Probabilistic Consistency Boosts MAC and SAC

Deepak Mehta* **and M.R.C. van Dongen**
Computer Science Department, University College Cork, Ireland

Abstract

Constraint Satisfaction Problems (CSPs) are ubiquitous in Artificial Intelligence. The backtrack algorithms that maintain some local consistency during search have become the de facto standard to solve CSPs. Maintaining higher levels of consistency, generally, reduces the search effort. However, due to ineffective constraint propagation, it often penalises the search algorithm in terms of time. If we can reduce ineffective constraint propagation, then the effectiveness of a search algorithm can be enhanced significantly. In order to do so, we use a probabilistic approach to resolve when to propagate and when not to. The idea is to perform only the useful consistency checking by not seeking a support when there is a high probability that a support exists. The idea of probabilistic support inference is general and can be applied to any kind of local consistency algorithm. However, we shall study its impact with respect to arc consistency and singleton arc consistency (SAC). Experimental results demonstrate that enforcing probabilistic SAC almost always enforces SAC, but it requires significantly less time than SAC. Likewise, maintaining probabilistic arc consistency and maintaining probabilistic SAC require significantly less time than maintaining arc consistency and maintaining SAC.

1 Introduction

Constraint Satisfaction Problems (CSPs) are ubiquitous in Artificial Intelligence. They involve finding values for problem variables subject to constraints. For simplicity, we restrict our attention to binary CSPs. Backtrack algorithms that maintain some local consistency during search have become the de facto standard to solve CSPs. Maintaining a higher level of local consistency before and/or during search usually reduces the thrashing behaviour of a backtrack algorithm. However, the amount of ineffective constraint propagation also increases, which can penalise the algorithm in terms of time.

Arc consistency (AC) is the most widely used local consistency technique to reduce the search space of CSPs. Coarse-grained arc consistency algorithms such as AC-3 [Mackworth, 1977] and AC-2001 [Bessière and Régin, 2001] are competitive. These algorithms repeatedly carry out revisions, which require support checks for identifying and deleting all unsupported values from the domain of a variable. However, for difficult problems, in many revisions, some or *all* values successfully find some support, that is to say, ineffective constraint propagation occurs. For example, when RLFAP 11 is solved using either MAC-3 or MAC-2001 equipped with *dom/deg* as a variable ordering heuristic, 83% of the total revisions are ineffective. If we can avoid this ineffective propagation, then a considerable amount of work can be saved.

Recently, singleton arc consistency (SAC) [Debruyne and Bessière, 1997] has been getting much attention. It is a stronger consistency than AC. Therefore, it can avoid much unfruitful exploration in the search-tree. However, as pointed out earlier, as the strength of local consistency increases, so does the amount of ineffective propagation. Hence, applying it before search can be expensive in terms of checks and time, and maintaining it during search can be even more expensive.

At the CPAI'2005 workshop, [Mehta and van Dongen, 2005a] presented a *probabilistic approach* to reduce ineffective propagation and studied it with respect to arc consistency on random problems. This probabilistic approach is to *avoid the process of seeking a support, when the probability of its existence is above some, carefully chosen, threshold*. This way a significant amount of work in terms of support checks and time can be saved.

In this paper, we present a more extensive investigation. We study the impact of using the probabilistic approach not only in AC but also in SAC and LSAC (limited version of SAC proposed in this paper) on a variety of problems. We call the resulting algorithms Probabilistic Arc Consistency [1] (PAC), Probabilistic Singleton Arc Consistency (PSAC), and Probabilistic LSAC (PLSAC). Our four main contributions, then, are as follows: First, we investigate the threshold at which Maintaining PAC (MPAC) performs the best on different classes of random problems. Second, we carry out experiments to determine how well MPAC does on a variety of known problems including real-world problems. Our findings suggest that MPAC usually requires significantly less time than MAC. Next, we

*The first author is supported by the Boole Centre for Research in Informatics.

[1] Probabilistic Arc Consistency proposed in this paper has no relation with the one mentioned in [Horsch and Havens, 2000]

examine the performances of PSAC and PLSAC when used as a preprocessor before search. Experimental results demonstrate that enforcing probabilistic SAC and LSAC almost always enforces SAC and LSAC but usually require significantly less time. Finally, we investigate the impact of maintaining PSAC and PLSAC during search on various problems. Results show a significant gain in terms of time on quasi-group problems. Overall, empirical results presented in this paper demonstrate that the original algorithms are outperformed by their probabilistic counterparts.

