Two Set-Constraints for Modeling and Efficiency

Willem-Jan van Hoeve[1] and Ashish Sabharwal[2]

[1] Tepper School of Business, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213, U.S.A.
vanhoeve@andrew.cmu.edu
[2] Department of Computer Science, Cornell University
5160 Upson Hall, Ithaca, NY 14853-7501, U.S.A.
sabhar@cs.cornell.edu

Abstract. Set variables provide convenient modeling shorthands for many combinatorial problems. However, it is often challenging to efficiently handle set constraints when solving the problem. We present efficient filtering algorithms, establishing bounds consistency, for two such constraints: the `sum-free` constraint, and the `atmost1` constraint on pairs of set variables with known cardinality. The filtering algorithm for the `sum-free` constraint achieves the same pruning as the corresponding collection of constraints on the binary representation, but it does so more efficiently and without running into memory bottlenecks. For the `atmost1` constraint on pairs of set variables, the additional time spent on pruning more values pays off well in terms of overall efficiency. Our results show that set constraints can not only ease modeling the problem, they can also decrease the solution time and memory requirements.

1 Introduction

Many combinatorial problems, such as bin packing, set covering, and combinatorial design, can be conveniently expressed using set variables and constraints over these variables [10]. However, for solving such problems in practice, people often resort to integer programming (IP), or constraint programming (CP) on single-valued domains, for which efficient off-the-shelf solvers are available. From a constraint programming perspective, set variables (or variables with a structured domain) are not fundamentally different from 'normal' variables with single-valued domains. Set variables are present in most constraint programming solvers, such as ILOG Solver [11], Eclipse [6], and Gecode [9]. We can post constraints over them, enumerate their domains in a search tree, and filter their domains according to the constraints. This makes constraint programming a particularly suitable environment to both model a problem in its natural form using set variables and reason over those variables directly.

It is widely accepted that domain filtering algorithms for global constraints have greatly improved the performance of constraint programming solvers, and thus their use in practice [3, 18]. Most of these filtering algorithms, if not all, were designed for constraints on single-valued variables. Over the years, several

researchers have studied the application of set variables and set constraints in constraint programming (we refer to [10] for a survey). Nonetheless, the number of (global) set constraints with efficient domain filtering algorithms is limited. We believe that the development of such algorithms is indispensable for the successful application of set variables and set constraints, which would allow modeling and solving many problems in a more natural and efficient way. This paper is a step in this direction.

In the last few years, several researchers have studied filtering algorithms for set constraints. For example, Sadler and Gervet [15] presented filtering algorithms for the `atmost1` constraint, and for the `alldifferent` constraint on set variables with known cardinality. More recently, Dooms and Katriel [5] introduced filtering algorithms for the minimum spanning tree constraint over graph variables, which are defined by two set variables representing the vertices and the edges of a graph, respectively.

Filtering constraints over set variables can be much more challenging than filtering constraints on single-valued variables. Intuitively this can be explained by the fact that a constraint on a single set variable may capture a structure that must otherwise be represented by a global constraint on multiple single-valued variables. Hence, a constraint on a single set variable may be as difficult to filter as a global constraint on multiple single-valued variables. Filtering a global constraint involving more than one set variables can therefore be even more difficult.

In this work, we present efficient filtering algorithms for two constraints on set variables. First, we study the `sum-free` constraint. This constraint is defined on a single integer-valued set variable and states that for each pair of (possibly identical) elements of the set, their sum may not be an element of that set. This constraint arises naturally in a well-known number-theoretic problem: finding the Schur number. This latter problem has been studied, for example, by Fredricksen and Sweet [8], who give lower bounds for the Schur number. Our experimental results on this problem highlight, in addition to efficiency gains, the succinctness of the set representation, which allows us to solve several problems that cannot even be modeled with an equivalent integer representation without exceeding memory limits.

The `atmost1` constraint was introduced by Sadler and Gervet [15]. It states that for $n \geq 2$ set variables with known cardinality, each pair of variables can share at most one element. It was shown by Bessiere et al. [4] that it is NP-hard to establish bounds consistency for this constraint. The filtering algorithm proposed by Sadler and Gervet [15] runs in polynomial time and filters the domains partially, i.e., it does not necessarily establish bounds consistency.[3] In this paper, we give in on complete bounds consistency for n sets for general n, and instead focus on `atmost1` constraints on *pairs* of variables (i.e., for $n = 2$). We show that in this case, it is possible to establish bounds consistency in polynomial time. The filtering algorithm here is more involved than for the `sum-free` constraint, and exploits the indistinguishability of a few different kinds

[3] This is true even for $n = 2$, as one of our example scenarios will show.

of elements in the two set variables. We use as a test bed for this constraint the Social Golfer problem, which has been studied extensively in the context of symmetry breaking. Our method, however, is orthogonal to symmetry-based techniques and focuses on better filtering. Our experiments highlight a significant reduction in the number of failed search nodes as well as the runtime.

2 Preliminaries

We recall basic definitions and concepts from constraint programming. We refer to Rossi et al. [14] for an overview.

