Efficient Filtering for the Unary Resource with Family-based Transition Times

Sascha Van Cauwelaert, Cyrille Dejemeppe, Jean-Noël Monette*, Pierre Schaus

UCLouvain, ICTEAM,
Place Sainte Barbe 2,
1348 Louvain-la-Neuve, Belgium
{firstname.lastname}@uclouvain.be, jean-noel.monette@tacton.com

Abstract. We recently proposed an extension to Vilím's propagators for the unary resource constraint in order to deal with sequence-dependent transition times. While it has been shown to be scalable, it suffers from an important limitation: when the transition matrix is sparse, the additional filtering, as compared to the original from Vilím's algorithm, drops quickly. Sparse transition time matrices occur especially when activities are grouped into families with zero transition times within a family. The present work overcomes this weakness by relying on the transition times between families of activities. The approach is experimentally evaluated on instances of the Job-Shop Problem with Sequence Dependent Transition Times. Our experimental results demonstrate that the approach outperforms existing ones in most cases. Furthermore, the proposed technique scales well to large problem instances with many families and activities.

Keywords: Constraint Programming, Scheduling, Job-Shop, Sequence-Dependent Transition Times, Family, Global Constraint, Traveling Salesman Problem, Dynamic Programming, Lower Bound

1 Introduction

Unary resources with sequence-dependent transition times (also called set-up times) for non-preemptive activities are very frequent in real-life scheduling problems. A first example is the quay crane scheduling in container terminals [20], where the crane is modeled as a unary resource and transition times represent the moves of the crane on the rail between positions where it needs to load or unload containers. A second example is the continuous casting scheduling problem [9], where a set-up time is required between production programs.

Although efficient propagators have been designed for the standard unary resource constraint (UR) [16], transition time constraints between activities generally make the problem harder to solve because the existing propagators do not take them into account. We recently introduced in [5] a propagator for the unary

* This work was started during Jean-Noël's invited stay at UCLouvain in 2015.

resource constraint with transition times (URTT) as an extension to Vilím's algorithms, in order to strengthen the filtering in the presence of transition times.

Unfortunately, the additional filtering quickly drops in the case of a sparse transition time matrix, which typically occurs when activities are grouped into families with zero transition times within a family. The reason for a weak filtering with sparse matrices is that it is based on a shortest path problem with free starting and ending nodes and a fixed number of edges. The length of this shortest path drops in the case of zero transition times.

The main contribution of the present article is to introduce adapted filtering rules considering the families. The new propagator also relies on a shortest path problem but over a different underlying graph. The main asset of our approach is its scalability: we obtain an important amount of filtering while keeping a low time complexity of $O(n \log(n) \log(f))$, for n activities and f families. In general $f \ll n$, hence the theoretical complexity is very close to the one of the propagators in [16] and [5]. The filtering is experimentally tested on instances of the Job-Shop Problem with Sequence Dependent Transition Times (JSPSDTT), although it can be used for any type of problems, e.g., with other kinds of objective function than the makespan minimization. The results show that our propagator improves the resolution time over existing approaches and is more scalable.

The paper starts by providing the background for the considered problems in Section 2. The work on the URTT propagator [5] is also briefly recalled and its limitations are highlighted. Then, Section 3 presents the stronger filtering making use of the families. Section 4 reviews alternative approaches and Section 5 compares the results of the different approaches.

2 Background

Non-preemptive scheduling problems are usually modeled in constraint programming (CP) by associating three variables to each activity A_i: s_i, c_i, and p_i representing respectively the starting time, completion time, and processing time of A_i. These variables are linked together by the following relation: $\mathrm{s}_i + \mathrm{p}_i = \mathrm{c}_i$. Depending on the problem, the scheduling of the activities can be restricted by the availability of different kinds of resources required by the activities. In this work, we are interested in the unary resource (sometimes referred to as a machine or a disjunctive resource) and the propagators associated to one unary resource. Let T be the set of activities requiring the considered unary resource. The unary resource constraint prevents any two activities in T to overlap in time:

$$\forall A_i, A_j \in T : A_i \neq A_j \implies (\mathrm{c}_i \leq \mathrm{s}_j) \vee (\mathrm{c}_j \leq \mathrm{s}_i)$$

The unary resource can be generalized by requiring transition times between activities. The transition times are described by a square matrix $\mathcal{TT}$ in which $tt_{i,j}$, the entry at line i and column j, represents the minimum amount of time that must occur between the activities A_i and A_j when A_i directly precedes A_j. We assume that transition times respect the triangular inequality. That is,

inserting an activity between two activities never decreases the transition time between these two activities: $\forall A_i, A_j, A_k \in T : tt_{i,j} \leq tt_{i,k} + tt_{k,j}$.

The unary resource with transition times constraint imposes the following relation:

$$\forall A_i, A_j \in T : A_i \neq A_j \implies (\mathrm{c}_i + tt_{i,j} \leq \mathrm{s}_j) \vee (\mathrm{c}_j + tt_{j,i} \leq \mathrm{s}_i) \tag{1}$$

The earliest starting time of an activity A_i is denoted est_i and its latest starting time is denoted lst_i. The domain of s_i is thus the interval $[est_i; lst_i]$. Similarly the earliest completion time of A_i is denoted ect_i and its latest completion time is denoted lct_i. The domain of c_i is thus the interval $[ect_i; lct_i]$. These definitions can be extended to a set of activity Ω. For instance, est_Ω is the earliest time when any activity in Ω can start and ect_Ω is the earliest time when all activities in Ω can be completed. We also define $p_\Omega = \sum_{A_j \in \Omega} \mathrm{p}_j$ to be the sum of the processing times of the activities in Ω. While one can directly compute $est_\Omega = \min\{est_j | A_j \in \Omega\}$ and $lct_\Omega = \max\{lct_j | A_j \in \Omega\}$, it is NP-hard to compute the exact values of ect_Ω and lst_Ω [16]. Instead, one usually computes a lower bound for ect_Ω and an upper bound for lst_Ω. The propagators of [16] and [5] allow to compute efficiently such lower bounds, but have limitations in the presence of family-based transition times.