The remainder of this paper is organised as follows: Section 2 gives an introduction to constraint satisfaction and consistency algorithms. Section 3 introduces the notions of a probabilistic support condition and probabilistic revision condition. Experimental results are presented in section 4 followed by conclusions in section 5.

2 Preliminaries

A *Constraint Satisfaction Problem* $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ is defined as a set $\mathcal{V}$ of n variables, a non-empty domain $D(x) \in \mathcal{D}$, for all $x \in \mathcal{V}$ and a set $\mathcal{C}$ of e constraints among subsets of variables of $\mathcal{V}$. The *density* of a CSP is defined as $2\,e/(n^2 - n)$. 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. The *tightness* of the constraint C_{xy} is defined as $1 - |\,C_{xy}\,|/|\,D(x) \times D(y)\,|$. A value $b \in D(y)$ (also denoted as, (y, b)) is called a *support* for (x, a) if $(a, b) \in C_{xy}$. Similarly, (x, a) is called a support for (y, b) if $(a, b) \in C_{xy}$. A *support check* is a test to find if two values support each other. The *directed constraint graph* of a CSP is a 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.

A value $a \in D(x)$ is *arc consistent* if $\forall y \in \mathcal{V}$ constraining x the value a is supported by some value in $D(y)$. A CSP is *arc consistent* if and only if $\forall x \in \mathcal{V}$, $D(x) \neq \emptyset$, and $\forall a \in D(x)$, (x, a) is arc consistent. We denote the CSP $\mathcal{P}$ obtained after enforcing arc consistency as $ac(\mathcal{P})$. If there is a variable with an empty domain in $ac(\mathcal{P})$, we denote it as $ac(\mathcal{P}) = \bot$. Usually, an input CSP is transformed into its arc consistent equivalent, before starting search. We call the domain of x in this initial arc consistent equivalent of the input CSP the *first arc consistent domain* of x. We use $D_{ac}(x)$ for the first arc consistent domain of x, and $D(x)$ for the current domain of x. The CSP obtained from $\mathcal{P}$ by assigning a value a to the variable x is denoted by $\mathcal{P}|_{x=a}$. The value $a \in D(x)$ is singleton arc consistent if and only if $ac(\mathcal{P}|_{x=a}) \neq \bot$. A CSP is singleton arc consistent if and only if each value of each variable is singleton arc consistent.

MAC is a backtrack algorithm that maintains arc consistency after every variable assignment. Forward Checking (FC) can be considered a degenerated form of MAC. 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. MSAC is a backtrack algorithm that maintains singleton arc consistency. Throughout this paper, familiarity with the arc consistency algorithm AC-3 [Mackworth, 1977] is assumed.

3 Probabilistic Support Inference

The traditional approach to infer the existence of a support for a value $a \in D(x)$ in $D(y)$ is to *identify* some $b \in D(y)$ that supports a. This usually results in a sequence of support checks. Identifying the support is more than needed: knowing that a support exists is enough. The notions of a *support condition* (SC) and a *revision condition* (RC) were introduced in [Mehta and van Dongen, 2005b] to reduce the task of identifying a support up to some extent for arc consistency algorithms. If SC holds for (x, a) with respect to y, then it guarantees that (x, a) has *some* support in $D(y)$. If RC holds for an arc, (x, y), then it guarantees that *all* values in $D(x)$ have *some* support in $D(y)$. In the following paragraphs, we describe the special versions of SC and RC which facilitates the introduction of their probabilistic versions.

Let C_{xy} be the constraint between x and y, let $a \in D(x)$, and let $R(y) = D_{ac}(y) \setminus D(y)$ be the values removed from the first arc consistent domain of y. The *support count* of (x, a) with respect to y, denoted $sc(\,x, y, a\,)$, is the number of values in $D_{ac}(y)$ supporting a. Note that $|\,R(y)\,|$ is an upper bound on the number of lost supports of (x, a) in y. If $sc(x, y, a) > |\,R(y)\,|$ then (x, a) is supported by y. This condition is called a special version of a support condition. For example, if $|D_{ac}(y)| = 20$, $sc(x, y, a) = 2$, and $|\,R(y)\,| = 1$, i.e. 1 value is removed from $D_{ac}(y)$, then SC holds and (x, a) has a support in $D(y)$ with a probability of 1. Hence, there is no need to seek support for a in $D(y)$.

