Buffered Resource Constraint: Algorithms and Complexity

Christian Bessiere[1], Emmanuel Hebrard[2], Marc-André Ménard[3],
Claude-Guy Quimper[3], and Toby Walsh[4]

[1] CNRS, Université Montpellier, LIRMM
bessiere@lirmm.fr
[2] CNRS, Université de Toulouse, LAAS
hebrard@laas.fr
[3] Université Laval
marc-andre.menard.2@ulaval.ca, claude-guy.quimper@ift.ulaval.ca
[4] NICTA, University of New South Wales
toby.walsh@nicta.com.au

Abstract. The notion of buffered resource is useful in many problems. A buffer contains a finite set of items required by some activities, and changing the content of the buffer is costly. For instance, in instruction scheduling, the registers are a buffered resource and any switch of registers has a significant impact on the total runtime of the compiled code. We first show that sequencing activities to minimize the number of switches in the buffer is NP-hard. We then introduce an algorithm which, given a set of already sequenced activities, computes a buffer assignment which minimizes the number of switches in linear time, i.e., $O(nd)$ where n is the length of the sequence and d the number of buffered items. Next, we introduce an algorithm to achieve bound consistency on the constraint SWITCH, that bounds the number of changes in the buffer, in $O(n^2d + n^{1.5}d^{1.5})$ time. Finally, we report the results of experimental evaluations that demonstrate the efficiency of this propagator.

1 Introduction

We consider a special type of resource, a *buffer*, corresponding to a set of items required by some tasks. In order to process a task, all items required by the task must be present in the buffer. However, the buffer has a limited capacity, and adding a new item is costly. Therefore, one may want to minimize the total number of changes, or *switches*. For instance, in instruction scheduling, the buffer can stand for memory caches, and minimizing the number of switches corresponds to minimizing page faults. Alternatively, the buffer may correspond to the reels of colored threads on an embroidery machine, and minimizing the number of reels changes over a sequence helps reducing the overall processing time. Yet another example arises in the design of validation plans for satellite payload [3]. Here, each test requires some components of the payload to be in a given configuration, and again the total number of configuration changes during the test campaign is a significant factor of its total duration.

We show that achieving hybrid consistency on the BUFFEREDRESOURCE constraint, i.e., domain consistency on the integer variables and bound consistency on the set variables, is NP-hard. We therefore consider a decomposition involving an ALLDIFFERENT constraint to enforce that the sequence is a permutation of the original set of tasks, and the constraint SWITCH, that counts the number of *switches* along the sequence.

We introduce an algorithm for finding a support of SWITCH in linear time, that is, $O(nd)$ where n is the length of the sequence and d the number of items. Moreover, we show how bound consistency can be enforced in $O(n^2d + n^{1.5}d^{1.5})$ time using a flow representation.

Finally, we compare our filtering algorithm against a decomposition on two crafted optimization problems, albeit derived from industrial applications. In both cases the objective function is defined using one or several SWITCH constraints. In these experiments, the proposed propagation algorithm for SWITCH greatly outperforms the standard decomposition.

The paper is organized as follows: In Section 2 we recall some background about consistency on constraints involving both integer and set variables. In Section 3 we define the BUFFEREDRESOURCE and SWITCH constraints and discuss their complexities. Then, in Section 4, we introduce an algorithm for achieving bound consistency on the SWITCH constraint. Finally, in Section 5, we report experimental results.

2 Formal Background

A constraint satisfaction problem consists of a set of variables, each with a finite domain of values, and a set of constraints specifying allowed combinations of values for subsets of variables. We write $\text{dom}(X)$ for the domain of a variable X. For totally ordered domains, we write $\min(X)$ and $\max(X)$ for the minimum and maximum values. A solution is an assignment of values to the variables satisfying the constraints. We consider both *integer* and *set* variables. A set variable S is represented by its lower bound $\text{lb}(S)$ which contains the definite elements and an upper bound $\text{ub}(S)$ which also contains the potential elements.

Constraint solvers typically explore partial assignments enforcing a local consistency property using either specialized or general purpose propagation algorithms. Given a constraint C, a *bound support* on C is a tuple that assigns to each integer variable a value between its minimum and maximum, and to each set variable a set between its lower and upper bounds which satisfies C. According to [1], a bound support in which each integer variable is assigned a value in its domain is called a *hybrid support*. If C involves only integer variables, a hybrid support is a *support*. A constraint C is *bound consistent* (BC) iff for each integer variable X_i, its minimum and maximum values belong to a bound support, and for each set variable S_j, the values in $\text{ub}(S_j)$ belong to S_j in at least one bound support and the values in $\text{lb}(S_j)$ belong to S_j in all bound supports. A constraint C is *hybrid consistent* (HC) iff for each integer variable X_i, every value in $\text{dom}(X_i)$ belongs to a hybrid support, and for each set variable S_j, the

values in $\mathrm{ub}(S_j)$ belong to S_j in at least one hybrid support, and the values in $\mathrm{lb}(S_j)$ are all those from $\mathrm{ub}(S_j)$ that belong to S_j in all hybrid supports. A constraint C involving only integer variables is *generalized arc consistent* (*GAC*) iff for each variable X_i, every value in $\mathrm{dom}(X_i)$ belongs to a support.

If all variables in C are integer variables, hybrid consistency reduces to generalized arc-consistency, and if all variables in C are set variables, hybrid consistency reduces to bound consistency.

To illustrate these concepts, consider the constraint $C(X_1, X_2, S)$ that holds iff the set variable S is assigned exactly the values used by the integer variables X_1 and X_2. Let $\mathrm{dom}(X_1) = \{1, 3\}$, $\mathrm{dom}(X_2) = \{2, 4\}$, $\mathrm{lb}(S) = \{2\}$ and $\mathrm{ub}(S) = \{1, 2, 3, 4\}$. BC does not remove any value since all domains are already bound consistent. On the other hand, HC removes 4 from $\mathrm{dom}(X_2)$ and from $\mathrm{ub}(S)$ as there does not exist any tuple satisfying C in which X_2 does not take value 2. Note that as BC deals with bounds, value 2 was considered as possible for X_1.

3   The BUFFEREDRESOURCE and SWITCH Constraints

We consider the problem of performing a set of tasks, each requiring a set of resources to be available on a buffer, whilst bounding the number of switches of resources on the buffer. We are given:

- A maximum buffer size $\overline{k}_i$ and a minimum buffer usage $\underline{k}_i$ at time i;
- A set of resources $R = \{r_1, \ldots, r_m\}$;
- A set of tasks $\mathcal{T} = \{T_i\}_{1 \leq i \leq n}$, where each task $T_i \in \mathcal{T}$ is associated with the set of resources it requires, that is, $\forall i \in [1, n],\ T_i \subseteq R$.

