Maintaining Singleton Arc-Consistency

Christophe Lecoutre[1] and Patrick Prosser[2]

[1]CRIL-CNRS FRE 2499,
Université d'Artois
Lens, France
lecoutre@cril.univ-artois.fr

[2]Department of Computing Science
University of Glasgow
Scotland
pat@cs.gla.ac.uk

Abstract. Singleton Arc-Consistency (SAC) [8] is a simple and strong level of consistency but is costly to enforce. To date, research has focused on improving the performance of algorithms that achieve SAC, and comparing algorithms as a preprocessing step before actually solving a problem. Here, we show for the first time how a basic SAC algorithm can be readily incorporated into an open source constraint programming toolkit and then used within the search process i.e. the search process maintains SAC. We also present three new levels of SAC: Bound-SAC where the first and last values in domains are SAC, First-SAC where only the first value is SAC, and Existential-SAC where some value in the domain is SAC. Again, we show how these levels of SAC can be maintained by the search process, and present the first empirical study of their behaviours. This leads us to the point where we can investigate the effect of maintaining different levels of consistency on different sets of variables within a problem. We show experimentally that it can result in significant performance improvements.

1 Introduction

In 1997, Debruyne and Bessière introduced Singleton Arc-Consistency (SAC) [8]. In a constraint network P, a value a in the domain of a variable x is SAC if the variable x can be assigned the value a and P can then be made arc-consistent. P is SAC if all values in the domains of all variables are SAC. This gives a stronger level of consistency than AC but at a substantially higher cost. The complexity of achieving AC is $O(ed^2)$ [17] whereas the optimal cost of SAC is $O(end^3)$ [3], where e is the number of constraints, d is the size of the largest domain, and n is the number of variables. In [2], it was proved that SAC is a non-local property, unlike AC. Consequently, we should expect that it will be non-trivial to achieve practically efficient algorithms for this consistency. There have been three notable attempts at proposing practical algorithms. The first one [1] aims at avoiding some useless singleton checks by recording supports while the two others [3, 13] exploit the incrementality of arc-consistency. However, except for SAC3 [13], all proposed algorithms either require large data structures or are non-trivial to implement.

To date, SAC has been studied only as a preprocessing step prior to actually solving a problem, and has been typically applied to random instances, frequency assignment problems and problems of distance [8, 19, 1, 3, 13]. The study in [19] showed that SAC, as a preprocess, was rarely cost effective on random instances, but on structured problems such as networks with small-world graphs or Golomb rulers, SAC was often beneficial. Therefore it appears, so far, that although SAC may be promising it has not yet

been exploited inside search in the same way AC has [20], and that it has not yet been practically tested[1].

In this paper, we go some way towards putting this right. First, we go back to the basic SAC algorithm proposed in [8] and incorporate it into a constraint programming toolkit. We do this showing the actual code, demonstrating just how easily this can be engineered. This then allows us to investigate, for the first time, the behaviour of SAC inside search whilst exploiting all of the features of the constraint toolkit. That is, we maintain SAC within the search process and compare this to the gold standard of constraint programming, namely MAC [20].[2] We then take a pragmatic approach in our quest for performance improvements and restrict SAC such that only some of the variables in the problem are made SAC, and further, that only some of the values in the domains are SAC.

Therefore, we present three partial forms of SAC. The first is Bound-SAC where the first and last values in the domains of variables are SAC, and all other domain values are arc-consistent. The second level of SAC follows on immediately and we call it First-SAC, where the first value in the domain of a variable is SAC and all other values are AC. Finally we present Existential-SAC ($\exists$-SAC), where we guarantee that some value in the domain is SAC and all others are AC. These different levels of consistency can then be maintained on different sets of variables within a problem. For example when modelling a problem we might maintain SAC on one set of variables, Bound-SAC on another set of variables, and AC on the remaining variables. That is, we might use varying levels of consistency across different parts of a problem, attempting to find a good balance between inference and exploration. It is related to what is called *mixed-consistency* in [10] and *hybrid-consistency* in [4]. We then show how such a feature might be engineered into a solver so that a constraint programmer can control the mix of consistency and we present an empirical study that shows how this can be put to good effect.

The paper is organized as follows. We start by introducing some partial forms of Singleton Arc-Consistency and show their relations. We then show how to incorporate SAC, and its partial forms, into an object-oriented constraint programming toolkit. Next, we present the analysis and results of empirical studies on random, scheduling and Golomb ruler problems. A new algorithm (using a greedy approach) that checks if a constraint network is Existential-SAC is then presented along with an empirical study. Finally we conclude.

---

[1] However, one exception is the Quick Shaving approach of Lhomme [16]. The Quick Shaving principle is to test when backtracking occurs at depth k the consistency of values that were shavable (i.e. singleton arc inconsistent) at depth $k+1$. Filtering is operational (i.e. a feature of the search algorithm) and does not correspond to a property of the constraint network.

[2] In a sense this part of our work is then somewhat in the spirit of [20] where MAC was compared to forward checking. Now we compare maintaining SAC against maintaining AC.

2 Partial Singleton Arc Consistencies

In this section, we introduce some technical background about constraint networks and consistencies. In particular, we introduce three partial forms of Singleton Arc-Consistency called Bound-SAC, First-SAC and Existential-SAC.

