Singleton Consistencies

Patrick Prosser[1], Kostas Stergiou[2], and Toby Walsh[3]

[1] Glasgow University, Glasgow, Scotland. pat@dcs.strath.ac.uk
[2] University of Strathclyde, Glasgow, Scotland. ks@cs.strath.ac.uk
[3] University of York, York, England. tw@cs.york.ac.uk

Abstract. We perform a comprehensive theoretical and empirical study of the benefits of singleton consistencies. Our theoretical results help place singleton consistencies within the hierarchy of local consistencies. To determine the practical value of these theoretical results, we measured the cost-effectiveness of pre-processing with singleton consistency algorithms. Our experiments use both random and structured problems. Whilst pre-processing with singleton consistencies is not in general beneficial for random problems, it starts to pay off when randomness and structure are combined, and it is very worthwhile with structured problems like Golomb rulers. On such problems, pre-processing with consistency techniques as strong as singleton generalized arc-consistency (the singleton extension of generalized arc-consistency) can reduce runtimes. We also show that limiting algorithms that enforce singleton consistencies to a single pass often gives a small reduction in the amount of pruning and improves their cost-effectiveness. These experimental results also demonstrate that conclusions from studies on random problems should be treated with caution.

1 Introduction

Local consistency techniques lie close to the heart of constraint programming's success. They can prune values from the domain of variables, saving much fruitless exploration of the search tree. They can also terminate branches of the search tree, again saving much fruitless exploration. But how do we balance effort between inference (enforcing some level of local consistency) and search (exploring partial assignments)? If we maintain a local consistency technique at each node in the search tree, then experience suggests that it must not be too expensive to enforce. We may, however, be able to afford a (relatively expensive) local consistency technique if it is only used for pre-processing or for the first few levels of search. We are then faced with a large number of choices as a vast menagerie of local consistencies have been defined over the last few years. Debruyne and Bessiere identified singleton arc-consistency as one of the most promising candidates [DB97]. This paper therefore explores its usefulness in greater detail, as well as that of other singleton consistencies.

2 Formal background

A constraint satisfaction problem (CSP) P is a triple (X, D, C). X is a set of variables. For each $x_i \in X$, D_i is the domain of the variable. Each k-ary

constraint $c \in C$ is defined over a set of variables $(x_1, \ldots x_k)$ by the subset of the Cartesian product $D_1 \times \ldots D_k$ which are consistent values. Following [DB97], we denote by $P|_{D_i=\{a\}}$ the CSP obtained by assigning value a to variable x_i. An all-different constraint over the variables $(x_1, \ldots x_k)$ allows the values $\{(a_1, \ldots a_k) \mid a_i \in D_i \ \&\forall u, v. a_u \neq a_v\}$. A solution is an assignment of values to variables that is consistent with all constraints.

Many lesser levels of consistency have been defined for binary constraint satisfaction problems (see [DB97] for additional references). A problem is (i, j)-consistent iff it has non-empty domains and any consistent instantiation of i variables can be extended to a consistent instantiation involving j additional variables [Fre85]. A problem is strong (i, j)-consistent iff it is (k, j)-consistent for all $k \leq i$. A problem is arc-consistent (AC) iff it is $(1, 1)$-consistent. A problem is path-consistent (PC) iff it is $(2, 1)$-consistent. A problem is strong path-consistent iff it is strong $(2, 1)$-consistent. A problem is path inverse consistent (PIC) iff it is $(1, 2)$-consistent. A problem is neighbourhood inverse consistent (NIC) iff any value for a variable can be extended to a consistent instantiation for its immediate neighbourhood [FE96]. A problem is restricted path-consistent (RPC) iff it is arc-consistent and if a variable assigned to a value is consistent with just a single value for an adjoining variable then for any other variable there exists a value compatible with these instantiations. A problem is singleton arc-consistent (SAC) iff it has non-empty domains and for any instantiation of a variable, the resulting subproblem can be made arc-consistent.

Many of these definitions can be extended to non-binary constraints. For example, a (non-binary) CSP is generalized arc-consistent (GAC) iff for any variable in a constraint and value that it is assigned, there exist compatible values for all the other variables in the constraint [MM88]. Regin gives an efficient algorithm for enforcing generalized arc-consistency on a set of all-different constraints [Reg94]. We can also maintain a level of consistency at every node in a search tree. For example, the MAC algorithm for binary CSPs maintains arc-consistency at each node in the search tree [Gas79]. As a second example, on a non-binary problem, we can maintain generalized arc-consistency (MGAC) at every node in the search tree.

3 Singleton consistencies

The notion of a singleton consistency is general, and can be applied to other levels of local consistency than arc-consistency. For instance, a problem is singleton restricted path-consistent (SRPC) iff it has non-empty domains and for any instantiation of a variable, the resulting subproblem can be made restricted path-consistent [DB97]. As a second (and we believe previously undefined) example, a non-binary problem is singleton generalized arc-consistent (SGAC) iff it has non-empty domains and for any instantiation of a variable, the resulting subproblem can be made generalized arc-consistent. As generalized arc-consistency is itself a high level of consistency to achieve (see, for example, [SW99]), singleton generalized arc-consistency is a very high level of consistency to achieve. However, as our experimental results demonstrate, it can be very worthwhile enforcing it.

