Adaptive Parameterized Consistency for Non-binary CSPs by Counting Supports*

Robert J. Woodward[1,2], Anthony Schneider[1], Berthe Y. Choueiry[1], and Christian Bessiere[2]

[1] Constraint Systems Laboratory, University of Nebraska-Lincoln, USA
{rwoodwar,aschneid,choueiry}@cse.unl.edu
[2] CNRS, University of Montpellier, France
bessiere@lirmm.fr

Abstract. Determining the appropriate level of local consistency to enforce on a given instance of a Constraint Satisfaction Problem (CSP) is not an easy task. However, selecting the right level may determine our ability to solve the problem. Adaptive parameterized consistency was recently proposed for binary CSPs as a strategy to dynamically select one of two local consistencies (i.e., AC and maxRPC). In this paper, we propose a similar strategy for non-binary table constraints to select between enforcing GAC and pairwise consistency. While the former strategy approximates the supports by their rank and requires that the variables domains be ordered, our technique removes those limitations. We empirically evaluate our approach on benchmark problems to establish its advantages.

1 Introduction

There is an abundance of local consistency techniques of varying cost and pruning power to apply to a Constraint Satisfaction Problem (CSP), but choosing the right one for a given instance remains an open question. In a portfolio approach [22,11,7], we typically choose a single consistency level and enforce it on the entire problem (or a subproblem). Heuristic-based methods have been proposed to dynamically switch, at various stages of search and depending on the constraint, between a weak and a strong level of consistency, AC and maxRPC for binary CSPs [20] and GAC and maxRPWC for non-binary CSPs [18]. The above-mentioned approaches do not allow us to enforce different levels of consistency on the values in the domain of the same variable. To this end, Balafrej et al. introduced *adaptive parameterized consistency*, which selects, for each value in the domain of a variable, one of two consistency levels based on the value of a parameter [1]. That parameter is determined by the *rank* of the support of the value in a constraint (assuming a fixed total ordering of the variables' domains),

* This research was supported by NSF Grant No. RI-111795 and EU project ICON (FP7-284715). Woodward was supported by an NSF GRF Grant No. 1041000 and a Chateaubriand Fellowship. Experiments were conducted on the equipment of the Holland Computing Center at the University of Nebraska–Lincoln.

B. O'Sullivan (Ed.): CP 2014, LNCS 8656, pp. 755–764, 2014.

and updated depending on the weight of the constraint [5]. Their study targeted enforcing AC and maxRPC on binary CSPs.

In this paper, we extend their mechanism to enforcing GAC and pairwise-consistency on non-binary CSPs with table constraints. Our approach is based on *counting* the number of supporting tuples, which is automatically provided by the algorithms that we use. Thus, we remove the restriction on maintaining ordered domains and the approximation of a support's count by its rank. We establish empirically the advantages of our approach.

The paper is structured as follows. Section 2 provides background information. Section 3 describes our approach, and Section 4 discusses our empirically evaluation on benchmark problems. Finally, Section 5 concludes the paper.

2 Background

We first summarize the main concepts and definitions used.

2.1 Constraint Satisfaction Problem

A *Constraint Satisfaction Problem* (CSP) is defined by a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X}$ is a set of variables, $\mathcal{D}$ is a set of domains, and $\mathcal{C}$ is a set of constraints. Each variable $x_i \in \mathcal{X}$ is associated a finite domain $dom(x_i) \in \mathcal{D}$. We denote a variable-value pair as $\langle x_i, v_i \rangle$, where $v_i \in dom(x_i)$. Each constraint $c_j \in \mathcal{C}$ is defined in extension by a relation R_j specified over the scope of the constraint, $scope(c_j)$, which is the set of variables to which the constraint applies. For readability, we refer to the scope of a relation $scope(R_j) = scope(c_j)$. A tuple $\tau \in R_j$ is a combination of allowed values for the variables in $scope(R_j)$. $\tau[x_i]$ is the value that the variable x_i takes in τ. We denote $cons(x_i)$ as the set of constraints that apply to variable x_i, and $neigh(c_j)$ the set of constraints whose scopes overlap with c_j. When $|scope(c_j)| = 2$, c_j is said to be a binary constraint, otherwise, it is non-binary. A solution to the CSP assigns, to each variable, a value taken from its domain such that all the constraints are satisfied. Deciding the existence of a solution for a CSP is NP-complete.