Let x be a variable. The *domain* of x, denoted $D(x)$, is a finite set of elements (also called domain values) that can be assigned to x. A *set variable* is a variable whose domain values are sets. Since the number of possible values of a set variable can be enormous (the size of a power set, in the worst case), one often represents the domain of a set variable S by an "interval" $[L(S), U(S)]$, where $L(S)$ and $U(S)$ are sets such that $L(S) \subseteq U(S)$ and $D(S) = \{s \mid L(S) \subseteq s \subseteq U(S)\}$. In words, this says that all elements of $L(S)$ must be in S and any elements not in $U(S)$ must not be in S. In this sense, $L(S)$ and $U(S)$ form a lowerbound and an upperbound for S, respectively. For example, let V be a set, let $a, b \in V$ be two elements of V, and let S have domain $D(S) = [\{a\}, V \setminus \{b\}]$. Then $D(S)$ consists of all possible subsets of V that contain a but do not contain b. We assume throughout that variable domains are finite.

The solution process of constraint programming interleaves *constraint propagation* and *search*. The search process essentially consists of enumerating all possible combinations of variable domain values, until we find a solution to the CSP at hand or prove that none exists. This process can be thought of as constructing a *search tree*, starting with no variable restrictions at the root and constraining the domains of more and more variables as one goes down the tree. To reduce exhaustive search of the exponential number of variable-value combinations, we *filter* the domains of the variables and *propagate* this information through all constraints:

> Filter and Propagate: Given the current domains and a constraint C, remove domain values that do not belong to any solution to C. Repeat for all constraints until no more domain values can be removed.

We typically apply constraint propagation at each node in the search tree. In order to be effective, filtering algorithms must be efficient because they are executed several times during the solution process. Furthermore, they should remove as many domain values that are not part of a solution as possible. If a filtering algorithm for a constraint C removes *all* such values from the domains with respect to C, we say that it makes C *domain consistent*. In the context of set variables, which are represented by an interval defined by a lower bound and an upper bound, we apply *bounds consistency* instead:

Definition 1. *Let $S_1, \ldots, S_n$ be set variables. A constraint $C(S_1, \ldots, S_n)$ is called* bounds consistent *if for all $i = 1, \ldots, n$, $L(S_i)$ and $U(S_i)$ are the intersection and the union, respectively, of all values in $D(S_i)$ that can be assigned to S_i in a solution to C.*

Note that Definition 1 does not require the lower or upper bounds of a variable domain to themselves belong to a solution of the constraint. When a filtering algorithm for set constraints does not necessarily establish bounds consistency, we call it a *partial* filtering algorithm.

Finally, we recall the concept of *domain-delta* [17]. It represent the changes in the domain of a variable x between two filtering events, and is denoted by $\Delta(x)$. For a set variable S, the domain-delta consists of two sets: a set $\Delta_L(S)$ of elements added to the lower bound and a set $\Delta_U(S)$ of elements removed from the upper bound. Domain-deltas can be very helpful in designing efficient incremental algorithms. Specifically, they can allow the complexity of a filtering algorithm to be amortized over an entire path in the search tree from the root to any leaf. Both of our algorithms in this paper exploit domain-deltas, although our current implementation for `atmost1` constraint does not use domain-deltas because of technical reasons (see end of Section 4).

3 The Sum-Free Constraint

In this section, we present a relatively straightforward yet effective filtering algorithm for the `sum-free` constraint. As we will demonstrate in the experimental section, its main purpose is to address ease of modeling and memory requirements for the problem. We first give a formal definition of the constraint:

Definition 2. *Let S be a set variable with domain $D(S) = [L(S), U(S)]$, where $\varnothing \subseteq L(S) \subseteq U(S) \subseteq \mathbb{N}^+$. The* `sum-free` *constraint on S is defined as*

$$\texttt{sum-free}(S) = \{s \mid s \in D(S), (i, j \in s) \Rightarrow (i + j \notin s)\}\,.$$

$\mathbb{N}^+$ here denotes the set of positive integers. Note that the constraint does not require $i \neq j$, so that for each $i \in s$, we have that $2i \notin s$. A filtering algorithm for this constraint is described as Algorithm 1. For each element i added to the lower bound of S, we compare i to all other elements j in the lower bound of S, and remove the corresponding possible sums $i + j$ and $|i - j|$ from the upper bound of S. For efficiency, we only consider elements i in the domain-delta $\Delta_L(S)$; this is justifiable because once two elements are added to the lower bound, they never need to be processed again as a pair.[4] Also note that the domain of S may be the empty set according to the `sum-free` constraint. Hence the filtering algorithm never returns 'inconsistent'.

Proposition 1. *Algorithm* FilterSumFree *establishes bounds consistency on the* `sum-free` *constraint.*

[4] A technical detail is that when ILOG Solver makes the first call for filtering, the domain-deltas may be empty while the lower bounds are potentially non-empty. In this special case, we use $L(S)$ instead of $\Delta_L(S)$ in the first for-loop.

```
FilterSumFree(S)
begin
    for i ∈ Δ_L(S) do
        for j ∈ L(S) do
            U(S) ← U(S) \ {i + j}
            U(S) ← U(S) \ {|i − j|}
end
```

Algorithm 1: Filtering algorithm for the `sum-free` constraint.

