Pruning for the *cardinality-path* Constraint Family

Nicolas Beldiceanu

**SICS**
Lägerhyddsvägen 18
SE-75237 Uppsala, Sweden
Email: nicolas@sics.se

August 22 2001, revised
SICS Technical Report T2000/11A
ISSN 1100-3154
ISRN: SICS-T--2000/11A-SE

**Abstract**  This paper presents generic propagation algorithms for the *cardinality-path* constraint family. This is a restricted form of the *cardinality* operator that allows stating constraints on sliding sequences of consecutive variables. Taking advantage of these restrictions permits coming up with more efficient algorithms. Moreover the paper shows how to extend these propagation algorithms in order to partially integrate external constraints that have to hold. From an application point of view the *cardinality-path* constraint allows to express a huge variety of regulation constraints occurring in personnel planning problems. This revised edition of the SICS report T2000/11 incorporates one correction as well as some minor improvements.

**Keywords**  Global constraint, cardinality, timetabling.
1 Introduction

The purpose of this paper is to introduce a new family of global constraints named \textit{cardinality-path} and to present generic propagation algorithms for this family. This family regroups a set of global constraints that were described in [1]. It allows stating constraints on sliding sequences of consecutive variables. More precisely the \textit{cardinality-path} family constraint has the form \( CTR(v_1, ..., v_n) \), where \( C \) is a domain variable\(^1\), \( \{v_1, ..., v_n\} \) is a collection of domain variables and \( CTR \) is a \( k \)-ary elementary constraint \((2 \leq k \leq n)\). The constraint holds iff:

\[
C = \sum_{i=1}^{n-k+1} \#CTR(v_i, ..., v_{i+k-1}),
\]

where \( \#CTR(v_i, ..., v_{i+k-1}) \) is equal to 1 if constraint \( CTR(v_i, ..., v_{i+k-1}) \) holds and 0 otherwise. Condition (1) expresses the fact that the \textit{cardinality-path} constraint holds if exactly \( C \) constraints out of the set \( \{CTR(v_1, ..., v_k) \wedge CTR(v_2, ..., v_{k+1}) \wedge \ldots \wedge CTR(v_{n-k+1}, ..., v_n)\} \) are satisfied. Constraint \( CTR \) is defined by the following functions that will be used in order to make our propagation algorithms generic:

- \( \text{enforce}_\text{CTR}(v_i, ..., v_{i+k-1}) \): adds constraint \( CTR(v_i, ..., v_{i+k-1}) \) to the constraint store,
- \( \text{enforce}_\text{NOT}_\text{CTR}(v_i, ..., v_{i+k-1}) \): adds the negation\(^2\) of constraint \( CTR(v_i, ..., v_{i+k-1}) \) to the constraint store.

The previous functions trigger constraint propagation that will be carried on until saturation. Failure detection should be independent from the order in which constraints \( CTR \) are posted. In addition we use also the following primitives:

- \( \text{create}_\text{choice}_\text{point} \): creates a choice point in order to be able to return to the current state later on,
- \( \text{backtrack} \): restores the state of the domain variables and of the constraint store as it was on the last call to \( \text{create}_\text{choice}_\text{point} \).

The \textit{cardinality-path} constraint family can be seen as a special case of the cardinality operator introduced in [3], where all the elementary constraints have the same type and where the elementary constraints can be ordered in a way such that each constraint has \( k-1 \) variables in common with its predecessor. The generic propagation algorithms that we present in this paper take advantage of this specific structure in order to derive stronger pruning than the one that can be achieved by those rules described in [3, pages 749-751].

\(^1\) A domain variable is a variable that ranges over a finite set of integers; \( \text{dom}(V) \) denotes the set of possible values of variable \( V \).

\(^2\) The negation of constraint \( CTR \) is denoted \( \neg CTR \); \( \neg CTR(v_1, ..., v_k) \) holds iff \( CTR(v_1, ..., v_k) \) does not hold.
The cardinality-path constraint family is also useful for those over constrained problems having the structure described in the previous paragraph. In this case it allows to get an upper bound of the maximum number of constraints that hold and to propagate in order to try to achieve this upper bound.