2.2 Local Consistency Properties

CSPs are typically solved with backtrack search. To reduce the severity of the combinatorial explosion, CSPs are usually filtered by enforcing a given *local consistency property* [2].

A variable-value pair $\langle x_i, v_i \rangle$ has an arc-consistent support (AC-support) $\langle x_j, v_j \rangle$ if the tuple $(v_i, v_j) \in R_{ij}$ where $scope(R_{ij}) = \{x_i, x_j\}$ [16,3]. A CSP is arc consistent if every variable-value pair has an AC-support in every constraint. Generalized Arc Consistency (GAC) generalizes arc consistency to non-binary CSPs [16]. $\langle x_i, v_i \rangle$ has a GAC-support in constraint c_j if $\exists \tau \in R_j$ such that $\tau[x_i] = v_i$. A CSP is GAC if every $\langle x_i, v_i \rangle$ has a GAC-support in every constraint in $cons(x_i)$. GAC can be enforced by removing domain values that have

no GAC-support, leaving the relations unchanged. Simple Tabular Reduction (STR) algorithms not only enforce GAC on the domains, but also remove all tuples $\tau \in R_j$ where $\exists x_i \in scope(R_j)$ such that $\tau[x_i] \notin dom(x_i)$ [21,13,14].

A CSP is m-wise consistent if, every tuple in a relation can be extended to every combination of $m-1$ other relations in a consistent manner [8,10]. Keeping with relational-consistency notations, Karakashian et al. denoted m-wise consistency by R($*$,m)C, and proposed a first algorithm for enforcing it [12]. Their implementation finds an extension (i.e., support) for a tuple by conducting a backtrack search on the other $m-1$ relations, and removes the tuples that have no support. After all relations are filtered, they are projected onto the domains of the variables. Pairwise consistency (PWC) corresponds to m=2, R($*$,2)C≡PWC. Lecoutre et al. introduced the algorithm extended STR (eSTR) [15], which enforces PWC on a CSP using the STR mechanism [21]. eSTR maintains counters on the intersections of two constraints to determine if a tuple is pairwise consistent or not. In this paper, we enforce PWC using the algorithm for R($*$,2)C [12], and not eSTR, because it is prohibitively expensive to continuously maintain the counters of eSTR in a strategy where PWC is only selectively enforced.

2.3 Adaptive Parameterized Consistency

Balafrej et al. introduced *the distance to the end of value v_i for variable x_i* as:

$$\Delta(x_i, v_i) = \frac{|dom^o(x_i)| - rank(v_i, dom^o(x_i))}{|dom^o(x_i)|}$$

where $dom^o(x_i)$ is the original, unfiltered domain of x_i, and $rank(v_i, dom^o(x_i))$ is the position of v_i in the ordered set $dom^o(x_i)$ [1]. In Figure 1, borrowed from [1], $\Delta(x_2, 1) = 0.75$, $\Delta(x_2, 2) = 0.50$, $\Delta(x_2, 3) = 0.25$, and $\Delta(x_2, 4) = 0.00$.

Further, for a given parameter p, they defined $\langle x_i, v_i \rangle$ to be *p-stable for AC* for c_{ij} where $scope(c_{ij}) = \{x_i, x_j\}$ if there exists an AC-support $\langle x_j, v_j \rangle$ with $\Delta(x_j, v_j) \geq p$ for c_{ij}. Figure 1 illustrates an example for the constraint $x_1 \leq x_2$ with $p = 0.25$. $\langle x_1, 1 \rangle$,$\langle x_1, 2 \rangle$,$\langle x_1, 3 \rangle$ are all 0.25-stable for AC for the constraint, but $\langle x_1, 4 \rangle$ is not, because its only AC-support, $\langle x_2, 4 \rangle$, has distance 0.