A (finite) Constraint Network (CN) P is a pair $(\mathscr{X}, \mathscr{C})$ where $\mathscr{X}$ is a finite set of variables and $\mathscr{C}$ a finite set of constraints. Each variable $X \in \mathscr{X}$ has an associated domain, denoted *dom*(X), which contains the set of values allowed for X. Each constraint $C \in \mathscr{C}$ involves a subset of variables of $\mathscr{X}$, called scope and denoted *vars*(C), and has an associated relation, denoted *rel*(C), which contains the set of tuples allowed for the variables of its scope. We will respectively denote the number of variables and constraints of a CN by n and e. For any variable X, min(X) and max(X) represents the smallest and greatest values in *dom*(X). Note that a value will usually refer to a pair (X,a) with $X \in \mathscr{X}$ and $a \in$ *dom*(X). We will note $(X, a) \in P$ (respectively, $(X, a) \notin P$) iff $X \in \mathscr{X}$ and $a \in$ *dom*(X) (respectively, $a \notin$ *dom*(X)).

A CN is said to be satisfiable iff it admits at least one solution. The Constraint Satisfaction Problem (CSP) is the NP-complete task of determining whether a given CN, also called CSP instance, is satisfiable. To solve a CSP instance, a depth-first search algorithm with backtracking can be applied, where at each step of the search, a variable assignment is performed followed by a filtering process called constraint propagation. Usually, constraint propagation algorithms are based on domain filtering consistencies [9], among which the most widely studied ones are called arc-consistency, max-restricted path consistency and singleton arc-consistency. Arc-Consistency (AC) is the basic property of CNs. It guarantees that each value admits at least one support in each constraint.

Definition 1. *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN. A pair (X, a), with $X \in \mathscr{X}$ and $a \in dom(X)$, is arc consistent (AC) iff $\forall C \in \mathscr{C} \mid X \in vars(C)$, there exists a support of (X, a) in C, i.e., a tuple $t \in rel(C)$ such that $t[X] = a$ and $t[Y] \in dom(Y)$ $\forall Y \in vars(C)$[3]. A variable $X \in \mathscr{X}$ is AC iff dom(X) $\neq \emptyset$ and $\forall a \in dom(X)$, (X, a) is AC. P is AC iff $\forall X \in \mathscr{X}$, X is AC.*

Singleton Arc-Consistency (SAC) is a stronger consistency than AC. It means that SAC can identify more inconsistent values than AC can. SAC guarantees that enforcing arc-consistency after performing any variable assignment does not show unsatisfiability, i.e., does not entail a domain wipe-out. To give a formal definition of SAC, we need to introduce some notations. AC(P) denotes the CN obtained after enforcing arc-consistency on a given CN P. AC(P) is such that all values of P that are not arc consistent have been removed. If there is a variable with an empty domain in AC(P), denoted AC(P) = $\bot$, then P is clearly unsatisfiable. $P|_{X=a}$ is the CN obtained from P by restricting the domain of X to $\{a\}$.

Definition 2. *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN. A pair (X, a), with $X \in \mathscr{X}$ and $a \in dom(X)$, is singleton arc consistent (SAC) iff AC($P|_{X=a}$) $\neq \bot$. X is SAC iff $\forall a \in dom(X)$, (X, a) is SAC. P is SAC iff $\forall X \in \mathscr{X}$, X is SAC.*

[3] $t[X]$ denotes the value assigned to X in t.

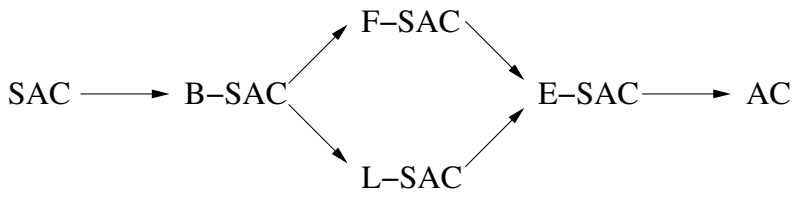

Fig. 1. Relationships between consistencies. $A \rightarrow B$ means consistency A is stronger than B

A consistency ϕ is stronger than a consistency λ iff whenever ϕ holds on a CN, λ holds too. ϕ is strictly stronger than λ iff ϕ is stronger than λ and there exists a CN on which λ holds and ϕ does not hold. It is possible to define partial forms of SAC (i.e. consistencies weaker than SAC) still stronger than AC by restricting SAC to some values.

Definition 3. *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN.*

- *P is First-SAC iff $\forall X \in \mathscr{X}$, X is AC and min(X) is SAC.*
- *P is Last-SAC iff $\forall X \in \mathscr{X}$, X is AC and max(X) is SAC.*
- *P is Bound-SAC iff P is both First-SAC and Last-SAC.*
- *P is Existential-SAC iff $\forall X \in \mathscr{X}$, X is AC and $\exists b \in dom(X)$ s.t. (X,b) is SAC.*

Figure 1 shows the relations existing between the consistencies introduced just above, SAC and AC. An arrow from a consistency ϕ to another consistency λ indicates that ϕ is strictly stronger than λ.

It is natural to conceive algorithms to enforce First-SAC, Last-SAC and Bound-SAC on CNs. Indeed, it suffices to remove all values detected as arc inconsistent and bound values (only the minimal ones for First-SAC and the maximal ones for Last-SAC) detected as singleton arc inconsistent. When enforcing a CN P to be First-SAC, Last-SAC or Bound-SAC, one then obtains the greatest sub-network of P which is First-SAC, Last-SAC or Bound-SAC. As a consequence, if a consistency ϕ is stronger than another consistency λ, then it means that all values removed when enforcing λ on a given network are also removed when enforcing ϕ [9].

