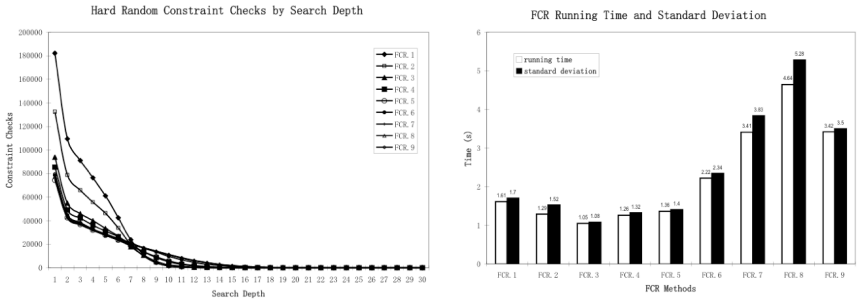
less work than AC itself, and do substantially less work (as does AC) after some point, here when about 9 variables have been bound. We tested FCR for r = .1, .2,…,.9 on a set of 100 problems. We therefore investigate propagation methods of the form x-FC with *terminal switch* s_t, where x is itself 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 accomplishes as it does so. This data is stored by search depth. The same problems



(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.

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		

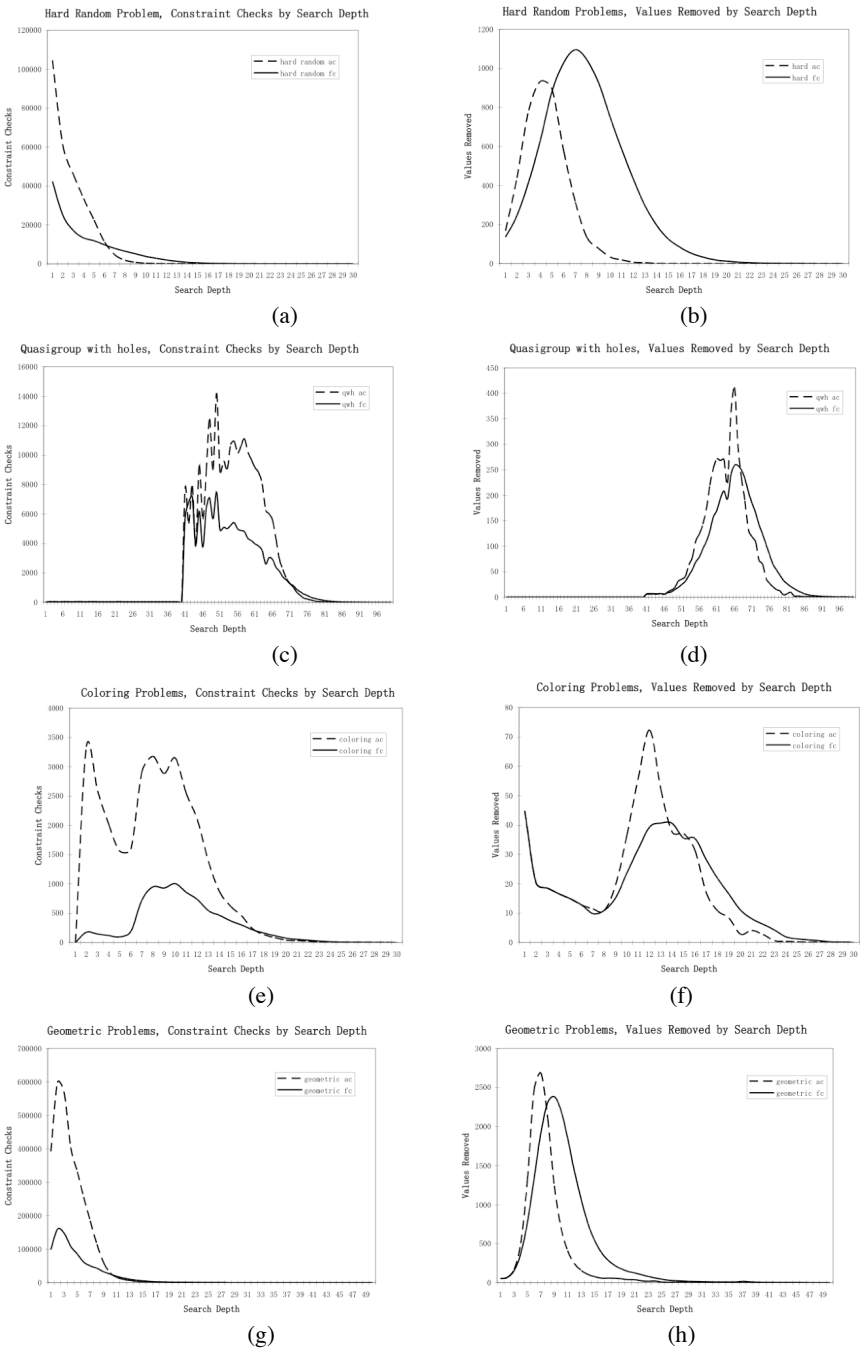


Figure 3: Constraint checks performed at each search depth (a, c, e, g) and values removed (b, d, f, h) by a traditional solver on 100 solvable random hard problems, quasigroups with holes, coloring problems, and geometric problems, respectively. An AC initialization pass was performed on each problem.

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. Among our problem classes, FC appears to be a viable alternative only on sparse and

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

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,…, n/2, and FCR and ACR for r = .1,.2,…,.9. For x-FC methods we tested terminal switch s_t = 5, 10, …, 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,…, 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 came close to the best time for the hard problems and the coloring problems. Note

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

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

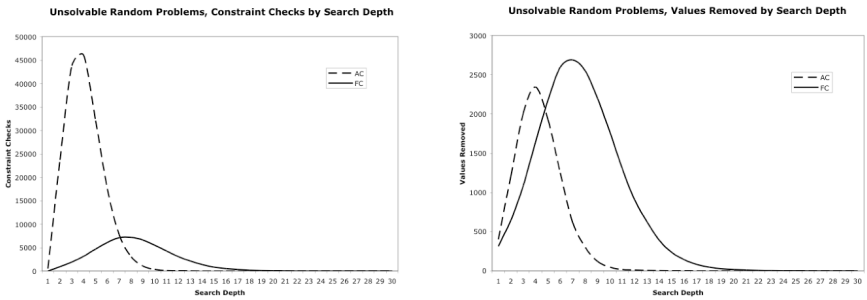
unlikely that a method will achieve such reductions consistently, in a state with f future variables and an average dynamic domain size g, one could hope for $t^{1/2}fg$ removals and continue to propagate with AC as long as r% of that gauge was removed. This method, however, is equivalent to AC-bound 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 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 re-

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.

sults. 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.

Structure and Problem Hardness: Asymmetry and DPLL Proofs in SAT-Based Planning

Jörg Hoffmann[1], Carla Gomes[2], and Bart Selman[2]

[1] Max-Planck-Institute for CS, Saarbrücken, Germany
[2] Cornell University, Ithaca, NY, USA

Abstract. In applications from AI Planning and Model-Checking, a successful method is to compile the application task into boolean satisfiability (SAT), and solve it with state-of-the-art DPLL-based procedures. There is a lack of formal understanding why this works so well. Focussing on the AI Planning context, we identify a structural parameter, called *AsymRatio*, that measures a kind of subgoal asymmetry in planning tasks. *AsymRatio* ranges between 0 and 1, and we show empirically that it correlates strongly with SAT solver performance in a broad range of AI Planning benchmarks, namely the domains used in the 3rd International Planning Competition. We then examine carefully crafted *synthetic planning domains* that allow to control the value of *AsymRatio*, and that are clean enough to allow a rigorous analysis of the combinatorial search space, while meaningful enough to allow conclusions about more practical domains. The domains are parameterized by size n, and by a structure parameter k, so that *AsymRatio* is asymptotic to k/n. We investigate the best (smallest) possible sets of branching variables for DPLL, as a function of n, for different settings of k. With minimum k, we identify minimal sets of branching variables linear in the total number of variables, $\Theta(n^2)$. With maximum k, we identify sets of size $O(log_2 n)$, and thus size $O(n)$ DPLL proofs.

1 Introduction

There has been a long interest in a better understanding of what makes combinatorial problems hard or easy. The most successful work in this area involves random instance distributions with phase transition characterizations (e.g., [1, 2]). However, the link of these results to more *structured* instances is less direct. A random unsatisfiable 3-SAT instance from the phase transition region with 1,000 variables is beyond the reach of any current solver. But many unsatisfiable formulas from verification and planning contain well over 100,000 variables and can be proved to be unsatisfiable within a few minutes (e.g., with Chaff [3]). This raises the question as to whether one can obtain general measures of *structure* in SAT encodings, and use them to characterize typical case complexity. To this end, our overall goal in this paper is to identify general problem features that characterize problem hardness in practice. We focus on formulas from AI planning. We view this as an entry point to similar studies in other areas.

The main spirit of our work is a two-step approach: first, identify a measure of "structure" that, empirically, correlates with CSP/SAT solver performance in practical benchmarks; then, design synthetic domains that capture this structure

in a clean form, and analyze the behavior of DPLL (or any other search algorithm of interest), within these synthetic domains, in detail. The latter step serves to obtain a deeper understanding of what causes the empirical correlation observed in the first step. For this to make sense, the synthetic domains have to be simple enough to be rigorously analyzed, yet meaningful enough to allow conclusions about more practical domains. We remark that, while under development, the two research steps may well be – and have been, in our case – intermingled: increasingly accurate intuitions are obtained in a trial-and-error fashion.

Note that our approach is very different from identifying tractable classes. Generally, our research is aimed at understanding the behavior of existing algorithms, not at identifying new algorithms. More technically, the first research step outlined above establishes an *empirical* correlation between structure and performance. The second research step may, or may not, yield results on polynomial best-case or worst-case behavior. But even if so, these results hold only for the specific *examples* (synthetic domains) considered. In that sense, the analytical step is merely a *case-study*, aimed at obtaining more accurate intuitions.

We focus on showing infeasibility. Precisely, we consider the difficulty of showing the non-existence of a plan with one step less than the shortest possible (*optimal*) plan. SAT-based search for a plan works by iteratively incrementing a plan length bound b, and testing in each iteration a formula that is satisfiable iff there exists a plan with b steps (this was first implemented in the Blackbox system [4]). So, our focus is on the last unsuccessful iteration in a SAT-based plan search. This is typically the hardest iteration in practice. SAT-based planning is currently state-of-the-art for finding optimal plans: e.g., Blackbox won the 1st prize for optimal planners in the 4th International Planning Competition [5].

We consider SAT encodings of our synthetic domains and investigate the best possible sets of branching variables for DPLL proof trees. Such variable sets were recently coined "backdoors" [6]. In our context, a backdoor is a *subset of the variables so that, for every value assignment to these variables, unit propagation (UP) yields an empty clause.*[3] That is, a smallest possible backdoor encapsulates the best possible branching variables for DPLL, a question of huge practical interest. Also, the size of the backdoor provides an upper bound on the size of the DPLL search tree: if the backdoor contains l variables, then the maximum number of nodes in the proof tree is 2^l. In particular, if l is logarithmic in the formula size, then there exists a polynomial size DPLL proof. In all considered formula classes, we determine a backdoor subset of variables. We prove that the backdoors are *minimal*: no variable can be removed without losing the backdoor property. In small enough instances, we prove empirically that the backdoors are in fact *optimal* - of minimal *size*. We conjecture that the latter is true in general.

Our synthetic planning domains are (1) a logistics planning domain (*MAP*) and (2) a stacking domain (*SBW*). We also consider a third synthetic domain

[3] In general, a backdoor is defined relative to an arbitrary polynomial time "subsolver" procedure. The subsolver can solve some class of formulas that does not necessarily have a syntactic characterization. Our definition here instantiates the subsolver with the widely used unit propagation procedure.

called *SPH*, a structured version of the Pigeon Hole problem. The domains are characterized by a size parameter, called n, and by a structure parameter, called k. The structure parameter controls the amount of an intuitive "asymmetry" in the underlying task: as the value of k increases, one part of the task becomes more and more difficult to achieve, while the other parts become relatively easier. Concretely, in Planning, we define the parameter *AsymRatio* as the ratio between maximum sub-goal difficulty – the maximum number of steps needed to achieve any single sub-goal – and the overall difficulty, i.e., the number of steps needed to achieve the conjunction of *all* goals. *AsymRatio* ranges between 0 and 1. A value close to 1 represents a large structural asymmetry. In MAP and SBW, *AsymRatio* is asymptotic to the ratio between k and n. In particular, for the lowest value of k (symmetrical case), *AsymRatio* converges to 0 for increasing n; for the highest k value (asymmetrical case), the ratio converges to 1. Note that such a high amount of asymmetry appears unlikely to occur in purely randomly generated problem instances. Note further that *AsymRatio* characterizes a kind of *hidden* structure. It can not be computed efficiently even based on the original planning task representation. Much less is it evident to a satisfiability tester attacking the CNF representation of the task, without even knowing that the formula originates from a planning task. More concretely, the constraint graphs of our formulas (more on this below) generally don't change much over k.

In some initial experiments, we observed that a high value of *AsymRatio* enables the (unsatisfiable) formulas to be effectively solved by current SAT solvers. Investigating this in our synthetic domains, we found dramatic differences in backdoor size. At the bottom ends of the k scales, with symmetrical subgoals, the backdoor sets are of polynomial size (in n) in all cases. With increasing value of k, the backdoors become smaller; in the two synthetic planning domains, at the top end of the k scales, the backdoors are of *logarithmic size*.[4]

To confirm that *AsymRatio* correlates with SAT solver performance in practice (i.e., in more complex benchmarks than our synthetic domains), we ran large-scale experiments in the six benchmark domains used in the 3rd International Planning Competition [7]. This is a recent (published in 2002) and widely used set of benchmarks, and is provided, by the IPC-3 organizers, with instance generators; the latter are essential for our experiments, where we generated and examined tens of thousands of instances in each domain.[5] We plotted the performance of a state-of-the-art SAT solver, namely, ZChaff [3], as a function of *AsymRatio*. Our experiments show that a larger *AsymRatio* results in planning CNFs that are significantly easier to solve. *AsymRatio* thus provides a useful indicator of typical problem hardness for Planning domains.[6] This is of course

[4] It is important to note that we obtain logarithmic size backdoors. This suggests that our underlying planning problems do not become "trivial" — in particular, they still require some subtle branching choices of the DPLL procedure, and are not just solved by unit propagation.

[5] For the domains used in the 4th International Planning Competition [8], run in 2004, there are no random generators.

[6] While *AsymRatio* can not be computed efficiently, there exists a variety of techniques to approximate the number of steps needed to achieve a goal (e.g., [9–11]).

just a first example of a hardness measure for structured problems; presumably, other useful measures exist.

The investigation of structure in constraint reasoning problems is not new, see for example [12–20]. However, to the best of our knowledge, our particular approach – to empirically identify a relevant structural parameter and then analyze that in synthetic domains – has not been pursued before. Still, one structural concept is particularly closely related to the concept of a backdoor, and should be discussed in more detail: *cutsets* (e.g. [12–14]). A cutset is a set of variables so that, once these variables are removed from the constraint graph – the undirected graph where nodes are variables and edges indicate common membership in at least one clause – that graph has a property that enables efficient reasoning: an *induced width* of at most a constant bound b (if $b = 1$ then the graph is cycle-free, i.e., can be viewed as a tree). Backdoors are a generalization of cutsets in the sense that any cutset is a backdoor relative to an appropriate subsolver (that exploits properties of the constraint graph).[7] The main *difference* between a backdoor and a cutset is, from a general point of view, that, given a set of variables, one can determine in polynomial time whether or nor that set is a cutset. The same test is, in general, not possible for a backdoor. In particular, it is not possible for the unit propagation procedure we consider here, that depends heavily on *what values* are assigned to the backdoor variables. In that sense, a cutset is a backdoor that can be detected statically, and that can thus be directly exploited in a search algorithm. Backdoors in general only provide a parameter measuring properties of search spaces. We will see that, in the particular formula families we consider here, there are no small statically detectable cutsets. Indeed, as we detail below, the constraint graphs do generally not change much with k and are thus not suitable to capture what happens on the structural scale. In that sense, the structure in our formulas is "hidden".

In Section 2, we provide background on AI planning and the SAT encodings we use. In Section 3, we present our empirical findings showing the relevance of *AsymRatio* as a measure of problem hardness in structured domains. In Section 4, we describe our synthetic domains and our analysis of backdoors. Section 5 provides a summary of results and directions for future research.

2 Background

We consider the "STRIPS" formalism. States are described as sets of (the currently true) propositional facts. A planning *task* is a tuple of *initial state* (a set of facts), *goal* (also a set of facts), and a set of *actions*. Actions a are fact set triples: the *precondition* $pre(a)$, the *add effect* $add(a)$, and the *delete effect* $del(a)$. The semantics are that an action is applicable to a state (only) if $pre(a)$ is contained in the state. When executing the action, the facts in $add(a)$ are included into the state, and the facts in $del(a)$ are removed from it (the intersection between

Such techniques can be used to approximate *AsymRatio*, and, with that, predict SAT solver performamnce in Planning. Exploring this is a topic for future work.

[7] We thank Rina Dechter for insightful discussions on this issue.

$add(a)$ and $del(a)$ is assumed empty; executing a non-applicable action results in an undefined state). A *plan* for the task is a sequence of actions that, when executed iteratively, maps the initial state into a state that contains the goal.

Planning can be mapped into a sequence of SAT problems, by incrementally increasing a plan length bound b: start with $b = 0$; generate a CNF $\phi(b)$ that is satisfiable iff there is a plan with b steps; if $\phi(b)$ is satisfiable, stop; else, increment b and iterate. This process was first implemented in the Blackbox system [4].

There are, of course, different ways of generating the formulas $\phi(b)$, i.e., there are different *encoding methods*. In our empirical experiments, we use the original *Graphplan-based* encoding used in Blackbox. In our theoretical investigations, we use a somewhat simplified version of that encoding.

The Graphplan-based encoding is a straightforward translation of a b-step *planning graph* [9] into a CNF. The encoding has b *time steps* $1 \leq t \leq b$. It features variables for facts at time steps, and for actions at time steps. There are artificial *NOOP* actions, i.e. for each fact p there is an action *NOOP-p* whose only precondition is p, and whose only (add) effect is p. The NOOPs are treated just like normal actions in the encoding. Amongst others, there are clauses to ensure that all action preconditions are satisfied, that the goals are true in the last time step, and that no "mutex" actions are executed in the same time step.[8] The set of fact and action variables at each time step, as well as pairs of "mutex" facts and actions, are read off the planning graph (which is the result of a non-trivial propagation of constraints).

We do not describe the Graphplan-based encoding in detail since that is not necessary to understand our experiments. For the simplified encoding used in our theoretical investigations, some more details are in order. The encoding uses variables only for the actions, i.e., $a(t)$ is 1 iff action a is to be executed at time t, $1 \leq t \leq b$. A variable $a(t)$ is included in the CNF iff a is *present* at t. An action a is present at $t = 1$ iff a's precondition is true in the initial state; a is present at $t > 1$ iff, for every $p \in pre(a)$, at least one action a' is present at $t-1$ with $p \in add(a')$. For each action a present at a time t and for each $p \in pre(a)$, there is a *precondition* clause of the form $\neg a(t) \vee a_1(t-1) \vee \ldots \vee a_l(t-1)$, where $a_1, \ldots, a_l$ are all actions present at $t-1$ with $p \in add(a_i)$. For each goal fact $g \in G$, there is a *goal* clause $a_1(b) \vee \ldots \vee a_l(b)$, where $a_1, \ldots, a_l$ are all actions present at b that have $g \in add(a_i)$. Finally, for each *incompatible* pair a and a' of actions present at a time t, there is a *mutex* clause $\neg a(t) \vee \neg a'(t)$. Here, a pair a, a' of actions is called incompatible iff either both are not NOOPs, or a is a NOOP for fact p and $p \in del(a')$ (or vice versa).

We interprete CNF formulas as sets of clauses, where each clause is a set of literals. For a CNF formula ϕ with variable set V, a variable subset $B \subseteq V$, and a value assignment a to B, we say that a is *UP-consistent* if applying a to (the literals in) ϕ, and performing unit propagation on the resulting formula, does not yield an empty clause. B is a *backdoor* if it has no UP-consistent assignment.

[8] Actions can be executed in the same time step if their effects and preconditions are not contradictory.

3 Asymmetric Structure in Planning

As discussed above, we quantify subgoal asymmetry as follows.

Definition 1. *Let P be a planning task with goal G. For each fact $g \in G$, let $cost(g)$ denote the length of a shortest plan achieving just g; let $cost(G)$ denote the length of a shortest plan achieving all facts in G. The* asymmetry ratio *of P is:*

$$AsymRatio(P) := \frac{max_{g \in G} cost(g)}{cost(G)}$$

Note that $cost(G)$, in this definition, is the optimal plan length; to simplify notation, we will henceforth denote this with m. Note also that, of course, a definition as simple as Definition 1 can not be fail-safe. Imagine replacing G with a single goal g and an additional action with precondition G and add effect $\{g\}$: the (new) goal is then no longer a set of "subgoals". However, in the benchmark domans that are actually *used* by researchers to evaluate their algorithms, G is almost always composed of several goal facts, and the single goal facts correspond quite naturally to different sub-problems of the task.[9] Our hypothesis in the experiments is:

Hypothesis 1 *Let $\mathcal{P}_m$ be a set of planning tasks from the same domain with the same size parameter values, and with the same optimal plan length m. For $P \in \mathcal{P}_m$, let $\phi(P, m-1)$ denote the Graphplan-based CNF encoding of $m-1$ action steps. Then, over $\mathcal{P}_m$, the hardness of proving $\phi(P, m-1)$ unsolvable is strongly correlated with $AsymRatio(P)$.*

First, note that, certainly, whether this hypothesis holds or not *depends on the domain*; in that sense it is a different hypothesis for every domain. Second, note that the instance size parameter values (nr. of vehicles for transportation, e.g.), together with the number of action steps encoded – the optimal plan length minus 1 – determine the size of the formula. Of course, formula size is typically correlated with SAT solver performance. Our hypothesis concerns performance in formulas of *similar* size. Please note that we do *not* wish to imply that *AsymRatio* is "the" parameter predicting SAT solver performance in Planning CNFs. There are, presumably, many important factors and interplay between them. Our (only) observation, below, is that *AsymRatio* works well in an important range of domains.

To test our hypothesis, as said, we ran large experiments in all STRIPS domains used in the 3rd International Planning Competition [7] (*IPC-3*), as was carried out in 2002. The domains are called Depots, Driverlog, Freecell, Rovers, Satellite, and Zenotravel. Depots is a mixture between the classical Blocksworld and Logistics domains; Blocksworld requires arrangement of blocks in stacks on a

[9] A more stable approach would be to identify a hierarchy of layers of "landmarks" [21], and define *AsymRatio* based on that. In the benchmarks, because of what we just said, this does not seem to add much value. Exploring the issue in more depth is a topic for future work.

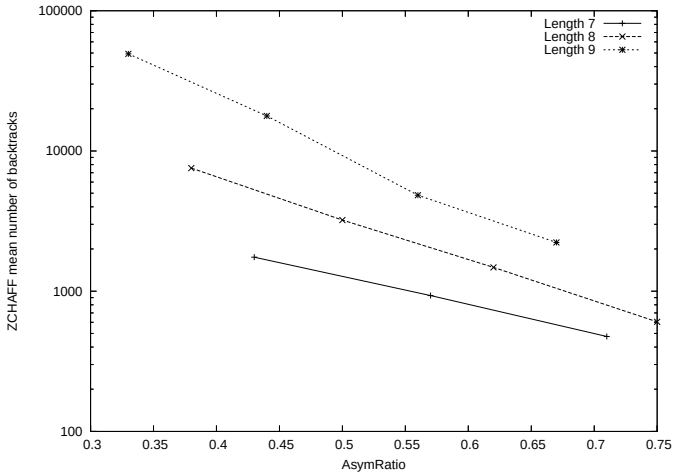


Fig. 1. Log-scaled mean number of backtracks needed by ZChaff, plotted over *AsymRatio*, in CNF formulas encoding planning instances from the IPC-3 benchmark Rovers. Curves for different subsets of more than 40000 randomly generated instances: all instances with optimal plan length 7, all instances with optimal plan length 8, and all instances with optimal plan length 9. Entire distribution of optimal plan length is 4 . . . 19; 7, 8, and 9 are the most frequent, and together contain 60% of all instances.

table, using a robot arm; Logistics requires transportation of packages via trucks and airplanes; in Depots, blocks must be transported and arranged in stacks. Driverlog is a version of Logistics with drivers, where drivers and trucks move on different (arbitrary) road maps. Freecell encodes the well-known solitaire card game where the task is to re-order a random arrangement of cards, following certain stacking rules, using a number of "free cells" for intermediate storage. Rovers and Satellite are simplistic encodings of NASA space-applications. In Rovers, rovers move along individual road maps, and have to gather data about rock or soil samples, take images, and transfer the data to a lander. In Satellite, satellites must take images of objects in space, which involves calibrating cameras, turning the right direction, etc. Zenotravel is a version of Logistics where moving a vehicle consumes fuel that can be re-plenished using a "refuel" operator. It is important to note that, *within each of the IPC-3 domains, deciding bounded plan existence — the problem encoded by our CNFs — is NP-hard* [22]. So our experiments are on challenging, if not real-world realistic, problems.

To obtain a reliable picture of how a complex DPLL-based SAT solver (ZChaff) typically behaves in CNF formulas generated from a domain, within each domain we generated and examined tens of thousands of instances. We chose the instance size parameters by testing the original IPC-3 instances, and selecting the largest one for which we could compute *AsymRatio* reasonably fast.[10] E.g. in Driverlog we selected the instance indexed 9 out of 20 (instance size here scales with growing index), and, accordingly, generated random instances with 5 road junctions, 2 drivers, 6 packages, and 3 trucks. According to the setup in

[10] That computation was done by a combination of calls to Blackbox [4].

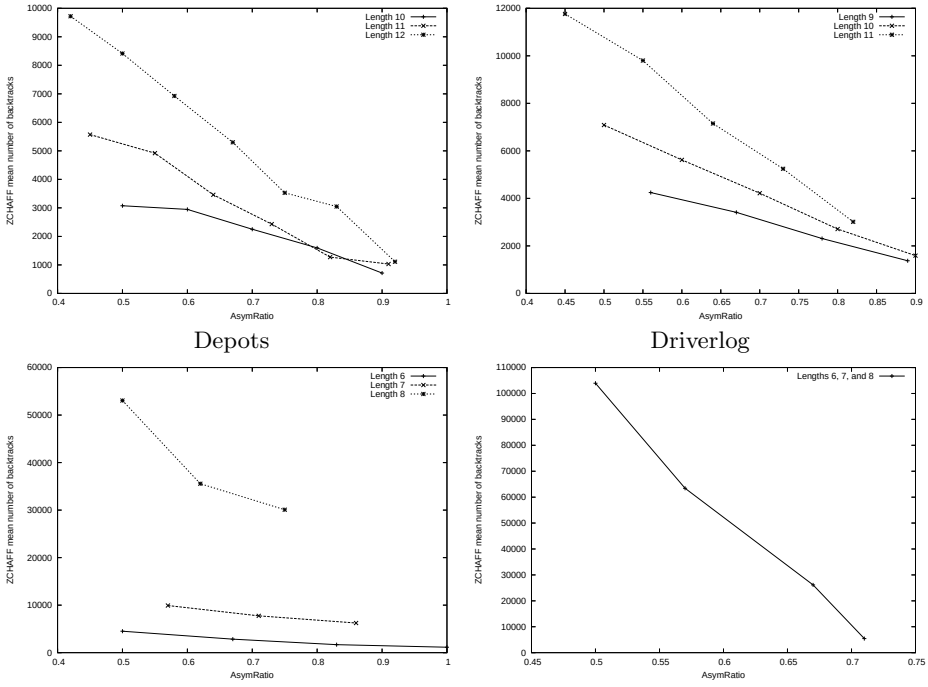


Fig. 2. Mean number of backtracks of ZChaff, plotted against *AsymRatio*, in CNF formulas encoding planning instances from the IPC-3 benchmark domains except Satellite (see text), and Rovers (which is displayed in Figure 1). Curves for different subsets $\mathcal{P}_m$ of around 50000 random instances in each domain: the subsets corresponding to the 3 most frequently occurring optimal plan lengths m. For all domains except Zenotravel, the curves are shown separately for each m. For Zenotravel, in each $\mathcal{P}_m$ there are at most two bins with over 100 instances; so the curve is for the union of $\mathcal{P}_6$, $\mathcal{P}_7$, and $\mathcal{P}_8$.

Hypothesis 1 (we also use the notations), within each domain we assigned the instances to sub-sets $\mathcal{P}_m$ with identical optimal plan length m. For each P in a set $\mathcal{P}_m$, we computed $AsymRatio(P)$, and ran ZChaff[3] on the formula $\phi(P, m-1)$, measuring the number of backtracks. We plotted the latter over $AsymRatio$ by dividing each $\mathcal{P}_m$ into 100 bins, with $AsymRatio(P) \in [0, 0.01), \ldots, [0.99, 1]$; we took the mean value out of each bin, avoiding noise by skipping bins with less than 100 elements. The results are in Figures 1 and 2. (Plots for medium values are almost identical.)

For the Rovers domain, Figure 1 clearly shows the hypothesized correlation within each of the displayed subsets $\mathcal{P}_m$, $m \in \{7, 8, 9\}$. Note that, from the relative positions of the different curves, one can see the influence of optimal plan length/formula size — the higher m, the more backtracks are needed. These observations are also typical for the other IPC-3 domains. Figure 2 shows the plots, which clearly support Hypothesis 1.[11]

[11] There is no plot for Satellite because, there, due to a lack of variance in subgoal hardness, all instances within the sets $\mathcal{P}_m$ have the same $AsymRatio$.

4 Asymmetric Structure in Synthetic Domains

As said, in order to obtain a deeper understanding of the observed correlation, we studied three classes of synthetic formulas, called MAP, SBW, and SPH. MAP and SBW come from planning domains, SPH is a structured version of the pigeon hole. Each of the formula classes/domains is parameterized by size n and structure k. In the planning domains, we use the simplified Graphplan-based encoding (see Section 2), and consider CNF formulas that are one step short of a solution. We denote the formulas with MAP_n^k, SBW_n^k, and SPH_n^k, respectively.

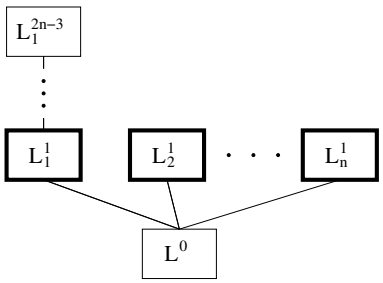
Due to space restrictions, we consider only MAP in detail, and we omit all proofs. The missing informations are available in a technical report [23]. We remark that the proofs are rather involved (the MAP proofs alone occupy 9 pages in this format), due to the many details one needs to take account of when determining the effects of UP in complicated formulas.

MAP. In the MAP domain, one moves on the road map graph, parameterized by n, shown in Figure 3 (a) and (b). The available actions take the form *move-x-y*, where x is connected to y with an edge in the graph. The precondition is $\{at\text{-}x\}$, the add effect is $\{at\text{-}y, visited\text{-}y\}$, and the delete effect is $\{at\text{-}x\}$. Initially one is located at L^0. The goal is to visit a number of locations. *What* locations must be visited depends on the value of $k \in \{1, 3, \ldots, 2n-3\}$. If $k = 1$ then the goal is to visit each of $\{L_1^1, \ldots, L_n^1\}$. For each increase of k by 2, the goal on the L_1-branch goes up by two steps, and one of the other goals is skipped. For $k = 2n-3$, the goal is $\{L_1^{2n-3}, L_2^1\}$.[12] We refer to $k = 1$ as the *bottom case*, and to $k = 2n-3$ as the *top case*, see Figure 3 (a) and Figure 3 (b), respectively.

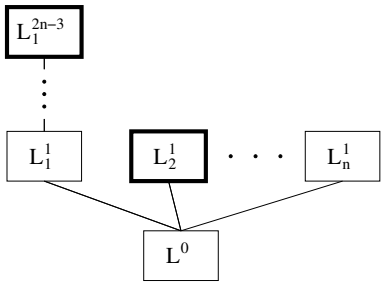
The length of a shortest plan is $2n-1$ independently of k; our CNFs encode $2n-2$ steps; *AsymRatio* is $\frac{k}{2n-1}$. Figure 3 (c) and (d) illustrate that the setting of k has a quite drastic effect on backdoor size. We will detail this below. First, observe that the setting of k has only very little impact on the size and shape of the constraint graph, illustrated in Figure 3 (e) and (f). Between formulas MAP_n^k and $MAP_n^{k'}$, $k' > k$, there is *no difference* except that $k'-k$ goal clauses are skipped, and that the content of the goal clause for the L_1-branch changes. Precisely, the number of clauses in MAP_n^k is $3n^3+27n^2-73n+39-(k+1)/2$. The number of variables is $16n^2 - 33n + 14$, irrespectively of k. Also irrespectively of k, the constraint graph contains, at each time step $2 \leq t \leq 2n-2$, large *cliques* of variables, for example the $2n$ variables corresponding to moves from or to L^0, which are fully connected due to the mutex clauses. From a clique of size m, one has to remove $m-1-b$ nodes in order to get to an induced width of $1 \leq b \leq m-1$. Since the mentioned cliques are disjoint, this shows that, for any constant b, the b-cutset size in MAP_n^k is a square function in n, irrespectively of k. Details, also on other kinds of cutsets, are in the TR [23].

The hidden structure in our formulas can not be characterized in terms of b-cutsets. It *can* be characterized in terms of the effects of unit propagation. For the bottom case, we identify a backdoor called MAP_n^1B, defined as follows:

[12] For $k = 2n-1$, MAP_n^k contains an empty clause: no supporting action for the goal is present at the last time step.



