SLIDE: A Useful Special Case of the CARDPATH Constraint

Christian Bessiere, Emmanuel Hébrard, Brahim Hnich, Zeynep Kiziltan, Toby Walsh

To cite this version:


HAL Id: lirmm-00329876
https://hal-lirmm.ccsd.cnrs.fr/lirmm-00329876
Submitted on 13 Oct 2008
SLIDE: A Useful Special Case of the CARDPATH Constraint

Christian Bessiere\textsuperscript{1} and Emmanuel Hebrard\textsuperscript{2} and Brahim Hnich\textsuperscript{3} and Zeynep Kiziltan\textsuperscript{4} and Toby Walsh\textsuperscript{5}

Abstract. We study the \textsc{CARDPATH} constraint. This ensures a given constraint holds a number of times down a sequence of variables. We show that \textsc{SLIDE}, a special case of \textsc{CARDPATH} where the slide constraint must hold always, can be used to encode a wide range of sliding sequence constraints including \textsc{CARDPATH} itself. We consider how to propagate \textsc{SLIDE} and provide a complete propagator for \textsc{CARDPATH}. Since propagation is NP-hard in general, we identify special cases where propagation takes polynomial time. Our experiments demonstrate that using \textsc{SLIDE} to encode global constraints can be as efficient and effective as specialised propagators.

1 INTRODUCTION

In many scheduling problems, we have a sequence of decision variables and a constraint which applies down the sequence. For example, in the car sequencing problem, we need to decide the sequence of cars on a production line. We might have a constraint on how often a particular option is met (e.g. 1 out of 3 cars can have a sun-roof). As a second example, in a nurse rostering problem, we need to decide on how many consecutive night shifts any nurse can work. We might have a constraint on how many consecutive night shifts any nurse can work. Such constraints have been classified as sliding sequence constraints [7]. To model such constraints, we can use the \textsc{CARDPATH} constraint. This ensures that a given constraint holds a number of times down a sequence of variables [5]. We identify a special case of \textsc{CARDPATH} which we call \textsc{SLIDE}, that is interesting for several reasons. First, many sliding sequence constraints, including \textsc{CARDPATH}, can easily be encoded using this special case. \textsc{SLIDE} is therefore a “general-purpose” constraint for encoding sliding sequencing constraints. This is an especially easy way to provide propagators for such global constraints within a constraint toolkit. Second, we give a propagator for enforcing generalised arc-consistency on \textsc{SLIDE}. By comparison, the previous propagator for \textsc{CARDPATH} given in [5] does not prune all possible values. Third, \textsc{SLIDE} can be as efficient and effective as specialised propagators in solving sequencing problems.

2 \textsc{CARDPATH AND SLIDE CONSTRAINTS}

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 given sets of variables. We use capital letters for variables (e.g. \(X\)), and lower case for values (e.g. \(d\)). We write \(D(X)\) for the domain of variable \(X\). Constraint solvers typically explore partial assignments enforcing a local consistency property. A constraint is \textit{generalised arc consistent (GAC)} iff when a variable is assigned any value in its domain, there exist compatible values in the domains of all the other variables of the constraint.

The \textsc{CARDPATH} constraint was introduced in [5]. If \(C\) is a constraint of arity \(k\) then \textsc{CARDPATH}(\(N, \{X_1, \ldots, X_n\}, C\)) holds iff \(C(X_i, \ldots, X_{i+k-1})\) holds \(N\) times for \(1 \leq i \leq n - k + 1\). For example, we can count the number of changes in the type of shift used by nurses. In \textsc{SLIDE}(\(C, \{X_1, \ldots, X_n\}\)) holds iff \(C(X_i, \ldots, X_{i+k-1})\) holds for all \(1 \leq i \leq n - k + 1\). That is, a \textsc{CARDPATH} constraint in which \(N = n - k + 1\). We also consider a more complex form of \textsc{SLIDE} that applies only every \(j\) variables. More precisely, \(\textsc{SLIDE}_j(C, \{X_1, \ldots, X_n\})\) holds iff \(C(X_{ij+1}, \ldots, X_{ij+k})\) holds for \(0 \leq i \leq \frac{n-k}{j}\). By definition \(\textsc{SLIDE}_j\) for \(j = 1\) is equivalent to \textsc{SLIDE}.