One advantage of singleton consistencies (which is shared with inverse consistencies like path inverse consistency and neighbourhood inverse consistency, as well as with restricted path-consistency) is that enforcing them only requires values to be pruned from the domain of variables. Enforcing path-consistency, by comparison, can change the constraint graph by adding new binary constraints. Note that a singleton consistency can be achieved using any algorithm that achieves the relevant local consistency. The definition of singleton consistency only insists we can make the resulting subproblem locally consistent. We are not interested in what values need to be pruned (or nogoods added) to make the subproblem locally consistent. We can therefore use a lazy approach to enforcing the local consistency. For example, we can use the lazy AC7 algorithm [SRGV96] when achieving SAC.

In this paper, we have used the algorithm proposed in [DB97] to achieve SAC and a simple generalization of this algorithm to $n-$ary CSPs to achieve SGAC. To achieve SAC (SGAC) in a CSP P, this algorithm first achieves AC (GAC) and then goes through each variable x_i in P. For every value a in the domain of x_i it checks if the subproblem $P|_{D_i=\{a\}}$ is AC (GAC). If it is not then a is removed from the domain of x_i and AC (GAC) is enforced. Failure to do so means that P is not SAC (SGAC). The process of going through the variables in the CSP continues while new inconsistent values are detected and deleted. In short, there is an inner loop that goes through the variables and an outer loop that keeps this process going while new values are deleted.

The worst-case complexity of achieving SAC is $O(en^2d^4)$, where e is the number of constraints, n the number of variables, and d the domain size. For non-binary constraints, if we assume that GAC-schema [BR97] is used to enforce GAC then the worst case complexity of achieving SGAC is $O(en^2d^{2k})$, where k is the arity of the constraints. For the specialized case of all-different constraints, taking advantage of Regin's algorithm means that SGAC can be achieved with $O(cn^4d^4)$ worst-case complexity, where c is the number of all-different constraints.

We can reduce the average cost of the above algorithm by making just one pass, i.e., going through the variables and deleting inconsistent values only once. This deletes less values and thus achieves a lesser level of consistency than SAC (SGAC), but as our experimental results show, is, in some cases, very cost-effective. We call this algorithm *restricted* SAC (SGAC).

4 Theoretical results

Following [DB97], we call a consistency property A stronger than B ($A \geq B$) iff in any problem in which A holds then B holds, and strictly stronger ($A > B$) iff it is stronger and there is at least one problem in which B holds but A does not. We call a local consistency property A incomparable with B ($A \sim B$) iff A is not stronger than B nor vice versa. Finally, we call a local consistency property A equivalent to B iff A implies B and vice versa. The following relationships summarize the most important results from [DB97] and elsewhere: strong PC $>$ SAC $>$ PIC $>$ RPC $>$ AC, NIC $>$ PIC, NIC $\sim$ SAC, and NIC $\sim$ strong PC.

Our first result shows that a singleton consistency is stronger than the corresponding local consistency. A local consistency property A is monotonic iff when a problem is A-consistent then any subproblem formed by instantiating a variable is also A-consistent. Most local consistencies (e.g. all those introduced so far) are monotonic.

Theorem 1. *If A-consistency is monotonic then singleton A-consistency $\geq$ A-consistency.*

Proof. Immediate from the definitions of monotonic and singleton consistency.

Note that it is possible to construct (admittedly bizarre) local consistencies which are not monotonic. For example, consider a weakened form of AC which is equivalent to AC on every arc except the arc between variables x_1 and x_2 when either are instantiated. If we take a problem in which the arc between x_1 and x_2 is not AC, then this weakened form of AC will detect the arc-inconsistency but the singleton consistency will not. On this problem, the singleton consistency is actually weaker than the corresponding local consistency. Note also that a singleton consistency is not necessarily *strictly* stronger than the corresponding monotonic local consistency. For example, on problems whose constraint graphs are trees, SAC is only equivalent to AC (since arc-consistency is already enough to guarantee global consistency).

Our next result allows us to map many previous results up to singleton consistencies. For example, as RPC is stronger than AC, singleton RPC is stronger than SAC.

Theorem 2. *If A-consistency $\geq$ B-consistency then singleton A-consistency $\geq$ singleton B-consistency.*

Proof. Consider a problem that is singleton A-consistent, and a subproblem formed from instantiating a variable. Now this subproblem is A-consistent. As $A \geq B$, this subproblem is B-consistent. Hence the original problem is singleton B-consistent.