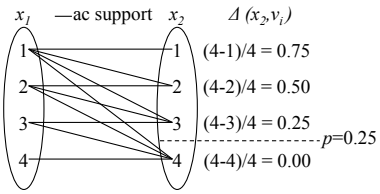

Fig. 1. The constraint $x_1 \leq x_2$. $\langle x_1, 4 \rangle$ is not 0.25-stable for AC [1]

The *parameterized* strategy p-LC [1] enforces, on each variable-value pair, either AC or some local consistency (LC) property strictly stronger than AC

depending on the value of the parameter p. The idea is to enforce LC only on the variable-value pairs with few supports, approximated with the rank ($< p$) of the first found AC-support. We focus on the constraint-based version, *pc*-LC, where $\langle x_i, v_i \rangle$ is *pc*-LC if for *every* constraint $c_j \in cons(x_i)$, $\langle x_i, v_i \rangle$ is p-stable for AC on c_j or $\langle x_i, v_i \rangle$ is LC on c_j. In *pc*-LC, the value of p is given as input. In the *adaptive* version, *apc*-LC, it is dynamically determined for each constraint c_j using the *weight* of c_j, $w(c_j)$, which is the number of times c_j caused a domain wipe-out like in the variable-ordering heuristic dom/wdeg [5]:

$$p(c_j) = \frac{w(c_j) - \min_{c_k \in \mathcal{C}}(w(c_k))}{\max_{c_k \in \mathcal{C}}(w(c_k)) - \min_{c_k \in \mathcal{C}}(w(c_k)) + 1}. \quad (1)$$

In [1], *apc*-maxRPC was experimentally shown to outperform AC and maxRPC [6].

3 Modifying *apc*-LC for Non-binary CSPs

For binary CSPs, p-stability for AC of $\langle x_i, v_i \rangle$ estimates how many supports are left for $\langle x_i, v_i \rangle$ in other constraints using the rank of the AC-support in the corresponding domain. This estimate should not directly applied to non-binary table constraints because the GAC-support of $\langle x_i, v_i \rangle$ is a tuple in a relation that is unsorted, which would make the estimate way too imprecise. Consider the example with $\langle x_i, v_i \rangle$ and a relation R_j of 100 tuples. Assume that the only tuple $\tau \in R_j$ supporting $\langle x_i, v_i \rangle$ appears at the top of the table of R_j. The estimate would indicate that there are many supports for $\langle x_i, v_i \rangle$ because there are 99 tuples that appear after it. However, in reality, $\langle x_i, v_i \rangle$ has a unique support. Below, we introduce p-stability for GAC, which counts the number of supports for each variable-value pair. Then, we introduce a mechanism to compute p-stability for GAC, and finally give an algorithm for enforcing *apc*-LC, which adaptively enforces STR or LC. In this paper, we study R(∗,2)C as LC, and discuss the implementation of *apc*-R(∗,2)C.

3.1 *p*-Stability for GAC

We say that $\langle x_i, v_i \rangle$ is *p-stable for GAC* if for every constraint $c_j \in cons(x_i)$,

$$\frac{|\sigma_{x_i=v_i}(R_j)|}{|R_j^o|} \geq p(c_j),$$

where $\sigma_{x_i=v_i}(R_j)$ selects the tuples in R_j where $\langle x_i, v_i \rangle$ appears, and R_j^o is the original, unfiltered relation. A CSP is p-stable for GAC if every variable-value pair is p-stable for GAC for every constraint that applies to it.