The next section presents some instances of the cardinality-path constraint family. Sections 3 and 4 show how to compute a lower and an upper bound of the number of elementary constraints that hold. Section 5 indicates how to prune variables $V_1,...,V_n$ according to the minimum and maximum value of $C$. Finally, the last section shows how to integrate external constraints within the previous propagation algorithms.

2 Examples of the cardinality-path Constraint Family

The purpose of this section is to provide various concrete examples of the cardinality-path constraint family. These examples are given in Table 1 and a possible practical use is provided for each of them at the end of this section.

The first column of Table 1 describes a member of the family in terms of the parameters of the cardinality-path family: it gives the initial lower and upper values for variable $C$ and defines the elementary constraint $CTR$. Finally the last column of Table 1 describes the parameters of a family member and provides an example where the constraint holds. In order to make these examples more readable, two spaces after the value of a variable $iV_i$ ($11+\leq k\leq i$) indicate that constraint $CTR(1,...,i+1)$ does hold. For instance, the example change3,4,3,4,4,3change of the first row of Table 1 denotes that all the next three following constraints hold change,4,3,4,4,3change and that constraint change3 does not hold. Note that the meaning of a family member can be derived from Condition (1) and from the first column of Table 1.

<table>
<thead>
<tr>
<th>Bound for $C$ and elementary constraint $CTR$</th>
<th>Member and example of a solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C:0..n-1$ $X_i \neq X_{i+1}$</td>
<td>change($C, {V_1,..,V_n}$) $\neq$</td>
</tr>
<tr>
<td>$C:0..n-1$ $X_i \neq X_{i+1}$</td>
<td>change(3, ${4,4,3,4,1}$) $\neq$</td>
</tr>
<tr>
<td>$(X_i+1)\mod L \neq X_{i+1}$</td>
<td>cyclic_change($C, L, {V_1,..,V_n}$)</td>
</tr>
<tr>
<td>$(X_i+1)\mod L \neq X_{i+1}$</td>
<td>cyclic_change(2, 5, ${2,3,4,0,2,3,1}$)</td>
</tr>
<tr>
<td>$C:0..n-1$ $(X_i+1)\mod L \neq X_{i+1} \land X_i &lt; L \land X_{i+1} &lt; L$</td>
<td>cyclic_change_joker($C, L, {V_1,..,V_n}$)</td>
</tr>
<tr>
<td>$C:0..n-1$ $</td>
<td>X_i - X_{i+1}</td>
</tr>
<tr>
<td>$C:0..n-1$ $</td>
<td>X_i - X_{i+1}</td>
</tr>
</tbody>
</table>

Table 1. Members of the cardinality-path constraint family
Constraints *change*, *cyclic_change*, *cyclic_change_joker*, *smooth*, *among_seq*, *sliding_sum* and *relaxed_sliding_sum* were respectively described at pages 43, 44, 45, 46, 40, 41 and 42 of [1]. From a practical point of view, the constraints of the previous table can be used for the following purpose:

- **change** can be used for timetabling problems in order to put an upper limit on the number of changes during a given period,
- **cyclic_change** may be used for personnel cyclic timetabling problems where each person has to work according to cycles that have to be sometimes broken,
- **cyclic_change_joker** may be used in the same context as the *cycle_change* constraint with the additional interpretation that holidays (i.e. those values that are greater or equal than $L$) are not subject to cyclic constraint,
- **smooth** can be used to put a limit on the number of drastic variations on a given attribute (for example the number of persons working on consecutive weeks),
- **number_of_rest** allows controlling the number of rest days over a period of work, where a rest day is a period of at least two consecutive days off and one work day,
- **among_seq** may be used to express frequency constraints for producing goods for which one can have different variants (for example the car sequencing problem [2]),
- **sliding_sum** allows to restrict the total number of working hours on periods of consecutives days,
- **relaxed_sliding_sum** has the same utility as the *sliding_sum* constraint, but in addition allows to express the fact that the rule may be broken sometimes.