Proof. The algorithm removes all elements that can never be part of a solution. Hence, the resulting upper bound respects the bounds consistency definition. For bounds consistency w.r.t. the lower bound, observe that we can never add more elements to the lower bound and indeed the `sum-free` constraint can never force any element to be in the set. The justification for using domain-deltas was discussed already. □

A single call to FilterSumFree takes $O(|\Delta_L(S)|\,|L(S)|)$ time because of the two loops. Here we assume linear-time element listing for $\Delta_L(S)$ and $L(S)$ using implementations such as array, linked list, or tree, and constant time set deletion operations for $U(S)$ using implementations such as a bit-vector. This is $O(n^2)$ in the worst case, where n is the integer domain size for $U(S)$. We can obtain a tighter analysis by amortizing this complexity for any path from the root of the search tree to a leaf. Specifically, the cumulative complexity along any such path is $O(\sum_{\text{path}} |\Delta_L(S)|\,|L(S)|)$, which is at most $O(n \sum_{\text{path}} |\Delta_L(S)|) = O(n^2)$.

As a final remark, we note that the natural integer representation of the `sum-free` constraint achieves exactly the same pruning as FilterSumFree.

4 The Atmost1 Constraint on Pairs of Variables

In this section, we present a bounds consistent filtering algorithm for the `atmost1` constraint on pairs of set variables. The general `atmost1` constraint specifies, for a collection of set variables with given cardinalities, that each pair of variables overlaps in at most one element. Formally:

Definition 3 (adapted from [15]). *Let $S_1, \ldots, S_n$ be set variables and let $c_1, \ldots, c_n \geq 1$ be integers. The* `atmost1` *constraint on the n sets S_i and the corresponding cardinalities c_i is defined as* $\texttt{atmost1}(S_1, \ldots, S_n, c_1, \ldots, c_n) =$

$$\{(s_1, \ldots, s_n) \mid \forall i, 1 \leq i \leq n : s_i \in D(S_i), |s_i| = c_i; \\ \forall i, j, 1 \leq i < j \leq n : |s_i \cap s_j| \leq 1\}.$$

As mentioned earlier, filtering the `atmost1` constraint to bounds consistency is NP-hard [4]. In this work, we consider the `atmost1` constraint involving two set variables only (i.e., for $n = 2$), which we will refer to as the `pair-atmost1`

constraint. A natural way of implementing this constraint is to use the following decomposition of `pair-atmost1`(S_1, S_2, c_1, c_2) into three constraints:

$$|S_1| = c_1, \quad |S_2| = c_2, \quad |S_1 \cap S_2| \leq 1.$$

We will refer to this as the *standard decomposition* for `pair-atmost1`. Unfortunately, this decomposition treats the three constraints separately, and filtering these constraints separately does not lead to bounds consistency on the `pair-atmost1` constraint. For example, the intersection constraint by itself never forces any element to be added to the lower bound of either set, although this is sometimes possible through reasoning together with the cardinality constraints for the sets. This is illustrated by the following example:

Example 1. Consider the following scenario: $D(S_1) = [\{1,2\}, \{1,2,3,5,6\}]$, $D(S_2) = [\{3\}, \{1,2,3,4\}]$, and the required cardinalities are $c_1 = c_2 = 3$. Since 1 and 2 both must belong to S_1, at most one of them may be in S_2 due to the intersection constraint. This, because of the cardinality requirement for S_2, forces 4 to always be in S_2. Also, again because of the cardinality requirement for S_2, at least one of 1 and 2 must belong to S_2, which in turn implies that the pairs (1,3) and (2,3) cannot be in S_1 due to the intersection constraint; that is, 3 cannot be in S_1. With this reasoning, a bounds-consistent filtering algorithm will achieve the following domain filtering: $D(S_1) = [\{1,2\}, \{1,2,5,6\}]$, $D(S_2) = [\{3,4\}, \{1,2,3,4\}]$.

In this example, the standard decomposition does not achieve any filtering at all because it does not add values to the domain lower bounds to begin with. In particular, it will not add 4 to $L(S_2)$. In fact, the original algorithm of Sadler and Gervet [15] for `atmost1` also does *not* achieve bounds consistency on this example scenario.

4.1 Bounds Consistency

We now describe algorithm BC-FilterPairAtmost1 (shown as Algorithm 2), which exploits the interplay between the intersection constraint and the two cardinality constraints, and achieves bounds consistency for `pair-atmost1`. It applies a somewhat involved data structure, but its eventual filtering steps are surprisingly simple, taking constant time for the actual checks. The complexity of the algorithm is dominated by the availability of incremental set operations during search. We next describe in words the idea behind the algorithm.