In fact, this last statement is true for all (known) consistencies, except for Existential-SAC. Indeed, enforcing Existential-SAC on a CN is meaningless. Either the network is (already) Existential-SAC, or the network is singleton arc inconsistent. It is then better to talk about checking Existential-SAC. An algorithm to check Existential-SAC will have to find a singleton arc consistent value in each domain. As a side-effect, if singleton arc inconsistent values are encountered, they will be, of course, removed. However, we have absolutely no guarantee about the network obtained after checking Existential-SAC due to the non-deterministic nature of this consistency.

3 Maintaining Singleton Arc Consistencies

We now show how SAC can be incorporated into a constraint programming toolkit so that the search process maintains SAC on a specified set of variables. This then leads us

to naturally introduce Bound-SAC and First-SAC. These different levels of consistency (AC, SAC, Bound-SAC, First-SAC) can then be applied selectively across different sets of variables in a problem by the constraint programmer, allowing the programmer to control the blend of mixed-consistency maintained during search. We use the JChoco constraint programming toolkit [12] to demonstrate this.

3.1 Engineering SAC into JChoco

JChoco [12] is a freely available constraint programming toolkit, using the java programming language. Most of the methods used to model and solve problems are called via the $Problem$ class. Constrained variables (be they enumerated, bound, real, or set variables) are added to a problem, and constraints between those variables are posted to it. A problem instance (i.e. an object of class $Problem$) can then be made arc-consistent via the $propagate$ method and solutions found via the $solve$ method. Therefore, in order to incorporate SAC into JChoco we merely produce a new subclass of $Problem$ called $SacProblem$ and over-ride the $propagation$ method. All the $Problem$ methods are inherited and we can then use the constraint toolkit as usual, but rather than maintaining AC, we maintain SAC.

The java code for this is shown below. The boolean method $isSac$ determines if a value a for a variable x is SAC. The method $propagate$ now maintains SAC rather than AC, and the method call $super.propagate()$ is a call to the inherited arc-consistency algorithm used within JChoco. Therefore, if constraints are expressed explicitly as tuples (allowed or disallowed), JChoco will use the optimal algorithm reported in [5], and if a specialised constraint is used, then the appropriate specialised propagator will be applied. The code for the method $propagate$ below should be compared to the procedure $SingletonAC$ given in [8] and the SAC1 procedure in [1]. Our java code is a straightforward translation of these procedures. However, the complexity of this procedure is $O(en^2d^4)$ [19], clearly far worse than the optimal $O(end^3)$ [3].

```
public class SacProblem extends Problem {

     private boolean isSac(IntVar x,int a) throws ContradictionException {
        boolean consistent = true;
        worldPush();
        try{x.setVal(a);super.propagate();}
        catch (ContradictionException e) {consistent = false;}
        worldPop();
        return consistent;
     }

     public void propagate() throws ContradictionException {
        super.propagate();
        boolean change = true;
        while (change) {
           change = false;
           for (int i=0;i<getNbIntVars();i++){
              IntVar x = getIntVar(i);
              IntDomain d = x.getDomain();
              IntIterator domIter = d.getIterator();
              while (domIter.hasNext()){
                 int a = domIter.next();
                 if (!isSac(x,a))
                     {x.remVal(a);change = true;super.propagate();}
              }
           }}}}
```

We believe that SAC can be similarly incorporated in other constraint toolkits that take an object oriented approach (e.g. Koalog's constraint solver [11]). Therefore, we expect that the above engineering approach could be quite generic. However, one obvious limitation of the $SacProblem$ class above is that it will only work on variables with enumerated domains. How can we handle bound integer variables?

3.2 Bound-SAC and First-SAC

In [15], an algorithm is proposed for establishing Bound-SAC. Bound-SAC means that the first and last values in any domain is SAC while all other values are arc consistent. Again we can produce yet another subclass of $Problem$ which we might call $BoundSacProblem$ with a $propagate$ method as shown below.

```
public void propagate() throws ContradictionException {
   super.propagate();
   boolean change = true;
   while (change) {
      change = false;
      for (int i=0;i<getNbIntVars();i++){
         IntVar x = getIntVar(i);
         if (x.getDomainSize()>1){
            while (!isSac(x,x.getInf()))
               {x.remVal(x.getInf());change = true;super.propagate();}
            while (!isSac(x,x.getSup()))
               {x.remVal(x.getSup());change = true;super.propagate();}
         }
      }}}
```

The method call $x.getInf()$ above gets the lower bound of x and $x.getSup()$ gets the upper bound of x. The inner $while$ loops find respectively the smallest and largest SAC values in the domain of x. In addition we can decide to only make the first value in the domain SAC, and we call this First-SAC. To engineer First-SAC all that need be done is to delete the second inner $while$ loop in the code above.

3.3 Mixed-Consistency

If we adopt the implementations above we are then in the position that we can either model problems with enumerated domains or bound domains, but not both. An obvious engineering fix is to be able to detect inside the $propagate$ method the class of variable and then either apply AC, SAC, Bound-SAC or First-SAC. Another approach is to associate three lists with our $SacProblem$: one for enumerated variables to be made SAC, another for enumerated and bound variables to be made Bound-SAC, and a third for enumerated and bound variables to be made First-SAC. Any other variables will be made arc-consistent due to the default call to the AC propagator via $super.propagate()$. This is what we in fact do (but don't show), and have an additional method such that we explicitly add to a $SacProblem$ the variables to be made SAC, Bound-SAC, and First-SAC. This then allows the programmer to blend the level of mixed-consistency across variables in a problem.