For a given arc, (x, y), the *support count* of x with respect to y, denoted $sc(x, y)$, is defined by $sc(x, y) = \min(\{sc(x, y, a) : a \in D(x)\})$. If $sc(\,x, y\,) > |\,R(y)\,|$, then each value in $D(x)$ is supported by y. This condition is a special version of what is called a revision condition in [Mehta and van Dongen, 2005b]. For example, if $|\,D_{ac}(y)\,| = 20$, $sc(x, y) = 2$ and $|\,R(y)\,| = 1$ then each value $a \in D(x)$ is supported by some value of $D(y)$ with a probability of 1. Hence, there is no need to revise $D(x)$ against $D(y)$.

In the examples discussed above, if $|R(y)| = 2$, then SC and RC will fail. Despite of having a high probability of the support existence, the algorithm will be forced to search for it in $D(y)$. Avoiding the process of seeking a support with a high probability can also be worthwhile. In order to do so, the notions of a *probabilistic support condition* (PSC) and a *probabilistic revision condition* (PRC) were introduced in [Mehta and van Dongen, 2005a]. The PSC holds for (x, a) with respect to y, if the probability that a support exists for (x, a) in $D(y)$ is above some, carefully chosen, threshold. The PRC holds for an arc (x, y), if the probability of having some support for each value $a \in D(x)$ in $D(y)$, is above some, carefully chosen, threshold.

Let $Ps_{(x,y,a)}$ be the probability that there exists some support for (x, a) in $D(y)$. If we assume that each value in $D_{ac}(y)$ is equally likely to be removed during search, then it follows that

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

Let T, $0 \leq T \leq 1$, be some desired threshold. If $Ps_{(x,y,a)} \geq T$ then (x, a) has some support in $D(y)$ with a probability

of T or more. This condition is called a *Probabilistic Support Condition* (PSC) in [Mehta and van Dongen, 2005a]. If it holds, then the process of seeking a support for (x, a) in $D(y)$ is avoided. For example, if $T = 0.9$, $|D_{ac}(y)| = 20$, $sc(x, y, a) = 2$, and this time $|R(y)| = 2$, then PSC holds and (x, a) has a support in $D(y)$ with a probability of 0.994.

Let $Ps_{(x,y)}$ be the least probability of the values of $D_{ac}(x)$ that there exists some support in y. If $Ps_{(x,y)} \geq T$ then, each value in $D(x)$ is supported by y with a probability of T or more. This condition is called a *Probabilistic Revision Condition*. If it holds then the revision of $D(x)$ against $D(y)$ is skipped. The interested reader is referred to [Mehta and van Dongen, 2005a] for details.

3.1 PAC-3

PSC and PRC can be embodied in any coarse-grained AC algorithm. Figure 1 depicts the pseudocode of PAC-3, the result of incorporating PSC and PRC into AC-3 [Mackworth, 1977]. Depending on the threshold, sometimes it may achieve less than full arc consistency. If PSC holds then the process of identifying a support is avoided. This is depicted in Figure 2. If PRC holds then it is exploited *before* adding the arcs to the queue, in which case the corresponding arc (x, y) is not added to the queue. This is depicted in Figure 1. Note that coarse-grained arc consistency algorithms require $\mathcal{O}(ed)$ revisions in the worst-case to make the problem arc consistent. Nevertheless, the maximum number of *effective revisions* (that is when at least a single value is deleted from a domain) cannot exceed $\mathcal{O}(nd)$, irrespective of whether the algorithm used is optimal or non-optimal. Thus, in the worst case, it can perform $\mathcal{O}(ed - nd)$ ineffective revisions. In order to use PSC and PRC, the support count for each arc-value pair must be computed prior to search. If T = 0 then PAC-3 makes the problem FC consistent. If T = 1 then PAC-3 makes the problem arc consistent. The support counters are represented in $\mathcal{O}(e\,d)$, which exceeds the space-complexity of AC-3. Thus, the space-complexity of PAC-3 becomes $\mathcal{O}(e\,d)$. The worst-case time complexity of PAC-3 is $\mathcal{O}(e\,d^3)$. The use of PSC and PRC in PAC-3 is presented in such a way that the idea is made as clear as possible. This should not be taken as the real implementation. [Mehta and van Dongen, 2005a] describes how to implement PSC and PRC efficiently.

```
Function PAC-3( current_var ) :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
      if y = current_var then
        revise(x, y, change_x)
      else
        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;
```

Figure 1: PAC-3

```
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
      \* do nothing *\
    else
      if ∄b ∈ D(y) such that b supports a then
        D(x) := D(x) \ { a }
        change_x := True
end;
```

