# Fast Set Bounds Propagation using BDDs

**Graeme Gange** and **Vitaly Lagoon**
**Department of Computer Science and Software Engineering**
**The University of Melbourne, Vic. 3010, Australia**

**Peter J. Stuckey**
**NICTA Victoria Laboaratory**
**Department of Computer Science and Software Engineering**
**The University of Melbourne, Vic. 3010, Australia**

**Abstract.**

Set bounds propagation is the most popular approach to solving constraint satisfaction problems (CSPs) involving set variables. The use of reduced ordered Binary Decision Diagrams (BDDs) to represent and solve set CSPs is well understood and brings the advantage that propagators for arbitrary set constraints can be built. This can substantially improve solving. The disadvantages of BDDs is that creating and manipulating BDDs can be expensive. In this paper we show how we can perform set bounds propagation using BDDs in a much more efficient manner by generically creating set constraint predicates, and using a marking approach to propagation. The resulting system can be significantly faster than competing approaches to set bounds propagation.

## 1 Introduction

It is often convenient to model a constraint satisfaction problem (CSP) using finite set variables and set relationships between them. A common approach to solving finite domain CSPs is using a combination of a global backtracking search and a local constraint propagation algorithm. The local propagation algorithm attempts to enforce consistency on the values in the domains of the constraint variables by removing values from the domains of variables that cannot form part of a complete solution to the system of constraints. The most common level of consistency is *set bounds consistency* [4] where the solver keeps track for each set of which elements are definitely in or out of the set. Many solvers use set bounds consistency including ECLiPSe, Gecode, and ILOG SOLVER.

Set bounds propagation is supported by solvers since stronger notions of propagation such as domain propagation require representing exponentially large domains of possible values. However, [8] demonstrated that it is possible to use reduced ordered binary decision diagrams (BDDs) as a compact representation of both set domains and of set constraints, thus permitting *set domain* propagation. A domain propagator ensures that every value in the domain of a set variable can be extended to a complete assignment of all of the variables in a constraint. The use of the BDD representation comes with several additional benefits. The ability to easily conjoin and existentially quantify BDDs allows the removal of intermediate variables, thus strengthening propagation, and also makes the construction of propagators for global constraints straightforward.

Given the natural way in which BDDs can be used to model set constraint problems, it is therefore worthwhile utilising BDDs to construct other types of set solver. Indeed it has been previously demonstrated [5, 6] that set bounds propagation can be efficiently implemented using BDDs to represent constraints and domains of variables. A major benefit of the BDD-based approach is that it frees us from the need to laboriously construct set bounds propagators for each new constraint by hand. Moreover, correctness and optimality of such BDD-based propagators follow by construction. The other advantages of the BDD-based representation identified above still apply, and the resulting solver performs very favourably when compared with existing set bounds solvers.

But set bounds propagation using BDDs still constructs BDDs during propagation, which is a considerable overhead. In this paper we show how we can perform BDD-based set bounds propagation using a marking algorithm that perform linear scans of the BDD representation of the constraint without constructing new BDDs. The resulting set bounds propagators are substantially faster than those using BDDs. We can use the same linear pass to detect elements of the set which can make further difference in propagation, and construct a filter on the propagator to prevent invoking it unless one of the variables that can make a difference changes.

To summarize, the benefits of the approach of this paper are:

- efficiency, no new BDDs are constructed during propagation, so it is very fast;
- reuse, we can reuse a single BDD for multiple copies of the same constraint, and hence handle larger problems;
- ordering, we are not restricted to a single global ordering of Booleans for constructing BDDs; and
- filtering, we can keep track of which parts of the set variable can really make a difference, and reduce the amount of propagation.

We illustrate a prototype solver using the approach on well-known set problems, comparing against the state of the art Gecode set bounds propagation solver.

## 2 Preliminaries

Propagation based approaches to solving set constraint problems represent the problem using a domain storing the possible values of each set variable, and propagators for each constraint, that remove values from the domain of a variable that are inconsistent with values for other variables. Propagation is combined with backtracking search to find solutions.