More complete examples of utilization of the previous constraints can be found in [1]. The next two sections indicate respectively how to compute a lower and an upper bound for the number of elementary constraints that hold.

---

\[ X \text{ in Values is equal to 1 if } X \in \text{Values}, \text{ and 0 otherwise.} \]
3 Computing a Lower Bound of the Minimum Number of Elementary Constraints that Hold

The following greedy algorithm returns in \( min\_break \) the minimum number of elementary constraints that hold. It tries to impose the negation of constraint \( CTR \) on consecutives variables as long as no failure occurs. A failure will correspond to the fact that posting a new constraint \( \neg CTR \) leads to a contradiction. In order to keep the propagation implied by \( \text{enforce\_NOT\_CTR(Ui,..,Ui+k-1)} \) (line 11) local to the constraints we state, we duplicate variables \( V_1...V_n \). However, usual saturation is used for the constraints we enforce; in particular they can trigger each other until no more deduction is possible. Variables \( U_1...U_n \) will be deallocated when the last backtrack occurs.

```plaintext
1  exist_choice_point:=1;
2  create_choice_point;
3  copy variables V1..Vn to U1..Un;
4  min_break:=0;
5  i:=1;
6  WHILE i ≤ n-k+1 DO
7     IF exist_choice_point=0 THEN
8        exist_choice_point:=1;
9        create_choice_point;
10    END;
11    IF enforce_NOT_CTR(Ui,..,Ui+k-1) fails THEN
12       backtrack;
13       exist_choice_point:=0;
14       min_break:=min_break+1;
15    ENDIF;
16    i:=i+1;
17  ENDWHILE;
18  IF exist_choice_point THEN backtrack END;
```

Let’s call a maximal sequence, a sequence of consecutive variables \( V_r,...,V_s \) \((r≥1,s≤n,s−r+1≥k)\) such that:

- Propagation on the conjunction of constraints \( \neg CTR(V_r,..,V_{r+k-1}),\neg CTR(V_{s-k+1},..,V_s) \) does not find a contradiction\(^4\),
- \( s \) is equal to \( n \) or the propagation on the conjunction of constraints \( \neg CTR(V_r,..,V_{s-k-1}),\neg CTR(V_{s-k+2},..,V_{s+1}) \) finds a contradiction.

The greedy algorithm constructs a suite of maximal sequences of consecutive variables. It returns a valid lower bound since stopping a maximum sequence earlier will not allow expanding the next maximum sequence further on to the right. The lower bound may not be sharp since:

- It depends whether the propagation algorithm associated to constraint \( \neg CTR \) is complete\(^5\) or not.

\(^4\) This does not mean that there is a solution for this conjunction of constraints since the propagation may be incomplete.
It depends on if we have a global propagation algorithm, which can take into account or not the fact that consecutive constraints partially overlap \( (i.e. \) have \( k - 1 \) variables in common). For example, consider \(-CTR\) being the constraint \( V_i + V_{i+1} + V_{i+2} + V_{i+3} = 2 \). Furthermore assume we have four 0-1 domain variables \( V_1, V_2, V_3, V_4 \). Suppose now that we apply \(-CTR\) on each sliding sequence of four consecutive variables of the series \( 0, V_1, V_2, V_3, V_4 \) \( (\text{e.g. we have the two constraints} \ V_1 + V_2 + V_3 = 2 \ \text{and} \ V_4 + V_2 + V_3 + V_4 = 2) \). If each of the previous constraint is propagated in an independent way we will miss the fact that \( V_4 \) is equal to 0.

If \(-CTR\) is a constraint that involves more than 2 variables \((i.e. \ k > 2)\) then, the fact that we backtrack after a failure restores the domain of the variables to their initial state; however, since two consecutive maximum sequences have \( k - 2 \) variables in common, there is an interaction that is ignored by our algorithm.