(a) goals "bottom" case



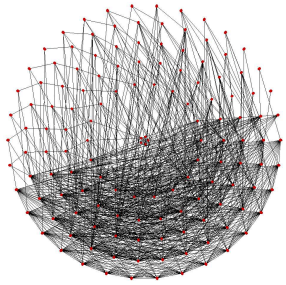
(b) goals "top" case

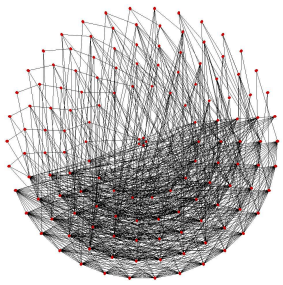
	1	2	3	4
1	NA–0			
2				
3		MV–2	MV–3 NV–3	MV–4 NV–4
4				
5		MV–2	MV–3 NV–3	MV–4 NV–4
6				

(c) backdoor "bottom" case, $n = 4$

	1	2	3	4
1	MV–1			
2				
3	MV–23			
4				
5				
6				

(d) backdoor "top" case, $n = 4$





(e) constraint graph "bottom" case, $n = 4$ (f) constraint graph "top" case, $n = 4$

Fig. 3. Goals, backdoors, and constraint graphs in MAP. In (a) and (b), goal locations are indicated in bold face, for the bottom end (a) and the top end (b) of the k scale. In (c) and (d), the horizontal axis indicates branches in the map, and the vertical axis indicates time steps; abbreviations: "NA-0" for $NOOP\text{-}at\text{-}L^0(1)$, "MV-i" for $move\text{-}L^0\text{-}L^1_i$, "NV-i" for $NOOP\text{-}visited\text{-}L^1_i$, and "MV-23" for $move\text{-}L^2_1\text{-}L^3_1$. In (e) and (f), the variables at growing time steps lie on circles with growing radius. Edges indicate common membership in at least one clause. Stepping from (e) to (f), three edges within the outmost circle disappear (one of these is visible on the left side of the pictures, just below the middle) and one new edge within the outmost circle is added.

$$
\begin{aligned}
MAP^1_nB := \; & \{move\text{-}L^0\text{-}L^1_i(t) \mid t \in T, 2 \leq i \leq n\} \; \cup \\
& \{NOOP\text{-}visited\text{-}L^1_i(t) \mid t \in T, 3 \leq i \leq n\} \; \cup \\
& \{NOOP\text{-}at\text{-}L^0(1)\} \; \cup \\
& \{move\text{-}L^0\text{-}L^1_1(t) \mid t \in T \setminus \{2n-5, 2n-3\}\}
\end{aligned}
$$

Here, $T = \{3, 5, \ldots, 2n-3\}$. Compare Figure 3(c). The size of MAP^1_nB is $\Theta(n^2)$.

Theorem 1 (MAP bottom case, backdoors). *Let $n > 1$. $MAP_n^1 B$ is a backdoor for MAP_n^1.*

To prove Theorem 1, one has to examine the effects of UP in the formulas MAP_n^1 quite closely [23]. The proof goes as follows. First, note that, in our encoding, *any pair of move actions is incompatible*. So if one move action is set to 1 at a time step, then all other move actions at that step are forced out by UP over the mutex clauses. Now, think about the backdoor variables in a backward fashion, assigning values to them starting at the last time step. In that step, the goal clauses form n constraints requiring to either visit a location L_i^1, or to have visited it earlier already (i.e., to achieve it via a NOOP). When assigning values to all $MAP_n^1 B$ variables at that time, at least $n-1$ goal constraints will be transported to the time step below. Iterating the argument, one gets at least 1 goal constraint at time 2. Taking account of several case distinctions, e.g. about the value assigned to $NOOP\text{-}at\text{-}L^0(1)$, one can show that, after UP, $n-2$ of the $move\text{-}L^0\text{-}L_i^1(t)$ variables, $i \neq 1$, are set to 1 in non-adjacent time steps t. With case distinctions about at exactly what non-adjacent t the move variables are set to 1, one can show that UP also enforces commitments to accommodate the remaining 2 $move\text{-}L^0\text{-}L_i^1$ actions – for which there is not enough room left.

We conjecture that the backdoor identified in Theorem 1 is also a minimum size (i.e., an optimal) backdoor; for $n \leq 4$ we verified this empirically. Note that the total number of variables in the CNF is also a square function in n, so the backdoor is a linear-size variable subset. We proved that the backdoor is minimal, i.e., does not contain redundant variables.

Theorem 2 (MAP bottom case, backdoors minimality). *Let $n > 1$. Let B' be a subset of $MAP_n^1 B$ obtained by removing one variable. Then the number of UP-consistent assignments to the variables in B' is always greater than 0, and at least $(n-3)!$ for $n \geq 3$.*

The proof of this theorem is a matter of figuring out how wrong things can go when a variable is missing in the proof of Theorem 1.

Using the convention that L_1^0 stands for L^0, the backdoors we identify for the top case, called MAP_n^{2n-3}, have the form:

$$MAP_n^{2n-3} B := \{move\text{-}L_1^{2^i-2}\text{-}L_1^{2^i-1}(2^i - 1) \mid 1 \leq i \leq \lceil log_2 n \rceil\}$$

Compare Figure 3 (d). Obviously, the size of $MAP_n^1 B$ is $\lceil log_2 n \rceil$.

Theorem 3 (MAP top case, backdoors). *Let $n > 1$. $MAP_n^{2n-3} B$ is a backdoor for MAP_n^{2n-3}.*

We again conjecture that this is also a minimum size, optimal, backdoor. For $n \leq 8$ we verified this empirically. We can show that the backdoor is minimal.

Theorem 4 (MAP top case, backdoors minimality). *Let $n > 1$. Let B' be a subset of $MAP_n^{2n-3} B$ obtained by removing one variable. Then there is exactly one UP-consistent assignment to the variables in B'.*

The $\Theta(log_2 n)$ backdoor size proved here for $AsymRatio = \frac{2n-3}{2n-1}$, compared to the $\Theta(n^2)$ backdoor from Theorem 1 for $AsymRatio = \frac{1}{2n-1}$, nicely reflects our empirical findings. We consider it particularly interesting that the MAP_n^{2n-3} formulas have *logarithmic* backdoors. This shows, on the one hand, that these formulas are (potentially) easy for Davis Putnam procedures, having polynomial-size proofs. On the other hand, the formulas are non-trivial, in two important respects. First, they do have non-constant backdoors and are not just solved by unit propagation. Second, *finding* the logarithmic backdoors involves, at least, a non-trivial branching heuristic: the *worst*-case DPLL proofs for MAP_n^{2n-3} are still exponential in n.

The MAP_n^{2n-3} formulas being interesting in that way, it is instructive to have a closer look at how the logarithmic backdoors arise. The proof of Theorem 3 uses the following two properties of UP, in MAP_n^{2n-3}:

(1) If one sets a variable $move\text{-}L_1^{i-1}\text{-}L_1^i(i)$ to 1, then at all time steps $j < i$ a move variable is set to 1 by UP.
(2) If one sets a variable $move\text{-}L_1^{i-1}\text{-}L_1^i(i)$ to 0, then at all time steps $j > i$ a move variable is set to 1 by UP.

Both properties are caused by the "tightness" of branch 1, i.e., by UP over the precondition clauses of the actions moving along that branch, in combination with the goal to visit the outmost location. Other than what one may think at first sight, the two properties by themselves are *not* enough to determine the log-sized backdoor. The properties just form the foundation of a subtle interplay between the different settings of the backdoor variables, exploiting exponentially growing UP implication chains on branch 1. The interplay is best explained with an example. For $n = 8$, the backdoor is $\{move\text{-}L^0\text{-}L_1^1(1), move\text{-}L_1^2\text{-}L_1^3(3), move\text{-}L_1^6\text{-}L_1^7(7)\}$. Figure 4 contains an illustration.

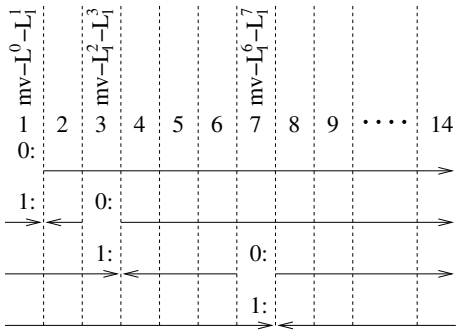


Fig. 4. The workings of the optimal backdoor for MAP_8^{13}. Arrows indicate moves on the L_1-branch forced to 1 by UP. Direction $\rightarrow$ means towards L_1^{13}, $\leftarrow$ means towards L^0. When only a single open step is left, $move\text{-}L^0\text{-}L_2^1$ is forced to 1 at that step by UP, yielding a contradiction.

Consider the first (lowest) variable in the backdoor, $move\text{-}L^0\text{-}L_1^1(1)$. If one sets this to 0, then property (2) applies: only 13 of the 14 available steps are

left to move towards the goal location L_1^{13}; UP recognizes this, and forces moves towards L_1^{13} at all steps $2 \leq t \leq 14$. Since $t = 1$ is the only remaining time step not occupied by a move action, UP over the L_2^1 goal clause sets *move*-L^0-$L_2^1(1)$ to 1, yielding a contradiction to the precondition clause of the move set to 1 at time 2. So *move*-L^0-$L_1^1(1)$ must be set to 1.

Consider the second variable in the backdoor, *move*-L_1^2-$L_1^3(3)$. Say one sets this to 0. By property (2) this forces moves at all steps $4 \leq t \leq 14$. So the goal for L_2^1 must be achieved by an action at step 3. But we have committed to *move*-L^0-L_1^1 at step 1. This forces us to move back to L^0 at step 2 and to move to L_2^1 at step 3. But then the move forced in earlier at 4 becomes impossible. It follows that we must assign *move*-L_1^2-$L_1^3(3)$ to 1. With property (1), this implies that, by UP, all time steps below 3 get occupied with move actions. (Precisely, in our case here, *move*-L_1^1-$L_1^2(2)$ is also set to 1.)

Consider the third variable in the backdoor, *move*-L_1^6-$L_1^7(7)$. If we set this to 0, then by property (2) moves are forced in by UP at the time steps $8 \leq t \leq 14$. So, to achieve the L_2^1 goal at step 7, we have to take **three steps to move back from L_1^3 to L^0**: steps 4, 5, and 6. A move to L_2^1 is forced in at step 7, in contradiction to the move at 8 forced in earlier. Finally, if we assign *move*-L_1^6-$L_1^7(7)$ to 1, then by property (1) moves are forced in by UP at all steps below 7. We need **seven steps to move back from L_1^7 to L^0**, and an eighth step to get to L_2^1. But we have only the 7 steps $8, \ldots, 14$ available.

The key to the logarithmic backdoor size is that, to achieve the L_2^1 goal, we have to move back from L_1^t locations we committed to earlier (as indicated in bold face above for $t = 3$ and $t = 7$). We committed to move to L_1^t, and the UP propagations force us to move back, thereby occupying $2 * t$ steps in the encoding. This yields the possibility to double the value of t between variables.

Proving Theorem 4 is a matter of figuring out what can go wrong in the proof to Theorem 3, after removing one variable [23]. Note that, with the above, the DPLL proof for MAP_n^{2n-3} actually *degenerates to a line*, and has only $\lceil log_2 n \rceil$ (non-failed) nodes. Besides small backdoors, such degenerated proof trees are probably also typical in structured examples.[13]

It would be interesting to determine what the optimal backdoors are in general, i.e. in MAP_n^k, particularly at what point the backdoors become logarithmic. Such an investigation turns out to be extremely difficult. For interesting combinations of n and k it is practically impossible to find the optimal backdoors empirically, and so get a start into the theoretical investigation. We developed an enumeration program that exploits symmetries in the planning task to cut down on the number of variable sets to be enumerated. Even with that, the enumeration didn't scale up far enough. We leave this topic for future work.

SBW. This is a block-stacking domain (n blocks), with restrictions on what blocks can be stacked onto what other blocks. These are initially all located

[13] In fact, we proved in the meantime that the size of DPLL search trees for MAP_n^1 is exponentially lower-bounded in n. (The proof goes by a reduction to the Pigeon Hole problem, and is not yet available in the TR.) This shows a *doubly exponential* complexity gap between DPLL proofs in the bottom and top cases.

side-by-side on a table t_1. The goal is to bring all blocks onto another table t_2, that has only space for a single block; so the n blocks must be arranged in a single stack on top of t_2. The parameter k, $0 \leq k \leq n$, defines the amount of *stacking restrictions*. There are k "bad" blocks $b_1, \ldots, b_k$ and $n-k$ "good" blocks $g_1, \ldots, g_{n-k}$. For $1 < i \leq k$, b_i can only be stacked onto b_{i-1}; b_1 can be stacked onto t_2 and any g_i. The g_i can be stacked onto each other, and onto t_2.

Independently of k, the optimal plan length is n: a single move action stacks one block onto another block or a table. *AsymRatio* is $\frac{1}{n}$ if $k = 0$, and $\frac{k}{n}$ otherwise. Our CNF formulas encode $n-1$ action steps. In the bottom case, $k = 0$, we prove the existence of backdoors of size $\Theta(n^3)$. In the top case, $k = n-2$, there are $O(log_2 n)$ backdoors.

SPH. Finally, we constructed a *non-Planning* example that also exhibits similar asymmetric structure and backdoor size behavior. We modified the well-known Pigeon Hole problem. In our SPH_n^k formulas, like in the classical Pigeon Hole problem, the task is to assign $n+1$ pigeons to n holes. The difference lies in that there is now one "bad" pigeon that requires k holes, and $k-1$ "good" pigeons that can share a hole with the bad pigeon. The remaining $n-k+1$ pigeons are normal, i.e., need exactly one hole each. The range of k is between 1 and $n-1$. Independently of k, $n+1$ holes are needed overall. Apart from identifying minimal backdoors for all k and n, for $k = n-1$ we identify an $O(n)$ DPLL proof, which implies an exponential complexity gap to $k = 1$ [24].

5 Conclusions

Current DPLL-based SAT solvers are very efficient in "structured" formulas encoding real-world applications from Planning and Verification. We considered the effect of structure on the hardness of Planning formulas. Our findings reveal a mechanism – strong asymmetries in subgoal structure, a semantic notion – which can give rise to very small DPLL proofs. Most interestingly, with high subgoal asymmetry, we identified classes of planning formulas with *logarithmic* size backdoors and DPLL proofs.

Our results promote the understanding of what is relevant for solver performance in practice. Such an understanding is, we think, important in itself. From a more practical point of view, it may inspire the development of new search heuristics. As a simple example, our analysis of minimal backdoors suggests to use different branching heuristics depending on the value of *AsymRatio* (which can be approximated using various techniques from the literature [9–11]): with a high *AsymRatio*, concentrate on the actions supporting the most difficult subgoal; with a low *AsymRatio*, do not concentrate on any particular sub-goal and distribute the branching more uniformly.

Our results are, so far, mainly for planning domains. We are currently extending our work to consider other constraint reasoning applications. In general, we hope that our approach will lead to the investigation of other forms of problem structure that can be identified empirically, and be captured in synthetic domains and analyzed rigorously.

References

1. Cheeseman, P., Kanefsky, B., Taylor, W.M.: Where the *really* hard problems are. In: Proc. IJCAI'91. (1991) 331–337
2. Hogg, T., Huberman, B., Williams, C.: Phase Transitions and Complexity. Artificial Intelligence **81** (1996)
3. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. DAC'01. (2001) 530–535
4. Kautz, H., Selman, B.: Unifying SAT-based and graph-based planning. In: Proc. IJCAI'99. (1999) 318–325
5. Hoffmann, J., Edelkamp, S.: The deterministic part of IPC-4: An overview. Journal of Artificial Intelligence Research (2005) To appear.
6. Williams, R., Gomes, C.P., Selman, B.: Backdoors to typical case complexity. In: Proc. IJCAI'03. (2003)
7. Long, D., Fox, M.: The 3rd international planning competition: Results and analysis. Journal of Artificial Intelligence Research **20** (2003) 1–59
8. Edelkamp, S., Hoffmann, J., Englert, R., Liporace, F., Thiebaux, S., Trüg, S.: Engineering benchmarks for planning: the domains used in the classical part of IPC-4. Journal of Artificial Intelligence Research (2005)
9. Blum, A.L., Furst, M.L.: Fast planning through planning graph analysis. Artificial Intelligence **90** (1997) 279–298
10. Hoffmann, J., Nebel, B.: The FF planning system: Fast plan generation through heuristic search. Journal of Artificial Intelligence Research **14** (2001) 253–302
11. Edelkamp, S.: Planning with pattern databases. In Cesta, A., Borrajo, D., eds.: Recent Advances in AI Planning. 6th European Conference on Planning (ECP'01), Toledo, Spain, Springer-Verlag (2001) 13–24
12. Dechter, R.: Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. Artificial Intelligence **41** (1990) 273–312
13. Rish, I., Dechter, R.: Resolution versus search: Two strategies for SAT. JAR **24** (2000) 225–275
14. Dechter, R.: Constraint Processing. Morgan-Kauffmann (2003)
15. Climer, S., Zhang, W.: Searching for backbones and fat: A limit-crossing approach with applications. In: Proc. AAAI-02, AAAI Press (2002)
16. Sang, T., Bacchus, F., Beame, P., Kautz, H., Pitassi, T.: Combining Component Caching and Clausal Learning for Effective Model Counting. In: SAT04. (2004)
17. Hulubei, T., O'Sullivan, B.: Optimal refutations for constraint satisfaction problems. In: Proc. the International Joint Conference on Artificial Intelligence (IJCAI05), Seattle, AAAI Pess (2001)
18. Li, W., van Beek, P.: Guiding Real-world SAT Solving with Dynamic Hypergraph Separator Decomposition. Proc. ICTAI-04 (2004)
19. Nudelman, E., Leyton-Brown, K., Hoos, H., Devkar, A., Shoham, Y.: Understanding random SAT: beyond the clauses-to-variable ratio. In: CP-04. (2004) 438–452
20. Slaney, J., Walsh, T.: Backbones in optimization and approximation. In: IJCAI01. (2001)
21. Hoffmann, J., Porteous, J., Sebastia, L.: Ordered landmarks. Journal of Artificial Intelligence Research (2004)
22. Helmert, M.: Complexity results for standard benchmark domains in planning. Artificial Intelligence **143** (2003) 219–262
23. Hoffmann, J., Gomes, C., Selman, B.: Structure and problem hardness: Asymmetry and DPLL proofs in SAT-based planning. Technical Report (2005) Available at http://www.mpi-sb.mpg.de/∼hoffmann/tr05.ps.
24. Buss, S., Pitassi, T.: Resolution and the weak pigeon-hole principle. In: Proceedings of Computer Science Logic (CSL'97). (1997) 149–156

Bound Consistencies for the discrete CSP

Christophe Lecoutre and Julien Vion

CRIL-CNRS FRE 2499,
Université d'Artois
Lens, France
{*lecoutre*, *vion*}@cril.univ-artois.fr

Abstract. Many works in the area of Constraint Programming have focused on inference, and more precisely, on filtering methods based on properties of constraint networks. Such properties are called domain filtering consistencies when they allow removing some inconsistent values from the domains of variables, and bound consistencies when they focus on bounds of domains. In this paper, we study the relationship between consistencies introduced with respect to discrete and continuous constraint networks, and experiment the effectiveness of exploiting bound consistencies on discrete instances.

1 Introduction

Many problems arising in Artificial Intelligence and Computer Science involve constraint satisfaction as an essential component. Such problems occur in numerous domains such as scheduling, planning, molecular biology and circuit design. The methods that have been developed for processing constraints can be classified into inference and search [13]. Inference is used to transform a problem into an equivalent form which is simpler than the original one while search is used to traverse the search space of the problem in order to find a solution. Problems involving constraints are usually represented by so-called constraint networks.

A constraint network is simply composed of a set of variables and of a set of constraints. Finding a solution to a constraint network involves assigning a value to each variable such that all constraints are satisfied. The Constraint Satisfaction Problem (CSP) is the task to determine whether or not a given constraint network, also called CSP instance, is satisfiable. It comes in two forms. The first one, called discrete or finite CSP, corresponds to constraint networks such that each variable takes its values in an associated discrete domain while the second one, called continuous or numeric CSP, corresponds to networks such that each variable takes its values in an associated continuous domain.

Many works in the area of Constraint Programming have focused on inference, and more precisely, on filtering methods based on properties of constraint networks. The idea is to exploit such properties in order to identify some no-goods where a no-good corresponds to a set of variable assignments that can not lead to any solution. Properties that allow identifying no-goods of size 1, which correspond to inconsistent values, are called domain filtering consistencies [12]. In this paper, we focus on domain filtering consistencies that have been introduced with respect to discrete and continuous CSP instances.

On the one hand, when dealing with discrete CSP instances, a usual approach to solve them is to use the MAC algorithm, i.e. the algorithm which maintains arc consistency during search. Arc consistency (AC) means that any value occurring in the associated domain of a variable X admits at least a support in each constraint involving X. Recent works have shown that there exist promising alternatives to AC, namely, max-restricted path consistency (Max-RPC) [10] and singleton arc consistency (SAC) [11]. Max-RPC and SAC are stronger consistencies than AC, that is to say, they allow identifying more inconsistent values than AC does. Max-RPC holds when all values have at least one path consistent support on each constraint whereas SAC holds when the constraint network can be made arc consistent after any variable assignment. It can be useful to establish Max-RPC or SAC at pre-processing time (i.e. before search) [22, 12], but it seems that maintaining such a strong consistency during search does require some control about the effort performed at each step. In fact, it remains an open issue although recent advances [1, 5, 18] show it is a direction of future research.

On the other hand, when dealing with continuous CSP instances, one has to reason about intervals. For instance, it is possible to represent a domain by a finite set of (disjoint) continuous intervals and to propose some adaptations [16, 14] of the arc consistency enforcing algorithm which, otherwise, is subject to early quiescence and infinite iterations. However, it is more usual that domains are considered as convex, i.e. represented by a single interval. By restricting arc consistency with respect to the bounds of each (convex) domain, new consistencies can be introduced. The consistency that is based on an approximation (in order to maintain domains convex) of projection functions for the narrowing of domains is called 2B-consistency (2B) by [19] and hull-consistency by [2]. However, it requires, for each pair (C,X) composed of a constraint C and a variable X, the existence of two functions computing the min bound and the max bound of the set of values given by the projection over X of the set of supports of C. When such functions can not be exhibited, it is necessary to perform some decomposition of the constraint system. Another consistency, called box-consistency [2], exploits interval arithmetic and does not require any constraint decomposition. It is known [9] that 2B and box-consistency match when no variable occurs several times in the expression of a constraint. It is also possible to define stronger consistencies than 2B or box-consistency by assuming that each variable is assigned, in turn, with the two bounds of its domain and by checking consistency when establishing 2B or box-consistency. Such consistencies are called 3B-consistency (3B) [19] and bound-consistency, respectively. Finally, it is possible to introduce a recursive definition of kB-consistencies [20] with $k \geq 2$.

In the following, we will define a consistency restricted to the bounds of the domains as a bound consistency. For example, 2B corresponds to bound AC while 3B is a relaxation of bound SAC. The aim of this paper is to study the practical effectiveness of exploiting bound consistencies with respect to discrete CSP instances, as even if 2B has been integrated into some constraint logic programming solvers [8, 15], we are not aware of any experimental comparison involving different bound consistencies wrt finite domains.

2 Domain Filtering Consistencies

In this section, we introduce some consistencies that allow removing some inconsistent values from the domains of a constraint network (CN). Such consistencies, called domain filtering consistencies in [12], share the nice property of not modifying the structure of the network.

Definition 1. *A Constraint Network P is a pair $(\mathscr{X}, \mathscr{C})$ where:*

- *$\mathscr{X} = \{X_1, \ldots, X_n\}$ is a finite set of n variables such that each variable X_i has an associated domain $dom(X_i)$ denoting the set of values allowed for X_i,*
- *$\mathscr{C} = \{C_1, \ldots, C_m\}$ is a finite set of m constraints such that each constraint C_j has an associated relation $rel(C_j)$ denoting the set of tuples allowed for the variables $vars(C_j) \subseteq \mathscr{X}$ involved in the constraint C_j.*

For any variable X, $\min(X)$ and $\max(X)$ will respectively denote the smallest and greatest values in $\mathrm{dom}(X)$. Note that a value will usually refer to a pair (X,a) with $X \in \mathscr{X}$ and $a \in \mathrm{dom}(X)$. We will note $(X, a) \in P$ (respectively, $(X, a) \notin P$) iff $X \in \mathscr{X}$ and $a \in \mathrm{dom}(X)$ (respectively, $a \notin \mathrm{dom}(X)$).

A constraint network is said to be satisfiable iff it admits at least a solution. The Constraint Satisfaction Problem (CSP) is the NP-complete task of determining whether a given constraint network, also called CSP instance, is satisfiable. To solve a CSP instance, a depth-first search algorithm with backtracking can be applied, where at each step of the search, a variable assignment is performed followed by a filtering process called constraint propagation. Usually, constraint propagation algorithms are based on domain filtering consistencies, among which the most widely studied ones are called arc consistency, max-restricted path consistency and singleton arc consistency.

Arc Consistency (AC) is the basic property of constraint networks. It guarantees that each value occurs in at least a support of each constraint. Algorithms to establish AC entails removing all arc inconsistent values and can be classified into coarse-grained and fine-grained algorithms. Optimal worst-case time complexity to establish AC is $\mathrm{O}(md^2)$ where d is the size of the largest domain.

Definition 2. *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN, $X \in \mathscr{X}$ and $a \in dom(X)$. (X, a) is arc consistent iff $\forall C \in \mathscr{C} | X \in vars(C)$, there exists a support of (X, a) in C, i.e., a tuple $t \in rel(C)$ such that $t[X] = a$. P is arc consistent iff $\forall X \in \mathscr{X}$, $dom(X) \neq \emptyset$ and $\forall a \in dom(X)$, (X, a) is arc consistent.*

Max-Restricted Path Consistency (Max-RPC) can be seen as a generalization of restricted path consistency [3] and k-restricted path consistency [10] and also as a restriction of path consistency [21]. It is defined with respect to binary constraint networks, i.e. networks that only involves binary constraints. Max-RPC guarantees that each value can be found a path in each 3-clique of the network. Optimal worst-case time complexity to establish Max-RPC is $\mathrm{O}(mn + md^2 + cd^3)$ where c denotes the number of 3-cliques in the constraint network.

Definition 3. *Let $P = (\mathscr{X}, \mathscr{C})$ be a binary CN, $X_i \in \mathscr{X}$ and $a \in dom(X_i)$. (X_i, a) is max-restricted path consistent iff $\forall C_{ij} \in \mathscr{C}$, $\exists b \in dom(X_j)$ s.t. $(a, b) \in rel(C_{ij})$ and $\forall X_k \in \mathscr{X} | C_{ik} \in \mathscr{C} \wedge C_{jk} \in \mathscr{C}$, $\exists c \in dom(X_k)$ s.t. $(a, c) \in rel(C_{ik}) \wedge (b, c) \in rel(C_{jk})$. P is max-restricted path consistent iff $\forall X_i \in \mathscr{X}$, $dom(X_i) \neq \emptyset$ and $\forall a \in dom(X_i)$,(X_i, a) is max-restricted path consistent.*

Singleton Arc Consistency (SAC) is a stronger consistency than Max-RPC which is itself stronger than AC. It means that SAC can identify more inconsistent values than Max-RPC can, and subsequently more than AC can. SAC guarantees that enforcing arc consistency after performing any variable assignment does not show unsatisfiability, i.e., does not entail a domain wipe-out. Optimal worst-case time complexity to establish SAC is $O(mnd^3)$ [5].

To give a formal definition of SAC, we need to introduce some notations. $AC(P)$ denotes the constraint network obtained after enforcing arc consistency on a given constraint network P. $AC(P)$ is such that all values of P that are not arc consistent have been removed. If there is a variable with an empty domain in $AC(P)$, denoted $AC(P) = \bot$, then P is clearly unsatisfiable. $P|_{X=a}$ is the constraint network obtained from P by restricting the domain of X to $\{a\}$.

Definition 4. *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN, $X \in \mathscr{X}$ and $a \in dom(X)$. (X, a) is singleton arc consistent iff $AC(P|_{X=a}) \neq \bot$. P is singleton arc consistent iff $\forall X \in \mathscr{X}$, $dom(X) \neq \emptyset$ and $\forall a \in dom(X)$, (X, a) is singleton arc consistent.*

Finally, [4] have proposed an extension of SAC that is called Singleton Propagated Arc Consistency (SPAC). It is based on the following observation. If $(Y, b) \notin AC(P|_{X=a})$ then it corresponds to the detection of the nogood $\neg(X = a \wedge Y = b)$ and we can deduce that $(X, a) \notin AC(P|_{Y=b})$. We can exploit this inference when checking the singleton arc consistency of (Y, b) as it gives more chances to detect an inconsistency.

Definition 5. *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN, $X \in \mathscr{X}$ and $a \in dom(X)$. (X, a) is singleton propagated arc consistent iff $\tilde{P}|_{X=a} \neq \bot$ where $\tilde{P}|_{X=a}$ is the constraint network obtained from P by removing any value (Y,b) of P (i.e. b from $dom(Y)$) such that $(X, a) \notin AC(P|_{Y=b})$. P is singleton propagated arc consistent iff $\forall X \in \mathscr{X}$, $dom(X) \neq \emptyset$ and $\forall a \in dom(X)$, (X, a) is singleton propagated arc consistent.*

It is possible to define a bound version for any domain filtering consistency Φ as follows.

Definition 6. *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN. P is bound Φ-consistent iff $\forall X \in \mathscr{X}$, $dom(X) \neq \emptyset$ and both $min(X)$ and $max(X)$ are Φ-consistent.*

On the other hand, 2B and 3B [19] are consistencies that have been introduced wrt continuous constraint networks. $2B(P)$ denotes the constraint network obtained after enforcing 2B on a given constraint network P and $2B(P) = \bot$ indicates that there is a variable with an empty domain in $2B(P)$.

Definition 7. *$P = (\mathscr{X}, \mathscr{C})$ is 2B-consistent iff $\forall X \in \mathscr{X}$, $dom(X) \neq \emptyset$ and both $min(X)$ and $max(X)$ are arc consistent. P is 3B-consistent iff $\forall X \in \mathscr{X}$, $dom(X) \neq \emptyset$ and both $2B(P|_{X=min(X)}) \neq \bot$ and $2B(P|_{X=max(X)}) \neq \bot$.*

Algorithm 1 seekSupportArc(C : Constraint, X : Variable, a : Value) : boolean

1: $t \leftarrow \bot$
2: **while** $t \neq \top \wedge C(t) = false$ **do**
3: $\quad t \leftarrow setNextTuple(C, X, a, t)$
4: return $t \neq \top$

Algorithm 2 revise(C : Constraint, X : Variable) : boolean

1: $domainSize \leftarrow |dom(X)|$
2: **while** $|dom(X)| > 0 \wedge \neg\, seekSupportArc(C, X, min(X))$ **do**
3: $\quad$ remove $min(X)$ from dom(X)
4: **while** $|dom(X)| > 1 \wedge \neg\, seekSupportArc(C, X, max(X))$ **do**
5: $\quad$ remove $max(X)$ from dom(X)
6: return $domainSize \neq |dom(X)|$

Clearly, 2B-consistency corresponds to bound AC while 3B-consistency is a relaxation of bound SAC since for each pair (X,a) with $a = \min(X)$ or $a = \max(X)$, 3B-consistency requires that 2B$(P|_{X=a}) \neq \bot$ whereas bound SAC requires that AC$(P|_{X=a}) \neq \bot$. We can also observe (see next sections) that a consistency and its bound version admit the same optimal worst-case time complexity. For example, establishing AC or 2B is O(md^2) while establishing SAC, bound SAC or even 3B is O(mnd^3). This statement seems to be in contradiction with the optimality of 2B-consistency and 3B-consistency algorithms which is O(md) [19] and O(mnd^2) [7], respectively. However, it is then assumed that all constraints are basic, that is to say, that for each constraint C, it is possible to find two functions that compute in bounded time the min bound and the max bound of the domain of any variable involved in C.

One nice advantage of exploiting bound consistencies is that space complexity can be very affordable. Indeed, it is possible to reduce the space required by some algorithms by a factor d or even d^2 as we can just generate data structures wrt two bounds. Further, if convex domains are considered, i.e. domains are such that all values between the min and the max bounds belong to the domain, then a constraint network can be represented in O($n + m$). It can be very useful for networks involving variables with large domains as for some scheduling instances.

Finally, remark that we have ignored in this paper the adaptation of (numeric) consistencies such as box-consistency and bound-consistency wrt discrete CSP instances.

3 2B (Bound arc consistency)

Arc consistency (AC) is the most studied and used local consistency. Algorithm 4 is the bound adaptation of the coarse-grained arc consistency algorithm AC3 [21]. It just calls Algorithm 3 with the set of variables of the given constraint network as a second parameter. This second algorithm allows establishing bound arc consistency of the given constraint network by initializing a set Q with some

Algorithm 3 2B ($P = (\mathscr{X}, \mathscr{C})$: CN, S : set of Variables)

1: $Q \leftarrow \{(C, X) \mid C \in \mathscr{C} \wedge X \in vars(C) \wedge \exists Y \in S \cap vars(C) | Y \neq X\}$
2: **while** $Q \neq \emptyset$ **do**
3: pick and delete (C, X) in Q
4: **if** revise(C,X) **then**
5: $Q \leftarrow Q \cup \{(C', X') \mid X \in vars(C') \wedge X' \in vars(C') \wedge C \neq C'\}$
6: **end while**

Algorithm 4 2B ($P = (\mathscr{X}, \mathscr{C})$: CN)

1: 2B(P,$\mathscr{X}$)

arcs and then performing successive revisions until a fix-point is reached. Algorithm 3 has been introduced as it is useful later in the paper. But, assuming that no unary constraint is allowed, one should observe that the call $2B(P, \mathscr{X})$ (line 1 of Algorithm 4) involves the following standard initialization of the set Q (line 1 of Algorithm 3):

$$Q \leftarrow \{(C, X) \mid C \in \mathscr{C} \wedge X \in vars(C)\}$$

Hence, initially, all arcs (C, X) are put in a set Q. Then, each arc is revised in turn, and when a revision is effective (at least one value has been removed), the set Q has to be updated. A revision is performed by a call to the function $revise(C, X)$, depicted in Algorithm 2 and entails removing values at bounds of dom(X) that have become inconsistent with respect to C. The algorithm is stopped when the set Q becomes empty. Remark that when a revision of an arc (C, X) is effective, it is necessary to take into account the arcs of the form (C', X) (with $C \neq C'$) since the consistency of the new bound(s) of dom(X) is not guaranteed wrt C'. The function *seekSupportArc*, depicted in Algorithm 1, determines from scratch whether or not there exists a support of (X, a) in C. It iteratively calls the function *setNextTuple* which returns either the smallest valid tuple t' in C such that $t \prec t'$ and $t'[X] = a$ or $\top$ if it does not exist. Note that $C(t)$ must be understood as a constraint check and that $C(\bot)$ returns false.