A *domain* $D$ is a complete mapping from the fixed finite set of variables $\mathcal{V}$ to finite collections of finite sets of integers. The *domain of a variable* $v$ is the set $D(v)$. A domain $D_1$ is said to be *stronger* than a domain $D_2$, written $D_1 \sqsubseteq D_2$, if $D_1(v) \subseteq D_2(v)$ for all $v \in \mathcal{V}$. A domain $D_1$ is equal to a domain $D_2$, written $D_1 = D_2$, if $D_1(v) = D_2(v)$ for all variables $v \in \mathcal{V}$. A domain $D$ can be interpreted as the constraint $\bigwedge_{v \in \mathcal{V}} v \in D(v)$.

For set constraints we will often be interested in restricting variables to take on *convex* domains. A set of sets $K$ is *convex* if $a, b \in K$ and $a \subseteq c \subseteq b$ implies $c \in K$. We use interval notation $[a, b]$ where $a \subseteq b$ to represent the (minimal) convex set $K$ including $a$ and $b$. For any finite collection of sets $K = \{a_1, a_2, \ldots, a_n\}$, we define the convex closure of $K$: $conv(K) = [\cap_{a \in x} a, \cup_{a \in x} a]$. We extend the concept of convex closure to domains by defining $ran(D)$ to be the domain such that $ran(D)(x) = conv(D(x))$ for all $x \in \mathcal{V}$.

A *valuation* $\theta$ is a set of mappings from the set of variables $\mathcal{V}$ to sets of integer values, written $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$. A valuation can be extended to apply to constraints involving the variables in the obvious way. Let *vars* be the function that returns the set of variables appearing in an expression, constraint or valuation. In an abuse of notation, we say a valuation is an element of a domain $D$, written $\theta \in D$, if $\theta(v_i) \in D(v_i)$ for all $v_i \in vars(\theta)$.

**Constraints, Propagators and Propagation Solvers** A constraint is a restriction placed on the allowable values for a set of variables. We shall use *primitive set constraints* such as (membership) $k \in v$, (equality) $u = v$, (subset) $u \subseteq w$, (union) $u = v \cup w$, (intersection) $u = v \cap w$, (cardinality) $|v| = k$, (upper cardinality bound) $|v| \leq k$, (lexicographic order) $u < v$, where $u, v, w$ are set variables, $k$ is an integer. We can also construct more complicated constraints which are (possibly existentially quantified) conjunctions of primitive set constraints. We define the *solutions* of a constraint $c$ to be the set of valuations $\theta$ on $vars(c)$ that make the constraint true.

We associate a *propagator* with every constraint. A propagator $f$ is a monotonically decreasing function from domains to domains, so $D_1 \sqsubseteq D_2$ implies that $f(D_1) \sqsubseteq f(D_2)$, and $f(D) \sqsubseteq D$. A propagator $f$ is *correct* for a constraint $c$ if and only if for all domains $D$: $\{\theta \mid \theta \in D\} \cap solns(c) = \{\theta \mid \theta \in f(D)\} \cap solns(c)$

A *propagation solver* $solv(F, D)$ for a set of propagators $F$ and a domain $D$ repeatedly applies the propagators in $F$ starting from the domain $D$ until a fixpoint is reached. $solv(F, D)$ is the weakest domain $D' \sqsubseteq D$ where $f(D') = D'$ for all $f \in F$.

**Domain and Bounds Consistency** A domain $D$ is *domain consistent* for a constraint $c$ if $D$ is the smalest domain containing all solutions $\theta \in D$ of $c$. We define the *domain propagator* for a constraint $c$ as

$$dom(c)(D)(v) = \begin{cases} \{\theta(v) \mid \theta \in solns(D \wedge c)\} & \text{if } v \in vars(c) \\ D(v) & \text{otherwise} \end{cases}$$

Then $dom(c)(D)$ is always domain consistent with $c$.

A domain $D$ is *(set) bounds consistent* for a constraint $c$ if for every variable $v \in vars(c)$ the upper bound of $D(v)$ is the union of the values of $v$ in all solutions of $c$ in $D$, and the lower bound of $D(v)$ is the intersection of the values of $v$ in all solutions of $c$ in $D$. We define the *set bounds propagator* for a constraint $c$ as

$$sb(c)(D)(v) = \begin{cases} conv(dom(c)(ran(D))(v)) & \text{if } v \in vars(c) \\ D(v) & \text{otherwise} \end{cases}$$

Then $sb(c)(D)$ is always bounds consistent with $c$.

