Noname manuscript No.
(will be inserted by the editor)

The Unary Resource with Transition Times and Optional Activities

Sascha Van Cauwelaert · Cyrille Dejemeppe · Pierre Schaus

Received: date / Accepted: date

Abstract This paper describes a unified global constraint to model scheduling problems with unary resources, i.e., that can only process a single activity at a time. In addition, the constraint enforces sequence-dependent transition times between the activities. It often happens that activities are grouped into families with zero transition times within a family. Moreover, some of the activities might be optional from the resource viewpoint (typically in the case of alternative resources). The global constraint unifies reasoning with both optional activities and families of activities. The scalable filtering algorithms we discuss keep a low time complexity of $\mathcal{O}(n \cdot \log(n) \cdot \log(f))$, where n is the number of tasks on the resource and f is the number of families. This results from the fact that we extend the Θ-tree data structure used for the Unary Resource constraint without transition times. Our experiments demonstrate that our global constraint strengthen the pruning of domains as compared with existing approaches, leading to important speedups. Moreover, our low time complexity allows maintaining a small overhead, even for large instances. These conclusions are particularly true when optional activities are present in the problem.

Keywords Constraint Programming · Scheduling · Global Constraint · Unary Resource · Transition Times · Optional Activities · Scalability

Sascha Van Cauwelaert
Place Sainte Barbe 2 bte L5.02.01 1348 Louvain-la-Neuve
E-mail: sascha.vancauwelaert@uclouvain.be

Cyrille Dejemeppe
n-Side, Avenue Baudouin 1er 25, 1348 Louvain-la-Neuve
E-mail: cde@n-side.com

Pierre Schaus
Place Sainte Barbe 2 bte L5.02.01 1348 Louvain-la-Neuve
E-mail: pierre.schaus@uclouvain.be

1 Introduction

Over the last decades, constraint programming has been successfully applied to solve scheduling problems [4,5], while substantial improvements are still ongoing [21,23]. One reason for this success is the incorporation of operation research techniques into global constraints. In addition to ease the modeling, they improve the solving time by capturing and efficiently solving subproblems of the main problem. This paper describes a unified global constraint to model unary resources with transition times and optional activities.

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 [36], 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 [18], where a set-up time is required between production programs. Figure 1 illustrates a minimalistic example of two activities running on a unary resource with transition times.

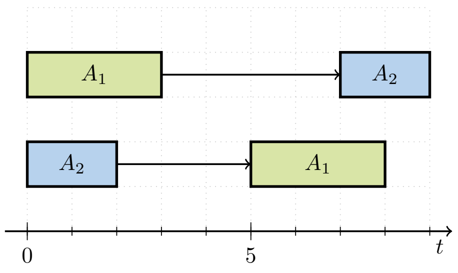


Fig. 1 Two possible schedule for two activities A_1 and A_2 running on the same unary resource with transition times. They can never overlap in time so either activity A_2 starts after activity A_1 has completed, or A_1 starts after activity A_2 has completed. Moreover a minimum transition time (represented by the arrows) must occur between the end of an activity and its successor. Notice the value of the transition depends on the processing order of the activities.

Although efficient propagators have been designed for the standard unary resource constraint (UR) [30], transition time constraints between activities generally make the problem harder to solve because the existing propagators do not take them into account. A propagator for the unary resource constraint with transition times (URTT) was recently introduced [14] 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. In addition, while Vilím algorithms allow to reason with *optional* activities, the approach from [14] does not support them.

The main contribution of the present paper is to introduce a generalized unary resource with transition times that unifies filtering rules and algorithms such that they consider family-based transition times and optional activities. The main asset of our approach is its scalability: we obtain a strong filtering while keeping a low time complexity of $\mathcal{O}(n.\log(n).\log(f))$, 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 [30] and [14]. 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. We first consider the case where it is known prior to search on which machine the activities must be executed, and then the more general case where activities must be executed by exactly one of a set of alternative machines. The results show that our propagator improves the resolution time over existing approaches and is more scalable.

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 grouped in *batches* (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 Constraint Programming (CP) approach for batch problems with transition times is described in [30]. 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 paper is non-batch sequence-dependent transition times problems.

Over the years, many CP approaches have been developed to solve such problems [16,2,35,19,14]. 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. Grimes and Hebrard proposed an efficient solution to job shop with transition times problems by using a simple lightweight CP model combined with restarts and weighted degree search heuristics [19]. Recently, a bounded dynamic programming approach [26] has improved the state of the art of standard benchmarks of the job shop with transition times problem.

Paper Outline Section 2 provides the background required to read the paper. The content is then described in a top-down fashion: Section 3 describes the filtering rules for the unary resource with transition times and the different algorithms to apply those rules. Then, we explain in Section 4 the data structures required by the filtering algorithms. They rely on lower bounds for the minimum total transition time that must hold in a given set of activities. We discuss those lower bounds in Section 5. Finally, Section 6 compares the results of the different existing approaches for the unary resource with transition times on various applications.