4 Empirical Studies

We now present experimental studies showing the effects of maintaining different levels of SAC during search. First, we investigate problems with no structure (random problems), then we look at problems with obvious structure and demonstrate the effect of blending mixed-consistency.

4.1 A Study of Maintaining Levels of SAC on Random Problems

First, we study the effect of maintaining SAC during search on random instances of the class $\langle 20, 10, 0.5 \rangle$ (i.e. problems with 20 variables, each with domain size 10, with a probability of 0.5 that there is a constraint between a pair of variables), answering the decision problem "Is there a solution?". Our problems are modelled in JChoco using enumerated integer variables, constraints represented as allowed tuples, and arc-consistency achieved via the optimal coarse grained algorithm proposed in [5]. We compare MAC against different restrictions of SAC. We realise MAC within our framework as a problem with no SAC or Bound-SAC variables. Consequently, in the $propagate$ method only the call $super.propagate()$ is made. The next experiment is of SAC, where all the values in the domains of variables are maintained singleton arc-consistent. In our third experiment all the variables are maintained Bound-SAC. Finally, we maintain First-SAC on all variables.

Experiments were performed on a domestic machine with a 2.79 GHz processor, with 512 MB RAM, and Windows XP. Measurements were taken of average runtime in milliseconds and the number of nodes explored. We could not measure consistency checks, as is the norm, as these are not available within the JChoco toolkit. However, we consider run times to be as reliable and meaningful a measure as consistency checks.

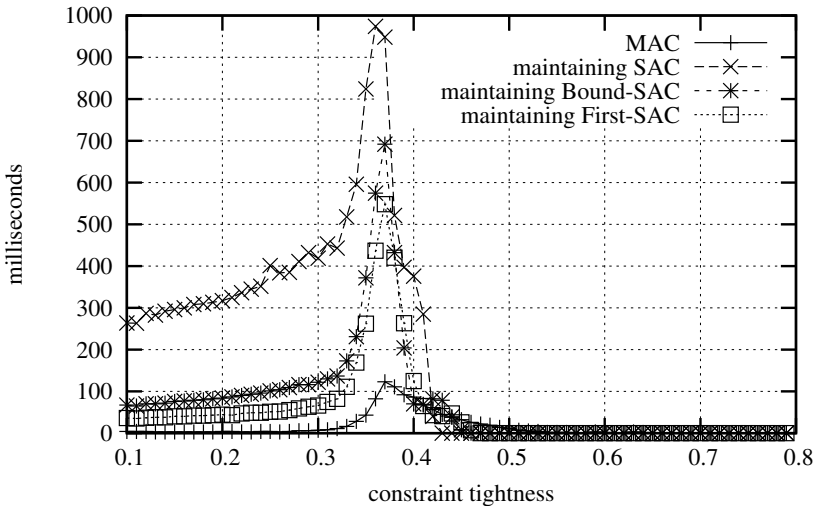

Fig. 2. Average CPU time in milliseconds for $\langle 20, 10, 0.5 \rangle$

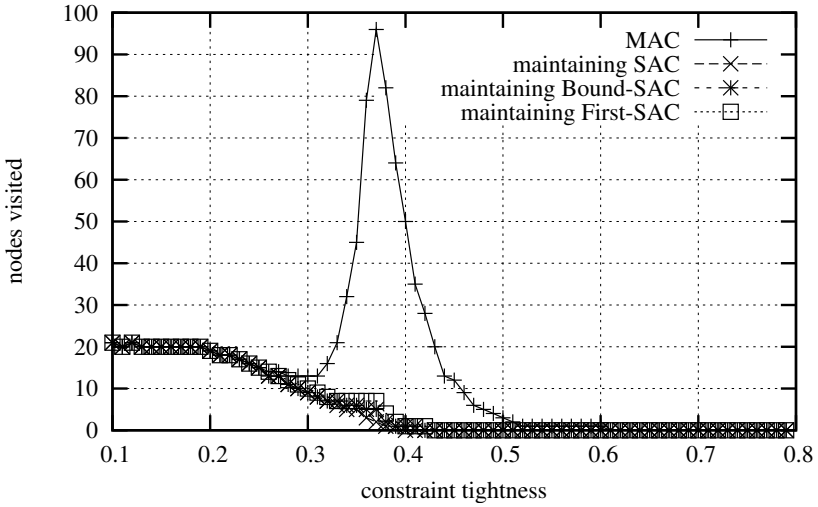

Fig. 3. Average number of nodes visited for $\langle 20, 10, 0.5 \rangle$

In Figure 2, we show the average run time in milliseconds to answer the decision problem, with a sample size of 100. We see the familiar complexity peak at the phase transition, with MAC outperforming the other consistency levels in all but the easy insoluble region ($p_2 > 0.45$). However, we do see that in the easy soluble region ($p_2 < 0.3$) Bound-SAC and First-SAC perform quite well, at least when compared to SAC. In previous studies, it was shown that SAC was a very expensive preprocess in the easy soluble region, and now we see quite acceptable costs for Bound and First-SAC, at least when applied to the decision problem.