Beldiceanu and Carlsson have shown that \textsc{CARDPATH} can encode a wide range of constraints like \textsc{CHANGE}, \textsc{SMOOTH}, \textsc{AMONGSEQ} and \textsc{SLIDINGSUM} [5]. As we discuss later, \textsc{SLIDE} provides a simple way to encode such sliding sequencing constraints. It can also encode many other more complex sliding sequencing constraints like \textsc{REGULAR} [16], \textsc{STRETCH} [13], and \textsc{LEX} [7], as well as many types of channelling constraints like \textsc{ELEMENT} [19] and optimisation constraints like the soft forms of \textsc{REGULAR} [20]. More interestingly, \textsc{CARDPATH} can itself be encoded into a \textsc{SLIDE} constraint. In [5], a propagator for \textsc{CARDPATH} is proposed that greedily constructs 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. We propose here a complete propagator for enforcing GAC on \textsc{SLIDE}. \textsc{SLIDE} thus provides a GAC propagator for \textsc{CARDPATH}. In addition, \textsc{SLIDE} provides a GAC propagator for any of the other global constraints it can encode. As our experimental results reveal, \textsc{SLIDE} can be as efficient and effective as specialised propagators.

We illustrate the usefulness of \textsc{SLIDE} with the \textsc{AMONGSEQ} constraint which 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 are a “night shift”. More
3 SLIDE WITH MULTIPLE SEQUENCES

We often wish to slide a constraint down two or more sequences of variables at once. For example, suppose we want to ensure that two vectors of variables, $X_1$ to $X_n$ and $Y_1$ to $Y_m$ differ at every index. We can encode such a constraint by interleaving the two sequences and sliding a constraint down the single sequence with a suitable offset.

In our example, we simply post $\text{SLIDE}_2(X_1, Y_1, \ldots, X_n, Y_m)$. As a second example of sliding down multiple sequences of variables, consider the constraint $\text{REGULAR}(A, \{X_1, \ldots, X_n\})$. This ensures that the values taken by a sequence of variables form a string accepted by a deterministic finite automaton $A$ [16]. 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: $\text{AMONG}$ [8], $\text{CONTIGUITY}$ [15], $\text{LEX}$ and $\text{PRECEDENCE}$ [14].

To encode the $\text{REGULAR}$ constraint with $\text{SLIDE}$, we introduce variables, $Q_i$ to record the state of the automaton. We then post $\text{SLIDE}_2(F(Q_0, X_1, Q_1, \ldots, X_n, Q_n))$ where $Q_0$ is set to the starting state, $Q_0$ is restricted to accepting states, and $F(Q_i, X_{i+1}, Q_{i+1})$ holds iff $Q_{i+1} = \delta(X_i, Q_i)$ where $\delta$ is the transition function of the automaton. If we decompose this encoding into the conjunction of slid constraints, we get a set of constraints similar to [6]. Enforcing GAC on this encoding ensures GAC on $\text{REGULAR}$ and, by exploiting functionality of $F$, takes $O(ndq)$ time where $d$ is the number of values for $X_i$ and $q$ is the number of states of the automaton. This is asymptotically identical to the specialised $\text{REGULAR}$ propagator [16]. This encoding is highly competitive in practice with the specialized propagator [2].

One advantage of this encoding is that it gives 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. This can be specified with a simple $\text{REGULAR}$ constraint. Suppose now we want to minimise the number of times a worker has to work for three consecutive shifts. To encode this, we can post an $\text{AMONG}$ constraint on the state variables to count the number of times we visit the state representing three consecutive shifts, and minimise the value taken by this variable. As we shall see later in the experiments, the encoding also gives an efficient incremental propagator. In fact, the complexity of repeatedly enforcing GAC on this encoding of the $\text{REGULAR}$ constraint down the whole branch of a backtracking search tree is just $O(ndq)$ time.