Figure 2 gives the relation for the constraint $x_1 \leq x_2$. $\langle x_1, 1 \rangle$ and $\langle x_1, 2 \rangle$ are 0.25-stable for GAC. Indeed, $\sigma_{x_1=1}$ returns four rows $\{0, 1, 2, 3\}$ in the table, and $\langle x_1, 1 \rangle$ is 0.25-stable: $\frac{4}{10} \geq 0.25$. Similarly, $\langle x_1, 2 \rangle$ also is 0.25-stable: $\frac{3}{10} \geq 0.25$. $\langle x_1, 3 \rangle$ and $\langle x_1, 4 \rangle$ are not 0.25-stable, because $\frac{2}{10} \not\geq 0.25$ and $\frac{1}{10} \not\geq 0.25$. This example illustrates how, on binary constraints, and for a given p, p-stable for AC does not guarantee p-stable for GAC. (Recall that $\langle x_1, 3 \rangle$ is 0.25-stable for AC in Figure 1).

	x_1	x_2	
0	1	1	$gacSupports[R_j](\langle x_1,1\rangle)=\{0,1,2,3\}$
1	1	2	$gacSupports[R_j](\langle x_1,2\rangle)=\{4,5,6\}$
2	1	3	$gacSupports[R_j](\langle x_1,3\rangle)=\{7,8\}$
3	1	4	$gacSupports[R_j](\langle x_1,4\rangle)=\{9\}$
4	2	2	
5	2	3	
6	2	4	$gacSupports[R_j](\langle x_2,1\rangle)=\{0\}$
7	3	3	$gacSupports[R_j](\langle x_2,2\rangle)=\{1,4\}$
8	3	4	$gacSupports[R_j](\langle x_2,3\rangle)=\{2,5,7\}$
9	4	4	$gacSupports[R_j](\langle x_2,4\rangle)=\{3,6,8,9\}$

Fig. 2. The relation of $x_1 \leq x_2$. $\langle x_1, 3\rangle$ and $\langle x_1, 4\rangle$ are not 0.25-stable for GAC.

3.2 Computing p-Stability for GAC

For each constraint c_j, we introduce for every $\langle x_i, v_i\rangle$ a set of integers indicating the position of the tuples returned by $\sigma_{x_i=v_i}(R_j)$, which is similar to the data structure in GAC4 [17]. We denote this table $gacSupports[R_j][\langle x_i, v_i\rangle]$. The check for p-stable can be verified by using $|gacSupports[R_j][\langle x_i, v_i\rangle]|$. Figure 2, shows the $gacSupports[R_j]$ for the constraint $x_1 \leq x_2$. For each relation, the space complexity to store each $gacSupports[R_j]$ is $\mathcal{O}(k \cdot t)$, where k is the maximum constraint arity and t is the maximum number of tuples in a relation. The time complexity to generate $gacSupports[R_j]$ is $\mathcal{O}(k \cdot t)$, by iterating through every tuple.

3.3 Algorithm for Enforcing *apc*-LC

With the *gacSupports* data-structure, we can apply STR by verifying, for each constraint c_j, that every variable $x_i \in scope(c_j)$ and $v_i \in dom(x_i)$ has a non-zero $|gacSupports[R_j][\langle x_i, v_i\rangle]|$. Living-STR (Algorithm 1) does precisely this operation (ignoring Lines 1 and 1, which apply to the *apc*-LC operation introduced next). $past(\mathcal{P})$ denotes the variables of the CSP $\mathcal{P}$ already instantiated by search, and delTuples$(R_k, S, level)$ deletes all the tuples in the subset $S \subseteq R_k$, and marks their removal level at the level of search $level$. When deleting a tuple from the relation R_k, c_k's neighboring constraints, $neigh(c_k)$, should be re-queued to be processed with Living-STR. Initially, all constraints are in the queue. Living-STR is similar to STR3 in that it iterates over variable-value pairs rather than over tuples. However, it does not use as much book-keeping for optimizing the number of STR checks as STR3 [14]. Instead, Living-STR uses the same data structures as STR and STR2(+) to manage tuple deletions in a relation [13,21].

Including Lines 1 and 1 in Algorithm 1 yields *apc*-LC, which adaptively applies LC. The adaptive level $p(c_j)$ is defined by Balafrej et al. [1] and recalled in Equation (1). The local consistency technique used here is the implementation of R(∗,2)C [12], *apc*-R(∗,2)C. Apply-R(∗,2)C (Algorithm 2) takes as input the list of tuples of a constraint on which R(∗,2)C must be enforced.