Figure 2: Algorithm $revise_p$

4 Experimental Results

4.1 Introduction

Overview

We present some empirical results demonstrating the practical use of PSC and PRC. We investigate several probabilistic consistencies, particularly, Probabilistic Arc Consistency (PAC), Probabilistic Singleton Arc Consistency (PSAC), and a limited version of PSAC (PLSAC). First, we find out the threshold at which Maintaining PAC (MPAC) performs the best by experimenting on model B random problems. Next, we carry out experiments to determine how well MPAC does when compared to MAC and FC. The results for FC are also included to show that MPAC is not only better than MAC on problems on which FC is better than MAC but as well as on the problems on which MAC is better than FC. Finally, we examine the usefulness of PSAC and PLSAC when used as a preprocessor and when maintained during search.

Problems Studied

We have used model B [Gent *et al.*, 2001] random problems and several problems from the literature to evaluate the competence of probabilistic consistencies. In model B, a random CSP instance is represented as $\langle n, d, c, t \rangle$ where n is the number of variables, d is the uniform domain size, c is the number of constraints, and t is the number of forbidden pairs of values. For each combination of parameter, 50 instances are generated and their mean performances is reported. The remaining problems, except odd_even_n [2] have all been used as benchmarks for the First International CSP Solver Competition and are described in [Boussemart *et al.*, 2005]. Informally, the odd-even_n problem is an undirected constraint graph with a cycle with n variables. The domain of each variable is $\{1, 2, 3, 4\}$. For each constraint C_{xy}, if a (x, a) is odd (even) then it is supported by even (odd) values of $D(y)$. The problem is unsatisfiable if n is odd.

Throughout, it has been our intention to compare *general-purpose* propagation algorithms, and not special-purpose algorithms, which make use of the semantics of the constraints. Some readers may argue that the probabilistic constraint propagation algorithms should have been compared with special-purpose propagation algorithms. For example, it is well known that for anti-functional constraints, there is no need to look for a support unless there is only one value left.

[2] The problem is mentioned in the invited talk "Towards theoretical frameworks for comparing constraint satisfaction models and algorithms" by Peter van Beek in CP'2001.

Table 1: Comparison between FC, MAC, and MPAC (with T=0.9) on random problems.

$\langle n, d, c, t \rangle$	Algorithm	#chks	time	#vn
$\langle 20, 30, 190, 271 \rangle$	FC	60,435,849	13.084	1,336,291
	MAC	223,034,030	18.203	281,954
	MPAC	49,496,904	11.602	443,292
$\langle 20, 40, 190, 515 \rangle$	FC	286,321,115	61.306	6,181,026
	MAC	1,052,973,654	86.216	1,287,709
	MPAC	234,024,186	51.102	2,048,660
$\langle 50, 10, 500, 20 \rangle$	FC	121,226,499	31.230	2,581,395
	MAC	276,708,237	35.178	334,566
	MPAC	72,993,605	27.913	678,370
$\langle 150, 15, 400, 134 \rangle$	FC	3,809,711,519	2754.135	346,075,062
	MAC	908,870,372	217.939	709,387
	MPAC	706,309,188	193.746	1,149,995
$\langle 100, 20, 290, 240 \rangle$	FC	555,592,958	119.888	17,268,137
	MAC	135,546,970	17.777	97,939
	MPAC	81,031,221	16.731	132,983
$\langle 90, 20, 222, 272 \rangle$	FC	58,969,203	8.686	1,871,351
	MAC	18,255,078	1.086	7,719
	MPAC	8,019,466	1.081	10,734

Indeed, this should improve constraint propagation. However, probabilistic algorithms can be improved similarly.

Implementation Details

AC-3 is used to implement the arc consistency component of MAC and SAC. The reason why AC-3 is chosen is that it is easier to implement and is also efficient. For example, the best solvers in the binary and overall category of the First International CSP Solver Competition were based on AC-3. Similarly, PAC-3 is used to implement the probabilistic arc consistency component of the probabilistic versions of MAC and SAC. SAC-1 is used to implement singleton arc consistency. All algorithms were equipped with a $dom/wdeg$ [Boussemart *et al.*, 2004] conflict-directed variable ordering heuristic. The performance of the algorithms is measured in terms of checks (#chks), time in seconds (time), the number of revisions (#rev), and the number of visited nodes (#vn). 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.