Example 1. For instance, suppose that we want to embroid n garments. Each garment requires a set of colors. However, only $\overline{k}$ reels and therefore $\overline{k}$ different colors of thread can be loaded on the embroidery machine. Hence, whenever we embroid a garment requiring a color of thread that is not already mounted on the machine, we need to switch it with one of the currently mounted reels. Each such switch is time consuming. Therefore, the goal is to sequence the garments so that we minimize the number of reel changes. In other words, we want to compute a permutation of the tasks $p : [1, n] \mapsto [1, n]$ and an assignment $\sigma : [1, n] \mapsto 2^R$ of the buffer over time such that the items required by each task are buffered ($\forall 1 \leq i \leq n,\ T_i \subseteq \sigma(p_i)$), the size of the buffer is not exceeded ($\underline{k}_i \leq |\sigma(i)| \leq \overline{k}_i$) and the number of switches $\sum_{1 \leq i < n} |\sigma(i+1) \setminus \sigma(i)|$ is minimized.

We introduce the BUFFEREDRESOURCE constraint to model this pattern.

Definition 1 (BUFFEREDRESOURCE). *Let $X_1, \ldots, X_n$ be integer variables, $S_1, \ldots, S_n$ be set variables, $\underline{k}_1, \ldots, \underline{k}_n$ and $\overline{k}_1, \ldots, \overline{k}_n$ be integers, and M an integer variable. The constraint* BUFFEREDRESOURCE$([X_1, \ldots, X_n], [S_1, \ldots, S_n], [\underline{k}_1, \ldots, \underline{k}_n], [\overline{k}_1, \ldots, \overline{k}_n], M)$ *holds if and only if:*

1. *$\forall i, j \in [1, n],\ i \neq j \rightarrow X_i \neq X_j$ ($X_1, \ldots, X_n$ is a permutation)*
2. *$\forall i \in [1, n],\ \underline{k}_i \leq |S_i| \leq \overline{k}_i$ (the buffer has a bounded capacity)*
3. *$\forall i,\ T_i \subseteq S_{X_i}$ (when a task is processed, all required resources are buffered)*

4. $\sum_{1 \leq i < n} |S_{i+1} \setminus S_i| \leq M$ *(the number of switches is less than or equal to M)*

We shall see that this constraint is NP-hard (even if the buffer's size is fixed). Hence, throughout the rest of the paper we shall consider a decomposition:

$$\text{BufferedResource}(X_1, \ldots, X_n, S_1, \ldots, S_n, [\underline{k}_1, \ldots, \underline{k}_n], [\overline{k}_1, \ldots, \overline{k}_n], M) \Leftrightarrow$$
$$\text{AllDifferent}(X_1, \ldots, X_n)$$
$$\wedge \qquad\qquad \forall i \in [1, n], \ \underline{k}_i \leq |S_i| \leq \overline{k}_i$$
$$\wedge \qquad\qquad \forall i, \ T_i \subseteq S_{X_i}$$
$$\wedge \qquad \text{Switch}([S_1, \ldots, S_n], [\underline{k}_1, \ldots, \underline{k}_n], [\overline{k}_1, \ldots, \overline{k}_n], M)$$

And in particular the constraint Switch, defined as follows:

Definition 2 (Switch). *Let $S_1, \ldots, S_n$ be set variables, $\underline{k}_i$ a lower bound on the cardinality of S_i, $\overline{k}_i$ an upper bound on the cardinality of S_i, and M an integer variable. The constraint $\text{Switch}([S_1, \ldots, S_n], [\underline{k}_1, \ldots, \underline{k}_n], [\overline{k}_1, \ldots, \overline{k}_n], M)$ holds if and only if: $\forall i \in [1, n], \ \underline{k}_i \leq |S_i| \leq \overline{k}_i \ \wedge \ \sum_{1 \leq i < n} |S_{i+1} \setminus S_i| \leq M$*

Example 1 (Continued). Assume that we want to embroid 5 garments, each requiring one of 5 colors as shown in Fig. 1a, on a machine with 3 reels of thread. Let the domains shown Fig. 1b represent the possible permutations at some point during search. To this sequence of variables corresponds a sequence of set variables shown in Fig. 1c. Whereas the BufferedResource constraint defines the possible combinations for all X's and S's, the constraint Switch involves only the set variables. We illustrate a support for Switch with $M = 2$ in Fig. 1d and a feasible solution for BufferedResource corresponding to the permutation $2, 1, 3, 4, 5$ with 4 switches in Fig. 1e.

$T_1 = \{B, G, Y\}$	$X_1 = \{1, 2\}$	$\{B, G\} \subseteq S_1 \subseteq \{B, G, R, Y\}$	$S_1 = \{B, G, R\}$	$S_1 = \{B, G, R\}$
$T_2 = \{B, G, R\}$	$X_2 = \{1, 2\}$	$\{B, G\} \subseteq S_2 \subseteq \{B, G, R, Y\}$	$S_2 = \{B, G, R\}$	$S_2 = \{B, G, Y\}$
$T_3 = \{W, Y\}$	$X_3 = \{3, 4, 5\}$	$\{R\} \subseteq S_3 \subseteq \{B, R, W, Y\}$	$S_3 = \{B, W, R\}$	$S_3 = \{B, W, Y\}$
$T_4 = \{B, R, W\}$	$X_4 = \{3, 5\}$	$\{W, Y\} \subseteq S_4 \subseteq \{R, W, Y\}$	$S_4 = \{Y, W, R\}$	$S_4 = \{B, W, R\}$
$T_5 = \{R, W, Y\}$	$X_5 = \{4, 5\}$	$\{R, W\} \subseteq S_5 \subseteq \{B, R, W, Y\}$	$S_5 = \{Y, R, W\}$	$S_5 = \{Y, W, R\}$
(a) Garments	(b) Perm.	(c) Buffer bounds	(d) Support	(e) Solution

Fig. 1: Illustration of Example 1.

Theorem 1. *Achieving HC on BufferedResource is NP-hard.*

Proof. We reduce Hamiltonian path (on undirected graphs) to the problem of finding a satisfying solution to BufferedResource. Let $G = (V, E)$ be an undirected graph with $|V| = n$ and $|E| = m$. We build an instance of BufferedResource on the integer variables $[X_1, \ldots, X_n]$, the set variables $[S_1, \ldots, S_n]$ and switch variable M.

- For each $v_i \in V$, we have a task T_i requiring m items, that is, a set T_i of cardinality m that contains one value j for every edge $e_j \in E$ such that $v_i \in e_j$. In order to make sure that $|T_i| = m$, we use as fillers values appearing in no other task ($\{i * m + j\}_{1 \leq j \leq m - d(v_i)}$ where $d(v_i)$ is the degree of v_i).
- There is one integer variable X_i per vertex $v_i \in V$ with domain $\{1, \ldots, n\}$.
- There are as many set variables S_i as vertices in the graph G, with domains ranging from the empty set to the whole universe of values: $\{\} \subseteq S_i \subseteq \{1, \ldots, (n+1)m\}$ for each $v_i \in V$. The cardinality of each one of these set variables is $\underline{k}_i = \overline{k}_i = m$.
- The domain of M is set to the single value $(n-1)(m-1)$.

