A Domain Consistency Algorithm for the Stretch Constraint

Lars Hellsten[1], Gilles Pesant[2], and Peter van Beek[1]

[1] University of Waterloo, Waterloo, Canada
[2] École Polytechnique de Montréal, Montreal, Canada
lars@uwaterloo.ca, pesant@crt.umontreal.ca, vanbeek@uwaterloo.ca

Abstract. The stretch constraint occurs in many rostering problems that arise in the industrial and public service sectors. In this paper we present an efficient algorithm for domain consistency propagation of the stretch constraint. Using benchmark and random instances, we show that this stronger consistency sometimes enables our propagator to solve more difficult problems than a previously proposed propagation algorithm for the stretch constraint. We also discuss variations of the stretch constraint that seem simple and useful, but turn out to be intractable to fully propagate.

1 Introduction

Many rostering and scheduling problems that arise in the industrial and public service sectors involve constraints on stretches of variables, such as limits on the maximum number of shifts a person may work consecutively and limits on the minimum number of days off between two consecutive work stretches. Typical examples arise in automotive assembly plants, hospitals, and fire departments, where both multi-shift rotating schedules and personalized schedules within some scheduling time window are often needed.

Pesant [5] introduced the stretch global constraint and offered a filtering algorithm for the constraint which is capable of providing significant pruning and works well on many realistic problems. In this paper, we present an algorithm based on dynamic programming that achieves a stronger, fully domain consistent level of constraint propagation. This stronger consistency, although more expensive to achieve by a linear factor, enables our algorithm to solve more difficult problems. We also present some natural extensions of the constraint that seem simple and useful, but turn out to be NP-complete to fully propagate.

In Section 2, we provide some background on the stretch constraint. Section 3 presents the main algorithm. A discussion of the algorithm's runtime and correctness is given in Section 4. Section 5 offers some empirical results and discusses some advantages and disadvantages of our algorithm. In Section 6, we examine extensions of the stretch constraint.

2 The Stretch Constraint

The `stretch` constraint applies to a sequence of n *shift variables*, which we will denote $s_0, s_1, \ldots, s_{n-1}$. We index from 0 to $n-1$ for convenience when performing modular arithmetic for cyclic instances. A `stretch` formulation also includes a set of m *values* representing *shift types*, $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_m\}$. Each variable s_i has associated with it a set $\mathrm{dom}(s_i) \subseteq \mathcal{T}$ representing values it is allowed to take. An *assignment* of values to variables is an n-tuple from the set $\mathrm{dom}(s_0) \times \cdots \times \mathrm{dom}(s_{n-1})$. We use the notation $\mathrm{value}(s_i)$ to denote the value assigned to s_i when its domain contains exactly one value.

For a given assignment of values to variables, a *stretch* is a maximal sequence of consecutive shift variables that are assigned the same value. Thus, a sequence $s_i, s_{i+1}, \ldots s_{i+k-1}$ is a stretch if $s_i = s_{i+1} = \cdots = s_{i+k-1}$, $i = 0$ or $s_{i-1} \neq s_i$, and $i + k = n$ or $s_{i+k} \neq s_i$. We say that such a stretch *begins* at s_i, has *span* (alternatively, *length*) k, and is of type $\mathrm{value}(s_i)$. We write $\mathrm{span}(s_j) = k$ once the value of s_j has been bound to denote the span of the stretch through s_j.

The following parameters specify a problem instance of `stretch`. A variable assignment is a solution if it satisfies the requirements specified by these parameters.

- $\Pi \subset \mathcal{T} \times \mathcal{T}$ is a set of ordered pairs, called *patterns*. A stretch of type τ_i is allowed to be followed by a stretch of type τ_j, $\tau_j \neq \tau_i$, if and only if $(\tau_i, \tau_j) \in \Pi$. Note that pairs of the form (τ_k, τ_k) are redundant, since by the definition of a stretch two consecutive stretches do not have the same value.
- $\mathit{shortest}[\tau]$ denotes the minimum length of any stretch of type τ.
- $\mathit{longest}[\tau]$ denotes the maximum length of any stretch of type τ.

We call a stretch of type τ through a variable s_j *feasible* if it satisfies

$$\mathit{shortest}[\tau] \leq \mathrm{span}(s_j) \leq \mathit{longest}[\tau].$$

A *feasible sequence* of stretches is a sequence of feasible stretches which do not overlap, cover a contiguous range of variables, and for which each pair of types of consecutive stretches is in Π.

The kind of rostering problems `stretch` solves can also be cyclic, in that the roster repeats itself and we assume there is no beginning or end. Such problems are similar enough to the non-cyclic version that only some slight changes to the algorithm and analysis are necessary. Therefore, the discussion in this paper applies to both versions, except where otherwise stated.

3 Propagation Algorithm

Here we present a propagator for `stretch` that enforces domain consistency. In general, a constraint is said to be domain consistent (also referred to as generalized arc consistent) if for each variable in the constraint, each value in the domain of the variable is part of a solution to the constraint (when the

constraint is looked at in isolation). A constraint can be made domain consistent by repeatedly removing values from the domains of the variables that could not be part of a solution to the constraint. Our domain consistency propagation algorithm for the stretch constraint enforces both stretch length constraints and allowed pattern constraints. It is based on a dynamic programming algorithm for the corresponding decision problem, and extended to perform pruning. Initially we will consider only the non-cyclic version.