4 SLIDE WITH COUNTERS

We may want to slide a constraint on a sequence of variables computing a count. We can use $\text{SLIDE}$ to encode such constraints by incrementally computing the count in an additional sequence of variables. Consider, for example, $\text{CARDPATH}(N, \{X_1, \ldots, X_n\})$. For simplicity, we consider $k = 2$ (i.e., $C$ is binary). The generalisation to other $k$ is straightforward. We introduce a sequence of integer variables $M_i$ in which to accumulate the count. We encode $\text{CARDPATH}$ as $\text{SLIDE}_2(G, \{M_1, X_1, \ldots, M_n, X_n\})$ where $M_1 = 0, M_n = N + 1$ and $G(M_i, X_i, M_{i+1}, X_{i+1})$ is defined as: if $C(X_i, X_{i+1})$ holds then $M_{i+1} = M_i + 1$, otherwise $M_{i+1} = M_i$. GAC on $\text{SLIDE}$ ensures GAC on $\text{CARDPATH}$.

As a second example, consider the $\text{STRETCH}$ constraint [13]. Given variables $X_1$ to $X_n$ taking values from a set of shift types $\tau$, a set $\pi$ of ordered pairs from $\tau \times \tau$, and functions $\text{shortest}(t)$ and $\text{longest}(t)$ giving the minimum and maximum length of a stretch of type $t$, $\text{STRETCH}(X_1, \ldots, X_n)$ holds iff each stretch of type $t$ has length between $\text{shortest}(t)$ and $\text{longest}(t)$; and consecutive types of stretches are in $\pi$. We can encode $\text{STRETCH}$ as $\text{SLIDE}_2(H, \{X_1, Q_1, \ldots, X_n, Q_n\})$ where $Q_1 = 1$ and $H(X_1, X_{i+1}, Q_1, Q_{i+1})$ holds iff $(1) X_i = X_{i+1}, Q_{i+1} = 1 + Q_1$, and $Q_{i+1} \leq \text{longest}(X_i)$; or $(2) X_i \neq X_{i+1}, \forall (X_i, X_{i+1}) \in \pi$, $Q_i \geq \text{shortest}(X_i)$ and $Q_{i+1} = 1$. GAC on $\text{SLIDE}$ ensures GAC on $\text{STRETCH}$.

5 OTHER EXAMPLES OF SLIDE

There are many other examples of global constraints which we can encode using $\text{SLIDE}$. For example, we can encode $\text{LEX}$ [7] using $\text{SLIDE}$. $\text{LEX}$ holds iff a vector of variables $\{X_1, \ldots, X_n\}$ is lexicographically smaller than another vector of variables $\{Y_1, \ldots, Y_n\}$. We introduce a sequence of Boolean variables $B_i$ to indicate if the vectors have been ordered by position $i$ - 1. Hence $B_0 = 1$. We then encode $\text{LEX}$ as $\text{SLIDE}_2(I, \{B_1, X_1, Y_1, \ldots, B_n, X_n\})$ where $I(B_i, X_i, Y_i, B_{i+1})$ holds iff $B_i = B_{i+1} = 0 \land X_i = Y_i$ or $(B_i = 0 \land B_{i+1} = 1 \lor X_i < Y_i)$ or $(B_i = B_{i+1} = 1)$. This gives us a linear time propagator as efficient and incremental as the specialised algorithm in [12]. As a second example, we can encode many types of channelling constraints using $\text{SLIDE}$ like $\text{DOMAIN}$ [17], $\text{LINKSET2BOOLEANS}$ [7] and $\text{ELEMENT}$ [19]. As a final example, we can encode “optimisation” constraints like the soft form of the $\text{REGULAR}$ constraint which measures the Hamming edit distance to a regular string [20]. There are, however, constraints that can be encoded using $\text{SLIDE}$ which do not give as efficient and effective propagators as specialised algorithms (e.g. the global ALLDIFFERENT constraint [18]).