Finally, Algorithm 4 can also be seen as an adaptation of the procedure IP_1 proposed in [19] where it is assumed that constraints are basic. Also, a variant with a constraint-oriented propagation scheme can be found in [7].

Proposition 1. *Applied to binary constraint networks, Algorithm 4 admits a worst-case time and space complexity in O(md^2) and O(m), respectively.*

Proof. Each arc (C,X) may enter d times in Q to be revised [21, 7, 6]. When a revision entails no removal, at most $2 \times d$ constraint checks are performed. When some removals occur, there are at most d additional constraint checks per value removed. For each arc, we then obtain $2 \times d \times d + d \times d$ as an upper bound of the global number of constraint checks. As there are $2 \times m$ different arcs, we obtain a worst-case time complexity in O(md^2). On the other hand, the only structure used by the algorithm is the queue Q which is O(m). □

Algorithm 5 seekSupportPath(C_{ij} : Constraint, X_i : Variable, a : Value) : bool

```
1: for each value b ∈ dom(X_j) s.t. C_ij(a, b) do
2:    for each variable X_k s.t. (X_i,X_j,X_k) forms a 3-clique do
3:       for each value c ∈ dom(X_k) do
4:          if C_ik(a, c) ∧ C_jk(b, c) then
5:             continue loop 2:
6:       end for
7:       continue loop 1:
8:    end for
9:    return true
10: end for
11: return false
```

It is interesting to note that even if last supports are recorded as with an underlying optimal arc consistency technique such as AC2001/3.1, the worst-case time complexity remains $O(md^2)$ although one could have expected a better complexity as bound consistencies only consider the min and the max values.

4 2B+ (Bound max-restricted path consistency)

Max-Restricted Path Consistency (Max-RPC) [10] is one of the most promising local consistencies. Max-RPC is stronger than arc consistency, restricted path consistency [3] and k-restricted path consistency [10] but weaker than singleton arc consistency. In this section, we propose a bound adaptation of Max-RPC in the context of a coarse-grained algorithm. Actually, as this adaptation, denoted 2B+, does not guarantee that each bound has a path consistent support with respect to each constraint, it should be viewed as an opportunistic algorithm that is simple to define and implement.

The algorithm 2B+ is obtained from 2B by simply replacing, in function *revise*, calls to function *seekSupportArc* by calls to function *seekSupportPath* which is described by Algorithm 5. The function *seekSupportPath* returns true (line 10) iff the given value has a path consistent support on the given constraint. In order to guarantee that the resulting constraint network is (at least) arc consistent, we have to replace $C \neq C'$ by $(C \neq C' \vee X \neq X')$ in line 5 of Algorithm 3. It means that, when the revision of an arc (C_{ij}, X_i) is effective, it is necessary to take into account the arc (C_{ij}, X_j). Indeed, let us suppose that (C_{ij}, X_j) has been revised and that $a = \min(X_i)$ has been found as a path consistent support for $b = \min(X_j)$ requiring a value c for a variable X_k. Next, some revision is performed that entails the removal of (X_k,c) and (C_{ij}, X_i) is revised. Imagine that $a = \min(X_i)$ has no more path consistent support in $\mathrm{dom}(X_j)$ ($b = \min(X_j)$ was one such support but it required c that has been removed) then a is removed. If the arc (C_{ij}, X_j) is not added to Q, then it is possible that propagation finishes although (X_j,b) is not supported by X_i.

Algorithm 6 3B-X($P = (\mathscr{X}, \mathscr{C})$: CN)

```
1: P ← 2B(P,𝒳)
2: repeat
3:    P_before ← P
4:    for each X ∈ 𝒳 do
5:       domainSize ← |dom(X)|
6:       while |dom(X)| > 0 ∧ ¬ check2B-X(P,X,min(X)) do
7:          remove min(X) from dom(X)
8:       while |dom(X)| > 1 ∧ ¬ check2B-X(P,X,max(X)) do
9:          remove max(X) from dom(X)
10:      if |dom(X)| < domainSize then
11:         P ← 2B(P,{X})
12:   end for
13: until P = P_before
```

5 3B (Bound singleton arc consistency)

There is a recent attraction about singleton consistencies, and more particularly about SAC (Singleton Arc Consistency), as illustrated by recent works of [11, 22, 1, 4, 5, 18]. Even if it is possible to propose an algorithm to establish bound SAC, it does not seem quite appropriate when dealing with large domains as AC requires to represent domains in extension (and not by simple intervals). We propose here two algorithms to establish 3B which can be seen as a relaxation of bound SAC.

5.1 3B-X

Algorithm 6 is the bound adaptation, denoted 3B-X, of a basic singleton arc consistency algorithm. 3B-X starts by enforcing bound arc consistency (2B) on the given network (line 1). Then, each bound of the domain of each variable is checked to be 2B-consistent by calling the function $check2B-X$ (lines 6 and 8). Two variants, denoted $check2B-1$ and $check2B-2$, of this function are given in the subsequent subsections. Bounds that are not consistent are then removed (lines 7 and 9). When the domain of a variable is modified, bound arc consistency is maintained (lines 10 and 11). The process continues until a fix-point is reached.

5.2 3B-1

3B-1 corresponds to the algorithm 6 that uses the function $check2B-1$ depicted by Algorithm 7. Roughly speaking, 3B-1 is the bound adaptation of the singleton arc consistency algorithm SAC-1 [11].

Proposition 2. *Applied to binary constraint networks, Algorithm 3B-1 admits a worst-case time and space complexity in $O(mn^2d^3)$ and $O(m)$, respectively.*

Algorithm 7 check2B-1($P = (\mathscr{X}, \mathscr{C})$: CN, X : Variable, a : Value) : boolean

1: return 2B($P|_{X=a}, \{X\}) \neq \bot$

Proof. The number of turns of the main loop of Algorithm 3B-X is at most nd, one element being removed at each turn. The number of calls to check2B-X is $2 * n$ at each turn. As a call to check2B-1 is equivalent to a call to 2B which is O(md^2), we obtain an overall worst-case time complexity in O(mn^2d^3). As Algorithm 3B-1 does not require any additional data structure, its space complexity is the same as check2B-1, namely O(m). □

5.3 3B-2

3B-2 corresponds to the algorithm 6 that uses the function $check2B-2$ depicted by Algorithm 10. The idea is to improve the performance of the basic algorithm by recording and exploiting some information. For instance, when the consistency of a value must be checked again, it is inefficient to restart checking from scratch [7, 5]. Hence, we introduce three data structures:

- *initialized* is an array that indicates for any pair (X,a) whether the 2B-consistency of (X,a) has been checked at least one time,
- $minInferences$ is a three-dimensional array that allows recording for any triplet (X,a,Y) the value min(Y) in $2B(P|_{X=a})$,
- $maxInferences$ is a three-dimensional array that allows recording for any triplet (X,a,Y) the value max(Y) in $2B(P|_{X=a})$.

We will assume that *initialized* is an array whose elements are initially set to false (it does not appear in the given algorithm). Inferences with respect to a pair (X,a) are relevant only when $initialized[X, a]$ is equal to true. For instance, imagine that after achieving $2B(P|_{X=a})$, we obtain a network such that min(Y) $= c$ and max(Y) $= d$ (hence, dom(Y) $= \{c, \ldots, d\}$). Then, we set $initialized[X, a]$ to true, $minInferences[X, a, Y]$ to c and $maxInferences[X, a, Y]$ to d.

When running $check2B - 2$ (Algorithm 10), recorded information is first exploited (line 2) by a call to $exploitInferences$. After exploitation of recorded

Algorithm 8 exploitInferences(X : Variable, a : Value) : Set of variables

```
1: if ¬initialized[X, a] then
2:    return {X}
3: S ← ∅
4: for each Y ∈ 𝒳 do
5:    min(Y) ← max(min(Y),minInferences[X,a,Y])
6:    max(Y) ← min(max(Y),maxInferences[X,a,Y])
7:    if min(Y) > minInferences[X,a,Y] or max(Y) < maxInferences[X,a,Y] then
8:       add Y to S
9: end for
10: return S
```

Algorithm 9 recordInferences(X : Variable, a : Value)

```
1: initialized[X,a] ← true
2: for each Y ∈ 𝒳 do
3:    minInferences[X,a,Y] ← min(Y)
4:    maxInferences[X,a,Y] ← max(Y)
5: end for
```

Algorithm 10 check2B-2($P = (\mathscr{X}, \mathscr{C})$: CN, X : Variable, a : Value) : boolean

```
1: P_store ← P
2: S ← exploitInferences(X,a)
3: if S = ∅ then
4:    consistent ← true
5: else
6:    consistent ← 2B(P|_{X=a}, S) ≠ ⊥
7:    if consistent then
8:       recordInferences(X,a) then
9: end if
10: P ← P_store
11: return consistent
```

inferences, either the 2B-consistency of (X,a) still holds (line 4) as the empty set is returned by $exploitInferences$ (since no value in $2B(P|_{X=a})$ has been removed), or we have to check the 2B-consistency of (X,a) from the set S of variables whose domain has been reduced (line 6). Inferences are updated (line 8) by a call to the function $recordInferences$.

The function $exploitInferences$ (Algorithm 8) returns either the singleton $\{X\}$ (line 2) if the 2B-consistency of (X,a) has never been checked or the set of variables whose domain has lost a value which does not belong to (the last achievement of) $2B(P|_{X=a})$ (line 8). The function $recordInferences$ (Algorithm 9) just updates data structures.

The algorithm described here can be seen as a bound adaptation of the SAC-OPT algorithm proposed in [5] and also as a variant of the optimized 3B algorithm described in [7].

Proposition 3. *Applied to binary constraint networks, Algorithm 3B-2 admits a worst-case time and space complexity in $O(mnd^3)$ and $O(n^2)$, respectively.*

Proof. By storing information and avoiding unnecessary computation [7], Algorithm 3B-2 exploits the incrementality of arc consistency [5]. It means that the $2 * n$ potential successive calls to check2B-2 wrt a value (X,a) is in $O(md^2)$. Functions $exploitInferences$ and $recordInferences$ are in $O(n)$ and then can be ignored. As there are nd values, the overall worst-case time complexity is $O(mnd^3)$. It is possible to modify the data structures in order to keep only storage wrt two bounds per variable. With this slight modification (not proposed here due to lack of space), we obtain $2 * n$ values to be recorded for $2 * n$ current bounds. Hence, we obtain $O(n^2)$. □

6 3B+ (Bound singleton propagated arc consistency)

Finally, we propose an improvement of the algorithm 3B-2 presented above. Indeed, it is possible to benefit from some inferences when exploiting recorded information. For example, we know that if $\min(X) > a$ in $2B(P|_{Y=min(Y)})$ then we can infer that $Y > \min(Y)$ in $2B(P|_{X=a})$. By inserting the following instructions between lines 6 and 7 of Algorithm 8:

if $initialized[Y, min(Y)] \wedge minInferences[Y, min(Y), X] > a$ **then**
    $\min(Y) \leftarrow \min(Y) + 1$
if $initialized[Y, max(Y)] \wedge maxInferences[Y, max(Y), X] < a$ **then**
    $\max(Y) \leftarrow \max(Y)$ - 1

we obtain an algorithm which corresponds to an extension of the 3B-consistency and which can be seen as a coarse relaxation of bound SPAC.

7 Experiments

To prove the practical interest of the properties introduced in this paper, we have implemented the different algorithms described in the previous sections and conducted an experimentation with respect to some scheduling and frequency assignment instances. Algorithms 2B, 2B+, 3B-1, 3B-2, 3B-2+ correspond to the descriptions given in previous sections while 3B-1d is a dichotomic version (not described here) of 3B-1 related to the procedure $ref\_filtering$ in [19].

First, we have searched to establish a comparison between all algorithms wrt a set of 20 job-shop scheduling instances generated using the model of Taillard [23] by fixing 8 jobs and 8 machines. For each instance, a lower bound LB as well as the optimal makespan OPT have been computed. We have considered different sets of unsatisfiable CSP instances by setting different time windows between LB and OPT (only point where instances are in fact satisfiable). Figure 1 shows the proportion of instances that have been proved to be unsatisfiable at each variation x of the time window when establishing different consistencies. For example, at $x = 0$, the time window considered is (from 0 to) LB while at $x = 1$, it is (from 0 to) OPT. One can observe that 3B (and 3B+) allow a level of filtering which is sufficient to detect the inconsistency of most of the instances, unlike 2B+ and, especially unlike 2B and AC which behave similarly. Figure 2 shows the effort, in terms of number of constraint checks, required to establish the different consistencies (similar results are obtained when considering cpu times). Quite naturally, the more filtering a consistency is, the more costly it is. One can notice that the dichotomic variant of 3B outperforms the other ones.

Next, we have considered the real-world instances of the fullRLFAP archive. Table 1 shows the results obtained on some selected instances. Here, it clearly appears that AC is the best approach wrt these instances both in terms of cpu and of filtering (#rmvs denote the number of detected inconsistent values). It can be explained by the fact that frequency assignment instances are less favorable to bound consistencies than scheduling ones (which involve many precedence constraints). One can also notice that 2B+ allows a slight improvement of filtering wrt 2B (see $graph4$ and $graph10$) unlike 3B-2+ wrt 3B-2.

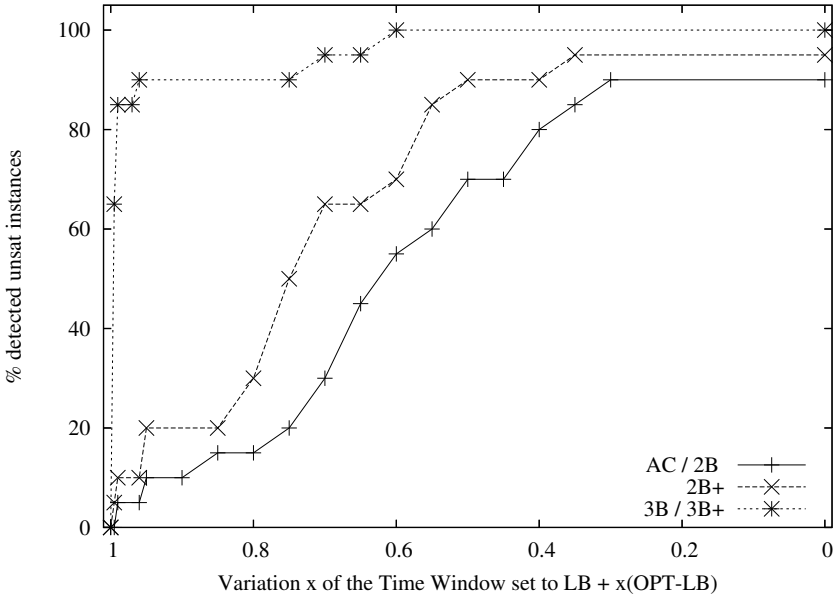


Fig. 1. Proportion of detected unsat instances when establishing consistencies

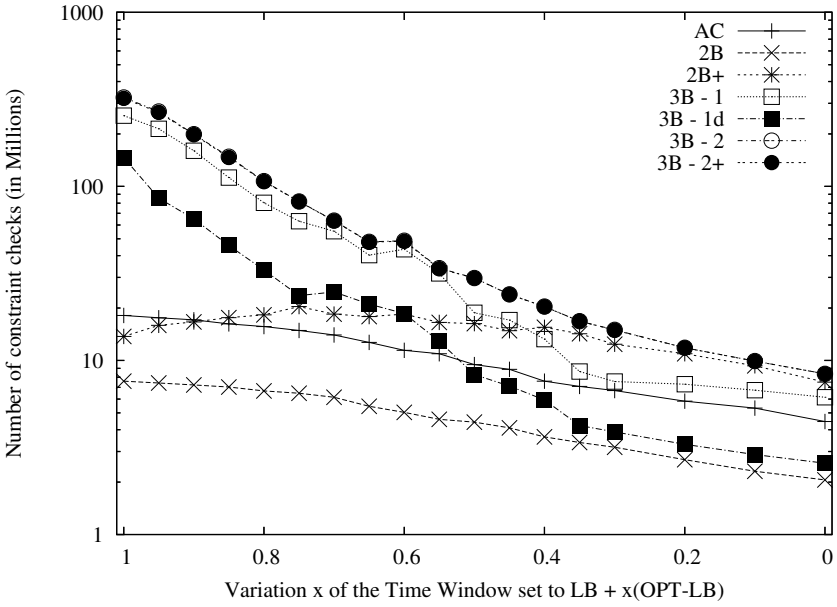


Fig. 2. Number of performed constraint checks when establishing consistencies

		AC	2B	2B+	3B - 1	3B - 1d	3B - 2	3B - 2+
*graph*4	cpu	0.47	0.1	2.23	8.29	10.25	34.26	44.4
	#rmvs	776	0	187	411	411	411	411
*graph*10	cpu	0.86	0.15	4.15	11.87	13.42	32.75	42.24
	#rmvs	386	0	46	122	122	122	122
*graph*14-*f*27	cpu	0.43	0.16	1.93	3.25	2.48	3.32	5.76
	#rmvs	2,314	0	0	0	0	0	0
*graph*14-*f*28	cpu	0.43	0.16	2.1	5.81	4.72	4.56	6.73
	#rmvs	3,230	0	0	2	2	2	2
*scen*02-*f*25	cpu	0.14	0.07	0.55	0.58	0.52	0.58	0.75
	#rmvs	106	0	0	0	0	0	0
*scen*11-*f*8	cpu	0.55	0.13	2.7	3.27	2.95	3.33	4.16
	#rmvs	4,992	0	0	0	0	0	0
*scen*11-*f*10	cpu	0.51	0.21	4.11	2.98	2.66	3.12	4.24
	#rmvs	6,324	3,024	3,024	3,024	3,024	3,024	3,024

Table 1. Establishing consistencies on RLFAP instances

In a second stage, we have searched to maintain all consistencies during the search of a solution. We have first studied the 10 open-shop scheduling instances with 7 jobs and 7 machines described in [23]. For each instance, we have searched to reach the optimal makespan in less than 300 seconds by using a branch and bound approach while exploiting constraint propagation. Table 2 gives the average relative distance, as well as the standard deviation, between the optimal makespan and the makespan found by the solver for different filtering algorithms. The results clearly show that 2B is the worst approach while 3B is the best one. In particular, 3B-1d and 3B-2+ have the best behaviour. We have also again considered the 20 job-shop scheduling instances already described above. Table 3 presents the average time (cpu) required to reach the optimal makespan and the proportion of instances that have been detected as unsatisfiable in less than 300 seconds with a time window fixed to $OPT-1$. Once again, Maintaining 3B is the best approach.

Table 4 shows the results (time-out has been set to 900 seconds) obtained when solving the selected RLFAP instances mentioned above. On some difficult instances, it is interesting to note that maintaining 2B is the quickest approach while maintaining a stronger consistency is always penalizing.

Finally, we must remember that all algorithms are based on AC3. We believe that using a more sophisticated foundation such as AC2001/3.1 [6] or AC3.2/3.3 [17] to establish 2B or 3B will not change the results a lot (but using $AC3_d$

	AC	2B	3B - 1	3B - 1d	3B - 2	3B - 2+
Average Distance	4.79%	28.6%	4.28%	3.00%	3.32%	3.30%
Standard Deviation	3.5%	8.2%	2.4%	2.2%	2.4%	1.8%

Table 2. Maintaining consistencies on Taillard's 7x7 open-shop instances

	AC	2B	2B+	3B - 1	3B - 1d	3B - 2	3B - 2+
Average cpu (TW = OPT)	212	216	282	88	73	117	117
% detected unsat (TW = OPT-1)	45%	55%	30%	85%	85%	85%	85%

Table 3. Maintaining consistencies on 8x8 job-shop instances

		AC	2B	2B+	3B - 1	3B - 1d	3B - 2	3B - 2+
*graph*04	cpu	2.11	*timeout*	*timeout*	260	314	391	678
*graph*10	cpu	8.18	*timeout*	14.27	*timeout*	*timeout*	*timeout*	*timeout*
*graph*14-*f*27	cpu	6.34	*timeout*	*timeout*	*timeout*	847	*timeout*	*timeout*
*graph*14-*f*28	cpu	40.73	8.11	20.48	347.57	435.91	*timeout*	*timeout*
*scen*02-*f*25	cpu	4.12	2.86	43.46	47.84	43.61	49.95	159.1
*scen*11-*f*8	cpu	115.26	74.62	98.86	*timeout*	*timeout*	*timeout*	*timeout*
*scen*11-*f*10	cpu	6.69	6.4	15.59	388.28	372.71	652.25	*timeout*

Table 4. Maintaining consistencies on RLFAP instances

[24] could be worthwhile). Indeed, we know for example that establishing 2B remains O(md^2) even if it is based on an optimal arc consistency algorithm. Further, our preliminary tests have confirmed this prediction. Nevertheless, 2B+ is one consistency that could benefit from such sophistication since many path consistency checks could be avoided.

8 Conclusion

The modest contribution of this paper is to establish a, hopefully clearer, connection between domain filtering consistencies, taken from the discrete CSP model, and bound consistencies, taken from the continuous CSP model. In particular, we have studied bound versions of well-known domain filtering consistencies.

The great advantage of using bound consistencies is that space requirement can be very limited, especially when domains are convex. For some discrete CSP instances with very large domains, it can be the only realistic approach. On the other hand, when space saving is not mandatory, worst-case time complexities of establishing bound consistencies wrt discrete instances (for which, no constraint semantics is available) are rather disappointing. For instance, in this context, the worst-case time complexity of establishing 2B (i.e. bound AC) is similar to the one of establishing AC. From a practical point of view, using bound consistencies is a good approach when dealing with problems which involve "bound-oriented" constraints such as precedence constraints. But, in this case, it is often possible to adopt a specific filtering by exploiting constraint semantics and also obtain a better complexity. In a less favorable context, our experimental results from some frequency assignment problems does not show an overall real advantage of using bound consistencies wrt arc consistency. However, we believe that bound consistencies could play a role in the development of methods for controlling the effort required to maintain a strong consistency during search.

References

1. R. Bartak and R. Erben. A new algorithm for singleton arc consistency. In *Proceedings of FLAIRS'04*, 2004.
2. F. Benhamou, D. MacAllester, and P. Van Hentenryck. CLP(Intervals) revisited. In *Proceedings of ILPS'94*, pages 124–138, 1994.
3. P. Berlandier. Improving domain filtering using restricted path consistency. In *Proceedings of IEEE-CAIA'95*, 1995.
4. C. Bessière and R. Debruyne. Theoretical analysis of singleton arc consistency. In *Proceedings of ECAI'04 workshop on modelling and solving problems with constraints*, pages 20–29, 2004.
5. C. Bessière and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI'05*, pages 54–59, 2005.
6. C. Bessiere, J.C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.
7. L. Bordeaux, E. Monfroy, and F. Benhamou. Improved bounds on the complexity of kB-consistency. In *Proceedings of IJCAI'01*, pages 303–308, 2001.
8. P. Codognet and D. Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.
9. H. Collavizza, F. Delobel, and M. Rueher. Comparing partial consistencies. *Reliable computing*, 5:213–228, 1999.
10. R. Debruyne and C. Bessière. From restricted path consistency to max-restricted path consistency. In *Proceedings of CP'97*, pages 312–326, 1997.
11. R. Debruyne and C. Bessière. Some practical filtering techniques for the constraint satisfaction problem. In *Proceedings of IJCAI'97*, pages 412–417, 1997.
12. R. Debruyne and C. Bessière. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.
13. R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.
14. B. Faltings. Arc consistency for continuous variables. *Artificial Intelligence*, 65:363–376, 1994.
15. P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation and evaluation of the constraint language cc(FD). *Journal of Logic Programming*, 37(1-3):139–164, 1998.
16. E. Hyvonen. Constraint reasoning based on interval arithmetic: the tolerance approach. *Artificial Intelligence*, 58:71–112, 1992.
17. C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *Proc. of CP'03*, pages 480–494, 2003.
18. C. Lecoutre and S. Cardon. A greedy approach to establish singleton arc consistency. In *Proceedings of IJCAI'05*, pages 199–204, 2005.
19. O. Lhomme. Consistency techniques for numeric csps. In *Proceedings of IJCAI'93*, pages 232–238, 1993.
20. O. Lhomme. *Contribution à la résolution de contraintes sur les réels par propagation d'intervalles*. PhD thesis, Université de Nice-Sophia Antipolis, 1994.
21. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
22. P. Prosser, K. Stergiou, and T. Walsh. Singleton consistencies. In *Proceedings of CP'00*, pages 353–368, 2000.
23. E. Taillard. Benchmarks for basic scheduling problems. *European journal of operations research*, 64:278–295, 1993.
24. M.R.C. van Dongen. $AC3_d$ an efficient arc consistency algorithm with a low space complexity. In *Proceedings of CP'02*, pages 755–760, 2002.

Maintaining Probabilistic Arc Consistency*

Deepak Mehta and M.R.C. van Dongen

Boole Centre for Research in Informatics/Cork Constraint Computation Centre

Abstract. The two most popular backtrack algorithms for solving Constraint Satisfaction Problems (CSPs) are Forward Checking (FC) and Maintaining Arc Consistency (MAC). MAC maintains full arc consistency while FC, during search, maintains a limited form of arc consistency. Previous work has shown that there is no single champion algorithm: MAC is more efficient on sparse problems which are tightly constrained, but FC has an increasing advantage as problems become dense and constraints loose. Ideally a good search algorithm should find the right balance—for any problem—between visiting fewer nodes in the search tree and reducing the work that is required for detecting inconsistent values. In order to do so, we propose to maintain *probabilistic arc consistency*. The idea is to assume the existence of a support, skip the process of seeking a support, if the probability of having some support for a value is at least equal to, some, carefully chosen, stipulated bound. Experimental results show that the probabilistic approach performs well on both sparse and dense problems and in fact better than MAC and FC on the hardest problems in the phase transition.

1 Introduction

Many problems in artificial intelligence can be formulated as Constraint Satisfaction Problems (CSPs). For the purpose of this paper, we only consider binary CSPs. However, the ideas presented in this paper can be generalised to other kinds of CSP and to other kinds of probabilistic consistency. Local consistency algorithms are used to reduce the search space of CSPs. Maintaining local consistency during search reduces the thrashing behaviour of the backtrack algorithm, which usually fails many times as a result of the same local inconsistencies. The two most popular backtrack algorithms that maintain consistency during search are MAC [14] and FC [9]. MAC maintains full arc consistency during search. It ensures that each value in the domain of *each* variable is supported by at least one value in the domain of every variable by which it is constrained. FC maintains a limited form of arc consistency. It ensures that each value in the domain of each *future* variable is FC *consistent*, i.e. supported by the value assigned to every past and current variable by which it is constrained.

MAC applies a stronger form of propagation than FC. Therefore, it usually visits fewer nodes in the search tree compared to FC. However, visiting fewer nodes at the expense of more constraint propagation may not always help in solving the problem more quickly. Previous work [8, 7, 3] has shown that MAC is more efficient than FC on sparse problems which are tightly constrained but FC has an increasing advantage as problems

* This work has received support from Science Foundation Ireland under Grant No. 00/PI.1/C075.

become dense and constraints become loose. In particular, it has been observed that MAC is usually better in terms of checks and time for hard *sparse* problems but FC is usually better in terms of checks and time for hard *dense* problems. For difficult problems the relationship between sparsity and tightness and between density and looseness roughly allows us to say that hard loose problems are better solved with FC, whereas hard tight problems are better solved with MAC. The reason why FC performs better than MAC for hard dense problems is that it exploits a common sense probabilistic argument: *the looser the constraints (the denser the hard problem), the higher the probability that* FC *consistency is tantamount to arc consistency*. MAC's spending checks to prove that this actually holds translates to a penalty in solution time.

There is no single champion algorithm which performs well on all types of problems [8, 3]. Ideally, a good search algorithm should find the right balance—for any problem—between visiting fewer nodes in the search tree and reducing the work that is required for detecting and removing inconsistent values. More specifically, for hard *dense* problems a good search algorithm should keep the best features of MAC and FC by staying closer to MAC in terms of the number of visited nodes and closer to FC in terms of checks while for hard *sparse* problems it should behave like MAC. In order to do so, we propose to maintain *probabilistic arc consistency* during search. The idea is to assume the existence of a support (skip the process of seeking a support) if the probability of having some support for a value is at least equal to some, carefully chosen, stipulated threshold.

Arc consistency involves revisions of domains, which requires support checks to detect and remove unsupported values from the domain of a variable. In many revisions, *some* or *all* values find some support. For example, when RLFAP #11 is solved using MAC-3 or MAC-2001, 83% of the total revisions are *ineffective*, i.e. they cannot delete any value. If we can predict the existence of a support with a high probability and avoid the process of seeking a support when the probability is at least equal to some, carefully chosen, stipulated bound then a considerable amount of work can be saved.

We first show how to compute the probability of having some support for a value. Next, we introduce the notion of a *Probabilistic Support Condition* (PSC). The PSC holds if and only if this likelihood is above our threshold. If the PSC holds then we assume that a support exists and we will not seek it. Further, we introduce a coarser condition called a *Probabilistic Revision Condition* (PRC). The PRC holds if and only if for each value in a domain the probability of having some support is above the threshold. If the PRC holds then we assume that a support exists for each value and we will avoid the corresponding revision. Experimental results show that the probabilistic approach performs well on sparse *and* dense problems and in fact better than MAC and FC on the hardest problems in the phase transition.

The remainder of this paper is organised as follows: Section 2 gives a brief introduction to constraint satisfaction and discusses the arc consistency algorithm AC-3. Section 3 discusses related work. Section 4 explains the notion of a probabilistic support condition (PSC) and the notion of a probabilistic revision condition (PRC). Section 4 describes how to integrate PSC and PRC in the coarse-grained algorithm AC-3. Experimental results are presented in Section 6, followed by a discussion in Section 7. Conclusions are presented in Section 8.

2 Background

A *Constraint Satisfaction Problem* is defined as a set $\mathcal{V}$ of n variables, a non-empty domain $D(x)$ for each variable $x \in \mathcal{V}$ and a set of e constraints among subsets of variables of $\mathcal{V}$. A binary constraint C_{xy} between variables x and y is a subset of the Cartesian product of $D(x)$ and $D(y)$ that specifies the allowed pairs of values for x and y. We only consider CSPs whose constraints are binary. A value $b \in D(y)$ is called a *support* for $a \in D(x)$ if $(a, b) \in C_{xy}$. Similarly $a \in D(x)$ is called a support for $b \in D(y)$ if $(a, b) \in C_{xy}$. A *support check* (consistency check) is a test to find if two values support each other. A value $a \in D(x)$ is called *viable* if for every variable y constraining x the value a is supported by some value in $D(y)$. A CSP is called *arc consistent* if for every variable $x \in \mathcal{V}$, each value $a \in D(x)$ is viable.

Coarse-grained arc consistency algorithms such as AC-3 [10], AC-2001 [1], and AC-3_d [16] are efficient when it comes to transforming a CSP to its arc-consistent equivalent. They use *revision ordering heuristics* to select an arc from a data structure called a queue (a set really). When an arc, (x, y), is selected from the queue, $D(x)$ is *revised* against $D(y)$. Here to *revise* $D(x)$ against $D(y)$ means removing all values from $D(x)$ that are not supported by any value of $D(y)$. Revision ordering heuristics [17, 2, 16] can influence the efficiency of arc consistency algorithms.

Pseudo-code for AC-3 equipped with *reverse variable-based* [12] revision ordering heuristics is depicted in Figure 1. The revise function upon which AC-3 depends is depicted in Figure 2. Reverse variable based revision ordering heuristics first select a variable x and repeatedly select arcs of the form (x, y) to determine the next revision until there are no more such arcs or $D(x)$ becomes empty. Selecting a variable x and revising it against all its neighbours y such that (x, y) is currently present in the queue, we call a *complete relaxation* of x. In Figure 1, if $D(x)$ was changed after a complete relaxation and if this was the result of *only one* effective revision (*effective_revisions* $= 1$), which happened to be against $D(y'')$, then all the arcs of the form (y', x) where y' is a neighbour of x and $y' \neq y''$ are added to the queue. However, if $D(x)$ was changed as the result of *more than one* effective revision (*effective_revisions* > 1) then *all* the arcs of the form (y', x) where y' is a neighbour of x are added to the queue. Modulo constraint propagation effects this saves work for maintaining the queue compared to the original version [10] of the algorithm.

For the purpose of this paper, before starting search all search algorithms transform the input CSP to its arc consistent equivalent. During backtrack search, variables are chosen in some order and each is instantiated with a value from its domain. MAC maintains arc consistency after each variable assignment. MAC-*x* uses AC-*x* for maintaining arc consistency during search. To re-establish arc consistency following the instantiation of a variable x the queue is initialised to all arcs incident to x. More specifically, all arcs of the form (y, x) are added to the queue where y is a future variable constrained by x. When values are deleted, more arcs may have to be added to the queue to determine if these deletions lead to further deletions.

FC can be considered as a degenerate form of MAC. After instantiating a variable x in FC, the queue is initialised to all arcs of the form (y, x). Here arcs are never added to the queue, not even after an effective revision. Note that the original version of FC does not transform the input CSP to its arc-consistent equivalent prior to search.

```
Function AC-3: Boolean;
begin
  Q := G
  while Q not empty do begin
    select any v from {x : (x, y) ∈ Q }
    effective_revisions := 0
    for each y such that (x, y) ∈ Q do
      remove (x, y) from Q
      revise(x, y, change_x)
      if D(x) = ∅ then
        return False
      else if change_x then
        effective_revisions := effective_revisions + 1
        y'' := y;
    if effective_revisions = 1 then
      Q := Q ∪ { (y', x) ∈ G : y' ≠ y''}
    else if effective_revisions > 1 then
      Q := Q ∪ { (y', x) ∈ G}
  return True;
end;
```

Fig. 1. AC-3

```
Function revise(x, y, var change_x)
begin
  change_x := False
  for each a ∈ D(x) do
    if ∄b ∈ D(y) such that b supports a then
      D(x) := D(x) \ { a }
      change_x := True
end;
```

Fig. 2. Algorithm $revise$ of AC-3

The *density* p_1 of a CSP is defined as $2\,e/(n^2 - n)$ where e is the number of constraints and n is the number of variables. The *tightness* p_2 of the constraint C_{xy} between the variables x and y is defined as $1 - |\,C_{xy}\,|/|\,D(x) \times D(y)\,|$. The *degree* of a variable is the number of constraints involving that variable. Before starting search the input CSP is transformed to its arc consistent equivalent. The *original* domain of a variable is the domain of that variable in this arc consistent equivalent. For the remainder of this paper for any variable x, we use $D(x)$ for the current domain of x and $D_o(x)$ for the original domain of x. The *directed constraint graph* of a given CSP is a directed graph having an arc (x, y) for each combination of two mutually constraining variables x and y. We will use G to denote the directed constraint graph of the input CSP.