Given set variables S_1 and S_2, and cardinalities c_1 and c_2, we first scan the elements of the lower and upper bounds of the domains of S_1 and S_2, and partition the elements of each set variable into 6 disjoint sets. For S_1, we have L1only, L1L2, L1U2, U1only, U1L2, U1U2; for S_2, we have L2only, L2L1, L2U1, U2only, U2L1, U2U1. The semantics of these sets are straightforward, except for U1 here being a shorthand for $U(S_1) \setminus L(S_1)$, and similarly for U2. For instance, L1L2 denotes $L(S_1) \cap L(S_2)$, L1only denotes $L(S_1) \setminus L(S_2)$, U1L2 denotes $(U(S_1) \setminus L(S_1)) \cap L(S_2)$, etc. Note that L1L2 = L2L1, U1L2 = L2U1, and U2L1 = L1U2.

For these three pairs, we explicitly maintain only one set per pair, namely, L1L2, U1L2, and U2L1, respectively. (While U1U2 = U2U1 as well, we still maintain both of these sets because we need different "flags" for these two sets, as will become clear shortly.) Henceforth, we will talk of the remaining 9 sets that we consider and into which the elements are partitioned.

Example 2. Consider the scenario of Example 1, where $L(S_1) = \{1, 2\}, U(S_1) \setminus L(S_1) = \{3, 5, 6\}, L(S_2) = \{3\}$, and $U(S_2) \setminus L(S_2) = \{1, 2, 4\}$. The 9 sets in this case are: L1only = $\varnothing$, L2only = $\varnothing$, L1L2 = $\varnothing$, U1only = $\{5, 6\}$, U2only = $\{4\}$, U1L2 = $\{3\}$, U2L1 = $\{1, 2\}$, U1U2 = $\varnothing$, and U2U1 = $\varnothing$.

A key observation is that *the elements in each of these 9 sets behave identically* as far as filtering for `pair-atmost1` is concerned. That is, if, say, the filtering algorithm concludes that *some* element $x \in$ L1L2 needs to be removed from $U(S_2)$, then it must be the case that *all* elements of L1L2 need to be removed from $U(S_2)$. In this sense, all elements within each of the 9 sets are indistinguishable from each other. In Example 2, elements 5 and 6 are indistinguishable, and so are elements 1 and 2. The algorithm exploits this fact, and once these 9 sets are constructed, it is able to identify in constant time the elements to be added to the lower bounds or to be removed from the upper bounds. Of course, filtering the identified elements then takes time proportional to the number of elements filtered. Accordingly, until the final filtering step of the algorithm, we only maintain the cardinality of and one arbitrary representative element from each of the 9 sets.

For each of the 9 sets, we maintain two Boolean flags throughout, which are all initialized to False to begin with. These are the "can-have" flag and the "not-necessary" flag. During the course of the algorithm, we turn the can-have flag for one of the 9 sets to True when we determine that there is a solution with *some* element from that set included in S_1 or S_2, as appropriate. Similarly, we turn the not-necessary flag to True when we determine that there is a solution to the constraint without using *any* element of this set in S_1 or S_2, as appropriate. The actual filtering is done at the very end, once we have gone through every case of the filtering algorithm. At this point, we examine each of the 18 flags. If, say, U1L2.can-have is still False, this implies that none of the solutions contain an element of U1L2, so that we can remove U1L2 from $U(S_1)$. Similarly, if, say, U1U2.not-necessary is still False, this implies that all elements in U1U2 are in fact necessary for any solution and we can add U1U2 to $L(S_1)$. These 18 checks are the only real filtering steps of the algorithm.

We are now ready to describe filtering for `pair-atmost1`(S_1, S_2, c_1, c_2). If $|\text{L1L2}| > 1$, we have an immediate failure. Otherwise, BC-FilterPairAtmost1 implicitly goes through every possible solution to the constraint, given the domain values of S_1 and S_2. For this, it considers several cases, based on which of the 9 sets, if any, the *shared element*, i.e., the element common to both sets in the solution, comes from. We will refer to the case that S_1 and S_2 do *not* share any element in the solution as Case0. Updating the can-have and not-necessary flags for this case will form the basic block of the overall algorithm, the other

cases reducing to this one with appropriate minor modifications to the involved constants.

Filtering for Case0. If |L1L2| = 1, there is no solution in Case0 and we stop updating flags for this case. Otherwise, let $k_1 = c_1 - |L(S_1)|$, i.e., the number of new elements that must be added to S_1 to achieve cardinality c_1. Similarly define k_2 for S_2. The number of potential elements that may be added to S_1 is $\ell_1 = |\text{U1only}| + |\text{U1U2}|$, because elements in U1L2 cannot be in S_1 in Case0. Let slack1 $= \ell_1 - k_1$, that is, the number of "extra" elements we have available for S_1. Similar define slack2. Finally define slack3 to be the total number of elements available for S_1 and S_2 combined, less $k_1 + k_2$. Formally, slack3 = (|U1only| +|U2only| + |U1U2|) − $(k_1 + k_2)$.

