The SLIDE Meta-Constraint

Christian Bessiere
LIRMM
Montpelier, France
bessiere@lirmm.fr

Emmanuel Hebrard
NICTA and UNSW
Sydney, Australia
e.hebrard@nicta.com.au

Brahim Hnich
Izmir University of
Economics, Turkey
brahim.hnich@ieu.edu.tr

Zeynep Kiziltan
DEIS
Univ. di Bologna, Italy
zkiziltan@deis.unibo.it

Toby Walsh
NICTA and UNSW
Sydney, Australia
tw@cse.unsw.edu.au

Abstract

We study the SLIDE meta-constraint. This slides a constraint down one or more sequences of variables. We show that SLIDE can be used to encode and propagate a wide range of global constraints. We consider a number of extensions including sliding down sequences of set variables, and combining SLIDE with a global cardinality constraint. We also show how to propagate SLIDE. Our experiments demonstrate that using SLIDE to encode constraints can be just as efficient and effective as using specialized propagators.

1 Introduction

In scheduling, rostering and related problems, we often have a sequence of decision variables and a constraint which applies down the sequence. For example, in the car sequencing problem (prob001 in CSPLib), we need to decide the sequence of cars on the production line. We might have a constraint on how often a particular option is met along each sequence (e.g. only 1 out of 3 cars can have the sun-roof option). As a second example, in a nurse rostering problems, we need to decide the sequence of shifts worked by each nurse. We might have a constraint on how many consecutive night shifts any nurse can work. To model such problems, we consider a *meta*-constraint, SLIDE which ensures that a constraint repeatedly holds down a sequence of variables. This is a special case of the previously introduced CARDPATH constraint [Beldiceanu and Carlsson, 2001]. Although SLIDE is very simple, we demonstrate that it is surprisingly powerful. In addition, we describe methods to propagate such constraints, which unlike the previous methods proposed for CARDPATH, can prune all possible values.

The rest of the paper is organised as follows. After presenting the necessary formal background, we introduce the simplest form of the SLIDE meta-constraint. In later sections, we consider a number of generalizations and give examples of global constraints that can be encoded using these various forms of SLIDE. These encodings therefore provide a simple and easy way to implement these global constraints. In most cases, propagating our encoding is as efficient and as effective as a specialized propagator.

2 Background

A constraint satisfaction problem consists of a set of variables, each with a finite domain of values, and a set of constraints specifying allowed combinations of values for some subset of variables. We use capital letters for variables (e.g. X, Y and S), and lower case for values (e.g. d and d_i). We consider both finite domain and set variables. A set variable can be represented by its lower bound which contains the definite elements in the set and an upper bound which contains the definite and potential elements. Constraint solvers typically explore partial assignments enforcing a local consistency property. A constraint is *generalized arc consistent* (*GAC*) iff when a variable is assigned any of the values in its domain, there exist compatible values in the domains of all the other variables. For binary constraints, generalized arc consistency is often called simply arc consistency (AC).

3 SLIDE constraint

We start with the simplest form of the SLIDE meta-constraint. If C is a constraint of arity k then we consider the meta-constraint:

$$\text{SLIDE}(C, [X_1, \ldots, X_n])$$

This holds iff $C(X_i, \ldots, X_{i+k-1})$ itself holds for $1 \leq i \leq n-k+1$. That is, we slide the constraint C down the sequence of variables, X_1 to X_n. This simple form of SLIDE is a special case of the CARDPATH$(N, [X_1, \ldots, X_n], C)$ meta-constraint, which holds iff C holds N times on the sequence $[X_1, \ldots, X_n]$ [Beldiceanu and Carlsson, 2001]. As we shall see, its simple structure will permit us to enforce GAC. Also, we will consider more complex forms of SLIDE that, for instance, slide over multiple sequences or over set variables.

We illustrate this simple form of SLIDE with an example of a global constraint used in car sequencing problems. In Section 11, we discuss how to propagate such encodings into SLIDE. The AMONGSEQ constraint ensures that values occur with some given frequency. For instance, we might want that no more than 3 out of every sequence of 7 shift variables be a "night shift". More precisely, AMONGSEQ$(l, u, k, [X_1, \ldots, X_n], v)$ holds iff between l and u values from the ground set v occur in every k sequence of variables [Beldiceanu and Contejean, 1994]. We can decompose this using a SLIDE; AMONGSEQ$(l, u, k, [X_1, \ldots, X_n], v)$ can be encoded as

SLIDE($D^{k,v}_{l,u}, [X_1, \ldots, X_n]$) where $D^{k,v}_{l,u}$ is an instance of the AMONG constraint [Beldiceanu and Contejean, 1994]. That is, $D^{k,v}_{l,u}(X_i, \ldots, X_{i+k-1})$ holds iff $l \leq \sum_{j=i}^{i+k-1}(X_j \in v) \leq u$.