4.2 Probabilistic Arc Consistency

Maintaining probabilistic arc consistency in such a way that the amount of ineffective constraint propagation is minimised and simultaneously the least amount of effective propagation is avoided depends heavily on the threshold value T. Therefore, a natural question is for which values of T, MPAC resolves the trade-off, in terms of time, between the effort required to search and that required to detect inconsistent values. To find this out, experiments were designed to examine the behaviour of MPAC with different thresholds ranging from 0 to 1 in steps of 0.02 on random problems. Note that at $T = 1$, MPAC maintains full arc consistency and at $T = 0$, it becomes forward checking. It is well known that on hard dense, loosely constrained random CSPs, FC performs better than MAC and on hard sparse, tightly constrained random CSPs, MAC performs better than FC [Chmeiss and Saïs, 2004]. Therefore, we studied MPAC with these classes of problems.

Random Problems

In our investigations, we found that inferring the existence of a support with a likelihood, roughly between 0.8 and 0.9, enables MPAC to outperform both MAC and FC on both classes of problems. Table 1 presents results on hard dense, loosely constrained problems (first 3 rows), on which FC is better than MAC, and on hard sparse, tightly constrained problems (last 3 rows) on which MAC is better than FC. Results demonstrate that MPAC (T = 0.9) is better than the best of FC and MAC on both classes of random problems. Also, the number of nodes visited by MPAC (T = 0.9) is nearer to MAC than those visited by FC. This is the first time an algorithm has been presented that outperforms MAC and FC on both classes of problems. Seeing the good performance of MPAC for threshold values ranging between 0.8 and 0.9, we decided to choose $T = 0.9$ for the remainder of the experiments.

Table 2: Comparison between FC, MAC, and MPAC (with T = 0.9) on a variety of known problems.

Problem	Algorithm	#chks	time	#vn	#rev
frb40-19	FC	74,944,982	11.564	1,508,169	13,406,750
	MAC	177,424,742	11.488	158,816	25,493,093
	MPAC	47,423,325	8.407	268,115	14,899,791
frb45-21	FC	972,804,340	196.002	17,843,238	172,256,029
	MAC	1,501,335,748	215.092	1,861,016	333,222,409
	MPAC	599,255,164	145.864	3,532,973	195,139,744
scen5	FC	20,492,062	1.791	384,458	1,515,356
	MAC	879,425	0.061	498	21,934
	MPAC	7,713,406	0.466	59,048	585,416
scen11	FC	2,891,066	0.325	11,788	131,020
	MAC	8,285,681	0.517	3,749	276,601
	MPAC	10,391,078	0.298	4,624	98,334
scen11_f7	FC	2,203,289,973	550.03	14,543,828	170,755,686
	MAC	8,028,568,317	777.38	3,471,542	408,551,247
	MPAC	1,518,283,911	360.86	6,935,594	157,729,437
scen11_f6	FC	5,372,894,595	1,295.79	36,611,678	421,021,114
	MAC	16,066,360,455	1,767.93	7,643,388	817,697,705
	MPAC	4,233,509,071	954.79	19,056,709	413,380,973
bqwh15_106	FC	206,389	0.054	11,588	77,051
	MAC	231,871	0.036	1,594	91,721
	MPAC	81,520	0.027	1,893	50,785
bqwh18_141	FC	9,368,452	3.093	500,794	3,615,749
	MAC	5,516,466	1.224	23,229	2,283,944
	MPAC	1,402,175	0.788	27,232	1,101,786
geom	FC	12,213,843	2.487	149,151	2,040,421
	MAC	49,937,553	4.119	28,220	5,895,601
	MPAC	7,313,488	2.006	43,511	2,371,765
qa-5	FC	129,662,303	7.510	1,656,165	15,002,347
	MAC	52,300,903	3.999	94,531	11,736,269
	MPAC	3,2433,059	2.342	105,304	4,731,980
qa-6	FC	2,135,903,729	1,526.256	140,853,896	1,704,256,885
	MAC	911,855,065	129.094	672,252	164,556,262
	MPAC	960,010,551	104.517	1,462,548	99,707,970
odd_even_27	FC	287,382,172	37.730	96,173,136	96,873,380
	MAC	948	0.000	4	162
	MPAC	1,218	0.000	4	162
$K_{25} \oplus Q_8$	FC	4,190,654,127	211.518	33,918,459	40,155,040
	MAC	23,117,603,455	308.070	122,934	1,804,139
	MPAC	5,512,831,722	180.914	246,026	2,004,268
$K_{25} \otimes Q_8$	FC	5,959,952,959	503.351	32,977,840	174,400,084
	MAC	23,472,721,326	335.107	122,549	12,963,828
	MPAC	5,550,542,521	198.914	260,750	4,182,518