3 Related work

In this section, we discuss some work which is related to finding the right balance between the effort required for constraint propagation and that required for search.

First, let us mention the work of [5] where a class of algorithms, termed *selective relaxation*, is described. In particular, constraint propagation is restricted based on two local criteria, which are the distance of the variable from the variable which is instantiated (distance-bounded) and the proportion of values deleted (response-bounded). Chmeiss and Sais [3] present a backtrack search algorithm, MAC(*dist$_k$*), that also uses a distance parameter k as a bound to maintain a partial form of arc consistency.

El Sakkout, Wallace and Richards [4] introduce anti-functional reduction (AFR). AFR can be viewed as an improvement to AC-4. It reduces some propagation in AC-4 which helps to minimise the amount of backtrack recording and restoration. However, AFR is specific to the fine-grained algorithm AC-4, whose inefficiency lies in its space complexity $\mathcal{O}(e\,d^2)$ and the necessity of maintaining huge data structures during search.

The traditional approach to find if $a \in D(x)$ (also denoted (x, a)) is supported by y is to *identify* some $b \in D(y)$ that supports (x, a), which usually results in a sequence

of support checks. Identifying the support is more than is needed to guarantee that a value is supportable: knowing that a support exists is enough. Most arc consistency algorithms proposed so far put a lot of effort in identifying a support to confirm the existence of a support. To reduce the task of identifying a support up to some extent, the notions of a *support condition* and a *revision condition* are introduced in [12] (see also [2]). A support condition (SC) guarantees that a value has *some* support while revision condition (RC) guarantees that *all* values have *some* support without identifying it. In the following paragraph we present a special version of SC and RC which facilitates the introduction of their probabilistic equivalents, which are to be presented in the following section.

Let C_{xy} be the constraint between x and y, let $a \in D(x)$, and let $R(y) = D_o(y) \setminus D(y)$ be the removed values from the original domain of y. The *support count* of (x, a) with respect to y, denoted *sc*$(\, x, y, a\,)$, is the number of values in $D_o(y)$ supporting a. Note that $|\, R(y)\,|$ is an upper bound on the number of lost supports of (x, a) in y. Therefore, if the following condition holds then (x, a) is supported by y:

$$sc(x, y, a) > |\, R(y)\,|. \tag{1}$$

For instance, if $a \in D(x)$ has 3 supports in $D_o(y)$ and 2 values are removed from $D_o(y)$, i.e. $|\, R(y)\,| = 2$ then (x, a) has at least one support in $D(y)$. Hence, there is no need to seek support for a in $D(y)$. The condition in Equation (1) is a (special version of what is called a) *Support Condition* (SC) in [12]. SCs help avoiding many (but not necessarily all) sequences of support checks eventually leading to a support.

For a given arc, (x, y), the *support count* of x with respect to y, denoted $sc(x, y)$, is defined by $sc(x, y) = \min(\{sc(x, y, a) : a \in D(x)\})$. Therefore, if the following condition holds then any value in $D(x)$ is supported by y:

$$sc(\, x, y\,) > |\, R(y)\,|\,. \tag{2}$$

The condition in Equation (2) is a (special version of what is called a) *Revision Condition* (RC) in [12]. RCs avoid many (but not all) unnecessary revisions and much of queue maintenance overhead. The basic idea presented in [12] is to avoid looking for a support if the SC holds and to avoid a candidate revision if the RC holds. Independent work by [2] has proposed a static version of RC where $sc(x, y)$ is the least support count of the values in $D_o(x)$ as opposed to $D(x)$.

4 Probabilistic Approach

Even if the support condition and revision condition are used they do not always make MAC solve more quickly than FC. We propose a probabilistic approach to achieve this. We generalise the notions of a support condition and a revision condition to the notions of a *probabilistic support condition* (PSC) and a *probabilistic revision condition* (PSC). The idea is to assume that a support exists (avoid the process of seeking a support) if the probability of having some support for a value is relatively high. Similarly, if the probability of having some support is relatively high for *each* value in a domain then we avoid the corresponding revision.

4.1 Probabilistic Support Condition

Let $Ps_{(x,y,a)}$ be the probability that (x, a) has some support in $D(y)$. Then

$$Ps_{(x,y,a)} = 1 - \binom{|\,R(y)\,|}{sc(x,y,a)} \Big/ \binom{|\,D_o(y)\,|}{sc(x,y,a)}. \tag{3}$$

The justification for this equation is that its right hand side is equal to the probability that none of the $\binom{|\,R(y)\,|}{sc(x,y,a)}$ subsets of size $sc(\,x,y,a\,)$ of $R(y)$ contain all supports of (x, a). To calculate $Ps_{(x,y,a)}$, we assume any combination of size $sc(\,x,y,a\,)$ of $D_o(y)$ is equally likely to occur. Note that if the SC is satisfied, i.e. $sc(x, y, a) > |\,R(y)\,|$, then Equation (3) reduces to $Ps_{(x,y,a)} = 1$. Indeed, if fewer values have been removed from the initial domain than there were supports in the original domain then the probability that a support exists is equal to 1.

We now introduce a probabilistic version of the support condition. Let T be some desired threshold. If $Ps_{(x,y,a)} \geq T$ then (x, a) will have support with y with a probability of T or more. We call this condition a *Probabilistic Support Condition*. If it holds then we avoid seeking support for a.

4.2 Probabilistic Revision Condition

Remember that the support count of x with respect to y is denoted $sc(x, y)$. It is the least support count of the values of $D(x)$ with respect to y. Similar to the definition of a probabilistic support condition, we now define a probabilistic revision condition. To this end let $Ps_{(x,y)}$ be the least probability that some value in $D(x)$ is supported by y. Note that for any value $a \in D(x)$, we immediately have

$$Ps_{(x,y)} = 1 - \binom{|\,R(y)\,|}{sc(x,y)} \Big/ \binom{|\,D_o(y)\,|}{sc(x,y)} \leq Ps_{(x,y,a)}. \tag{4}$$

Let T be some threshold. If $Ps_{(x,y)} \geq T$ then, each value in $D(x)$ is supported by y with a probability of T or more. We call the condition $Ps_{(x,y)} \geq T$ a *Probabilistic Revision Condition* (PRC). If it holds then we skip the revision of $D(x)$ against $D(y)$. Note that when $sc(x, y) > |\,R(y)\,|$, then $\binom{|\,R(y)\,|}{sc(x,y)} = 0$ and, with a probability of 1, all values in $D(x)$ are supported by y.

5 Description of the new algorithm

In the previous section, we introduced the notion of a *probabilistic support condition* (PSC) and the notion of a *probabilistic revision condition* (PRC) to determine when to seek support for a given arc-value pair and when to consider an arc for a revision. However, satisfying PSC or PRC does not always guarantee the existence of a support. As a consequence, they may not always allow us to achieve full arc consistency. This may leave more inconsistent values in the domains of the variables and we can expect more nodes in the search tree. However, this needs not necessarily be less efficient. Even though there are more nodes in the tree, more visited nodes may be the result of

less work done at each node. More visited nodes with fewer support checks per node may result in fewer support checks for the overall search tree.

In order to use PSC and PRC, the support count for each combination of arc (x, y) and value $a \in D(x)$, i.e. $sc(\, x, y, a\,)$ must be computed prior to search. Once these support counts are computed, unlike AC-4 where they are decremented during search, they remain static. Hence, there is no overhead of maintaining them. The pseudo-code for computing the support count for each arc-value pair is depicted in Figure 4. In the algorithm, *last*$(\, x, y, a\,)$ gives AC-2001's last known support for (x, a) in y. Note that the algorithm does not repeat checks and uses the bidirectional property of constraints.

PSC and PRC can be incorporated into any coarse-grained arc consistency algorithm. Figure 3 depicts the result of incorporating them into AC-3. We call this algorithm PAC-3 (Probabilistic AC-3). If PRC holds then it can be exploited either *after* selecting the arc (x, y) for the next revision or when arcs are added to the queue. In the former case the corresponding revision is not carried out and in the latter case (x, y) is not added to the queue. We will use the PRC by tightening the condition for adding arcs to the queue: arcs should only be added if the PRC does not hold. This is depicted in Figure 3. The new revise function, which we call $revise_p$, uses PSC as shown in Figure 5. It avoids the process of seeking a support if $Ps_{(x,y,a)}$ is at least equal to the threshold.

We will now study conditions for the threshold T which will guarantee that, for any given problem, PAC-3 does at least the amount of constraint propagation which is carried out by FC. If during search any variable y is instantiated to a value then $|\, R(y)\,|$ becomes $|\, D_o(y)\,| - 1$. When considering the arc (x, y), to ensure FC consistency, we can trigger the probabilistic support condition by making T greater than the probability that for each value all its supports are in $R(y)$. This probability is at least $1-1/|\, D_o(y)\,|$.

In the implementation, which is depicted in Figure 3, the same threshold T was chosen for all arcs. The previous argument justifies the choice of any $T > 1 - 1/d_{max}$, where d_{max} is the maximum domain size of the variables.

Note that both PSC and PRC are presented in such a way that the idea is made as clear as possible. This should not be taken as the real implementation. Putting more effort into estimating the probability of having some support for each arc-value pair does not generally pay off in terms of the CPU time. There are a few ways to overcome this. Due to space restrictions we will only discuss one of them. By expanding Equation (3) and rearranging the terms, we derive the following less accurate estimate of $Ps_{(x,y,a)}$:

$$Ps_{(x,y,a)} = 1 - \sum_{i=0}^{sc(x,y,a)-1} \left(1 - \frac{|\, D(y)\,|}{|\, D_o(y)\,| - i}\right) \geq 1 - \left(1 - \frac{|\, D(y)\,|}{|\, D_o(y)\,|}\right)^{sc(x,y,a)}.$$

Substituting the right hand side of this inequality for $Ps_{(x,y,a)}$ into the probabilistic support condition $Ps_{(x,y,a)} \geq T$ an rearranging terms gives us the following condition:

$$|\, R(y)\,| \leq |\, D_o(y)\,| \times (1 - T)^{1/sc(x,y,a)}. \tag{5}$$

Note that this condition implies the exact PSC. It states that if the number of values removed from $D_o(y)$, i.e. $|R(y)|$ is less than the right hand side of Equation (5) then $Ps_{(x,y,a)}$ is at least equal to the threshold value T. The advantage of Equation (5) is that the right hand side is constant. It gives the critical value of $|R(y)|$ with respect to

```
Function PAC-3: Boolean;
begin
   Q := G
   Set threshold T such that 1 − 1/d_max < T ≤ 1.
   while Q not empty do begin
      select any v from {x : (x, y) ∈ Q }
      effective_revisions := 0
      for each y such that (x, y) ∈ Q do
         remove (x, y) from Q
         revise_p(x, y, change_x)
         if D(x) = ∅ then
            return False
         else if change_x then
            effective_revisions := effective_revisions + 1
            y'' := y;
      if effective_revisions = 1 then
         Q := Q ∪ { (y', x) ∈ G : y' ≠ y'', Ps_(y',x) < T }
      else if effective_revisions > 1 then
         Q := Q ∪ { (y', x) ∈ G : Ps_(y',x) < T }
   return True;
end;
```

Fig. 3. PAC-3

```
Function InitialiseSupportCounters ( )
begin
   call AC-2001
   if the problem is arc-consistent then
      for each (x, y) ∈ G do
         for each a ∈ D_o(x) do
            sc( x, y, a ) := 1
      for each (x, y) ∈ G such that x < y do
         for each a ∈ D_o(x) do
            for each b ∈ D_o(y)
                  such that b > last( x, y, a ) do
               if b supports a then
                  sc(x, y, a) := sc(x, y, a) + 1
                  sc(y, x, b) := sc(y, x, b) + 1
end;
```

Fig. 4. Initialisation of support counts

```
Function revise_p(x, y, var change_x)
begin
   change_x := False
   for each a ∈ D(x) do
      if Ps_(x,y,a) ≥ T then
         continue
      if ∄b ∈ D(y) such that b supports a then
         D(x) := D(x) \ { a }
         change_x := True
end;
```

Fig. 5. Algorithm $revise_p$

T above which PSC will always hold. Instead of recomputing the probability in every iteration, this critical value can be computed *prior* to search for each arc-value pair. During search if $|R(y)|$ is at least equal to this critical value then PSC holds and the process of seeking a support is skipped.

Since PAC-3 needs to be invoked at each node of the search tree, we call the new backtrack algorithm that maintains PAC-3 during search MPAC-3. The space complexity of using support counts is $\mathcal{O}(e\,d)$. The space complexity of storing the support count for each arc is $\mathcal{O}(e)$ but it may increase to $\mathcal{O}(e\,n)$ during search, since the support count of an arc may change during search. Therefore, the overall space complexity of MPAC-3 is $\mathcal{O}(e\,d + e\,n) = \mathcal{O}(e\,max(d, n))$.

6 Experimental Results

In this section, we shall present some results demonstrating the practical efficiency of MPAC-3 when compared to MAC-3 and FC. We experimented with random problems which were generated by Frost *et al.*'s model B generator [6]. In this model a random CSP instance is typically represented as $\langle\, n, d, p_1, p_2 \,\rangle$ where n is the number of variables, d is the uniform domain size, p_1 is the average density, and p_2 is the uniform tightness. For each combination of $\langle n, d, p_1, p_2 \rangle$, 100 random problems were generated and their mean performance is reported. The domain size d was kept at 10 which defines the range $(\,0.9, 1\,]$ for T, required by MPAC-3. Unless otherwise stated, the value of T

used is 0.95. Experiments were conducted in the same fashion as reported in [8] to study the behaviour of MPAC-3 in solving dense and sparse problems. The brief introduction about the experiments is as follows:

Different problem topologies Our main aim was to investigate how MPAC-3 behaves with different problem topologies, i.e. with sparse and dense problems. For the first set of experiments, which is described in Section 6.1, n was kept at 30. We varied the density p_1 in steps of 0.1 in the range $[\,0.1, \ldots, 1\,]$ and the tightness p_2 was varied in steps of 0.01 in the range as shown in Figures 6 and 7.

Sparse problems From the first set of experiments the results were not clear for sparse problems. In Section 6.2 we describe experiments which were conducted on sparse problems by maintaining the same average degree $2\,e/n$ whilst increasing the number of variables n. We investigated a class, which is known as *exceptionally hard* problems (EHPs) [15]. For these problems the average degree is kept as close as possible to 4.92. Problems were generated for domain sizes $n \in \{20, 35, 50, 65\}$.

Dense problems Finally, we wanted to see how MPAC-3 performs with hard dense problems where constraints are very loose. To see this we devised the following experiments, which are described in Section 6.3. We set the density to $p_1 = 1$ and varied n in steps of 5 in the range 25–45. Next the critical tightness p_2 was calculated at which the search effort can be expected to be maximum. For a given $\langle\, n, d, p_1 \rangle$, the average search effort can be expected to be maximum when p_2 is $1 - d^{-2/(p_1(n-1))}$ [13].

All algorithms were equipped with a dom/deg variable ordering with a lexicographical tie breaker. The variable ordering heuristic *minimum dom* proposed by Haralick and Elliot [9] is a prominent heuristic and has been proven efficient with forward checking search. However, in our experiments, we found that *dom/deg* was a better heuristic than the *minimum dom* for FC, where *dom* is the domain size and *deg* is the original degree of a variable. Note that due to the dynamic nature of *dom/deg* heuristic MAC-3, MPAC-3, and FC may follow different search paths. Algorithms were equipped with a reverse variable based revision ordering heuristic [11] *comp*.The experiments were carried out on a PC Pentium III having 256 MB of RAM running at 2.266 GHz processor with linux. All algorithms were written in C.

6.1 Different Problem Topologies

Figures 6 and 7 show the mean performance of MPAC-3, MAC-3, and FC in terms of total support checks and the nodes visited. As expected MAC-3 always outperforms FC in terms of the visited nodes while FC outperforms MAC-3 in terms of the number of support checks. Note that for dense problems when a peak occurs in the phase transition, MPAC-3 is more efficient compared to both MAC-3 and FC, when the effort is measured in terms of support checks. The same was observed for other high densities $(\,0.9, 0.8, 0.7, 0.6\,)$ for which results are not shown in the graph. The number of nodes visited by MPAC-3 is closer to MAC-3 than FC. The gap between MAC-3 and FC in terms of the visited nodes increases as the problems become sparse but it decreases between MAC-3 and MPAC-3. As the problems become sparse MAC-3 starts to perform better

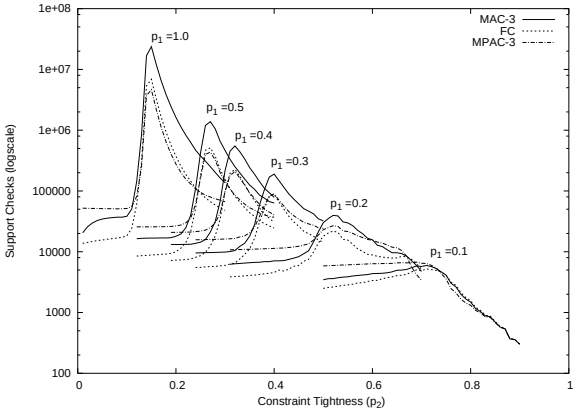


Fig. 6. Mean performance of algorithms for $\langle\, 30, 10, p_1, p_2 \,\rangle$ in terms of checks.

than FC, although this may not be clear from Figure 6. Results shown in Figures 8 and 9 confirm this. For easy problems MPAC-3 spends more support checks, what is caused by its computing of support counts prior to search. However, in terms of search effort it always outperforms both FC and MAC-3which is not visible in the graphs. Overall, unlike MAC-3 and FC, MPAC-3 performs well on average, both in terms of support checks and the nodes visited.

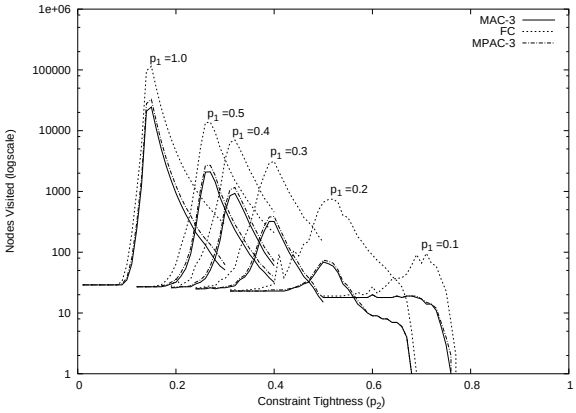


Fig. 7. Mean performance of algorithms for $\langle\, 30, 10, p_1, p_2 \,\rangle$ in terms of visited nodes.

6.2 Sparse Problems

Figures 8 and 9 show comparison of the algorithms on sparse difficult problems $\langle n, d, p_1 \rangle$ for checks and nodes visited. When the effort is measured in terms of checks, we see

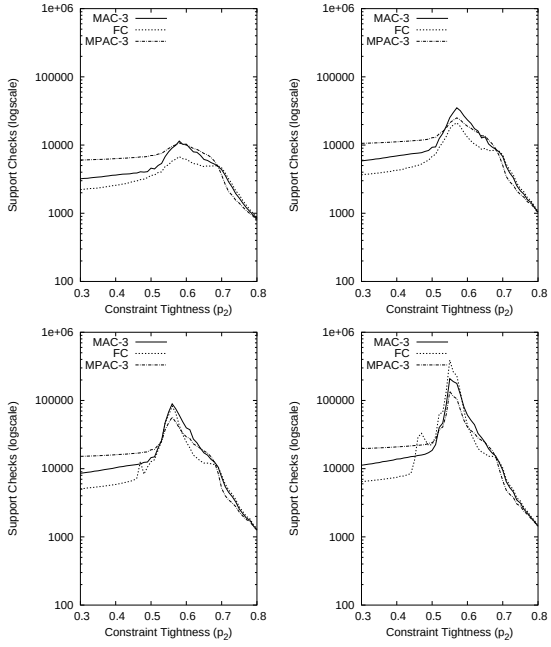


Fig. 8. Mean performance of algorithms in terms of checks for $\langle 20, 10, 0.25 \rangle$ (top left), $\langle 35, 10, 0.14 \rangle$ (top right), $\langle 50, 10, 0.10 \rangle$ (bottom left), and $\langle 65, 10, 0.07 \rangle$ (bottom right).

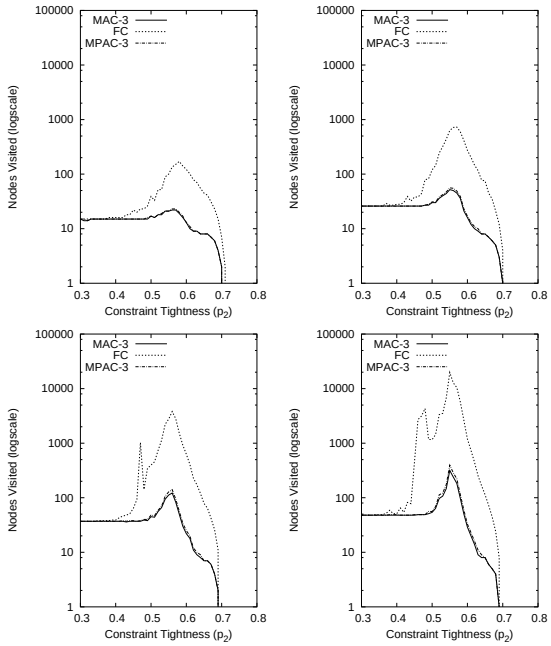


Fig. 9. Mean performance of algorithms in terms of visited nodes for $\langle 20, 10, 0.25 \rangle$ (top left), $\langle 35, 10, 0.14 \rangle$ (top right), $\langle 50, 10, 0.10 \rangle$ (bottom left), and $\langle 65, 10, 0.07 \rangle$ (bottom right).

Table 1. Comparison between FC, MAC-3, and MPAC-3 on sparse problems.

$\langle n, d, p_1, p_2 \rangle$	Algorithm	Checks	Time (seconds)	Revisions	Visited nodes
$\langle 65, 20, 0.08, 0.65 \rangle$	FC	24,088,980	5.04	2,531,453	739,054
	MAC-3	15,138,669	1.44	974,903	10,545
	MPAC-3	8,271,149	1.55	948,283	12,918
$\langle 90, 20, 0.07, 0.59 \rangle$	FC	1,375,383,859	360.81	187,037,415	47,250,225
	MAC-3	867,722,685	105.86	67,002,984	617,760
	MPAC-3	507,456,096	113.43	72,798,693	892,139

Table 2. Comparison between MPAC-3, MAC-3 and FC in terms of checks on dense problems.

$\langle n, d, p_1, p_2 \rangle$	MPAC-3	MAC-3	ratio	FC	ratio
$\langle$ 25,10,1,0.18 $\rangle$	878,319	4,077,197	4.64	1,171,866	1.33
$\langle$ 30,10,1,0.15 $\rangle$	4,641,200	23,673,088	5.10	6,883,315	1.48
$\langle$ 35,10,1,0.13 $\rangle$	21,342,919	116,125,948	5.44	34,339,035	1.60
$\langle$ 40,10,1,0.11 $\rangle$	169,780,372	899,363,792	5.29	288,667,462	1.70
$\langle$ 45,10,1,0.10 $\rangle$	662,418,341	3,728,725,077	5.62	1,194,686,264	1.80

that both MAC-3 and MPAC-3 perform poorly against FC, when n is 20 and 35. However, the gap decreases as n increases. When n is 50 they perform almost the same amount of work at the crossover point and at $n = 65$ MAC-3 actually outperforms FC for the hardest problems in the phase transition. At $n = 20$, MPAC-3 and MAC-3 perform the same amount of work at peak; at $n = 35$, MPAC-3 outperforms MAC-3 and when n is 50 and 65, it dominates both MAC-3 and FC for the hardest problems in the phase transition. If this trend continues then MAC-3 will dominate FC while MPAC-3 will dominate both algorithms as n is further increased. Note that in Figure 9 the gap between MAC-3 and FC in terms of visited nodes increases as n increases. Interestingly, the number of nodes visited by MPAC-3 and MAC-3 on average remains the same.

Table 1 presents results for difficult sparse problems for a constant domain size $d = 20$. This time the value of threshold was set to 0.951, since T should be greater than $1 - 1/20$. Notice that both MAC-3 and MPAC-3 outperform FC. Again MPAC-3 is the best when it comes to saving checks.

6.3 Dense Problems

The columns *ratio* in Table 2 represent how much MPAC-3 was better than MAC-3 and FC for checks. The order of magnitude by which MPAC-3 saves checks when compared to MAC-3 increases from 4.64 ($n = 25$) to 5.62 ($n = 45$). Similarly, the order of magnitude by which MPAC-3 outperforms FC increases from 1.33 ($n = 25$) to 1.80 ($n = 45$). It seems that the probabilistic approach becomes more and more efficient as n increases. The reason for this is that for a given domain size d and density p_1, the hardest problems in the phase transition will have a tightness p_2 which drops, as the number of variables increases. In other words, on average the constraints will become loose, on average the number of supports will increase, on average the problems will become more and more arc consistent, and on average MAC-3 will carry out more and more unnecessary ineffective revisions. In terms of the visited nodes, MAC-3 was the best algorithm. However, on average MPAC-3 was between FC and MAC-3 in terms of visited nodes. It only visited 1.36 times more nodes then MAC-3, whereas FC visited 4.7 times more nodes than MAC-3.

To convince the reader that MPAC-3 is really efficient, we also compared it against MAC-2001. Due to space restrictions results are not shown for all the experiments. Our experiments demonstrated that MAC-2001 was spending more time than other algorithms. Table 3 clearly demonstrates that MPAC-3 is again better in saving checks.

Table 3. Mean performance in terms of the number of checks, the solution time (seconds), the number of revisions, and the number of nodes visited for $\langle 45, 10, 1.0, 0.10 \rangle$

Algorithm	Checks	Time (seconds)	Revisions	Visited nodes
MAC-2001	1,142,906,628	1103.55	990,831,993	2,866,391
FC	1,194,686,264	576.64	354,836,954	13,601,483
MAC-3	3,728,725,077	1074.50	990,831,993	2,866,391
MPAC-3	662,418,341	677.69	411,383,748	4,039,925

We also conducted experiments by setting the value of T to 0.91. Not much difference was observed in the performance of MPAC-3 with sparse problems but with the dense problems MPAC-3 was saving more checks. We are currently investigating the effect of using different thresholds. Overall in our experiments MPAC-3 was better when it comes to saving checks (except for small problems which require almost no solution time) when compared to MAC-3, MAC-2001, and FC. MPAC-3 never lost against FC, MAC-3 and MAC-2001 in terms of time and visited nodes. For hard dense problems FC was better when it comes to saving time. On average MPAC-3 required 1.2 times more time than FC, whereas MAC-3 required 1.95 times more time. MAC-3 and MPAC-3 were solving hard sparse problems quickly on average compared to FC.

7 Discussion

Figure 10 shows graphically how the estimate of the likelihood of having some support in $D(y)$ for (x, a) with different values of support count vary with the number of values removed from $D_o(y)$. The original domain size of y is 10. Notice that when the number of supports of (x, a) are fewer the probability of having some support decreases rapidly as the number of values removed from y increases. This is not the same case when the number of supports are more. In that case, the probability of having some support remains relatively high for a while with respect to the number of values removed. This insight may help in understanding why MAC-3 performs poorly in terms of checks in solving hard dense problems where constraints are loose and why MPAC-3 is able to perform well on both hard dense and sparse problems.

For *hard dense problems* constraints are generally loose, and on average each arc-value pair has several supports. When the number of supports is high, the probability of having some support remains relatively high for a while with respect to the number of values removed. This explains why for dense problems, when MAC-3 propagates deletions, many times a value will find some support. Therefore, one can expect much ineffective propagation. For this reason the overhead of MAC-3 becomes very large for dense problems where constraints are loose. Contrary to that, FC, which does minimal propagation during search, avoids most ineffective propagation. FC visits more nodes because it leaves more unsupported values, but is still more efficient than MAC-3 when

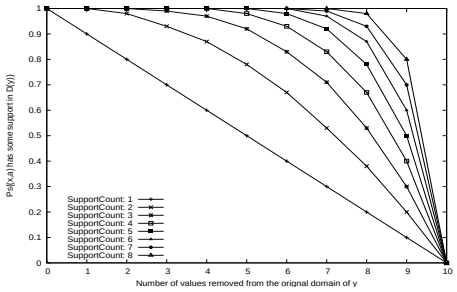


Fig. 10. $Ps_{(x,y,a)}$ versus $|R(y)|$.

the effort is measured in terms of checks. MPAC-3 also avoids much ineffective propagation using PSC and PRC. It propagates only when the probability of having some support falls below a stipulated bound. Unlike MAC-3 and FC where the strength of constraint propagation is *static*, MPAC-3 adjusts *dynamically* during search. MPAC-3 keeps the best features of MAC-3 and FC by staying closer to MAC-3 in terms of the number of visited nodes and FC in terms of checks.

For *hard sparse problems* constraints are generally tight, which means that on average each arc-value has *fewer* supports. When the number of supports is low, the probability of having some support drops rapidly with respect to the number of values removed. This forces PSC and PRC to fail quickly which in turn forces MPAC-3 to behave like MAC-3. When a deletion takes place both MAC-3 and MPAC-3 propagate immediately. This is the reason why they both visit almost the same number of nodes on average. The probabilistic approach allows to increase or decrease the propagation depending on the problem characteristics. Unlike MAC-3 and FC which have different behaviour on dense and sparse problems, MPAC-3 performs well on both dense and sparse problems and in fact better than MAC-3 and FC on the hardest problems in the phase transition.

8 Conclusions and Future Work

This paper presents a new search algorithm, the so called MPAC-3, which maintains probabilistic arc consistency during search using *probabilistic support condition* and *probabilistic revision condition*. More specifically, it assumes the existence of a support (avoids the process of seeking a support) if the probability of having some support is at least equal to the threshold. Unlike MAC and FC where the strength of constraint propagation is static and the behaviour is different on dense and sparse problems, maintaining probabilistic arc consistency allows to adjust the strength of constraint propagation dynamically during search and performs well on both dense and sparse problems.

In future, we would like to investigate the effect of maintaining probabilistic arc consistency on real-world and academic problems. It seems relatively straightforward to generalise the notions of PSC and PRC to achieve probabilistic singleton consistencies and probabilistic hyper-arc consistency for non-binary CSPs.

References

1. C. Bessière and J.-C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'2001)*, pages 309–315, 2001.
2. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Support inference for generic filtering. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming*, 2004.
3. A. Chmesis and L. Sais. Constraint satisfaction problems:backtrack search revisited. In *Proceedings of the Sixteenth IEEE International Conference on Tools with Artificial Intelligence*, pages 252–257, Boca Raton, FL, USA, 2004. IEEE Computer Society.
4. H. El Sakkout, M. Wallace, and E.B. Richards. An instance of adaptive constraint propagation. In E.C. Freuder, editor, *Proceedings of the second International Conference on Principles and Practice of Constraint Programming*, number 1118 in Lecture Notes in Computer Science, pages 164–178, 1996.
5. E.C. Freuder and R.J. Wallace. Selective relaxation for constraint satisfaction problems. In *Proceedings of the Third International Conference on Tools for Artificial Intelligence*, San Diego, CA., 1991.
6. I.P. Gent, E. MacIntyre, P. Prosser, B. Smith, and T. Walsh. Random constraint satisfaction: Flaws and structure. *Journal of Constraints*, 6(4):345–372, 2001.
7. I.P. Gent and P. Prosser. Apes report: Apes-20-2000 inside mac and fc, 2000.
8. S.A. Grant and B.M. Smith. The phase transition behaviour of maintaining arc consistency. In W. Wahlster, editor, *Proceedings of the* 12^{th} *European Conference on Artificial Intelligence (ECAI'96)*, pages 175–179, 1996.
9. R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.
10. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
11. D. Mehta and M.R.C. van Dongen. Two new lightweight arc consistency algorithms. In M.R.C. van Dongen, editor, *Proceedings of the First International Workshop on Constraint Propagation and Implementation (CPAI'2004)*, pages 109–123, 2004.
12. D. Mehta and M.R.C. van Dongen. Reducing checks and revisions in coarse-grained mac algorithms. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, 2005.
13. P. Prosser. An empirical study of the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.
14. D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A.G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94)*, pages 125–129. John Wiley and Sons, 1994.
15. B.M. Smith and S.A. Grant. Sparse constraint graphs and exceptionally hard problems. In C.S Mellish, editor, *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'95)*, volume 1, pages 646–651. Morgan Kaufmann, 1995.
16. M.R.C. van Dongen. Saving support-checks does not always save time. *Artificial Intelligence Review*, 21(3–4):317–334, 2004.
17. R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *Proceedings of the Ninth Canadian Conference on Artificial Intelligence*, pages 163–169, Vancouver, B.C., 1992.

Static Value Ordering Heuristics for Constraint Satisfaction Problems

Deepak Mehta and M.R.C. van Dongen

Boole Centre for Research in Informatics/Cork Constraint Computation Centre

Abstract. Many problems in artificial intelligence can be formulated as Constraint Satisfaction Problems (CSPs). They involve assigning values to variables such that they satisfy all the constraints among them. In solving a hard satisfiable CSP, much time is often spent in searching branches of the search space which do not lead to a solution. If a CSP has a solution, then assigning the right value to each variable would enable a solution to be found without backtracking. Making such a perfect choice cheaply, in general, seems impossible, but a bit of guidance can make a substantial impact on the time required to find a solution. However, value ordering heuristics are relatively neglected. In this paper, we shall introduce the concept of static value ordering heuristic. The idea is to rearrange the values in their respective domains *prior* to search. More specifically, a weight is assigned to each value in the domain of each variable and the values are sorted based on their weights. The heuristics are not perfect, but experimental results show that they are frequently better than the initial ordering of values in the input problem. Further, we will show that rearranging values, prior to search in this fashion, also results in saving support checks, when a problem has no solution or if search has to be done for all the solutions.

1 Introduction

Many problems in artificial intelligence can be formulated as Constraint Satisfaction Problems (CSPs). They involve assigning values to variables such that they satisfy all constraints among them. For the purpose of this paper, we will only focus on binary CSPs, where the constraints have two variables. The basic search procedure to solve CSPs is a systematic backtracking. This involves repeated selection of an unassigned variable and selecting a value for it from its domain, or backtracking in case of failure, until a solution is found or all the possible sets of assignments have been tried. The worst-case time complexity of finding a solution for a CSP is exponential, i.e. d^n, where d is the maximum domain size and n is the number of variables. Nevertheless, many algorithms and heuristics have been developed to enhance this basic search procedure.