We first show that the existence of an Hamiltonian path entails the existence of a solution of BufferedResource on the construction above. We set the value of X_i to the rank of the vertex v_i in the Hamiltonian path. Observe that given a permutation, there is a unique valuation of the set variables satisfying the constraint: $\forall i \in [1, n]$, $S_{X_i} = T_i$. Consider any two consecutive set variables S_j, S_{j+1}. Their domains correspond to two consecutive vertices in the Hamiltonian path, hence they share exactly one value: the common edge between the two nodes. The number of switches between these two set variables is thus equal to $m - 1$, hence the total number of switches is $(n-1)(m-1)$.

Next we show that the existence of a solution entails the existence of an Hamiltonian path in G. Consider any two consecutive set variables S_j, S_{j+1}, and assume that $X_i = j$ and $X_k = j + 1$. There are two cases, either there exists an edge $e_x = (v_i, v_k)$ in E and then $T_i \cap T_k = \{x\}$ or such edge does not exist, and therefore $T_i \cap T_k = \emptyset$. Since there are $(n-1)$ consecutive pairs of set variables, and since the only possible value for M is $(n-1)(m-1)$, the first case must hold for every consecutive pair. We can therefore conclude that there exists a path visiting every vertex of the graph exactly once. □

Observe that the cardinality of each set variable S_i can be as low as 3 since Hamiltonian path is still NP-hard when the maximum degree of a vertex is 3. On the other hand, the proof above requires both the total number of resources and the bound on the number of switches to be large.

4   Filtering Algorithm for Switch

In this section, we show how bound consistency can be enforced on Switch in $O(n^2 d + n^{1.5} d^{1.5})$ time. First we introduce a greedy algorithm that finds an assignment minimizing the number of switches in $O(nd)$ time where d is the total number of resources. Let L be the number of switches of that assignment. Then we introduce a filtering procedure based on a network flow representation. The cost of the flow represents the number of switches and for each pair S_i, v such that $v \in \mathrm{ub}(S_i)$ there is an edge in the network that can receive a unit of flow if and only if the set S_i may contain the value v in a support. Observe that forcing $v \in S_i$ may never entail more than two extra switches in the optimal

assignment with L switches. Therefore, we need to prune the set variables only if the difference between L and the upper bound of M is at most 1.

The algorithm we propose therefore proceeds as follows. First, we compute a flow of minimum cost, which is a support, and provides a lower bound on M. Then, if $\max(M) - \min(M) \leq 1$, we consider the residual graph with respect to this flow, and re-weight the edges so as to eliminate negative costs. Then we find all null cycles and, if $\max(M) - \min(M) = 1$, all cycles of weight 1. Table 1 summarizes the complexity of these four steps.

Step	Complexity
Finding an optimal assignment	$O(nd)$
Re-weighting the residual graph	$O((nd)^{1.5})$
Finding null cycles	$O(nd)$
Finding cycles of weight 1	$O(n^2 d)$
Total	$O(n^2 d + n^{1.5} d^{1.5})$

Table 1: Summary of the algorithm and its complexity.

4.1 Finding a Support

We present an algorithm that greedily constructs a support for the SWITCH constraint. This support is optimal in the sense that it minimizes the number of switches. The algorithm `FindSupport` (Algorithm 1) successively assigns the variables S_1 to S_n to sets σ_1 to σ_n. At each step i, the algorithm computes a priority for each value. The lower the priority is for value v, the more likely the value v will belong to σ_i. While processing the variables S_i for $i = 1..n$, we maintain for each value v the index $\text{next}_\in(v) = \min(\{j \mid v \in \text{lb}(S_j), j \geq i\} \cup \{n+1\})$ that is the smallest variable index no smaller than i such that $v \in \text{lb}(S_j)$. We also maintain the index $\text{next}_\notin(v) = \min(\{j \mid v \notin \text{ub}(S_j), j \geq i\} \cup \{n+1\})$ that is the smallest variable index no smaller than i such that $v \notin \text{ub}(S_j)$. If $\text{next}_\in(v) < \text{next}_\notin(v)$, the value v will be required in the sequence $S_i \ldots S_n$ before it gets forbidden. We assign such a value a priority between 1 and n. If $\text{next}_\in(v) > \text{next}_\notin(v)$, the value v will be forbidden in the sequence $S_i, \ldots, S_n$ before it gets required. We assign such a value a priority between $n+2$ and $2n+1$. Finally, if $\text{next}_\in(v) = \text{next}_\notin(v)$, we assign to value v a priority of $n+1$. This later case only occurs if the value v is allowed to appear but not required to appear in every variable of the sequence $S_i, \ldots, S_n$. The insertion of a value that is not required or that does not belong to the previous set σ_{i-1} induces a unnecessary switch. Such a value is given a penalty of $2n+1$ on its priority. Once the priority is computed, we add the value $l_v = \text{prio}(v) \times (d+1) + v$ to a set L. From l_v, we can retrieve the value using the arithmetic operation $l_v \bmod (d+1)$. Moreover, the smaller the priority is, the smaller the value l_v is. The algorithm keeps a counter k of the number of values that can be added to σ_i without causing an unnecessary switch, i.e. a switch that is not due to the requirement $v \in \text{lb}(S_i)$. This counter is the cardinality that will be given to σ_i. If $k \notin [\underline{k}_i, \overline{k}_i]$, we update k to the closest value between $\underline{k}_i$ and $\overline{k}_i$. We call the algorithm `Selection`(L, k)

Algorithm 1: FindSupport($[S_1, \ldots, S_n], [\underline{k}_1, \ldots, \underline{k}_n], [\overline{k}_1, \ldots, \overline{k}_n]$)

---

    for $v = 1..d$ **do** $\text{next}_\in(v) \leftarrow \text{next}_\notin(v) \leftarrow 1$;
    $\sigma_0 \leftarrow \emptyset$;
    for $i = 1..n$ **do**
        $L \leftarrow \emptyset$, $k \leftarrow 0$;
        for $v \in \text{ub}(S_i)$ **do**
            if $\text{next}_\in(v) < i$ **then** $\text{next}_\in(v) \leftarrow i$;

1   **while** $\text{next}_\in(v) \leq n \wedge v \notin \text{lb}(S_{\text{next}_\in(v)})$ **do** $\text{next}_\in(v) \leftarrow \text{next}_\in(v) + 1$;
            if $\text{next}_\notin(v) < i$ **then** $\text{next}_\notin(v) \leftarrow i$;

