A Study of Encodings of Constraint Satisfaction Problems with 0/1 Variables*

Patrick Prosser and Evgeny Selensky

Department of Computing Science, University of Glasgow, Scotland.
pat/evgeny@dcs.gla.ac.uk

Abstract. Many constraint satisfaction problems (csp's) are formulated with 0/1 variables. Sometimes this is a natural encoding, sometimes it is as a result of a reformulation of the problem, other times 0/1 variables make up only a part of the problem. Frequently we have constraints that restrict the sum of the values of variables. This can be encoded as a simple summation of the variables. However, since variables can only take 0/1 values we can also use an occurrence constraint, e.g. the number of occurrences of 1 must be k. Would this make a difference? Similarly, problems may use channelling constraints and encode these as a biconditional such as $P \leftrightarrow Q$ (i.e. P if and only if Q). This can also be encoded in a number of ways. Might this make a difference as well? We attempt to answer these questions, using a variety of problems and two constraint programming toolkits. We show that even minor changes to the formulation of a constraint can have a profound effect on the run time of a constraint program and that these effects are not consistent across constraint programming toolkits. This leads us to a cautionary note for constraint programmers: take note of how you encode constraints, and don't assume computational behaviour is toolkit independent.

1 Introduction

A constraint satisfaction problem (csp) is composed of a set of variables, each with a domain of values. Constraints restrict combinations of variable assignments. The problem is to find an assignment of values to variables that satisfies the constraints, or show that none exists [9]. There are many real world instances of csp's, such as scheduling, timetabling, routing problems, frequency assignment, design problems, etc. Since many of these problems are commercially important, we now have toolkits that allow us to express these problems as csp's.

Even when we have decided upon a formulation of a csp, we are then faced with choice of how we implement the constraints using the toolkit provided. What we investigate here is how the implementation of the constraints can influence the execution time of our constraint programs. We limit our study to problems with variables that can only take the values zero or one, i.e. 0/1 variables, and to two constraints: a restriction on the sum of the variables, and

* This work was supported by EPSRC research grant GR/M90641 and ILOG SA.

the biconditional constraint. We use three different problems as vehicles for this study. The first problem is the *independent set* of a hypergraph. The second problem is closely related, the *maximal independent set* of a hypergraph. Both of these problems have a natural encoding using 0/1 variables. The third problem is the balanced incomplete block design problem (bibd), and is again naturally formulated using 0/1 variables. For each of these problems we encode the constraints in a number of different ways and measure the run time to find the first solution. We use two constraint programming toolkits, Ilog Solver 5.0 [4] and Choco 1.07 [1].

In the next section we introduce the three problems, independent set, maximal independent set, and balanced incomplete block design. We also present a proof that our various encodings achieve the same level of consistency. Section 3 gives the results of our empirical study. We then imagine a study based on the encodings we have studied and show how this can lead us to a contradiction. We then present an explanation of the sensitivity of performance with respect to the implementation of our constraints. Section 5 concludes this paper.

2 Three Problems

We now present the three problems we will investigate, and their various encodings. The first problem is *independent set* and we use this as a vehicle to examine the implementation of a constraint that restricts the sum of variables. The second problem is *maximal independent set* and we use this to explore how we can implement the biconditional. The third problem, balanced incomplete block designs uses both constraints, summation and biconditional.

2.1 Independent Set

Given a graph $G = (V, E)$, where V is a set of vertices and E is a set of edges, an independent set I is a subset of V such that no two vertices in I share an edge in E. Independent set is one of the first NP-complete problems. A variant of it, GT20 in Garey and Johnson [2], asks if there is an independent set of size k or larger. For a hypergraph $H = (V, E)$ each edge in E is a subset of the vertices in V, and an independent set I of H is then a subset of V such that no edge $e \in E$ is subsumed by I. For example we might have a hypergraph H of 9 vertices, v_1 to v_9, and 4 hyper edges $\{(v_1, v_2, v_3), (v_2, v_4, v_5), (v_4, v_6), (v_3, v_7, v_8, v_9)\}$. This is shown in Figure 1. An independent set of size 7 is then $I = \{v_1, v_2, v_5, v_6, v_7, v_8, v_9\}$.