During backtrack search, variables are chosen in some order and each is instantiated with a value from its domain. It is well known that the efficiency of a search can be greatly affected by the choices made during search i.e. selecting which variable to consider next and which value to assign to this variable. When picking the next variable to instantiate, a commonly used heuristic picks the one which is most likely to fail. In other words, a variable should be selected so that search can fail with as little effort as possible, if it cannot lead to a solution. This is the rationale behind the *fail first*

principle [8]. On the other hand, when picking the next value to assign to a variable, a commonly used heuristic is to select a value which is most likely to be a part of a solution. If the value chosen to instantiate the variable has a higher probability of being part of a solution then it can make a substantial impact on the time required to find a solution.

When solving a hard satisfiable CSP, much time is often spent by searching branches of the search space which do not lead to a solution. If a CSP has a solution, then assigning the right value to each variable would enable a solution to be found without backtracking. Making such a perfect choice cheaply seems impossible, but a bit of guidance can make a substantial impact on the time required to find a solution. However, the use of value ordering heuristic, the question of determining which value should be assigned to the selected variable, is relatively neglected. This may be because there are no cheap and generic value ordering heuristics. All the heuristics proposed so far to the best of our knowledge are expensive because they are dynamic in nature and result in much CPU overhead. It seems to be an accepted fact [5] that with backtracking (k-way branching) the order in which values are chosen has no impact, when a problem has no solution or if all the solutions have to be searched. This could be another possible reason that value ordering heuristics have not attracted many researchers. However, in this paper we will demonstrate that this accepted fact is not always true.

In this paper, we shall introduce the concept of *static* value ordering heuristics. The idea is to rearrange the values in their respective domains *prior* to search. More specifically, each value is assigned some weight based on some reasonable criterion and then they are sorted based on their weights. Of course, the heuristics do not always make the perfect decision, but experimental results show that they are frequently better than the initial ordering of values in the input problem. Further, we will show that rearranging values prior to search in this fashion may result in saving support checks even when a problem has no solution or if search has to be done for all the solutions.

The rest of this paper is organized as follows: Section 2 describes background information. Section 3 introduces static value ordering heuristics. Section 4 shows experimental results. Finally, conclusions are presented in Section 5.

2 Background

2.1 Constraint Satisfaction

A *Constraint Satisfaction Problem* (*csp*) is defined as a set $\mathcal{V}$ of n variables, a non-empty domain $D(x)$ for each variable $x \in \mathcal{V}$ and a set of e constraints among subsets of variables of $\mathcal{V}$. A binary constraint C_{xy} between variables x and y is a subset of the Cartesian product of $D(x)$ and $D(y)$ that specifies the allowed pairs of values for x and y. We only consider CSPs whose constraints are binary. A *solution* is an assignment of values to all the variables such that no constraint is violated. A problem is said to be satisfiable (or consistent) if it has a solution, and unsatisfiable otherwise.

The *density* p_1 of a CSP is defined as $2\,e/(n^2 - n)$, where e is the number of constraints and n is the number of variables. The *tightness* p_2 of the constraint C_{xy} between the variables x and y is defined as $1 - \mid C_{xy} \mid / \mid D(x) \times D(y) \mid$. The *degree* of a variable

is the number of constraints involving that variable. A constraint satisfaction problem can be represented by a *constraint graph* which has a node for each variable and an arc connecting each pair of variables that are contained in a constraint. The *directed constraint graph* of a given CSP is a directed graph having an arc (x, y) for each combination of two mutually constraining variables x and y. We will use G to denote the directed constraint graph of the input CSP.

We shall use the notation proposed in [15] for describing and composing heuristics for selecting variables and arcs. Let $\delta_o(v)$ be the original degree of v, let $\delta_c(v)$ be the current degree of v, let $\delta_w(v)$ be the weighted degree [4] of v, let $\#(v)$ be a unique number for v, and let $s(v)$ be the current domain size of v. Finally, let $\pi_i((v_1, \ldots, v_n)) = v_i$ denote the *i-th projection operator*. The *composition* of order $\preceq_2$ and linear quasi-order $\preceq_1$ is denoted by $\preceq_2 \bullet \preceq_1$. Selection is done using $\preceq_1$ and ties are broken using $\preceq_2$. Composition associates to the left. The result of *lifting* linear quasi-order $\preceq$ and function f is denoted $\otimes^f_{\preceq}$. It is the linear quasi-order such that $v \otimes^f_{\preceq} w$ if and only if $f(v) \preceq f(w)$. For example, using this notation the *dom/wdeg* dynamic variable ordering heuristic with a lexicographical tie breaker can be described as $\otimes^{\#}_{\leq} \bullet \otimes^f_{\leq}$, where $f(v) = s(v)/\delta_o(v)$. The lexicographical arc selection heuristic can be described as $\otimes^{\#\circ\pi_2}_{\leq} \bullet \otimes^{\#\circ\pi_1}_{\geq}$. The reader is referred to [15] for more examples and further details.

2.2 Arc Consistency

A value $b \in D(y)$ is called a *support* for $a \in D(x)$ if $(a, b) \in C_{xy}$. Similarly $a \in D(x)$ is called a support for $b \in D(y)$ if $(a, b) \in C_{xy}$. A *support check* (consistency check) is a test to find if two values support each other. A value $a \in D(x)$ is called *viable* if for every variable y constraining x the value a is supported by some value in $D(y)$. A CSP is called *arc-consistent* if for every variable $x \in \mathcal{V}$, each value $a \in D(x)$ is viable.

Arc consistency algorithms are widely used to prune the search space of binary constraint satisfaction problems. Coarse-grained arc consistency algorithms such as AC-3 [11], AC-2001 [3], and AC-3_d [15] are efficient when it comes to transforming a CSP to its arc-consistent equivalent. These algorithms repeatedly revise the domains to remove all unsupported values. They use *revision ordering heuristics* [16, 10, 15], to select an arc from a data structure called a queue (a set really). When an arc, (x, y), is selected from the queue, $D(x)$ is *revised* against $D(y)$. Here to *revise* $D(x)$ against $D(y)$ means removing all values from $D(x)$ that are not supported by any value of $D(y)$. Pseudo-code for AC-3 equipped with *reverse variable-based* [12] revision ordering heuristics is depicted in Figure 1. Reverse variable based revision ordering heuristics first select a variable x and repeatedly select arcs of the form (x, y) to determine the next revision until there are no more such arcs or $D(x)$ becomes empty as shown in Figure 1. Selecting a variable x and revising it against all its neighbours y such that (x, y) is currently present in the queue, we call a *complete relaxation* of x. The revise function upon which AC-3 depends is depicted in Figure 2.

In Figure 1, if $D(x)$ was changed after a complete relaxation and if this was the result of *only one* effective revision (*effective_revisions* $= 1$), which happened to be against $D(y'')$, then all arcs of the form (y', x) where y' is a neighbour of x and $y' \neq y''$ are added to the queue. However, if $D(x)$ was changed as the result of *more than one*

Function AC-3: **Boolean**;

```
begin
   Q := G
   while Q not empty do begin
      select any x from {x : (x, y) ∈ Q }
      effective_revisions := 0
      for each y such that (x, y) ∈ Q do
         remove (x, y) from Q
         revise(x, y, change_x)
         if D(x) = ∅ then
            return False
         else if change_x then
            effective_revisions := effective_revisions + 1
            y'' := y;
      if effective_revisions = 1 then
         Q := Q ∪ { (y', x) ∈ G : y' ≠ y''}
      else if effective_revisions > 1 then
         Q := Q ∪ { (y', x) ∈ G}
   return True;
end;
```

Fig. 1. AC-3

effective revision (*effective_revisions* > 1) then *all* arcs of the form (y', x) where y' is a neighbour of x are added to the queue. Modulo constraint propagation effects this avoids queue maintenance overhead.

Function $revise(x, y,$ **var** $change_x)$

```
begin
   change_x := False
   for each a ∈ D(x) do
      if ∄b ∈ D(y) such that b supports a then
         D(x) := D(x) \ { a }
         change_x := True
end;
```

Fig. 2. Algorithm *revise* of AC-3

2.3 Maintaining Arc Consistency

MAC [13] is a backtrack algorithm that maintains arc consistency during search. It reduces the thrashing behaviour of a backtrack algorithm, which usually fails many times as a result of the same local inconsistencies. Before starting search MAC transforms the input CSP to its arc-consistent equivalent. The *arc-consistent* domain of a variable is the domain of that variable in this arc-consistent equivalent. During backtrack search,

variables are chosen in some order and each is instantiated with a value from its domain. MAC maintains arc consistency after each variable assignment. To re-establish arc consistency following the instantiation of a variable x the queue is initialised to all arcs incident to x. More specifically, all arcs of the form (y, x) are added to the queue where y is a future variable constrained by x. When values are deleted, more arcs may have to be added to the queue to determine if these deletions lead to further deletions.

For the remainder of this paper for any variable x, we use $D_o(x)$ for the original domain of x, $D_{ac}(x)$ for the arc-consistent domain of x, and $D(x)$ for the current domain of x. MAC-*x* uses AC-*x* for maintaining arc consistency during search.

3 Static Value Ordering Heuristics

Value ordering heuristics are used to select a value from the domain of a variable to instantiate that variable during search. If the value selected has a higher probability of being part of a solution then selecting this value can make a significant difference in terms of the solution time. Heuristics for ordering the values may help in finding the first solution more efficiently in terms of the solution time and support checks. To the best of our knowledge all value ordering heuristics [6, 5] proposed so far are *dynamic* in nature. We call them dynamic because generally rankings are established among the values in the domain of the selected variable *after* each variable selection. Until a solution is found or all possible sets of assignments have been tried, a value having the highest rank which is untried is selected to instantiate the selected variable. There are no cheap general-purpose dynamic value ordering heuristics (see e.g. [14, 2]).

We propose static value ordering heuristics as opposed to the dynamic value ordering heuristics. The idea is to assign a weight to each value in the domain of each variable. These weights remain static throughout the search. After associating a weight with each value, the next step is to rearrange the values in their respective domains in the decreasing order of their weights, *prior* to search. The order of values is only calculated once. The advantage is that there is no overhead of establishing rankings during search after every variable selection and selecting the best value based on the weight during search only requires $\mathcal{O}(1)$ time-complexity. Of course, these heuristics will not always make the correct decision but experimental results show that they are frequently better than the initial ordering of values in the input problem.

There exist many possible ways to assign a weight to each value. However, we explore the feasibility of using the knowledge of *support count* gathered before the search to compute the weight for each value to improve the ordering of values. The support count of $a \in D(x)$ with respect to y, is the number of values in the domain of y supporting a. Let C_{xy} be the constraint between x and y, let $a \in D(x)$ (also denoted as (x, a)), and let *weight*$[\, x, a \,]$ be the weight of (x, a) and let *scount*$[\, x, y, a \,]$ be the number of supports of (x, a) in $D_{ac}(y)$. In general many weights are possible. In this paper, we will study the following three weights:

$$weight[\, x, a \,] = \sum_{(x,y) \in G} scount[\, x, y, a \,] \tag{1}$$

$$weight[\,x, a\,] = \sum_{(x,y)\in G} scount[\,x, y, a\,]/|D_{ac}(y)| \tag{2}$$

$$weight[\,x, a\,] = \prod_{(x,y)\in G} scount[\,x, y, a\,] \tag{3}$$

Frost and Dechter [5] have proposed a *min-conflict* value ordering heuristic, the formula of which is equivalent to the one as mentioned in Equation (1). Geelen [6] has shown three formulae for value-selection which are called *total-cost*, *cruciality* and *promise* which are, before search, equivalent to the Equations (1), (2) and (3) respectively. This constitutes the main difference between their approach and ours. For a static value ordering heuristic rankings are established for all values only once. For a dynamic value ordering heuristic due to backtracking and failed assignments of a variable rankings are established among the values of the selected variable after each variable selection which may be computationally expensive.

Pseudo-code for ordering the values is shown in Figure 3. The first step is to compute the support count for each arc-value pair, involving the arc (x, y) and the value a in $D(x)$. Note that the algorithm uses the bidirectional property of the constraints. Next step is to compute the weight for each value by using either Equation (1), (2) or (3). Final step is to rearrange the values in the decreasing order of their weights. The time-complexity of this algorithm is $\mathcal{O}(e\,d^2)$ and the space-complexity required is $\mathcal{O}(\max(e, n)\,d)$. This algorithm can be used to order the values in the domains after making the problem initially arc-consistent.

Function OrderValues ( )

```
begin
   for each (x, y) ∈ G do
      for each a ∈ D_ac(x) do
         scount[ x, y, a ] := 0
   for each (x, y) ∈ G such that x < y do
      for each a ∈ D_ac(x) do
         for each b ∈ D_ac(y) do
            if (a, b) ∈ C_xy then
               scount[ x, y, a ] := scount[ x, y, a ] + 1
               scount[ y, x, b ] := scount[ y, x, b ] + 1
            end
   for each x ∈ V do
      for each a ∈ D_o(x) do
         compute weight for (x, a) by using either Equation (1), (2) or (3)
         assign it to weight[ x, a ]
      sort D_ac(x) in the decreasing order of the weights
end;
```

Fig. 3. Initialisation of weights and rearrangement of the values.

Arranging the values in the decreasing order of their weights (which can be computed by using either Equation (1), (2) or (3)), in some sense can be seen as arranging the values in the increasing order of their constrainedness. This can be advantageous while revising the domains of the variables. Putting the least constrained value at the

beginning of the domain list may help other values to find their support quickly during revision. This may allow to save a few negative support checks. The further the first support is away from the start, the more negative support checks are required to find it. Ordering of values in the increasing order of their constrainedness is a novel approach to reduce support checks during revisions. This may also allow to save support checks for the problems which has no solution or if all solutions have to be searched.

4 Experimental Results

4.1 Introduction

In this section, we shall present some experimental results to prove the practical efficiency of static value ordering heuristics. All experiments are conducted using MAC-3 as a backtrack algorithm equipped with a search heuristic that learns from conflicts [4]. More specifically, we used a conflict-directed variable ordering heuristic *dom/wdeg* with a lexicographical tie breaker, where *dom* is the domain size and *wdeg* is the weighted degree [4] of a variable. Weighted degree of a variable x can be defined as follows:

$$x_{wdeg} = \sum_{(x,y)\in G} warc[\,x, y\,],$$

where $warc[\,x, y\,]$ is a counter associated with each arc, which is initialised to 1 prior to search. During search these counters are incremented whenever a domain wipe-out occurs. This heuristic has been shown very stable and efficient when used with MAC algorithms in [4, 9]. Using the notations, as explained in Section 2, this heuristic can be described as $\otimes^{\#}_{\leq} \bullet \otimes^{f}_{\leq}$, where $f(v) = s(v)/\delta_w(v)$.

In all our experimental results, we use *initial* to refer to the ordering of the values in the domains in the input problem. When the values in the domains are ordered using Equation (1), we refer to it as *svoh*$_1$ (static value ordering heuristic using Equation (1)). Similarly, we refer *svoh*$_2$ and *svoh*$_3$ to the ordering of values using Equations (2) and (3) respectively. The arc consistency component, AC-3, of MAC-3 is equipped with a reverse variable based revision ordering heuristic [12]. Let *comp* [15] be the variable selection order $\otimes^{\#}_{\leq} \bullet \otimes^{\delta_c}_{\geq} \bullet \otimes^{s}_{\leq}$, then the reverse variable based heuristic used in the arc consistency component of MAC-3 can be given by $\otimes^{\#\circ\pi_2}_{\leq} \bullet \otimes^{s\circ\pi_2}_{\leq} \bullet \otimes^{\pi_1}_{comp}$.

All algorithms are written in C. The experiments are carried out on linux on a PC Pentium III (2.266 GHz processor and 256 MB RAM). We compare the performance of different static value ordering heuristics with that of the initial ordering of values in terms of constraint checks (chks), visited nodes (vn) and the solution time (cpu) on random problems, real-word problems and academic problems.

4.2 Comparison

First, we experimented with random problems which were generated by Frost *et al.*'s model B generator [7] [1]. In this model a random CSP instance is typically represented

[1] (`http://www.lirmm.fr/~bessiere/generator.html`)

as $\langle N, D, C, T \rangle$ with N variables, each having the domain size of D. The parameter C specifies the number of constraints out of $N \times (N-1)/2$ possible constraints and the parameter T (tightness) specifies the number of tuples not allowed by the constraint out of D^2 possible tuples. We studied four different combination of $\langle N, D, C, T \rangle$. For each combination, 50 random problems were generated and their mean performance is reported in Table 1. Parameters are selected in such a way that the problem instances are located at the phase transition. Problems are relatively harder to solve in the phase transition region, the region between an under-constrained region where all instances are almost surely satisfiable and an over-constrained region where all instances are almost surely unsatisfiable. The instances for the third and fourth problems are generated using the parameters mentioned in [5].

Table 1. Random Problems

instances		initial	svoh$_1$	svoh$_2$	svoh$_3$
$\langle 65, 20, 167, 260 \rangle$	cpu	0.225	0.197	0.201	0.201
(28/50 sat)	chks	5,447,656	4,513,250	4,596,678	4,590,326
	vn	3,263	2,919	2,950	2,975
$\langle 90, 20, 280, 230 \rangle$	cpu	3.494	1.025	0.982	1.057
(50/50 sat)	chks	60,424,559	16,734,753	16,209,553	17,290,406
	vn	38,550	12,317	11,841	12,829
$\langle 125, 3, 929, 1 \rangle$	cpu	0.105	0.097	0.099	0.097
(17/50 sat)	chks	370,311	353,926	360,944	353,936
	vn	1,473	1,385	1,418	1,385
$\langle 350, 3, 2292, 1 \rangle$	cpu	1.121	0.326	0.293	0.326
(50/50 sat)	chks	1,575,393	501,721	456,418	501,721
	vn	5,165	1,895	1,785	1,895

The results presented in Table 1 demonstrate that MAC-3 equipped with a *svoh*$_i$ (where i = 1, 2 or 3) performs better in terms of the solution time, support checks and visited nodes, when compared to MAC-3 equipped with the *initial* value ordering heuristic. Note that selecting a value with *initial* and *svoh*$_i$, all take $\mathcal{O}(1)$ time. One can observe the huge gain obtained by *svoh*$_i$ heuristics on second and fourth problem, where all instances are satisfiable. This clearly shows that arranging values in the domains in the increasing order of their constrainedness, prior to search, can make a significant difference to find *one* solution.

We performed a second set of experiments on the forced satisfiable instances [17], which are generated using model RB [18]. This model guarantees the existence of an asymptotic phase transition by applying a limited restriction on domain size and on constraint tightness and a threshold point can be precisely located to generate hard instances. A class of random CSP instance of model RB is denoted as RB(k, n, α, r, p) where k is the arity of the constraint, n is the number of variables, α and r are used to determine the domain size and the number of constraints respectively and p denotes the tightness of each constraint. Experiments were carried out on binary CSP instances of class RB$(2, n, 0.8, 0.8/ln\frac{4}{3}, 0.25)$ for $n \in \{30, 35, 40, 45\}$. The results of these experi-

ments are summarised in Table 2. Each set is denoted as frb-n-d where n is the number of variables and d is the uniform domain size for each variable. Results for each set shown in Table 2 represent the average of 5 forced satisfiable instances of CSPs. Results show that *svoh*$_i$ heuristics are usually better than the initial ordering of values except for frb-40-19.

Table 2. Forced satisfiable random problems

instances		initial	svoh$_1$	svoh$_2$	svoh$_3$
frb-30-15	cpu	0.192	0.112	0.123	0.137
(5/5 sat)	chks	3,114,885	1,838,268	2,020,902	2,240,825
	vn	3,470	2,124	2,331	2,619
frb-35-17	cpu	2.594	1.094	1.093	1.446
(5/5 sat)	chks	40,878,755	17,294,249	17,217,755	18,315,004
	vn	39,265	17,099	17,061	18,034
frb-40-19	cpu	11.552	11.997	11.679	13.832
(5/5 sat)	chks	177,424,742	180,016,269	175,061,284	207,451,296
	vn	158,816	168,536	163,959	194,592
frb-45-21	cpu	194.058	116.728	113.723	195.198
(5/5 sat)	chks	1,501,335,748	1,413,489,771	1,415,111,821	704,453,902
	vn	1,861,016	1,153,753	1,154,926	1,974,567

Next, we experimented with 3-sat instances which are converted to binary csp instances[2] [9]. There are two sets denoted as *ehi-85* and *ehi-90*. Each set has 100 easy random 3-sat instances with a small unsatisfiable part. One can notice in Table 3 that even for unsatisfiable instances, *svoh*$_i$ causes MAC-3 to visit on average 30% fewer nodes than the initial ordering of values in the input problem. It is worth emphasising that conflict-directed variable ordering heuristics are heavily influenced by the value selected to instantiate the variable and the arc considered for the next revision. However, this observation deserves further exploration.

Table 3. Random 3-SAT instances

instances		initial	svoh$_1$	svoh$_2$	svoh$_3$
ehi-85	cpu	1.775	1.415	1.408	1.310
(0/100 sat)	chks	4,974,455	4,417,270	4,358,458	4,175,680
	vn	2,289	1,172	1,764	1,648
ehi-90	cpu	2.099	1.388	1.221	1.333
(0/100 sat)	chks	5,527,397	4,265,716	3,812,202	4,118,905
	vn	2,280	1,672	1,561	1,631

Next, we experimented with the modified versions of real-world binary instances of RLFAP (Radio Link Frequency Assignment) problems. In [1, 4], it has been shown that

[2] These converted instances are available at `www.cril.univ-artois.fr/~lecoutre/`

harder instances of these problems are possible by removing some frequencies. For example *graph*14_f27 corresponds to the *graph*14 from which the 27 highest frequencies have been removed. We did not consider optimisation but satisfiability only. The results of these experiments are summarised in Table 4. Again, results shown in Table 4 clearly shows that *svoh*$_i$ are usually better than the initial ordering of the values.

Table 4. RLFAP Instances

instances		initial	svoh$_1$	svoh$_2$	svoh$_3$
graph14_f27	cpu	0.853	0.570	0.574	0.571
(sat)	chks	10,671,050	8,192,428	8,193,737	8,193,532
	vn	28,595	20,131	20,136	20,135
graph02_f24	cpu	0.036	0.050	0.047	0.049
(sat)	chks	534,527	1,251,682	1,174,798	1,231,069
	vn	634	591	414	546
graph08_f10	cpu	6.806	0.817	0.882	0.786
(sat)	chks	74,091,888	11,353,501	12,306,468	11,265,384
	vn	81,803	9,745	10,679	9,309
graph02_f25	cpu	17.762	0.340	0.382	0.365
(unsat)	chks	230,895,326	5,204,626	5,572,219	5,548,506
	vn	292,075	4,156	4,569	4,552
graph08_f11	cpu	1.696	0.185	0.188	0.189
(unsat)	chks	20,286,505	3,296,196	3,306,326	3,304,455
	vn	12,027	982	987	993
scen01_f8	cpu	0.222	0.216	0.224	0.220
(sat)	chks	3,061,420	5,482,347	5,483,551	5,482,548
	vn	1,486	818	818	818
scen02_f25	cpu	6.478	5.155	5.160	5.161
(unsat)	chks	85,584,141	66,106,418	66,106,418	66,106,418
	vn	41,899	39,053	39,053	39,053
scen03_f11	cpu	1.458	1.462	1.466	1.465
(unsat)	chks	16,085,497	15,979,153	15,979,153	15,979,153
	vn	6717	7865	7865	7865

Finally, we experimented with the academic instances of Queens-Knights problems as mentioned in [4]. This problem is basically an integration of two different problems. The first one is to place k knights on a $n \times n$ chessboard such that no two knights share the same square and all knights form a cycle with respect to their moves. The second one is to place q queens on a chessboard of size $n \times n$ such that no two queen attack on each other. Further on, two variations of these problems are possible which are as follows:

I. $K_k \oplus Q_q$: Here, k-knights and q-queens instances are merged without any interaction.
II. $K_k \otimes Q_q$: Here, k-knights and q-queens instances are merged such that queens and knights cannot share the same square on the chessboard.

When the value of k (the number of knights) is odd, the problem is unsatisfiable. The results for the unsatisfiable instances of Queen-Knights problems are depicted in Table 5. Notice that in a few cases there is only a marginal difference in terms of the nodes visited by MAC-3 when equipped with *initial* and *svoh*$_i$ value ordering heuristic. However, a *svoh*$_i$ allows to save support checks. This is because a *svoh*$_i$ value ordering heuristic arranges the values in the domains such that the least constrained value is at the beginning of the domain. On average this speeds up the process of finding a support during revisions of the domains.

Table 5. Academic Instances

instances		initial	svoh$_1$	svoh$_2$	svoh$_3$
	cpu	0.041	0.033	0.033	0.023
$K_5 \oplus Q_8$	chks	3,334,548	2,373,825	2,373,825	1,786,865
$(n = 8)$	vn	1,284	1,343	1,343	1,007
	cpu	0.041	0.036	0.036	0.040
$K_5 \otimes Q_8$	chks	3,135,368	2,209,157	2,209,157	25,446,045
$(n = 8)$	vn	1,183	1,193	1,193	1,405
	cpu	2.621	1.741	0.927	0.941
$K_5 \oplus Q_{12}$	chks	295,804,348	180,595,256	97,326,791	98,005,633
$(n = 12)$	vn	28,497	26,094	13,863	13,989
	cpu	3.622	3.153	1.906	1.774
$K_5 \otimes Q_{12}$	chks	371,731,471	271,980,451	164,329,385	149,292,105
$(n = 12)$	vn	29,768	33,927	20,519	18,378

4.3 Discussion

Frost and Dechter [5] have showed that with backtracking the order in which values are chosen makes no *difference* on problems which have no solution, or when searching for all solutions. Smith and Sturdy [14] have further clarified that this is only true for k-*way* branching but not for *binary-branching*. In k-way branching, k branches are formed, when a variable x with k values in its domain is selected for the instantiation. In binary-branching, when a variable x is selected for instantiation, the values are assigned via a sequence of binary choices. With k-way branching, k subtrees are explored independently and the search spaces of these k subtrees are commutative. Therefore, the order in which values are assigned cannot affect the search. However, with binary-branching the subtrees resulting from successive assignments to a variable are not explored independently. Here, the order in which values are assigned can affect the search.

We argue that rearranging the values prior to search as reported in this paper can make a difference at least in terms of the number of support checks for any (static and dynamic) variable ordering heuristic with both k-way branching and binary-branching even when the problem has no solution and if all solutions have to be searched. For illustration, for $K_5 \otimes Q_8$ using *dom/deg* [2], MAC-3 with *initial* value ordering heuristic results in spending $10,044,320$ checks, $5,890$ visited nodes and 0.151 *cpu* time in

seconds while MAC-3 with *svoh*$_1$ results in spending $7,783,790$ checks, $5,890$ visited nodes and 0.126 seconds. Note that though both the heuristic cause MAC-3 to visit the same number of nodes but due to the different arrangement of values in the domains, there is a saving of 22% of checks during revisions of domains by using *svoh*$_1$.

Further, we also argue that even with k-way branching, value ordering heuristics for a certain class of variable ordering heuristics e.g. conflict-directed heuristics can make a difference in terms of the visited nodes and checks on problems, which have no solution or when all solutions are required to be searched. Conflict-directed heuristics learn from encountered failures to manage the choice of the variable to be instantiated. These heuristics exploit the weighted degree of a variable which keeps on changing during search and is heavily influenced by the value which is selected to instantiate the variable or the arc considered for the next revision. The subtrees rooted from the selected variable are not independent. Hence, the order in which values are selected can make a difference for conflict-directed heuristics. Results shown in Tables 3 and 5 confirm this.

The static value ordering heuristics proposed in this paper, exploit the knowledge of the support counts to compute the weight for each value in the domain of each variable. They have a small overhead of computing the support count for each arc-value pair before search. For easy problems, where not much search is required, the overhead of computing the support counts is almost always worse than the benefit at least in terms of support checks. However, in many cases it should be possible to compute the support count using the semantics of the constraint rather than using the algorithm as mentioned in Figure 3. For example, if there is an equality constraint between x and y then for each value a in domain of x, $scount[\,x, y, a\,]$ can be set to 1 after making the problem initially arc-consistent. The overall conclusion we draw from our experiments is that on average MAC-3 equipped with a *svoh*$_i$ value ordering heuristic usually performs better than the *initial* ordering of values in finding one solution, all solutions, or detecting the insolubility of the problem.

5 Conclusions and Future Work

The purpose of this paper is to introduce static value ordering heuristics which are generic, cheap and easy to implement. The idea is to assign a weight to each value based on some reasonable criterion and rearrange the values in their respective domains prior to search based on their weights. The ordering of values is only established once and it remains static throughout the search. The advantages are twofold. The first advantage is that it helps to select better values to instantiate the selected variable. The second advantage is that it helps to reduce negative checks during revisions of domains. We have compared the static value ordering heuristics, which use the knowledge of support counts, against the initial ordering of values in the domains in the input problem. The overall conclusions we draw from our experiments is that arranging values in the increasing order of their constrainedness in the domains prior to search is usually better than the initial ordering of values in the input problem.

In future, we would like to compare them against dynamic value ordering heuristics which are generally considered to be expensive. We would also like to devise other possibilities of computing the weights.

Acknowledgement

We would like to thank Christophe Lecoutre for providing most of the problems and converting them into the required format for our solver. The first author is supported by Boole Centre for Research in Informatics (BCRI). This work has received some support from Science Foundation Ireland under Grant No. 00/PI.1/C075.

References

1. C. Bessière, A. Chmeiss, and L. Sais. Neighborhood-based variable ordering heuristics for the constraint satisfaction pproblem. In *Principles and Practice of Constraint Programming*, pages 565–569, 2001.
2. C. Bessière and J.-C. Régin. Mac and combined heuristics: Two reasons to forsake fc (and cbj ?) on hard probelms. In E. Freuder, editor, *Principles and Practice of Constraint Programming*, pages 61–75. Springer, 1996.
3. C. Bessière and J.-C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'2001)*, pages 309–315, 2001.
4. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 13th European Conference on Artificial Intelligence*, 2004.
5. D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'95*, pages 572–578, Montreal, Canada, 1995.
6. P. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of the* 12^{th} *European Conference on Artificial Intelligence (ECAI'92)*, 1992.
7. I. Gent, E. MacIntyre, P. Prosser, B. Smith, and T. Walsh. Random constraint satisfaction: Flaws and structure. *Journal of Constraints*, 6(4):345–372, 2001.
8. R. Haralick and G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.
9. C. Lecoutre, F. Boussemart, and F. Hemery. Backjump-based techniques versus conflict-directed heuristics. In *ICTAI*, pages 549–557, 2004.
10. C. Lecoutre, F. Boussemart, and F. Hemery. Revision ordering heuristics for the constraint satisfaction problems. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming*, 2004.
11. A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
12. D. Mehta and M. van Dongen. Reducing checks and revisions in coarse-grained mac algorithms. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, 2005.
13. D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94)*, pages 125–129. John Wiley and Sons, 1994.
14. B. Smith and P. Sturdy. Value ordering for finding all solutions. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, 2005.

15. M. van Dongen. Saving support-checks does not always save time. *Artificial Intelligence Review*, 21(3–4):317–334, 2004.
16. R. Wallace and E. Freuder. Ordering heuristics for arc consistency algorithms. In *Proceedings of the Ninth Canadian Conference on Artificial Intelligence*, pages 163–169, Vancouver, B.C., 1992.
17. K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. A simple model to generate hard satisfiable instances. In *Proceedings of the 19th Internationl Joint Conference on Artificial Intelligence*, Edinburgh, Scotland, 2005.
18. K. Xu and W. Li. Exact phase transitions in random constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 12:93–103, 2000.

Constraint Propagation versus Local Search for Conditional and Composite Temporal Constraints

Malek Mouhoub and Amrudee Sukpan

University of Regina
Dept of Computer Science
Wascana Parkway, Regina, SK, Canada, S4S 0A2
\{mouhoubm,sukpan1a\}@cs.uregina.ca

Abstract. A well known approach to managing the numeric and the symbolic aspects of time is to view them as Constraint Satisfaction Problems (CSPs). Constraint propagation techniques can then be used to efficiently check for the consistency of the CSP and to find possible solutions. Our aim is to extend the temporal CSP formalism in order to include activity constraints and composite variables. Indeed, in many real life applications the set of variables involved by the temporal constraint problem to solve is not known in advance. More precisely, while some temporal variables (called events) are available in the initial problem, others are added dynamically to the problem during the resolution process via activity constraints and composite variables. Activity constraints allow some variables to be activated (added to the problem) when activity conditions are true. Composite variables are defined on finite domains of events. We propose in this paper two methods based respectively on constraint propagation and stochastic local search (SLS) for solving temporal constraint problems with activity constraints and composite variables. We call these problems Conditional and Composite Temporal Constraint Satisfaction Problems (CCTCSPs). Experimental study we conducted on randomly generated CCTCSPs demonstrates the efficiency of our exact method based on constraint propagation in the case of middle constrained and over constrained problems while the SLS based method is the technique of choice for under constrained problems and also in case we want to trade search time for the quality of the solution returned (number of solved constraints).

1 Introduction

Representing and reasoning about numeric and/or symbolic aspects of time is crucial in many real world applications such as scheduling and planning [1–4], natural language processing [5, 6], molecular biology [7] and temporal database [8]. A well-know approach to managing these two aspects of time is to view them as Constraint Satisfaction Problems (CSPs). We talk then about temporal constraint networks [9–12]. Here, a CSP involves a list of variables defined on discrete domains of values and a list of relations constraining the values that the variables can simultaneously take [13–15].

In a temporal constraint network, variables, corresponding to temporal objects, are defined on a set of time points or time intervals while constraints can either restrict the

domains of the variables and/or represent the relative position between variables. The relative position between variables can be expressed via qualitative or quantitative relations. Quantitative relations are temporal distances between temporal variables while qualitative relations represent incomplete and less specific symbolic information between variables. Constraint propagation techniques and backtrack search are then used to check the consistency of the temporal network and to infer new temporal information. While a considerable research work has been concerned with reasoning on the metric or the symbolic aspects of time (respectively through metric and qualitative networks), little work such as [16, 17, 2, 18] has been developed to manage both types of information. In [19, 20], we have developed a temporal model, TemPro, based on Allen's interval algebra [9] and a discrete representation of time, to express numeric and symbolic time information in terms of qualitative and quantitative temporal constraints. More precisely, TemPro translates an application involving numeric and symbolic temporal information into a binary CSP[1] called Temporal CSP (or TCSP[2]) where variables are temporal events defined on domains of numeric intervals and binary constraints between variables correspond to disjunctions of Allen primitives [9]. The resolution method for solving the TCSP is based on constraint propagation and requires two stages. In the first stage, local consistency is enforced by applying the arc consistency on variable domains and the path consistency on symbolic relations. A backtrack search algorithm is then performed in the second stage to check the consistency of the TCSP by looking for a feasible solution. Note that for some TCSPs local consistency implies the consistency of the TCSP network [17]. The backtrack search phase can be avoided in this case.