For example, suppose 2 of every 3 variables along a sequence $X_1 \ldots X_5$ should take the value a, where $X_1 = a$ and $X_2, \ldots, X_5 \in \{a, b\}$. Then we can encode this as SLIDE($E, [X_1, X_2, X_3, X_4, X_5]$) where $E(X_i, X_{i+1}, X_{i+2})$ is an instance of the AMONG constraint that ensures two of its three variables take a. This SLIDE constraint ensures that the following three constraints hold: $E(X_1, X_2, X_3)$, $E(X_2, X_3, X_4)$ and $E(X_3, X_4, X_5)$. Note that each ternary constraint is GAC. However, enforcing GAC on the SLIDE constraint will set $X_4 = a$ as there are only two satisfying assignments for X_1 to X_5 and neither of them have $X_4 = b$.

4 SLIDE over multiple sequences

We often wish to slide a constraint down two or more sequences of variables at once. We therefore consider a more complex form of SLIDE. If F is a constraint of arity $2k$ then:

$$\text{SLIDE}(F, [X_1, \ldots, X_n], [Y_1, \ldots, Y_n])$$

holds iff $F(X_i, \ldots, X_{i+k-1}, Y_i, \ldots, Y_{i+k-1})$ itself holds for $1 \leq i \leq n - k + 1$. We can slide down three or more sequences of variables in a similar way. Note we could view these as syntactic sugar for a SLIDE down a single sequence of variables where the different sequences are interleaved (e.g., $[X_1, Y_1, X_2 \ldots, Y_{n-1}, X_n, Y_n]$) , and the constraint is loosened so it trivially holds if it is applied with the wrong offset. This loosening is direct if all X_i and Y_i have distinct domains (the constraint is satisfied for all tuples starting by a value from Y_i). Otherwise an extra sequence of marking variables S_i with a dummy value can be added, and the constraint sliding on $[S_1, X_1, Y_1, S_2, \ldots, Y_n]$ enforces E on the X_i and the Y_i only when its first argument takes the dummy value.

As an example of sliding down multiple sequences of variables, consider the constraint REGULAR($\mathcal{A}, [X_1, \ldots, X_n]$). This ensures that the values taken by a sequence of variables form a string accepted by a deterministic finite automaton [Pesant, 2004]. This global constraint is useful in scheduling, rostering and sequencing problems to ensure certain patterns do (or do not) occur over time. It can be used to encode a wide range of other global constraints including: AMONG [Beldiceanu and Contejean, 1994], CONTIGUITY [Maher, 2002], LEX and PRECEDENCE [Law and Lee, 2004].

To encode the REGULAR constraint with SLIDE, we introduce finite domain variables, Q_i to record the state of the automaton. We then post the constraint SLIDE($F, [X_1, \ldots, X_{n+1}], [Q_1, \ldots, Q_{n+1}]$) where X_{n+1} is a "dummy" variable, Q_1 is assigned to the starting state of the automaton, Q_{n+1} is restricted to any of the accepting states, and $F(X_i, X_{i+1}, Q_i, Q_{i+1})$ holds iff $Q_{i+1} = \delta(X_i, Q_i)$ where δ is the transition function of the finite automaton. Note that F is independent of its second argument so is effectively ternary. Since the automaton is deterministic, F is also functional on X_i and Q_i. Enforcing GAC on this encoding takes $O(ndQ)$ time where d is the number of values for the X_i and Q is the number of states of the automaton. This is identical to the specialized propagator for REGULAR proposed in [Pesant, 2004].

One advantage of our encoding of the REGULAR constraint it that it gives us explicit access to the states of the automaton. Consider, for example, a rostering problem where workers are allowed to work for up to three consecutive shifts and then must take a break. This can be specified with a simple REGULAR language constraint. Suppose now we want to minimize the number of times a worker has to work for three consecutive shifts. To model this, we can post an AMONG constraint on the state variables to count the number of times we visit the state representing three consecutive shifts, and minimize the value taken by this variable.

5 SLIDE with counters

We often wish to slide a constraint down one or more sequences of variables computing some count. We can use SLIDE to encode such constraints by incrementally computing the count in an additional sequence of variables. As an example, consider the meta-constraint CARDPATH($C, [X_1, \ldots, X_n], N$) where C is any constraint of arity k [Beldiceanu and Carlsson, 2001]. This holds iff $C(X_i, \ldots, X_{i+k-1})$ holds N times down the sequence of variables. As we observed earlier, SLIDE is a special case of CARDPATH where $N = n - k + 1$. However, as we show here, CARDPATH can itself be encoded into a SLIDE constraint using a sequence of counters.