3.1 Computing Reachability

The main observation we use is that any stretch that appears in a solution is independent of the stretches chosen before and after it, aside from the enforcement of the patterns. Pattern enforcement only depends on the variables adjacent to the stretch. Therefore, a particular stretch appears in a solution if and only if there exists a sequence of stretches ending with a compatible shift type covering the preceding variables, and a sequence of stretches beginning with a compatible shift type covering the subsequent variables.

An alternate way to look at the problem is one of computing reachability in a graph; the nodes in the graph are (variable, type) pairs, and edges correspond to stretches. Our goal is to find all edges that are in some path from some node (s_0, τ_i) to some node (s_{n-1}, τ_j). If the roster needs to be cyclic, then we may have edges between the ending and beginning positions, and will want to find a cycle rather than a path. The set of all edges found will indicate which values are part of some solution, and which values can be safely removed.

The basis of our algorithm is to use dynamic programming to compute, for each variable s_i, and type τ_j, whether there is a feasible sequence of stretches covering the variables $s_0, s_1, \ldots, s_{i-1}$ such that the value of s_{i-1} is compatible with τ_j with respect to Π. The results of this computation are stored in a table of values, *forward* (see Algorithm `ComputeForward`). Likewise, we compute whether there is a feasible sequence of stretches covering variables $s_{i+1}, s_{i+2}, \ldots, s_{n-1}$ such that the value of s_{i+1} is compatible with τ_j, and store the result in *backward* (see Algorithm `ComputeBackward`).

Once the reachability information is computed, it is used by a second step that prunes the domains. To make the first step as efficient as possible, we actually store the reachability information in *forward* and *backward* as arrays of prefix sums over the variables. The element $\mathit{forward}[\tau, i]$ indicates the number of variables s_j with $j < i-1$ such that a feasible sequence covers $s_0, s_1, \ldots, s_j$, and ends with a stretch type compatible with τ. The prefix sums allow us to, for a given type, query whether a compatible sequence ends within an arbitrary range in constant time. For example, the difference $\mathit{forward}[\tau, i+1] - \mathit{forward}[\tau, j]$ $(j \leq i)$ is greater than zero if and only if there is some feasible sequence of stretches beginning at s_0 and ending between s_{j-1} and s_{i-1} (inclusive) that is compatible with τ.

Another prefix array, *runlength*, is precomputed at the beginning of each stage of the algorithm. For each type τ and variable s_i, it stores the size of the maximal contiguous block of variables whose domains contain τ, up to and

including s_i (or including and following s_i for `ComputeBackward`). This gives an upper bound on the maximum length of a stretch ending (or beginning) at s_i, which may be less than $\mathit{longest}[\tau]$. Note that to make the algorithm concise, we use 1-based indices when looping over variables, rather than the 0-based indices used for shift variables. Indices 0 and $n+1$ correspond to initial values.

Algorithm `ComputeForward`()
1. $\mathit{forward}[\tau_j, 0] \leftarrow 0$ for all τ_j
2. $\mathit{forward}[\tau_j, 1] \leftarrow 1$ for all τ_j
3. $\mathit{runlength}[\tau_j, i] \leftarrow 0$ for all τ_j, i
4. **for** $j \leftarrow 1$ **to** m **do**
5. **for** $i \leftarrow 1$ **to** n **do**
6. **if** $\tau_j \in \mathrm{dom}(s_{i-1})$ **then** $\mathit{runlength}[\tau_j, i] \leftarrow \mathit{runlength}[\tau_j, i-1] + 1$
7. **for** $i \leftarrow 1$ **to** n **do**
8. **for** $j \leftarrow 1$ **to** m **do** $\mathit{forward}[\tau_j, i+1] \leftarrow \mathit{forward}[\tau_j, i]$
9. **for** $j \leftarrow 1$ **to** m **do**
10. $\mathrm{hi} \leftarrow i - \mathit{shortest}[\tau_j]$
11. $\mathrm{lo} \leftarrow i - \min(\mathit{longest}[\tau_j], \mathit{runlength}[\tau_j, i])$
12. **if** $\mathrm{hi} \geq \mathrm{lo}$ **and** $\mathit{forward}[\tau_j, \mathrm{hi}+1] - \mathit{forward}[\tau_j, \mathrm{lo}] > 0$ **then**
13. **for** $k \leftarrow 1$ **to** m **do**
14. **if** $(\tau_j, \tau_k) \in \Pi$ **then** $\mathit{forward}[\tau_k, i+1] \leftarrow \mathit{forward}[\tau_k, i] + 1$