In order to deal with a large variety of real world applications, we present in this paper an extension of the modeling framework TemPro including the following :

- Managing composite temporal variables. Composite temporal variables are variables whose values are temporal events.
- Handling activity constraints. This is the case where temporal variables (composite or events) can have either active or non active status. Only active variables require an assignment from their domain of values. Non active variables will not be considered during the resolution of the temporal network until they are activated. A variable can be activated by default (in the initial problem) or by an activity constraint. Given two variables X_i and X_j, an activity constraint has the following form $(X_i = a_{i1}) \vee \ldots (X_i = a_{ip}) \rightarrow X_j$. This activity constraint will activate X_j if the active variable X_i is assigned one of the values $a_{i1} \ldots a_{ip}$ from its domain.

We call conditional TCSP (CTCSP) a TCSP augmented by activity constraints. Solving a CTCSP can be seen like solving a TCSP dynamically i.e when some of the variables and their corresponding constraints are added dynamically during the resolution of the TCSP. We call a composite CTCSP (CCTCSP) a CTCSP including composite temporal variables. A CCTCSP represents a finite set of possible CTCSPs where

[1] In a binary CSP constraints can only be unary or binary.

[2] Note that the acronym TCSP was used in [11]. The well known TCSP, as defined by Dechter et al, is a quantitative temporal network used to represent only numeric temporal information. Nodes represent time points while arcs are labeled by a set of disjoint intervals denoting a disjunction of bounded differences between each pair of time points.

each CTCSP corresponds to a complete assignment of values (temporal events) to composite variables. Solving a CCTCSP consists of finding a feasible scenario for one of its possible CTCSPs. Solving a CTCSP requires a backtrack search algorithm with exponential complexity in time $O(D^N)$ where N is the total number of temporal events and D the domain size of each event. The possible number of CTCSPs the CCTCSP involves is d^M where M is the number of composite variables and d their domain size. Thus, solving a CCTCSP requires a backtrack search algorithm of complexity $O(D^N \text{ x } d^M)$. To overcome this difficulty in practice, we propose in this paper two methods respectively based on constraint propagation and stochastic local search (SLS) for solving efficiently CCTCSPs. Constraint propagation includes arc consistency [14] as well as forward check and full look ahead strategies [15]. On the other hand, the SLS method we use is based on the Min-Conflict-Random-Walk (MCRW) algorithm [21]. Experimental study on randomly generated CCTCSPs demonstrates the efficiency of our exact method based on constraint propagation in case we look for a complete solution while the SLS based method is the technique of choice in case we want to trade search time for the quality of the solution.

The rest of the paper is structured as follows. In the next section we introduce the CCTCSP framework through an example. Sections 3 and 4 are respectively dedicated to the constraint propagation techniques and SLS method for solving CCTCSPs. Section 5 describes the experimental comparative tests we have conducted on random CCTCSPs. Finally, concluding remarks are covered in Section 6.

2 Conditional and Composite Temporal Constraint Satisfaction Problems (CCTCSPs)

Managing conditional, composite and dynamic CSPs has already been reported in the literature [22–29]. [22] introduced the notion of *Dynamic Constraint Satisfaction Problems* for configuration problems (renamed *Conditional Constraint Satisfaction Problems (CCSPs)* later). In contrast with the standard CSP paradigm, in a CCSP the set of variables requiring assignment is not fixed by the problem definition. A variable has either *active* or *nonactive* status. An activity constraint enforces the change of the status of a given variable from $nonactive$ to $active$. In [23], Freuder and Sabin have extended the traditional CSP framework by including the combination of three new CSP paradigms : *Meta CSPs, Hierarchical Domain CSPs, and Dynamic CSPs*. This extension is called *composite CSP*. In a composite CSP, the variable values can be entire sub CSPs. A domain can be a set of variables instead of atomic values (as it is the case in the traditional CSP). The domains of variable values can be hierarchically organized. The participation of variables in a solution is dynamically controlled by activity constraints. Jónsson and Frank [27] proposed a general framework using procedural constraints for solving dynamic CSPs. This framework has been extended to a new paradigm called Constraint-Based Attribute and Interval Planning (CAIP) for representing and reasoning about plans [28]. CAIP and its implementation, the EUROPA system, enable the description of planning domains with time, resources, concurrent activities, disjunctive preconditions and conditional constraints. The main difference, comparing to the for-

malisms we described earlier, is that in this latter framework [27] the set of constraints, variables and their possible values do not need to be enumerated beforehand which gives a more general definition of dynamic CSPs. Note that the definition of dynamic CSPs in [27] is also more general than the one in [26] since in this latter work variable domains are predetermined. Finally, in [29], Tsamardinos et al propose the Conditional Temporal Problem (CTP) formalism for Conditional Planning under temporal constraints. This model extends the well known qualitative temporal network proposed in [11] by adding instantaneous events (called observation nodes) representing conditional constraints.

We adopt both the CCSP [22] and the composite CSP [23] paradigms and extend the modeling framework TemPro [20] by including conditional temporal constraints and composite temporal events as shown in introduction. TemPro will then have the ability to transform constraint problems involving numeric information, symbolic information, conditional constraints and composite variables into the CCTCSP we have described in introduction. Comparing to the formalisms we mentioned above, ours has the following specificities.

1. Our work focuses on temporal constraints while the previous literature is on general constraints, if we exclude the work in [29] and [28]. Both these latter formalisms handle only quantitative time information while ours combines both quantitative and qualitative temporal constraints.
2. Our model is domain independent and is not restricted to a particular area such as planning or scheduling. It can however be used in a large variety of applications involving symbolic and|or numeric temporal constraints. Moreover, the qualitative constraints are based on the whole Allen Algebra [9] which offers more expressiveness. Altough this will lead to NP-hard problems, the solving techniques that we will present in the next 2 Sections overcome this difficulty, in practice, as we will see in Section 5.
3. Our model is based on a discrete representation of time. Thus, events are defined on discrete values (numeric intervals). This offers an easier way to handle numeric temporal information with different granularities. It will also enable the constraint propagation techniques and approximation methods to be applied in a straight forward manner.
4. Numeric and symbolic temporal constraints as well as conditional constraints and composite variables, are managed within the same constraint graph.

In the following we will define the CCTCSP model and its corresponding network (graph representation) through an example.

Definition

A Conditional and Composite Temporal Constraint Satisfaction Problem (CCTCSP) is a tuple $\langle E, D_E, X, D_X, IV, C, A\rangle$, where

E=$\{e_1, \ldots e_n\}$ is a finite set of temporal variables that we call events. Events have a uniform reified representation made up of a proposition and its temporal qualification : $Evt = OCCUR(p, I)$ defined by Allen [9] and

denoting the fact that the proposition p occurred over the interval I. For the sake of notation simplicity, an event is used in this paper to denote its temporal qualification.

D_E= $\{D_{e_1}, \ldots D_{e_n}\}$ is the set of domains of the events. Each domain D_{e_i} is the finite and discrete set of numeric intervals the event e_i can take. D_{e_i} is expressed by the fourfold $[begintime_{e_i}, endtime_{e_i}, duration_{e_i}, step_{e_i}]$ where $begintime_{e_i}$ and $endtime_{e_i}$ are respectively the earliest start time and the latest end time of the corresponding event, $duration_{e_i}$ is the duration of the event and $step_{e_i}$ defines the distance between the starting time of two adjacent intervals within the event domain. The discretization step $step_{e_i}$ allows us to handle temporal information with different granularities.

X= $\{x_1, \ldots x_m\}$ is the finite set of composite variables.

D_X=$\{D_{x_1}, \ldots D_{x_m}\}$ is the set of domains of the composite variables. Each domain D_{x_i} is the set of possible events the composite variable x_i can take.

IV is the set of initial variables. An initial variable can be a composite variable or an event. $IV \subseteq E \bigcup X$.

C = $\{C_1, \ldots C_p\}$ is the set of *compatibility constraints*. Each compatibility constraint is a qualitative temporal relation between two variables in case the two variables are events or a set of qualitative relations if at least one of the two variables involved is composite. A qualitative temporal relation is a disjunction of Allen primitives [9] (see table 1 for the definition of the Allen primitives).

A is the set of *activity constraints*. Each activity constraint has the following form: $(X_i = a_{i1}) \vee \ldots (X_i = a_{ip}) \rightarrow X_j$ where X_i and X_j are events or composite variables. This activity constraint is fired if X_i is active and is assigned one of the values $a_{i1} \ldots a_{ip}$ from its domain. The variable X_j will then be activated.

Example

Consider the following temporal problem:

John, Mike and Lisa are going to see a movie on Friday. John will pick Lisa up and Mike will meet them at the theater. If John arrives at Lisa's before 7:30, then they will stop at a convenience store to get some snacks and pops. It will take them 30 minutes to reach the theater if they stop at the store and 15 minutes otherwise. There are three different shows playing: $movie_1, movie_2$ and $movie_3$. If they finish the movie by 9:15, they will stop at a Pizza place 10 minutes after the end of the movie and will stay there for 30 minutes. John leaves home between 7:00 and 7:20. Lisa lives far from John (15 minutes driving). Mike leaves home between 7:15 and 7:20 and it takes him 20 minutes to go to the theater. $movie_1, movie_2$ and $movie_3$ start at 7:30, 7:45 and 7:55 and finish at 9:00, 9:10 and 9:20 respectively.

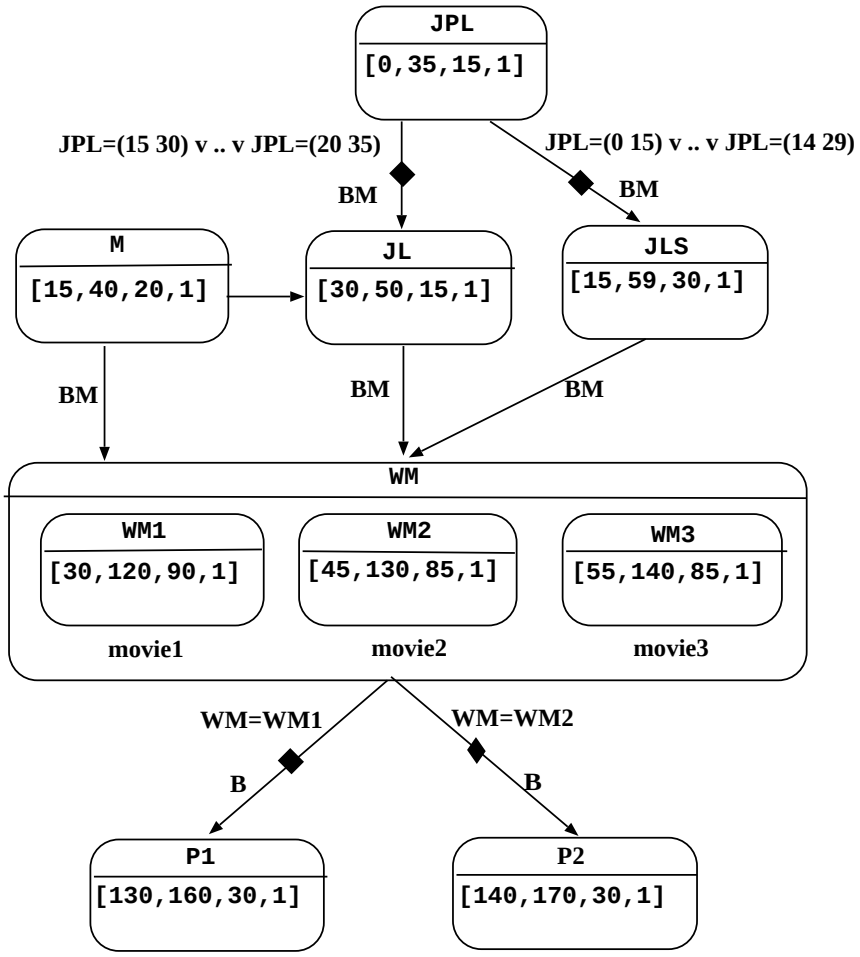


Fig. 1. CCTCSP of example 1.

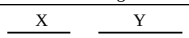
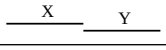
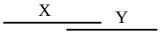
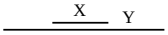
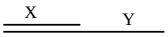
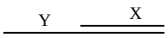
Relation	Symbol	Inverse	Meaning
X Before Y	B	Bi	
X Equals Y	E	E	
X Meets Y	M	Mi	
X Overlaps Y	O	Oi	
X During Y	D	Di	
X Starts Y	S	Si	
X Finishes Y	F	Fi	

Table 1. Allen primitives.

The goal here is to check if this story is consistent (has a feasible scenario). The story can be represented by the CCTCSP of figure 1. There are 6 events JPL, JL, JLS, M, $P1$ and $P2$ and 1 composite variable WM representing the following information :

- JPL : *John will pick Lisa up.*
- JL : *John and Lisa are going to see a movie.*
- JLS : *John and Lisa will stop at a convenient store.*
- M : *Mike is going to see a movie.*
- $P1$: *John, Mike and Lisa will stop at a Pizza place after watching* $movie_1$.
- $P2$: *John, Mike and Lisa will stop at a Pizza place after watching* $movie_2$.
- WM : *John, Mike and Lisa are watching a movie.* WM can take one of the following three values from its domain : WM_1, WM_2 and WM_3 corresponding to $movie_1, movie_2$ and $movie_3$ respectively.

Each event domain is represented by the fourfold $[begintime$, $endtime$, $duration$, $step]$. In the case of JPL, the domain is $[0, 35, 15, 1]$ where 0 (the time origin corresponding to 7:00) is the earliest start time, 35 is the latest end time, 15 is the duration, and 1 (corresponding to 1 min) is the discretization step. For the sake of simplicity all the events in this story have the same step. Arcs represent either a compatibility constraint or an activity constraint (case of arcs with diamond) between variables. The compatibility constraint is denoted by one or more qualitative relations. The activity constraint shows the condition to be satisfied and the qualitative relation between the two variables in case the condition is true. Each qualitative relation is a disjunction of some Allen primitives [9]. For example, the relation BM between JPL and JL denotes the disjunction $Before \vee Meets$.

3 Constraint Propagation for Solving CCTCSPs

Different methods for solving conditional CSPs have been reported in the literature [25, 22, 24, 30]. In [25], all possible CSPs are first generated from the CCSP to solve. CSP techniques are then used on the generated CSPs in order to look for a possible solution. Dependencies between the activity constraints are considered in order to generate a directed a-cyclic graph (DAG), where the root node corresponds to the set of initially active variables. Activity constraints are applied during the derivation of one total order from the partial order given by the resulting DAG. In [22, 24] resolution methods have been proposed and are directly applied on CCSPs. Maintaining arc consistency (MAC) is used to prune inconsistent branches by removing inconsistent values during the search [24]. The solving method starts by instantiating the active variables. For each active variable instantiation, the algorithm first checks the compatibility constraints and then activates the activity constraints. The method will then enforce look-ahead consistency (through arc consistency) along the compatibility constraints and prunes inconsistent values from the domains of future variables. When activity constraints come into play, newly activated variables are added to the set of future variables. MAC is then applied to the set of all active variables. In [30, 24], a CCSP is reformulated into an equivalent standard CSP. A special value "null" is added to the domains of all the variables which are not initially active. A variable instantiation with "null" indicates that the variable does not participate in the problem resolution. The CCSP is transformed into a CSP by including the "null" values. The disadvantage is that, in a large constraint problem, all variables and all constraints are taken into account simultaneously even if some are not relevant to the problem at hand. In the above methods, backtrack search is used for both the generation of possible CSPs and the search for a solution in each of the generated CSPs. Thus, these methods require an exponential time for generating the different CSPs and an exponential time for searching a solution in each generated CSP. Moreover these methods are limited to handle only activity constraints. The other problem of the above methods is the redundant work done when checking at each time the consistency of the same set of variables (subset of a given generated CSP).

The goal of the constraint propagation method we propose for solving CCTCSPs is to overcome, in practice, the difficulty due to the exponential search space of the possible TCSPs generated by the CCTCSP to solve and also the search space we consider when solving each TCSP. In the same way as reported in [22, 24], we use constraint propagation in order to detect earlier later failure. This will allow us to discard at the early stage any subset containing conflicting variables. The description of the method we propose is as follows :

1. The method starts with an initial problem containing a list of initially activated temporal events and composite variables. Arc consistency is applied on the initial temporal events and composite variables in order to reduce some inconsistent values which will reduce the size of the search space. If the temporal events are not consistent (in the case of an empty domain) then the method will stop. The CCTCSP is inconsistent in this case.

2. Following the forward check principle [15], pick an active variable v, assign a value to it and perform arc consistency between this variable and the non assigned active variables. If one domain of the non assigned variables becomes empty then assign another value to v or backtrack to the previously assigned variable if there are no more values to assign to v. Activate any variable v' resulting from this assignment and perform arc consistency between v' and all the active variables. If arc inconsistency is detected then deactivate v' and choose another value for v (since the current assignment of v leads to an inconsistent CCTCSP). If v is a composite variable then assign an event to it (from its domain). Basically, this consists of replacing the composite variable with one event evt of its domain. We then assign a value to evt and proceed as shown before except that we do not backtrack in case all values of evt are explored. Instead, we will choose another event from the domain of the composite variable v or backtrack to the previously assigned variable if all values (events) of v have been explored. This process will continue until all the variables are assigned in which case we obtain a solution to the CCTCSP.

The arc consistency in the above two steps is enforced as follows.

- **Case 1 : the temporal constraint is** (Evt_1, Evt_2) **where** Evt_1 **and** Evt_2 **are two events**
  - The traditional arc consistency [14] is applied here i.e. each value a of Evt_1 should have a support in the domain of Evt_2.
- **Case 2 : the temporal constraint is** (X, Evt) **where** X **is a composite variable and** Evt **is an event**
  - Each value a, from the domain of a given event Evt_{X_k} within X, should have a support in the domain of Evt.
- **Case 3 : the temporal constraint is** (Evt, X)
  - Each value a, from the domain of Evt, should have a support in at least one domain of the events within X.
- **Case 4 : the temporal constraint is** (X, Y) **where** X **and** Y **are two composite variables**
  - Apply case 2 between X and each event Evt_{Y_k} within Y.

Using the above rules, we have implemented a new arc consistency algorithm for CCTCSPs as shown in Figure 2. This algorithm is an extension of the well known AC-3 procedure [14].

4 Approximation methods for CCTCSPs

The method we presented in the previous Section is an exact technique that guarantees a complete solution. The method suffers however from its exponential time cost as we will see in the next Section. In many real-life applications where the execution time is an issue, an alternative will be to trade the execution time for the quality of the solution returned (number of solved constraints). This can be done by applying approximation methods such as local search and where the quality of the solution returned is proportional to the running time. In this Section we will study the applicability of a local

```
REVISE(D_i, D_j)
    REVISE ← false
    For each value a ∈ D_i do
        if not compatible(a, b) for any value b ∈ D_j then
                remove a from D_i
                REVISE ← true
        end if
    end for


REVISE_COMP(D_i, D_j)
    REVISE_COMP ← false
    if i is a single variable and j is a composite variable
        D_tmp ← ∅
        For each event k ∈ D_j do
            D ← D_i − D_tmp
            REVISE_COMP ← REVISE_COMP OR REVISE(D, D_k)
            D_tmp ← D_tmp ∪ D
        end for
        D_i ← D_tmp
    end if
    if i is a composite variable and j is a single variable
        For each event k ∈ D_i do
            REVISE_COMP ← REVISE_COMP OR REVISE(D_k, D_j)
        end for
    end if
    if i and j are composite variables
        For each event k ∈ D_i do
            REVISE_COMP(D_k, D_j)
        end for
    end if


AC − 3 − CCTCSP
            Given a graph G = (X, U)
            Q ← {(i, j)|i, j ∈ U}
            while Q ≠ Nil do
                Q ← Q − {(i, j)}
                if i or j is composite variable
                    if REVISE_COMP(D_i, D_j) then
                            Q ← ∪{(k, i)|k, i ∈ U and k ≠ j}
                    end if
                else  if REVISE(D_i, D_j) then
                            Q ← ∪{(k, i)|k, i ∈ U and k ≠ j}
                    end if
                end if
            end while
```

Fig. 2. AC-3 for CCTCSPs.

search technique based on the Min-Conflict-Random-Walk (MCRW) [21] algorithm for solving CCTCSPs. MCRW has already been applied to solve TCSPs [20]. Basically, the method consists of starting from a complete assignment of temporal intervals to events and iterates by improving at each step the quality of the assignment (number of solved constraints) until a complete solution is found or a maximum number of iterations is reached. Given the dynamic aspect of CCTCSPs (some variables are added|removed dynamically during the resolution process) we propose the following algorithm based on MCRW for solving CCTCSPs.

MCRW-CCTCSP

1. The algorithm starts with a random assignment of values to the initial variables. If the initial variable is an event then it will be randomly assigned a value (temporal interval) from its domain. In the case where the initial variable is composite then it will be replaced by one variable selected randomly from its domain. This latter variable will then be randomly assigned a value from its domain.

2. Activate any variable where the activating condition is true and randomly assign to it a value from its domain as shown in the previous step.
3. If a complete solution is not found and the maximum number of iterations is not reached, randomly select an active variable v and proceed with one of the following cases:
   - If v belongs to the domain of a given composite variable X then select the pair $< v_i, int_{v_i} >$ that increases the quality of the current solution (number of solved constraints). v_i belongs here to the domain of X and int_{v_i} is a value of v_i's domain,
   - otherwise, assign to v a value that increases the quality of the solution.
4. Deactivate any variable activated by the old assignment of v and goto 2.

5 Experimentation

In order to evaluate the methods we propose, we have performed experimental tests on randomly generated consistent CCTCSPs. The experiments are performed on a PC Pentium 4 computer under Linux system. All the procedures are coded in C/C++. Consistent CCTCSPs are generated from consistent TCSPs. A consistent TCSP of size N (N is the number of variables) has at least one complete numeric solution (set of N numeric intervals satisfying all the constraints of the problem). Thus, to generate a consistent TCSP we first randomly generate a numeric solution (set of N numeric intervals), extract the symbolic Allen primitives that are consistent with the numeric solution and then randomly add other numeric and symbolic constraints to it. After generating a consistent TCSP, some of the temporal events are randomly picked and grouped in subsets to form composite variables. Each activity constraint $V_i \stackrel{V_i=a}{\rightarrow} V_j$ is generated by randomly choosing a pair of variables (V_i , V_j) and a value a from the domain of V_i. This activity constraint activates the variable V_j if V_i is activated and is assigned the value a. The generated TCSPs are characterized by their tightness, which can be measured, as shown in [31], as the fraction of all possible pairs of values from the domain of two variables that are not allowed by the constraint. The tests we have performed compare the following four propagation strategies.

Forward Check (FC). This is the strategy we have described in Section 3 which consists basically of maintaining the arc consistency, during the search, between the current variable (the variable that we are assigning a value) and the future active variables (variables not yet assigned) sharing a constraint with the current variable.

Full Look Ahead (FLA). This strategy maintains a full arc consistency on the current and future active variables.

FC+. Same as FC except that the applicability of the arc consistency is extended to non active variables as well.

FLA+. Same as FLA except that the applicability of the arc consistency is extended to non active variables as well.

Figure 3 and table 2 present the results of comparative tests performed on random CCTCSPs where the total number of variables is 150 including 10 composite variables.

The domain sizes of composite variables and events (including those belonging to the composite variables domains) are respectively 5 and 30. The number of activity constraints is 500. In each test, the methods are executed on 100 instances and the average running time (in seconds) is taken. The left chart of figure 3 presents comparative results of the four constraint propagation strategies when the number of initial variables varies from 30 until 100. The tightness is equal here to 0.06 which corresponds to the hardest problems. The right chart of figure 3 presents the comparison of the four strategies when the tightness of the TCSPs, from which the CCTCSPs are generated, varies from 0.1 to 0.7. The number of initial variables is equal to 80. As we can easily see FC and FC+ outperform FLA and FLA+ in all cases in the left chart and in most of the cases in the right chart. However, in the right chart, the strategy of choice in the phase transition is FLA+ since it is the only strategy which returns a complete solution in these situations. In both charts FC and FC+ have similar running times. Table 2 compares the four constraint propagation strategies and the MCRW method we described in Section 4, when the percentage of constraints varies from 0.1 (10%) until 1 (100% which corresponds to a complete constraint graph). Since MCRW is an approximation method, we report in each case the number of times (in percent) this method succeded to provide a complete solution. As we can easily see, MCRW is the fastest method for under constrained problems (where the percentage of constraints is between 0.1 and 0.3) while some propagation strategies (FC and FLA) fail sometimes to to find a solution. For middle and over constrained problems, MCRW does not always guarantee a complete solution when the percentage of constraints is between 0.4 and 0.5 and fails in all the cases when the percentage is between 0.6 and 0.9. FC, FC+ and FLA also have difficulty to find a solution in the case of middle constrained problems and the method of choice in this case is FLA+. In the case of over constrained problems (0.8 to 1) FC+ is the fastest complete methods while MCRW has better running time but succeds only in 30% of the cases for complete graphs.

% of	MCRW		FC	FC+	FLA	FLA+
Cons	Time	success(%)				
0.1	0.1	100	10	11	173	173
0.2	0.1	100	10	10	174	173
0.3	0.1	100	-	11	-	193
0.4	8.7	70	-	-	-	209
0.5	14.8	70	-	19	-	987
0.6	18.3	0	12	12	200	209
0.7	18.5	0	-	-	201	187
0.8	16.9	0	-	16	-	220
0.9	17.9	0	16	16	198	194
1	14.9	30	22	22	217	204

Table 2. Comparative tests on random CCTCSPs.

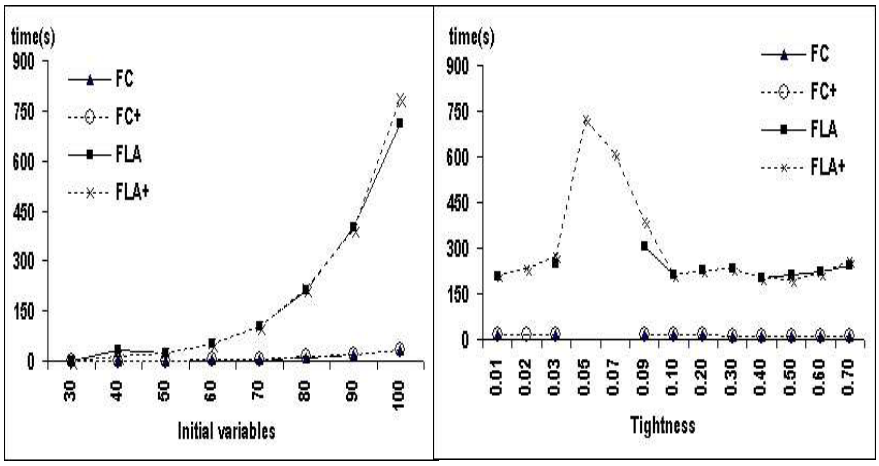


Fig. 3. Comparative tests on random CCTCSPs.

6 Conclusion

We have presented in this paper a CSP based framework for representing and managing numeric and symbolic temporal constraints, activity constraints and composite variables with a unique constraint network that we call Conditional Composite Temporal Constraint Satisfaction Problem (CCTCSP). Solving a CCTCSP consists of finding a solution for one of its possible TCSPS. This requires an algorithm with $O(D^N d^M)$ time cost where N, D, M and d are respectively the number of events and their domain size, the number of composite variables and their domain size. In order to overcome this difficulty in practice, we have proposed 2 methods respectively based on constraint propagation and stochastic local search. Constraint propagation prevents earlier later failure which improves, in practice, the performance in time of the backtrack search. On the other hand, because of its polynomial time cost, the stochastic local search method has better time performance than constraint propagation but does not always guarantee a complete solution. Experimental tests we have performed on randomly generated CCTCSPs demonstrates the efficiency of MCRW method for under constrained problems while variants of the full look ahead and the forward check strategies are the methods of choice respectively for middle constrained and over constrained problems. For these kinds of problems MCRW can be used in case we want to trade search time for the quality of the solution returned (number of solved constraints).

References

1. Baptiste, P., Pape, C.L.: Disjunctive constraints for manufacturing scheduling : Principles and extensions. In: Third International Conference on Computer Integrated Manufacturing, Singapore (1995)
2. Ghallab, M., Laruelle, H.: Representation and Control in IxTeT, a Temporal Planner. In: AIPS 1994. (1994) 61–67
3. Laborie, P., Ghallab, M.: Planning with Sharable Resource Constraints. In: IJCAI-95. (1995) 1643–1649
4. Laborie, P.: Resource Temporal Networks: Definition and Complexity. In: Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03). (2003) 948–953
5. Song, F., Cohen, R.: Tense interpretation in the context of narrative. In: AAAI'91. (1991) 131–136
6. Hwang, C., Shubert, L.: Interpreting tense, aspect, and time adverbials: a compositional, unified approach. In: Proceedings of the first International Conference on Temporal Logic, LNAI, vol 827, Berlin (1994) 237–264
7. Golumbic, C., Shamir, R.: Complexity and algorithms for reasoning about time: a graphic-theoretic approach. Journal of the Association for Computing Machinery **40(5)** (1993) 1108–1133
8. Dean, T.: Using Temporal Hierarchies to Efficiently Maintain Large Temporal Databases. JACM (1989) 686–709
9. Allen, J.: Maintaining knowledge about temporal intervals. CACM **26** (1983) 832–843
10. Vilain, M., Kautz, H.: Constraint propagation algorithms for temporal reasoning. In: AAAI'86, Philadelphia, PA (1986) 377–382
11. Dechter, R., Meiri, I., Pearl, J.: Temporal Constraint Networks. Artificial Intelligence **49** (1991) 61–95

12. van Beek, P.: Reasoning about qualitative temporal information. Artificial Intelligence **58** (1992) 297–326
13. Montanari, U.: Fundamental properties and applications to picture processing. Information Sciences **7** (1974) 95–132
14. Mackworth, A.K.: Consistency in networks of relations. Artificial Intelligence **8** (1977) 99–118
15. Haralick, R., Elliott, G.: Increasing tree search efficiency for Constraint Satisfaction Problems. Artificial Intelligence **14** (1980) 263–313
16. Kautz, H., Ladkin, P.: Integrating metric and qualitative temporal reasoning. In: AAAI'91, Anaheim, CA (1991) 241–246
17. Meiri, I.: Combining qualitative and quantitative constraints in temporal reasoning. Artificial Intelligence **87** (1996) 343–385
18. Thornton, J., Beaumont, M., Sattar, A., Maher, M.: A Local Search Approach to Modelling and Solving Interval Algebra Problems. Journal of Logic and Computation **14** (2004) 93–112
19. Mouhoub, M., Charpillet, F., Haton, J.: Experimental Analysis of Numeric and Symbolic Constraint Satisfaction Techniques for Temporal Reasoning. Constraints: An International Journal **2** (1998) 151–164, Kluwer Academic Publishers
20. Mouhoub, M.: Reasoning with numeric and symbolic time information. Artificial Intelligence Review **21** (2004) 25–56
21. Selman, B., Kautz, H.: Domain-independent extensions to gsat: Solving large structured satisfiability problems. In: IJCAI-93. (1993) 290–295
22. Mittal, S., Falkenhainer, B.: Dynamic constraint satisfaction problems. In: Proceedings of the 8th National Conference on Artificial Intelligence, Boston, MA, AAAI Press (1990) 25–32
23. Sabin, D., Freuder, E.C.: Configuration as composite constraint satisfaction. In Luger, G.F., ed.: Proceedings of the (1st) Artificial Intelligence and Manufacturing Research Planning Workshop, AAAI Press (1996) 153–161
24. Sabin, M., Freuder, E.C., Wallace, R.J.: Greater efficiency for conditional constraint satisfaction. Proc., Ninth International Conference on Principles and Practice of, Constraint Programming - CP 2003 **2833** (2003) 649–663
25. Gelle, E., Faltings, B.: Solving mixed and conditional constraint satisfaction problems. Constraints **8** (2003) 107–141
26. Dechter, R., Dechter, A.: Belief maintenance in dynamic constraint networks. In: 7th National Conference on Artificial Intelligence, St Paul (1988) 37–42
27. Jónsson, A.K., Frank, J.: A framework for dynamic constraint reasoning using procedural constraints. In: ECAI 2000. (2000) 93–97
28. Frank, J., Jónsson, A.K.: Constraint-based attribute and interval planning. Constraints **8** (2003) 339–364
29. Tsamardinos, I., Vidal, T., Pollack, M.E.: CTP: A New Constraint-Based Formalism for Conditional Temporal Planning. Constraints **8** (2003) 365–388
30. Gelle, E.: On the generation of locally consistent solution spaces in mixed dynamic constraint problems. Ph.D.thesis **1826** (1998) 101–140
31. Sabin, D., Freuder, E.C.: Contradicting conventional wisdom in constraint satisfaction. In: Proc. 11th ECAI, Amsterdam, Holland (1994) 125–129

Heuristic Policy Analysis and Efficiency Assessment in Constraint Satisfaction Search*

Richard J. Wallace

Cork Constraint Computation Centre and Department of Computer Science
University College Cork, Cork, Ireland
r.wallace@4c.ucc.ie

Abstract. This paper argues that assessments of search effort made within a new framework for characterizing heuristic performance based on adherence to optimal policies can elucidate many differences in search effort that occur from using different variable ordering heuristics. After providing a brief overview of this framework and discussing the manner in which adherence to a policy can be measured, the paper presents results from a series of experiments to show that many complex patterns of performance can indeed be explained by assessments based on such measures of adherence. In particular, differences found for different categories of heuristics and differences in heuristic performance related to amount of propagation (specifically, MAC versus forward checking) can be elucidated by this approach to performance assessment.

1 Introduction

In order to study algorithms and their implementation, we need to have methods of assessment. And to use methods of assessment properly, we need to understand something of the basic mechanics of algorithmic strategies. For CSP search, the proper approach to assessment is not always obvious. For example, while the concept of a search tree is a familiar one, the means of assessing search by taking measures that reflect the size of the search tree is subject to pitfalls. Recent work has shown this for the common measure of backtracks, which is not necessarily monotonic in the size of the search tree. This work also shows that there are potential problems even for the straightforward measure of total search nodes [1].