The CARDPATH constraint is useful in rostering problems. For example, we can count the number of changes in the type of shift given to a single worker using CARDPATH($\neq$, $[X_1, \ldots, X_n], N$). CARDPATH can also be used to model a range of Boolean connectives since $N \geq 1$ gives disjunction, $N = 1$ gives exclusive or, and $N = 0$ gives negation. For notational simplicity, we will consider the case when $k = 2$ and C is a binary constraint. The generalization to other k is straightforward. We introduce a sequence of integer variables M_i in which to accumulate the count. More precisely we decompose a CARDPATH constraint on a binary constraint C into SLIDE($G, [X_1, \ldots, X_{n+1}], [M_1, \ldots, M_{n+1}]$) where X_{n+1} is a "dummy" variable, $M_1 = 0$, $M_{n+1} = N$, and $G(X_i, X_{i+1}, M_i, M_{i+1})$ holds iff $C(X_i, X_{i+1})$ implies $M_{i+1} = M_i + 1$ else $M_{i+1} = M_i$.

6 SLIDE over sets

In some cases, we want to slide a constraint down one or more sequences of set variables. We therefore consider SLIDE meta-constraints which involve set variables. We give an example useful for breaking symmetry in problems like the social golfer's problem (prob010 in CSPLib).

Law and Lee have introduced the idea of value precedence for breaking the symmetry of indistinguishable values [Law and Lee, 2004]. They proposed a global constraint to deal with set variables containing indistinguishable values. More precisely, PRECEDENCE($[v_1, \ldots, v_m], [S_1, \ldots, S_n]$) holds iff $\min\{i \mid (v_j \in S_i \wedge v_k \notin S_i) \vee i = n+1\} \leq \min\{i \mid (v_k \in S_i \wedge v_j \notin S_i) \vee i = n+2\}$ for all $1 \leq j < k \leq m$, where $[v_1, \ldots, v_m]$ are the indistinguishable values. That is, the first time we distinguish v_j and v_k (because both don't occur in

a given set variable), we have v_j occurring and not v_k. For example, the following sequence of sets satisfies value precedence: $\{1,2,3\}, \{4,5,6\}, \{1,4,5\}$. The first two sets distinguish apart 1, 2, and 3 from 4, 5 and 6, whilst the third set distinguishes apart 1 from 2 and 3, and 4 and 5 from 6. However, this next sequence of sets does not satisfy value precedence as we distinguish apart 3 before 2: $\{1,2,3\}, \{1,3,4\}$.

We can encode such a symmetry breaking constraint using a SLIDE. For simplicity, we consider just two indistinguishable values, v_j and v_k. However, we can deal with multiple values using SLIDE but it is notationally more messy. We introduce 0/1 variables, B_i to record whether the two values have been distinguished apart so far. We then post SLIDE$(H, [S_1, \ldots, S_{n+1}], [B_1, \ldots, B_{n+1}])$ where S_{n+1} is a "dummy" set variable, $B_1 = 0$ and $H(S_i, S_{i+1}, B_i, B_{i+1})$ holds iff $B_i = B_{i+1} = 1$, or $B_i = B_{i+1} = 0$ and $v_j \in S_i$ and $v_k \in S_i$, or $B_i = B_{i+1} = 0$ and $v_j \notin S_i$ and $v_k \notin S_i$, or $B_i = 0$ and $B_{i+1} = 1$ and $v_j \in S_i$ and $v_k \notin S_i$. Note that H is again independent of its second argument.

7 Other examples of SLIDE

There are many other examples of global constraints which can be encoded using SLIDE. For example, we can encode the lexicographical ordering constraint LEX using SLIDE. LEX holds iff a vector of variables $[X_1..X_n]$ is lexicographically smaller than a vector $[Y_1..Y_n]$. We introduce a sequence of Boolean variables B_i to indicate if the vectors have been ordered yet at position $i-1$. X_{n+1} and Y_{n+1} are dummy variables and $B_1 = 0$. We slide the constraint $U(X_i, X_{i+1}, Y_i, Y_{i+1}, B_i, B_{i+1})$ which holds iff $(B_i = B_{i+1} = 0 \wedge X_i = Y_i)$ or $(B_i = 0 \wedge B_{i+1} = 1 \wedge X_i < Y_i)$ or $(B_i = B_{i+1} = 1)$. This gives us a linear time propagator as efficient and incremental as the specialized algorithm in [Frisch *et al.*, 2002]. As a second example, we can encode many types of channeling constraints using SLIDE like the DOMAIN constraint [Refalo, 2000], the LINKSET2BOOLEANS constraint [Beldiceanu *et al.*, 2005] and the ELEMENT constraint [Hentenryck and Carillon, 1988]. As a final example, we can encode "optimization" constraints like the soft form of the REGULAR constraint which measures the Hamming or edit distance to a regular string [van Hoeve *et al.*, 2006].