We can formulate this problem as a csp, such that each vertex v_i corresponds to a 0/1 variable x_i, and if $x_i = 1$ then v_i is in the independent set I. The *independence* constraint restricts the sum of the variables/vertices in a hyperedge to be less than the arity of that hyperedge, where arity is the number of variables/vertices involved in that hyperedge. There is an independence constraint for each hyperedge. Finally we have the constraint that the sum of all the variables has to be greater than or equal to k, i.e. the size of the independent set.

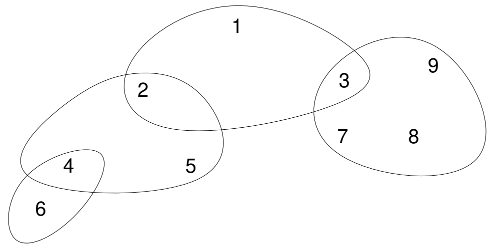

Fig. 1. Our example hypergraph, $H = (V, E)$ where $E = \{(v_1, v_2, v_3), (v_2, v_4, v_5), (v_4, v_6), (v_3, v_7, v_8, v_9)\}$. An independent set of size 7 is then $I = \{v_1, v_2, v_5, v_6, v_7, v_8, v_9\}$

The problem of finding an independent set of a given size is of particular interest. This is because any constraint satisfaction problem, with discrete and finite domains, can be formulated as the problem of finding an independent set [11]. Assume that we have a csp P with n variables x_1 to x_n, each with a domain d_i of size m_i. In its hypergraph representation H we have $\sum_{i=1}^{n} m_i$ variables each with a domain $\{0, 1\}$. Variable $z_{i,a} = 1$ corresponds to the instantiation $x_i = a$ in P, and $z_{i,a} = 0$ corresponds to $x_i \neq a$ in P. The domain of each variable x_i in P is represented as a clique K_m in H, such that there are edges $(z_{i,a}, z_{i,b})$ for all $a, b \in d_i$. The constraints in the original problem P are represented as hyperedges in H. For example, a nogood $(x_i = a, x_j = b, ..., x_r = j)$ in P has the corresponding hyperedge $(z_{i,a}, z_{j,b}, ..., z_{r,j})$ in H. Finding a solution to P corresponds to finding an independent set of size n in H.

Therefore, given a hyperedge e, where $vert(e)$ is the set of vertices involved in e and $arity(e)$ is the number of vertices involved in e, we have the corresponding independence constraint $\sum_{x_i \in vars(c)} x_i < arity(c)$, where each of the variables x_i in constraint c corresponds to the vertices v_i in hyperedge e.

We consider encoding the independence constraint in two ways. First, and most obviously, we perform arithmetic on the values of the variables in the hyperedge and restrict them to be less than the arity. In Choco this would be done as $sumVars(x) <= length(x)$ where x is a list of integer variables (and in Ilog Solver $IloSum(x) <= x.getSize()$, where x is an array of integer variables). Alternatively, since variables are constrained to take $0/1$ values, we can state that the number of occurrences of the value 1 must be less than or equal to k. Again in Choco, we express this as $occur(1, x) <= length(x)$, and in Solver we use the *IloDistribute* function. We call the first encoding $ind1$ and the second $ind2$. We now prove that these two encodings achieve the same level of consistency [1].

Theorem 1. *Generalised arc consistency (GAC) on the occur constraint in* $ind2$ *achieves the same level of consistency as bounds consistency (BC) on the sum constraint in* $ind1$.

[1] This proof is due to Francois Laburthe.