2.1 Propagator for the Unary Resource

The filtering rules presented in [16] for the UR constraint fall in several categories known as Overload Checking (OC), Detectable Precedences (DP), Not-First/Not-Last (NF/NL), and Edge Finding (EF). The implementation of these filtering rules runs in $\mathcal{O}(n \log(n))$, with $n = |T|$. It relies on an efficient computation of a lower bound ect^{LB0}_Ω of the earliest completion time of a set of activities $\Omega \subseteq T$, defined as:

$$ect^{LB0}_\Omega = \max_{\Omega' \subseteq \Omega} \{est_{\Omega'} + \mathrm{p}_{\Omega'}\} \tag{2}$$

The rules OC, DP, and NF/NL, rely on the so-called Θ-tree data structure, while EF relies on the Θ-Λ-tree data structure. The Θ-tree and the Θ-Λ-tree are used to compute efficiently and incrementally ect^{LB0}_Θ on a set of activities Θ. For instance, the OC rule is used to detect when $ect^{LB0}_\Theta > lct_\Theta$ for any $\Theta \subseteq T$, which triggers a failure. We refer the reader to [16] for a detailed description of this and the other rules. The following example illustrates the missed filtering for UR when it does not consider the transition times globally.

Example 1 *Consider a set of 3 activities $\Omega = \{A_1, A_2, A_3\}$ as shown in Figure 1. Consider also, for simplicity, that all pairs of activities from Ω have the same transition time $tt_{i,j} = 3$. The OC rule detects a failure when $ect^{LB0}_\Omega > lct_\Omega$. The filtering as described in [16] computes:*

$$ect^{LB0}_\Omega = est_\Omega + \sum_{A_i \in \Omega} p_i = 0 + 5 + 5 + 3 = 13$$

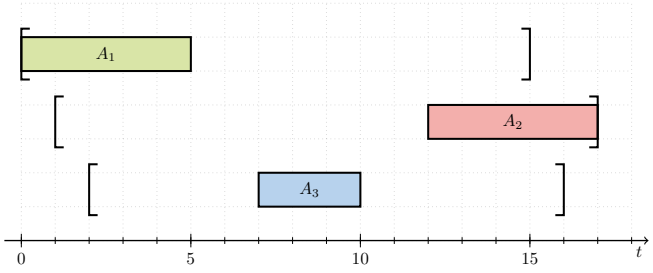

Fig. 1: Example illustrating the missed failure detection of OC when not considering transition times.

As we have $lct_\Omega = \max_{A_i \in \Omega} lct_i = lct_2 = 17$*, the OC rule from [16], combined with the transition times binary decomposition (Equation (1)), does not detect a failure. However, as there are 3 activities in* Ω*, at least two transitions occur between these activities and it is actually not possible to find a feasible schedule. Indeed, taking these transition times into account, one could compute* $ect_\Omega = 13 + 2 \cdot tt_{i,j} = 13 + 2 \cdot 3 = 19 > 17 = lct_\Omega$*, and thus detect the failure.*

2.2 Propagator for the Unary Resource with Transition Times

In [5], we extended Vilím's work [16] to the case of the unary resource with transition times constraint. By extending the Θ-tree and the Θ-Λ-tree to take the transition times into account to compute a lower bound of ect_Θ, we could strengthen the filtering without increasing the time and space complexities.[1]

Let Π_Ω be the set of all possible permutations of activities in Ω. For a given permutation $\pi \in \Pi_\Omega$, where $\pi(i)$ is the activity taking place at position i, we can define the total time spent by transition times, tt_π, as follows:

$$tt_\pi = \sum_{i=1}^{|\Omega|-1} tt_{\pi(i),\pi(i+1)}$$

A lower bound for ect_Ω can then be defined as:

$$ect_\Omega^{LB1} = \max_{\Omega' \subseteq \Omega} \left\{ est_{\Omega'} + \mathrm{p}_{\Omega'} + \min_{\pi \in \Pi_{\Omega'}} tt_\pi \right\} \tag{3}$$

Unfortunately, computing this value is NP-hard as computing the optimal permutation $\pi \in \Pi$ minimizing tt_π amounts to solving a TSP. Since embedding an exponential algorithm in a propagator is generally impractical, a looser lower bound can be used instead.

[1] Strictly speaking, the propagators are not sufficient to prove Equation (1) is respected, so the binary propagators for Equation (1) must remain active to ensure correctness.

More precisely, for each possible subset of cardinality $k \in \{0, \ldots, n\}$, we compute the smallest transition time permutation of size k on the set T of all activities requiring the resource:

$$\underline{tt}(k) = \min_{\{\Omega' \subseteq T:\ |\Omega'|=k\}} \left\{ \min_{\pi \in \Pi_{\Omega'}} tt_\pi \right\} \tag{4}$$

For each k, the lower bound computation thus requires one to find the shortest node-distinct $(k-1)$-edge path between any two nodes, which is also NP-hard as it can be casted into a resource-constrained shortest path problem. We proposed in [5] various lower bounds to achieve the pre-computation in polynomial time. Our final lower bound formula for the earliest completion time of a set of activities, making use of pre-computed lower-bounds of transition times, is:

$$ect_\Omega^{LB2} = \max_{\Omega' \subseteq \Omega} \{est_{\Omega'} + \mathrm{p}_{\Omega'} + \underline{tt}(|\Omega'|)\} \tag{5}$$

The different lower bounds of ect_Ω can be ordered as follows:

$$ect_\Omega^{LB0} \leq ect_\Omega^{LB2} \leq ect_\Omega^{LB1} \leq ect_\Omega$$

In order to compute ect_Ω^{LB2} incrementally, adapted versions of the Θ-tree and the Θ-Λ-tree were introduced in [5].

Limitation An important limitation of this approach arises in the context of *sparse* transition matrices. Indeed, when there exists a node-distinct path with K zero-transition edges, we have: $\underline{tt}(k) = 0\ \forall k \in \{1, \ldots, K\}$. The pruning achieved by the propagator is then equivalent to the one of the original algorithms from Vilím [16], which has been shown to perform poorly when transition times are involved (see [5]). To cope with that problem, we propose to reason with *families* of activities, as described in the next section.