We illustrate the previous algorithm with an example of the cyclic_change constraint that was introduced in [1, page 44] and described in row 2 of Table 1. The cyclic_change constraint is a member of the cardinality-path family constraint where \( CTR \) is the following binary constraint: \( (X_i + 1) \mod L \neq X_{i+1} \). \( L \) is a strictly positive integer. Constraint cyclic_change\((2,5,\{2,3,4,0,2,3,1\})\) holds since \((X_i + 1) \mod 5 \neq X_{i+1}\) is verified exactly 2 times, namely \((0+1) \mod 5 \neq 2\) and \((3+1) \mod 5 \neq 1\).

Let’s assume we have the constraint cyclic_change\((C,5,\{V_1,V_2,V_3,V_4,V_5,V_6,V_7,V_8\})\) with the following initial domains: \( V_1: \{0,3\}, \ V_2: \{2,3,4\}, \ V_3: \{0,4\}, \ V_4: \{0,1,2,3,4\}, \ V_5: \{0,1,2,3\}, \ V_6: \{0,2,4\}, \ V_7: \{0,1,2\}, \ V_8: \{0,1,2\}\). Table 2 gives the 2 maximum sequences \( V_1, V_2, V_3, V_4, V_5 \) and \( V_6, V_7, V_8 \) built by the algorithm in order to evaluate the minimum number of constraints that hold, namely 2 in this case. For each maximum sequence, a line in the table represents the constraint that is currently added and the state of the domains after posting that constraint.

---

5 A propagation algorithm for constraint \( CTR(V_1, ..., V_n) \) is called complete if after propagation \( \forall i \in \{1,2, ..., n\}, \forall v \in \text{dom}(V_i) \), there exists at least one feasible solution for \( CTR(V_1, ..., V_n) \) with \( V_i = v \).
Table 2. Maximum sequences of consecutive variables built by the algorithm

<table>
<thead>
<tr>
<th>First maximal sequence</th>
<th>Second maximal sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V_1 : {0,3} )</td>
<td>( V_6 : {0,2,4} )</td>
</tr>
<tr>
<td>((V_1 + 1) \mod 5 = V_2 : V_1 : {3} V_2 : {4} )</td>
<td>((V_6 + 1) \mod 5 = V_7 : V_6 : {0,4} V_7 : {0,1} )</td>
</tr>
<tr>
<td>((V_2 + 1) \mod 5 = V_3 : V_2 : {3} V_3 : {4} V_3 : {0} )</td>
<td>((V_7 + 1) \mod 5 = V_8 : V_6 : {0,4} V_7 : {0,1} V_8 : {1,2} )</td>
</tr>
<tr>
<td>((V_3 + 1) \mod 5 = V_4 : ) contradiction</td>
<td>((V_8 + 1) \mod 5 = V_9 : ) contradiction</td>
</tr>
</tbody>
</table>

4 Computing an Upper Bound of the Maximum Number of Elementary Constraints that Hold

We derive an upper bound (4) of the maximum number of elementary constraints that hold from the following two identities (2) and (3):

\[
D = \sum_{i=1}^{n-k+1} \#\negCTR(V_i,...,V_{i+k-1}), \tag{2}
\]

\[
C + D = n - k + 1, \tag{3}
\]

\[
\max(C) \leq n - k + 1 - \min(D). \tag{4}
\]

Identity (2) introduces quantity \( D \), which is the number of times that the negation of constraint \( CTR \) holds on variables \( V_1,...,V_n \) (i.e. the number of discontinuities). Identity (3) states that the number of elementary constraints that hold plus the number of constraints that do not hold is equal to the total number of constraints \( n - k + 1 \). Finally, Inequality (4) expresses the upper bound of the maximum number of elementary constraints that hold in term of the lower bound of the minimum number of continuities. In order to evaluate \( \min(D) \), we use the algorithm described in Section 3, where we replace \texttt{enforce\_NOT\_CTR} by \texttt{enforce\_CTR}. 
5 Pruning According to the Minimum and Maximum Number of Elementary Constraints that Hold