Lemma 1. *In Case0, there is a solution iff slack1* ≥ 0*, slack2* ≥ 0*, and slack3* ≥ 0*. Moreover, the can-have and not-necessary flags should be updated as described in Algorithm 2, Case0.*

We omit the tedious formal proof of this lemma. It can be shown to be correct by examining one-by-one the conditions under which the flags are updated. For example, when all three slacks are non-negative, we have a solution. In particular, this solution can always use elements of U1only without violating the `atmost1` constraint; we therefore set U1only.can-have to True. Similarly, since we are in Case0, the solution will not use elements of U1L2, and we can mark this set as not-necessary. Next, when, say, slack1 is strictly positive, this means that there are more than the minimum required elements (k_1) available for S_1, and therefore some of the shared elements are free to be used by S_2. This translates into turning both U2U1.can-have and U1U2.not-necessary to True, i.e, S_2 can use some elements of U2U1 and S_1 does not need all elements of U1U2. Finally, when we also have slack3 strictly positive, we can deduce that even the corresponding U1only or U2only set is not necessary in its entirety.

Filtering for other cases. When we are not in Case0, i.e., we are considering possible solutions in which S_1 and S_2 do share an element x, x can come from one of L1L2, U1L2, U2L1, U1U2, and U2U1. Note that, as observed earlier, the elements within these five sets are indistinguishable so that x can be taken as any representative element from these sets. If |L1L2| = 1, x must come from L1L2. Otherwise (when |L1L2| = 0), x can come from either of the other four sets. For each of these possibilities, we compute k_1, k_2, slack1, slack2, and slack3 taking the source of x into account, and look for a solution with cardinalities $c_1 - 1$ and $c_2 - 1$ from the remaining elements while exploiting the fact that no more elements can be shared (so that the reduced sub-problem is in Case0). If a solution is determined to exist, we update the can-have and not-necessary flags as in Case0, and in addition also set to True the can-have flag for the set x came from.

We have the following property, which is crucial for the correctness of the above filtering mechanism for bounds consistency:

Lemma 2. *For each of the 9 sets T, the following holds: T should be removed from the upper bound of the appropriate $S_i, i \in \{1, 2\}$, iff T.can-have is False*

```
BC-FilterPairAtmost1(S_1, S_2, c_1, c_2)
begin
    Scan L(S_1), U(S_1), L(S_2), and U(S_2), and compute the cardinality of and a
    representative element from each of the 9 sets:
        L1only, L2only, L1L2, U1only, U2only,
        U1L2, U2L1, U1U2, U2U1
    Initialize all can-have and not-necessary flags to False
    if |L1L2| > 1 then Fail
    if |L1L2| = 1 then
        Let x be the (representative) element of L1L2; share x
        Perform BC-Case0 on (S_1 \ {x}, S_2 \ {x}, c_1 − 1, c_2 − 1)
        Perform BC-UpdateDomains
        Return
    // |L1L2| = 0
    for each s ∈ { U1L2, U2L1, U1U2, U2U1 } do
        // possible solution has a shared element from s; use representative x
        Perform BC-Case0 on (S_1 \ {x}, S_2 \ {x}, c_1 − 1, c_2 − 1)
        if all three slack conditions are non-negative then
            s.can-have ← True
    Perform BC-UpdateDomains
end

sub BC-Case0(S_1, S_2, c_1, c_2)
begin
    k_1 ← c_1 − (|L1only| + |L1L2| + |U2L1|)
    k_2 ← c_2 − (|L2only| + |L1L2| + |U1L2|)
    slack1 ← (|U1only| + |U1U2|) − k_1
    slack2 ← (|U2only| + |U2U1|) − k_2
    slack3 ← (|U1only| + |U2only| + |U1U2|) − (k_1 + k_2)
    if (slack1 ≥ 0) and (slack2 ≥ 0) and (slack3 ≥ 0) then
        // solution exists
        U1only.can-have ← True; U2only.can-have ← True
        U1L2.not-necessary ← True; U2L1.not-necessary ← True
        if slack1 > 0 then
            U2U1.can-have ← True; U1U2.not-necessary ← True
            if slack3 > 0 then U1only.not-necessary ← True
        if slack2 > 0 then
            U1U2.can-have ← True; U2U1.not-necessary ← True
            if slack3 > 0 then U2only.not-necessary ← True
end

sub BC-UpdateDomains
begin
    for each s in the 9 sets do
        if s.can-have = False or s.not-necessary = False then
            for all y ∈ s computed by re-scanning L(S_1), U(S_1), L(S_2), U(S_2) do
                if s.can-have = False then Remove y from U(S_i) for correct i
                if s.not-necessary = False then Add y to L(S_i) for correct i
end
```

Algorithm 2: Filtering `pair-atmost1`. Case0 shown here in detail.

at the end, and T should be added to the lower bound for the appropriate S_i iff T.not-necessary is False at the end.

We again omit a formal proof of this lemma and only mention that it follows by recognizing that the algorithm implicitly goes through every possible solution of the problem instance so that the can-have and not-necessary flags have their intended semantic meaning.