Example 2 *Consider again the three activities $\Omega = \{A_1, A_2, A_3\}$ shown in Figure 1 with A_1 belonging to family F_1, A_2 to family F_2, and A_3 to family F_3. The transition times are equal to 3 between activities from different families and equal to 0 between activities of the same family. Assume that 3 additional activities (not represented) also belong to family F_1. Because the transition times between any pair of activity from a same family is 0, we have that $\underline{tt}(2) = \underline{tt}(3) = 0$ and $ect_\Omega^{LB2} = 13 = ect_\Omega^{LB0}$, hence the OC of [5] is unable to detect the failure.*

3 Filtering with Families of Activities

When transition times are present, it is often the case that activities are grouped in families on which the transition times are expressed. Formally, we denote by $F(A_i)$ the family of activity A_i and by $\mathcal{F}$ the set of all families. In a family-based setting, the transition times are described as a square matrix $\mathcal{TT}^{\mathcal{F}}$ of size $|\mathcal{F}|$. The transition time between two activities A_i and A_j is the transition time

between their respective families $F(A_i)$ and $F(A_j)$, and it is zero if $F(A_i) = F(A_j)$:

$$\forall A_i, A_j \in T : tt_{i,j} = tt^{\mathcal{F}}_{F(A_i),F(A_j)} \wedge \Big(F(A_i) = F(A_j) \implies tt^{\mathcal{F}}_{F(A_i),F(A_j)} = 0\Big) \tag{6}$$

The matrix $\mathcal{TT}^{\mathcal{F}}$ is smaller and less sparse than the original matrix $\mathcal{TT}$.

To cope with the limitations highlighted in Section 2.2, we adapt in the present section Vilím's propagators [16] to include transition times between families while keeping a low time complexity: $\mathcal{O}(n \log(n) \log(|\mathcal{F}|))$, where $n = |T|$. To do so, we adapt the algorithms and the Θ-tree and Θ-Λ-tree data structures in a way similar to [5]: the number of different families present in a set Ω of activities is used instead of the cardinality of Ω. Counting the number of families results in non-zero lower bounds even for small sets, assuming that there are no zero transition times between families. Formally, Equation (5) is replaced by:

$$ect^{LB3}_{\Omega} = \max_{\Omega' \subseteq \Omega} \{est_{\Omega'} + \mathrm{p}_{\Omega'} + \underline{tt}(|F_{\Omega'}|)\} \tag{7}$$

where $F_{\Omega} = \{F(A_i) \mid A_i \in \Omega\}$. The term $\underline{tt}(|F_{\Omega'}|)$ in Equation (7) is pre-computed using the lower bounds introduced in [5] for $\underline{tt}(|\Omega'|)$, but using $\mathcal{TT}^{\mathcal{F}}$ instead of $\mathcal{TT}$.

Lemma 1. *In the presence of families,* $ect^{LB2}_{\Omega} \leq ect^{LB3}_{\Omega}$.

Proof. (Sketch) $\mathcal{TT}^{\mathcal{F}}$ induces a graph that is isomorphic to a subgraph of the graph induced by $\mathcal{TT}$ and any (shortest) path induced by $\mathcal{TT}^{\mathcal{F}}$ has a corresponding valid path induced by $\mathcal{TT}$. □

Computing ect^{LB3}_{Ω} requires some careful adaptations to the algorithms (Section 3.1) and data structures (Sections 3.2 and 3.3).

3.1 Adapting the Algorithms

We adapt the original algorithms of [16] in order to consider transition times. While most of the modifications impact the underlying Θ-tree and Θ-Λ-tree data structures, the filtering rules are also slightly adapted. This is done in a similar manner to [5], but reasoning with F_{Ω} instead of Ω.

For instance, in the original algorithms of [16] (OC, DP, NF/NL and EF), if the activity i is detected as having to take place after all activities in a set Θ, the following update rule can be applied: $est_i \leftarrow \max\left\{est_i, ect^{LB0}_{\Theta}\right\}$. As transition times are involved, we can replace ect^{LB0}_{Θ} by ect^{LB3}_{Θ} but, additionally, the minimal transition from any family $F_j \in F_{\Theta}$ to the family $F(A_i)$ should also be added as it was not taken into account in the computation of ect^{LB3}_{Θ}. This transition is the minimal one from any family to $F(A_i)$, because we do not know which activity will be just before A_i in the final schedule. The update rule becomes:

$$est_i \leftarrow \max\left\{est_i, ect^{LB3}_{\Theta} + \min_{F_j \in F_{\Theta}} tt^{\mathcal{F}}_{F_j,F(A_i)}\right\}$$

An analogous reasoning can be applied to the rule updating the *lct* of an activity. Finally, notice that as in [5], the transition times binary decomposition from Equation (1) must be added to the model in order to ensure correctness. Indeed, ect^{LB3}_{Θ} contains only a lower bound of the total transition time in Θ. The propagators based on ect^{LB3}_{Θ} are thus not sufficient to ensure correctness.

3.2 Extending the Θ-tree with Families

A Θ-tree is a balanced binary tree in which each leaf represents an activity from a set Θ and internal nodes gather information about the set of activities represented by the leaves under this node, denoted $Leaves(v)$. We write $l(v)$ for the left child of v and $r(v)$ for the right one. Leaves are ordered in non-decreasing order of the *est* of the activities: for two activities A_i and A_j, if $est_i < est_j$, then the leaf representing A_i is at the left of the leaf representing A_j.

The main value stored in a node v is the lower bound of $ect_{Leaves(v)}$, denoted ect_v. To be able to compute this value incrementally upon insertion or deletion of an activity in the Θ-tree, one needs to maintain additional values.

In [16], Vilím has shown that, by defining $ect_v = ect^{LB0}_{Leaves(v)}$, it suffices to store additionally $p_v = p_{Leaves(v)}$. In a leaf v representing an activity A_i, one can compute $p_v = \mathrm{p}_i$ and $ect_v = ect_i$. In an internal node v, one can compute:

$$\begin{aligned} p_v &= p_{l(v)} + p_{r(v)} \\ ect_v &= \max\left\{ect_{r(v)}, ect_{l(v)} + p_{r(v)}\right\} \end{aligned}$$

Hence, the values only depend on the values stored in the two children.