We use the following algorithm in order to prune variables $V_1,...,V_n$ according to the maximum value of variable $C$. We remove values that otherwise would cause a too big number of elementary constraints $CTR$ to hold.

```
  i:=1;
  FOR inc:=1 TO -1 (STEP -2) DO
    exist_choice_point:=1;
    create_choice_point;
    copy variables V1..Vn to U1..Un;
    min_break:=0;
    WHILE i≤i AND iSn-k+1 DO
      IF inc=1 THEN before[i]:=min_break
      ELSE after [i]:=min_break ENDIF;
      IF exist_choice_point=0 THEN
        exist_choice_point:=1;
        create_choice_point;
      ENDIF;
      IF enforce_NOT_CTR(Ui,..,Ui+k-1) fails THEN
        backtrack;
        exist_choice_point:=0;
        min_break:=min_break+1;
      ENDIF;
      IF inc=1 THEN record dom(Ui+k-1) in vbefore[i+k-1]
      ELSE record dom(Ui ) in vafter [i ] ENDIF;
      I:=i+inc;
    ENDWHILE;
    IF exist_choice_point THEN backtrack END;
    i:=n-k+1;
  ENDFOR;
  IF min_break=max(C) THEN
    FOR i:=1 TO n-k+1 DO
      IF before[i]+after [i]+1>max(C) THEN
        enforce_NOT_CTR(Vi,..,Vi+k-1);
      ENDIF;
    ENDFOR;
  ENDIF;
  IF max(C)-min_break≤1 THEN
    FOR i:=k TO n-k+1 DO
      IF before[i-k+2]+after[i-1]+2>max(C) THEN
        remove values v not in vbefore[i]∪vafter[i] from Vi;
      ENDIF;
    ENDFOR;
  ENDIF;
```

When `inc` is equal to 1, the first part of the algorithm (lines 1 to 25) computes the minimum number of constraints that hold in the set of constraints $\{CTR(V_i,...,V_j),CTR(V_{i+1},...,V_{j+1})\}$ for each $i$ between $1$ and $n-k+1$. This number is recorded in $before[i]$. We also initialize the sets of values $v$ for $i$ such that $k≤i≤n$.

---

6 When $i$ is equal to 1, the previous set is empty and $before[i]$ is equal to 0.
to the values that are still in the domain of variable $V_i$ just after propagating constraint $\neg CTR(V_{i-k+1},..,V_i)$. This is the first constraint that mentions variable $V_i$ when we scan the constraints from left to right.

When $\text{inc}$ is equal to $-1$, the first part of the algorithm (lines 1 to 25) computes the minimum number of constraints that hold in the set of constraints $\{CTR(V_{i+1},..,V_{i+k})...CTR(V_{n-k+1},..,V_n)\}$ for each $i$ between 1 and $n-k+1$. This number is stored in $\text{after}[i]$. We also initialize the sets of values $\text{vafter}[i]$ to the values that are still in the domain of variable $V_i$ just after propagating constraint $\neg CTR(V_{i},..,V_{i+k-1})$. This is the first constraint that mentions variable $V_i$ when we scan the constraints from right to left.

The $\text{min\_break}$ counter (line 17) is processed twice by the $+1$ and $-1$ loops (line 2). Because of the hypothesis made in the introduction “failure detection should be independent from the order in which constraints $CTR$ are posted” we get twice the same value for the $\text{min\_break}$ counter.

The second part of the algorithm (lines 26 to 32) enforces constraint $\neg CTR(V_{i},..,V_{i+k-1})$ to hold if the minimum number of constraints that hold in the set $\{CTR(V_{i-1},..,V_{i+k-2})\}$ plus the minimum number of constraints that hold in the set $\{CTR(V_{i+1},..,V_{i+k})...CTR(V_{n-k+1},..,V_n)\}$ is just equal to the maximum possible number of elementary constraints that hold.

The third part of the algorithm (lines 33 to 39) removes from a variable the values that cause two distinct additional elementary constraints to hold. We now prove that the third part of the algorithm removes only values that would lead to a failure of the cardinality-path constraint.

**CASE 1:** Assume that posting constraint $\neg CTR(V_{i-k+1},..,V_i)$ or constraint $\neg CTR(V_{i},..,V_{i+k-1})$ did generate a failure (line 14). Since we backtrack (line 15), $v_{\text{before}}[i]$ (line 19) or $v_{\text{after}}[i]$ (line 20) would contain all values of variable $V_i$. We derive from this fact that no value will be removed from variable $V_i$ (line 36).