Algorithm 1. LIVING-STR(c_i): set of variables

Input: c_j: a constraint of $\mathcal{P}$
Output: Set of variables in $scope(c_j)$ whose domains have been modified

1 $X_{modified} \leftarrow \emptyset$
2 **foreach** $x_i \in scope(c_j) \mid x_i \notin past(\mathcal{P})$ **do**
3 **foreach** $v_i \in dom(x_i)$ **do**
4 **if** $|gacSupports[R_j](\langle x_i, v_i \rangle)| \neq 0$ ***and*** $\frac{|gacSupports[R_j](\langle x_i, v_i \rangle)|}{|R_j^o|} \not\geq p(c_j)$ **then**
5 APPLY-LC($R_j, gacSupports[R_j](\langle x_i, v_i \rangle)$)
6 **if** $|gacSupports[R_j](\langle x_i, v_i \rangle)| = 0$ **then**
7 **foreach** $c_k \in cons(x_i)$ **do**
8 delTuples($c_k, gacSupports[R_k](\langle x_i, v_i \rangle), |past(\mathcal{P})|$)
9 $dom(x_i) \leftarrow dom(x_i) \setminus \{v_i\}$
10 **if** $dom(x_i) = \emptyset$ **then throw** INCONSISTENCY
11 $X_{modified} \leftarrow X_{modified} \cup \{x_i\}$
12 **return** $X_{modified}$

Algorithm 2. APPLY-R(∗,2)C(c_i, *tuples*)

Input: c_i: a constraint; *tuples*: a set of tuples from the constraint c_i
Output: The *tuples* are either R(∗,2)C or deleted

1 **foreach** $\tau \in tuples$ **do**
2 **foreach** $c_j \in neigh(c_i)$ **do**
3 **if** SEARCHSUPPORT($R_i, \tau, \{R_j\}$) *returns inconsistent* **then**
4 delTuples($c_i, \{\tau\}, |past(\mathcal{P})|$)

SEARCHSUPPORT($R_i, \tau, \{R_j\}$) on Line 2 of Algorithm 2 searches for a support for the tuple $\tau \in R_i$, the pairwise check [12].

Theoretical analysis: Let k be the maximum constraint arity, d the maximum domain size, and δ the maximum number of neighbors of a constraint. The time complexity of Algorithm 1 is $\mathcal{O}(k \cdot d)$. Algorithm 2 is $\mathcal{O}(\delta \cdot t^2)$ because it makes $\mathcal{O}(\delta \cdot t)$ calls to SEARCHSUPPORT, which is $\mathcal{O}(t)$ in our context. The correctness of Algorithms 1 and 2 can be shown in straightforward manner by contradiction.

4 Empirical Evaluations

The goal of our experimental analysis is to assess if *apc*-R(∗,2)C effectively selects when to apply STR and R(∗,2)C when used in a pre-processing step and in a real full lookahead strategy [9] during backtrack search to find the first solution to a CSP. In our experiments, we use the variable ordering dom/wdeg [5]. The experiments are conducted on the benchmarks of the CSP Solver Competition[1]

[1] `http://www.cril.univ-artois.fr/CPAI08/`

with a time limit of two hours per instance and 8 GB of memory. Because STR and R(∗,2)C enforce the same level of consistency on binary CSPs [4], we focus our experiments on 21 non-binary benchmarks[2] consisting of 623 CSP instances. We chose these benchmarks because they are given in extension and at least one algorithm completed 5% of the instances in the benchmark.

Table 1 summarizes the results in terms of number of instances solved. Importantly, *apc*-R(∗,2)C completes the largest number of instances (552). Considering the instances solved by all algorithms (485 instances), *apc*-R(∗,2)C has the smallest average and median CPU time. Row 3 indicates the number of instances STR solved but R(∗,2)C and *apc*-R(∗,2)C did not solve (18 and 11 instances, respectively), thus showing that *apc*-R(∗,2)C, although it may have enforced R(∗,2)C too often, outperformed R(∗,2)C and missed fewer instances than it (11 vs. 18). Row 4 exhibits similar results showing the number of instances that R(∗,2)C could solve, but that were missed by STR and *apc*-R(∗,2)C (64 and 6 instances, respectively). Here, *apc*-R(∗,2)C did not enforce R(∗,2)C often enough, but managed to outperform STR missing significantly fewer instances than STR (6 vs. 64).