In this work, we would like instead to define $ect_v = ect^{LB3}_{Leaves(v)}$ in order to take family-based transition times into account. As this value cannot easily be computed incrementally, we compute a lower bound, denoted ect^*_v. In addition to ect^*_v, one needs to store not only p_v, but also $F_v = F_{Leaves(v)}$, the set of the families of the activities in $Leaves(v)$. In a leaf v representing an activity A_i, one can compute $p_v = \mathrm{p}_i$, $ect^*_v = ect_i$, and $F_v = \{F(A_i)\}$. In an internal node v, one can compute:

$$\begin{aligned} p_v &= p_{l(v)} + p_{r(v)} \\ F_v &= F_{l(v)} \cup F_{r(v)} \\ ect^*_v &= \max \begin{cases} ect^*_{r(v)} \\ ect^*_{l(v)} + p_{r(v)} + \underline{tt}\left(\left|F_{r(v)} \setminus F_{l(v)}\right| + eI\left(F_{l(v)}, F_{r(v)}\right)\right) \end{cases} \end{aligned}$$

where $eI\left(F_A, F_B\right)$ is equal to 1 if $(F_A \cap F_B) = \emptyset$, and to 0 otherwise.

Lemma 2. $ect^*_v \leq ect^{LB3}_{Leaves(v)}$

Proof. (Sketch) By induction. If v is a leaf representing activity A_i, then $ect^*_v = ect_i = ect^{LB3}_{\{A_i\}}$. Otherwise, our induction hypothesis is that $ect^*_{l(v)} \leq ect^{LB3}_{Leaves(l(v))}$ and $ect^*_{r(v)} \leq ect^{LB3}_{Leaves(r(v))}$. Let us call $\Omega^{LB3} \subseteq Leaves(v)$ the optimal set to compute $ect^{LB3}_{Leaves(v)}$. Either:

- $ect^*_v = ect^*_{r(v)}$. This rule assumes $\Omega^{LB3} \subseteq Leaves(r(v))$. If this is the case, then we already know by induction that $ect^*_{r(v)} \leq ect^{LB3}_{Leaves(r(v))}$.
- $ect^*_v = ect^*_{l(v)} + p_{r(v)} + \underline{tt}\big(|F_{r(v)} \setminus F_{l(v)}| + eI\left(F_{l(v)}, F_{r(v)}\right)\big)$. This rule assumes $\Omega^{LB3} \cap Leaves(l(v)) \neq \emptyset$. If this is the case, then one only needs to ensure that:

$$ect^{LB3}_{Leaves(l(v))} + p_{r(v)} + \underline{tt}\big(|F_{r(v)} \setminus F_{l(v)}| + eI\left(F_{l(v)}, F_{r(v)}\right)\big) \leq ect^{LB3}_{Leaves(v)}$$

Intuitively, we only add to $ect^{LB3}_{Leaves(l(v))}$ a time quantity that was not considered in $ect^{LB3}_{Leaves(l(v))}$ and that has yet to be spent: durations of activities in $Leaves(r(v))$ and a number of transitions in $F_{r(v)} \setminus F_{l(v)}$ (plus an extra transition when the intersection between $F_{l(v)}$ and $F_{r(v)}$ is empty).

□

Complexity We use bit sets to represent the set of families in each node. The space complexity of the Θ-tree is therefore $\mathcal{O}(n|\mathcal{F}|)$. The set operations we use are *union*, *intersection*, *difference* and *cardinality*. Using bit sets, the 3 former ones are $\mathcal{O}(1)$ and the latter one is $\mathcal{O}(\log(|\mathcal{F}|))$ with a *binary population count* [18]. The time complexity of insertion and deletion of an activity in the Θ-tree is therefore $\mathcal{O}(\log(n)\log(|\mathcal{F}|))$.

Example 3 *Let us consider the activities presented in Figure 2 (left). The transition matrix $\mathcal{TT}^{\mathcal{F}}$ between families is given in Figure 2 (center). The pre-computed values of $\underline{tt}(k)$ are reported in Figure 2 (right). Figure 3 illustrates the extended Θ-tree when all activities are inserted. Note that the value at the root of the tree is indeed a lower bound as the real ect is 85 and $ect^{LB3}_{\Theta} = 80$.*

	A_1	A_3	A_2	A_4
est	0	15	25	30
p	10	10	20	25
F	F_1	F_2	F_3	F_3

$$\mathcal{TT}^{\mathcal{F}} = \begin{pmatrix} 0 & 10 & 15 \\ 5 & 0 & 10 \\ 5 & 15 & 0 \end{pmatrix}$$

$\underline{tt}(k)$	k
0	0
1	0
2	5
3	15

Fig. 2: Example: four activities and their families (left), transition times for the families (center), and pre-computed lower bounds for the transition times (right).

3.3 Extending the Θ-Λ-tree with Families

The Edge-Finding (EF) algorithm requires an extension of the original Θ-tree, called Θ-Λ-tree [16]. In this extension, leaves are marked as either *white* or *gray*. White leaves represent activities in the set Θ and gray leaves represent activities that are in a second set, Λ, with $\Lambda \cap \Theta = \emptyset$. In addition to ect_v, a lower bound

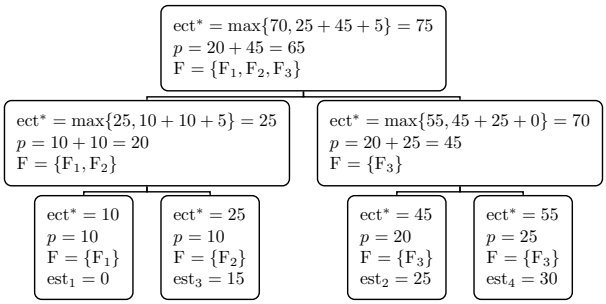

Fig. 3: A Θ-tree when all activities of Figure 2 are inserted.

to the *ect* of Θ, a Θ-Λ-tree also aims at computing $\overline{ect}_v$, which is a lower bound to $\overline{ect}_{(\Theta,\Lambda)}$, the largest *ect* obtained by including *one* activity from Λ into Θ:

$$\overline{ect}_{(\Theta,\Lambda)} = \max_{A_i \in \Lambda} \ ect_{\Theta \cup \{A_i\}}$$

In addition to p_v, ect_v, the original Θ-Λ-tree structure also maintains $\overline{p}_v$ and $\overline{ect}_v$, respectively corresponding to p_v and ect_v, *if* a single gray activity in the sub-tree rooted at v maximizing $ect_{Leaves(v) \cup \{A_i\}}$ *was* included.