**BDDs** We assume a set $\mathcal{B}$ of Boolean variables with a total ordering $\prec$. We make use of the following Boolean operations: $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation), $\rightarrow$ (implication), $\leftrightarrow$ (bi-implication) and $\exists$ (existential quantification). We denote by $\exists_V F$ the formula $\exists x_1 \cdots \exists x_n F$ where $V = \{x_1, \ldots, x_n\}$, and by $\overline{\exists}_V F$ we mean $\exists_{V'} F$ where $V' = vars(F) \setminus V$.

Reduced Ordered Binary Decision Diagrams are a well-known method of representing Boolean functions on Boolean variables using directed acyclic graphs with a single root. Every internal node $n(v, f, t)$ in a BDD $r$ is labelled with a Boolean variable $v \in \mathcal{B}$, and has two outgoing arcs — the 'false' arc (to BDD $f$) and the 'true' arc (to BDD $t$). Leaf nodes are either $\mathcal{F}$ (false) or $\mathcal{T}$ (true). Each node represents a single test of the labelled variable; when traversing the tree the appropriate arc is followed depending on the value of the variable. Define the size $|r|$ as the number of internal nodes in a BDD $r$, and $VAR(r)$ as the set of variables $v \in \mathcal{B}$ appearing in some internal node in $r$.

Reduced Ordered Binary Decision Diagrams (BDDs) [1] require that the BDD is: *reduced*, that is it contains no identical nodes (that is, nodes with the same variable label and identical then and else arcs) and has no redundant tests (no node has both then and else arcs leading to the same node); and *ordered*, if there is an arc from a node labelled $v_1$ to a node labelled $v_2$ then $v_1 \prec v_2$. A BDD has the nice property that the function representation is canonical up to variable reordering. This permits efficient implementations of many Boolean operations.

BDDs can represent an arbitrary Boolean formula over variables $\mathcal{B}$. We shall be interested in *stick BDD*s where for every internal node $n(v, f, t)$ exactly one of $f$ or $t$ is the constant $\mathcal{F}$ node. Stick BDDs represent exactly the formulae of the form $\bigwedge_{v \in T} v \wedge \bigwedge_{v \in F} \neg v$ where $T$ and $F$ are disjoint subsets of $\mathcal{B}$.

A Boolean variable $v$ is said to be *fixed* in a BDD $r$ if either for every node $n(v, t, e) \in r$ $t$ is the constant $\mathcal{F}$ node, or for every node $n(v, t, e)$ $e$ is the constant $\mathcal{F}$ node. Such variables can be identified in a linear time scan over the domain BDD. For convenience, if $\phi$ is a BDD, we write $[\![\phi]\!]$ to denote the BDD representing the conjunction of the fixed variables of $\phi$. Note $[\![\phi]\!]$ is a stick BDD.

## 3 Set Propagation using BDDs

The key step in building set propagation using BDDs is to realize that we can represent a finite set domain using a BDD.

**Representing domains** If $v$ is a set variable ranging over subsets of $\{1, \ldots, N\}$, then we can represent $v$ using the Boolean variables $V(v) = \{v_1, \ldots, v_N\} \subseteq \mathcal{B}$, where $v_i$ is true iff $i \in v$. We will order the variables $v_1 \prec v_2 \cdots \prec v_N$. We can represent a valuation $\theta$ using a formula

$$R(\theta) = \bigwedge_{v \in vars(\theta)} \left( \bigwedge_{i \in \theta(v)} v_i \wedge \bigwedge_{i \in \{1, \ldots, N\} - \theta(v)} \neg v_i \right).$$

Then a domain of variable $v$, $D(v)$ can be represented as $\bigvee_{a \in D(v)} R(\{v \mapsto a\})$. This formula can be represented by a BDD.

**Representing constraints** We can similarly model any set constraint $c$ as a BDD $B(c)$ using the Boolean variable representation $V(v)$ of its set variables $v$. By ordering the variables in each BDD carefully we can build small representations of the formulae. The *pointwise* order of Boolean variables is defined as follows. Given set variables $u \prec v \prec w$ ranging over sets from $\{1, \ldots, N\}$ we order the Boolean variables as $u_1 \prec v_1 \prec w_1 \prec u_2 \prec v_2 \prec w_2 \prec \cdots u_n \prec v_n \prec w_n$.