Proof. We need to prove that $(BC(a \leq sum(X) \leq b) \Leftrightarrow GAC(a \leq occur(1, X) \leq b)$, where X is a set of 0/1 variables $\{x_1, ..., x_n\}$. Since both constraints have the same solution set, and GAC is stronger than BC we know that $GAC(occur(1, X))$ is at least as strong as $BC(sum(X))$ i.e. any value removed by $BC(sum(X))$ is also removed by $GAC(occur(1, X))$. We now prove that any value removed by $GAC(occur(1, Xi))$ is also removed by $BC(sum(X))$. Let nb_1 be the number of variables instantiated to 1, nb_0 be the number of variables instantiated to 0, and $nb_{0/1}$ be the number of uninstantiated variables. Suppose the value $x_i = 0$ is removed by $GAC(occur(1, X))$. We consider two cases:

(i) All other variables are instantiated. Consequently both constraints reduce to unary constraints, and $x_i = 0$ is also removed by $BC(sum(X))$.
(ii) Variables $x_{j1}, ..., x_{jk}$ are not instantiated, and the tuple $(x_i = 0, x_{j1} = 1, ..., x_{jk} = 1)$ along with the instantiated variables is infeasible. Consequently $nb_1 + nb_{0/1} - 1$ is not in the interval $[a, b]$ Similarly, the tuple $(x_i = 0, x_{j1} = 0, ..., x_{jk} = 0)$ along with the instantiated variables is infeasible. Consequently nb_1 is not in $[a, b]$. Therefore the intervals $[nb_1, nb_1 + nb_{0/1} - 1]$ and $[a, b]$ are disjoint. Either (a) or (b) hold
  (a) $nb_1 + nb_{0/1} - 1 < a$. So, $sum(X - \{x_i\}) < a$, where all uninstantiated variables in $X - \{x_i\}$ contribute the value 1 to the sum. $BC(sum(X))$ then removes the value $x_i = 0$
  (b) $nb_1 > b$. So $sum(X - \{x_i\}) > b$, where all uninstantiated variables in $X - \{x_i\}$ contribute the value 0 to the sum. Again, $BC(sum(X))$ removes the value $x_i = 0$.

  Therefore, when $GAC(occur(1, X))$ removes a value so does $BC(sum(X))$.

Since GAC is stronger than BC, and whenever $GAC(occur(1, X))$ removes a value so does $BC(sum(X))$, the two representations $ind1$ and $ind2$ achieve the same level of consistency. *QED*

2.2 Maximal Independent Set

Given a hypergraph $H = (V, E)$, where V is the set of vertices and E is the set of edges, a *maximal* independent set M is a set such that there is no independent set M' that subsumes M, i.e. we cannot add any vertex to M without losing the independence property. The problem is then, given some integer $k < |V|$, is there a maximal independent set of size k [7]?

Using Figure 1 as an example, when $k = 5$ there are 3 maximal independent sets, one of these being $\{v_2, v_3, v_4, v_8, v_9\}$. When $k = 6$ there are 11 maximal independent sets, one of these being $\{v_2, v_3, v_5, v_6, v_8, v_9\}$. When $k = 7$ there is a single maximal independent set $\{v_1, v_2, v_5, v_6, v_7, v_8, v_9\}$ and this is the largest, and is therefore the maximum independent set. There are no maximal independent sets for any other values of k.

We need a constraint to specify when a vertex is in the maximal independent set, and when it is not in the maximal independent set. Looking again at Figure 1 we can see that vertex v_2 is not in the maximal independent set M if vertices

v_4 and v_5 are in M or v_1 and v_3 are in M. We can express this with the following constraint:

$$(v_4 + v_5 = 2) \vee (v_1 + v_3 = 2) \leftrightarrow v_2 = 0$$

where $\leftrightarrow$ is the biconditional *if and only if*. Alternatively we can express when v_2 is in M. Therefore we might alternatively have the constraint

$$(v_4 + v_5 < 2) \wedge (v_1 + v_3 < 2) \leftrightarrow v_2 = 1$$

The biconditional $p \leftrightarrow q$ is logically equivalent to $(p \wedge q) \vee (\neg p \wedge \neg q)$ and to $(p \rightarrow q) \wedge (q \rightarrow p)$. Therefore using the definitions below for (1) p, (2) $\neg p$, (3) q and (4) $\neg q$ we can describe the maximality constraint equivalently as $p \leftrightarrow q$, or as $(p \rightarrow q) \wedge (q \rightarrow p)$, or as $(p \wedge q) \vee (\neg p \wedge \neg q)$