2   **while** $\text{next}_\notin(v) \leq n \wedge v \in \text{ub}(S_{\text{next}_\notin(v)})$ **do** $\text{next}_\notin(v) \leftarrow \text{next}_\notin(v) + 1$;

$$\text{prio} \leftarrow \begin{cases} \text{next}_\in(v) & \text{if } \text{next}_\in(v) < \text{next}_\notin(v) \\ n+1 & \text{if } \text{next}_\in(v) = \text{next}_\notin(v) \\ 2(n+1) - \text{next}_\notin(v) & \text{if } \text{next}_\in(v) > \text{next}_\notin(v) \end{cases};$$

3   **if** $v \notin \text{lb}(S_i) \wedge v \notin \sigma_{i-1}$ **then** $\text{prio} \leftarrow \text{prio} + 2n + 1$ **else** $k \leftarrow k + 1$
            $L \leftarrow L \cup \{\text{prio} \times (d+1) + v\}$;
        $k \leftarrow \min(\max(k, \underline{k}_i), \overline{k}_i)$;
        $l_{\max} \leftarrow \text{Selection}(L, k)$;
        $\sigma_i \leftarrow \{l \bmod (d+1) \mid l \in L \wedge l \leq l_{\max}\}$;
    return $[\sigma_1, \ldots, \sigma_n]$;

---

to retrieve the kth smallest element in L. This algorithm has a running time complexity of $O(d)$ when implemented using a divide-and-conquer strategy and a randomized partition algorithm [4]. We finally retrieve the k smallest elements l_v from L and include their respective value v in the set σ_i. Some values might share a same priority, the algorithm breaks ties on the lexicographical order of the values in order not to obtain sets with cardinality greater than k. The vector $[\sigma_1, \ldots, \sigma_n]$ constitutes an optimal support to the constraint.

Theorem 2. *The algorithm* FindSupport *returns a support of* SWITCH *that minimizes the number of switches.*

Proof. Let $\text{next}_\in^i(v)$, $\text{next}_\notin^i(v)$, and $\text{prio}^i(v)$ be the values of $\text{next}_\in(v)$, $\text{next}_\notin(v)$ and $\text{prio}(v)$ at iteration i.

We consider an optimal solution σ that cannot be built by FindSupport. Let θ_i be the instantiation of S_i by FindSupport and let i be the first index such that $\theta_i \neq \sigma_i$. One of the three following propositions is true:

1. $\sigma_i \subset \theta_i$, hence $\exists v$ such that $|\{w \mid \text{prio}^i(w) < \text{prio}^i(v)\}| \leq |\theta_i|$, $v \notin \sigma_i$;
2. $\theta_i \subset \sigma_i$, hence $\exists v$ such that $|\{w \mid \text{prio}^i(w) < \text{prio}^i(v)\}| > |\theta_i|$, $v \in \sigma_i$;
3. $\sigma_i \not\subset \theta_i$ & $\theta_i \not\subset \sigma_i$, hence $\exists v, w$ s.t. $\text{prio}^i(v) < \text{prio}^i(w)$, $v \notin \sigma_i$ and $w \in \sigma_i$;

Case 1: Since $v \notin \sigma_i$, we have $v \notin \text{lb}(S_i)$ and $|\theta_i| > \underline{k}_i$. Therefore, $|\sigma_{i-1} \cup \text{lb}(S_i)| > \underline{k}_i$. It follows that $v \in \sigma_{i-1}$, hence adding v to σ_i does not add any v-switch (and might prevent one latter).

Case 2: Since $v \notin \theta_i$, we have $v \notin \text{lb}(S_i)$ and since $v \in \sigma_i$ we have $|\theta_i| > \underline{k}_i$. Therefore, $|\sigma_{i-1} \cup \text{lb}(S_i)| < \overline{k}_i$. It follows that $v \notin \sigma_{i-1}$, hence removing v from σ_i suppresses one v-switch (and might entail one latter).

Case 3: From now on, we assume $\mathrm{prio}^i(v) < \mathrm{prio}^i(w)$, $v \notin \sigma_i$ and $w \in \sigma_i$. We now show that for some $j > i$, we can swap all instances of w for v in the sets $\sigma_i, \ldots, \sigma_{j-1}$ whilst not increasing the number of switches. Let σ' be the solution obtained by this transformation on σ. For each such operation, we get strictly closer to a solution that can be obtained by `FindSupport`. We say that there is a v-switch at index i in solution σ iff $v \in \sigma_i \setminus \sigma_{i-1}$. Since the transformation only changes indices i to $j-1$ and values v and w, we only have to count v-switches and w-switches from index i to j.

Notice that $\mathrm{prio}^i(v) < \mathrm{prio}^i(w)$ & $w \in \sigma_{i-1}$ implies $v \in \sigma_{i-1}$. Indeed, in Line 3 of Algorithm 1, we make sure that values in $\sigma_{i-1} \cup \mathrm{lb}(S_i)$ have the best priority. Moreover, since w and v can be interchanged, none of them is in $\mathrm{lb}(S_i)$.

We first consider the case where only v is in the previous buffer: $w \notin \sigma_{i-1}$ and $v \in \sigma_{i-1}$. Let j be the minimum integer greater than i such that either $w \notin \sigma_j$ or $v \in \sigma_j$ or $w \in \mathrm{lb}(S_j)$ or $v \notin \mathrm{ub}(S_j)$ or $j = n+1$.

Now we count v- and w-switches on the solutions σ and σ'.

- On σ there is a w-switch at index i, and there may be a v-switch at index j.
- On σ' there is no v-switch, however there may be a w-switch at index j.

Therefore, in this case the transformation can only decrease the number of switches, or leave it unchanged.

Now we consider the case where both v and w are either in or out the previous buffer: $w \in \sigma_{i-1} \Leftrightarrow v \in \sigma_{i-1}$. Let j be the minimum integer greater than i such that either $w \notin \sigma_j$ or $v \in \sigma_j$ or $j = n+1$. We show that $v \in \mathrm{ub}(S_l)$ and $w \notin \mathrm{lb}(S_l)$ for all $i \le l < j$, i.e.,

$$j \le \mathrm{next}^i_\in(w) \ \textit{and} \ j \le \mathrm{next}^i_\notin(v) \tag{1}$$

Now, by hypothesis, $\mathrm{prio}^i(w) > \mathrm{prio}^i(v)$. There are two cases:

1. $\mathrm{next}^i_\in(v) < \mathrm{next}^i_\notin(i)v$: In this case, $\mathrm{next}^i_\in(v) < \mathrm{next}^i_\in(w)$. However, $v \in \sigma_{\mathrm{next}^i_\in(v)}$ entails $j \le \mathrm{next}^i_\in(v)$, therefore proposition 1 is correct in this case.
2. $\mathrm{next}^i_\in(v) \ge \mathrm{next}^i_\notin(i)v$: In this case, $\mathrm{next}^i_\in(w) \ge \mathrm{next}^i_\notin(w)$ and $\mathrm{next}_\notin(i)v > \mathrm{next}^i_\notin(w)$. However, $w \notin \sigma_{\mathrm{next}^i_\notin(w)}$ we have $j \le \mathrm{next}^i_\notin(w)$, therefore proposition 1 is correct in this case too.