**CASE 2:** Let us assume that posting constraint $\neg CTR(V_{i-k+1},..,V_i)$ and constraint $\neg CTR(V_{i},..,V_{i+k-1})$ did not generate any failure (line 14). In this case, we show that if $V_i \notin v_{\text{before}}[i] \cup v_{\text{after}}[i]$ then the minimum number of constraints of $\{CTR(V_{i-1},..,V_{i+k-2})...CTR(V_{n-k+1},..,V_n)\}$ that hold is greater than or equal to $\text{before}[i-k+2] + \text{after}[i-1] + 2$.

---

7 If $\neg CTR(V_{i-k+1},..,V_i)$ finds a contradiction then $v_{\text{before}}[i]$ is initialized to the initial domain of variable $V_i$.

8 When $i$ is equal to $n-k+1$, the previous set of constraints is empty and $\text{after}[i]$ is equal to 0.

9 If $\neg CTR(V_{i},..,V_{i+k-1})$ finds a contradiction then $\text{vafter}[i]$ is initialized to the initial domain of variable $V_i$. 
Let us note:

- \( \min(a, b) \) the minimum number of constraints that hold in the conjunction of CTR constraints \( CTR(V_{a}, \ldots, V_{a+k-1}) \land CTR(V_{a+k}, \ldots, V_{b+k-1}) \), where \( 1 \leq a \leq b \leq n-k+1 \).
- \( f \) the smallest value less than or equal to \( i-k+1 \) such that the following two conditions are true:
  - No failure was detected during the first iteration of the algorithm (when \( inc=1 \)) on the conjunction of constraints \( \neg CTR(V_{f}, \ldots, V_{f+k-1}) \land \neg CTR(V_{f+1}, \ldots, V_{f+k}) \land \cdots \land \neg CTR(V_{i-k+1}, \ldots, V_{i}) \).
  - \( f = 1 \) or a failure was detected after stating \( \neg CTR(V_{f-1}, \ldots, V_{f+k-2}) \).
- \( l \) the largest value greater than or equal to \( i \) such that the following two conditions are true:
  - No failure was detected during the second iteration of the algorithm (when \( inc=-1 \)) on the conjunction of constraints \( \neg CTR(V_{i}, \ldots, V_{i+k-1}) \land \neg CTR(V_{i+1}, \ldots, V_{i+k-2}) \land \cdots \land \neg CTR(V_{i+k}, \ldots, V_{i+k-1}) \).
  - \( l = n-k+1 \) or a failure was detected after stating \( \neg CTR(V_{i+1}, \ldots, V_{i+k}) \).

We have:

Partitioning the set of constraints \( \{ CTR(V_{1}, \ldots, V_{k}) \land CTR(V_{k+1}, \ldots, V_{i}) \} \) in the set \( \{ CTR(V_{1}, \ldots, V_{k}) \land CTR(V_{f-1}, \ldots, V_{f+k-2}) \} \) and in \( \{ CTR(V_{f}, \ldots, V_{f+k-1}) \land CTR(V_{i-k+1}, \ldots, V_{i}) \} \) leads to \( \min(l, i-k+1) \geq \min(l, f-1) + \min(f, i-k+1) \).

From the definition of \( f \) we have that: \( \min(l, f-1) \geq \text{before}[f] = \text{before}[i-k+2] \).

Since \( V_{i} \not\in \text{before}[f] \) we also have that: \( \min(f, i-k+1) \geq 1 \).

So we conclude that: \( \min(l, i-k+1) \geq \text{before}[i-k+2] + 1 \).

In a similar way, we have:

Partitioning the set of constraints \( \{ CTR(V_{1}, \ldots, V_{i+k-1}) \land CTR(V_{i+k}, \ldots, V_{n}) \} \) in the set \( \{ CTR(V_{1}, \ldots, V_{i+k-1}) \land CTR(V_{i}, \ldots, V_{i+k-1}) \} \) and in \( \{ CTR(V_{i+1}, \ldots, V_{i+k}) \land CTR(V_{n-k+1}, \ldots, V_{n}) \} \) leads to \( \min(l, n-k+1) \geq \min(l, i) + \min(l+1, n-k+1) \).