Problems from the Literature

Table 2 shows the results obtained on some instances of variety of known problems: (1) forced random binary problems *frb-40-19* and *frb-45-21* (2) RLFAP instances *scen5* and *scen11*, (3) modified RLFAP instances *scen11_f7* and *scen11_f6*, (4) average of 100 satisfiable instances of balanced Quasigroup with Holes problems *bqwh15_106* and *bqwh18_141* (5) average of 100 (92 satisfiable, 8 satisfiable) instances of geometric problem *geom* (6) two instances of attacking prime queen problem *qa-5* and *qa-6*, (7) one instance of odd-even problem *odd-even_27*, (8) two instances of queen-knights problem $K_{25} \oplus Q_8$ and $K_{25} \otimes Q_8$.

One can observe in Table 2, that in terms of time, on some problems, FC is better than MAC, while on some, MAC is better than FC. It is surprising that FC is not much inferior than MAC as anticipated. This is partially because of the robust-

ness of the conflict directed variable ordering heuristic. For easy problems, due to the expense entailed by computing the number of supports for each arc-value pair, MPAC may not be beneficial. However, the time required to initialise the support counters is not much. Furthermore, for all the harder instances, that require at least 1 second to solve, MPAC generally pays off, by avoiding much ineffective propagation. In summary, by visiting few extra nodes than MAC, MPAC is able to save much ineffective propagation and solves the problem more quickly than both MAC and FC.

Note that, if the domain size is small or if the values have only a few supports, then keeping the threshold high, fails PSC and PRC quickly, since the likelihood of support existence decreases rapidly with respect to the number of values removed from the domain and the prospect of effective propagation increases. This in turn permits MPAC to act like MAC. For example, in case of odd_even_27, the domain size of each variable is 4 and each arc-value pair is supported by 2 values. For this problem, FC visits exactly $2^{n+1} - 4$ nodes, while MAC visits only 4 nodes and for $T = 0.9$ MPAC also visits only 4 nodes. The probability of support existence for a value (x, a) in $D(y)$ becomes 0.66, as soon as 2 values are removed from y, and since the threshold is set to 0.9, both PSC and PRC fails, enabling MPAC to maintain full arc consistency. This again shows that MPAC with the right threshold is able to resolve when to propagate and when not to.

We also experimented with MAC-2001 which uses an optimal arc consistency algorithm AC-2001. However, for almost all the problems, MAC-2001 was consuming more time than MAC-3, since there is a huge overhead of maintaining auxiliary data structures [van Dongen, 2003].

4.3 Probabilistic Singleton Arc Consistency

Although there are stronger consistencies than arc consistency, the standard is to make the problem full/partial arc consistent before and/or during search. However, recently, there has been a surge of interest in SAC [Bessière and Debruyne, 2005; Lecoutre and Cardon, 2005] as a preprocessor of MAC. The advantage of SAC is that it improves the filtering power of AC without changing the structure of the problem as opposed to other stronger consistencies such as k-consistency $(k > 2)$ and so on. But, establishing SAC can be expensive, and can be a huge overhead, especially for loose problems [Lecoutre and Cardon, 2005]. We investigate the *advantages* and *disadvantages* of applying PSC and PRC to SAC.

Enforcing SAC in SAC-1 [Debruyne and Bessière, 1997] style works by having an outer loop consisting of variable-value pairs. For each (x, a) if $ac(\mathcal{P}|_{x=a}) = \perp$, then it deletes a from $D(x)$. Then it enforces AC. Should this fail then the problem is not SAC. The main problem with SAC-1 is that deleting a single value triggers the addition of all variable-value pairs in the outer loop. The *restricted* SAC (RSAC) algorithm [Prosser *et al.*, 2000] avoids this triggering by considering each variable-value pair only once. We propose *limited* SAC (LSAC) which lies between *restricted* SAC and SAC. The idea is that if a variable-value pair (x, a) is found arc-inconsistent then, only the pairs involving neighbours of x as a variable are added in the outer-loop. Our experience is that LSAC is more effective than RSAC.