$$p: \quad v_i = 1 \tag{1}$$
$$\neg p: \quad v_i = 0 \tag{2}$$
$$q: \bigwedge_{e \in E(v_i)} \sum_{v_j \in V(e) - v_i} v_j < arity(e) - 1 \tag{3}$$
$$\neg q: \bigvee_{e \in E(v_i)} \sum_{v_j \in V(e) - v_i} v_j = arity(e) - 1 \tag{4}$$

where $E(v_i)$ is the set of edges involving variable/vertex v_i and $V(e) - v_i$ is the set of variables/vertices in the edge e excluding vertex v_i.

As stated above, the maximum independent set is also a maximal independent set. Consequently when reformulating a csp P of n variables as a problem of finding an independent set of size n, we could also incorporate the redundant maximality constraints. Therefore one of the questions we investigate is, does the redundant maximality constraint improve search performance?

2.3 Balanced Incomplete Block Design

A balanced incomplete block design (bibd) is an arrangement of v objects into b blocks, each of size k. Each element of v occurs in r blocks and every possible pair of objects occurs together in λ blocks [5] Therefore, a bibd can be defined by the quintuple $\langle v, b, r, k, \lambda \rangle$, and visualised as a v by b matrix of 0/1 values. There are r 1's in each row, k 1's in each column, and the scalar product of any two rows is equal to λ. Tabulated below is a matrix for the bibd $\langle 6, 10, 5, 3, 2 \rangle$

We encode this in the most naive way [2]. We have $v \times b$ 0/1 variables, $v_{i,j}$. There are v sum constraints for each row and b sum constraints for each column. For every pair of rows, (i, j), we generate b additional variables $l_{i,j,1}$ to $l_{i,j,b}$, and b constraints of the form $v_{i,k} = 1 \wedge v_{j,k} = 1 \leftrightarrow l_{i,j,k} = 1$. Finally, for the pair of rows we have the $b-$ary constraint $\sum_{k=1}^{b} l_{i,j,k} = \lambda$.

The biconditional can again be encoded in three ways, as described in the previous subsection, and the summation constraints can be encoded as occurrence constraints.

[2] A more efficient encoding is proposed in [6]

0	0	0	0	0	1	1	1	1	1
0	0	1	1	1	0	0	0	1	1
0	1	0	1	1	0	1	1	0	0
1	0	1	0	1	1	0	1	0	0
1	1	0	1	0	1	0	0	0	1
1	1	1	0	0	0	1	0	1	0

Table 1. An instance of bibd $\langle 6, 10, 5, 3, 2\rangle$

3 The Empirical Study

The experiments were run on two different machines. The Choco experiments were run on a Pentium III 755 MHz processor with 256MB of ram, and the Solver experiments on a Pentium III 933 MHz processor with 1GB of ram. We use bibd's as data sets for our study of (maximal) independent sets. The bibd can be viewed as a regular hypergraph, with each vertex of degree r and each hyperedge of arity k. We use three such hypergraphs which we denote A, B, and C. Hypergraphs A and B both correspond to non-isomorphic instances of the bibd $\langle 25, 50, 8, 4, 1\rangle$ and C corresponds to an instance of $\langle 40, 130, 13, 4, 1\rangle$.

Table 2. Search effort (nodes) and CPU time (milliseconds) to find the first independent set of size k for hyper graphs A, B, and C.

	k	Sol	Nodes	ind1S	ind2S	ind1C	ind2C
A	14	yes	170	125	93	290	20
	15	no	36901	781	2812	70350	3530
	16	no	17652	406	1359	35190	1760
	17	no	7585	218	625	16420	800
	18	no	3150	125	297	7220	350
B	14	yes	16	62	62	20	0
	15	yes	16	62	62	20	0
	16	no	17516	422	1344	36280	1740
	17	no	7503	218	625	16960	790
	18	no	3058	125	281	7050	340
C	21	yes	19219	609	2640	78280	2760
	22	yes	101217	2875	13172	-	14320
	23	no	8237508	225703	1059660	-	1147860
	24	no	4136599	114734	512015	-	580200