Now we count v- and w-switches on the solutions σ and σ'.

- On σ there is a w-switch at index i iff $w \notin \sigma_{i-1}$. Moreover, if $v \in \sigma_j$ there is a v-switch at index j.
- On σ': there is a v-switch at index i iff $v \notin \sigma_{i-1}$. Moreover, if $w \in \sigma_j$ there is a w-switch at index j.

By definition of j, $w \in \sigma_j$ implies that $v \in \sigma_j$, hence there is a w-switch at index j in σ' only if there is a v-switch at index j in σ. Moreover, by hypothesis, $w \notin \sigma_{i-1} \Leftrightarrow v \notin \sigma_{i-1}$. It follows that the number of switches may not increase after the transformation. □

The increments on line 1 and 2 are executed at most n times for each value v. The `Selection` algorithm is called exactly n times and has a running time complexity of $O(d)$. Consequently, the algorithm `FindSupport` runs in time $O(nd)$.

4.2 Network Flow Model

Let $\sigma_1, \ldots, \sigma_n$ be an optimal solution with L switches, such as the one computed by `FindSupport`. For any set S_i, any value $v \in \mathrm{ub}(S_i)$ can be inserted into σ_i by adding at most 2 more switches (unless $|\,\mathrm{lb}(S_i)| = \overline{k}_i$). Indeed we can add a value v to S_i or replace any value $w \in S_i \setminus \mathrm{lb}(S_i)$ with v, entailing at most one switch with S_{i-1} and one switch with S_{i+1}. Hence no pruning is required unless $\max(M) - L < 2$,.

We therefore focus on the case where $\max(M) - L < 2$ and we describe an algorithm that finds the values that can be used without additional switches and the values that require exactly one additional switch.

We construct a network flow G_s where every flow of value d represents a solution to the SWITCH constraint. The network has the following nodes. For every $v \in \mathrm{ub}(S_i)$, we have the nodes a_i^v and b_i^v. For $0 \leq i \leq n$, we have a *collector* node C_i. The collector node C_0 is the source node and the collector node C_n is the sink node.

Each edge has a capacity of the form $[l, u]$ where l is the required amount and u is the allowed amount of flow that can circulate through the edge. The network G_s has the following edges:

1. An edge between the node a_i^v and b_i^v for every $v \in \mathrm{ub}(S_i)$
   (a) with capacity $[1, 1]$ if $v \in \mathrm{lb}(S_i)$;
   (b) with capacity $[0, 1]$ otherwise.
2. An edge between b_i^v and a_{i+1}^v of capacity $[0, 1]$ if these two nodes exist.
3. An edge between b_i^v and C_i of capacity $[0, 1]$ for $1 \leq i \leq n$ and $v \in \mathrm{ub}(S_i)$.
4. An edge between C_{i-1} and a_i^v of capacity $[0, 1]$ for $1 \leq i \leq n$ and $v \in \mathrm{ub}(S_i)$.
5. An edge between C_{i-1} and C_i of capacity $[d - \overline{k}_i, d - \underline{k}_i]$ for $1 \leq i \leq n$.

All edges have a cost of zero except the edges (C_{i-1}, a_i^v) for $2 \leq i \leq n$ and $v \in \mathrm{ub}(S_i)$ that have a cost of 1. The cost $c(f)$ of a flow f is the sum, over the edges, of the cost of the edge times the amount of flow on this edge, i.e. $c(f) = \sum_{(x,y) \in E} f(x, y) c(x, y)$. Figure 2 shows the network flow of Example 1.

Lemma 1. *Each flow of value d in G_s corresponds to an assignment of the* SWITCH *constraint where capacities are satisfied.*

Proof. We construct a solution $\sigma_1, \ldots, \sigma_n$ by setting $v \in \sigma_i$ if and only if the edge (a_i^v, b_i^v) accepts a unit of flow. Since the flow value is d and there are between $d - \overline{k}_i$ and $d - \underline{k}_i$ units of flow circulating through the edge (C_{i-1}, C_i), there are between $\underline{k}_i$ and $\overline{k}_i$ edges (a_i^v, b_i^v) accepting a unit of flow, thus $\underline{k}_i \leq |\sigma_i| \leq \overline{k}_i$. $\square$

Lemma 2. *The cost of a flow in G_s gives the number of switches in the solution.*

Proof. The amount of flow going out of the collector C_{i-1} to the nodes a_i^v indicates the number of switches between S_{i-1} and S_i. Since these edges have a cost of 1, the cost of the flow equals the number of switches in the solution. Note that a flow can go through the nodes b_{i-1}^v, C_{i-1}, and then a_i^v which counts as a switch. Such cases do not occur in a minimum-cost flow as the flow could go through the edge (b_{i-1}^v, a_i^v) at a lesser cost. $\square$
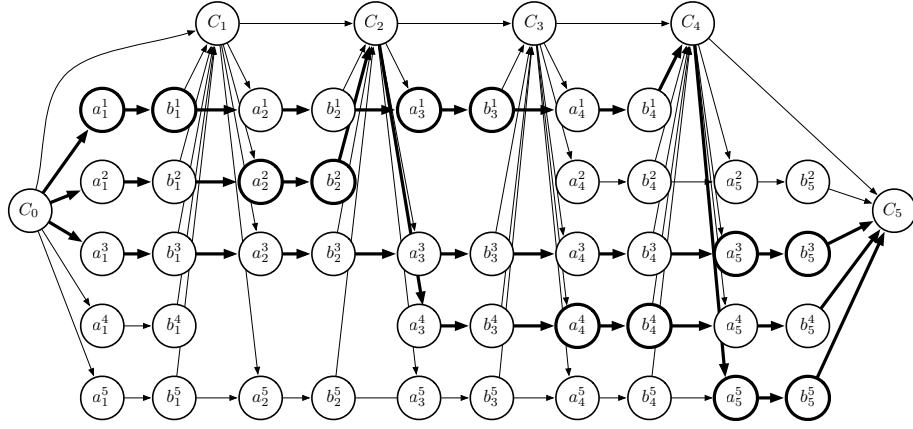
Fig. 2: The network flow associated to the SWITCH constraint of Example 1. Bold edges represent the flow. A pair of nodes (a_i^v, b_i^v) in bold indicates that $v \in \mathrm{lb}(S_i)$ and that the flow must go through the edge (a_i^v, b_i^v).