Note that we do not need A-consistency or B-consistency to be monotonic for this proof to work. Debruyne and Bessiere prove that SAC is strictly stronger than PIC [DB97]. We can generalize this proof to show that singleton (i, j)-consistency is strictly stronger than $(i, j+1)$-consistency. Debruyne and Bessiere's result is then a special case for $i = j = 1$. In addition, [DB97] does not give the proof of strictness, so for completeness we give it here for the case $i = j = 1$.

Theorem 3. *Singleton (i, j)-consistency $>$ $(i, j + 1)$-consistency.*

Proof. Consider a problem that is singleton (i, j)-consistent, and the subproblem resulting from any possible instantiation. This subproblem is (i, j)-consistent. Hence, for any consistent instantiation for i variables in the subproblem, we can extend it to j other variables. That is, for any instantiation of i variables in the original problem, we can extend it to $j + 1$ other variables. Hence the original

problem is $(i, j+1)$-consistent. To show strictness, consider $i = j = 1$ and a problem in four 0-1 variables with the constraints $x_1 \neq x_2$, $x_2 \neq x_3$, $x_2 \neq x_4$, $x_3 \neq x_4$. This is path inverse consistent. However, enforcing SAC proves that the problem is insoluble since if we instantiate x_1 with either of its values, the resulting subproblem cannot be made arc-consistent.

Debruyne and Bessiere also prove that strong PC is strictly stronger than SAC [DB97]. We can also generalize this proof, showing that strong $(i+1, j)$-consistency is strictly stronger than singleton (i, j)-consistency. Debruyne and Bessiere's result is again a special case for $i = j = 1$. As before, [DB97] does not give the proof of strictness, so for completeness we give it here for the case $i = j = 1$.

Theorem 4. *Strong $(i+1, j)$-consistency > singleton (i, j)-consistency.*

Proof. Consider a problem that is strongly $(i+1, j)$-consistent. Any consistent instantiation for $i+1$ variables can be extended to j other variables. As the original problem was strongly $(i+1, j)$-consistent, it is (i, j)-consistent. Hence a subproblem formed by instantiating one variable is (i, j)-consistent, and any consistent instantiation of i variables can be extended to j other variables. Thus the original problem is singleton (i, j)-consistent. To show strictness, consider $i = j = 1$ and a problem in three 0-1 variables with $x_1 \neq x_2$ and $x_1 \neq x_3$. The problem is SAC. But it is not path-consistent since the consistent partial assignment $x_2 = 0$ and $x_3 = 1$ cannot be extended. Enforcing path-consistency adds the constraint $x_2 = x_3$.

The last two results show that singleton (i, j)-consistency is sandwiched between strong $(i+1, j)$-consistency and $(i, j+1)$-consistency. Finally, we give some results concerning SGAC. Whilst this is a very high level of consistency to achieve in general, our experiments show that it can be very worthwhile provided we have an efficient algorithm to achieve it (as we do for the all-different constraint). In [SW99], GAC was compared against binary consistencies (like SAC) on decomposable non-binary constraints. These are non-binary constraints that can be represented by binary constraints on the same set of variables [Dec90]. For example, an all-different constraint can be decomposed into a clique of not-equals constraints. Decomposable constraints are a special case of non-binary constraints where comparisons between the binary and non-binary representations are very direct. Constraints which are not decomposable (like parity constraints) require us to introduce additional variables to represent them using binary constraints. These additional variables make comparisons more complicated.

Theorem 5. *On decomposable non-binary constraints, singleton generalized arc-consistency is strictly stronger than singleton arc-consistency on the binary decomposition.*

Proof. The proof follows immediately from Theorem 1, and the result of [SW99] that GAC is strictly stronger than AC on the binary decomposition. To show

strictness, consider three all-different constraints on $\{x_1, x_2, x_3\}$, on $\{x_1, x_2, x_4\}$, and on $\{x_1, x_3, x_4\}$, in which all variables have the domain $\{1, 2, 3\}$. The binary decomposition is SAC. But enforcing SGAC proves that the problem is unsatisfiable.

Though SGAC is a very high level of consistency to enforce, it is incomparable in general to both strong PC and NIC on the binary decomposition.

Theorem 6. *On decomposable non-binary constraints, singleton generalized arc-consistency is incomparable to strong path-consistency and to neighbourhood inverse consistency on the binary decomposition.*

Proof. Consider a problem with six all-different constraints on $\{x_1, x_2, x_3\}$, on $\{x_1, x_3, x_4\}$, on $\{x_1, x_4, x_5\}$, on $\{x_1, x_2, x_5\}$, on $\{x_2, x_3, x_4\}$, and on $\{x_3, x_4, x_5\}$. All variables have the domain $\{1, 2, 3, 4\}$. This problem is SGAC because any instantiation of a variable results in a problem that is GAC. Enforcing NIC, however, shows that the problem is insoluble. Consider a problem with three not-equals constraints, $x_1 \neq x_2$, $x_1 \neq x_3$, $x_2 \neq x_3$ in which each variable has the same domain of size two. This problem is SGAC but enforcing strong PC proves that it is insoluble.