In Table 2 we have the results of searching for the first independent set of size k for the three hypergraphs. The column *Sol* is "yes" if an independent set of size k was found. Note that k is increasing as we move down the table, and the last "yes" entry corresponds to the largest independent set. Results are given for Ilog Solver 5.0 and Choco 1.07 implementations, where $ind1$ uses summation of variables and $ind2$ uses the occurrence constraint, $ind1S$ and $ind2S$ are the Solver implementations and $ind1C$ and $ind2C$ are the Choco implementations. Both implementations explore the same number of nodes, as expected. Run time is given in milliseconds, and where there is a $-$ entry, the process was terminated after more than 16 hours (i.e. from 5pm to 9am).

The difference between the Solver implementations is large, with the summation constraint ($ind1S$) significantly faster than the occurrence constraint ($ind2S$). The difference is always in $ind1S$'s favour and is typically about a factor of three. In Choco the difference is typically a factor of about twenty, but this time in the favour of $ind2C$. That is, the occurrence implementation dominates the summation implementation in Choco by a factor of twenty, whereas in Solver the summation dominates the occurrence constraint by a factor of three.

Table 3. Search effort (nodes) and CPU time (milliseconds) to find the first maximal independent set of size k for hyper graphs A and B.

	k	Nodes	mis1S	mis2S	mis3S	mis1C	mis2C	mis3C
A	14	62	78	94	78	40	60	40
	15	25093	6781	9781	3375	18450	31090	35620
	16	15862	3469	5016	1969	13210	18630	22560
	17	7585	1516	2172	969	5530	7380	9200
	18	3150	641	891	437	1840	2630	3580
B	14	87	94	109	78	50	90	20
	15	16	62	62	62	10	10	1860
	16	15744	3453	4969	1937	14080	17980	22210
	17	7485	1484	2156	937	4480	7500	9300
	18	3058	609	875	422	2590	3610	3560

Table 3 gives the results of our study of the maximal independent set problems, i.e. a study of our different encodings of the biconditional constraint $p \leftrightarrow q$. We use the two hypergraphs A and B, and search for the first maximal independent set of size k. Again, experiments were performed in Ilog Solver and in Choco. The column $mis1S$ gives the Solver run time (milliseconds) for the encoding of the biconditional $p \leftrightarrow q$, and $mis1C$ is the Choco equivalent. Columns $mis2S$ and $mis2C$ report on the encoding of the biconditional in Solver and Choco as $(p \rightarrow q) \wedge (q \rightarrow p)$, and columns $mis3S$ and $mis3C$ as $(p \wedge q) \vee (\neg p \wedge \neg q)$. All these encodings use the summation constraint, rather than the occurrence constraint. From our results we see that in Solver $(p \wedge q) \vee (\neg p \wedge \neg q)$ (i.e. column $mis3S$) is the fastest implementation of the biconditional, typically 50% faster than $mis1S$ and twice as fast as $mis2S$. In Choco $p \leftrightarrow q$ (i.e. $mis1C$) is the fastest implementation of the biconditional.

Our final experiments are on first solution search for bibd using Choco. We use the same problems studied by Prestwich [8].

Table 4. CPU time (milliseconds) to find the first bibd that satisfies the given parameters

Parameters	bibd0	bibd1	bibd2	bibd3	bibd4	bibd5
$\langle 7,7,3,3,1 \rangle$	30	10	10	10	20	10
$\langle 6,10,5,3,2 \rangle$	70	30	40	30	40	3670
$\langle 7,14,6,3,2 \rangle$	400	150	220	170	360	20
$\langle 9,12,4,3,1 \rangle$	260	90	150	120	560	40
$\langle 8,14,7,4,3 \rangle$	1000	340	500	410	820	–

Six different encodings are used. *bibd*0 uses the summation constraint, whereas all others (*bibd*1 to *bibd*5) use the occurrence constraint. *bibd*0 and *bibd*1 encode the biconditional as $p \leftrightarrow q$, whereas *bibd*2 uses $(p \rightarrow q) \wedge (q \rightarrow p)$. Encoding *bibd*3 uses multiplication, i.e. the constraint $(X = 1 \wedge Y = 1) \rightarrow Z = 1$ can be replaced with $X \times Y = Z$, because $X, Y, Z \in \{0, 1\}$. Encoding *bibd*4 uses the Choco constraint $and(ifThen(p, q), ifThen(q, p))$, and this behaves as two lazy implications. Encoding *bibd*5 uses $(p \wedge q) \vee (\neg p \wedge \neg q)$. These six encodings are all logically equivalent, and the search processes all visit the same number of nodes.