There are, however, global constraints that can be encoded using SLIDE which do not give us as efficient and effective propagators as specialized algorithms. As an example, the ALLDIFFERENT constraint can easily be specified using SLIDE (we just need a SLIDE which accumulates in a sequence of set variables the values used so far and ensure the final set variable has cardinality n). However, this is not as effective as a specialized flow-based propagator [Régin, 1994]. There are also global constraints like the inter-distance constraint [Régin, 1997] which SLIDE provides neither a good propagator nor it seems even a simple encoding.

8 SLIDE with GCC

We often have a constraint on the values which should occur across the whole sequence. For example, in car sequencing problems, to ensure we build the correct orders, we have a constraint on the total number of occurrences of each value along the sequence. The global sequencing constraint (GSC) [Régin and Puget, 1997] augments an AMONGSEQ constraint with a global cardinality constraint (GCC) on the total number of occurrence of different values. More precisely, GSC$([X_1, \ldots, X_n], a, b, q, v, [l_1, \ldots, l_m], [u_1, \ldots, u_m])$ is satisfied iff for each $i \in [1..m]$, $l_i \leq |\{j \mid X_j = i\}| \leq u_i$ (that is, the value i occurs between l_i and u_i times in total), and for each $k \in [1..n]$, $a \leq |\{j \mid X_j \in v \,\&\, k \leq j \leq k+q-1\}| \leq b$ (that is, values in v occur between a and b times in each sequence of q consecutive variables). In [Régin and Puget, 1997], an algorithm that partially propagates GSC is proposed. Another way to propagate GSC is to decompose it into a separate SLIDE and GCC. Enforcing GAC on such a decomposition hinders propagation and is incomparable to the pruning of the algorithm in [Régin and Puget, 1997].

We prove here that adding cardinality constraints to a SLIDE makes propagation intractable. In fact, we shall prove that enforcing GAC on the GSC constraint is intractable. As GSC can easily be encoded into a SLIDE and a GCC, the result follows immediately. This also settles the open question of the complexity of propagating GSC.

Theorem 1 *Enforcing GAC on* GSC *is NP-hard.*

Proof: We reduce the 1in3-SAT problem on positive clauses to finding support for a particular GSC. Consider a 1in3-SAT problem in N variables and M positive clauses in which the Boolean variables are numbered from 1 to N. We let $n = 2NM$. The basic idea of the reduction is that each consecutive block of $2N$ CSP variables represents a given truth assignment. The even numbered CSP variables will represent the truth assignment. The odd numbered CSP variables will essentially be "junk" and serve only to ensure we have exactly N non-zero values in each $2N$ block. That is, X_{2jN+2i} will be non-zero iff the ith Boolean variable is true in the given truth assignment. To achieve this, we set $a = b = N$, $q = 2N$ and $v = \{1, \ldots, M+1\}$. Each $2N$ block thus contains the same pattern of N zeroes and N non-zeroes.

The jth block of $2N$ CSP variables will ensure that the jth clause is satisfied by the truth assignment. That is, just one of its positive literals is true. Suppose the jth clause is $r \vee s \vee t$. Then we let X_{2jN+2r}, X_{2jN+2s} and X_{2jN+2t} have the domain $\{0, j+1\}$. All other CSP variables in the block have 0/1 domains. We set $l_{j+1} = u_{j+1} = 1$ to ensure only one of X_{2jN+2r}, X_{2jN+2s} and X_{2jN+2t} is set to $j+1$. Finally, we let $l_0 = u_0 = NM$, and $l_1 = u_1 = NM - M$. An assignment for X_i then corresponds to a satisfying assignment for the original 1in3-SAT problem. Deciding if the GSC has support is thus NP-hard. □

The proof can be generalized to show that enforcing bounds consistency on such a constraint is NP-hard, as well as to the case where $u_i = 1$ (in other words, when we have an AMONGSEQ with an ALLDIFFERENT constraint).