Theorem 1. *Algorithm 2 establishes bounds consistency on the* `pair-atmost1` *constraint.*

The time complexity of BC-FilterPairAtmost1 is dominated entirely by the creation of the 9 sets during search. Computing the cardinalities of and a representative from these sets takes time $O(n)$ where n is the integer domain size. The rest of the algorithm has only a constant number of calls to BC-Case0 and one call to BC-UpdateDomains. Notice that BC-Case0 itself runs in constant time; all it does is process a constant number of flags based on the pre-computed cardinalities of 9 sets. Since we do not maintain all elements of the 9 sets throughout, the call to BC-UpdateDomains at the end re-creates and filters these elements for any set whose can-have or not-necessary flag is still False. This takes time $O(n + k \log n)$, where k is the number of elements removed from an upper bound or added to a lower bound, assuming standard set operations used for maintaining these upper and lower bounds take time $O(\log n)$.

We can tighten this analysis by amortizing over an entire path in the search tree from the root to any leaf. Observe that we can add to either lower bound and remove from either upper bound at most n elements in an entire path, so that the total filtering complexity is $O(n \log n)$ for the path. Similarly, the filtering algorithm can be called at most n times due to each of S_1 and S_2, so that updating the flags takes total time $O(n)$ for the path. Finally, by exploiting domain-deltas for S_1 and S_2 and computing not only the cardinalities and representatives of the 9 sets but rather the complete sets, we can similarly reduce the computation time for generating the 9 sets to $O(n \log n)$ per path, since the time for this is $O(|\Delta| \log n)$ per call (assuming $O(\log n)$ time set operations) and the Δ's sum to $O(n)$. This last part requires a way to "remember" the elements of the 9 sets from the parent node so that their cardinalities can be updated by only looking at domain-deltas of S_1 and S_2. We note that we used ILOG Solver 6.3 for our implementation of this algorithm. Unfortunately, currently the solver does not provide complete access to the domain-deltas for both S_1 and S_2 simultaneously. Moreover it does not support 'RevIntSet' types, which would be needed to maintain and update the 9 sets incrementally as discussed above.

5 Experimental Results

We now discuss computational results for the `sum-free` constraint and the `pair-atmost1` constraint. All our models were implemented in ILOG Solver 6.3,

and all experiments run on a 3.8 GHz Intel Xeon machine with 2 GB memory running Linux 2.6.9-22.ELsmp.

5.1 The Schur Problem

To evaluate the performance of the `sum-free` constraint, we applied it to solve *Schur* problems. Given an integer $k \geq 0$, the *Schur number* of k is the largest integer n for which we can partition the set $\{1, 2, \ldots, n\}$ into k sum-free sets. The related decision problem is: given two integers $k, n \geq 0$, does there exist a partition of $\{1, 2, \ldots, n\}$ into k sum-free sets? We next present two constraint programming models for the decision version.

First model: We introduce n integer variables $x_i, 1 \leq i \leq n$, representing the subset in which element i is placed. Thus, $D(x_i) = \{1, \ldots, k\}$. To ensure that the subsets are sum-free, we add the constraints

$$(x_i = s) \wedge (x_j = s) \;\Rightarrow\; (x_{i+j} \neq s),$$

for all subsets $s = 1, \ldots, k$, and elements $1 \leq i \leq j \leq n$ such that $i + j \leq n$. Note that there are $O(kn^2)$ such constraints.

Second model: We introduce k set variables S_i representing the sum-free subsets ($i = 1, \ldots, k$). Thus, $D(S_i) = [\varnothing, \{1, \ldots, n\}]$. To ensure that the subsets are sum-free, we add the constraints

$$\texttt{sum-free}(S_i)$$

for all subsets $i = 1, \ldots, k$. In this way, we just add k such constraints.

These two different representations achieve the same amount of filtering at each node of the search tree. However, for large n, the number of constraints in the integer representation often becomes an issue, which we hope to circumvent using the set representation.

From a search perspective, we observed that the representation using integer variables (the first model) provides a better handle to tune the search strategy. During our experiments we found that the best search strategy is to branch first on the integer variable with the smallest domain, breaking ties in favor of the variable with the smallest maximum domain value, and assigning the smallest domain value first. Hence, in our set model (the second model), we also apply an integer representation of the set variables, in order to benefit from the same effective search strategy. We note that we only use the integer variables, and not the corresponding constraints.

We compare the performance of the two models on a number of Schur problem instances; see Table 1. As the two models achieve the same amount of filtering, they always use same number of fails (backtracks). However, the set representation using `sum-free` constraints does so much more efficiently (when the number of fails is larger than 2). For example, `schur-7-750` cannot be solved using the