Figure 3 shows the average number of nodes visited. We see that there is very little difference in the number of nodes visited between our three versions of SAC. MAC again shows typical phase transition behaviour, with a complexity peak at the crossover point. However the SAC algorithms do not show this behaviour, but instead a gradual fall in nodes visited as we increase constraint tightness. Therefore, we do continue to see a complexity peak in terms of runtime, but this takes place within the SAC algorithms, i.e. it takes longer to reach the SAC fixed point as we approach the phase transition.

This raises an interesting question: if we do not see a complexity peak in the size of the search tree, and the cost of SAC is polynomial, will we actually fail to see a complexity peak as problems get larger? Put another way, will search cost scale polynomially at the phase transition? Of course, our intuition suggests that the answer would be no, and that nodes would rise again. However, to answer (at least, partially) this question, we have investigated the problem classes $\langle 20, 20, 0.5 \rangle$ and $\langle 50, 10, 0.1 \rangle$ using JChoco and abscon [14]. The results were similar to those for $\langle 20, 10, 0.5 \rangle$ with MAC being dominant in runtime. However in the class $\langle 20, 20, 0.5 \rangle$ a small but noticeable complexity peak in nodes visited begins to emerge whilst maintaining SAC.

4.2 A Study of Mixed-Consistency Applied to Scheduling Problems

We performed experiments on 15 of the Lawrence Job-Shop scheduling instances, la01 to la15, available at ORLIB. The instances la01 to la05 are 10×5 (i.e. 10 jobs and 5 resources), la06 to la10 are 15×5, and la11 to la15 are 20×5. Experiments were performed to determine if any particular blend of SAC was beneficial with respect to the quality of solution found when CPU time was bounded. Experiments were run on a 1.3 GHz machine with 256 MB RAM. CPU time was limited to 600 seconds (10 minutes) on each instance. The scheduling problems were represented conventionally, as a disjunctive graph. That is, for a job-shop instance with n jobs and m resources there would be $m.n(n-1)/2$ zero/one variables to decide the order of operations on resources and $(n.m+1)$ bound integer variables to represent operation start times along with the optimisation variable. Therefore we have two distinct sets of variables: the set of 0/1 variables that control disjunctive precedence constraints on resources and the set of start times attached to operations. Consequently, this is a good model to explore the effects of mixed-consistency, i.e. we can maintain different levels of consistency across different sets of variables. The problem specific objective is to find the schedule that minimises the makespan. The results of four of our experiments are shown in Table 1.

The first experiment used MAC (all variables were maintained arc-consistent) and is tabulated as column MAC. Again, MAC was realised by using our SAC solver but with an empty list of variables to make SAC. The second experiment used Bound-SAC on the 0/1 decision variables (it then corresponds to use SAC) and MAC on all other variables, and this is column B-SAC$_{dn}$. Experiment three maintains Bound-SAC on the start times of operations and the optimisation variable, and this is column B-SAC$_{st}$. Finally, in experiment four, all variables are made Bound-SAC, and this is column B-SAC. In all the experiments, the search variables were the 0/1 decision variables. In Table 1, we report the cost of the best solution found within the CPU time limit, and a entry of $-$ signifies that no solution was found in the time limit.

What we see is that Bound-SAC can indeed be beneficial, allowing us to frequently find better solutions than just using MAC on its own. In particular, the B-SAC$_{dn}$ results show that more often than not Bound-SAC on the decision variables alone results in significantly lower makespans than does MAC. However, too much SAC appears to be a bad thing. In experiments B-SAC$_{st}$ and B-SAC we see that too much time is spent in SAC processing compared to time spent in search. Consequently solution quality suffers. In fact, as instance size increases from la11 onwards no solutions were found as all the CPU time was spent in SAC and none in search.

These experiments have demonstrated that a small amount of SAC can be a good thing. But this raises the question: why? In experiment B-SAC$_{dn}$, we maintain Bound-SAC on the 0/1 decision variables. This might be thought of as a weak form of edge-finding [7], i.e. attempting to determine what operations must come first or last on a resource. In experiment B-SAC$_{st}$ we maintain Bound-SAC on the start times of operations, and this in turn is similar to shaving [18]. And finally, in experiment B-SAC we are maintaining weak edge-finding and shaving, but at the expense of reduced exploration.

		Maintaining		
Instance	MAC	B-SAC$_{dn}$	B-SAC$_{st}$	B-SAC
la01	666	666	666	666
la02	655	655	655	655
la03	653	597	603	603
la04	628	598	590	590
la05	593	665	665	665
la06	1245	1146	1233	1237
la07	1214	897	1336	1359
la08	1161	1084	1400	1393
la09	1498	1049	1527	1520
la10	1658	972	1192	1259
la11	1453	1787	–	–
la12	1467	1504	–	–
la13	2899	2310	–	–
la14	1970	1784	–	–
la15	2368	2200	–	–

Table 1. Cost of best solution found for Lawrence scheduling instances, given 10 minutes CPU.

4.3 A Study of Mixed-Consistency on Golomb Rulers

In [19], experiments were performed on Golomb rulers. In particular, given the length l of the shortest ruler with n ticks (or marks), the objective is to find that ruler and prove it optimal. The study showed that SAC preprocessing and restricted SAC preprocessing could lead to a modest reduction in run-times. We repeat those experiments, but now maintain a mix of SAC during the search process.