Algorithm `ComputeBackward`()
1. $\mathit{backward}[\tau_j, n+1] \leftarrow 0$ for all τ_j
2. $\mathit{backward}[\tau_j, n] \leftarrow 1$ for all τ_j
3. $\mathit{runlength}[\tau_j, i] \leftarrow 0$ for all τ_j, i
4. **for** $j \leftarrow 1$ **to** m **do**
5. **for** $i \leftarrow n$ **downto** 1 **do**
6. **if** $\tau_j \in \mathrm{dom}(s_{i-1})$ **then** $\mathit{runlength}[\tau_j, i] \leftarrow \mathit{runlength}[\tau_j, i+1] + 1$
7. **for** $i \leftarrow n$ **downto** 1 **do**
8. **for** $j \leftarrow 1$ **to** m **do** $\mathit{backward}[\tau_j, i-1] \leftarrow \mathit{backward}[\tau_j, i]$
9. **for** $j \leftarrow 1$ **to** m **do**
10. $\mathrm{lo} \leftarrow i + \mathit{shortest}[\tau_j]$
11. $\mathrm{hi} \leftarrow i + \min(\mathit{longest}[\tau_j], \mathit{runlength}[\tau_j, i])$
12. **if** $\mathrm{hi} \geq \mathrm{lo}$ **and** $\mathit{backward}[\tau_j, \mathrm{lo}-1] - \mathit{backward}[\tau_j, \mathrm{hi}] > 0$ **then**
13. **for** $k \leftarrow 1$ **to** m **do**
14. **if** $(\tau_k, \tau_j) \in \Pi$ **then** $\mathit{backward}[\tau_k, i-1] \leftarrow \mathit{backward}[\tau_k, i] + 1$

The following small example demonstrates how our prefix sums are computed. We consider a roster with three shift types A, B, and C with the bounds on the stretch lengths and initial domains as shown in Table 1 and pattern set $\Pi = \{(A,B), (A,C), (B,A), (C,A)\}$. It is easy to see that the initial configuration is domain consistent. There are five solutions: {*AAABBBAA, AABBBAAA, AAACCCCC, CCCCCAAA, AACCCCAA*}. Each value present in the domains is used in at least one solution. Now, suppose we assign $s_2 \leftarrow A$ (see Table 5). This has the effect of reducing the set of possible solutions to {*AAABBBAA,*

$AAACCCCC\}$. When we run our algorithm, it should remove the value C from the domains of s_0, s_1, and s_2, and A from s_5.

Table 1. (left) Bounds on stretch length; (right) Initial domains;

τ_k	$shortest[\tau_k]$	$longest[\tau_k]$
A	2	4
B	3	3
C	4	5

s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
A	A	A			A	A	A
		B	B	B	B		
C	C	C	C	C	C	C	C

Table 2 shows a trace of the state after each iteration of the outermost loop in `ComputeForward`. Table 3 shows the final form of the *backward* table. It is constructed in the same manner as the *forward* table.

Table 2. Building the *forward* table.

$i=0$	0	1	2	3	4	5	6	7	8	9
A	0	1								
B	0	1								
C	0	1								

$i=1$	0	1	2	3	4	5	6	7	8	9
A	0	1	1							
B	0	1	1							
C	0	1	1							

$i=2$	0	1	2	3	4	5	6	7	8	9
A	0	1	1	1						
B	0	1	1	2						
C	0	1	1	2						

$i=3$	0	1	2	3	4	5	6	7	8	9
A	0	1	1	1	1					
B	0	1	1	2	3					
C	0	1	1	2	3					

$i=4$	0	1	2	3	4	5	6	7	8	9
A	0	1	1	1	1	1				
B	0	1	1	2	3	3				
C	0	1	1	2	3	3				

$i=5$	0	1	2	3	4	5	6	7	8	9
A	0	1	1	1	1	1	1			
B	0	1	1	2	3	3	3			
C	0	1	1	2	3	3	3			

$i=6$	0	1	2	3	4	5	6	7	8	9
A	0	1	1	1	1	1	1	2		
B	0	1	1	2	3	3	3	3		
C	0	1	1	2	3	3	3	3		

$i=7$	0	1	2	3	4	5	6	7	8	9
A	0	1	1	1	1	1	1	2	3	
B	0	1	1	2	3	3	3	3	4	
C	0	1	1	2	3	3	3	3	4	

$i=8$	0	1	2	3	4	5	6	7	8	9
A	0	1	1	1	1	1	1	2	3	4
B	0	1	1	2	3	3	3	3	4	5
C	0	1	1	2	3	3	3	3	4	5

Table 3. The final *backward* table.

	0	1	2	3	4	5	6	7	8	9
A	3	3	3	3	2	1	1	1	1	0
B	5	4	3	3	3	3	2	1	1	0
C	5	4	3	3	3	3	2	1	1	0

3.2 Pruning Values

Once we have computed the forward and backward reachability information, we are ready to begin pruning values from domains. This process proceeds by

considering, for each type, every possible stretch of that type. We check if a stretch is in a solution by examining the *forward* and *backward* tables to see if there are feasible sequences of stretches that can come before and after the one we are considering. If so, we mark the type we are considering, for each of the variables in the stretch. The final pruning step then prunes any value that has not been marked.

In order to make the marking linear in n for each τ_j, we traverse the variables in reverse order, maintaining a queue of possible ending positions. For each position i, we pop any elements from the front of the queue that cannot possibly end a stretch of type τ_j beginning at i. A position $j \geq i$ is not a possible ending position if $j - i + 1 > \mathit{longest}[\tau_j]$, or if there exists some k, $i \leq k \leq j$ such that the variable s_k does not contain τ_j in its domain, i.e. $j - i + 1 > \mathit{runlength}[\tau_j, i]$. Notice that if a position is not a valid ending position for i, it is also not valid for any position smaller than i, so it is always safe to remove invalid positions from the queue.