The representation $B(c)$ is simply $\vee_{\theta \in solns(c)} R(\theta)$. For primitive set constraints (using the pointwise order) this size is linear in $N$. For more details see [6]. The BDD representation of $x = y \cup z$ is shown in Figure 2(a).

**BDD-based Set Bounds Propagation** We can build a set bounds propagator, more or less from the definition, since we have BDDs to represent domains and constraints.

$$\begin{aligned} \phi &= B(c) \wedge \bigwedge_{v' \in vars(c)} D(v') \\ sb(c)(D)(v) &= \overline{\exists}_{V(v)} [\![\phi]\!] \end{aligned}$$

We simply conjoin the domains to the constraint obtaining $\phi$, then extract the fixed variables from the result, and then project out the relevant part for each variable $v$. The set bounds propagation can be improved by removing the fixed variables as soon as possible. The improved definition is given in [5]. Overall the complexity can be made $O(|B(c)|)$.

The updated set bounds can be used to simplify the BDD representing the propagator. Since fixed variables will never interact further with propagation they can be projected out of $B(c)$, so we can replace $B(c)$ by $\exists_{VAR([\![\phi]\!])} \phi$.

## 4 Faster Set Bounds Propagation

While set bounds propagation using BDDs is much faster than set domain propagation and often better than set domain propagation (or other variations of propagations for sets) it still creates new BDDs. This is not necessary as long as we are prepared to give up the simplifying of BDDs that is possible in set bounds propagation.

We do not represent domains of variables as BDD sticks, but rather as arrays of integer values. A domain $D$ is an array where, for variable $v$ ranging over subsets of $\{1, \ldots, N\}$: $D[v_i] = 0$ indicates $i \notin v$, $D[v_i] = 1$ indicates $i \in x$, and $D[v_i] = 2$ means we don't know whether $i$ is in or not in $v$. Hence $D(v) = [\{i | D[v_i] = 1\}, \{i | D[v_i] \neq 0\}]$.

The BDD representation of a constraint $B(c)$ is built as before. A significant difference is that since constraints only communicate through the set bounds of variables we do not need them to share a global variable order hence we can if necessary modify the variable order used to construct $B(c)$ for each $c$, or use automatic variable reordering (which is available in most BDD packages) to construct $B(c)$. Another advantage is that we can reuse the BDD for a constraint $c(\bar{x})$ on variables $\bar{x}$ for the constraint $c(\bar{y})$ on variables $\bar{y}$ (as long as they range over the same initial sets), that is, the same constraint on different variables. Hence we only have to build one such BDD, rather than one for each instance of the constraint.

The set bounds propagator $sb(c(\bar{x})$ for constraint $c(\bar{x})$ is now implemented as follows. A generic BDD representation $r$ of the constraint $c(\bar{y})$ is constructed. The propagator copies the domain description of the actual parameters $x^1, \ldots, x^n$ onto a domain description $E$ for formal parameters $y^1, \ldots, y^n$. It constructs an array $E$ where $E[y_i^j] = D[x_i^j]$. Let $V = \{y_i^j \mid 1 \leq j \leq n, 1 \leq i \leq N\}$ be the set of Boolean variables occurring in the constraint $c(\bar{y})$. The propagator executes the code $\mathsf{bddprop}(r, V, E)$ shown in Figure 1 which returns $(r', V', E')$. If $r' = \mathcal{F}$ the propagator returns a false domain, otherwise the propagator copies back the domains of the formal parameters to the actual parameters so $D[x_i^j] = E[y_i^j]$. We will come back to the $V$ argument in the next subsection.

The procedure $\mathsf{bddprop}(r, V, E)$ traverses the BDD $r$ as follows. We visit each node $n(v, f, t)$ in the BDD in a top-down memoing manner. We record if, under the current domain, the node can reach the $\mathcal{F}$ node, and if it can reach the $\mathcal{T}$ node. If the $f$ child can reach the $\mathcal{T}$ node we add support for the variable $v$ taking value 0. Similarly if the $t$ child can reach $\mathcal{T}$ we add support for the variable $v$ taking 1. If the node can reach both $\mathcal{F}$ and $\mathcal{T}$ we record that the variable $v$ matters to the computation of the BDD. After the visit we reduce the variable set for the propagator to those that matter, and remove values with no support from the domain. The procedure assumes a global *time* variable which is incremented between each propagation, which is used to memo the marking phase. The $top(n, V)$ function returns the variable in the root node of $n$ or the largest variable (under $\prec$) in $V$ if $n = \mathcal{T}$ or $n = \mathcal{F}$.