Table 1. Number of instances completed by the tested algorithms

		STR	R(∗,2)C	*apc*-R(∗,2)C
1	#instances completed by	504	550	**552**
2	#instances completed only by	10	5	0
3	#instances solved by STR, but missed by	0	18	11
4	#instances solved by R(∗,2)C, but missed by	64	0	6
5	#instances solved by *apc*-R(∗,2)C, but missed by	59	8	0
	Average CPU time (sec.) over 458 instances	328.41	378.12	**313.31**
	Median CPU time (sec.) over 458 instances	7.23	17.35	**7.21**

Table 2 gives a finer analysis of the data, showing the number of completions and average and median CPU time per benchmark. Averages computed over only the instances completed by all techniques are shown in the column *All*. We split the table into four categories based on the *average* CPU time of *apc*-R(∗,2)C: *a*) *apc*-R(∗,2)C performs the best (5 benchmarks); *b*) *apc*-R(∗,2)C is competitive, performing between STR and R(∗,2)C (13 benchmarks); *c*) *apc*-R(∗,2)C performs the worst (2 benchmarks); and *d*) STR does not solve the benchmark but R(∗,2)C and *apc*-R(∗,2)C do (1 benchmark). The best average CPU time appears in bold face in the corresponding column. The median CPU time of *apc*-R(∗,2)C is bold faced when its rank differs from that of the average CPU time (on which the four categorized are based). On TSP-20, *apc*-R(∗,2)C ranks bottom on average CPU time but between STR and R(∗,2)C on median CPU time. On aim-100, jnhUnsat, rand-8-20-5, and ukVg, *apc*-R(∗,2)C is between STR and R(∗,2)C for average CPU time, but best for median CPU time.

[2] Aim-(50,100,200), allIntervalSeries, dag-rand, dubois, jnh(Sat/Unsat), lexVg, modifiedRenault, pret, rand-10-20-10, rand-3-20-20(-fcd), rand-8-20-5, ssa, travellingSalesman-20, travellingSalesman-25, ukVg, varDimacs, wordsVg.

Table 2. Results of the experiments per benchmark, organized in four categories

Benchmark	#Instances	#Completed STR	#Completed R(∗,2)C	#Completed *apc*-R(∗,2)C	#Completed All	Average CPU time (sec) STR	Average CPU time (sec) R(∗,2)C	Average CPU time (sec) *apc*-R(∗,2)C	Median CPU time (sec) STR	Median CPU time (sec) R(∗,2)C	Median CPU time (sec) *apc*-R(∗,2)C
a) *apc*-R(∗,2)C is the best											
aim-50	24	24	24	24	24	**0.04**	0.07	**0.04**	0.02	0.04	**0.03**
allIntervalSeries	25	22	22	22	22	7.09	141.85	**6.00**	0.13	0.31	0.12
jnhSat	16	16	16	16	16	13.07	357.66	**11.74**	8.15	142.24	7.21
modifiedRenault	50	50	50	50	50	6.39	11.17	**6.29**	7.24	8.79	6.98
rand-3-20-20	50	31	43	41	31	1,666.10	939.88	**932.77**	1,211.50	822.54	811.74
b) *apc*-R(∗,2)C is competitive											
aim-100	24	24	24	24	24	0.38	**0.26**	0.41	0.18	0.25	**0.16**
aim-200	24	22	24	24	22	414.48	**6.52**	286.27	2.39	1.37	**2.60**
jnhUnsat	34	34	34	34	34	**13.61**	294.77	13.95	10.74	153.50	**9.78**
lexVg	63	63	63	63	63	**69.81**	341.87	338.74	0.50	1.38	0.89
pret	8	4	4	4	4	**117.89**	347.03	136.04	115.81	354.82	145.70
rand-3-20-20-fcd	50	39	48	47	39	928.06	**546.84**	615.23	501.30	422.24	464.00
rand-8-20-5	20	9	20	20	9	2,564.94	**355.57**	372.76	1,987.35	314.26	**261.68**
rand-10-20-10	20	12	12	12	12	6.72	**1.67**	2.76	6.40	1.66	2.75
ssa	8	6	5	6	5	**64.60**	100.64	69.59	1.51	1.60	1.58
TSP-25	15	13	10	13	10	**232.38**	1,072.72	743.33	69.00	211.41	131.69
ukVg	65	37	31	34	31	**166.82**	796.90	421.35	36.29	54.65	**30.39**
varDimacs	9	6	6	6	6	**89.23**	587.55	319.20	1.56	6.43	2.94
wordsVg	65	65	58	58	58	**119.76**	532.05	400.22	0.39	0.95	0.59
c) *apc*-R(∗,2)C is the worst											
dubois	13	7	8	6	6	1,000.54	**451.91**	1,456.01	552.13	255.25	779.57
TSP-20	15	15	15	15	15	**101.20**	318.37	335.13	23.32	61.55	**46.34**
d) Not solved by STR											
dag-rand	25	0	25	25	0	-	**123.70**	149.64	-	124.47	151.33