Consider the following 2-colouring problem. We have 5 variables, x_1 to x_5 arranged in a ring. Each variable has the same domain of size 2. Between each pair of neighbouring variables in the binary decomposition, there is a not-equals constraint. In the non-binary representation, we post a single constraint on all 5 variables. This problem is NIC, but enforcing SGAC on the non-binary representation shows that the problem is insoluble. Finally, consider an all-different constraint on 4 variables, each with the same domain of size 3. The binary representation of the problem is strong PC but enforcing SGAC shows that it is insoluble.

5 Random problems

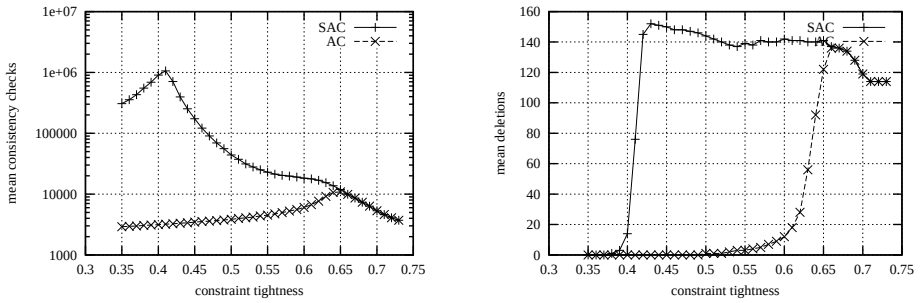
These theoretical results help place singleton consistencies within the hierarchy of local consistencies. But how useful are singleton consistencies in practice? To explore this issue, we ran experiments first with random problems, then with problems that combine structure and randomness, and afterwards with more realistic structured problems. One of our intentions was to determine how well results from random problems predicted behaviour on more realistic problems. Our starting point is [DB97] which reports a set of experiments on random problems with 20 variables and 10 values. These experiments identify how well consistency techniques like SAC approximate global consistency, and give the ratio of the number of values pruned to the CPU times at different points in the phase space. Debruyne and Bessiere conclude that SAC is a very promising local consistency technique, removing most of the strong path-inconsistent values while requiring less time than path inverse consistency.

Debruyne and Bessiere's experiments suffer from two limitations. First, their experiments only measure the ability of singleton arc-consistency to approximate global consistency. They do not tell us if SAC is useful within complete

search procedures like MAC. For instance, does pre-processing with singleton arc-consistency reduce MAC's search enough to justify its cost? Can we afford to maintain SAC within (a number of levels of) search? Second, their experiments were restricted to random binary problems. Do results on random problems predict well behaviour on real problems? What about non-binary problems? Can it pay to enforce the singleton version of non-binary consistencies like GAC? Our experiments tackle both these issues.

5.1 SAC and AC as a pre-process

Mackworth's AC3 algorithm was encoded and used to implement the AC and SAC pre-processes and the domain filtering within the FC and MAC search algorithms. The reason AC3 was chosen is because it allows a standard measure of comparison between algorithms, namely the consistency check. FC was implemented as a crippled version of MAC, i.e. propagation within AC3 was disabled beyond the constraints incident on the current variable.



(a) (b)

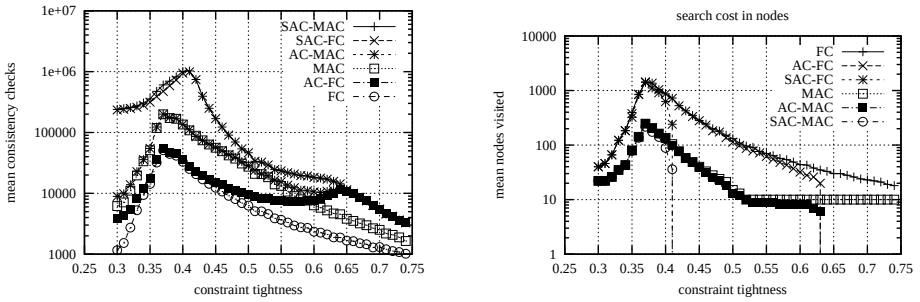
Fig. 1. SAC and AC pre-processing for $\langle 20, 10, 0.5\rangle$: on the left (a) effort measured as mean consistency checks and on the right (b) values deleted

Figure 1a shows the mean pre-processing cost measured in consistency checks for AC and SAC over $\langle 20, 10, 0.5\rangle$ model-B problems with a sample size of 500 (i.e. problems studied by Debruyne and Bessiere) at each value of constraint tightness p_2. Looking at the contours for SAC and AC we see that the two blend together at the arc-consistency phase transition ($p_2 \approx 0.65$). This is expected as the first phase of SAC is to make the problem arc-consistent. If this phase detects arc-inconsistency the problem is also SAC inconsistent and there is no more work to do.

Figure 1b shows the average number of values removed from the problem by pre-processing. Again, we see the SAC and AC contours blend together at the AC phase transition. About 80% of values are deleted in order to show SAC insolubility ($p_2 \approx 0.41$), and about 70% for AC insolubility. The solubility phase transition for this problem is round about $p_2 \approx 0.37$, and we see next to no values

being deleted by SAC until $p_2 \approx 0.38$. This does not bode well for reduction in search effort for this problem.