**Example 1** Consider the BDD for the constraint $x = y \cup z$ when $N = 2$ shown in Figure 2(a). Assuming a domain $E$ where $E[y_1] = 1$ ($1 \in y$) and $E[z_2] = 1$ ($2 \in z$), and the remaining variables take value 2, the algorithm traverses the edges shown with double lines in Figure 2(b). No path from $x_1$, or $x_2$ following the $f$ arc reaches $\mathcal{T}$ hence *alive*$[x_1,0]$ and *alive*$[x_2,0]$ are not marked with the current time. As a result $E[x_1]$ and $E[x_2]$ are set to 1. Hence we have determined $1 \in x$ and $2 \in x$.

Also, no nodes for $z_1$ are actually visited, and the left node for $y_2$ only reaches $\mathcal{F}$ and the right node only reaches $\mathcal{T}$. Hence *matters*$[z_1]$ and *matters*$[y_2]$ are not marked with the current time. The set of $vars$ collected by $\mathsf{bddprop}$ is empty, since the remaining variables are fixed. □

### 4.1 Waking up less often

In practice a bounds propagation solver does not blindly apply each propagator until fixpoint, but keeps track of which propagators must still be at fixpoint, and only executes those that may not be. For set bounds this is usually managed as follows. To each set variable $v$ is attached a list of propagators $c$ that involve $v$. Whenever $v$ changes, these propagators are rescheduled for execution.

We can do better than this with the BDD based propagators. The algorithm $\mathsf{bddprop}$ collects the set of Boolean variables that matter to the BDD, that is can change the result. If a variable is fixed that does not matter, then set bounds propagation cannot learn any new information. We modify the wakeup process as follows. Each variable $x^j$ stores a list of pairs $(f, S)$ of propagator $f$ with the subset $S$ of variables $x_i^j$ which matter to the propagator with the current domain. When the variable changes we traverse the list of propagators and wake those propagators where the change intersects with $S$. On executing a propagator we revise the set $S$ stored in the list for variable $x^j$ to be $\{x_i^j \mid y_i^j \in vars\}$ where $vars$ is the the set of "interesting" variables returned by $\mathsf{bddprop}$. Note the same optimization could be applied to the standard approach, but requires the overhead of computing $VAR(r')$ which here is folded into $\mathsf{bddprop}$.

```
bddprop($r$,$V$,$E$) {
    $(reachf, reacht)$ = bddp($r, V, E$);
    if ($\neg reacht$) return ($\mathcal{F}, \emptyset, E$);
    $vars = \emptyset$;
    for ($v \in V$) {
        for ($d \in \{0,1\}$)
            if ($alive[v,d] < time$) $E[v] = 1 - d$;
        if ($E[v] = 2 \wedge matters[v] \geq time$) $vars = vars \cup \{v\}$; }
    return ($r, vars, E$); }

bddp($node$,$V$,$E$) {
    switch $node$ {
    $\mathcal{F}$: return (1,0);
    $\mathcal{T}$: return (0,1);
    $n(v, f, t)$:
        if ($visit[node] \geq time$) return save[$node$];
        $reachf = 0$; $reacht = 0$;
        if ($E[v] \neq 1$) {
            $(rf0, rt0)$ = bbdp($f, V, E$);
            $reachf = reachf \wedge rf0$;
            $reacht = reacht \wedge rt0$;
            if ($rt0$) {
                for ($v' \in V, v \prec v' \prec top(f, V)$})
                    $alive[v',0] = alive[v',1] = time$;
                $alive[v,0] = time$; } }
        if ($E[v] \neq 0$) {
            $(rf1, rt1)$ = bbdp($t, V, E$);
            $reachf = reachf \wedge rf1$;
            $reacht = reacht \wedge rt1$;
            if ($rt1$) {
                for ($v' \in V, v \prec v' \prec top(t, V)$})
                    $alive[v',0] = alive[v',1] = time$;
                $alive[v,1] = time$; } }
        if ($reachf \wedge reacht$) $matters[v]$ = time;
        save[$node$] = ($reachf, reacht$);
        $visit[node] = time$;
        return ($reachf, reacht$); } }
```