We also need to ensure that recording the marked intervals is efficient. However, this is easy, since the ending positions we consider are non-increasing. Therefore, each interval we add either extends the previous interval, or is disjoint from the previous interval. We end up with an ordered list of $O(n)$ disjoint intervals which cover a total of $O(n)$ values. We can therefore iterate through the list of intervals and mark all variables within each interval in $O(n)$ time. The details of this part of the algorithm are omitted for clarity.

Algorithm `MarkValues`()
1. $\mathit{runlength}[\tau_j, i] \leftarrow 0$ for all τ_j, i
2. **for** $j \leftarrow 1$ **to** m
3. **for** $i \leftarrow 1$ **to** n **do**
4. **if** $\tau_j \in \mathrm{dom}(s_{i-1})$ **then** $\mathit{runlength}[\tau_j, i] \leftarrow \mathit{runlength}[\tau_j, i-1] + 1$
5. **for** $j \leftarrow 1$ **to** m **do**
6. clear queue and list of intervals
7. **for** $i \leftarrow n$ **downto** 0 **do**
8. **if** $i > 0$ **and** $\mathit{backward}[\tau_j, i] - \mathit{backward}[\tau_j, i+1] > 0$ **then**
9. push $i - 1$ onto queue
10. **if** $\mathit{forward}[\tau_j, i+1] - \mathit{forward}[\tau_j, i] = 0$ **then continue**
11. **repeat**
12. $e \leftarrow$ front of queue
13. remove $\leftarrow (\mathit{longest}[\tau_j] < e - i + 1)$ **or** $(\mathit{runlength}[\tau_j, e] < e - i + 1)$
14. **if** remove = **true then** pop front of queue
15. **until** remove = **false or** queue is empty
16. **if** queue is not empty **then**
17. $e \leftarrow$ front of queue
18. **if** $e - i + 1 \geq \mathit{shortest}[\tau_j]$ **then**
19. merge $[i, e]$ into the interval list
20. **foreach** $(s, e) \in$ the list of intervals **do**
21. mark all (i, τ_j) where $s \leq i \leq e$

Table 4 shows an execution trace of `MarkValues` on the example from the previous section. Finally, Table 5 shows the result, in which domain consistency has been re-established.

Table 4. Trace of `MarkValues`.

		queue	
τ_j	i	contents	intervals
A	8	7	{}
	7	7	{}
	6	7	{[6, 7]}
	5		{[6, 7]}
	4		{[6, 7]}
	3	2	{[6, 7]}
	2	2	{[6, 7]}
	1	2	{[6, 7]}
	0	2	{[0, 2], [6, 7]}

		queue	
τ_j	i	contents	intervals
B	8		{}
	7		{}
	6	5	{}
	5	5	{}
	4	5	{}
	3	5	{[3, 5]}
	2		{[3, 5]}
	1		{[3, 5]}
	0		{[3, 5]}

		queue	
τ_j	i	contents	intervals
C	8	7	{}
	7	7	{}
	6	7, 5	{}
	5	7, 5, 4	{}
	4	7, 5, 4	{}
	3	7, 5, 4	{[3, 7]}
	2		{[3, 7]}
	1	0	{[3, 7]}
	0		{[3, 7]}

Table 5. (left) Domains after setting $s_2 \leftarrow A$; (right) Domains after re-establishing domain consistency.

s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
A	A	A			A	A	A
			B	B	B		
C	C		C	C	C	C	C

s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
A	A	A				A	A
			B	B	B		
			C	C	C	C	C

4 Analysis of the Algorithm

It is clear that the three stages of the algorithm all terminate, and that each primitive operation in the pseudo-code can be performed in $O(1)$ time. Examining the bounds of the **for** loops, we see that `ComputeForward` and `ComputeBackward` run in $O(nm^2)$ time, where n is the number of shift variables and m is the number of shift types. `MarkValues` runs in $O(nm)$ time, since we iterate through all variables once for each shift type, building a list of disjoint intervals over $[1, n]$. To achieve domain consistency, we simply need to run these three stages, and remove values which were not marked. The latter step can be performed in $O(nm)$ time, simply by iterating over an $n \times m$ table indicating which variable-value pairs (s_i, τ) are contained in some solution. The overall algorithm therefore runs in $O(nm^2)$ time, and requires $O(nm)$ space. In contrast, Pesant's original stretch propagator runs in $O(m^2l^2)$ time, where l is the maximum of the maximum lengths of the shift types. In applications of the stretch constraint that

have been seen thus far, l is a small value ($6 \leq l \leq 9$). Thus, our algorithm is more expensive to achieve by a linear factor.

One of the limitations of Pesant's algorithm that he discusses is its inability to consider the entire sequence. It is possible for a value in a variable's domain to be inconsistent because it is incompatible with the domain values of a different, far away variable in the sequence of shifts. Even though the domain filtering acts locally, considering variables near a variable that was assigned to, sometimes the changes will cascade throughout the roster. However, this is not always the case, and particularly for large instances can cause Pesant's filtering method to fail where ours succeeds.