6 PROPAGATING SLIDE

A constraint like $\text{SLIDE}$ is only really useful if we can propagate it efficiently and effectively. The simplest possible way to propagate $\text{SLIDE}_i(C, \{X_1, \ldots, X_n\})$ is to decompose it into a sequence of constraints, $C(X_{ij} \ldots, X_{ij+k})$ for $0 \leq i \leq \frac{n-k}{2}$, and let the constraint solver propagate the decomposition. Surprisingly, this is enough to achieve GAC in many cases. For example, we can achieve GAC in this way on the $\text{SLIDE}$ encoding of the $\text{REGULAR}$ constraint. If the constraints in the decomposition overlap on just one variable then the constraint graph is Berge acyclic [4], and enforcing GAC on the decomposition of $\text{SLIDE}_i$ achieves GAC on $\text{SLIDE}_i$. Similarly, enforcing GAC on the decomposition achieves GAC on $\text{SLIDE}_i$ if
the constraint being slid is monotone. A constraint $C$ is monotone iff there exists a total ordering < of the values such that for any two values $v, w$, if $v < w$ then $v$ can replace $w$ 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 1** Enforcing GAC over each constraint in the decomposition of SLIDE$_j$, achieves GAC on SLIDE$_i$ if the constraint being slid is 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$_j$. For any variable other than $X$, we choose the smallest value in the total order. This is the value that can be substituted for any other value in the same domain. A tuple built this way satisfies all the constraints being slid since we know that there exists a support for each (they are GAC), and the values we chose can be substituted for this support. □

In the general case, when constraints overlap on more than one variable (e.g. in the SLIDE encoding of AMONGSEQ), we need to do more work to achieve GAC. We distinguish two cases: when the arity of the constraint being slid is not fixed, and when the arity is fixed.

We show that enforcing GAC in the former case is NP-hard.

**Theorem 2** 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:** We give a reduction from 3-SAT in $N$ variables and $M$ clauses. We introduce variables $X_i^j$ for $1 \leq i \leq N + 1$ and $1 \leq j \leq M$. For each clause $j$, if the clause is $x_a \vee \neg x_b \vee x_c$, then we set $X_i^j \in \{x_a, \neg x_b, x_c\}$ to represent the values that make this clause true. For each clause $j$, we set $X_i^j \in \{0, 1\}$ for $1 \leq i \leq N$ to represent a truth assignment. Hence, we duplicate the truth assignment for each clause. We now build the following constraint SLIDE$(C, [X_1^1, \ldots, X_{N+1}^1, \ldots, X_1^M, \ldots, X_{N+1}^M])$ where $C$ has arity $N + 1$. We construct $C(Y_1, \ldots, Y_{N+1})$ to hold iff $Y_1 = x_d$ and $Y_{i+1} = 1$, or $Y_1 = \neg x_d$ and $Y_{i+1} = 0$. (in these two cases, the value assigned to $Y_1$ represents the literal that makes clause $j$ true), or $Y_1 \in \{0, 1\}$ and $Y_i = Y_{i+1}$ (in this case, the truth assignment is passed down the sequence). Enforcing GAC on $C$ is polynomial and an assignment satisfying the SLIDE constraint corresponds to a satisfying assignment for the original 3-SAT problem. □

When the arity of the constraint being slid is not great, we can enforce GAC on SLIDE using dynamic programming (DP) in a similar way to the DP-based propagators for the REGULAR and STRETCH constraints [16, 13]. A much simpler method, however, which is just as efficient and effective as dynamic programming is to exploit a variation of the dual encoding into binary constraints [10] based on tuples of support. Such an encoding was proposed in [1] for a particular sliding constraint. Here we show that this method is more general and can be used for arbitrary SLIDE constraints. Using such an encoding, SLIDE can be easily added to any constraint solver. We illustrate the intersection encoding by means of an example.