Our extension to the Θ-Λ-tree is similar to the one outlined in Section 3.2: in addition to the previous values, each internal node also stores F_v and $\overline{F}_v$ in order to compute the lower bounds ect^*_v and $\overline{ect}^*_v$. This latter value is defined as:

$$\overline{ect}^*_{(\Theta,\Lambda)} = \max \left\{ ect^*_\Theta, \ \max_{A_i \in \Lambda} \left\{ ect^*_{\Theta \cup \{A_i\}} \right\} \right\}$$

Adapting the rules for the Θ-Λ-tree requires caution when families are involved. In [16] and [5], the rules only use implicitly the information about *which* gray activity is considered in the update. In our case, the rules must consider explicitly where the used gray activity is located: either in the left subtree, denoted (L), or in the right subtree, denoted (R). The rules are then defined as:

$$\overline{ect}^*_v = \max \begin{cases} \overline{ect}^*_{l(v)} + p_{r(v)} + \underline{tt}\big(|F_{r(v)} \setminus \overline{F}_{l(v)}| + eI\left(\overline{F}_{l(v)}, F_{r(v)}\right)\big) & \text{(L)} \\ ect^*_{l(v)} + \overline{p}_{r(v)} + \underline{tt}\big(|\overline{F}_{r(v)} \setminus F_{l(v)}| + eI\left(F_{l(v)}, \overline{F}_{r(v)}\right)\big) & \text{(R)} \\ \overline{ect}^*_{r(v)} & \text{(R)} \end{cases}$$

$$\overline{F}_v = \begin{cases} \overline{F}_{l(v)} \cup F_{r(v)} & \text{(L)} \\ F_{l(v)} \cup \overline{F}_{r(v)} & \text{(R)} \end{cases}$$

$$\overline{p}_v = \begin{cases} \overline{p}_{l(v)} + p_{r(v)} & \text{(L)} \\ p_{l(v)} + \overline{p}_{r(v)} & \text{(R)} \end{cases}$$

In the rules above, the choice of which formula to use for $\overline{F}_v$ and $\overline{p}_v$ depends on the letter, either (L) or (R), associated with the term maximizing $\overline{ect}^*_v$, hence this value must be computed first. If a leaf v represents an activity A_i, then we

simply have $\overline{ect}_v^* = ect_i$, $\overline{p}_v = \mathrm{p}_i$, and $\overline{F}_v = \{F(A_i)\}$. The rules for p_v, ect_v, and F_v are as presented in Section 3.2, but one must also define, for a gray leaf v, $ect_v^* = -\infty$, $p_v = 0$, and $F_v = \emptyset$.

For space reasons, we do not present the proof of correctness of our recursive rules. As for the extended Θ-tree introduced in Section 3.2, the time complexity for the insertion and the deletion of an activity is $\mathcal{O}(\log(n)\log(|\mathcal{F}|))$.

4 Related Work

As described in a recent survey [1], scheduling problems with transition times can be classified in different categories. First the activities can be in *batch* (i.e. a machine allows several activities of the same batch to be processed simultaneously) or not. Transition times may exist between successive batches. A CP approach for batch problems with transition times is described in [16]. Secondly the transition times may be *sequence-dependent* or *sequence-independent*. Transition times are said to be sequence-dependent if their durations depend on both activities between which they occur. On the other hand, transition times are sequence-independent if their durations only depend on the activity after which they take place. The problem category we study in this article is non-batch sequence-dependent transition times problems.

Over the years, many CP approaches have been developed to solve such problems [7, 2, 19, 10, 5]. For instance, in [2], a Traveling Salesman Problem with Time Window (TSPTW) relaxation is associated to each resource. The activities used by a resource are represented as vertices in a graph, and edges between vertices are weighted with the corresponding transition times. The TSPTW obtained by adding time windows to vertices from bounds of corresponding activities is then resolved. If one of the TSPTW is found unsatisfiable, then the corresponding node of the search tree is pruned. A similar technique is used in [3] with additional propagators, which are, to the best of our knowledge, the state of the art propagators when families of activities are present.

State-of-the-art filtering with Families

An idea from [17] that is also used in [3] is to pre-compute the exact minimal total transition time for every subset of families. For a subset of families $\mathcal{F}' \subseteq \mathcal{F}$, let $tt(\mathcal{F}')$ denote the minimal total transition time used for any activity set Ω such that $F_\Omega = \mathcal{F}'$. Similarly $tt(F_i \rightarrow \mathcal{F}')$ is the minimal total transition time when the processing starts with some activity of type $F_i \in \mathcal{F}'$, and $tt(\mathcal{F}' \rightarrow F_i)$ when it completes with an activity of type $F_i \in \mathcal{F}'$. We can pre-compute these values for every set of families $\mathcal{F}' \subseteq \mathcal{F}$ and every family $F_i \in \mathcal{F}'$ with a dynamic program running in $\Theta(|\mathcal{F}|^2.2^{|\mathcal{F}|})$ and requiring $\Theta(|\mathcal{F}|.2^{|\mathcal{F}|})$ of memory. For instance, for $tt(F_i \rightarrow \mathcal{F}')$, one defines:

$$\begin{cases} tt(F_i \rightarrow \{F_i\}) = 0 & \forall F_i \in \mathcal{F} \\ tt(F_i \rightarrow \{\mathcal{F}' \cup F_i\}) = \min\limits_{F_j \in \mathcal{F}'} \{tt^{\mathcal{F}}_{F_i,F_j} + tt(F_j \rightarrow \mathcal{F}')\} & \forall \mathcal{F}' \subseteq \mathcal{F}, \forall F_i \in \mathcal{F} \setminus \mathcal{F}' \end{cases}$$

Based on these pre-computed values, which are assumed to be obtainable in $\mathcal{O}(1)$ once the pre-computation is made, two propagators are introduced in [3]:

- A DP-like propagator called UpdateEarliestStart running in $\mathcal{O}(n^2 \log(n))$.
- An EF-like propagator called PrimalEdgeFinding running in $\mathcal{O}(|\mathcal{F}|n^2)$.