9 Circular SLIDE

Another generalization of SLIDE is when we wish to ensure that a constraint applies at any point round a cycle of variables. Such a meta-constraint is useful in scheduling and rostering problems where we need to ensure the schedule can be

repeated, say, every two weeks. If C is a constraint of arity k then we consider the meta-constraint:

$$\text{SLIDE}_\text{O}(C, [X_1, \ldots, X_n])$$

This holds iff $C(X_i, \ldots, X_{1+(i+k-1 \bmod n)})$ itself holds for $1 \leq i \leq n$.

As an example, we encode the circular form of the STRETCH constraint [Hellsten *et al.*, 2004]. First, let us consider the non-cyclic STRETCH constraint. In a STRETCH constraint, we are given a sequence of shift variables X_1 to X_n, each having domain a set of shift types τ, a set $\pi \subset \tau \times \tau$ of ordered pairs (called patterns), and the function $shortest(t)$ (resp. $longest(t)$) denoting the minimum (resp. maximum) length of any stretch of type t. STRETCH$([X_1, \ldots, X_n])$ holds iff (1) each stretch (i.e., a sequence of variables having the same type) of type t is feasible, i.e., each stretch has length between $shortest(t)$ and $longest(t)$; and (2) each pair of consecutive types of stretches is in π. We can encode STRETCH as SLIDE$(C, [X_1, \ldots, X_n], [Q_1, \ldots, Q_n])$ where $Q_1 = 1$ and $C[X_i, X_{i+1}, Q_i, Q_{i+1})$ holds iff (1) $X_i = X_{i+1}$, $Q_{i+1} = 1 + Q_i$, and $Q_{i+1} \leq longest(X_i)$; or (2) $X_i \neq X_{i+1}$, $\langle X_i, X_{i+1} \rangle \in \pi$, $Q_i \geq shortest(X_i)$ and $Q_{i+1} = 1$. Circular STRETCH is simply SLIDE$_\text{O}(C, [X_1, \ldots, X_n], [Q_1, \ldots, Q_n])$ in which we do not force $Q_1 = 1$.

10 A SLIDE algebra

When we negate a SLIDE, we get a disjunctive sequence of constraints. We therefore propose the SLIDEOR meta-constraint. More precisely, if C is a constraint of arity k then:

$$\text{SLIDEOR}(C, [X_1, \ldots, X_n])$$

holds iff one or more of $C(X_i, \ldots, X_{i+k-1})$ holds. We can also slide down multiple sequences simultaneously as with SLIDE. SLIDEOR can itself be encoded using SLIDE since SLIDEOR$(C, [X_1, \ldots, X_n])$ is equivalent to CARDPATH$(C, [X_1, \ldots, X_n], N)$ where $1 \leq N \leq n$, and CARDPATH can itself be encoded into SLIDE. One application of the SLIDEOR meta-constraint is to encode the global not all equals constraint, NOTALLEQUAL$([X_1, \ldots, X_n])$. This holds iff $X_i \neq X_j$ for some $1 < j \leq n$.

In fact, we can build up more complex sliding constraints using Boolean operators. We can simplify such complex constraint expressions by exploiting associativity, commutativity and De Morgan's identities. For example:

$$\begin{array}{rcl}
\neg\text{SLIDE}(C_1, [X_1, ., X_n]) & \leftrightarrow & \text{SLIDEOR}(\neg C_1, [X_1, ., X_n]) \\
\neg\text{SLIDEOR}(C_1, [X_1, ., X_n]) & \leftrightarrow & \text{SLIDE}(\neg C_1, [X_1, ., X_n]) \\
\text{SLIDE}(C_1, [X_1, ., X_n]) \wedge & & \\
\wedge\text{SLIDE}(C_2, [X_1, ., X_n]) & \leftrightarrow & \text{SLIDE}(C_1 \wedge C_2, [X_1, ., X_n])
\end{array}$$

11 Propagating SLIDE

A meta-constraint like SLIDE is only really useful if we can propagate it easily. The simplest possible way to propagate SLIDE$(C, [X_1, \ldots, X_n])$ is to decompose it into the sequence of constraints, $C(X_i, \ldots, X_{i+k-1})$ for $1 \leq i \leq n-k+1$ and let the constraint solver propagate the decomposition. Surprisingly, this is enough to achieve GAC in many cases. For example, we can achieve GAC this way using our SLIDE encoding of the REGULAR constraint. In such a case, propagating the decomposition is also an efficient means to achieve GAC. Only those constraints in the decomposition which have variables whose domains change need wake up. It thus provides an efficient incremental propagator for SLIDE.