Consider again the AMONGSEQ example in which 2 of every 3 variables of $X_1 \ldots X_5$ should take the value $a$, where $X_1 = a$ and $X_2, \ldots, X_5 \in \{a, b\}$. We can encode this as SLIDE$(E, [X_1, X_2, \ldots, X_4, X_5])$ where $E(X_1, X_2, X_4) = 1$ is an instance of the AMONG constraint that ensures two of its three variables take $a$. If the sliding constraint has arity $k$, we introduce an intersection variable for each subsequence of $k - 1$ variables of SLIDE. The first intersection variable $V_1$ has a domain containing all tuples from $D(X_1) \times \ldots \times D(X_{k-1})$. The $j$th intersection variable $V_j$ has domain containing $D(X_j) \times \ldots \times D(X_{k-1})$. And so on until $V_{n-k+2}$. In our example in Fig 1, this gives $D(V_1) = D(X_1) \times D(X_2)$, $\ldots$, $D(V_4) = D(X_4) \times D(X_5)$. We then post binary compatibility constraints between consecutive intersection variables. These constraints ensure that the two intersection variables assign $(k - 1)$-tuples that agree on the values of their $k - 2$ common original variables (like constraints in the dual encoding). They also ensure that the $k$-tuple formed by the two $(k - 1)$-tuples satisfies the corresponding instance of the slid constraint. For instance, in Fig 1, the binary constraint between $V_1$ and $V_2$ does not allow the pair $<a, b>$ because the second argument of $ab$ for $V_1$ (value $b$ for $X_2$) is in conflict with the first argument of $aa$ for $V_2$ (value $a$ for $X_2$). That same constraint between $V_2$ and $V_3$ does not allow the pair $<a, b>$ because the tuple $aab$ is not allowed by $E(X_1, X_2, X_3)$.

Enforcing AC on such compatibility constraints prunes $ab$ and $bb$ from $V_2$, $ab$ and $bb$ from $V_3$, and $ba$ and $bb$ from $V_4$. Finally, we post binary channelling constraints to link the tuples to the original variables. One such constraint for each original variable is sufficient. For example, we can have a channelling constraint between $V_4$ and $X_4$ which ensures that the first argument of the tuple assigned to $V_4$ equals the value assigned to $X_4$. Enforcing AC on this channelling constraint prunes $b$ from the domain of $X_4$. We could instead post a channelling constraint between $V_4$ and $X_4$ ensuring that the second argument in $V_5$ equals $X_4$. The AMONGSEQ constraint is now GAC.

**Theorem 3** Enforcing AC on the intersection encoding of SLIDE achieves GAC in $O(n d^{d^{-1}})$ space and $O(n d^{d^{-1}})$ time where $k$ is the arity of the constraint to slide and $d$ is the maximum domain size.

**Proof:** The constraint graph associated with the intersection encoding is a tree. Enforcing AC on this therefore achieves GAC. Enforcing AC on the channelling constraints then ensures that the domains of the original variables are pruned appropriately. As we introduce $O(n)$ intersection variables, and each can contain $O(d^{d-1})$ tuples, the intersection encoding requires $O(n d^{d-1})$ space. Enforcing AC on a compatibility constraint between two intersection variables $V_i$ and $V_{i+1}$ takes $O(d^d)$ time as each tuple in the intersection variable $V_i$ has at most $d$ supports which are the tuples of $V_{i+1}$ that are equal to $V_i$ on their $k - 2$ common arguments. Enforcing AC on $O(n)$ such constraints therefore takes $O(n d^d)$ time. Finally, enforcing AC on each of the $O(n)$ channelling constraints takes $O(d^{d-1})$ time as they are functional. Hence, the total time complexity is $O(n d^d)$. □

Arc consistency on the intersection encoding simulates pairwise consistency on the decomposition. It does this efficiently as intersection variables represent in extension ‘only’ the intersections. This is sufficient because the constraint graph is acyclic. This encoding is also very easy to implement in any constraint solver. It has good
incremental properties. Only those constraints associated with a variable which changes need to wake up.