Problem	Integer Model			Set Model		
	time	memory	fails	time	memory	fails
schur-5-123	60.7	98 MB	9,520	14.6	22 MB	9,520
schur-5-135	137.9	112 MB	17,550	35.8	22 MB	17,550
schur-5-139	3559.1	117 MB	415,705	980.8	23 MB	415,705
schur-6-300	3.8	552 MB	75	1.8	29 MB	75
schur-6-306	75.3	572 MB	3,050	17.9	30 MB	3,050
schur-7-500	3.6	1,762 MB	2	12.4	40 MB	2
schur-7-550	–	> 2 GB	–	16.8	42 MB	2
schur-7-750	–	> 2 GB	–	38.7	51 MB	54
schur-8-500	4.3	1,993 MB	2	16.2	43 MB	2
schur-8-550	–	> 2 GB	–	22.6	45 MB	2
schur-8-1000	–	> 2 GB	–	179.4	67 MB	2
schur-8-1400	–	> 2 GB	–	563.8	89 MB	3

Table 1. *Computational results on Schur problems. Instance* schur-k-n *asks to partition* $\{1, \ldots, n\}$ *into* k *sum-free subsets. Time is in seconds.*

integer representation within a time limit of 30 minutes (in fact, it requires more than 2 GB of memory to even run), while the set representation solves this problem in less than 40 seconds. When the number of backtracks is small, we don't benefit as much from the incremental set representation, in which case the integer representation can be more efficient (schur-7-500 and schur-8-500). However, in general, the high memory requirements (> 2 GB) of the integer model prevented it from being used on any problem instance with $n \geq 550$, while the set representation never needed more than 100 MB even for the largest instances. For smaller instances, such as schur-5-xxx, the set representation reduced the runtime by roughly a factor of four.

Overall, these results demonstrate that a set representation can not only be convenient from a modeling perspective, but also helpful from a computational perspective.

5.2 The Social Golfer Problem

We evaluated the performance of the pair-atmost1 constraint on the well-known social golfer problem (problem prob010 in CSPLib). The problem golf-g-s-w asks for a partition of n golfers into g groups, each of size s, for w weeks, such that no two golfers are in the same group more than once throughout the whole schedule. The problem was originally posted (on sci.op.research in May 1998) for 32 golfers, to be divided over 8 groups of size 4 over 10 weeks, i.e., problem instance golf-8-4-10. The social golfer problem has received much attention over the years in the constraint programming community, especially as a benchmark set for symmetry breaking techniques [2, 12]. As mentioned earlier, our work focuses on better filtering algorithms for this problem, orthogonal to the symmetry

exploitation approaches. While the original problem with 8 golfers over 10 weeks has been solved analytically [1], it has never been solved computationally to the best of our knowledge, making this family of problems a challenging benchmark.

We model this problem for constraint programming using set variables, following Smith [16]. For each week $i = 1, \ldots, w$ and each group $j = 1, \ldots, g$, we introduce a set variable S_{ij} representing the golfers that appear in this group. Thus, $D(S_{ij}) = [\varnothing, \{1, \ldots, n\}]$. The cardinality of each group is restricted to be s by the constraint $|S_{ij}| = s$. To ensure that in each week i the groups partition the set of golfers, we state for each $i \in \{1, 2, \ldots, w\}$ the constraint

$$\texttt{partition}(S_{i1}, \ldots, S_{ig}, \{1, \ldots, n\}),$$

which is readily available in the ILOG library. To ensure that each pair of golfers meet at most once, we apply the `atmost1` constraint on pairs of groups:

$$\texttt{atmost1}(S_{ij}, S_{kl}, s, s),$$

for $1 \leq i < k \leq w, 1 \leq j \leq g, 1 \leq l \leq g$. Together, these constraint are sufficient to model the problem. To this model we can apply the standard enumeration strategy on the set variables that is available in ILOG Solver.

As in the previous section, we experimented with a different search strategy based on an additional integer representation of the set variables. For each week $i = 1, \ldots, w$ and each golfer $j = 1, \ldots, n$, we introduce an integer variable x_{ij} representing the group in which golfer j plays in week i. Thus, $D(x_{ij}) = \{1, \ldots, g\}$. We implemented a second search strategy, based on these variables, choosing the variable with the smallest domain first and assigning the minimum value from its domain first. The integer representation furthermore allows us to apply a redundant *global cardinality constraint* [13] with respect to the partitioning of the golfers into groups. For each week i, we state that each group must be assigned to exactly s x_{ij}, by using a global cardinality constraint:

$$\texttt{gcc}(x_{i1}, \ldots, x_{in}, \boldsymbol{s}, \boldsymbol{s}),$$

where $\boldsymbol{s}$ denotes the g-tuple $(s, \ldots, s)$. In practice, this constraint does not add much overhead and is able to filter some additional values, provided that we branch on the integer variables. Hence we always apply it when using the search strategy on integer variables. In case we branch on the set variables, this constraint is not effective.