The following is a small example that proves our algorithm achieves stronger propagation (see Table 6). We begin with an instance that is initially domain consistent, thus ensuring that any inconsistent values are introduced by incomplete pruning, and were not present to begin with. The problem instance is for a circular roster, with $n = 8$, $m = 3$, and no pattern restrictions (Π contains all ordered pairs of shift types).

Table 6. (left) Bounds on stretch lengths; (right) Initial domains.

τ_k	$shortest[\tau_k]$	$longest[\tau_k]$
A	2	4
B	5	5
C	2	4

s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
A	A	A	A			A	A
	B	B	B	B	B		
C				C	C	C	C

It is easy to see that this configuration is domain consistent. There are three solutions: AAAACCCC, ABBBBBAA, and CBBBBBCC. Each value of each variable is used in some solution. We first assign the value C to variable s_7, and then to variable s_0 (see Table 7).

Table 7. (left) Domains after setting $s_7 \leftarrow C$; (right) Domains after setting $s_0 \leftarrow C$.

Algorithm	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
Pesant's	A	A	A	A				
Algorithm		B	B	B	B	B		
	C				C	C	C	C
Domain	A	A	A	A				
Consistency		B	B	B	B	B		
	C				C	C	C	C

Algorithm	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
Pesant's		A	A	A				
Algorithm		B	B	B	B	B		
	C				C	C	C	C
Domain								
Consistency		B	B	B	B	B		
	C						C	C

After the second assignment, since no stretch of B's can be shorter than 5, clearly choosing the value A for s_1, s_2 or s_3 requires a stretch of 5 C's, which is a violation of the maximum stretch of C's. Therefore, after the second assignment,

the solution can be determined. Pesant's algorithm observes that it is possible to choose a stretch of A's or B's beginning at s_1 without violating the length constraints for those values, but does not consider the cascading effect of removing values from those domains.

Having shown that our algorithm enforces a stronger level of consistency than Pesant's algorithm, we now show that our algorithm achieves domain consistency. In order to justify that our algorithm achieves domain consistency, we must prove that a value is removed from a variable if and only if that value is not contained in some solution to the constraint. We turn our attention to some facts about the intermediate computations.

Lemma 1. *The prefix sums computed by* `ComputeForward` *and* `ComputeBackward` *record, for each type τ and position p, the number of previous positions for which some sequence of stretches exists that can be appended with a stretch of type τ. More precisely:*

(a) The value of $forward[\tau, p]$, $1 \leq p \leq n+1$ as computed by `ComputeForward` *is equal to the number of variables s_i with $i < p$ such that some feasible sequence of stretches spans variables $s_0, s_1, \ldots, s_{i-1}$, and is either the empty sequence, or ends with a stretch of type τ' where $(\tau', \tau) \in \Pi$.*

(b) The value of $backward[\tau, p]$, $0 \leq p \leq n$ as computed by `ComputeBackward` *is equal to the number of variables s_i with $i \geq p$ such that some feasible sequence of stretches spans variables $s_i, s_{i+1}, \ldots, s_{n-1}$, and is either the empty sequence, or begins with a stretch of type τ' where $(\tau, \tau') \in \Pi$.*

Proof. We prove (a) by induction on p. Case (b) is analogous and omitted. In the base case, $p = 1$, an empty sequence of stretches is compatible with any type. The algorithm handles this on lines 1-2. For $p > 1$, suppose as our induction hypothesis that the lemma holds for $p' < p$. Clearly this hypothesis implies that if we already know $forward[\tau, p-1]$, it is sufficient to take $forward[\tau, p] = forward[\tau, p-1]$ and then increment this count by one if and only if there is a sequence of stretches compatible with τ spanning $s_0, \ldots, s_{p-2}$. All feasible sequences ending at an earlier shift variable have already been counted. We must show that this is precisely what the algorithm does.

Consider iteration $p-1$ of the outer loop, on lines 7-14. This is the only time a given $forward[\tau, p]$ entry will be modified. Line 8 initializes $forward[\tau, p]$ to the count for the previous variables. All types τ' compatible with τ are considered in the j loop, on lines 9-14. The values lo and hi computed give the range of all possible starting points for a stretch of type τ' ending at s_{p-2}. Moreover, by our induction hypothesis, $forward[\tau', hi+1] - forward[\tau', lo]$ gives the number of variables between $lo-1$ and $hi-1$ (inclusive) where a sequence of stretches compatible with τ' ends. If this value is greater than zero, we can append a stretch of type τ' to one such sequence, and obtain a sequence of stretches spanning $s_0, \ldots, s_{p-2}$, which corresponds to the algorithm setting $forward[\tau, p] = forward[\tau, p-1] + 1$. If there is no such sequence, the count will never be incremented.

Theorem 1 (Correctness of the algorithm). `MarkValues` *marks a value* (p, τ_j) *if and only if there is some solution which assigns shift type* τ_j *to* s_p.