**Figure 1.** Pseudo-code for BDD propagation.

## 5 Experimental Results

We have built a prototype set bounds solver implementing the algorithms described. Currently a Prolog engine takes the definition of the problem, and used an interface to the BDD package CUDD [10] to construct the generic BDDs. It then creates a C file for backtracking solver with data structures for the BDDs. This prototype is very expensive in terms of compilation time, ranging from 0.36–4.65s for Steiner and 0.52–2.42s for golfers, but the actual BDD creation time is a tiny proportion of this, at most 30ms and usually unmeasurable (0ms). In a direct implementation the compilation time will effectively shrink to the BDD creation time.

Experiments were conducted on a 2.66GHz Core2 Duo with 2 Gb of RAM running Ubuntu GNU/Linux 7.04. We compare against the state of the art set bounds propagators of Gecode 2.0 [3].

**Steiner Systems** A commonly used benchmark for set constraint solvers is the calculation of small Steiner systems. A Steiner system $S(t, k, N)$ is a set $X$ of cardinality $N$ and a collection $C$ of subsets of $X$ of cardinality $k$ (called 'blocks'), such that any $t$ elements of $X$ are in exactly one block. Any Steiner system must have exactly $m = \binom{N}{t}/\binom{k}{t}$ blocks (Theorem 19.2 of [9]).

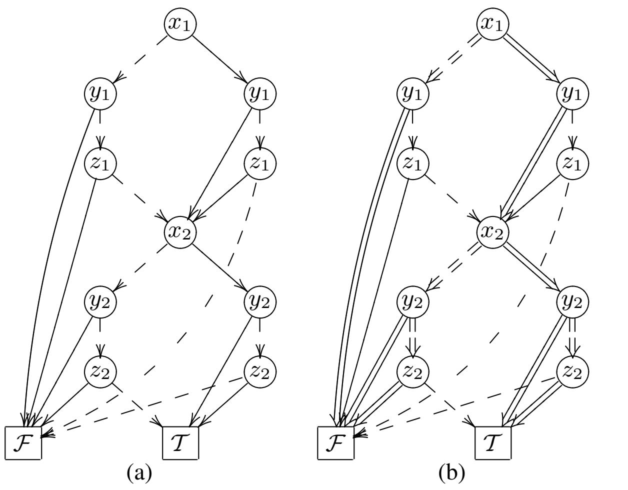

**Figure 2.** (a) The BDD representing $x = y \cup z$ where $N = 2$. A node $n(v, f, t)$ is shown as a circle around $v$ with a dashed arrow to $f$ and full arrow to $t$. (b) The edges traversed by bddprop, when $E[y_1] = 1$ and $E[z_2] = 1$ and $E[v] = 2$ otherwise, are shown doubled.

We use the same modelling of the problem as [8], extended for the case of more general Steiner Systems. We model each block as a set variable $s_1, \ldots, s_m$, with the constraints:

$$\bigwedge_{i=1}^{m} (|s_i| = k) \ \wedge$$

$$\bigwedge_{i=1}^{m-1} \bigwedge_{j=i+1}^{m} (\exists u_{ij}.u_{ij} = s_i \cap s_j \wedge |u_{ij}| \leq (t-1)) \wedge (s_i < s_j)$$

To compare the raw performance of the bounds propagators we performed experiments using a model of the problem with primitive constraints and intermediate variables $u_{ij}$ directly as shown above equivalent to the Gecode model. The results are shown in "Split Constraints" section of Table 1. Gecode has slightly better search behaviour than our solver because its set bounds propagators take into account cardinality information. Clearly the raw propagation speed of the BDD solver is better than Gecode except for the case where the $N$ is large. Note that the BDD solver of [6] cannot handle the largest four Steiner problems with split constraints, because there are too many Boolean variables for the BDD package.