4.3 Bound Consistency

The function `FindSupport` computes a minimum-cost flow with a flow value of d. Let $f(x, y)$ be the amount of flow circulating on the edge (x, y) that has capacity $[l, u]$. The residual graph G_r has the same nodes as the original network but has the following edges. If $f(x, y) < u$, there is an edge (x, y) with capacity $[l - f(x, y), u - f(x, y)]$ and cost $c(x, y)$. If $f(x, y) > l$ there is an edge (y, x) with capacity $[f(x, y) - u, f(x, y) - l]$ and cost $-c(x, y)$. Note that in the residual graph, all costs are either -1, 0, or 1.

Theorem 3. *Let f be a flow of minimum cost in G_s, and G_r the corresponding residual graph, SWITCH is BC if and only if:*

1. *For all $v \in \mathrm{ub}(S_i)$, $f(a_i^v, b_i^v) > 0$ or the edge (a_i^v, b_i^v) belongs to a cycle of cost lower than or equal to $\max(M) - c(f)$ in G_r.*
2. *For all $v \in \mathrm{lb}(S_i)$, $f(a_i^v, b_i^v) > 0$ and there is no cycle of cost lower than or equal to $\max(M) - c(f)$ in G_r that involves the edge (b_i^v, a_i^v).*

Proof. By Lemma 1, we know that a flow in G_s corresponds to an assignment of $S_1, \ldots, S_n$ that satisfies the capacities of the sets. Moreover, by Theorem 2, we know that the support corresponding to f minimizes the number of switches. Hence, f witnesses the existence of a support for all $v \in S_i$ such that $f(a_i^v, b_i^v) > 0$. Now, if $f(a_i^v, b_i^v) = 0$, it is known that there is a flow going through the edge (a_i^v, b_i^v) iff there exists a cycle in G_r passing by the edge (a_i^v, b_i^v). Moreover, by Lemma 2, the cost of the minimum cycle passing by the edge (a_i^v, b_i^v) gives the number of required additional switches if $v \in S_i$ is forced. It follows that in this case, $v \in S_i$ has a support iff there exists a cycle of cost $\max(M) - c(f)$ or less going through (a_i^v, b_i^v) in the graph G_r.

The flow f satisfies the bounds and capacities of the sets. Therefore, if $f(a_i^v, b_i^v) = 0$ then v cannot be in the lower bound of S_i. Now, suppose that $f(a_i^v, b_i^v) > 0$. By a reasoning similar as above, we have that there exists a bound support where $v \notin S_i$ iff there exists a cycle of cost $\max(M) - c(f)$ or less going through (b_i^v, a_i^v) in the graph G_r. Therefore, $v \in \mathrm{lb}(S_i)$ is entailed iff there is no such cycle. $\qquad\square$

Recall that we assume $\max(M) \geq c(f) + 2$ where $c(f)$ is the number of switches in an optimal solution. Therefore, we are interested to find all cycles of cost 0 and 1. We now describe how can this be done efficiently.

Since computing the minimum cycles is as hard as finding shortest paths, we first perform a preprocessing operation that will eliminate negative weights in the residual graph. This preprocessing is the same as the one used in Johnson's all-pair shortest path algorithm and was also used by Régin to filter COST-GCC [7].

We add a dummy node z to the residual graph that is connected to all other nodes with an edge of null cost. We compute the shortest path from the dummy node z to all other nodes. This can be done using Goldberg's algorithm [5] which computes the shortest path in a graph in $O(\sqrt{|V|}|E|\log W)$ time where $|V|$ is the number of vertices, $|E|$ is the number of edges, and W is the greatest absolute cost of an edge. In our case, we have $|V| \in O(nd)$, $|E| \in O(nd)$, and $W = 1$ which leads to a complexity of $O((nd)^{1.5})$. Let $\pi(x)$ be the shortest distance between the dummy node z and the node x. Let (x, y) be an edge in the residual graph with cost $c(x, y)$. We re-weight this edge with the cost function $c_\pi(a, b) = c(a, b) + \pi(a) - \pi(b)$. It is known that with the new cost function, there are no negative edges and that the cost of any cycle remains unchanged.

Finding null cycles in the new re-weighted residual graph becomes an easy problem. Since there are no negative edges, a cycle is null if and only if all its edges have a null cost. We can thus compute the strongly connected components in the graph induced by the edges of null costs. All values $v \in \mathrm{ub}(S_i)$ such that the edge (a_i^v, b_i^v) belongs to a null cycle has a support without extra switches.

To compute the supports with one additional switch, we modify once more the residual graph. We delete all edges with a cost greater than 1. Such edges necessarily lie on a path of weight greater than 1 and are not relevant. Only edges with weight 0 and 1 remain in the graph. We create two copies of the graph. We connect the two copies as follows: if there is an edge (x, y) of cost $c_\pi(x, y) = 1$, then we delete this edge in both copies and add an edge from x in the first graph to y in the second graph. The intuition is that to travel from a node in the first graph to a node in the second graph, one must pass through an edge of cost 1. Moreover, it is not possible to cross twice such an edge. So all paths in the resulting graph have cost at most 1.

To find a cycle of cost 1 passing by the edge (a_i^v, b_i^v) in the original graph, one needs to find a path from b_i^v in the first graph to a_i^v in the second graph. The problem is therefore transformed to a problem of reachability.

Computing whether there is a path from each of the $O(nd)$ nodes b_i^v to their associated nodes a_i^v can be done using $O(nd)$ depth first search (DFS) for a total computational time of $O(n^2 d^2)$. However, we use a key information to decrease

this complexity. We know that to generate one more switch, the flow needs to pass by a collector. We can restrict our search to the cycles passing by a collector. For each of the n collectors C_i, we can compute with a DFS the nodes F_i^0 that can be reached from this collector with a forward path of cost 0 and the nodes F_i^1 that can be reached with a path of cost 1. While doing the DFS, we use two bitsets to represent F_i^0 and F_i^1 and flag the nodes in the appropriate bitsets depending whether the node belongs to the first copy of the graph or the second copy. We perform the same operation on the transposed graph to compute the nodes B_i^0 and B_i^1 that can be reached with a backward path from collector C_i with a cost of at most 0 and at most 1. We then compute the set of nodes P_{ub} that lie on a cycle of cost at most 1 that passes by a collector as follows.

$$P_{ub} = \bigcup_{i=1}^{n} \left(F_i^0 \cap B_i^1 \cup F_i^1 \cap B_i^0 \right) \tag{2}$$

For every $v \in \mathrm{ub}(S_i)$ such that $v \notin \sigma_i$ and $a_i^v \in P_{ub}$, then it is possible to modify the solution σ to obtain a new solution σ' with $v \in \sigma_i'$. One simply needs to push one unit of flow on the cycle on which lies the node a_i^v. Since this node has (a_i^v, b_i^v) has unique outgoing edge in the residual graph, the new flow will pass by (a_i^v, b_i^v) and will have at most one more switch.