Proof. First, suppose (p, τ_j) is marked. This means that during iteration j of the outer **for** loop, and some iteration $u \leq p$ of the inner **for** loop, the interval $[u, v]$ was recorded, for some $v \geq p$. Thus, $\mathit{forward}[\tau_j, u+1] - \mathit{forward}[\tau_j, u] > 0$. By the lemma, there exists a feasible sequence of stretches spanning variables $s_0, s_1, \ldots, s_{u-1}$, such that the sequence is either empty, or the last stretch is compatible with τ_j. We also know that v was removed from the front of the queue, and must have been pushed onto the queue during some iteration $k \geq u$. Therefore, $\mathit{backward}[\tau_j, k] - \mathit{backward}[\tau_j, k+1] > 0$, so there exists some sequence of stretches spanning variables $s_{k+1}, s_{k+2}, \ldots, s_{n-1}$ such that the sequence is either empty, or the first stretch is compatible with τ_j. Moreover, line 13 removes any positions v' from the front of the queue which are too distant from u to satisfy $\mathit{longest}[\tau_j]$, or for which a stretch from u to v' is impossible. Meanwhile, line 18 ensures e is far enough from u to satisfy $\mathit{shortest}[\tau_j]$. Hence, we can form a feasible stretch of type τ_j covering variables $s_u, s_{u+1}, \ldots, s_v$, prefix this stretch with a sequence of stretches covering all of the preceding variables, and append to it a sequence of stretches covering all of the remaining variables, giving a solution which assigns τ_j to s_p.

Conversely, consider a solution which assigns τ_j to s_p. Let s_u and s_v be the first and last variables in the stretch containing τ_j. We have $\mathit{backward}[\tau_j, v+1] - \mathit{backward}[\tau_j, v+2] > 0$ by the lemma. Therefore, inside the j th iteration of the outermost loop, we will push v onto the queue when $i = v$. When $i = u$, we have $\mathit{forward}[\tau_j, u+1] - \mathit{forward}[\tau_j, u] > 0$, so the **repeat** loop will be entered. Following this loop, v must remain on the queue; the condition on line 13 cannot have been satisfied yet, as we know a stretch of type τ_j can exist spanning $s_u, s_{u+1}, \ldots, s_v$. Therefore, in line 21 $[u, v']$ will be added to the list of intervals, for some $v' \geq v$. It follows that all pairs (i, τ_j) such that $u \leq i \leq v$ will be marked, including (p, τ_j).

4.1 Cyclic Rosters

For the cyclic version of the problem we are not assured that a stretch starts at position 0, and the first and last stretch must differ. These requirements are easy to account for by simply trying all possible starting positions and starting values, and adding a check to ensure that a stretch that ends at position $n-1$ does not have the same value as the initial stretch. Naively, this adds a factor of $O(nm)$ to the running time. We can improve this by choosing some fixed variable s_i and only considering the possible stretches through s_i. Then the slowdown is at worst $O(m \times \max\{\mathit{longest}[\tau] : \tau \in \mathcal{T}\})$. By always choosing the variable s_i that minimizes the product $|\mathrm{dom}(s_i)| \times \max\{\mathit{longest}[\tau] : \tau \in \mathrm{dom}(s_i)\}$ we can greatly reduce this slowdown in most typical problems. If s_i is simply the most recently bound variable, we will have $|\mathrm{dom}(s_i)| = 1$, giving an upper bound of $O(\max\{\mathit{longest}[\tau] : \tau \in \mathcal{T}\})$ on the slowdown for cyclic instances.

5 Empirical Results

We implemented our new domain consistency algorithm for the stretch constraint (denoted hereafter as DC) using the ILOG Solver C++ library, Version 4.2 [3] and compared it to an existing implementation of Pesant's [5] algorithm which enforces a weaker form of consistency (denoted hereafter as WC; also implemented using ILOG Solver).

We compared the algorithms experimentally on various benchmark and random instances. The experiments on the benchmark instances were run on a 3.0 GHz Pentium 4 with 1 gigabyte of main memory. The experiments on the random instances were run on a 2.40 GHz Pentium 4 with 1 gigabyte of main memory.

We first consider benchmark problems. Our benchmark instances were gathered by Laporte and Pesant [4] and are known to correspond to real-life situations. The problems range from rostering an aluminum smelter to scheduling a large metropolitan police department (see [4] for a description of the individual problems). The constraint models of the benchmark instances combine one or more cyclic stretch constraints with many other constraints including global cardinality constraints [6] and sequencing constraints [7]. The problems are large: the largest stretch constraint has just over 500 variables in the constraint.

Table 8. Number of failed branches and CPU time (sec.) for cyclic rostering problems from the literature. The variable ordering heuristic fills in weekends first, followed by week days chronologically, and within days either (left) chooses the next shift using minimum domain size, or (right) lexicographically. The value ordering is random.