Theorem 2 *Enforcing GAC on the decomposition of* SLIDE *achieves GAC on* SLIDE *if the slide constraints overlap on just one variable.*

Proof: The constraint graph of the decomposition is Berge-acyclic [Dechter and Pearl, 1989]. □

Similarly, enforcing GAC on the decomposition achieves GAC on SLIDE if the constraints being slide are monotone. A constraint C is monotone iff there exists a total ordering $\prec$ of the domain values such that for any two values v, w, if $v \prec w$ then v is substitutable to w in any support for C. For instance, the constraints AMONG and SUM are monotone if either no upper bound, or no lower bound is given.

Theorem 3 *Enforcing GAC on the decomposition of* SLIDE *achieves GAC on* SLIDE *if the slide constraints are monotone.*

Proof: For an arbitrary value $v \in D(X)$, we show that if every constraint is GAC, then we can build a support for (X, v) on SLIDE. For any variable other than X, we choose the first value in the total order, that is, the value which can be substituted to any other value in the same domain. The tuple built this way satisfies all the constraints being slide since we know that there exists a support for each (they are GAC), and the values we chose can be substituted to this support. □

On the other hand, in the general case, if constraints overlap on more than one variable (e.g. in the SLIDE encoding of AMONGSEQ). we need to do more work to achieve GAC. For reasons of space, we only have room here to outline how to propagate SLIDE in these circumstances. We consider two cases. If the arity of the constraint being slide is fixed, then we show that propagation is polynomial. On the other hand, if the arity of the constraint is not fixed, then propagation is intractable even if the constraint being slide is itself polynomial to propagate. In other words, enforcing GAC on SLIDE is fixed parameter tractable.

When the arity of the constraint being slide is fixed, we can use dynamic programming to compute support along the SLIDE. This is similar to the propagators for the REGULAR and STRETCH constraints [Pesant, 2004; Hellsten *et al.*, 2004]. Alternatively, we can use a dual encoding [Dechter and Pearl, 1989]. We sketch how such a dual encoding works, but lack space to describe improvements that can be made to improve the overall efficiency. We introduce dual variables to contain the supports for each constraint in the decomposition of the SLIDE. Between consecutive dual variables, we have binary compatability constraints to ensure the supports agree on overlapping dual variables. As the constraint graph of the dual variables is Berge-acyclic, enforcing AC on these dual variables, achieves GAC on the original SLIDE constraint. Using such a dual encoding, SLIDE can be easily added to any constraint solver. In general, enforcing GAC on a SLIDE constraint takes in $O(nd^{k+1})$ time and $O(nd^k)$ space where k is the overlap between successive constraints in the decomposition of SLIDE and d is the maximum domain size.

When the arity of the constraint being slide is not fixed, enforcing GAC is NP-hard.

Theorem 4 *Enforcing GAC on* SLIDE$(C, [X_1, \ldots, X_n])$ *is NP-hard when the arity of C is not fixed even if enforcing GAC on C is itself polynomial.*

Proof: A simple reduction from 3-SAT in N variables and M clauses. We let $n = (N+1)M$. Each block of $N+1$ variables represents a clause and a truth assignment. We will have $X_{j(N+1)+i+1} = 1$ iff the Boolean variable x_i is true ($1 \leq i \leq N$). If the $k+1$th clause is $x_a \vee \neg x_b \vee x_c$ then $X_{k(N+1)} \in \{x_a, \neg x_b, x_c\}$. Finally $C(X_i, \ldots, X_{i+N+1})$ holds iff $X_i \notin \{0, 1\}$ and $X_{i+N+1} = X_i$, or $X_i = x_d$ and $X_{i+d} = 1$, or $X_i = \neg x_d$ and $X_{i+d} = 0$. An assignment for X_i then corresponds to a satisfying assignment for the original 3-SAT problem. □

12 Experiments

We wish to show that encoding problems using the SLIDE meta constraint can be just as efficient and effective as using specialized propagators. Experiments are done using ILOG Solver 5.3.

12.1 Balanced Incomplete Block Design Generation

Balanced Incomplete Block Design (BIBD) generation is a standard combinatorial problem from design theory with applications in cryptography and experimental design. A BIBD is specified by a binary matrix of b columns and v rows, with exactly r ones per row, k ones per column, and a scalar product of λ between any pair of distinct rows. Our model consists of sum constraints on each row and each column as well as the scalar product constraint between every pair of rows. Any pair of rows and any pair of columns of a solution can be exchanged to obtain another symmetrical solution. We therefore impose lexicographic ordering constraints on rows and columns and look for a solution, following the details in [Kiziltan, 2004]. We propagate the LEX constraints either using the specialised algorithm GACLex given in [Frisch *et al.*, 2002] or the SLIDE encoding described in Section 7.