Although the filtering obtained with these propagators can be stronger than their counterpart from [16] and our extensions, the time complexity of the propagators is quite high as compared to $O(n \log(n) \log(|\mathcal{F}|))$. In addition, they do not make use of a Not-First/Not-Last rule and the pre-computation of the minimal exact transition times for every subset of family is only tractable for small (typically less than 10) values of $|\mathcal{F}|$.

5 Experimentations

The experiments were conducted on JSPSDTT instances. We used AMD Opteron processors (2.7 GHz), the Java Runtime Environment 8 and the constraint solver *OscaR* [12]. The memory consumption was limited to 4Gb.

Problem instances We have used two sets of instances. First, we used the standard **t2ps** instances from Brucker and Thiele [4]. However, there are only 15 of them, and we wanted to evaluate instances with more families, jobs, and machines in order to challenge the scalability of the different approaches. We therefore generated a new set of 315 instances, here referred to as **uttf**, with up to 50 jobs, 15 machines and 30 families.[2]

Compared Propagators We compare models with the following propagators for Equation (1):

- *binary-decomp*: binary decomposition of Equation (1) only.
- *utt-no-families*: propagators for URTT from [5].
- *artigues-exact-tsp*: propagators of [3] using exact values for $tt(\mathcal{F})$, $tt(F \rightarrow \mathcal{F})$ and $tt(\mathcal{F} \rightarrow F)$.
- *artigues-lb-tsp*: propagators of [3] adapted to make use of cardinality-based lower bounds from [5] for $tt(\mathcal{F})$, $tt(F \rightarrow \mathcal{F})$ and $tt(\mathcal{F} \rightarrow F)$.
- *utt-families-exact-tsp*: propagators introduced in this paper making use of the exact values for $\underline{tt}(|\mathcal{F}|)$ computed with $\min_{\mathcal{F}':|\mathcal{F}'|=|\mathcal{F}|} tt(\mathcal{F}')$.
- *utt-families-lb-tsp*: propagators introduced in this paper making use of lower bounds for $\underline{tt}(|\mathcal{F}|)$. The bounds are computed with the lower bounds of [5].

All approaches also use the binary decomposition of Equation (1) in order to ensure correctness as specialized propagators are generally not checking.

[2] The instances are available at http://becool.info.ucl.ac.be/resources/uttf-instances.

Replay Evaluation In order to derive fair and representative conclusions about the propagators only (i.e., by removing the effects of the search heuristic), we used the *Replay* evaluation methodology [14]. First, for each instance, a *baseline* model is used to generate a search tree. This baseline model is, among the different compared approaches, the one that prunes the less the domains (here *binary-decomp*). Once the search tree is generated, it is *replayed* separately with each model. A replay basically consists in reapplying the exact same sequence of modifications to the constraint store (e.g., the branching constraints) that were used to generate the search tree with the baseline model.

The performance of those replays is then used to construct so-called *performance profiles* [6], that we built with a public web tool [15] made available to the community.[3] Performance profiles are cumulative distribution functions of a performance metric ratio τ. In our case, τ is a ratio of either time or number of backtracks. In the case of time, the function is defined as:

$$F_m(\tau) = \frac{1}{|\mathcal{I}|} \left| \left\{ i \in \mathcal{I} : \frac{time_{replay}(m,i)}{\min_{m' \in M} time_{replay}(m',i)} \leq \tau \right\} \right| \tag{8}$$

where $\mathcal{I}$ is the set of considered instances, m is a model and M is the set of all models. The function is similar for the number of backtracks.

To generate the search tree, the Conflict Ordering Search [8] was used, as it was shown to be a good search strategy for scheduling problems. The generation lasted for 300 seconds, and we enforced a timeout of 1,800 seconds for the replay. If a timeout occurs for a model m, we consider that $\frac{time_{replay}(m,i)}{\min_{m' \in M} time_{replay}(m',i)} = +\infty$. The running times reported here do not take into account the pre-computation step since they are negligible (generally less than 2 sec. and max 10 sec.).

Results on the t2ps Instances Figures 4 and 5 provide the performance profiles for the time and number of backtracks, respectively. Figure 5 shows that, interestingly, *utt-families-lb-tsp* prunes exactly as much as *utt-families-exact-tsp*. This is due to the fact that our lower bounds are here able to compute the same values than $\min_{\mathcal{F}':|\mathcal{F}'|=|\mathcal{F}|} tt(\mathcal{F}')$. This suggests that we often do not have to compute the exact values for $tt(\mathcal{F})$ with the resource-consuming dynamic program, which is interesting since it is not tractable when there are many families. We can see that from a time perspective, our approach is the fastest for 80% of the instances (*utt-families-exact-tsp* being here equivalent to *utt-families-lb-tsp*, see the function in $\tau = 1$ in Figure 4). But our approach is also robust, as the other instances (i.e., the remaining 20%) are solved within a factor $\tau < 2$ compared to the best model for those remaining instances. Considering the number of backtracks, our approach generally achieves less pruning than *artigues-exact-tsp* (not more than three times), but substantially more than *utt-no-families*. This lack of pruning as compared to *artigues-exact-tsp* is compensated in practice by the low time complexity. Although not reported, we tried to combine

[3] Accessible at http://sites.uclouvain.be/performance-profile/.