Table 3 shows the average number of STR and R(∗,2)C checks that *apc*-R(∗,2)C performs per benchmark. In allIntervalSeries, *no* calls are made to R(∗,2)C because the instance is solved backtrack free with STR alone. For *apc*-LC, *no call to LC is done during pre-processing* because the weights of all the

Table 3. Number of calls to STR and R(∗,2)C by benchmark

Benchmark	STR checks	R(∗,2)C checks	Benchmark	STR checks	R(∗,2)C checks
a) *apc*-R(∗,2)C is the best			*b*) *apc*-R(∗,2)C is competitive		
aim-50	456,823	39,491	aim-100	7,731,585	894,353
allIntervalSeries	38,281,694	0	aim-200	1,160,334,482	163,177,907
jnhSat	22,119,135	599,080	jnhUnsat	51,688,166	1,918,781
modifiedRenault	4,618,778	601,641	lexVg	564,010,457	2,180,503,026
rand-3-20-20	489,441,126	3,480,216,943	pret	422,987,946	13,973,748
			rand-3-20-20-fcd	455,664,100	2,956,467,994
c) *apc*-R(∗,2)C is the worst			rand-8-20-5	77,470,561	184,764,543
dubois	3,343,830,604	4,668,288	rand-10-20-10	72,608	3,972
TSP-20	622,949,698	991,590,957	ssa	156,631,370	11,689,961
			TSP-25	2,903,953,315	3,947,391,769
			ukVg	341,565,892	1,002,334,753
d) Not solved by STR			varDimacs	720,843,958	84,123,204
dag-rand	359,248	21,870	wordsVg	514,840,737	2,052,367,934

constraints are set to 1 (giving $p(c_j) = 0$ for all $c_j \in \mathcal{C}$) and updated only during search. For dag-rand, there is a smaller number of R(∗,2)C calls than STR calls (21,870 vs. 359,248). However, those few calls allow us to solve *all* the instances of this benchmark whereas STR alone could not solve *any* instance. This result is a glowing testimony of the ability of *apc*-R(∗,2)C to apply the appropriate level of consistency where needed.

5 Conclusions

In this paper, we extend the notion of p-stability for AC to GAC, and provide a mechanism for computing it. We give an algorithm for enforcing *apc*-R(∗,2)C on non-binary table constraints, which adaptively enforces GAC and R(∗,2)C. We validate our approach on benchmark problems. Future work is to investigate other adaptive criteria for selecting the level of consistency to apply, in particular one that operates during both pre-processing and search. To apply our approach to constraints defined in intension and other global constraints, we could use techniques that approximate the number of solutions in those constraints [19].