The social golfer problem is notoriously difficult due to the many symmetries. To account for some symmetry-breaking, we partly instantiate some of the set variables before starting the search, following Fahle et al. [7] (see also [12]). For the first week, we simply assign the groups by increasing order of the golfers, i.e., the first s golfers in group 1, the next s golfers in group 2, and so on. The s golfers of the first group in week 1 are then divided over the first s groups of all other weeks in increasing order. Finally, we assign the first group of the second week to its lower bound, being the first players in the first s groups of week 1. Although this static symmetry-breaking improved the performance significantly,

Problem	Standard Decomposition (partial filtering)		BC-FilterPairAtmost1 (bounds consistency)	
	time	**fails**	**time**	**fails**
golf-6-5-5	2106.7	10,986,224	75.5	239,966
golf-6-5-4	1517.7	10,930,370	39.7	197,837
golf-6-5-3	1060.5	10,930,016	29.6	197,607
golf-6-5-2	635.5	10,879,368	17.2	171,664
golf-8-4-4	226.7	1,555,561	157.7	738,393
golf-10-3-10	128.1	150,911	67.2	78,976
golf-10-3-9	86.0	150,452	52.4	78,613
golf-10-3-6	21.3	110,429	17.3	57,364
golf-10-3-3	0.02	50	0.02	6
golf-10-4-5	51.3	310,110	4.5	22,044
golf-10-4-4	42.5	310,109	4.0	22,043
golf-7-4-4	22.5	184,641	4.4	27,877
golf-7-3-2	0.01	48	0.01	5
golf-9-4-4	7.8	59,331	1.6	6,204
golf-9-4-3	4.7	50,468	0.3	1,853
golf-6-4-5	5.9	35,870	0.6	3,326
golf-6-4-4	4.5	35,832	0.5	3,299
golf-5-4-4	0.3	2,313	0.07	266

Table 2. *Computational results on social golfer instances. Instance* golf-*g*-*s*-*w* *asks to schedule g groups of size s over w weeks. Runtime is in seconds.*

it was not sufficient to close any open problem with our additional filtering. However, our filtering algorithm can be applied to any model, including those with more advanced symmetry-breaking techniques.

We evaluated the performance of the standard decomposition implementation of pair-atmost1 (achieving partial filtering) with our filtering algorithm BC-FilterPairAtmost1 (achieving bounds consistency) on a number of instances. The results are reported in Table 2. The results demonstrate that using the bounds consistency algorithm, one can solve many instances 5 to 50 times faster, with a similar reduction in the number of fails. For example, on the instance golf-6-5-5 with 6 groups of size 5 each to be scheduled for 5 weeks, the runtime is decreased from more than half an hour to 76 seconds and the number of fails reduces from nearly 11 million to 250 thousand, a 40-fold reduction. There are, of course, other instances where using BC-FilterPairAtmost1 does not pay off significantly, although in our experiments, using this algorithm almost never hurt the performance.

6 Conclusion

We studied two constraints on set variables: the `sum-free` constraint and the `atmost1` constraint on pairs of set variables with known cardinality. For both constraints we introduced efficient domain filtering algorithms, establishing bounds consistency. Experimental results on the Schur problem and the Social Golfer Problem showed that these constraints not only offer convenience in modeling, but also help in solving combinatorial problems more efficiently in terms of runtime and memory requirements.

Acknowledgments

We thank the reviewers for the many constructive comments. This research was partly supported by the Intelligent Information Systems Institute, Cornell University under AFOSR Grant FA-9550-04-1-0151.

References

[1] A. Aguado. A 10 days solution to the social golfer problem, 2004.
[2] N. Barnier and P. Brisset. Solving Kirkman's schoolgirl problem in a few seconds. *Constraints*, 10(1):7–21, 2005.
[3] N. Beldiceanu, M. Carlsson, and J. Rampon. Global constraint catalog, 2005. http://www.emn.fr/x-info/sdemasse/gccat/.
[4] C. Bessiere, E. Hebrard, B. Hnich, and T. Walsh. Disjoint, partition and intersection constraints for set and multiset variables. In *CP'04*, volume 3258 of *LNCS*, pages 138–152, 2004.
[5] G. Dooms and I. Katriel. The *minimum spanning tree* constraint. In *CP'06*, volume 4204 of *LNCS*, pages 152–166, 2006.
[6] Eclipse. URL `http://www.eclipse-clp.org`.
[7] T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In *CP'01*, volume 2239 of *LNCS*, pages 93–107, 2001.
[8] H. Fredricksen and M. Sweet. Symmetric sum-free partitions and lower bounds for Schur numbers. *Electronic J. Combinatorics*, 7(1):R32, 1–9, 2000.
[9] Gecode. URL `http://www.gecode.org`.
[10] C. Gervet. Constraints over structured domains. In Rossi et al. [14], chapter 17.
[11] ILOG Solver. URL `http://www.ilog.com/products/cp`.
[12] J.-F. Puget. Symmetry breaking revisited. *Constraints*, 10(1):23–46, 2005.
[13] J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *AAAI'96*, volume 1, pages 209–215, 1996.
[14] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.
[15] A. Sadler and C. Gervet. Global reasoning on sets. In *Proc. of Workshop on Modelling and Problem Formulation (FORMUL'01)*, 2001.
[16] B. Smith. Reducing symmetry in a combinatorial design problem. In *CP-AI-OR'01*, 2001.
[17] P. Van Hentenryck, Y. Deville, and C.-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2-3):291–321, 1992.
[18] W.-J. van Hoeve and I. Katriel. Global constraints. In Rossi et al. [14], chapter 6.