It has long been known that variable and value selection have enormous effects on the efficiency of search, which makes them an important part of any search algorithm implementation. However, the basis for these differences is still something of a terra incognita. Moreover, as shown earlier [2] and in the work below, heuristics can interact very strongly with other features of an algorithm such as the degree of constraint propagation (as well as with features of the problem, which is perhaps better recognized). The proper means of assessing differences in heuristic performance is, therefore, also important and is not straightforward.

One rule of good performance is often cited in this connection: the fail-first principle (sometimes confused with the min domain heuristic - more on this below). But as it is

* This work was supported by Science Foundation Ireland under Grant 00/PI.1/C075.

usually stated, this principle is a vaguely formulated goal whose rationale is not entirely clear. Also, because of the work of Haralick and Elliott [3], this principle leads to an emphasis on minimum domain size; the difficulties that this leads to (see below) indicate the limitations of assessments based on this principle alone.

Recently, a new framework was proposed for characterizing heuristic performance. The basic idea is to describe performance in terms of the choices that a heuristic would make if it were in some sense 'perfect', so that it could always make the best choice under the circumstances. (How "best" is defined is described below.) The novelty of this approach lies in the fact that such optimal policies cannot be determined as such in the sense of specifying courses of action. In fact, since there is not a single policy that is in force at all times during search (see next section), it cannot even be decided *which* policy a heuristic should adhere to at a given point in search, even if it were able to do so. Despite this degree of abstraction, a policy framework of this character is of potential value if we can measure how well a heuristic adheres to a given policy during search. This gives us a way of characterizing heuristic performance in terms of basic measures of search quality, rather than simply reporting on overall performance. Previous work has shown how to measure such adherence [4] [5] [6].

The purpose of the present paper is to elaborate on the relations between the policy framework and assessment of search effort. This, in turn, should improve our ability to measure *and comprehend* differences in search effort due to different heuristics. This means that the present paper will be focus on measurement of heuristic performance; although this is not the only area of interest in the area of assessment, it is a subtopic clearly worthy of investigation in its own right.

The next section gives a brief overview of the policy framework. Section 3 discusses performance measures that measure the extent to which a heuristic adheres to a given policy. Section 4 presents the results of experiments which evaluate differences in heuristics and algorithms in terms of measures related to search policies. Section 5 gives conclusions.

2 The Policy Framework

The present work is informed by a recently developed framework for characterizing the performance of search heuristics. This framework has two primary elements. A *policy* identifies goals or end-results that are desirable. A *heuristic* is a rule that is followed to make a decision.

For search problems, there is an overall policy of minimizing search effort in terms of the number of decisions that must be made. In this context, two subordinate policies can be distinguished depending on the state of search. When search is in a state that has solutions in its subtree, search effort will be minimized by making decisions to remain on a path to a solution. As this suggests making decisions to move to the most promising subtree, we call this the *promise policy*. However, for hard problems the best choice will not be made in all cases and search may enter a state where the subtree below it does not contain any solutions. In this case, to minimize effort search should fail as quickly as possible so it can return to a path that leads to a solution. We call this the *fail-first policy*.

Heuristics are based on features of the situation that serve to distinguish choices, so that a selection in these terms increases the likelihood of achieving a goal. In CSP search, these are the variable and value ordering "rules" that exist in the constraint literature (e.g.smallest domain first, Brelaz [7], domdeg [8]). The rationale usually given for variable ordering heuristics is related to the fail-first policy in some form, while that for value ordering is related to promise. However, recent work, which has shown how to assess variable ordering heuristics in terms of promise, indicates that this policy must also be taken into account in any complete evaluation of these heuristics [4] [9].

To understand the relation between policies and heuristics, notice that our two policies are based on a partition of the search nodes into those that have solutions in their subtree ("good" nodes) and those which do not ("bad" nodes). If the partition of a node is known, the policy which leads to minimal search effort is given. Achieving that goal usually does involve heuristics for two reasons. First, we do not typically know which policy to adhere to because we do not know if the current node is good or bad. Second, even knowing a policy, we do not know *how* to adhere to it.

The contribution of heuristic decisions to performance should depend on how well the heuristic conforms to either subordinate policy. Our initial expectation was that adherence to the promise policy will make a difference to search for problems with many solutions. As problems become more difficult, the proportion of time exploring bad subtrees becomes greater, so that the fail-first policy is more often in force and fidelity to that policy should be more important [4]. Note that if problems have no solutions, the only policy relevant to search effort is fail-first.

One of the benefits of the policy framework is that it allows us to characterize the well-known "Fail-First Principle" [3] more precisely than before. In its colloquial form, this principle says that "To succeed, try first where you are most likely to fail." We can restate the principle as saying that search should always proceed as if the fail-first policy were in force. In this case, the best heuristic is one that best conforms to this policy. In this form, the Principle can be seen as a kind of meta-heuristic for selecting a policy under conditions of ignorance, where one does not know what the appropriate policy actually is.

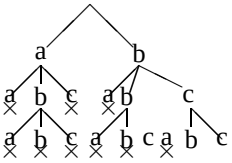
3 Policies and Performance Measures

We are interested in how the policy framework is related to overall performance, and to what degree the former serves to illuminate the latter. Since the policy framework is related to decisions made during search, we use number of search nodes as the basic performance measure, where each node is a partial instantiation of the variables. Thus, every time an assignment is extended by assigning a value to another variable (variable k+1), and every time the current variable (k) is given a different assignment, we consider that an additional search node has been generated.

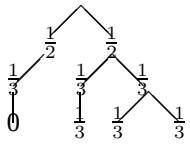
Earlier work has shown how to measure the adherence of a variable ordering heuristic to the different policies. For the promise policy, we define such a measure as the mean likelihood of choosing a value that will lead to a solution across all paths in the (all-solutions) search tree [4] [5]. In the present work, values for this measure were obtained by carrying out an exhaustive search while collecting sums of products that are

returned at successively higher levels of the search tree. Summing is done across the values at a given level of search, and products are taken along search paths. This method is an improvement over earlier Monte Carlo methods of estimating this measure used in [4] [5], since it is accurate up to the degree of precision in the underlying calculations and can be used for much more difficult problems than the earlier procedure.

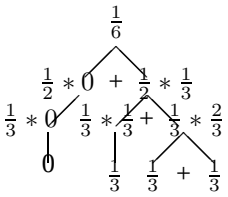
To clarify the process, consider the following example from [4]. For this problem, the search tree for simple backtracking (no filtering), using lexical variable ordering is:



To calculate promise, we consider the probability of choosing each viable value from a domain, when any value is equally likely to be chosen:



Based on these values, we calculate sums and products as indicated below:



So the overall promise for this problem and this consistency algorithm is $\frac{1}{6}$. The same calculations can, of course, be done in combination with any filtering strategy.

If search is for a single solution, the number of mistakes, i.e. assignments leading off a solution path and thus rooting insoluble subtrees, is also useful. In the all-solutions case, this measure correlates well with the basic promise measure, while in the one-solution case, it may capture certain peculiarities of heuristic search better than the basic measure (see below).

For the fail-first policy, an adequate measure must be based on the average size of the insoluble subtree associated with an assignment that *does not* lead to a solution, i.e. the average size of insoluble subtrees rooted at the first bad assignment. This is because an adequate measure must take into account the branching factor as well as the rapidity of failure. (The latter is the different between the level of search at which a mistake was made and the level at which search fails.)

This we call a “mistake tree” to distinguish it from insoluble trees in the ordinary sense [6]. By specifying the root as the first ‘bad’ assignment, we produce an intensity measure, and we are able to compare heuristics across soluble and insoluble problems with respect to the intensity of fail-firstness. We can also normalize the fail-firstness measure by taking the reciprocal of the mean mistake-tree size. This gives us a scale from 0 to 1, where increasing scale values correspond to increasing fail-firstness, and a score of 1 indicates optimal fail-firstness because in this case failure must have occurred at the level of the original mistake.

Note that in comparing soluble and insoluble problems, a more precise analysis involves comparing mistake trees for insoluble problems with mistakes trees for soluble problems that are rooted at level 1 of the search tree. For soluble problems with different parameters, for similar reasons, it is useful to consider mistake tree profiles, i.e. mean tree size at different levels of the search tree.

Other candidate measures of fail-firstness such as average depth of failure and number of failures are affected by promise as well as fail-firstness and for this reason are not adequate. Fail-length, which is the difference in depth between the initial mistake and an actual failure, avoids this problem, and is therefore a true intensity measure. However, earlier tests have shown that it is essential to take the branching factor into account in measuring fail-firstness as well as the rapidity of failure [6].

In a recent work, it was argued that the number of failures is a better measure than search nodes [1]. “Failures” (here and in our own work) are the number of search nodes associated with a domain wipeout so that the value must be retracted, leading to reassignment or backtracking. (Failed nodes are, therefore, a distinguished subset of the set of search nodes.) The key criteria were monoticity and equivalence under conditions where the authors felt that such properties were desirable. Although search nodes, as well as failures, met the criterion of monotonicity, the former did not meet the criterion of equivalence under certain conditions of branching (n-ary versus binary). However, the number of failures does not correspond to our measure of fail-firstness, which takes interior as well as leaf nodes into account. In addition, the conditions where search nodes did not give an equivalent result are arguably different since a difference in search nodes is a difference in number of decisions. So in the present work, we will continue to use the more common measure. (As an aside, it may be remarked that the same work [1] convincingly shows the inadequacy of backtracks as a measure of effort, since in addition to being only a partial tally of decisions made during search, it also has some disturbing non-monotonic properties.)

Number of failures may be a useful measure in the all-solutions case. Here, node counts include subtrees where all values are part of a solution. In particular, given that there are strategies for determining when search is in such a state, a measure that doesn’t include such subtrees may be preferable to one that does.

4 Relating Adherence to Optimal Policies to Overall Search Effort

4.1 Experimental methods

Heuristics used in basic tests included well-known heuristics based on simple CSP parameters, heuristics chosen for their analytic properties with respect to features of search

(the FFx series [10] and the promise variable ordering heuristic [11]), and a few other heuristics that have been used in a project on learning heuristics [12].

The initial analyses were based on a set of twelve heuristics (abbreviations in parentheses are those used in the following tables):

- Minimum domain size (dom). Choose a variable with the smallest current domain size
- Minimum domain over static degree (d/dg). Choose a variable for which this quotient is minimal.
- Minimum domain over forward degree (d/fd). Choose a variable for which this quotient is minimal.
- Maximum forward degree (fd). Choose a variable with the largest number of neighbors (adjacent nodes) in the set of uninstantiated variables.
- Maximum backward degree (bkd). Choose the variable with largest number of neighbors in the set of instantiated variables.
- Maximum product of static degree and forward degree (dg*fd).
- Maximum (future) edgesum (edgsm). Choose an edge between future (uninstantiated) variables for which the sum of the degrees of the two adjacent variables is maximal, then choose the variable in this pair with the largest forward degree.
- FF2 (ff2) The variable, v_i, chosen is the one that maximizes $(1 - (1 - p_2^m)^{d_i})^{m_i}$, where m_i is the current domain size of v_i, and d_i is the future degree of v_i. The FF2 heuristic takes into account an estimate (based on the initial parameters of problem generation) of the extent to which each value of v_i is likely to be consistent with the future variables of v_i.
- This heuristic (ff3) FF3 builds on FF2 by using the current domain size of future variables rather than m. The variable, v_i, chosen is the one that maximizes the following expression, where C is the set of all constraints in the problem, F is the set of unassigned variables, and $P = p_2$.

$$(1 - \prod_{(v_i,v_j)\in C, v_j \in F} (1 - P^{m_j}))^{m_i} \tag{1}$$

- FF4 (ff4) This heuristic modifies FF3 by using the current tightness, $P = p_{ij}$, of the future constraints (the fraction of tuples from the cross-product of the current domains that fail to satisfy the constraint) instead of p_2.
- Maximum promise (prom). Choose the variable with the largest summed promise values across its domain. (Promise for a value is the product ($\prod$) of the supporting values taken across all domains of neighboring future variables. Geelen's heuristic chose the smallest sum, but this proved to be an anti-heuristic, at least when used with lexical value ordering.)
- Static degree (stdeg). Order variables by descending degree in the constraint graph.
- Minimum kappa (kappa) This is the heuristic kappa of [13], which is designed to branch in to the subproblem that minimizes kappa, by choosing the most constrained variable to branch on. Selection is guided by the kappa formula adjusted to reflect the subproblem resulting from selection.
- Extended dynamic variable ordering based on d/fd (DVO1*) In the notation of [14], H_1_DD_x, a DVO extended to the immediate neighborhood of a variable and using

the multiplication operator to combine terms from different variables (cf. formula (5) of [14]). Specifically, for each future variable x_i this heuristic calculates a sum of products of its d/fd and the d/fd of every adjacent variable, divided by the square of ||fd|| for x_i.

All but static degree involve dynamic features of the problem. In all cases, ties were broken according to the lexical order of the variable labels. Values were chosen according to their lexical order.

It should be emphasized again that the FFx series and the promise heuristic are basically diagnostic tools used in earlier work and carried over to the present work for comparison. In fact, in this work they are essentially 'flawed oracles'; hence, the work they do in making a selection is not included in the constraint checks

Tests in this paper were done with homogeneous random CSPs. Problems were generated according to a probability-of-inclusion model for possible constraints, domain elements and constraint tuples (cf. [6]). In all cases graphs were fully connected. Densities given are graph densities (not proportion of edges added to a spanning tree). In the problem sets discussed in this paper, problems were in the critical region and all problems had solutions. For the <30,8,0.31,0.34> problems, the average number of solutions was 487. For the <50,10,0.18,0.37> problems, the average number of solutions was 43,834.

The algorithm used in these experiments was MAC-3 coded in lisp. Tests were done on a Unix server using Xlisp (although this is not of critical importance because the results are in terms of search nodes). It may be noted in passing that constraint checks showed a similar pattern of results across these heuristics.

4.2 Results

Tests with MAC. Table 1 shows statistics (means) for search effort for both the all-solutions and the one-solution problems. Obviously, there are considerable differences in efficiency with different search heuristics.

Table 1. Measures of Search Effort with MAC

heuristic	all solutions			one solution		
	nodes	failures	ccks(M)	nodes	failures	ccks(M)
dom	229,817	74,783	341	11334	5419	25.9
d/dg	85,224	7,580	33	2076	920	4.2
d/fd	174,161	6,334	27	1621	773	3.4
bkd	610,164	299,203	1054	27391	16808	64.1
ff2	145,736	12,793	57	3148	1398	6.5
ff3	284,451	13,913	52	2579	1383	5.5
ff4	375,732	8,779	35	1562	1004	3.9
fd	264,640	16,104	44	2625	2060	5.9
dg*fd	223,315	13,932	39	2418	1896	5.4
edgsm	264,367	15,907	44	2840	2228	6.3
prom	1,006,215	48,259	121	7777	6189	15.1
stdeg	127,218	11,317	31	2000	1456	4.1
kappa	169,566	7,962	27	1576	1039	3.6
DVO1*	222,206	5,002	19	1142	648	2.6

Note. <50,10,0.18,0.37> problems. Means for 100 problems.

Table 2. Policy Measures (All Solutions)

heuristic	promise	ff	mistakes	bad tree sz	faildepth
dom	0.00027	0.00118	407	845	12.4
d/dg	0.00060	0.00578	148	173	10.4
d/fd	0.00063	0.00694	136	144	9.4
bkd	0.00018	0.00085	637	1177	10.0
ff2	0.00057	0.00347	171	288	11.3
ff3	0.00048	0.00474	230	211	10.1
ff4	0.00058	0.00820	183	122	7.6
fd	0.00037	0.00719	231	139	5.5
dg*fd	0.00040	0.00752	220	133	5.6
edgsm	0.00037	0.00704	230	142	5.5
prom	0.00025	0.00299	293	334	6.2
stdeg	0.00040	0.00840	210	119	5.9
kappa	0.00059	0.00980	181	102	6.3
DVO1*	0.00069	0.01111	146	90	7.7

Notes. <50,10,0.18,0.37> problems. Measures are means for 100 problems. "ff" is reciprocal of bad tree sz.

Note that there are some important differences in the pattern of results for the one- and all-solutions cases, when search nodes are considered. In contrast, the data for failures are more similar (although even with this measure differences between heuristics are not always in the same direction).

Some of these data show that there are, indeed, difficulties with the search-nodes measure in the all-solutions case. This is especially true of the promise heuristic; note the large discrepancy between nodes and failures. This occurred because this heuristic

tends to choose variables with relatively large domains, so that when a subtree is entered in which most or all values are viable, the subtree is explored in an inefficient manner.

Tables 2 and 3 show policy-related measures, for all-solutions and one-solution search, respectively. As already noted, "promise" is the basic promise measure, and "mistakes" is also related to adherence to this policy, which is especially useful in the one-solution case. "bad tree size" (mistake-tree size), "faildepth", "mistake depth" and "fail length" are all fail-first measures.

These results indicate that poor performance tends to be associated with low values of the promise measure *and* with larger mistake trees. In other words, weak heuristics are less successful in adhering to either policy than stronger heuristics. However, there are also many interesting tradeoffs in these performance measures.

In particular, there are no consistent across-the-board differences between good and bad heuristics. For example, consider the differences among (min) domain/static-degree, (min) domain/forward-degree and (max) forward degree. For nodes and failures in the one-solution case and for failures in the all-solutions case, the ordering (good $>$ bad) is d/fd $>$ d/d $>$ fd. For the difference between d/d and d/fd, the data in Tables 2 and 3 show that they are very similar with respect to promise, but that d/fd generates smaller mistake trees. On the other hand, fd has a lower degree of promise, while generating mistake trees that are similar in size to those generated by d/fd. To take another comparison, ff2 is inferior to the domain/degree heuristics although it has an equally high promise value; this is because its mistake trees are on average much larger. As a final example, in the one-solution case stdeg has relatively poor promise in comparison with contention heuristics, but it has very strong fail-firstness (reflected in its small mistake-trees); consequently, it is one of the better-performing heuristics (Table 1).

Table 3. Search & Policy Measures (One Solution)

heuristic	$\|\mathrm{dom}\|$	fwdeg	mistakes	bad tree sz	mistakedepth	faillength
dom	1.87	7.2	9.4	1077	6.5	9.4
d/dg	1.78	7.3	8.4	211	6.0	7.5
d/fd	1.89	7.9	8.6	162	5.5	6.6
bkd	2.57	7.7	11.9	2046	5.6	7.3
ff2	1.77	7.3	8.0	334	6.0	8.1
ff3	2.07	7.8	9.6	233	5.7	6.9
ff4	2.57	8.1	10.0	140	4.6	4.7
fd	4.13	9.7	14.3	171	3.5	3.4
dg*fd	3.90	9.6	13.9	165	3.6	3.5
edgsm	4.27	10.4	14.7	179	3.5	3.4
prom	4.56	9.7	16.6	447	3.7	4.1
stdeg	4.45	9.8	13.4	142	3.8	3.7
kappa	2.77	9.1	11.9	125	3.9	4.1
DVO1*	2.24	8.5	9.5	111	4.8	5.2

Notes. $<$50,10,0.18,0.37$>$. Means for 100 problems. $|\mathrm{dom}|$ is average domain size of the variable chosen for instantiation (hence, it is the average branching factor. "fwdeg" is mean forward degree of the next variable chosen.

Another interesting finding is that the simple min domain heuristic has a rather poor fail-firstness overall. This is undoubtedly because it chooses 'blindly' at the top of the

search tree, since the original domain sizes are all equal, and the initial arc consistency filtering does not remove many values. But it does seem to show that assessments based on the policy framework (i.e. on how well a heuristic adheres to a given policy) give a picture that is at variance with the one based on the original analysis of likelihood of failure [3]. (In the same vein, the max promise heuristic does not actually score very well on the measure of promise; this must be because its branching factor tends to be quite large [Table 3].)

In Table 1 and several other tables, heuristics are grouped. (Groups are separated by horizontal lines in the table). The first two groupings represent two basic categories of heuristics recently identified by factor analysis [2]. The category that includes min domain has been tentatively labeled as "contention heuristics", while the category that includes max forward degree has been labeled as "propagation heuristics". This classification appears to reflect two independent heuristic actions, and most heuristics can be characterized as favouring one or the other action (although, of course, both involve both in differing degree) and the fact that problems differ in their response to them. The third grouping in the table simply serves to separate more recently proposed heuristics that use more elaborate means to select variables. These heuristics still fall into the just-mentioned categorisation: kappa is basically a propagation heuristic, while DVO1* is a contention heuristic with a seemingly better balance between the two kinds of heuristic action [2].

Tables 2 and 3 show that there are some striking patterns of differences in certain of the measures when the basic groups of heuristics are compared, which begins to elucidate the basis for differences in overall performance. Propagation heuristics tend to fail higher in the search tree than contention heuristics (Table 2); this is because mistakes occur at a higher level *and* failures occurs sooner once a mistake has been made (Table 3). There are also many more mistakes prior to finding a solution (Table 3). These differences can be ascribed to certain features of search, in particular, the larger branching factor for propagation heuristics (Table 3). Obviously, other things being equal, selecting variables with larger domains will result in more errors, i.e. diminished adherence to the promise policy. Together, these results give further evidence that assessments made within the policy framework help to elucidate the bases for differences in overall performance.

Tests comparing MAC and forward checking. Table 4 shows some statistics for the all-solutions case for both MAC and forward checking. Smaller problems were used in this case because of the markedly greater difficulty when forward checking was combined with heuristics in the propagation class (see later tables). The Table 4 data show that the expected increase in nodes searched with forward checking is based on differences in promise (reflected in an order-of-magnitude change in the measure of promise) as well as in fail-firstness.

In addition, other differences in performance are elucidated by the policy approach to assessment. For example, in the all-solutions case d/fd does not perform as well as d/d, and this is reflected in failures as well as total search nodes. In this case, fail-firstness measures favor d/fd, but this heuristic shows a serious deficiency in the promise measure, which must be the basis for the overall increase in search effort.

Table 4. MAC vs. FC - Search Effort & Policy Measures (All Solutions)

heuristic	nodes	fails	ccks(T)	promise	ff	mistakes	faildepth	bad tree sz	faillength
					MAC				
dom	2201	506	1,190	0.0068	0.03236	44	6.9	30.9	4.8
d/dg	1455	184	417	0.0086	0.06452	26	5.8	15.5	3.8
d/fd	2286	174	385	0.0092	0.07042	26	5.3	14.2	3.4
					FC				
dom	11,153	2345	231	0.00075	0.00486	96	12.5	205.7	10.1
d/dg	4,633	843	83	0.00082	0.01036	58	11.5	96.5	8.9
d/fd	5,299	1020	86	0.00027	0.01109	66	11.1	90.2	8.6

Notes. <30,8,0.31,0.34>. Means for 100 problems.

A comparison of the data in Tables 5 and 6 gives evidence of a marked difference between MAC and forward checking when propagation heuristics are used, as already noted. From Table 6, we can see that, although there is a degree of difference in the promise measure (here, mistakes) that is related to this difference, the main effect is on the fail-firstness measure. Differences among propagation heuristics are also based on differences in fail-firstness, *viz* the difference between max static degree and the other heuristics in this class.

Table 5. Search & Policy Measures with MAC (One Solution)

heuristic	nodes	$\|\mathrm{dom}\|$	fwdeg	mistakes	bad tree sz	mistakedepth	faillength
dom	262	1.88	6.4	6.3	33	3.9	4.4
d/dg	143	1.80	6.3	6.1	18	3.5	3.5
d/fd	130	1.85	6.5	6.2	15	3.2	3.1
bkd	481	2.43	6.8	8.2	52	3.3	3.4
ff2	163	1.80	6.4	6.1	21	3.5	3.8
ff3	154	1.95	6.5	6.3	18	3.2	3.3
ff4	122	2.16	6.3	6.5	14	2.8	2.4
fd	163	2.92	7.0	8.3	16	2.3	1.8
dg*fd	151	2.78	6.9	8.0	15	2.3	1.9
edgsm	160	3.21	8.3	8.4	16	2.4	1.9
prom	232	3.11	7.2	9.3	21	2.6	2.2
stdeg	147	3.27	7.4	7.9	15	2.4	1.9
kappa	129	2.27	6.8	7.2	14	2.5	2.2
DVO1*	113	2.04	6.6	6.4	12	2.8	2.5

Notes. <30,8,0.31,0.34>. Means for 100 problems.

Table 6. Search & Policy Measures with FC (One Solution)

heuristic	nodes	$\|dom\|$	fwdeg	mistakes	bad tree sz	mistakedepth	faillength
dom	2324	1.33	6.1	8.4	254	6.5	9.6
d/dg	1072	1.35	5.9	8.6	116	6.0	8.6
d/fd	1112	1.41	6.2	9.8	100	6.7	8.3
bkd	27,846	1.73	5.9	14.0	1716	7.4	9.8
ff2	1224	1.34	6.1	8.5	132	5.9	8.8
ff3	2653	2.23	6.6	15.8	158	8.9	7.9
ff4	1230	2.74	6.8	20.3	60	8.6	5.8
fd	50,154	2.86	6.9	23.0	2066	6.5	7.3
dg*fd	26,213	2.65	6.9	20.7	1265	6.5	7.1
edgsm	50,778	2.86	6.9	23.0	2103	6.5	7.3
prom							
stdeg	13,699	3.05	7.2	17.6	749	6.6	7.3
kappa	3594	1.80	6.6	15.1	232	6.7	7.0
DVO1*	2212	1.87	6.4	14.5	146	7.6	8.1

Notes. <30,8,0.31,0.34>. Means for 100 problems. Difficulties with array overflows prevented data for promise from being collected in time, although results appear to be similar to other propagation heuristics.

5 Conclusions and Future Work

The present results seem to bear out the contention made earlier, that by using the policy framework we can better assess heuristic performance. These results have also made it clear that there are quite a number of interesting differences in performance to elucidate. They also indicate that, in contradiction to our original expectations, it is probably necessary to consider both policies for difficult as well as for easy problems.

In future work, we will want to test other recently devised heuristics (impact, weighted degree). We also need to apply the approach to structured problems, where the behaviour of heuristics is sometimes quite different than with problems having unrestricted connectivity and patterns of relatedness within constraint relations. It may also help to elucidate the sometimes troublesome interactions between symmetry-breaking and variable ordering heuristics.

Eventually, of course, we need to move on to better statistical models of performance. However, the present framework can serve as a guide for work in this direction.

Acknowledgements. The present method for calculating the promise measure is due to J. C. Beck.

References

1. Bessière, C., Zanuttini, B., Fernández, C.: Measuring search trees. In: ECAI 2004 Workshop on Modelling and Solving Problems with Constraints. (2004) 31–40

2. Wallace, R.J.: Factor analytic studies of csp heuristics. In: Principles and Practice of Constraint Programming-CP'05. (2005) to appear
3. Haralick, R.M., Elliott, G.L.: Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence **14** (1980) 263–314
4. Beck, J.C., Prosser, P., Wallace, R.J.: Toward understanding variable ordering heuristics for constraint satisfaction problems. In: Proc. Fourteenth Irish Artificial Intelligence and Cognitive Science Conference-AICS'03. (2003) 11–16
5. Beck, J.C., Prosser, P., Wallace, R.J.: Variable ordering heuristics show promise. In: Principles and Practice of Constraint Programming-CP'04. LNCS No. 3258. (2004) 711–715
6. Beck, J.C., Prosser, P., Wallace, R.J.: Trying again to fail-first. In: Recent Advances in Constraints. Papers from the 2004 ERCIM/CologNet Workshop-CSCLP 2004. LNAI No. 3419, Berlin, Springer (2005) 41–55
7. Brélaz, D.: New methods to color the vertices of a graph. Communications of the ACM **22** (1979) 251–256
8. Bessière, C., Regin, J.C.: MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. Principles and Practice of Constraint Programming-CP'96 (1996) 61–75
9. Beck, J.C., Prosser, P., Wallace, R.J.: Failing first: An update. In: Proc. Sixteenth European Conference on Artificial Intelligence-ECAI'04. (2004) 959–960
10. Smith, B.M., Grant, S.A.: Trying harder to fail first. In: Proc. Thirteenth European Conference on Artificial Intelligence-ECAI'98, John Wiley & Sons (1998) 249–253
11. Geelen, P.A.: Dual viewpoint heuristics for binary constraint satisfaction problems. In: Proc. Tenth European Conference on Artificial Intelligence-ECAI'92. (1992) 31–35
12. Epstein, S.L., Freuder, E.C., Wallace, R., Morozov, A., Samuels, B.: The adaptive constraint engine. In van Hentenryck, P., ed.: Principles and Practice of Constraint Programming - CP2002. LNCS. No. 2470, Berlin, Springer (2002) 525–540
13. Gent, I., MacIntyre, E., Prosser, P., Smith, B., Walsh, T.: An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In: Principles and Practice of Constraint Programming-CP'96. LNCS No. 1118. (1996) 179–193
14. Bessière, C., Chmeiss, A., Saïs, L.: Neighborhood-based variable ordering heuristics for the constraint satisfaction problem. In: Principles and Practice of Constraint Programming-CP'01. LNCS No. 2239. (2001) 565–569

Declarative Approximate Graph Matching Using A Constraint Approach *

Stéphane Zampelli, Yves Deville, and Pierre Dupont

Université Catholique de Louvain,
Department of Computing Science and Engineering,
2, Place Sainte-Barbe
1348 Louvain-la-Neuve (Belgium)
\{sz,yde,pdupont\}@info.ucl.ac.be

Abstract. Graph pattern matching is a central application in many fields. However, in many cases, the structure of the pattern can only be approximated and exact matching is then far too accurate. This work aims at proposing a CSP approach for approximate subgraph matching where the potential approximation is declaratively included in the pattern graph as optional nodes and forbidden edges. The model, covering both monomorphism and isomorphism problem, also allows additional properties, such as distance properties, between pairs of nodes in the pattern graph. Such properties can be either stated by the user, or automatically inferred by the system. The model is built through the definition of parametric morphism constraints, allowing an efficient implementation of propagators. An Oz/Mozart implementation has been developped. Experimental results show that our general framework is competitive with a specialized C++ Ullman (exact) matching algorithm, while also offering approximate matching.

1 Introduction

Graph pattern matching is a central application in many fields [1]. Many different types of algorithms have been proposed, ranging from general methods to specific algorithms for particular types of graphs. In constraint programming, several authors [2, 3] have shown that graph matching can be formulated as a CSP problem, and argued that constraint programming could be a powerful tool to handle its combinatorial complexity. However, many issues should be considered such as the evaluation of the performance of a CSP approach against traditional algorithms, the development of new global constraints enhancing the pruning, the extension of exact matching to approximate matching.

In many areas, the structure of the pattern can only be approximated and exact matching is then far too accurate. Approximate matching is a possible solution, and can be handled in several ways. In a first approach, the matching

* Acknowledgments: This research is supported by the Walloon Region, project BioMaze (WIST 315432). Thanks also to the EC/FP6 Evergrow project for their computing support.

algorithm may allow part of the pattern to mismatch the target graph (e.g. [4–6]. The matching problem can then be stated in a probabilistic framework (see, e.g. [7]). In a second approach, the approximations are declared by the user within the pattern, stating which part could be discarded (see, e.g. [8]). This approach is especially useful in fields, such as bioinformatics, where one faces a mixture of precise and imprecise knowledge of the pattern structures. In this approach, which will be followed in this paper, the user is able to choose parts of the pattern open to approximation.

Within the CSP framework, a model for graph monomorphism has been proposed by Rudolf [3] and Valiente et al. [2]. Our modeling is based on these works. Sorlin [9] proposed a filtering algorithm based on paths for graph isomorphism and part of our approach can be seen as a generalization of this filtering. A declarative view of matching has also been proposed in [10].

Objectives This work aims at proposing a CSP approach for approximate subgraph matching where the potential approximation is declaratively included in the pattern graph as mandatory/optional nodes/edges. We also want a framework where additional properties between pairs of nodes in the pattern graph, such as distance properties, can be either stated by the user, or automatically inferred by the system. In the former case, such properties can define new approximate patterns. In the latter case, these redundant constraints enhance the pruning.

Results The main contributions of this paper are the following:

- An extension of the CSP model for pattern subgraph matching covering both monomorphism and isomorphism problems, and allowing the specification of additional constraints between pairs of nodes, as well as the derivation of redundant constraints providing more pruning.
- A definition of approximate subgraph matching, including the specification of optional nodes and forbidden edges in the pattern graph, and its associated CSP model.
- A definition of parametric morphism constraints, allowing a simple expression of the above problems.
- An implementation of propagators for the generic constraints.
- Experimentations showing that our general framework is competitive with a specialized C++ Ullman (exact) matching algorithm, while also offering approximate matching.

Outline Sections 2 and 3 introduce basic definitions of subgraph matching and describe the basic framework for monomorphism. Section 4 generalizes the basic monomorphism constraints to parametric constraints in the exact case. Instances of these parametric constraints, such as isomorphism constraints and path constraints, are described in Section 5. Section 6 introduces the approximate matching problem and describes how this problem can be solved by an approximate version of the parametric constraints. Section 7 presents a comparison of our CSP approach with Ullman based graph matching algorithm and shows that the proposed approach is competitive. Section 8 concludes this paper and presents research directions.

2 Background

Before presenting the basic CSP for exact and approximate subgraph matching, we define the notion of subgraph matching.

A **graph** $G = (N, E)$ consists of a **node set** N and an **edge set** $E \subseteq N \times N$, where an edge (u, v) is a pair of nodes. The nodes u and v are the endpoints of the edge (u, v).

The **neighborhood function** $V(a)$ is the set of neighbors of a node a in the underlying graph.

A **subgraph** of a graph $G = (N, E)$ is a graph $S = (N', E')$ where N' is a subset of N and E' is a subset of E.

A **subgraph isomorphism** between a pattern graph $G_p = (N_p, E_p)$ and a target graph $G_t = (N_t, E_t)$ is an injective function $f : N_p \rightarrow N_t$ respecting $(u, v) \in E_p \Leftrightarrow (f(u), f(v)) \in E_t$.

A **subgraph monomorphism** between G_p and G_t is an injective function $f : N_p \rightarrow N_t$ respecting $(u, v) \in E_p \Rightarrow (f(u), f(v)) \in E_t$.

A **subgraph matching** is either a subgraph isomorphism or a subgraph monomorphism.