References

1. Balafrej, A., Bessiere, C., Coletta, R., Bouyakhf, E.H.: Adaptive Parameterized Consistency. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 143–158. Springer, Heidelberg (2013)
2. Bessiere, C.: Constraint Propagation. In: Handbook of Constraint Programming, pp. 29–83. Elsevier (2006)
3. Bessière, C., Régin, J.C., Yap, R.H., Zhang, Y.: An Optimal Coarse-Grained Arc Consistency Algorithm. Artificial Intelligence 165(2), 165–185 (2005)
4. Bessière, C., Stergiou, K., Walsh, T.: Domain Filtering Consistencies for Non-Binary Constraints. Artificial Intelligence 172, 800–822 (2008)
5. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting Systematic Search by Weighting Constraints. In: Proc. ECAI 2004, pp. 146–150 (2004)
6. Debruyne, R., Bessière, C.: From Restricted Path Consistency to Max-Restricted Path Consistency. In: Smolka, G. (ed.) CP 1997. LNCS, vol. 1330, pp. 312–326. Springer, Heidelberg (1997)
7. Geschwender, D., Karakashian, S., Woodward, R., Choueiry, B.Y., Scott, S.D.: Selecting the Appropriate Consistency Algorithm for CSPs Using Machine Learning Techniques. In: Proc. of AAAI 2013, pp. 1611–1612 (2013)
8. Gyssens, M.: On the Complexity of Join Dependencies. ACM Trans. Database Systems 11(1), 81–108 (1986)
9. Haralick, R.M., Elliott, G.L.: Increasing Tree Search Efficiency for Constraint Satisfaction Problems. Artificial Intelligence 14, 263–313 (1980)
10. Janssen, P., Jégou, P., Nougier, B., Vilarem, M.C.: A Filtering Process for General Constraint-Satisfaction Problems: Achieving Pairwise-Consistency Using an Associated Binary Representation. In: IEEE Workshop on Tools for AI, pp. 420–427 (1989)
11. Kadioglu, S., Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Algorithm Selection and Scheduling. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 454–469. Springer, Heidelberg (2011)

12. Karakashian, S., Woodward, R., Reeson, C., Choueiry, B.Y., Bessiere, C.: A First Practical Algorithm for High Levels of Relational Consistency. In: Proc. AAAI 2010, pp. 101–107 (2010)
13. Lecoutre, C.: STR2: Optimized Simple Tabular Reduction for Table Constraints. Constraints 16(4), 341–371 (2011)
14. Lecoutre, C., Likitvivatanavong, C., Yap, R.H.C.: A Path-Optimal GAC Algorithm for Table Constraints. In: Proc. of ECAI 2012, pp. 510–515 (2012)
15. Lecoutre, C., Paparrizou, A., Stergiou, K.: Extending STR to a Higher-Order Consistency. In: Proc. AAAI 2013, Bellevue, WA, pp. 576–582 (2013)
16. Mackworth, A.K.: Consistency in Networks of Relations. AI 8, 99–118 (1977)
17. Mohr, R., Masini, G.: Good Old Discrete Relaxation. In: European Conference on Artificial Intelligence (ECAI 1988), pp. 651–656. W. Germany, Munich (1988)
18. Paparrizou, A., Stergiou, K.: Evaluating Simple Fully Automated Heuristics for Adaptive Constraint Propagation. In: Proc. of ICTAI 2012, pp. 880–885 (2012)
19. Pesant, G., Quimper, C.G., Zanarini, A.: Counting-Based Search: Branching Heuristics for Constraint Satisfaction Problems. JAIR 43, 173–210 (2012)
20. Stergiou, K.: Heuristics for Dynamically Adapting Propagation. In: Proc. of ECAI 2008, pp. 485–489 (2008)
21. Ullmann, J.R.: Partition Search for Non-binary Constraint Satisfaction. Information Sciences 177(18), 3639–3678 (2007)
22. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-Based Algorithm Selection for SAT. JAIR 32, 565–606 (2008)