5.2 Search after SAC



(a) (b)

Fig. 2. Search cost for MAC and FC over $\langle 20, 10, 0.5 \rangle$ with and without pre-processing: on the left (a) effort measured as mean consistency checks and on the right (b) effort measured as nodes visited

Figure 2a shows the total cost of determining if a $\langle 20, 10, 0.5 \rangle$ problem is soluble using the MAC and FC algorithm with various pre-processing steps, both algorithms using the MRV dvo heuristic. Constraint tightness p_2 was incremented in steps of 0.01, and at each value of p_2 100 problems were analyzed. Cost is measured as average consistency checks, which also directly corresponds to cpu times. The cost of SAC pre-processing dominates search cost. SAC-MAC and SAC-FC compare poorly with their AC and null pre-processing equivalents. At the solubility phase transition, $p_2 \approx 0.37$, the average cost of SAC-MAC is 605K checks whereas MAC without any pre-process costs 198K checks. The cost of SAC pre-processing from Figure 1 is 432K checks at $p_2 = 0.37$. This suggests that in these problems SAC is an uneconomical overhead. In fact we see the solubility complexity peak dominated to such a degree that it appears shifted right to the higher value of constrainedness associated with the SAC phase transition. Around the solubility phase transition it was observed that for all algorithms studied soluble problems were easier than insoluble problems. This was most notable for SAC-FC, the reason being that SAC pre-processing frequently detected insolubility, but this was at the cost of deleting many values from variables, changing the problem and this in turn initiates more iterations of the outermost loop of the SAC algorithm. As an aside it should be noted that AC-FC exhibits a twin peaked complexity contour, the second (and lower) peak due to the AC phase transition.

Figure 2b shows cost measured in median nodes visited. SAC pre-processing makes no impact on the size of the search tree explored until it starts to delete

values. As noted in Figure 1a, this does not begin to occur until just after the solubility phase transition. Consequently we see a reduction in nodes visited only as we approach the SAC phase transition, i.e. values of $p_2 > 0.4$.

5.3 Dense problems and large sparse problems

We investigated denser problems and large sparse problems. For the dense $\langle 20, 10, 1.0 \rangle$ problems search costs dominate pre-processing when problems are hard. At the solubility complexity peak $p_2 = 0.21$ the cost of SAC pre-processing was about 680K checks whereas SAC-MAC took 1835K checks, MAC alone took 1163K checks, SAC-FC took 931K checks, and FC alone took 258K checks. Therefore, although SAC pre-processing shows no advantage it is now substantially less effort than the search process on hard problems.

In the sparse $\langle 50, 10, 0.1 \rangle$ problems MAC and FC compete with each other over hard problems. Although the SAC pre-process continues to be uneconomic, it is just beginning to break even. In particular, on 100 (hard) instances of $\langle 50, 10, 0.1, 0.55 \rangle$ of the 26 insoluble instances 22 were detected by the SAC pre-process, and 23 of the 74 soluble instances were discovered without backtracking. In total 43 of the soluble instances took less than 100 search nodes. A study of $\langle 50, 10, 0.2 \rangle$ problems, i.e large but slightly denser, showed that SAC pre-processing was again uneconomical.

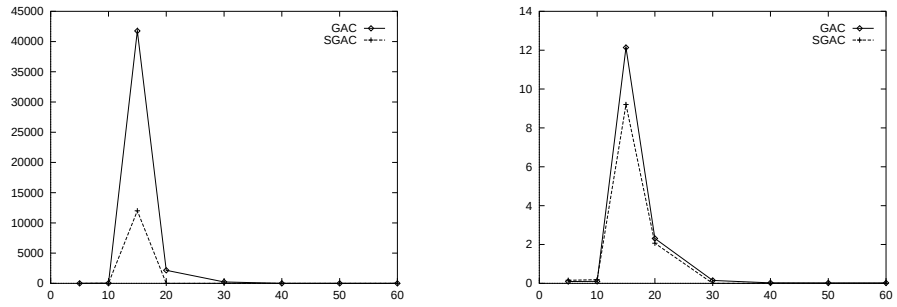
These experiments suggest that SAC pre-processing may be worthwhile on larger sparse problems with tight constraints, but uneconomical on dense problems with relatively loose constraints.

6 Small-world problems

To test the efficiency of singleton consistency techniques on problems with structure, we first studied "small-world" problems. These are problems that combine structure and randomness. In graphs with "small world" topology, nodes are highly clustered, whilst the path length between them is small. Recently, Watts and Strogatz have shown such graphs occur in many biological, social and man-made systems that are often neither completely regular nor completely random [WS98]. Walsh has argued that such a topology can make search problems hard since local decisions quickly propagate globally [Wal99]. To construct graphs with such a topology, we start from the constraint graph of a structured problem like a quasigroup. Note that a quasigroup can be modelled using either all-different constraints for each row and column or cliques of binary "not-equals" constraints. To introduce randomness, we add edges at random in the binary representation. Small world problems created in this way quickly become very hard when the order of the quasigroup is increased.