We use the same idea to test whether $v \notin S_i$ has a bound support whenever $f(a_i^v, b_i^v) > 0$. We compute the set P_{lb} that contains all indices i such that there does not exist a cycle of cost at most 1 that passes by a collector and by the edge (b_i^v, a_i^v) as follows.

$$P_{lb} = \{i \mid \nexists j \text{ s.t. } (a_i^v \in B_j^0 \wedge b_i^v \in F_j^1) \vee (a_i^v \in B_j^1 \wedge b_i^v \in F_j^0)\} \tag{3}$$

For every $v \notin \mathrm{lb}(S_i)$ such that $v \in \sigma_i$ and $i \in P_{lb}$, then it would not be possible to modify the solution σ to obtain a new solution σ' with $v \notin \sigma_i'$. Indeed there is no alternative for the unit of flow going through (a_i^v, b_i^v) without increasing the cost above $\max(M)$. We can therefore deduce that v should be added to $\mathrm{lb}(S_i)$.

Each of the n DFS runs in time $O(nd)$. The union and intersection operations required for the computation of P_{ub} and P_{lb} can be done using bitwise conjunction and disjunctions in time $O(n^2d)$. So the computation of the cycles are done in time $O(n^2d)$.

5   Experimental Evaluation

We tested our propagator for SWITCH on two optimization problems based on industrial applications. However, they have been somewhat abstracted and simplified for the purpose of our experiments. Moreover, we randomly generated two sets of instances.[1] All experiments were run on Intel Core i5 2.30GHz machine with 6GB of RAM on Windows 7. For each problem, we have generated 50 instances of four classes, each defined by a tuple of parameters. We compare two

---

[1] Available at `http://homepages.laas.fr/ehebrard/switch/`

Choco programs that differ in the representation of the objective function. In the first model it is decomposed into a sum of reified LESSTHAN ($<$) constraints. In the second model, the objective function is represented using a single SWITCH constraint (for the first problem), or several (for the second problem). All other constraints are the same in both models. Finally, we used two search heuristics (Impact-based Search [6] and the Domain over Weighted Degree Heuristic [2], denoted respectively Impact and Wdeg).

Embroidery Scheduling. This first problem is derived from a real life scheduling problem in the textile industry involving job-dependent setup times.

A set of n *garments* have to be embroidered using m *machines*. Each garment is characterized by a set $col_i \subseteq \{1, \ldots, k\}$ of *colors* required for embroidering a given pattern. Last, at most c_j reels of threads can be mounted on machine j (i.e., at most c_j colors of threads can be used without changing the reels).

The load on each machine must be balanced, so we assume that each machine will process n/m garments. A feasible solution for this problem is a mapping of the garments to the machines $f : \{1, \ldots, n\} \mapsto \{1, \ldots, m\}$. Moreover, each garment i must be assigned a position s_i on the machine it is assigned to, and for each machine j the set of colors of threads S_g^j available when processing the the gth garment must be sufficient for embroidering that garment (Eq. 5) while taking into account the maximum number of reels that can be mounted on each machine (Eq. 6). However, whenever the next garment to be embroidered requires a color of thread that is not loaded on the machine, one need to turn the machine off, change some reels and restart the machine. The number of reel changes must therefore be minimized (Eq. 4).

$$\text{minimize :} \qquad \sum_{1 \leq j \leq m} \sum_{1 \leq g < n/m} |S_{g+1}^j \setminus S_g^j| \qquad (4)$$

$$\text{subject to :} \qquad \forall i \in \{1, \ldots, n\} \ \ col_i \subseteq S_{s(i)}^{f(i)} \qquad (5)$$

$$\forall j \in \{1, \ldots, m\}, g \in \{1, \ldots, n/m\} \ \ |S_g^j| \leq c_j \qquad (6)$$

We have a set of n integer variables with domain $\{1, \ldots, m\}$ standing for the mapping of garments to position in the sequence (we consider here the whole sequence obtained by concatenating the sequences on each machine). Then we have n set variables, one for each garment and standing for the set of colors of threads on the machine when embroidering the ith garment. Then we used a set of ELEMENT constraints to channel these two sets of variables. In the first model, the objective function is implemented through a decomposition using a SUM of reified LESSTHAN and MEMBER constraints. In the second model, the objective is stated as a sum of m SWITCH constraints (one per machine).

Test Sequencing. Next we consider the design of validation tests. A set of n *tests* have to be performed in order to verify a system involving a set F of k *features*. A test i is defined by a set $on_i \subseteq F$ of features that must be turned ON, and a set $off_i \subseteq F \setminus on_i$ of features that must be turned OFF while doing the test.

A *configuration* is a complete characterization of the system, represented by the set $C_j \subseteq F$ of features that are ON (all other features are considered OFF).

The verification will go through m *phases* during which the system will be in a given configuration, and a subset of the tests will be performed. A feasible solution is a sequence of m configurations $[S_1, \ldots, S_m]$ and a mapping of the tests to a phase $f : \{1, \ldots, n\} \mapsto \{1, \ldots, m\}$ such that each test is compatible with the configuration in which it is done (Eq. 8). Moreover, there are restrictions on the number of features that can be ON simultaneously (Eq. 9). Finally, the total duration of the test campaign depends on the set of features that need to be turned ON during the transitions between two configurations (Eq. 7).

$$\text{minimize} : \qquad \sum_{1 \leq j < m} |S_{j+1} \setminus S_j| \qquad\qquad (7)$$

$$\text{subject to} : \qquad \forall j \in \{1, \ldots, m\} \ \ l \leq |S_j| \leq u \qquad\qquad (8)$$

$$\forall i \in \{1, \ldots, n\} \ \ on_i \subseteq S_{f(i)} \ \& \ off_i \cap S_{f(i)} = \emptyset \qquad (9)$$

We use two straightforward models for this problem. In both models, we have a set of n integer variables with domain $\{1, \ldots, m\}$ standing for the mapping between tests to phases and m set variables standing for the configuration during phase j. Then we use a set of ELEMENT constraints to channel these two sets of variables. In the first model, the objective function is implemented through a decomposition using a SUM of reified LESSTHAN and MEMBER constraints. In the second model, the objective is stated as a SWITCH constraint.

Results. We generated 4 classes of 50 instances, for each of the two problems, parameterized respectively by number of garments, colors and machines, and number of tests, features and configurations. For each class, we report the mean value of the objective function over time in both models. Results are shown in Figure 3 and Figure 4, for the Embroidery and Test Sequencing problems, respectively. These curves were obtained by averaging the step functions corresponding to the runs of one algorithm on every instance of the class.