Since \( V_{i} \not\in \text{after}[l] \) we also have that: \( \min(l, i) \geq 1 \).

From the definition of \( l \) we have that: \( \min(l+1, n-k+1) \geq \text{after}[l] = \text{after}[i-1] \).

So we conclude that: \( \min(l, n-k+1) \geq 1 + \text{after}[l-1] \).

So the minimum number of constraints that hold in \( \{ CTR(V_{1}, \ldots, V_{k}) \land CTR(V_{n-k+1}, \ldots, V_{n}) \} \) is greater than or equal to \( \text{before}[i-k+2] + \text{after}[l-1] + 2 \).  

\( \square \)
Table 3 gives an example of execution of the previous algorithm on the example introduced at the end of Sect. 3.

**Table 3.** Tables before[], after[], vbefore[], vafter[] built by the algorithm

<table>
<thead>
<tr>
<th>variables</th>
<th>V_1</th>
<th>V_2</th>
<th>V_3</th>
<th>V_4</th>
<th>V_5</th>
<th>V_6</th>
<th>V_7</th>
<th>V_8</th>
<th>V_9</th>
</tr>
</thead>
<tbody>
<tr>
<td>domains</td>
<td>0,3</td>
<td>2,3,4</td>
<td>0,4</td>
<td>0,1,2,3,4</td>
<td>0,1,2,3</td>
<td>0,2,4</td>
<td>0,1,2</td>
<td>0,1,2</td>
<td>0,4</td>
</tr>
<tr>
<td>before[]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>vbefore[]</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0,2,4</td>
<td>0,1</td>
<td>1,2</td>
<td>0,4</td>
<td>0,4</td>
</tr>
<tr>
<td>after[]</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>vafter[]</td>
<td>3</td>
<td>3,4</td>
<td>0,4</td>
<td>2</td>
<td>3</td>
<td>0,4</td>
<td>0,1</td>
<td>0,1,2</td>
<td>0,1,2</td>
</tr>
</tbody>
</table>

If at most two constraints should hold then part 2 (lines 26-32) will perform the following pruning:

- since before[1]+after[1]+1=3>2, line 29 imposes constraint \((V_1 + 1) \mod 5 = V_2\), which fixes \(V_1\) to 3 and \(V_2\) to 4,
- since before[2]+after[2]+1=3>2, line 29 imposes constraint \((V_2 + 1) \mod 5 = V_3\), which fixes \(V_3\) to 0,
- since before[6]+after[6]+1=3>2, line 29 imposes constraint \((V_6 + 1) \mod 5 = V_7\), which removes value 2 from variables \(V_6\) and \(V_7\),
- since before[7]+after[7]+1=3>2, line 29 imposes constraint \((V_7 + 1) \mod 5 = V_8\), which removes value 0 from variable \(V_8\).

If at most two constraints should hold then part 3 (lines 33-39) will perform the following pruning:

- since before[2]+after[1]+2=4>2, line 36 removes values in \(\text{dom}(V_2) - (\text{vbefore}[2] \cup \text{vafter}[2]) = \{2,3,4\} - \{3,4\} = \{2\}\) from variable \(V_2\),
- since before[3]+after[2]+2=4>2, line 36 removes values in \(\text{dom}(V_3) - (\text{vbefore}[3] \cup \text{vafter}[3]) = \{0,4\} - \{0,4\} = \emptyset\) from variable \(V_3\),
- since before[4]+after[3]+2=3>2, line 36 removes values in \(\text{dom}(V_4) - (\text{vbefore}[4] \cup \text{vafter}[4]) = \{0,1,2,3,4\} - \{1,2\} = \{0,3,4\}\) from variable \(V_4\),
- since before[5]+after[4]+2=3>2, line 36 removes values in \(\text{dom}(V_5) - (\text{vbefore}[5] \cup \text{vafter}[5]) = \{0,1,2,3\} - \{2,3\} = \{0,1\}\) from variable \(V_5\),
- since before[6]+after[5]+2=4>2, line 36 removes values in \(\text{dom}(V_6) - (\text{vbefore}[6] \cup \text{vafter}[6]) = \{0,2,4\} - \{0,2,4\} = \emptyset\) from variable \(V_6\),
- since before[7]+after[6]+2=4>2, line 36 removes values in \(\text{dom}(V_7) - (\text{vbefore}[7] \cup \text{vafter}[7]) = \{0,1,2\} - \{0,1,2\} = \emptyset\) from variable \(V_7\),
- since before[8]+after[7]+2=4>2, line 36 removes values in \(\text{dom}(V_8) - (\text{vbefore}[8] \cup \text{vafter}[8]) = \{0,1,2\} - \{0,1,2\} = \emptyset\) from variable \(V_8\).
Finally, if at most three constraints should hold then part 3 (lines 33-39) will perform the following pruning:

- since before[3]+after[2]+2=4>3, line 36 removes values in dom(V_3)−\{v_{before[3]}∪v_{after[3]}\}={0,4}−\{0,4\}=∅ from variable V_3,
- since before[6]+after[5]+2=4>3, line 36 removes values in dom(V_6)−\{v_{before[6]}∪v_{after[6]}\}=[0,2,4]−\{0,2,4\}=∅ from variable V_6,
- since before[7]+after[6]+2=4>3, line 36 removes values in dom(V_7)−\{v_{before[7]}∪v_{after[7]}\}=[0,1,2]−\{0,1\}=[2] from variable V_7,
- since before[8]+after[7]+2=4>3, line 36 removes values in dom(V_8)−\{v_{before[8]}∪v_{after[8]}\}=[0,1,2]−\{0,1,2\}=∅ from variable V_8.

A similar pruning for variables V_1,...,V_n is done according to the minimum value of variable C. For this purpose we use the same algorithm, where we replace enforce_NOT_CTR by enforce_CTR and max(C) by n−k+1−min(C). The previous quantity is the maximum number of constraints of the form ¬CTR that hold.

6 Integrating the “External World”

The purpose of Section 6 is to show how to partially integrate external constraints within some of the propagation algorithms of the cardinality-path constraint family. This is not a very common approach, since usually most of the constraint algorithms are local to a given constraint. However, this is especially relevant for getting stronger propagation.

The algorithm of Section 3, which computes a lower bound of the minimum number of elementary constraints that hold, can be modified as follows. We do not duplicate (line 3) any more the variables V_1,...,V_n, but work directly on them. This will result in waking the constraints we state inside the algorithm but also the external constraints mentioning a variable for which the domain is reduced. Finally, this may produce shorter maximum sequences than those obtained by the original algorithm. If this were the case, this would allow getting an improved lower bound of the minimum number of elementary constraints that hold.

7 Conclusion and Open Questions

We have presented generic propagation algorithms for the cardinality-path constraint family. We have also showed how to extend these propagation algorithms in order to consider the influence of external constraints that share some variable in common with the cardinality-path constraint. As one can observe, one of the main advantages
of generic propagation algorithms is that they can be applied to all constraints having some internal structure [1] in common.

The following example shows that, even when the arity of the elementary constraint is equal to 2, our algorithm does not always find out that no solution exists. If we consider constraint cardinality_path(1, {0, V1, V2, V3, 0}, ≠) such that V1, V2, V3 are 0-1 domain variables, then there is no solution since the number of satisfied disequality constraints is even. However the current algorithm seems to make a complete pruning in the case of binary constraints such as the less or the greater constraints. From the previous remarks one can ask the following questions:

- For which class of binary constraints our algorithm leads to a complete pruning?
- For which class of binary constraints there is no need to perform saturation in order to get a complete pruning? In this case, propagation concerns only the elementary constraint that is currently posted and not the elementary constraints that were already posted.
- How to extend our algorithm in order to get more propagation for some classes of elementary constraints?

Acknowledgements

Thanks to Mats Carlsson and Per Mildner for useful comments on an earlier draft of this report.

References