	WC		DC	
	fails	time	fails	time
atc-1	6	0.01	4	0.00
atc-2	11	0.01	0	0.08
atc-3	48335	13.02	9	0.06
atc-4	108	0.03	25	0.18
alcan-1	0	0.00	0	0.01
alcan-2	970	0.17	967	0.39
burns	1	0.00	80	0.08
butler	2	0.01	1	0.10
heller	3671	1.34	88	0.27
horot	0	0.01	0	0.01
hung	22	0.05	0	0.93
laporte	200	0.04	37	0.05
lau	3	0.01	0	0.09
mot-1	0	0.00	0	0.05
mot-2	9	0.01	9	0.05
mot-3	3331	0.76	17	0.06
slany1	0	0.00	0	0.00

	WC		DC	
	fails	time	fails	time
atc-1	2	0.00	2	0.01
atc-2	266	0.04	1	0.06
atc-3	100982	21.86	2	0.04
atc-4	2356	0.30	1	0.09
alcan-1	0	0.00	0	0.01
alcan-2	955	0.17	995	0.39
burns	6	0.01	6	0.04
butler	2	0.00	2	0.10
heller	3259	1.24	88	0.27
horot	1	0.01	0	0.01
hung	3	0.04	0	0.93
laporte	223	0.04	28	0.04
lau	6	0.01	0	0.09
mot-1	3	0.00	3	0.05
mot-2	9	0.01	9	0.05
mot-3	2799	0.59	39	0.08
slany1	0	0.01	0	0.00

On the benchmark problems, our DC propagator offers mixed results over the previously proposed propagator when considering just CPU time (see Table 8). When the number of fails is roughly equal, our stronger consistency can be slower because it is more expensive to enforce. However, our DC propagator also sometimes leads to substantial reductions in number of fails and CPU time. Overall, we conclude that our propagator leads to more robust and predictable performance on these problems. All of the benchmark problems are solved in under one second by our propagator, whereas the propagator based on the weaker form of consistency cannot solve one of the problems in less than twenty seconds.

Table 9. Number of failed branches, CPU time (sec.), and number of problems solved within 10 minutes when finding first solution for random cyclic stretch problems. Each fail and time value is the average of only the tests that completed within the time bound of 10 minutes. A total of 50 tests were performed for each combination of n and m.

		WC			DC		
n	m	fails	time	solved	fails	time	solved
50	4	7628.1	0.09	50	0	0.01	50
	6	100089.6	1.21	48	0	0.04	50
	8	138855.7	2.14	50	0	0.08	50
100	4	666002.1	10.17	42	0	0.06	50
	6	281044.4	3.15	38	0	0.15	50
	8	757859.3	11.32	40	0	0.30	50
200	4	246781.2	4.40	19	0	0.26	50
	6	3.5	2.64	24	0	0.59	50
	8	2.9	6.60	22	0	1.09	50
400	4	50653.1	1.19	15	0	1.02	50
	6	90051.8	1.75	17	0	2.40	50
	8	10.2	0.01	14	0	4.25	50

To systematically study the scaling behavior of the algorithm, we next consider random problems. In our first random model, problems were generated that consisted of a single *cyclic* stretch over n shift variables and each variable had its initial domain set to include all m shift types. The minimum $shortest[\tau]$ and the maximum $longest[\tau]$ of the lengths of any stretch of type τ where set equal to a and $a+b$ respectively, where a was chosen uniformly at random from $[1,4]$ and b was chosen uniformly at random from $[0,2]$. These particular small constants were chosen to make the generated problems more realistic, but the experimental results appear to be robust for other choices of small values. No pattern restrictions were enforced (all ordered pairs of shift types were allowed). A random variable ordering was used to approximate a realistic context in which fragments of the sequence may be preassigned or fixed through the intervention of other constraints. In such scenarios, there is no guarantee that the chosen variable ordering will be favourable to the stretch propagator. Predictable per-

formance under arbitrary variable orderings indicates that the propagator is robust. In these pure problems nearly all of the run-time is due to the stretch propagators. These problems are trivial for domain consistency, but not so for the weaker form of consistency. We recorded the number of problems that were not solved by WC within a fixed time bound (see Table 9). As n increases, the difference between DC and WC becomes dramatic.

Table 10. WC versus DC when finding first solution for random non-cyclic stretch problems: ten best improvements in time (sec.) of WC over DC and ten best improvements in time (sec.) of DC over WC. A total of 1500 tests were performed for each value of n. A blank entry means the problem was not solved within a 10 minute time bound.

	10 best for WC		10 best for DC	
n	WC	DC	WC	DC
100	0.05	0.13		0.00
	0.00	0.06		0.05
	0.00	0.06		0.05
	0.00	0.06	164.69	0.05
	0.00	0.06	145.58	0.05
	0.00	0.06	126.70	0.00
	0.00	0.06	51.64	0.05
	0.00	0.06	38.11	0.05
	0.00	0.06	0.80	0.00
	0.00	0.06	0.69	0.00

	10 best for WC		10 best for DC	
n	WC	DC	WC	DC
200	0.06	0.88		0.06
	0.05	0.77		0.06
	0.05	0.77		0.06
	0.05	0.55		0.06
	0.06	0.41		0.06
	0.13	0.48		0.06
	0.13	0.42		0.08
	0.11	0.42		0.11
	0.17	0.44		0.17
	0.17	0.42		0.17

In our second random model, problems were generated that consisted of a single *non-cyclic* stretch. The domain of each variable was set in two steps. First, the initial domain of the variable was set to include all m shift types ($m = 4$, 6, or 8). Second, each of the shift types was removed from the domain with some given probability p, $0.0 \leq p < 0.2$. The minimum and the maximum of the lengths of any stretch of type where set equal to a and $a + b$ respectively, where a was chosen uniformly at random from $[1, 25]$ and b was chosen uniformly at random from $[0, 2]$. No pattern restrictions were enforced and a random variable ordering was again used. The WC propagator finds these non-cyclic problems much easier than the previous cyclic problems. Nevertheless, on these problems whenever WC is faster than DC the improvement is negligible, whereas our DC propagator can be dramatically faster than WC (see Table 10).