Of course, the BDD representation permits us to merge primitive constraints and remove intermediate variables, allowing us to model the problem as $\binom{m}{2}$ binary constraints (containing no intermediate variables $u_{ij}$) corresponding to second line above conjoined with the cardinality constraints for $s_i$ and $s_j$. Results for this improved model are shown in the "Merged Constraints" section of Table 1. Here the search is reduced and propagation speed usually significantly increased, though filtering is less beneficial.

**Social Golfers** Another common set benchmark is the "Social Golfers" problem, which consists of arranging $N = g \times s$ golfers into $g$ groups of $s$ players for each of $w$ weeks, such that no two players play together more than once. Again, we use the same model as [8], using a $w \times g$ matrix of set variables $v_{ij}$ where $1 \leq i \leq w$ and $1 \leq j \leq g$. Gecode is restricted to use separate constraints, while the BDD solver uses merged constraints.

**Table 1.** Performance results on Steiner Systems: first solution (F) and all solutions (A). Time in seconds for 1000 runs (first solution problems) and one run (all solutions) and number of failures are given for Gecode and the BDD solver for split constraints and the BDD solver for merged constraints. Two times for the BDD set bounds solver are shown: *time* without filtering and *time+f* with filtering. A first-fail "element-in-set" labelling strategy is used in all cases. "—" denotes failure to complete a test case within 240 minutes. $\times$ denotes a case where our naive trailing implementation for filtering runs out of space.

| Problem | | Gecode | | Split Constraints | | | Merged Constraints | | |
|---|---|---|---|---|---|---|---|---|---|
| | | time | fails | time | time+f | fails | time | time+f | fails |
| S(2,3,7) | F | 0.41 | 2 | 0.30 | 0.24 | 2 | 0.12 | **0.11** | 0 |
| S(3,4,8) | F | 5.43 | 14 | 1.70 | 1.56 | 14 | 0.96 | **0.90** | 2 |
| S(2,3,9) | F | 47.98 | 395 | 37.04 | 14.50 | 542 | **5.05** | 5.69 | 121 |
| S(2,4,13) | F | 3.33 | 4 | 3.58 | **1.98** | 4 | 2.24 | 2.17 | 2 |
| S(2,3,15) | F | 19.86 | 6 | 29.01 | 16.54 | 6 | **15.61** | 15.66 | 3 |
| S(3,4,16) | F | 1688.81 | 90 | **431.53** | $\times$ | 90 | 474.22 | 530.52 | 58 |
| S(2,5,21) | F | 14.97 | 4 | 20.50 | 19.68 | 4 | **19.38** | 19.68 | 3 |
| S(3,6,22) | F | 495.9 | 118 | 271.14 | **243.27** | 142 | 554.44 | 668.509 | 96 |
| S(2,3,31) | F | **1098.82** | 14 | 1659.71 | 1891.04 | 14 | 1198.95 | 1301.64 | 11 |
| S(2,3,7) | A | 0.27 | $6.10\times10^3$ | 0.29 | 0.22 | $1.17\times10^4$ | **0.01** | 0.02 | $1.07\times10^3$ |
| S(3,4,8) | A | 1018.84 | $6.36\times10^6$ | 10108.89 | 7875.12 | $1.44\times10^7$ | **58.93** | 58.95 | $4.32\times10^5$ |
| S(2,3,9) | A | 2593.03 | $3.15\times10^7$ | — | — | — | **287.05** | 324.10 | $8.81\times10^6$ |

**Table 2.** First-solution performance results on the Social Golfers problem. Time in seconds for 100 runs and number of failures are given for both solvers. A first-fail "element-in-set" labelling strategy is used in all cases.

| Problem | Gecode | | Merged Constraints | | |
|---|---|---|---|---|---|
| | time | fails | time | time+f | fails |
| 2-5-4 | 0.33 | 14 | 0.21 | **0.14** | 30 |
| 2-6-4 | 7.71 | 860 | 5.77 | **2.55** | 2036 |
| 2-7-4 | 34.30 | 2935 | 19.58 | **8.9** | 4447 |
| 3-5-4 | 0.81 | 14 | 0.46 | **0.44** | 30 |
| 3-6-4 | 18.57 | 863 | 22.91 | **11.82** | 2039 |
| 3-7-4 | 93.42 | 2974 | 64.06 | **41.77** | 4492 |
| 4-5-4 | 0.65 | 1388 | 0.30 | **0.26** | 2886 |
| 4-6-5 | 225.92 | 5355 | 298.43 | **209.22** | 12747 |
| 4-7-4 | 142.58 | 2979 | 137.80 | **103.25** | 4498 |
| 4-9-4 | 10.52 | 54 | 7.8 | **5.49** | 71 |
| 5-5-4 | 149.73 | 2495 | 50.73 | **28.29** | 2758 |
| 5-7-4 | 308.61 | 3062 | 218.58 | **190.9** | 4582 |
| 5-8-3 | 5.07 | 10 | 3.29 | **2.36** | 14 |
| 6-5-3 | 102.84 | 1621 | 35.93 | **17.05** | 1615 |
| 6-6-3 | 3.06 | 4 | 1.74 | **1.23** | 5 |