Comparing *bibd*0 with *bibd*1 we see again that in Choco the occurrence constraint is more efficient than the summation constraint, although not as significantly as in independent set (Table 2). Table 4 shows that the direct encoding of the biconditional, *bibd*1, gives the best performance. Multiplication, *bibd*3 is the next best thing, whereas the encoding $(p \wedge q) \vee (\neg p \wedge \neg q)$ is again the worst, failing to terminate in reasonable time on $\langle 8, 14, 7, 4, 3 \rangle$.

4 Drawing the Wrong Conclusions from the Wrong Model

The notion of maximality can be encoded as a redundant constraint when we are searching for an independent set of size k if and only if we know that k is the size of the largest independent set. In particular, we can use this constraint when we reformulate a csp of n variables into the problem of finding an independent set of size n in the corresponding hypergraph. Would this redundant constraint pay off? We can compare some of the results from Table 2 with those in Table 3 to allow us to imagine how such an investigation might proceed.

The largest tabulated value of k for which we find a maximal independent set also corresponds to the size of the largest independent set. Therefore we can compare Tables 2 and 3 for hypergraph A with $k \geq 14$, and B with $k \geq 15$ (i.e. B's independent set of size 14 might not be maximal, therefore we can only consider B problems with $k \geq 15$).

In our imaginary (Choco) experiments we encode our problem using the summation constraint and search for an independent set of size k. This corresponds to column $ind1C$ in Table 2. We then encode our problem again, this time using the redundant maximality constraint. Assume this uses the encoding of the biconditional corresponding to $mis1C$ in Table 3, i.e. using Choco's $ifOnlyIf$ constraint. This is re-tabulated in Table 5, comparing only $mis1$ with $ind1$ in both Choco and Solver. The redundant constraint gives us a speed up of a factor of about three. This suggests that our reformulation is an improvement, and we might recommend it. In fact, we get an improvement regardless of our encodings of the biconditional!

Table 5. CPU time (milliseconds) to find the largest independent set, with (mis) and without (ind) the redundant maximality constraint. Does maximality help? It depends on the toolkit.

	k	ind1S	mis1S	ind1C	mis1C
A	14	125	78	290	40
	15	781	6781	70350	18450
	16	406	3469	35190	13210
	17	218	1516	16420	5530
	18	125	641	7220	1840
B	15	62	62	20	10
	16	422	3453	36280	14080
	17	218	1484	16960	4480
	18	125	609	7050	2590

We can now imagine our experiments, but this time using Solver. We compare column $ind1S$ in Table 2 with columns $mis1S$, $mis2S$, and $mis3S$ in Table 3. Looking at Table 5 we see that the maximality constraint degrades performance, sometimes by a factor of ten! Our Solver experiments would suggest that the maximality constraint should be avoided, at all costs! Therefore, we can replicate our empirical study, changing only the implementation toolkit and reach an entirely different conclusion.

5 Why was there a difference?

Why should there have been such a dramatic difference in the performance of the $sumVars$ and $occur$ constraints in Choco? The following is an explanation due to Francois Laburthe, one of the authors of Choco:

> The constraint propagation phase in Choco is based on a dual event queueing mechanism. The filtering algorithm of a constraint can be implemented in two ways, either as one general revision procedure reaching consistency from any state or as parametrised procedure that reach consistency after a given domain reduction on a given variable. The first behaviour corresponds to the general description of arc consistency algorithms for constraints specified by list of feasible tuples; the second corresponds to specialised propagation patterns inspired from rule-based systems. Choco features two pools of pending events: domain updates and constraint revisions. The first pool is of higher priority, ie: whenever all immediate propagation after the domain updates have been done, the pending constraints are awoken.
> The implementation used in this study propagates linear constraints as "delayed constraints" (generic revision procedure) This was motivated by the fact that one linear pass is enough to reach bounds consistency, no matter the number of variables whose domain have been reduced since the former AC state. This turns out to be too slow on the examples above for the following reasons: (a) time is wasted checking whether there are pending variable event before popping each constraint event (b) it seems that this leads to more constraint checks in infeasible situations. Choco has now been modified in the light of these results. The change consisted in altering the management of linear constraints: up to a certain number of variables (set by default to 8), they are propagated through the "variable event mechanism". When they involve more, they keep being propagated in the "constraint revision event mechanism"

With these changes in place, Choco's performance is now no more than 6 times worse than Solver over the data sets presented here. We can see in some cases it is a close competitor. For example in Table 2, if we compare columns $ind2S$ and $ind2C$ we see that both toolkits run in roughly the same time, even though Choco is on a 755MHz machine and Solver is on a 933Hz machine.

6 Conclusion

We have examined two constraints, the biconditional and a restriction on the sum of 0/1 variables. The various implementations presented are logically equivalent and result in the same exploration of the search space. Yet the run times are often very different. For example, one encoding was reliably twenty times faster than another. Unfortunately, this did not translate across programming toolkits. We saw that summation was faster than counting occurrences in Ilog Solver, yet it was the other way round in Choco. Even more notable was that a good encoding of the biconditional in Solver corresponded to the worst in Choco!

Why are there differences in run times when we make these subtle reformulations? They have different types of constraints and different numbers of constraints. The performance of arc consistency can be affected by the order that constraints are processed (see for example [3] or [10]). This is one explanation. The other is most obviously down to the way the toolkit providers have implemented the various constraints in the first place (for example, section 5).

What lessons can we learn? First, what we learn with one toolkit might not carry over to another. Therefore, we need to exercise caution when picking up a new tool. Second, when we have an implementation we should not let it rest there; we should investigate other logically equivalent implementations. That is, we should experiment. We often explore different ways of formulating a problem, maybe reaching a conclusion that one formulation is better than the other, measured as run time. Before we do this, we should make sure that we have explored the finest level of formulation, the actual implementation. And finally, as in other branches of science we should be encouraged to replicate results, just as we have done here using two toolkits. Obviously there is value in doing this.

Acknowledgements

We would like to thank ILOG, our collaborators on this project, the OCRE team, Alice Miller for introducing us to these problems, and Francois Laburthe. Francois gave us the proof in subsection 2.1 and the explanation in 5.

References

1. CHOCO. http://www.choco-constraints.net/ home of the choco constraint programming system.
2. Michael. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, 1979.
3. Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Paul Shaw, and Toby Walsh. The constrainedness of arc consistency. In *Principles and Practices of Constraint Programming*, pages 327–340, 1997.
4. ILOG. http://www.ilog.com.
5. R. Mathon and A. Rosa. Tables of parameters of bibds with r $\leq$ 41 including existence, enumeration, and resolvability results. *Annals of Discrete Mathematics*, 26:275–308, 1985.
6. P. Meseguer and C. Torras. Exploiting symmetries within constraint satisfaction search. *Artificial Intelligence*, 129:133–163, 2001.
7. Nina Mishra and Leonard Pitt. Generating all maximal independent sets of bounded-degree hypergraphs. In *Tenth Annual Conference on Computational Learning Theory*, pages 211–217, 1997.
8. S. D. Prestwich. Balanced incomplete block design as satisfiability. In *12th Irish Conference on Artificial Intelligence and Cognitive Science*, 2001.
9. Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.
10. Rick Wallace. Why ac3 is almost always better than ac4 for establishing arc consistency in csp's. In *IJCAI-93*, pages 239–245, 1993.
11. R. Weigel and C. Bliek. On reformulation of constraint satisfaction problems. In *Proceedings of ECAI-98*, pages 254–258, 1998.