The intersection encoding of SLIDE, for \( j > 1 \) is less expensive to build than for \( j = 1 \) as we need intersection variables for subsequences of less than \( k - 1 \) variables. For \( 1 \leq j \leq k/2 \), we introduce intersection variables for subsequences of variables of length \( k - j \) starting at indices \( 1, j + 1, 2j + 1 \ldots \) whose domains contain \((k-j)\) tuples of assignments. Compatibility and chalklining constraints are defined as with \( j = 1 \). If \( j > k/2 \), two consecutive intersection variables (for two subsequences of \( k - j \) variables) involve less than \( k \) variables of the SLIDE. The compatibility constraint between them cannot thus ensure the satisfaction of the slid constraint. We therefore introduce intersection variables for subsequences of length \( \lceil k/2 \rceil \) starting at indices \( 1, j + 1, 2j + 1 \ldots \) and for subsequences of length \( \lfloor k/2 \rfloor \) finishing at indices \( k, j + k, 2j + k \ldots \). The compatibility constraint between two consecutive intersection variables representing the subsequence starting at index \( p_j \) and the subsequence finishing at index \( p_j + k \) ensures satisfaction of the \((p+1)\)th instance of the slid constraint. The compatibility constraint between two consecutive intersection variables representing subsequence finishing at index \( p_j + k \) and the subsequence starting at index \( (p+1)j + 1 \) ensures the consistency of the arguments in the intersection of two instances of the slid constraint.

7 EXPERIMENTS

We now demonstrate the practical value of SLIDE. Due to space limits, we only report detailed results on a nurse scheduling problem, and summarise the results on balanced incomplete block design generation and car sequencing problems. Experiments are performed with ILOG Solver 6.2 on a 2.8GHz Intel computer running Linux.

We consider a Nurse Scheduling Problem [9] in which we generate a schedule of shift duties for a short-term planning period. There are three types of shifts (day, evening, and night). We ensure that (1) each nurse takes a day off or is assigned to an available shift; (2) each shift has a minimum required number of nurses; (3) each nurse’s work load is between specific lower and upper bounds; (4) each nurse works at most 5 consecutive days; (5) each nurse has at least 12 hours of break between two shifts; (6) the shift assigned to a nurse does not change more than once every three days. We construct four different models, all with variables indicating what type of shift, if any, each nurse is working on each day. We break symmetry between the nurses with lex constraints. The constraints (1)-(3) are enforced using global cardinality constraints. Constraints (4), (5) and (6) form sequences of respectively 6-ary, binary and ternary constraints. Since (4) is monotone, we simply post the decomposition in the first three models. This achieves GAC by Theorem 1. The models differ in how (5) and (6) are propagated. In decomp, they are decomposed into conjunction of slid constraints. In amongseq, (5) is decomposed and (6) is enforced using the AMONGSEQ constraint of ILOG Solver (called liloSequence). The combination of (5) and (6) are enforced by SLIDE in slide. Finally, in slide, we use SLIDE for the combination of (4), (5), and (6).

We test the models using the instances available at http://www.projectmanagement.ugent.be/nsp.php in which nurses have no maximum workload, but a set of preferences to optimise. We ignore these preferences and post a constraint bounding the maximum workload to at most 5 day shifts, 4 evening shifts and 2 night shifts per nurse and per week. Similarly, each nurse must have at least 2 rest days per week. We solve three samples of instances involving 25, 30 and 60 nurses to schedule over 28 days.

We use the same variable ordering for all models so that heuristic choices do not affect results. We schedule the days in chronological order and within each day we allocate a shift to every nurse in lexicographical order. Initial experiments show that this is more efficient than the minimum domain heuristic. However, it restricts the variety of domains passed to the propagators, and thus hinders any demonstration of differences in pruning. We therefore also use a more random heuristic. We allocate within each day a shift to every nurse randomly with 20% frequency and lexicographically otherwise.