Figures 3a and 3b show the median number of branches explored and the cpu time used when GAC and SGAC are used for preprocessing small-world problems created by randomly adding edges to an order 6 quasigroup. GAC on the all-different constraints is maintained during search. The x-axis gives the percentage of added edges in the total number of edges left to turn the quasigroup into a

complete graph. 100 problems were generated at each data point. We do not include SAC and AC preprocessing in Figures 3a and 3b because they have no impact as they do no pruning at all. This is not surprising, because of the nature of the constraints. AC on a binary "not-equals" constraint may delete a value from one of the variables only if the other one has a singleton domain. Likewise, when SAC reduces a variable x to a singleton value v then v is removed from the domain of all variables constrained with x, but no more filtering can be made. As a result, there can be no singleton arc-inconsistent values in problems with domain size 6.



(a) (b)

Fig. 3. Search cost for GAC and SGAC. On the left (a) effort measured as branches explored and on the right (b) effort measured as cpu time used (in seconds).

Preprocessing with SGAC is very efficient especially at the solubility complexity peak and in the insoluble region, where insolubility is detected without search for most insoluble instances. SGAC preprocessing is also cost-effective for soluble instances, especially for hard ones near the complexity peak, as it cuts down the number of branches explored significantly. CPU times are also reduced despite the cost of preprocessing. The presence of structure in the problems makes SGAC much more efficient than on purely random problems.

7 Problems of distance

To test singleton consistency techniques on a class of structured, and non-binary problems, we ran experiments on a variety of *problems of distance*. This general class of problems is introduced in [SSW00], and models several challenging combinatorial problems including Golomb rulers and all-interval series. A problem of distances is defined by a graph in which nodes are labelled with integers, the edges are labelled by the difference between the node labels at either end of each edge, and there are constraints that all edge labels are different. As in [SSW00], we model such problems with variables for both the nodes and edges, ternary constraints of the form $d_{ij} = |x_i - x_j|$ that limit the values given to the edges, and a single all-different constraint on the edges.

7.1 Golomb rulers

Peter van Beek has proposed the Golomb ruler problem as a challenging constraint satisfaction problem for the CSPLib benchmark library (available as `prob006` at `http://csplib.cs.strath.ac.uk`). The problem specification given there is:

> "A Golomb ruler may be defined as a set of m integers $0 = x_1 < x_2 < ... < x_m$, such that the $m(m-1)/2$ differences $x_j - x_i$, $1 \leq i < j \leq m$, are distinct. Such a ruler is said to contain m marks and is of length x_m. The objective is to find optimal (minimum length) or near optimal rulers."

Golomb rulers are problems of distance in which the underlying graph is complete. To turn optimization into satisfaction, we build a sequence of decision problems, reducing a_m until the problem becomes unsatisfiable. The longest currently known optimal ruler has 21 marks and length 333. Peter van Beek reports that even quite small problems (with fewer than fifteen marks) are very difficult for complete methods such as backtracking search, and that their difficulty lies both in proving optimality and in finding a solution, since the problems have either a unique solution or just a handful of solutions.

Marks	Branches			CPU time		
	AC	SAC	restricted SAC	AC	SAC	restricted SAC
7-F	10	10	10	0.15	1.27	0.63
7-P	87	65	65	0.20	1.25	0.83
8-F	26	26	26	0.22	2.98	1.564
8-P	506	461	461	1.55	3.52	2.26
9-F	309	282	282	1.28	8.00	4.04
9-P	2489	2318	2318	8.44	13.73	10.30
10-F	1703	1692	1692	6.05	27.45	13.41
10-P	11684	9658	9665	56.18	68.16	54.17
11-F	7007	6584	6584	26.98	87.74	48.72
11-P	202137	193419	193498	1240.90	1170.77	1151.70

Table 1. Branches explored and cpu time in seconds when trying to find a ruler of optimal length (F) or prove that no shorter exists (P). Preprocessing was done with AC, SAC and restricted SAC.

Table 1 shows the search cost in branches and cpu time for algorithms that preprocess with AC, SAC and restricted SAC, and maintain GAC during search. Although preprocessing with SAC deletes considerably more values than preprocessing with AC, this is not reflected in the search effort.