Table 1 shows the results on some large instances described as v, b, r, k, λ. As both propagators maintain GAC, we report the runtime results. We observe that the SLIDE encoding of LEX is as efficient (and sometimes even slightly more efficient than) the specialised algorithm.

12.2 Nurses Scheduling Problem

We consider a variant of the Nurse Scheduling Problem [Burke *et al.*, 2004] that consists of generating a schedule for each nurse of shifts duties and days off within a short-term planning period. There are three types of shifts (day, evening, and night). We ensure that (1) each nurse should take a day off or be assigned to an available shift; (2) each shift has a minimum required number of nurses; (3) each nurse work load should be between specific lower and upper bounds; (4) each nurse can work at most 5 consecutive days; (5) each nurse must have at least 12 hours of break between two shifts; (6) each nurse should have at least two consecutive days on any shift. We wrote two models to solve this problem. In both models, we introduce one variable for each nurse and each day, indicating to what type of shift, if any, this nurse is affected on this day. The constraints (1)-(3) are enforced using a set of global cardinality constraints. Constraints (4), (5) and (6) form sequences of respectively 6-ary, binary and ternary constraints. Notice that (4) is monotone, hence we simply posted these constraints in both models. However, the conjunction of constraints (4) and (5) is slide in the first model whilst it is decomposed in the second.

To test the two models, we generated by hand four instances respecting common sense criteria, such as lower demand during evening and night shifts. We observe that in the four instances the slide model outperforms the other model in terms either of backtracks or both cpu time and backtracks. It manages to solve the four instances in less time (cpu time ratio of 1.37 in average) and with fewer backtracks (backtrack ratio of 2.75 in average). This shows the effectiveness of SLIDE both as a modelling construct and as well as a specialised propagator.

Instance	GACLex algorithm time (s)	Slide encoding time (s)
7,91,39,3,13	0.59	0.55
9,72,24,3,6	0.57	0.53
9,96,32,3,8	2.20	2.16
9,108,36,3,9	2.13	2.11
10,90,27,3,6	1.26	1.28
10,120,36,3,8	3.38	3.50
11,110,30,3,6	2.55	2.65
12,88,22,3,4	1.28	1.25
13,78,18,3,3	0.98	1.00
13,104,24,3,4	2.15	2.13
15,21,7,5,2	26.78	26.60
15,70,14,3,2	0.97	0.91
16,32,12,6,4	452.25	450.96
16,80,15,3,2	1.49	1.39
19,57,9,3,1	2.70	2.63
22,22,7,7,2	73.97	71.81

Table 1: BIBD generation.

Instance	Slide encoding time (s)	Slide encoding backtracks	No Slide encoding time (s)	No Slide encoding backtracks
10×14	82.32	271,348	133.44	776,019
12×14	4.52	13,484	11.57	58,709
14×14	0.37	1,356	0.29	1,877
10×16	0.83	4,116	1.35	10,017

Table 2: Nurse Schedule generation.

13 Related work

Beldiceanu and Carlsson introduced the CARDPATH meta-constraint [Beldiceanu and Carlsson, 2001]. They showed that it can be used to encode a wide range of constraints like CHANGE, SMOOTH, AMONGSEQ and SLIDINGSUM. They provided a propagator for CARDPATH that greedily con-

structs upper and lower bounds on the number of (un)satisfied constraints by posting and retracting (the negation of) each of the constraints. This propagator does not achieve GAC.

Pesant introduced the REGULAR constraint, and gave a propagation algorithm based on dynamic programming that enforces GAC [Pesant, 2004]. As we saw, the REGULAR constraint can be encoded using a simple SLIDE. There are, however, a number of important differences between the two. First, REGULAR only slides a ternary constraint down a sequence of variables. SLIDE, however, can slide a constraint of *any* arity. This permits us to deal with constraints like AMONGSEQ. Second, Pesant proposed a specialized propagator for REGULAR based on dynamic programming. This is unnecessary as we can achieve GAC by simply decomposing the SLIDE constraint into a sequence of ternary constraints. Third, as we described earlier, our encoding introduces variables for representing the states and access to these state variables can be useful (e.g. for expressing objective functions).

Beldiceanu, Carlsson and Petit have also proposed specifying global constraints by means of deterministic finite automata augmented with counters [Beldiceanu *et al.*, 2004]. Propagators for such automata are constructed automatically by decomposing the specification into a sequence of signature and transition constraints. If the automaton uses counters, this decomposition hinders propagation so pairwise consistency is needed in general to guarantee GAC. We can encode such automata using a SLIDE where we introduce an additional sequence of variables for each counter. Our methods thus provide a GAC propagator for such automata.