### Table 1. Nurse Scheduling with lexicographical variable ordering (\(^2\) on instances solved by all methods, \(^3\) on instances solved by the method).

<table>
<thead>
<tr>
<th>Model</th>
<th>#solved</th>
<th>bts(^1)</th>
<th>time(^1)</th>
<th>bts(^2)</th>
<th>time(^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>decomp</td>
<td>25 nurses, 28 days (99 instances)</td>
<td>99</td>
<td>301</td>
<td>0.11</td>
<td>301</td>
</tr>
<tr>
<td>amongseq</td>
<td>99</td>
<td>301</td>
<td>0.19</td>
<td>301</td>
<td>0.19</td>
</tr>
<tr>
<td>slide</td>
<td>99</td>
<td>301</td>
<td>0.19</td>
<td>301</td>
<td>0.19</td>
</tr>
<tr>
<td>slide</td>
<td>99</td>
<td>301</td>
<td>0.19</td>
<td>301</td>
<td>0.19</td>
</tr>
<tr>
<td>amongseq</td>
<td>55</td>
<td>1999</td>
<td>2.13</td>
<td>5500</td>
<td>2.13</td>
</tr>
<tr>
<td>slide</td>
<td>55</td>
<td>1999</td>
<td>2.13</td>
<td>5500</td>
<td>2.13</td>
</tr>
</tbody>
</table>

### Table 2. Nurse Scheduling with random variable ordering (\(^4\) on instances solved by all methods, \(^5\) on instances solved by the method).

<table>
<thead>
<tr>
<th>Model</th>
<th>#solved</th>
<th>bts(^3)</th>
<th>time(^3)</th>
<th>bts(^4)</th>
<th>time(^4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>decomp</td>
<td>25 nurses, 28 days (99 instances)</td>
<td>99</td>
<td>301</td>
<td>0.11</td>
<td>301</td>
</tr>
<tr>
<td>amongseq</td>
<td>99</td>
<td>301</td>
<td>0.19</td>
<td>301</td>
<td>0.19</td>
</tr>
<tr>
<td>slide</td>
<td>99</td>
<td>301</td>
<td>0.19</td>
<td>301</td>
<td>0.19</td>
</tr>
<tr>
<td>slide</td>
<td>99</td>
<td>301</td>
<td>0.19</td>
<td>301</td>
<td>0.19</td>
</tr>
<tr>
<td>amongseq</td>
<td>55</td>
<td>1999</td>
<td>2.13</td>
<td>5500</td>
<td>2.13</td>
</tr>
<tr>
<td>slide</td>
<td>55</td>
<td>1999</td>
<td>2.13</td>
<td>5500</td>
<td>2.13</td>
</tr>
</tbody>
</table>

Tables 1 and 2 report the mean runtime and fails to solve the instances with 5 minutes cutoff. Between the first three models, the best results are due to slide. We solve more instances with slide, as well as explore a smaller tree. By developing a propagator for a generic constraint like SLIDE, we can increase pruning without hurting efficiency. Note that slide always performs better than amongseq. A possible reason is that AMONGSEQ cannot encode constraint (6) as directly as SLIDE. As in previous models, we need to channel into Boolean variables and post AMONGSEQ on them. This may not give as effective and efficient pruning. SLIDE thus offers both modellling and solving advantages over existing sequencing constraints. Note also that slide solves additional instances in the time limit. This is not suprising, as the model slide is the combination of the constraints (4), (5), and (6). Recall that the sliding constraint of (4) is 6-ary. It is pleasing to note that the intersection encoding performs well even in the presence of such a high arity constraint.