The problem was represented in JChoco using n $tick$ variables with enumerated domains whose values range from 0 to l, and, in addition $n(n-1)/2$ $diff$ variables with similar domains. Constraints posted to the problem are: $diff[i][j] = tick[j] - tick[i]$ and $tick[i] < tick[j]$ for any pair (i,j) such that $1 \leq i < j \leq n$, and a $boundAllDiff$ constraint enforcing all the $diff$ variables to be different. The $tick$ variables are the decision variables and these were instantiated in a static lexicographic order. Again we have a problem with two obviously different sets of variables, the $tick$ variables and the $diff$ variables, and this again gives us an opportunity to investigate the effects of blending mixed-consistency. Our experiments were run on a 3.01 GHz processor with 512 MB of RAM using Windows XP.

The results of the experiments are given in Table 2 which clearly shows that maintaining Bound-SAC on the $tick$ variables (denoted B-SAC$_{tk}$) dominates MAC, whereas maintaining SAC on the $tick$ variables (denoted SAC$_{tk}$) is far too expensive. We also experimented with maintaining restricted Bound-SAC on the $tick$ variables (denoted RB-SAC$_{tk}$), i.e. the $propagate$ method for Bound-SAC was edited such that the outer $while(change)...$ loop was deleted, consequently only a single pass is made over the variables. This is the same as the restriction proposed in [19]. Table 2 shows that this results in our best performance.

Although not tabulated, we also investigated maintaining SAC, Bound-SAC, and First-SAC on all the variables ($tick$'s and $diff$'s) but run-times did not compete with

Instance	MAC	Maintaining		
		SAC$_{tk}$	B-SAC$_{tk}$	RB-SAC$_{tk}$
5/11	0.01 (5)	0.12 (3)	0.08 (3)	0.08 (3)
6/17	0.1 (18)	0.27 (5)	0.14 (5)	0.14 (5)
7/25	0.47 (116)	0.81 (6)	0.30 (7)	0.34 (11)
8/34	3.6 (904)	14.6 (19)	1.8 (23)	1.6 (33)
9/44	29.1 (5502)	136 (62)	11.3 (68)	9.6 (103)
10/55	217.3 (30097)	1075 (218)	68.8 (245)	59 (479)
11/72	7200 (773560)	—	5534 (11742)	4645 (20056)

Table 2. The runtime in seconds (and in brackets number of nodes visited) to find and prove optimal a Golomb ruler n/l, with n ticks of length l. Restricted Bound-SAC on the $tick$ variables (RB-SAC$_{tk}$) is fastest.

MAC over all instances. Also, First-SAC on the $tick$ variables was competitive with MAC except on the largest problem $11/72$ taking 12371 seconds and 41334 nodes, much slower than MAC.

Some of the experiments were also repeated but using a different model, i.e. we replaced the $boundAllDiff$ constraint with a clique of not-equals constraints. In this model MAC was dominant, typically running three times or more faster than Bound-SAC on the $tick$ variables. Similar behaviour was noted over the quasigroup completion problems in [19]. This is due to the weak propagation of the not-equals constraint, and that values will tend to be SAC until the domain of an adjacent variable is reduced to a singleton. Therefore, we see that when maintaining SAC, we not only have to consider the level of SAC to maintain and the variables over which to maintain that level, but also the model itself.

5 Checking Existential SAC

Existential-SAC is the weakest (see Figure 1) partial form of SAC that we have introduced. We now propose an algorithm to check existential-SAC and we present some empirical results.

5.1 ∃-SAC3

We have presented limited forms of SAC on the basis of the most simple algorithm, SAC1 [8]. SAC1 checks the singleton arc-consistency of all variables whenever a singleton arc-inconsistent value is detected and removed. Assuming an underlying optimal arc-consistency algorithm, worst-case space and time complexities of SAC1 are respectively O(ed) and O(en^2d^4).

In [13], an original approach to establish SAC has been proposed. The principle of this is to perform several runs of a greedy search, where at each step arc-consistency is maintained. As a result, the incrementality of arc-consistency algorithms is exploited but in a different manner to that proposed in [3]. Unfortunately, a bound-SAC version of this approach does not seem to be feasible. Indeed, the main goal is to build

Algorithm 1 buildBranch()

1: $branchSize \leftarrow 0$
2: $P_{before} \leftarrow P$
3: **repeat**
4: pick and remove X from Q
5: select a value $a \in dom(X)$
6: $P \leftarrow$ AC($P|_{X=a}$,$\{X\}$)
7: **if** $P = \bot$ **then**
8: add X to Q
9: **else**
10: $branchSize \leftarrow branchSize + 1$
11: **end if**
12: **until** $P = \bot \vee Q = \emptyset$
13: $P \leftarrow P_{before}$
14: **if** $branchSize$ = 0 **then**
15: remove a from dom(X)
16: $P \leftarrow$ AC(P,$\{X\}$)
17: $Q \leftarrow \{X \mid X \in \mathscr{X}\}$
18: **end if**

Algorithm 2 E-SAC-3($P = (\mathscr{X}, \mathscr{C})$: CN)

1: $P \leftarrow$ AC(P,$\mathscr{X}$)
2: $Q \leftarrow \{X \mid X \in \mathscr{X}\}$
3: **while** $P \neq \bot \wedge Q \neq \emptyset$ **do**
4: buildBranch()
5: **end while**

branches (corresponding to greedy runs) as long as possible in order to benefit from incrementality, and potentially to find solutions during inference. When we are exclusively maintaining Bound-SAC via this approach the resultant propagation branches tend to be short, and therefore uneconomical. However, using a greedy approach to check Existential-SAC seems to be quite appropriate. In particular, it is straight forward to adapt the algorithm SAC3 [13] to guarantee $\exists$-SAC. As mentioned in Section 2, such an algorithm can generate different constraint networks depending on the order that variables and values are considered i.e. it might have multiple fixed points.