Table 3: Comparison between SAC, LSAC, PSAC and PLSAC.

problem		SAC	LSAC	PSAC	PLSAC
bqwh15_106	#chks	274,675	186,172	14,384	11,436
	time	0.026	0.020	0.006	0.005
	#rem	23	23	23	23
bqwh18_114	#chks	409,344	299,821	44,534	36,973
	time	0.040	0.030	0.010	0.008
	#rem	15	15	15	15
qa-7	#chks	872,208,323	895,600,777	17,484,337	17,484,337
	time	50.885	52.390	1.929	1.979
	#rem	65	65	65	65
qa-8	#chks	3,499,340,592	3,533,080,066	51,821,816	51,821,816
	time	249.696	290.534	9.790	11.496
	#rem	160	160	160	160
$K_{25} \oplus Q_8$	#chks	206,371,887	206,371,887	16,738,737	16,738,737
	time	2.407	2.446	0.469	0.450
	#rem	3,111	3,111	3,111	3,111
$K_{25} \otimes Q_8$	#chks	1,301,195,918	1,301,195,918	19,252,527	19,252,527
	time	13.473	13.469	0.613	0.635
	#rem	3,112	3,112	3,112	3,112
scen5	#chks	9,896,112	11,791,192	2,793,188	2,348,189
	time	0.700	0.858	0.126	0.120
	#rem	13,814	13,814	13,794	13,794
scen11	#chks	622,229,041	622,229,041	16,376,619	16,376,619
	time	21.809	21.809	3.005	3.005
	#rem	0	0	0	0
scen11_f6	#chks	292,600,735	292,600,735	16,415,998	16,415,998
	time	11.775	11.763	0.811	0.813
	#rem	3660	3660	3660	3660
js-1	#chks	600,508,958	600,508,958	14,100,504	14,100,504
	time	15.549	15.411	0.446	0.415
	#rem	0	0	0	0
js-2	#chks	985,446,461	985,446,461	18,291,441	18,291,441
	time	24.963	25.393	0.601	0.631
	#rem	0	0	0	0
co-25-10-20	#chks	5,272,064	6,184,748	744,324	611,475
	time	0.258	0.303	0.076	0.061
	#rem	392	391	392	392
co-75-1-80	#chks	2,143,350	23,359	184,507	184,507
	time	0.087	0.001	0.012	0.013
	#rem	59	59	56	56

The algorithms SAC, LSAC, PSAC (Probabilistic SAC), and PLSAC (Probabilistic LSAC) were applied to forced random problems, RLFAP, modified RLFAP, quasi-group with holes, queens-knights, attacking prime queen, job-shop instances, composed random problems. Table 3 presents results for only a few instances of the above problems, due to space restriction. Again, the value of threshold used for PSC and PRC is 0.9. In Table 3 #rem denotes the number of removed values. The intention is not to test if the preprocessing by SAC has any effect in the global cost of solving the problem, but to see if the same results can be achieved by doing considerably less computation by using probabilistic support inference. When the input problem is already singleton arc consistent, PSAC and PLSAC avoid most of the unnecessary work. For example, for job-shop instances *js*-1 and *js*-2, both PSAC and PLSAC spend at least 34 times less time than their counterparts. Even when the input problem is not singleton arc consistent, probabilistic versions of the algorithms are as efficient as the original versions. For most of the problems, they remove *exactly* the same number of values as removed by SAC and LSAC, but consume significantly less time. For example, in case of attacking queen problems, all the algorithms remove the same number of values. However, PSAC and PLSAC are quicker by an order of at least 27.

Seeing the good performance of PSAC and PLSAC, the immediate question arises: can we afford to maintain them during search? So far SAC has been used only as a preprocessor. Maintaining SAC can reduce the number of branches significantly but at the cost of much constraint propagation, which may consume a lot of time. Maintaining it even for depth 1

Table 4: Comparison of MAC, MSAC, MLSAC against their probabilistic versions (T = 0.9) on structured problems. (Checks are in terms of 1000s.)

problem		MAC	MPAC	MSAC	MPSAC	MLSAC	MPLSAC
	#chks	5,433	1,220	45,165	3,835	31,580	1,418
qwh-20	time	2.17	1.34	10.83	3.08	8.69	1.32
(easy)	#vn	21,049	21,049	570	570	1,101	1,101
	#chks	186,872	42,769	756,948	70,666	293,152	34,330
qwh-20	time	75.35	51.01	172.23	54.56	76.86	31.39
(hard)	#vn	693,662	693,662	3,598	3,598	7,136	7,136
	#chks	502,148	93,136	807,478	57,235	656,270	40,650
qwh-25	time	252.89	154.00	236.20	63.71	221.46	53.506
(easy)	#vn	1,348,291	1,348,291	2,525	2,525	10,283	10,283
	#chks	600,697	670,370	2,090,689	768,556	954,767	211,826
qcp-20	time	2,753.77	1,990.31	8,242.28	1,679.01	5,688.19	877.540
(easy)	#vn	26,191,095	26,191,095	107,624	107,624	586,342	586,342