We also ran experiments on Balanced Incomplete Block Designs (BIBDs) and car sequencing. For BIBD, we use the model in [12] which contains LEX constraints. We propagate these either using the specialised algorithm of [12] or the SLIDE encoding. As both propagators maintain GAC, we only compare runtimes. Results on large instances show that the SLIDE model is as efficient as the LEX
model. For car sequencing, we test the scalability of SLIDE on large arity constraints and large domains using 80 instances from CSPLib. Unlike a model using IloSequence, our SLIDE model does not combine reasoning about overall cardinality of a configuration with the sequence of AMONG constraints. Hence, it is not as efficient: 26 instances were solved with SLIDE within the five minute cutoff, compared to 39 with IloSequence. However, 9 of the instances solved with SLIDE were not solved by IloSequence. The memory overhead of the SLIDE propagator was not excessive despite the slid constraints having arity 5 and domains of size 30. The SLIDE model used on average 22Mb of space, compared to 5Mb for IloSequence.

8 RELATED WORK

Pesant introduced the REGULAR constraint, and gave a propagator based on dynamic programming to enforce GAC [16]. As we saw, the REGULAR constraint can be encoded using a simple SLIDE constraint. In this simple case, the dynamic programming machinery of Pesant’s propagator is unnecessary as the decomposition into ternary constraints does not hinder propagation. We have found that SLIDE is as efficient as REGULAR in practice [2]. Furthermore, our encoding introduces variables for representing the states. Access to the state variables may be useful (e.g. for expressing objective functions). Although an objective function can be represented with the COSTREGULAR constraint [11], this is limited to the sum of the variable-value assignment costs. Our encoding is more flexible, allowing different objective functions like the min function used in the example in Section 3.

Beldiceanu, Carlsson, Debruyne and Petit have proposed specifying global constraints by means of deterministic finite automata augmented with counters [6]. They automatically construct propagators for such automata by decomposing the specification into a sequence of signature and transition constraints. This gives an encoding similar to our SLIDE encoding of the REGULAR constraint. There are, however, a number of advantages of SLIDE over using an automaton. If the automaton uses counters, pairwise consistency is needed to guarantee GAC (and most constraint toolkits do not support pairwise consistency). We can encode such automata using a SLIDE where we introduce an additional sequence of variables for each counter. SLIDE thus provides a GAC propagator for such automata. Moreover, SLIDE has a better complexity than a brute-force pairwise consistency algorithm based on the dual encoding as it considers only the intersection variables, reducing the space complexity by a factor of $d$.

Hellsten, Pesant and van Beek developed a GAC propagator for the STRETCH constraint based on dynamic programming similar to that for the REGULAR constraint [13]. As we have shown, we can encode the STRETCH constraint and maintain GAC using SLIDE. Several propagators for the AMONGSEQ are proposed and compared in [21, 3]. Among these propagators, those based on the REGULAR constraint do the most pruning and are often fastest. Finally, Bartak has proposed a similar intersection encoding for propagating a sliding scheduling constraint [1] We have shown that this method is more general and can be used for arbitrary SLIDE constraints.

9 CONCLUSIONS

We have studied the CARDPATH constraint. This slides a constraint down a sequence of variables. We considered SLIDE a special case of CARDPATH in which the slid constraint holds at every position. We demonstrated that this special case can encode many global sequencing constraints including AMONGSEQ, CARDPATH, REGULAR in a simple way. SLIDE can therefore serve as a “general-purpose” constraint for decomposing a wide range of global constraints, facilitating their integration into constraint toolkits. We proved that enforcing GAC on SLIDE is NP-hard in general. Nevertheless, we identified several useful and common cases where it is polynomial. For instance, when the constraint being slid overlaps on just one variable or is monotone, decomposition does not hinder propagation. Dynamic programming or a variation of the dual encoding can be used to propagate SLIDE when the constraint being slid overlaps on more than one variable and is not monotone. Unlike the previous proposed propagator for CARDPATH, this achieves GAC. Our experiments demonstrated that using SLIDE to encode constraints can be as efficient and effective as specialised 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 slid when they have many satisfying tuples. We believe this could improve the efficiency of our propagator in many cases.

REFERENCES