6 Extending Stretch

Now that we have an efficient algorithm for the STRETCH constraint, we turn to the possibility of extending the approach. In this section, we present some

variations of the constraint that seem simple and useful, but turn out to be NP-complete to fully propagate.

It is often useful to force a shift of a certain type to occur at least once. For example, there may be a mandatory cleaning shift that we want to include in a daily roster, but we don't care when it occurs. One approach would be to simply schedule the cleaning shift at a fixed time by binding variables ahead of time to create a pre-defined stretch. This has the drawback that it may limit the possible arrangements for the remaining shifts though. It would be better if we were to incorporate this capability directly into the constraint:

$$\textsc{ForcedShiftStretch}(B_1, \ldots, B_m, s_0, \ldots, s_{n-1})$$

Here $\mathrm{dom}(B_i) = \{0, 1\}$ initially. When $B_i = 1$, a solution includes a stretch of type τ_i, and when it does not, $B_i = 0$. If we want to force a stretch of a certain type to appear, we can set the domain of the corresponding B_i variable accordingly.

Unfortunately, this constraint turns out to be much harder than Stretch.

Theorem 2. *Deciding whether an instance of* ForcedShiftStretch *has a solution is NP-complete.*

Proof. A witness for the problem is a set of supports, one for each value in each variable's domain. This is polynomial in n and m, which shows that the problem is in NP. To show completeness, we proceed with a reduction from HamiltonianPath.

An instance of HamiltonianPath consists of an undirected graph $G = (V, E)$, and the answer is "yes" if and only if there is a path in G that visits each vertex precisely once. We introduce a shift type τ_v for each vertex $v \in V$, and $|V|$ shift variables. Each shift variable's domain contains all types. For each edge $uv \in E$, we let $\tau_u\tau_v$ be a pattern in Π. Finally, set $B_v = \{1\}$ for each $v \in V$. It is easy to see that by construction, G contains a Hamiltonian path if and only if the corresponding ForcedShiftStretch instance has a solution, and the construction has size $O(|V|^2)$.

Corollary 1. *Enforcing domain consistency for* ForcedShiftStretch *is NP-hard.*

A consequence of this is that any stretch constraint which individually restricts the number of occurrences of each shift type is intractable. Consider the stretch constraint where the minimum length $\mathit{shortest}[\tau_i]$ and the maximum length $\mathit{longest}[\tau_i]$ of every stretch of type τ_i is replaced with a set l_i which contains the lengths of the possible stretches of that type. Unfortunately, it is intractable to enforce domain consistency on such a generalized stretch constraint.

As another example, consider the stretch constraint which is extended to include domain variables c_i, where c_i denotes the number of possible stretches of type τ_i. Such a constraint would prove useful in modeling real-life rostering problems (see [4] for examples). Unfortunately, it is intractable to enforce domain consistency on this extended stretch constraint. One can model restrictions

on the number of stretches of a certain type using a combination of a (regular) stretch constraint and a generalized cardinality constraint over auxiliary variables. However, the amount of pruning achieved by enforcing domain consistency on such a decomposition will necessarily be less than on the extended stretch constraint since individually the problems are tractable but the combination is intractable (see [2]).

On a more positive note, we are currently examining a simple extension to the stretch constraint where a single domain variable c is added with domain $[l, u]$, where c denotes the total number of stretches. Such a constraint appears promising as it can be used to model Beldiceanu's [1] change/$\neq$ constraint (and perhaps other constraints in the cardinality-path constraint family; see [1]), yet seems to require only small changes to our algorithm in order to enforce domain consistency on the constraint.

7 Conclusion

We presented an efficient algorithm for domain consistency propagation of the stretch constraint, proved its correctness, and showed its usefulness on a set of benchmark and random problems. We also discussed natural generalizations of the stretch constraint that seemed simple and useful, but turned out to be intractable to fully propagate.

An important problem that we have not addressed in this paper, but one that we are currently working on, is to make our propagator incremental to be more efficient in the case where only a few of the domains have been changed since the most recent previous run of the propagator.

References

1. N. Beldiceanu. Pruning for the cardinality-path constraint family. Technical Report T2001/11A, SICS, 2001.
2. C. Bessière, E. Hebrard, B. Hnich, and T. Walsh. The complexity of global constraints. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, San Jose, California, 2004.
3. ILOG S. A. ILOG Solver 4.2 user's manual, 1998.
4. G. Laporte and G. Pesant. A general multi-shift scheduling system. *J. of the Operational Research Society*, 2004. Accepted for publication.
5. G. Pesant. A filtering algorithm for the stretch constraint. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, pages 183–195, Paphos, Cyprus, 2001.
6. J.-C. Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 209–215, Portland, Oregon, 1996.
7. J.-C. Régin and J.-F. Puget. A filtering algorithm for global sequencing constraints. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pages 32–46, Linz, Austria, 1997.