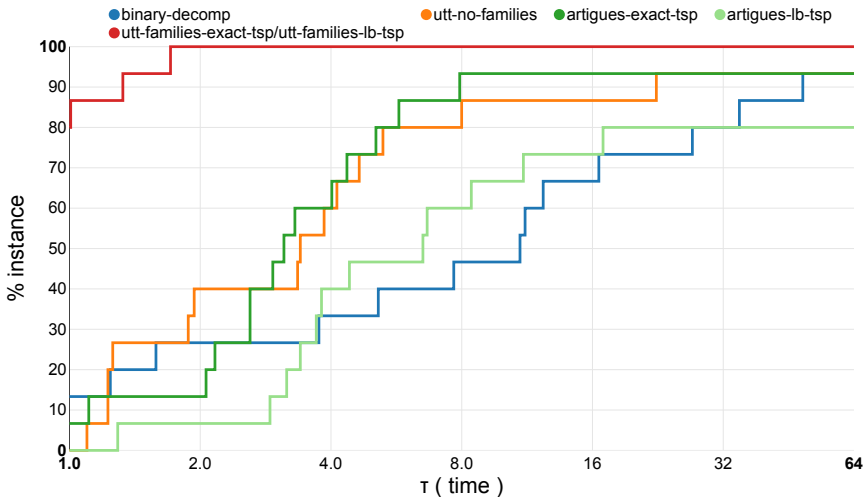

Fig. 4: Performance profiles on **t2ps** instances for the time metric.

utt-families-exact-tsp and *artigues-exact-tsp* and the performances were close to the ones of *artigues-exact-tsp* alone, thus only inducing a small overhead when *utt-families-exact-tsp* does not provide additional pruning.

Results on the uttf Instances First of all, we consider the approaches *artigues-exact-tsp* and *utt-families-exact-tsp* unable to solve (i.e., times out by default) the 120 instances (out of 315) with 20 families or more, since the pre-computation becomes too expensive in terms of CPU and memory usage according to our 4Gb limitation.

Figures 6 and 7 provide the time performance profiles for the instances with strictly less than and with more than 20 families, respectively. Figure 6 shows that our approach still outperforms the other ones, even if it is the fastest on a smaller percentage of instances than for the **t2ps** instances. The instances being less structured, the gain in pruning is weaker as compared to the decomposition. However, our method catches up very quickly; for example, it is at most ~ 1.3 and 2 times slower than the best approach for almost 60% and 80% of the instances, respectively. Another interesting point is that *utt-families-exact-tsp* and *utt-families-lb-tsp* have very similar time performances, while the values for $\underline{tt}(k)$ were here generally different (not reported here). This means that computing the exact values for $tt\,(\mathcal{F})$ is not mandatory[4] when used with our propagators, which is profitable since we also target scalability in terms of number of families.

Regarding the instances with more than 20 families (Figure 7), our approach is significantly better than the other ones, as we are the fastest on almost 70% of the instances and it is at most 4 times slower than the best approach on the

[4] Still, if it is available at a low cost, it can be beneficial to use it.

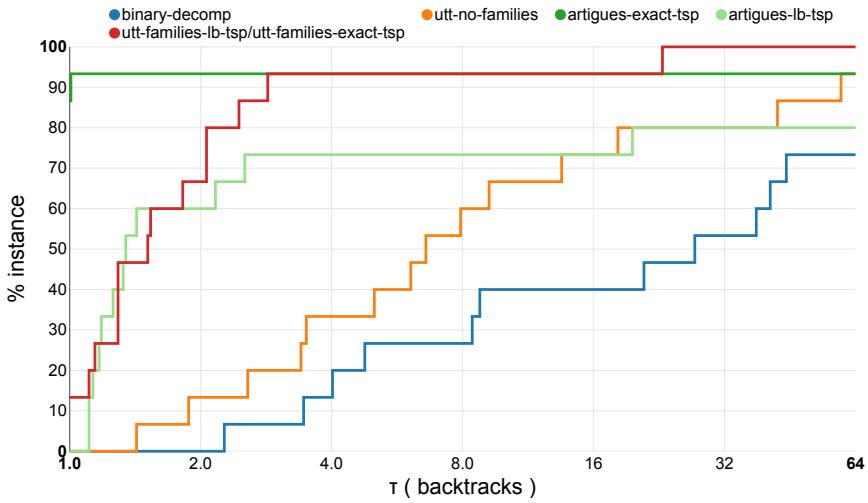

Fig. 5: Performance profiles on **t2ps** instances for the backtracks metric.

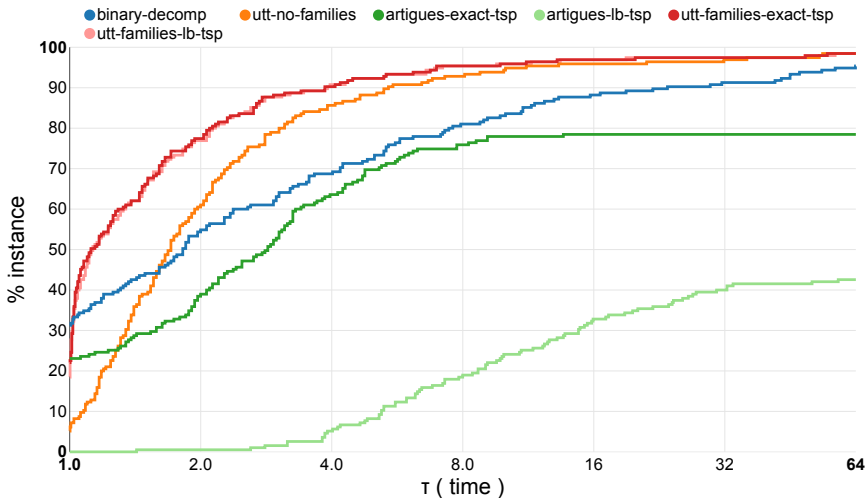

Fig. 6: Performance profiles on **uttf** instances with strictly less than 20 families for the time metric.

remaining instances. This teaches us that when more families are involved, our approach is both efficient and robust.