Below, we give the description of this new algorithm, denoted $\exists$-SAC3. It is given in the context of using an underlying coarse-grained arc-consistency algorithm such as AC2001/3.1 [5]. But first, we introduce some notations. If $P = (\mathscr{X}, \mathscr{C})$, then AC($P$,$Q$) with $Q \subseteq \mathscr{X}$ means enforcing arc-consistency on P from the given propagation set Q. For a description of AC, see, for instance, the function $propagateAC$ in [3]. Q is the set of variables whose existential consistency must be checked. Finally, an instruction of the form $P_{before} \leftarrow P$ should not be systematically considered as a duplication of the problem. Most of the time, it correspond to store or restore the domain of a network (and the structures of the underlying arc-consistency algorithm)

Algorithm 2 starts by enforcing arc-consistency on the given network (line 1). Then, all variables are put in the structure Q (line 2) and in order to check Existential-SAC,

successive branches are built (line 4). The process continues until Existential-SAC is checked, or singleton arc inconsistency detected (line 3). Algorithm 1 allows building a branch by performing successive variable assignments while maintaining arc-consistency (line 4 to 6). When an inconsistency is detected or the set Q becomes empty, the greedy run is stopped (line 12). If the branch is of size 0 (line 14), we have to manage the removal of a value (since it is singleton arc inconsistent), to reestablish arc-consistency and to restart checking Existential-SAC from scratch (line 17).

Space required specially by ∃-SAC3 is O(n) since the only structure introduced is Q which is O(n). The time complexity of ∃-SAC3 is that of SAC3, that is to say O(bed^2) where b denotes the number of branches built by the algorithm (using an optimal AC algorithm such as AC2001, each branch built is O(ed^2) due to the incrementality of AC2001). In the best case, only one branch will be built (leading then directly to a solution), and then we obtain O(ed^2). In the worst-case, before detecting a singleton arc inconsistent value, $n-1$ branches of size 1 can be built. As the number of values that can be removed is O(nd), we obtain O(en^2d^3). Finally, when no inconsistent value is detected, the worst-case time complexity of ∃-SAC3 is O(end^2).

5.2 Experimental Results

We believe that it is worth studying the effect of maintaining ∃-SAC on satisfiable instances using ∃-SAC3, as due to greedy runs solutions can be found at any step of the search. This is illustrated in Table 3 with some instances of the n-queens problem (we only searched the first solution). These instances were modelled (with binary constraints) in abscon [14] and run on a PC Pentium IV 2.4GHz 512MB RAM under Linux. AC2001 was used as the underlying AC algorithm and dom/wdeg [6] as the variable ordering heuristic. We also show results for forward checking (FC), maintaining arc-consistency (MAC), first-SAC (F-SAC), bound-SAC (B-SAC), and SAC maintained using the SAC1 algorithm. It is interesting to note that for all these satisfiable instances, maintaining SAC3 or ∃-SAC3 explore no more than 2 nodes. However, one should expect to find less impressive results with unsatisfiable instances. To check this, we have tested, using abscon, some difficult (modified) unsatisfiable instances of the Radio Link Frequency Assignment Problem that came from the CELAR (Centre electronique de l'armement). Here, we do not consider optimisation, but only satisfaction. These instances were used as benchmarks for the first CSP solver competition and can be downloaded at `http://cpai.ucc.ie/05/Benchmarks.html`. In Table 3, it appears that maintaining SAC3 or ∃-SAC3 really limits the number of nodes that have to be visited. It can be explained by the fact that both algorithms learn from failures (of greedy runs) as the employed heuristic is $dom/wdeg$.

We then compared maintaining ∃-SAC to MAC on the full set of 1064 instances in the benchmark suite. When counting the number of solved instances within 10 minutes, MAC outperforms ∃-SAC3 by 60 instances when using the $dom/wdeg$ heuristic and by only 23 instances with the dom heuristic. However, regardless of heuristics, ∃-SAC3 behaves relatively poorly on random problems with MAC dominating on the majority of instances in series frb (random instances forced to be satisfiable) and $random$-$\{23, 24, 25\}$. Interestingly, MAC and ∃-SAC3 behave quite differently on different series. One example is all the instances in $series5$ to $series40$ where ∃-SAC dominates.

Instance	FC	MAC	Maintaining				
			F-SAC	B-SAC	SAC1	SAC3	∃-SAC3
100-queens (sat)	0.5 (194)	4.2 (118)	267 (101)	421 (101)	–	17.4 (0)	18.9 (2)
110-queens (sat)	–	–	–	–	–	37.9 (0)	22.7 (1)
120-queens (sat)	–	1636 (323K)	–	–	–	16.7 (0)	47.3 (2)
scen11-f12 (unsat)	69.1 (18K)	3.6 (695)	63.3 (60)	110 (48)	1072 (41)	418 (5)	48.3 (30)
scen11-f10 (unsat)	131 (34K)	4.4 (862)	84.4 (70)	140 (55)	1732 (52)	814 (8)	38.3 (25)
scen11-f8 (unsat)	260 (66K)	67.8 (14K)	1660 (2K)	–	–	–	290 (213)

Table 3. CPU time (and number of visited nodes) for instances of the n-queens and the RLFAP, given 30 minutes CPU.