Hellsten, Pesant and van Beek proposed a propagator for the STRETCH constraint that achieves GAC based on dynamic programming similar to that for the REGULAR constraint [Hellsten *et al.*, 2004]. We can again encode the STRETCH constraint using a simple SLIDE.

14 Conclusions

We have studied the SLIDE meta-constraint. This slides a constraint down one or more sequences of variables. We have shown that SLIDE can be used to encode and propagate a wide range of global constraints including AMONGSEQ, CARDPATH, PRECEDENCE, and REGULAR. We have also considered a number of extensions including sliding down sequences of set variables, and combining SLIDE with a global cardinality constraint. When the constraint being slide overlap on just one variable, we argued that decomposition does not hinder propagation and SLIDE can be propagated simply by posting the sequence of constraints. Our experiments demonstrated that using SLIDE to encode constraints can be just as efficient and effective as specialized propagators. There are many directions for future work. One promising direction is to use binary decision diagrams to store the supports for the constraints being slide when they have many satisfying tuples. We believe this could improve the efficiency of our propagator in many cases.

References

[Beldiceanu and Carlsson, 2001] N. Beldiceanu and M. Carlsson. Revisiting the cardinality operator and introducing cardinality-path constraint family. In *Proc. of ICLP'01*, pp. 59–73. Springer-Verlag, 2001.

[Beldiceanu and Contejean, 1994] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20:97–123, no. 12 1994.

[Beldiceanu *et al.*, 2004] N. Beldiceanu, M. Carlsson, and T. Petit. Deriving filtering algorithms from constraint checkers. In *Proc. of CP'04*, pp. 107–122. Springer, 2004.

[Beldiceanu *et al.*, 2005] N. Beldiceanu, M. Carlsson, and J-X. Rampon. Global constraints catalog. SICS Technical Report T2005/08, 2005.

[Bessiere *et al.*, 2005] C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. The range and roots constraints: Specifying counting and occurrence problems. In *Proc. of IJCAI'05*, pp. 60–65. Professional Book Center, 2005.

[Burke *et al.*, 2004] Burke, E. K.; Causmaecker, P. D.; Berghe, G. V.; and Landeghem, H. V. 2004. The state of the art of nurse rostering. *J. of Scheduling* 7(6):441–499.

[Dechter and Pearl, 1989] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[Frisch *et al.*, 2002] A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In *Proc. of CP'02*, pp. 93–108. Springer, 2002.

[Hellsten *et al.*, 2004] L. Hellsten, G. Pesant, and P. van Beek. A domain consistency algorithm for the stratch constraint. In *Proc. of CP'04*, pp. 290–304. Springer, 2004.

[Hentenryck and Carillon, 1988] P. Van Hentenryck and J.-P. Carillon. Generality versus specificity: An experience with AI and OR techniques. In *Proc. of AAAI'88*, pp. 660–664. AAAI Press/The MIT Press, 1988.

[Kiziltan, 2004] Z. Kiziltan. Symmetry Breaking Ordering Constraints. PhD Thesis, Uppsala University, 2004.

[Law and Lee, 2004] Y.C. Law and J.H.M. Lee. Global constraints for integer and set value precedence. In *Proc. of CP'04*, pp. 362–376. Springer, 2004.

[Maher, 2002] M. Maher. Analysis of a global contiguity constraint. In *Proc. of the CP'02 Workshop on Rule Based Constraint Reasoning and Programming*, 2002.

[Pesant, 2004] G. Pesant. A regular language membership constraint for finite sequences of variables. In *Proc. of CP'04*, pp. 482–295. Springer, 2004.

[Refalo, 2000] P. Refalo. Linear formulation of constraint programming models and hybrid solvers. In *Proc. of CP'00*, pp. 369–383. Springer, 2000.

[Régin and Puget, 1997] J-C. Régin and J-F. Puget. A filtering algorithm for global sequencing constraints. In *Proc. of CP'97*, pp. 32–46. Springer, 1997.

[Régin, 1994] J-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proc. of AAAI'94*, pp. 362–367. AAAI Press, 1994.

[Régin, 1997] J-C. Régin. The global minimum distance constraint. Technical report, ILOG Inc, 1997.

[van Hoeve *et al.*, 2006] W-J. van Hoeve, G. Pesant, and L-M. Rousseau. On global warming : Flow-based soft global constaints. To appear in *Journal of Heuristics*.