*Improvements on Open **t2ps** Instances* Although not the focus of this paper, we were able to find tighter upper bounds for 3 of the 6 open **t2ps** instances within less than 5 minutes of computation. We simply combined our lightweight propagators with a LNS [13]. We used a basic relaxation of precedences of activ-

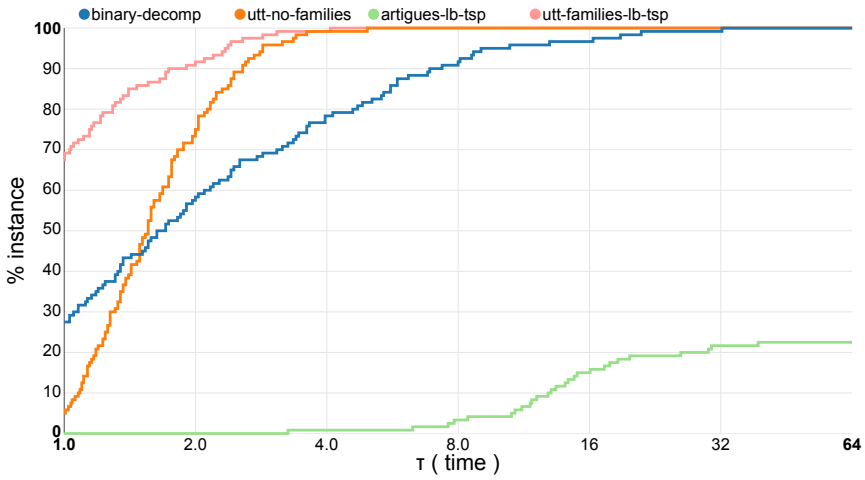

Fig. 7: Performance profiles on **uttf** instances with more than 20 families for the time metric.

Upper Bound	t2-ps11	t2-ps12	t2-ps15
Former	1,470	1,305	1,527
New	1,441	1,299	1,505

Table 1: New upper bounds for open **t2ps** instances.

ities on the same machine combined with a *Set Times* [11] search strategy. The improved bounds are given in Table 1.

6 Conclusion

This paper has extended the algorithms and data structures for the unary resource, taking into account family-based transition times in order to perform additional propagation. The original data structures and algorithms have been adapted accordingly. The approach is therefore lightweight from both the time and space perspectives. Experiments conducted on the JSPSDTT have demonstrated that the introduced approach provides a substantial gain and is quite robust to changes in instance characteristics (e.g., number of activities and families).

Future work We would like to consider other types of problems and combine this work with the use of good lower bounds in a branch-and-bound setting. More importantly, when there are no families defined *a priori* in an instance, we want to study the benefit of first creating them by means of *clustering* algorithms and then using the filtering introduced in this paper. This approach might prove to be helpful when the intra-cluster transition times are significantly smaller than the inter-cluster ones.

References

1. Allahverdi, A., Ng, C., Cheng, T.E., Kovalyov, M.Y.: A survey of scheduling problems with setup times or costs. European Journal of Operational Research 187(3), 985–1032 (2008)
2. Artigues, C., Belmokhtar, S., Feillet, D.: A new exact solution algorithm for the job shop problem with sequence-dependent setup times. In: Integration of AI and OR techniques in constraint programming for combinatorial optimization problems, pp. 37–49. Springer (2004)
3. Artigues, C., Feillet, D.: A branch and bound method for the job-shop problem with sequence-dependent setup times. Annals of Operations Research 159(1), 135–159 (2008)
4. Brucker, P., Thiele, O.: A branch & bound method for the general-shop problem with sequence dependent setup-times. Operations-Research-Spektrum 18(3), 145–161 (1996)
5. Dejemeppe, C., Van Cauwelaert, S., Schaus, P.: The unary resource with transition times. In: Principles and Practice of Constraint Programming. pp. 89–104. Springer (2015)
6. Dolan, E.D., Moré, J.J.: Benchmarking optimization software with performance profiles. Mathematical programming 91(2), 201–213 (2002)
7. Focacci, F., Laborie, P., Nuijten, W.: Solving scheduling problems with setup times and alternative resources. In: AIPS. pp. 92–101 (2000)
8. Gay, S., Hartert, R., Lecoutre, C., Schaus, P.: Conflict ordering search for scheduling problems. In: Pesant, G. (ed.) Principles and Practice of Constraint Programming, CP 2015. pp. 140–148. Springer (2015)
9. Gay, S., Schaus, P., De Smedt, V.: Continuous casting scheduling with constraint programming. In: Principles and Practice of Constraint Programming. pp. 831–845. Springer (2014)
10. Grimes, D., Hebrard, E.: Job shop scheduling with setup times and maximal time-lags: A simple constraint programming approach. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, pp. 147–161. Springer (2010)
11. Le Pape, C., Couronné, P., Vergamini, D., Gosselin, V.: Time-versus-capacity compromises in project scheduling (1994)
12. OscaR Team: OscaR: Scala in OR (2012), available from `https://bitbucket.org/oscarlib/oscar`
13. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Principles and Practice of Constraint Programming, pp. 417–431. Springer (1998)
14. Van Cauwelaert, S., Lombardi, M., Schaus, P.: Understanding the potential of propagators. In: Integration of AI and OR Techniques in Constraint Programming, pp. 427–436. Springer (2015)
15. Van Cauwelaert, S., Lombardi, M., Schaus, P.: A visual web tool to perform what-if analysis of optimization approaches. Tech. rep., UCLouvain (2016)
16. Vilım, P.: Global constraints in scheduling. Ph.D. thesis, PhD thesis, Charles University in Prague, Faculty of Mathematics and Physics, Department of Theoretical Computer Science and Mathematical Logic, KTIML MFF, Universita Karlova, Malostranské námestı 2/25, 118 00 Praha 1, Czech Republic (2007)
17. Vilım, P., Barták, R.: Filtering algorithms for batch processing with sequence dependent setup times. In: Proceedings of the 6th International Conference on AI Planning and Scheduling, AIPS (2012)

18. Warren, H.S.: Hacker's delight. Pearson Education (2013)
19. Wolf, A.: Constraint-based task scheduling with sequence dependent setup times, time windows and breaks. GI Jahrestagung 154, 3205–3219 (2009)
20. Zampelli, S., Vergados, Y., Van Schaeren, R., Dullaert, W., Raa, B.: The berth allocation and quay crane assignment problem using a cp approach. In: Principles and Practice of Constraint Programming. pp. 880–896. Springer (2013)