6 Conclusion

We have taken what is probably an unusual step, reporting on how we can engineer the least efficient version of the SAC algorithm into an actual constraint programming toolkit. By doing this we have been able to perform the first investigation of the behaviour of maintaining SAC within the search process. This has led us to proposing three new levels of SAC, i.e. Bound-SAC, First-SAC and ∃-SAC. We have also allowed ourselves to specify the set of variables in a problem that we make full, bound or first SAC, i.e. we have shown how the programmer can produce a blend of mixed-consistencies and we have shown empirically the effect this can have on runtime performance. We have shown that maintaining the right blend of consistencies can result in significant performance improvements.

When maintaining SAC in small random problems we see a peculiar signature when measuring the size of the search tree (nodes) as we pass through the phase transition. All of our SAC algorithms do not show a complexity peak. We also note that the size of the search tree is relatively insensitive to the amount of restriction put upon SAC, and that when problems are easy and soluble First-SAC and Bound-SAC perform remarkably well. This is one area where earlier studies have shown that SAC is nothing but an expense. For larger random problems, our preliminary study suggests that the size of the search tree should again exhibit a complexity peak, provided that the size of the problems is sufficiently large.

In the job-shop scheduling problem, restrictions on SAC have been beneficial, leading us to better solutions than MAC when CPU time is limited. One explanation is that our restrictions allow us to emulate a weak form of edge-finding or shaving, and that we can combine both of these. However, this has to be used with caution; we need to consider just what variables will benefit from SAC (and that was the 0/1 decision variables).

The Golomb ruler experiments show that we also need to take into consideration how we model the problem. In a model with weakly propagating constraints, values will tend to be SAC and SAC processing will tend to be nothing but an expense. However, with a good model and a well chosen level of SAC (Bound-SAC on the decision variables) we were able to outperform the gold standard, a MAC solver using state of the art constraint propagation algorithms.

The results are surprising when we consider that when using a basic sub-optimal algorithm for SAC we can frequently beat MAC. From an abstract point of view, we have demonstrated that rather than using the same level of inference (maintaining arc-consistency) all the time (during search) everywhere (over all the variables) we can often do much better by varying the level of inference (mixing the consistency levels AC, SAC, Bound-SAC, First-SAC) and doing this over only parts of the problem (a subset of the variables). Finally, we have introduced a simple and efficient implementation of an algorithm that checks $\exists$-SAC. Empirical results suggests that this represents a promising generic approach.

References

1. R. Bartak and R. Erben. A new algorithm for Singleton Arc Consistency. In *Proceedings of FLAIRS'04*, 2004.
2. C. Bessiere and R. Debruyne. Theoretical analysis of Singleton Arc Consistency. In *Proceedings of MSPC'04 workshop held with ECAI'04*, pages 20–29, 2004.
3. C. Bessiere and R. Debruyne. Optimal and suboptimal Singleton Arc Consistency algorithms. In *Proceedings of IJCAI'05*, pages 54–59, 2005.
4. C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. The range and roots constraints: Specifying counting and occurrence problems. In *Proc. of IJCAI'05*, pages 60–65, 2005.
5. C. Bessiere, J.C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained Arc Consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.
6. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.
7. J. Carlier and E. Pinson. A practical use of Jackson's preemptive schedule for solving the Job-Shop problem. *Annals of Operations Research*, 26:269–287, 1990.
8. R. Debruyne and C. Bessiere. Some practicable filtering techniques for the Constraint Satisfaction Problem. In *Proceedings of IJCAI'97*, pages 412–417, 1997.
9. R. Debruyne and C. Bessiere. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.
10. G. Dooms, Y. Deville, and P. Dupont. CP(Graph): Introducing a Graph Computation Domain in Constraint Programming. In *Proceedings of CP'005*, pages 211–225, 2005.
11. Y. Georget and M. Philip. The Koalog Constraint Solver. `http://www.koalog.com`.
12. F. Laburthe and N. Jussien. Jchoco: A java library for constraint satisfaction problems. `http://choco.sourceforge.net`.
13. C. Lecoutre and S. Cardon. A greedy approach to establish Singleton Arc Consistency. In *Proceedings of IJCAI'05*, pages 199–204, 2005.
14. C. Lecoutre and F. Hemery. Abscon 2006. `http://www.cril.univ-artois.fr/~lecoutre`.
15. C. Lecoutre and J. Vion. Bound consistencies for the discrete CSP. In *Proceedings of CPAI'05 workshop held with CP'05*, pages 17–31, 2005.
16. O. Lhomme. Quick shaving. In *Proceedings of AAAI'05*, pages 411–415, 2005.
17. A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.
18. P. Martin and D.B. Shmoys. A new approach to computing optimal schedules for the Job-Shop scheduling problem. In *Proceedings of IPCO'96*, pages 389–403, 1996.
19. P. Prosser, K. Stergiou, and T. Walsh. Singleton Consistencies. In *Proceedings of CP'00*, pages 353–368, 2000.
20. D. Sabin and E. Freuder. Contradicting conventional wisdom in Constraint Satisfaction. In *Proceedings of ECAI'94*, pages 125–129, 1994.