A constraint model to solve the exact subgraph matching problem has been proposed by several authors [2] [3]. This model focuses on monomorphism and will form our basic monomorphism constraints. The variables $\mathcal{X} = \{x_1, ..., x_n\}$ are the nodes of the pattern graph and their respective domain $D(x_i)$ is the set of target nodes. The assignment must respect two conditions: all variables have a different value and the structure of the pattern must be kept (monomorphism condition). The first condition is implemented with the classical *Alldiff*$(x_1, ..., x_n)$ constraint [11] [12]. The second condition is translated into a monomorphism constraint.

2.1 Monomorphism Constraint

The monomorphism constraint states that if an edge exists between two pattern nodes, then an edge must exist between their corresponding images :

$$\forall\ (i, j) \in E_p : (f(i), f(j)) \in E_t\ .$$

The corresponding basic monomorphism constraint is defined as :

$$MC(x_i, x_j) \equiv (i, j) \in E_p \Rightarrow (x_i, x_j) \in E_t\ .$$

In the rest of the paper, $N = |N_p|$, $E = |E_p|$, $D = |N_t|$ and d is the average degree of the target graph. A classical AC-consistency algorithm would cost $O(ED^2)$ amortized time [2]. By using the problem structure, its amortized complexity can be reduced to $O(NDd)$ [2]. We note n the average degree of the pattern graph.

A global constraint $MC(x_1, ..., x_n)$ can be formulated, instead of having one constraint MC per node pair:

$$MC(x_1, ..., x_n) = \bigwedge_{i,j} MC(x_i, x_j) \ .$$

The global basic monomorphism constraint $MC(x_1, ..., x_n)$ can be expressed as:

$$\forall\ i \in N_p\ \forall\ a \in N_t : |D(x_i) \cap V_t(a)| = 0 \Rightarrow a \notin D(x_j)\ \ \forall\ j \in V_p(i) \ ,$$

where $V_p(i) = \{j \in N_p \mid (i,j) \in E_p\}$ and $V_t(i) = \{j \in N_t \mid (i,j) \in E_t\}$.

The proposed propagator keeps track of relations between all the target nodes and the domain $D(x_i)$ in a structure $S(i,a) = |D(x_i) \cap V_t(a)|$ representing the number of relations between a target node a and $D(x_i)$. Whenever the neighbors of a target node a have no relation with $D(x_i)$, that is when $S(i,a) = 0$, node a is pruned from all neighbors of x_i. The Algorithm 2.1 shows an implementation of the global morphism constraint. It has a $O(NDd)$ amortized time complexity, and the structure $S(i,a)$ has O(ND) spatial complexity [2]. The preprocessing to compute $S(i,a)$ costs $O(NDd)$. The global MC constraint is thus *algorithmically* global as it achieves the same consistency than the original conjunction of constraints, but more efficiently.

Algorithm 1: Morphism Constraint

```
Propagate_MC(i,a)
// Element a exits from D(x_i)
for b ∈ V_t(a) do
    S(i,b) ← S(i,b) − 1
    if S(i,b) = 0 then
        foreach j ∈ V_p(i) do
            D(x_j) ← D(x_j) \ {b}
```

2.2 Local Alldiff Constraint

A redundant constraint pruning the search space has been proposed in [2]. This constraint is a local Alldiff constraint [11], enforcing that the number of candidates available in the union of the domains of x_i's neighbors should not be less than the actual number of x_i neighbors in the pattern graph:

$$LA(x_i) \equiv |\cup_{j \in V_p(i)} D(x_j) \cap V_t(x_i)| \geq |V_p(i)| \ . \tag{2}$$

An algorithmic global constraint $LA(x_1, ..., x_n)$ can be formulated, instead of having one constraint LA per node :

$$LA(x_1, ..., x_n) \equiv \bigwedge_i LA(x_i) \ .$$

A structure $CT(i,a) = |\cup_{j \in V_p(i)} D(x_j) \cap V_t(a)|$ is updated through the use of an intermediate structure $R(i,a) = |\{j \in V_p(i) \mid a \in D(x_j)\}|$. The structure

$R(i,a)$ counts the number of neighbors of x_i which have a in their domain. Whenever $R(i,a)$ equals 0, $CT(i,b)$ diminishes by 1 for all b in the neighbor of a in the target graph. The expression $|V_p(i)|$ can be obtained in $O(1)$. Algorithm 2 describes an implementation of the $LA(x_1,...,x_n)$ constraint.

The amortized complexity of this redundant constraint is $O(NDd)$ and its space complexity is $O(ND)$. The preprocessing time to build the $CT(i,a)$ and $R(i,a)$ structures is $O(NDd)$.

Algorithm 2: Local alldiff constraint

```
Propagate_LA(i,a)
// Element a exits from D(x_i)
for j ∈ V_p(i) do
    R(j,a) ← R(j,a) − 1
    if R(j,a) = 0 then
        foreach b ∈ V_t(a) do
            CT(j,b) ← Ct(j,b) − 1
            if CT(j,b) < |V_t(j)| then
                D(x_j) ← D(x_j) \ {b}
```

3 Generic Subgraph Matching Constraints

In this section we present new parameterized global constraints able to handle different type of constraints. These constraints will be instantiated to different matching constraints.

The MC constraint is redefined as a parametric constraint, taking pattern pair node relations A and target pair node relations B as parameter :

$$MC_p(x_1,...,x_n,A,B) \equiv \bigwedge_{i,j} (i,j) \in A \Rightarrow (x_i,x_j) \in B \ .$$

The propagator of this parametric MC constraint is given by Algorithm 1, where the neighborhood functions $V_p(\cdot)$ and $V_t(\cdot)$ are specialized to the considered instance of the constraint. More precisely, $V_p(i,A) = \{j \in Np \mid (i,j) \in A\}$ and $V_t(a,B) = \{b \in Np \mid (a,b) \in B\}$ respectively. As a consequence, $S(i,a)$ is redefined as $|D(x_i) \cap V_t(a,B)|$.

The LA constraint can also be parameterized with the relations A and B :

$$LA_p(x_i,A,B) \equiv |\cup_{j \in V_p(j,A)} D(x_j) \cap V_t(x_i,B)| \geq |V_p(i,A)|$$
$$LA_p(x_1,...,x_n,A,B) \equiv \bigwedge_i LA(x_i,A,B)$$

Algorithm 2, with suitable specific structures, is a possible implementation for instances of this constraint.

The problem of subgraph monomorphism can then be expressed as :

$$\text{alldiff}(x_1,...,x_n) \wedge MC_p(x_1,...,x_n,E_p,E_t) \wedge LA_p(x_1,...,x_n,E_p,E_t)$$

4 Specifying Additional Constraints

Additional constraints for the matching problem can be expressed as instances of the parametric morphism constraint.

4.1 Isomorphism as Monomorphism Matching

Isomorphism condition states that if an edge does not exist between two pattern nodes, then an edge should not exist between their corresponding images :

$$\forall\ (i,j) \notin E_p : (f(i), f(j)) \notin E_t\ .$$

The problem of subgraph isomorphism can be stated easily by using complementary edge sets $\overline{E}_p = \{(i,j) \in N_p \times N_p \mid (i,j) \notin E_p\ \}$ and $\overline{E}_t = \{(i,j) \in N_t \times N_t \mid (i,j) \notin E_t\ \}$ as parameters :

$$\text{alldiff}(x_1, ..., x_n)\ \wedge\ MC_p(x_1, ..., x_n, E_p, E_t)\ \wedge\ MC_p(x_1, ..., x_n, \overline{E}_p, \overline{E}_t)$$
$$\wedge\ LA_p(x_1, ..., x_n, E_p, E_t)\ \wedge\ LA_p(x_1, ..., x_n, \overline{E}_p, \overline{E}_t)$$

4.2 Path and Shortest Path Distance Constraint

In this section we formulate a new constraint between pair of nodes based on the path and shortest path distance. It can be seen as a generalization of other works based on shortest path distance as filtering and checking methods [9] [13], where only initial filtering and checking is achieved. In our method, the path constraints does this initial filtering but also propagates.

Definition 1 *A node a is at distance k from node b in a graph if and only if there exists a shortest path of distance k between them. $dist(a, b)$ denotes the shortest path distance between a and b.*

A shortest path monomorphism constraint (for a given distance k) can be formulated as $MC_{dist}(x_1, ..., x_n, k) \equiv \bigwedge_{i,j} dist(i,j) = k \Rightarrow dist(x_i, x_j) \leq k$.

Similarly, a shortest path isomorphism constraint (for a given distance k) can be formulated as $MC_{dist}(x_1, ..., x_n, k) \equiv \bigwedge_{i,j} dist(i,j) = k \Rightarrow dist(x_i, x_j) = k$.

Suppose $E_p^k = \{(i,j) \in N_p \times N_p \mid dist(i,j) = k\}$ and $E_t^k = \{(a,b) \in N_t \times N_t \mid dist(a,b) \leq k\}$. Then MC_{dist} is equivalent to $MC_{dist}(x_1, ..., x_n, k) \equiv MC(x_1, .., x_n, E_p^k, E_t^k)$.

The expression $path(i,j,k)$ denotes that there is a path of length k between i and j. The path constraint can be formulated as $MC_{path}(x_1, ..., x_n) \equiv \bigwedge_{i,j} path(i,j,k) \Rightarrow path(x_i, x_j, k)$.

Suppose $E_p^k = \{(i,j) \in N_p \times N_p \mid path(i,j,k)\}$ and $E_t^k = \{(a,b) \in N_t \times N_t \mid path(a,b,k)\}$. We can see that MC_{path} is equivalent to $MC_{path}(x_1, ..., x_n) \equiv MC(x_1, .., x_n, E_p^k, E_t^k)$.

The number of (E_p^k, E_t^k) couples is bound by the diameter of the pattern graph, which is, in the worst case, $O(N)$. The time complexity of all these new

constraints is thus $O(N^2Dd)$ and their spatial complexity $O(N^2D)$. Preprocessing time to compute path and shortest-path distance adjacency matrices is $O(D^3)$.

Shortest path constraints lead to poor pruning when k increases since the average degree of the graphs E_p^k and E_t^k is $O(d^k)$. All path constraints are however redundant, meaning they are necessary conditions of the matching. Only a subset of these constraints can be chosen. One could select only path of distance two and three, resulting in a $O(ND3d) = O(NDd)$ time complexity and a $O(3ND) = O(ND)$ spatial complexity. One could use the path distance only from one specific node to all the another nodes. We call this kind of node an *orbit*. Each orbit will cost an additional $O(NDd)$. One could select specific path distance constraints between chosen nodes.

5 Approximate Subgraph Matching

5.1 Problem Definition

A useful extension of subgraph matching is approximate subgraph matching, where the pattern graph and the found subgraph in the target graph may differ with respect to their structure.

Optional nodes In our framework, the approximation is *declared* upon the pattern graph. Some *nodes* are declared *optional*, i.e. nodes that may not be in the matching. Specifying optional edges in a monomorphism problem is useless as it is equivalent to omitting the edge in the pattern. The status of the edges depends on the optional state of their endpoints. An edge having an optional node as one of its endpoints is optional. An optional edge is not considered in the matching if one of its endpoints is not part of the matching. Otherwise, the edge must also be a part of the matching.

Forbidden edges *Edges* may also be declared as *forbidden* between their two endpoints (u, v), meaning that if u and v are in the domain of f, then (u, v) must not exist in the target graph. A pattern graph with all its complementary edges declared as forbidden induces a subgraph isomorphism instead of a subgraph monomorphism.

A pattern graph with optional nodes and forbidden edges forms an *approximate pattern graph*.

Definition 2 *An **approximate pattern graph** is a tuple (N_p, O_p, E_p, F_p) where (N_p, E_p) is a graph, $O_p \subseteq N_P$ is the set of optional nodes and $F_p \subseteq N_p \times N_p$ is the set of forbidden edges, with $E_p \cap F_p = \emptyset$.*

The corresponding matching is called an *approximate subgraph matching*.

Definition 3 *An **approximate subgraph matching** between an approximate pattern graph $G_p = (N_p, O_p, E_p, F_p)$ and a target graph $G_t = (N_t, E_t)$ is a* partial *function $f : N_p \to N_t$ such that :*

1. $N_p \setminus O_p \subseteq dom(f)$

2. $\forall\, i, j \in dom(f) : i \neq j \Rightarrow f(i) \neq f(j)$
3. $\forall\, i, j \in dom(f) : (i, j) \in E_p \Rightarrow (f(i), f(j)) \in E_t$
4. $\forall\, i, j \in dom(f) : (i, j) \in F_p \Rightarrow (f(i), f(j)) \notin E_t$

The notation $dom(f)$ represents the domain of f. Elements of $dom(f)$ are called the selected nodes of the matching. This means that $dom(f)$ can be represented by a finite set variable. Its lower bound f_{lb} consists of all selected nodes, and its upper bound f_{glb} consists of selected nodes and nodes that could be selected.

Condition 1 requires mandatory nodes to be in the matching. Condition 2 is the injective condition, also present in the exact case. Condition 3 enforces that an edge between two selected endpoints must always be present in the target. Condition 4 forbids the presence of an edge in the matching between node $(f(u), f(v))$ if the edge (u, v) was declared forbidden and u, v are in the matching. According to this definition, if $F_p = \emptyset$ the matching is a subgraph monomorphism, and if $F_p = N_p \times N_p \setminus E_p$, the matching is an isomorphism.

Condition 3 has an important impact on the set of possible matchings, as shown in Figure 1. In this figure, mandatory nodes are represented as filled nodes, and optional nodes are represented as empty nodes. Mandatory edges are represented with plain line, and optional edges are represented with dashed lines. Forbidden edges are represented with a plain line crossed. Intuitively, one could think that edge $(5, 6)$ in the pattern could be discarded, while node 6 could be selected together with edge $(4, 6)$. In fact, because of condition 3, matching of node 6 would require the edge $(5, 6)$ to be present in the target. Only two subgraphs match this pattern as shown on the right side of Figure 1. The nodes and edges not selected in the target graph are grey.

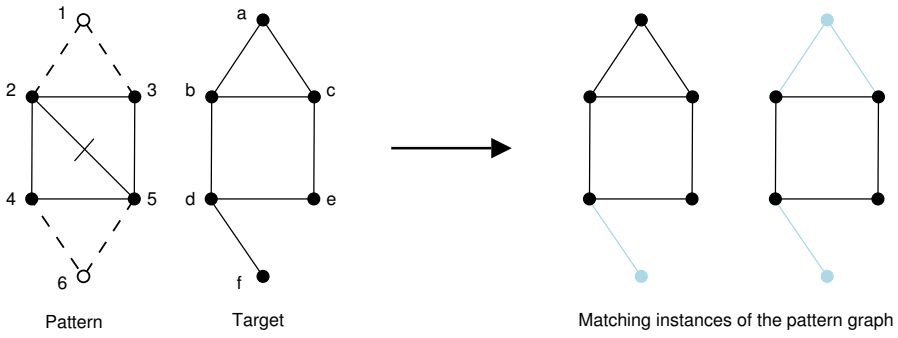


Fig. 1. Example of approximate matching

In an approximate subgraph matching, the number of possible solutions may be higher than in exact matching. One could therefore add some optimization criteria on the results, such as maximizing the number of edges in the matching.

5.2 Parametric Constraints for Optional Nodes

Morphism constraint on this approximate matching should handle the optional nodes, but also be parameterized, because expressiveness and pruning should be kept. The approximate morphism constraint can be defined as follows :

$$MC_{pa}(x_1, ..., x_n, A, B) \equiv \bigwedge_{i,j}(i,j) \in A \wedge i, j \in dom(f) \Rightarrow (x_i, x_j) \in B$$

The former constraint states that a morphism relation between two pattern nodes x_i and x_j must be forced if and only if they are present in the domain of f. Using a MC-like implementation, the MC_{pa} constraint can be rewritten as :

$$\forall\, i \in N_p\ \forall\, a \in N_t : (\ |D(x_i) \cap V_t(a)| = 0 \\ \wedge\ i \in dom(f)\ ) \Rightarrow a \notin D(x_j) \quad \forall\, j \in V_p(i)\ .$$

The additional condition $i \in dom(f)$ states that only selected nodes should propagate under the morphism condition. The propagation of the morphism constraint of an optional i is computed but performed only when i is in the domain of f.

As depicted in Figure 2, all selected nodes propagate in their neighborhood but optional nodes propagate only when they are selected.

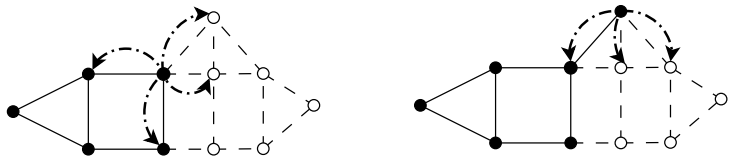


Fig. 2. Pruning method for the approximate morphism condition

MC_{pa} is a simple extension of the implementation of MC_p one (Algorithm 1). If i is not selected and there exists a such that $S(i,a) = 0$, the propagator waits for node i to be selected to trigger the actual propagation.

5.3 Constraints for Forbidden Edges

A constraint for the forbidden edges (condition 4 in the matching) can be obtained by using parameterized MC_{pa} :

$$MC_{pa}(x1, ..., x_n, F_p, \overline{E}_t)$$

The constraints for the approximate matching problem are then :

$$\text{alldiff}(x_1, ..., x_n)\ \wedge\ MC_{pa}(x1, ..., x_n, E_p, E_t)\ \wedge\ MC_{pa}(x1, ..., x_n, F_p, \overline{E}_t)\ .$$

Using these two MC_{pa} constraints has a major drawback. The neighborhood function $\overline{V}_p(i) = \{j \mid (i,j) \in F_p\}$ and especially $\overline{V}_t(a) = \{b \mid (a,b) \notin E_t\}$ may

increase time complexity, because most of the time is spent in the loop upon $\overline{V}_t(a)$. Whatever the average degree of the target graph is, one of the constraints has a $O(ND^2)$ complexity. Moreover, a second structure $\overline{S}(i,a) = |D(x_i) \cap \overline{V}_t(a)|$ has to be created. These two constraints can however be expressed within a single propagator, thanks to $\overline{S}(i,a) = 0 \Leftrightarrow S(i,a) = |D(x_i)|$. Indeed, $S(i,a) + \overline{S}(i,a) = |D(x_i) \cap V(a)| + |D(x_i) \cap \overline{V}(a)| = |D(x_i) \cap (V(a) \cup \overline{V}(a))| = |D(x_i) \cap N_t| = |D(x_i)|$.

The two propagators implementing the two instances of the MC_{pa} constraint can be implemented in an unique propagator $MCFA$ described in Algorithm 3. Record that $S(i,a) = num$ represents the number of relations between target node a and $D(x_i)$. Since $S(i,a)$ is computed over $D(x_i)$ that may be different from the actual $D(x_i)$, a counter $size$ is added to $S(i,a)$ structure, representing the size of $D(x_i)$ over which value num is computed.

Algorithm 3: Morphism and Forbidden Edges Constraint

```
PropagateMCFA(i,a,E_p1,E_p2,E_t)
// Element a exits from D(x_i)
S(i,a,E_t).size ← S(i,a,E_t).size − 1
for b ∈ V(a,E_t) do
    S(i,b,E_t).num ← S(i,b,E_t).num − 1
    num = S(i,b,E_t).num
    size = S(i,b,E_t).size
    if size == num then
        PropaNeigh(i,b,E_p2)
    if num == 0 then
        PropaNeigh(i,b,E_p1)
Procedure PropaNeigh(i,b,E_p)
Wait until i ∈ dom(f)
for j ∈ V(i,E_p) do
    D(x_j) ← D(x_j) \ b
```

The LA constraint may also be adapted for approximate matching. Constraint LA infers propagation on its x_i variable on the basis of x_i neighborhood.

Definition 4 *The selected neighborhood function $V_p^+(i)$, with respect to a finite set variable $D = [f_{lb}, f_{gb}]$ representing $dom(f)$ of a node i in an approximate pattern graph is the set $\{\, j \mid j \in V_p(i) \wedge j \in f_{lb} \}$.*

The function $V_p^+(i)$ creates an LA_{pa}^+ constraint, playing the same role as in the exact case :

$$LA_{pa}^+(x_i, A, B) \equiv |\cup_{j \in V_p^+(i,A)} D(x_j) \cap V_t(x_i, B)| \geq |V_p^+(i, A)|$$
$$LA_{pa}^+(x_1, ..., x_n, A, B) \equiv \bigwedge_i LA_{pa}^+(x_i, A, B)\ .$$

Similarly to the LA_p constraint, LA^+_{pa} plays a pruning role. It can be implemented by maintaining the neighborhood variable, with an $O(d)$ time complexity, whenever the domain of x_i is pruned. The structure $R^+(i,a) = |\{\ j \in V^+_p(i,A) \mid a \in D(x_j)\ \}|$ depends not only on the domain of the neighborhood of x_i but also on the neighborhood variable. Whenever the lower bound of $V^+_p(i,A)$ changes, the structure $R^+(i,\cdot)$ must be updated in $O(D)$, resulting in a $O(ND^2)$ amortized complexity. Moreover, $R^+(i,a)$ may be incremented from zero to one, resulting in an increment of $CT^+(i,a) = |\cup_{j \in V^+_p(i,A)} D(x_j) \cap V_t(a,B)|$, which is not monotone. Nevertheless, when condition $CT^+(i,a) < |V^+_p(i,A)|$ is fulfilled, a can be safely pruned from x_i, because if there is not enough candidates for a subgroup of the neighborhood, node i cannot be mapped to node a, even if the condition still holds for the group.

5.4 Extending approximate pattern

Until now parametric constraints has been used for designing global or redundant constraints. In fact they can also be instantiated to constraints declaring distance constraints between specific pattern nodes. Such properties state new information on the pattern graph. For example, a constraint $PathAtMost(x_i, x_j, k)$ could state that there exists a shortest path of distance k or less between node i and j :

$$PathAtMost(x_i, x_j, k) \equiv dist(x_i, x_j) \leq k\ .$$

Similarly, a constraint $Path(x_i, x_j, k)$ could state that there exists a path of length k between node i and j :

$$Path(x_i, x_j, k) \equiv path(x_i, x_j, k)\ .$$

A matching declaring this $Path$ constraint between two pattern nodes i and j states the existence of a path of length k, in the target graph, between nodes $f(i)$ and $f(j)$. Such additional constraints enriches the approximation on the pattern graph. It is clear that parametric constraints can be instantiated to other types of constraints as long as they are properties concerning pair nodes of the pattern and that those properties can be precomputed or dynamically computed on the target graph. For example, richer path constraints could state that there exists a path of length k containing two nodes of a given type.

6 Experiments

Our CSP model for approximate subgraph matching has been implemented inside the CSP framework of Oz/Mozart (`www.mozart-oz.org`). Both parametric propagators $MC_{pa}(x_1, ..., x_n, A, B)$ and $LA_{pa}(x_1, ..., x_n, A, B)$ were implemented as well as $MCFA$. Various transformations of E_p and E_t were automated to instantiate propagators for the forbidden edges and the distance constraints. We

also included facility constraints to declare distance constraints between specific pattern nodes.

First part of the experimental tests aims at comparing the CSP approach with a dedicated algorithm for subgraph matching. The selected algorithm is an improvement of Ullmann's algorithm [14] called vflib, described into [13]. The C++ implementation provided by the authors is used. We have also reimplemented the vflib algorithm in Oz/Mozart.

Two distinct sets of graphs were selected. The first set comes from [15] and consists of 3000 directed instances divided in three classes : first one has probability $\eta = 0.01$ (noted r001 in Table 1) that an edge is present between two distinct node n and n', second one has a 0.05 probability (noted r005) and third one has a 0.1 probability (noted r01). Those graphs were used to evaluate vflib algorithm performance [13]. In our experiments, target graph size (the number of nodes in the target graph) ranges from 20 to 200, pattern graph size is 20% of the target graph size, and all solutions are searched. From a topological point of view, a N nodes graph generated with a probability of η has a mean degree of ηN and each node has a degree close to this mean degree. We call that kind of graphs *uniform* because a subgraph has a structure close to another subgraph in the same graph. The second set contains graphs having different topological structures as explained in [2]. These graphs were generated using the Stanford GraphBase [16] and are all graphs tested in [2], consisting of 406 directed instances.

Tables 1,2 and 3 show the results for the first graph database and the GraphBase graphs. The subgraph matching is a monomorphism. Total and mean time reported concern solved instances only. Following the methodology used into [2], we put a time limit on any given run. In Table 1, left column describes the number of problems solved within a time limit of 5 minutes and right column within 10 minutes, for each set of a given target graph size 60, 80, 100, and 200. All benchmarks were performed on an Intel Xeon 3 Ghz. The three algorithms (original C++ vflib, vflib in Oz, and our CSP in Oz) solve all instances for graph size 20 and 40 within time limit. On the first graph database, one can also measure the overhead of implementing vflib algorithm in Oz. The CSP approach is outperformed by the vflib algorithm for the first graph database in Table 1, but outperforms the vflib algorithm for the Stanford GraphBase set. This comes from the fact that topological properties in the first graph database set are different from GraphBase set. In uniform graphs, the probability that a variable has an empty domain thanks to conjunction of $MC(x_i, x_j)$ constraints is low. This explains that CSP performances decrease as target size increases in Table 1. The vflib algorithm is effective in this case. When measuring performance on the set of graphs which is not uniform, CSP outperforms the vflib algorithm when looking either for one solution or for all solutions.

Benefits of the unique $MCFA$ propagator instead of the conjunction of the two MC_{pa} are shown in Table 4. The subgraph isomorphism problem is solved by using forbidden edges. Left column shows a set of runs with both MC_{pa} handling the isomorphism. Right column shows the same set of runs with the

unique propagator $MCFA$ handling the isomorphism. As expected, the $MCFA$ propagator solves more problems, and mean time over solved problems decreases.

In most cases redundant path constraints do not reduce the total time as average degree increases with distance. Path constraints are useful for enhancing the expressiveness of the pattern graph. We tested influence of specifying an additional distance constraint between two nodes (left graph in Figure 3). The pattern graph has size 20. As expected, the greater the distance, more time is needed to find all solutions as the search space is higher. This experiment underlies the feasibility using additional distance constraints between nodes, viewed as expressive constraints instead of redundant constraints, in the pattern graph if the distance is not too high (≤ 3). Such an approximate pattern may be especially usefull in domains such as bioinformatics. Approximate matching has been evaluated by declaring two constraints of distance 3 shortest path on the pattern graph and 40% of its nodes as optional. The pattern graph is matched against 100 distinct instances of a target graph made of 100 nodes, searching for all solutions. Two curves are shown in the right graph in Figure 3 in a logarithmic scale. The lower one shows the matching of the pattern graph without its distance constraints and optional nodes. The upper one represents the running time of the approximate matching. A constant factor exists between the two sets of runs and a majority of the executions are below 10 seconds.

Table 1. Comparison over uniform directed graphs.

vflib C++ 5 min.						
	r001		r005		r01	
	solved	unsol	solved	unsol	solved	unsol
60	100	0	100	0	96	4
80	100	0	94	6	98	2
100	100	0	88	12	99	1
200	74	26	84	16	97	3

vflib C++ 10 min.						
	r001		r005		r01	
	solved	unsol	solved	unsol	solved	unsol
60	100	0	100	0	96	4
80	100	0	94	6	98	2
100	100	0	89	11	99	1
200	81	19	87	13	99	1

ozvflib 5 min.						
	r001		r005		r01	
	solved	unsol	solved	unsol	solved	unsol
60	100	0	85	15	76	24
80	100	0	76	24	83	17
100	100	0	56	44	88	12
200	16	84	53	47	69	31

ozvflib 10 min.						
	r001		r005		r01	
	solved	unsol	solved	unsol	solved	unsol
60	100	0	89	11	81	19
80	100	0	79	21	85	15
100	100	0	62	38	91	9
200	18	82	57	43	79	21

CSP MC 5 min.						
	r001		r005		r01	
	solved	unsol	solved	unsol	solved	unsol
60	100	0	96	4	86	14
80	100	0	84	16	91	9
100	100	0	82	18	93	7
200	33	67	77	23	51	49

CSP MC 10 min.						
	r001		r005		r01	
	solved	unsol	solved	unsol	solved	unsol
60	100	0	96	4	91	9
80	100	0	86	14	93	7
100	100	0	86	14	99	1
200	40	60	87	13	86	14

Table 2. Comparison over GraphBase directed graphs.

One solution 5 min.				
	solved	unsol	total time	mean time
vflib C++	80,5%	19,5%	8.89 min.	0.02 min.
ozvflib	78,5%	21,5%	17.67 min.	0.04 min.
CSP	87%	13%	36.64 min.	0.09 min.

All solutions 5 min.				
	solved	unsol	total time	mean time
vflib C++	63,7%	36,3%	12.01 min.	0.02 min.
ozvflib	59,8%	40,2%	11.52 min.	0.02 min.
CSP	68,7%	31,3%	31.4 min.	0.07 min.

Table 3. Comparison over GraphBase undirected graphs.

One solution 5 min.				
	solved	unsol	total time	mean time
vflib C++	64,4%	35,6%	8.14 min.	0.006 min.
ozvflib	58,2%	41,8%	8.6 min.	0.04 min.
CSP	64,4%	35,6%	18.24 min.	0.01 min.

All solutions 5 min.				
	solved	unsol	total time	mean time
vflib C++	48,3%	51,7%	9.31 min.	0.007 min.
ozvflib	39,5%	60,5%	4.43 min.	0.003 min.
CSP	57,7%	42,3%	11.39 min.	0.009 min.

Table 4. MC_{pa} and MC_{fa} versus $MCFA$

CSP MC_p and MC_{fa} 5 min.						
	r001		r005		r01	
	solved	unsol	solved	unsol	solved	unsol
100	100	0	100	0	96	4
200	79	21	62	38	10	90

CSP $MCFA$ 5 min.						
	r001		r005		r01	
	solved	unsol	solved	unsol	solved	unsol
100	100	0	100	0	99	1
200	83	17	80	20	34	66

7 Conclusion

In this paper, we proposed a CSP approach for approximate subgraph matching. The model handles both monomorphism and isomorphism problems. It also allows the specification of additional constraints between pairs of nodes (such as distance constraints), as well as the derivation of redundant constraints providing more pruning. Approximation is specified through optional nodes and forbidden edges, as well as additional constraints. The CSP model is expressed through two parametric constraints, allowing a simple and versatile modeling of various classes of matching problems. Propagators of the constraints have been described, supported by an Oz/Mozart implementation. Experimentations showed that our CSP approach for exact matching is competitive with a specialized C++ Ullman matching algorithms, and illustrated its versatility for approximate subgraph matching.

The proposed framework for declarative approximate subgraph matching open various research directions. Better heuristics could be developed when searching for an approximate matching. Our algorithm for exact matching could also be compared with other algorithms dedicated to the largest common subgraph problem. We also intend to apply our approximate matching algorithm for the analysis of biochemical networks. New approximations could be defined on the pattern graph, along with new constraints and propagators. Finally, as

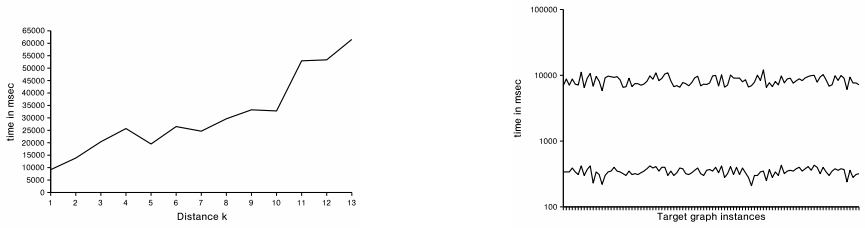


Fig. 3. Influence of distance over running time and approximate matching running times

the (approximate) matching is expressed as a combination of (parameterized) constraints, subgraph matching could be integrated in a constraint language handling graph variables, such as CP(Graph) [17].

References

1. Conte, D., Foggia, P., Sansone, C., Vento, M.: Thirty years of graph matching in pattern recognition. IJPRAI **18**(3) (2004) 265–298
2. Larrosa, J., Valiente, G.: Constraint satisfaction algorithms for graph pattern matching. Mathematical. Structures in Comp. Sci. **12**(4) (2002) 403–422
3. Rudolf, M.: Utilizing constraint satisfaction techniques for efficient graph pattern matching. In Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: TAGT. Volume 1764 of Lecture Notes in Computer Science., Springer (1998) 238–251
4. Wang, J.T.L., Zhang, K., Chirn, G.W.: Algorithms for approximate graph matching. Inf. Sci. Inf. Comput. Sci. **82**(1-2) (1995) 45–74
5. Messmer, B.T., Bunke, H.: A new algorithm for error-tolerant subgraph isomorphism detection. IEEE Trans. Pattern Anal. Mach. Intell. **20**(5) (1998) 493–504
6. DePiero, F., Krout, D.: An algorithm using length-r paths to approximate subgraph isomorphism. Pattern Recogn. Lett. **24**(1-3) (2003) 33–46
7. Robles-Kelly, A., Hancock, E.: Graph edit distance from spectral seriation. IEEE Transactions on Pattern Analysis and Machine Intelligence **27-3** (2005) 365–378
8. Giugno, R., Shasha, D.: Graphgrep: A fast and universal method for querying graphs. In: ICPR (2). (2002) 112–115
9. Sorlin, S., Solnon, C.: A global constraint for graph isomorphism problems. In Régin, J.C., Rueher, M., eds.: CPAIOR. Volume 3011 of Lecture Notes in Computer Science., Springer (2004) 287–302
10. Mamoulis, N., Stergiou, K.: Constraint satisfaction in semi-structured data graphs. In Wallace, M., ed.: CP. Volume 3258 of Lecture Notes in Computer Science., Springer (2004) 393–407
11. Regin, J.C.: A filtering algorithm for constraints of difference in CSPs. In: Proc. 12th Conf. American Assoc. Artificial Intelligence. Volume 1., Amer. Assoc. Artificial Intelligence (1994) 362–367
12. van Hoeve, W.J.: The alldifferent constraint: A survey. CoRR **cs.PL/0105015** (2001)
13. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: Performance evaluation of the vf graph matching algorithm. In: ICIAP, IEEE Computer Society (1999) 1172–1177
14. Ullmann, J.R.: An algorithm for subgraph isomorphism. J. ACM **23**(1) (1976) 31–42
15. Foggia, P., Sansone, C., Vento, M.: A database of graphs for isomorphism and sub-graph isomorphism benchmarcking. CoRR **cs.PL/0105015** (2001)
16. Knuth, D.E.: The Stanford GraphBase. A Platform for Combinatorial Computing. acm, ny (1993)
17. Dooms, G.: Cp(graph): Introducing a graph computation domain in constraint programming (accepted paper). CP2005 (2005)