within search has been found very expensive in [Prosser *et al.*, 2000]. We investigate if PSAC and PLSAC can be maintained within search economically. Table 4 shows the comparison of MAC, MPAC, MSAC (maintaining SAC), MLSAC (maintaining LSAC), MPSAC (maintaining PSAC), and MPLSAC (maintaining PLSAC) on structured problems. Mean results are shown only for quasigroup with holes (QWH) and quasi-completion problems (QCP) categorised as *easy* and *hard*. Note that here *easy* does not mean easy in the real sense. The results are first of their kind and highlight the following points: (1) the probabilistic version of the algorithm is better than its corresponding original version, (2) maintaining full or probabilistic (L)SAC reduces the branches of the search tree drastically, (3) though MLSAC and MPLSAC visit a few nodes more than MSAC and MPSAC, their run-times are low, (4) MPLSAC is the best in terms of checks and solution time.

In our experiments, MPSAC/MPLSAC outperformed MSAC/MLSAC for almost all the problems which we have considered. But, when compared to MAC and MPAC, they were found to be expensive for most of the problems except for quasi-group problems. However, this observation is made only for threshold value 0.9. Thorough testing remains to be done with different values of T.

5 Conclusions and Future Work

This paper investigates the use of probabilistic approach to reduce ineffective constraint propagation. The central idea is to avoid the process of seeking a support when there is a high probability of its existence. Inferring the existence of a support with a high probability allows an algorithm to save a lot of checks and time and slightly affects its ability to prune values. For example, by visiting a few nodes more than MAC, MPAC is able to outperform both MAC and FC on a variety of different problems. Similarly, enforcing probabilistic SAC almost always enforces SAC, but it requires significantly less time than SAC. Overall, experiments highlight the good performance of probabilistic support condition and probabilistic revision condition. We believe that the idea of probabilistic support inference deserves further investigation. The notions of PSC and PRC can be improved further by taking into account the semantics of the constraints.

References

[Bessière and Debruyne, 2005] C. Bessière and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 54–59, 2005.

[Bessière and Régin, 2001] C. Bessière and J.-C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 309–315, 2001.

[Boussemart *et al.*, 2004] F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs. Boosting systematic search by weighting constraints. In *Proceedings of the 13th European Conference on Artificial Intelligence*, 2004.

[Boussemart *et al.*, 2005] F. Boussemart, F. Hemery, and C. Lecoutre. Description and representation of the problems selected for the 1st international constraint satisfaction solver competition. In *Proceedings of the 2nd International Workshop on Constraint Propagation and Implementation, Volume 2 Solver Competition*, 2005.

[Chmeiss and Saïs, 2004] A. Chmeiss and L. Saïs. 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.

[Debruyne and Bessière, 1997] R. Debruyne and C. Bessière. Some practical filtering techniques for the constraint satisfaction problem. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 412–417, 1997.

[Gent *et al.*, 2001] I.P. Gent, E. MacIntyre, P. Prosser, B. Smith, and T. Walsh. Random constraint satisfaction: Flaws and structure. *Journal of Constraints*, 6(4), 2001.

[Horsch and Havens, 2000] M. Horsch and W Havens. Probabilistic arc consistency: A connection between constraint reasoning and probabilistic reasoning. In *16th Conference on Uncertainity in Artificial Intelligence*, 2000.

[Lecoutre and Cardon, 2005] C. Lecoutre and S. Cardon. A greedy approach to establish singleton arc consistency. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 199–204, 2005.

[Mackworth, 1977] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mehta and van Dongen, 2005a] D. Mehta and M.R.C. van Dongen. Maintaining probabilistic arc consistency. In *Proceedings of the 2nd International Workshop on Constraint Propagation and Implementation*, pages 49–64, 2005.

[Mehta and van Dongen, 2005b] 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.

[Prosser *et al.*, 2000] P. Prosser, K. Stergiou, and T. Walsh. Singleton consistencies. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming*, pages 353–368, 2000.

[van Dongen, 2003] M.R.C. van Dongen. Lightweight MAC algorithms. In *Proceedings of the 23rd SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence*, pages 227–240. Springer, 2003.