Table 2 shows the search effort for algorithms that preprocess with GAC, SGAC, and restricted SGAC, and maintain GAC on the all-different constraint during search. SGAC deletes a large number of values during preprocessing (more than 60% in some cases) and that has a notable effect on search. The number

Marks	Branches			CPU time		
	GAC	SGAC	restricted SGAC	GAC	SGAC	restricted SGAC
7-F	10	5	6	0.15	1.06	0.58
7-P	87	0	0	0.18	0.34	0.32
8-F	26	22	22	0.21	2.49	1.29
8-P	506	265	339	1.57	3.21	1.55
9-F	309	261	262	1.30	5.14	2.90
9-P	2489	1844	1862	8.64	8.56	5.92
10-F	1703	1592	1592	6.16	14.61	9.15
10-P	11684	7823	7924	56.35	37.65	30.77
11-F	7007	6464	6464	27.04	65.53	37.96
11-P	202137	98967	99602	1239.81	491.58	442.96

Table 2. Branches explored and cpu time in seconds when preprocessing with GAC, SGAC and restricted SGAC.

of explored branches is cut down, especially when trying to prove optimality, and despite the cost of preprocessing, there is a gain in cpu times for the harder instances. Restricted SGAC seems a better option than full SGAC since it deletes almost the same number of values and is more efficient in cpu times.

Given the good results obtained by preprocessing with SGAC, we investigated whether maintaining such a high level of consistency during search is worthwhile. Our results showed that maintaining SGAC even for depth 1 in search (i.e., at the first variable) is too expensive. When trying to find an optimal ruler, we enforced SGAC after instantiating the first variable. As a result, the number of branches was cut down (though not significantly), but runtimes were higher. When trying to prove optimality, we enforced SGAC after each value of the first variable was tried. Again there was a gain in branches, but runtimes were much higher than before.

7.2 2-d Golomb rulers and all-interval series

A Golomb ruler is a problem of distance in which the underlying graph is complete (i.e. a clique). Our results with random problems suggest that singleton consistencies will show more promise on sparser problems. What happens then with problems of distance in which the underlying graph (and hence the associated constraint graph) is sparser? For example, in a 2-d Golomb ruler we have (2 or more) layers of cliques, with edges between node i in clique j and node i in clique $j+1$. A 2-d Golomb ruler with k layers has a constraint graph with approximately $1/k$ the edges of that of a 1-d Golomb ruler of the same size.

Table 3 shows the search effort for algorithms that preprocess with GAC, SGAC, and restricted SGAC, and maintain GAC on the all-different constraint during search. SGAC preprocessing reduces the number of branches, and the cpu times in the harder instances (rulers with 6 marks), but the effect is not as significant as in the 1-d case.

Marks	Branches			CPU time		
	GAC	SGAC	restricted SGAC	GAC	SGAC	restricted SGAC
3-F	1	0	0	0.051	0.120	0.068
3-P	6	0	0	0.048	0.052	0.050
4-F	32	26	26	0.27	0.693	0.407
4-P	210	74	191	0.389	1.228	0.598
5-F	1404	1276	1276	2.552	3.767	3.111
5-P	8177	7521	7521	14.764	14.389	13.554
6-F	133010	113723	113723	376.23	321.553	317.033
6-P	433087	357320	357320	1420.63	1071.82	1067.32

Table 3. Branches explored, and cpu time in seconds when GAC, SGAC and restricted SGAC are used for preprocessing 2-d Golmb rulers.

An even sparser problem of distance is the all-interval series problem. This problem was proposed by Holger Hoos as a challenging constraint satisfaction problem for the CSPLib benchmark library (available as `prob007` at `http://csplib.cs.strath.ac.uk`). All-interval series are problems of distance in which the underlying graph is a simple ring. They therefore have an associated constraint graph which is very sparse compared to 1-d and 2-d Golomb rulers. In the case of all-interval series, preprocessing with SAC and SGAC had no effect as no values were pruned. Also, enforcing SAC (SGAC) at depth 1 had very little impact on the number of branches explored and increased runtimes.

8 Related work

As mentioned briefly before, Debruyne and Bessiere compared the ability of a variety of different local consistencies (e.g. AC, RPC, PIC, SAC, strong PC, and NIC) at approximating global consistency on randomly generated binary problems with 20 variables and 10 values. [DB97]. In addition, they computed the ratio of CPU time to number of values pruned. They concluded that SAC and RPC are both promising, the first having a good CPU time to number of values pruned, and the second requiring little additional CPU time to AC but pruning most of the values of path inverse consistency. Debruyne and Bessiere also studied singleton restricted path-consistency (SRPC) but concluded that it is too expensive despite its ability to prune many values.

Closely related inference techniques have shown promise in the neighbouring field of propositional satisfiability (SAT). One of the best procedures to solve the SAT problem is the Davis-Putnam (DP) procedure [DLL62]. The DP procedure consists of three main rules: the empty rule (which fails and backtracks when an empty clause is generated), the unit propagation rule (which deterministically assigns any unit literal), and the branching or split rule (which non-deterministically assigns a truth value to a chosen variable). The effectiveness of DP is in large part due to the power of unit propagation. Note that the unit propagation rule is effectively the "singleton" empty rule. That is, if we assign the complement of an unit clause, the empty rule will show that the

resulting problem is unsatisfiable; we can therefore delete this assignment. Other "singleton" consistencies (specifically that provided by the "singleton" unit rule) might therefore be of value. Indeed, some of the best current implementations of DP already perform a limited amount of "singleton" unit reasoning, having heuristics that choose between a set of literals to branch upon by the amount of unit propagation that they cause [LA97].