Experimental results are shown in Table 2. Interestingly the merged constraints here are not enough to compete with the set bounds propagators of Gecode that include cardinality considerations. Not withstanding the greater search space, the BDD set solver is still substantially faster than Gecode. For these examples filtering is always beneficial, sometimes 2 times faster. If we compare against the BDD solver of [6] on these examples, our new solver is around 30 times faster (although the machines used are not identical).

## 6 Related Work

BDD based set solvers were introduced by [8] originally for domain propagation, and then extending to bounds, split and lex and cardinality propagation [6]. The combination of BDD based set bounds propagation with nogoods was introduced in [7]. Another approach automatically constructing set bounds propagators is defined in [12].

A similar approach to using BDDs in propagation was previously defined for solving SAT problems in [2]. This approach informally defines a marking approach to BDD propagation, but does not consider sets, generic constraints, or filtering.

## 7 Conclusion

In this paper we have improved the BDD-based technique of set bounds propagation. The traversal approach to propagation we presented is at least an order of magnitude faster than the previous technique utilizing BDD operations. The prototype implementation of our method is significantly faster than the state of the art set constraint solver of Gecode. As demonstrated by [7], further improvements in the solver performance can be straightforwardly achieved by incorporating nogoods generation [11].

## REFERENCES

[1] Randal E. Bryant, 'Graph-based algorithms for Boolean function manipulation', *IEEE Trans. Comput.*, **35**(8), 677–691, (1986).

[2] R.F. Damiano and J.H. Kukula, 'Checking satisfiability of a conjunction of BDDs', in *Proceedings of Design Automation Conference*, pp. 818–823, (2003).

[3] Gecode. www.gecode.org. Accessed Jan 2008.

[4] Carmen Gervet, 'Interval propagation to reason about sets: Definition and implementation of a practical language', *Constraints*, **1**(3), 191–246, (1997).

[5] P. Hawkins, V. Lagoon, and P.J. Stuckey, 'Set bounds and (split) set domain propagation using ROBDDs', in *17th Australian Joint Conference on Artificial Intelligence*, volume 3339 of *LNCS*, pp. 706–717, (2004).

[6] P. Hawkins, V. Lagoon, and P.J. Stuckey, 'Solving set constraint satisfaction problems using ROBDDs', *Journal of Artificial Intelligence Research*, **24**, 106–156, (2005).

[7] P. Hawkins and P.J. Stuckey, 'A hybrid BDD and SAT finite domain constraint solver', in *Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages*, volume 3819 of *LNCS*, pp. 103–117, (2006).

[8] V. Lagoon and P.J. Stuckey, 'Set domain propagation using ROBDDs', in *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *LNCS*, pp. 347–361, (2004).

[9] J. H. van Lint and R. M. Wilson, *A Course in Combinatorics*, Cambridge University Press, 2nd edn., 2001.

[10] Fabio Somenzi. CUDD: Colorado University Decision Diagram package. Accessed May 2004. http://vlsi.colorado.edu/ fabio/CUDD/.

[11] S. Subbarayan, 'Efficent reasoning for nogoods in constraint solvers with BDDs', in *Proceedings of Tenth International Symposium on Practical Aspects of Declarative Languages*, volume 4902 of *LNCS*, pp. 53–57, (2008).

[12] G. Tack, C. Schulte, and G. Smolka, 'Generating propagators for finite set constraints', in *Twelfth International Conference on Principles and Practice of Constraint Programming*, volume 4204 of *LNCS*, pp. 575–589, (2006).