We observe that the model using the propagator for SWITCH clearly outperforms the decomposition model. In particular, the decomposition does not seem able to significantly improve the objective after the initial drop. It is interesting to notice that whilst Wdeg outperforms Impact on the decomposition model, it is slightly worse in the model using the global constraint (except for the classes $\langle 100, 2, 5 \rangle$ in the Embroidery benchmark and $\langle 300, 10, 15 \rangle$ for the Test Sequencing benchmark). This is actually not surprising since Wdeg relies heavily on the shape of the constraint network and is severely hindered when using a global constraint. Despite this, the propagator yields better results, irrespective of the heuristic that is used. On the Embroidery problem, the improvement on the objective value obtained by using the propagator ranges from 32 to 48% (counted on the best heuristic in each case). On the Test Sequencing problem, we observe more modest but still sizeable improvements, ranging from 12 to 29%.
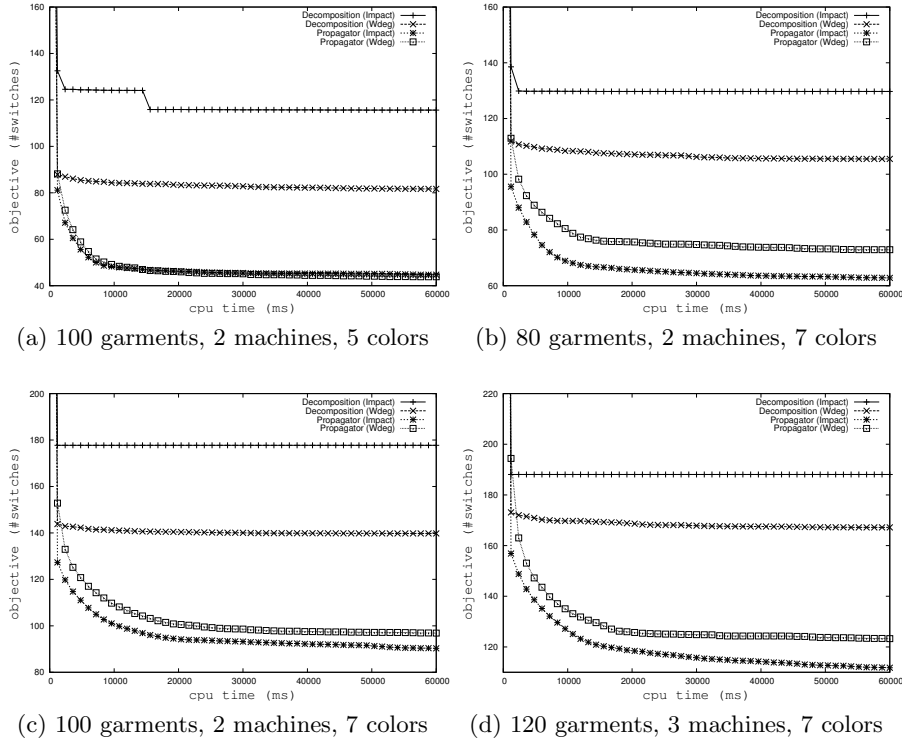
(a) 100 garments, 2 machines, 5 colors

(b) 80 garments, 2 machines, 7 colors

(c) 100 garments, 2 machines, 7 colors

(d) 120 garments, 3 machines, 7 colors

Fig. 3: Embroidery: #switches over time

6 Conclusion

We have introduced the constraints BUFFEREDRESOURCE and SWITCH to reason about the number of item switches in a buffered resource. The former constraint is NP-hard, however it can be effectively decomposed using the latter in conjunction with an ALLDIFFERENT. We have introduced a linear algorithm to find a support to the SWITCH constraint, that is, to assign a sequence of set variables standing for the buffer so that the number of switches is minimized. Furthermore, using this algorithm and a flow-based model, we have shown that bound consistency can achieved in $O(n^2d + n^{1.15}d^{1.5})$ time. Finally, our experimental results show that this propagator is a significant improvement with respect to expressing this relation with primitive constraints.

References

1. C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. The range and roots constraints: Specifying counting and occurrence problems. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 60–65, 2005.

(a) 150 tests, 15 config., 15 features

(b) 300 tests, 20 config., 10 features

(c) 300 tests, 20 config., 15 features
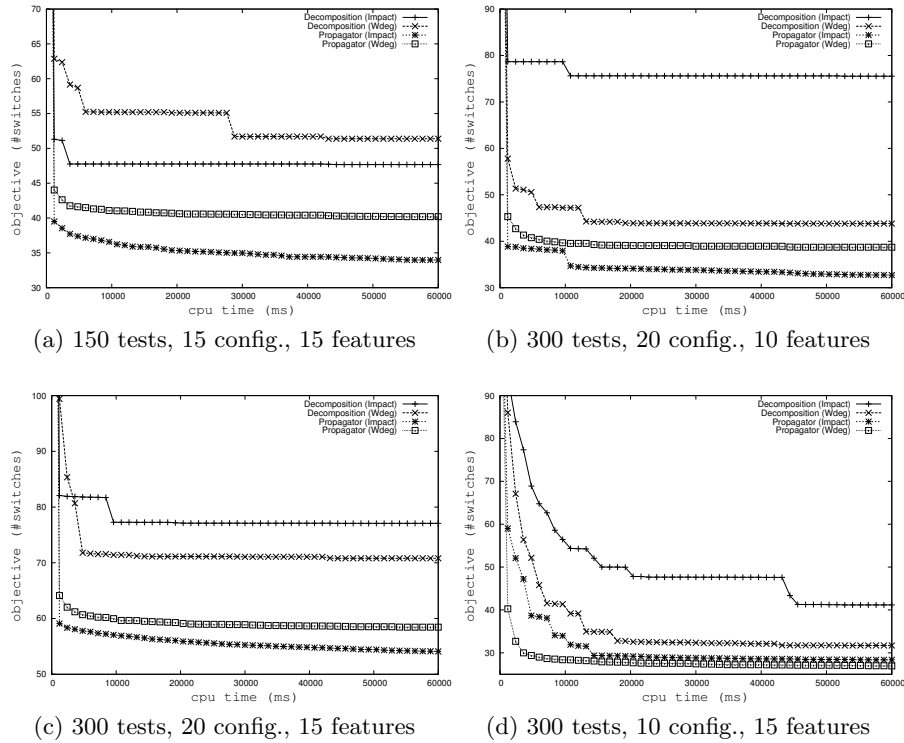

(d) 300 tests, 10 config., 15 features

Fig. 4: Test Sequencing: #switches over time

2. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting Systematic Search by Weighting Constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence - ECAI'04*, pages 146–150, 2004.
3. G. Verfaillie C. Maillet and B. Cabon. Constraint programming for optimising satellite validation plans. In *7th International Workshop on Planning and Scheduling for Space (IWPSS'11)*, 2011.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, third edition edition, 2009.
5. A. V. Goldberg. Scaling Algorithms for the Shortest Path Problem. *SIAM Journal on Computing*, 24:494–504, 1995.
6. P. Refalo. Impact-Based Search Strategies for Constraint Programming. In *Proceedings of Principles and Practice of Constraint Programming - CP'04. LNCS No. 3258*, pages 557–571, 2004.
7. J.-C. Régin. Cost-based arc consistency for global cardinality constraints. *Constraints*, 7(3–4):387–405, 2002.