Smith, Stergiou and Walsh performed an extensive theoretical and empirical analysis of the use of auxiliary variables and implied constraints in models of problems of distance [SSW00]. They identified a large number of different models, both binary and non-binary, and compared theoretically the level of consistency achieved by GAC on them. Their experiments on 1-d, 2-d and circular Golomb rulers showed that the introduction of auxiliary variables and implied constraints significantly reduces the size of the search space. For instance, their final models reduced the time to find an optimal 10-mark Golomb ruler 50-fold.

9 Conclusions

We have performed a comprehensive theoretical and empirical study of the benefits of singleton consistencies. For example, we proved that singleton (i, j)-consistency is sandwiched between strong $(i + 1, j)$-consistency and $(i, j + 1)$-consistency. We also proved that, on non-binary constraints, singleton generalized arc-consistency (the singleton extension of generalized arc-consistency) is strictly stronger than both generalized arc-consistency and singleton arc-consistency (on the binary decomposition). Singleton generalized arc-consistency is, however, incomparable to neighbourhood inverse consistency and strong path-consistency (on the binary decomposition). Singleton generalized arc-consistency is a very high level of consistency to achieve. Nevertheless our experiments showed that it can be worthwhile if we have an efficient algorithm (as we do for all-different constraints). We ran experiments on both random and structured problems. On random problems, singleton arc-consistency was rarely cost-effective as a pre-processing technique. However, it did best on sparse problems. Results on problems with structure were quite different. On small-world problems, 1-d and 2-d Golomb rulers, singleton generalized arc-consistency was often cost-effective as a pre-processing technique, especially on large and insoluble problems. Unlike random problems, more benefits were seen on dense problems than on sparse problems. Our experiments also showed that restricting algorithms that enforce singleton consistencies to one pass only gave a small reduction in the amount of pruning.

What general lessons can be learned from this study? First, singleton consistencies can be useful for pre-processing but can be too expensive for maintaining, even during the initial parts of search only. Second, singleton consistencies appear to be most beneficial on large, unsatisfiable and structured problems. Third, limiting algorithms that enforce singleton consistencies to a single pass makes a small dent on their ability to prune values, and can thus improve their cost-effectiveness. Fourth, provided we have an efficient algorithm, it can pay to enforce consistencies as high as singleton generalized arc-consistency. And finally, random problems can be very misleading. Our experiments on random

problems suggested that pre-processing with singleton consistencies was rarely cost-effective and that it was most beneficial on sparse problems. The results of our experiments on structured problems could hardly be more contradictory.

Acknowledgements

The third author is supported by an EPSRC advanced research fellowship. The authors are members of the APES (http://www.cs.strath.ac.uk/~apes) research group and thank the other members for their comments and feedback.

References

[BR97] C. Bessière and J.C. Régin. Arc consistency for general constraint networks: Preliminary results. In *Proceedings IJCAI-97*, pages 398–404, 1997.

[DB97] R. Debruyne and C. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of the 15th IJCAI*, pages 412–417. International Joint Conference on Artificial Intelligence, 1997.

[Dec90] R. Dechter. On the expressiveness of networks with hidden variables. In *Proceedings of the 8th National Conference on AI*, pages 555–562. American Association for Artificial Intelligence, 1990.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[FE96] E. Freuder and C.D. Elfe. Neighborhood inverse consistency preprocessing. In *Proceedings of the 12th National Conference on AI*, pages 202–208. American Association for Artificial Intelligence, 1996.

[Fre85] E. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the Association for Computing Machinery*, 32(4):755–761, 1985.

[Gas79] J. Gaschnig. Performance measurement and analysis of certain search algorithms. Technical report CMU-CS-79-124, Carnegie-Mellon University, 1979. PhD thesis.

[LA97] C.M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th IJCAI*, pages 366–371. International Joint Conference on Artificial Intelligence, 1997.

[MM88] R. Mohr and G. Masini. Good old discrete relaxation. In *Proceedings of the European Conference on Artificial Intelligence (ECAI-88)*, pages 651–656, 1988.

[Reg94] J-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 12th National Conference on AI*, pages 362–367. American Association for Artificial Intelligence, 1994.

[SRGV96] T. Schiex, J.C. Régin, C. Gaspin, and G. Verfaillie. Lazy arc consistency. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-96)*, pages 216–221, Portland, Oregon, 1996.

[SSW00] B. Smith, K. Stergiou, and T. Walsh. Using auxiliary variables and implied constraints to model non-binary problems. In *To appear in Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000)*, Austin, Texas, 2000.

[SW99] K. Stergiou and T. Walsh. The difference all-difference makes. In *Proceedings of 16th IJCAI*. International Joint Conference on Artificial Intelligence, 1999.

[Wal99] T. Walsh. Search in a small world. In *Proceedings of IJCAI-99*, 1999.

[WS98] D.J. Watts and S.H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.