2 Background

Non-preemptive scheduling problems are usually modeled in CP by associating three variables to each activity i: s_i, c_i, and p_i[1] representing respectively the starting time, completion time, and processing time of i. These variables are linked together by the following relation: $s_i + p_i = c_i$. 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 paper, we are interested in the unary resource (sometimes referred to as *machine* or *disjunctive resource*) and the propagators associated to a single unary resource. Let T be the set of activities requiring the unary resource. The unary resource constraint prevents any two activities in T to overlap in time:

$$\forall i, j \in T : i \neq j \implies (c_i \leq s_j) \vee (c_j \leq s_i)$$

Transition Times The unary resource can be generalized by requiring transition times between activities. They are described by a square *transition matrix* *tt* in which $tt_{i,j}$, the entry at line i and column j, represents the minimum amount of time between the activities i and j when i directly precedes j. We assume that transition times respect the triangular inequality. That is, inserting any activity between two activities never decreases the transition time between these two activities: $\forall i, j, k \in T : tt_{i,j} \leq tt_{i,k} + tt_{k,j}$.

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

$$\forall i, j \in T : i \neq j \implies (c_i + tt_{i,j} \leq s_j) \vee (c_j + tt_{j,i} \leq s_i) \qquad \text{(URTT)}$$

An example of a transition matrix is given in Figure 2, where we can notice that it is not symmetric (e.g., $tt_{1,2} = a \neq c = tt_{2,1}$ in Figure 2). As exemplified, it induces a *transition graph*, that will be used in the forthcoming sections.

[1] In this paper we assume without loss of generatlity that p_i is constant.

$$tt = \begin{pmatrix} 0 & a & b \\ c & 0 & d \\ e & f & 0 \end{pmatrix}$$

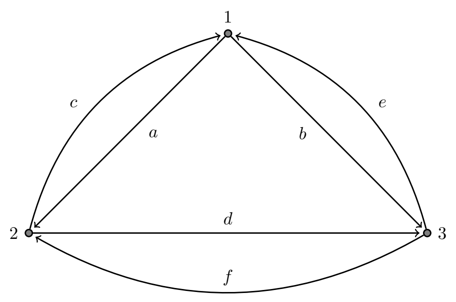


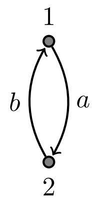
Fig. 2 Example of a transition matrix tt and its induced Transition Graph.

Family-Based Transition Times When transition times are present, it is often the case that activities are grouped into families on which the transition times are expressed. Formally, we denote by F_i the family of activity i and by $\mathcal{F}$ the set of all families. Moreover, for a given set of activities Ω, we write $F_\Omega = \{F_i \mid i \in \Omega\}$. In a family-based setting, the transition times are described as a square *family transition matrix* $tt^{\mathcal{F}}$ of size $|\mathcal{F}|$. The transition time between two activities i and j is the transition time between their respective families F_i and F_j, and it is zero if $F_i = F_j$ [2]:

$$\forall i, j \in T : tt_{i,j} = tt^{\mathcal{F}}_{F_i,F_j} \wedge \Big(F_i = F_j \implies tt^{\mathcal{F}}_{F_i,F_j} = 0\Big) \tag{1}$$

Given a set of activities, their families and a transition matrix between families, $tt^{\mathcal{F}}$ one can expand $tt^{\mathcal{F}}$ into a transition matrix between activities tt. tt is then larger and sparser than $tt^{\mathcal{F}}$. An example of this expansion is given in Figure 3, where the *family transition graph* induced by $tt^{\mathcal{F}}$ is also illustrated. Notice that $tt = tt^{\mathcal{F}}$ is the special case occurring when each activity is in its own family.

$$tt^{\mathcal{F}} = \begin{pmatrix} 0 & a \\ b & 0 \end{pmatrix}$$



$$tt = \begin{pmatrix} 0 & 0 & a & a & a \\ 0 & 0 & a & a & a \\ b & b & 0 & 0 & 0 \\ b & b & 0 & 0 & 0 \\ b & b & 0 & 0 & 0 \end{pmatrix}$$

Fig. 3 Example of a family transition matrix $tt^{\mathcal{F}}$, its induced Family Transition Graph, and the expanded transition matrix tt for 5 activities with $F_1 = F_2 = 1$ and $F_3 = F_4 = F_5 = 2$.

[2] A more general case is when a positive transition $tt^{\mathcal{F}}_{f,f}$ must occur between activities of the *same* family f. In this case, one can fall back to zero transition times within a family, assuming $tt^{\mathcal{F}}_{f,f} \leq tt^{\mathcal{F}}_{f',f} \forall f' \in tt^{\mathcal{F}}$. One can artificially: (1) increase the duration of activities of the family f with $tt^{\mathcal{F}}_{f,f}$; and (2) decrease the transition times from family f by $tt^{\mathcal{F}}_{f,f}$. Yet, one cannot always perform this trick, so we keep the hypothesis in the rest of the paper that $tt^{\mathcal{F}}_{f,f} = 0$.

Optional Activities Some activities can optionally be used by the resource, i.e., it is unknown a priori if a given optional activity must be processed by the resource in the final schedule. This case typically occurs when an activity must run on one of several alternative resources [16], or when so-called conditional time-intervals [22] are available in the solver. Following Vilím's notation, we call R the set of regular activities (known to be running on the resource) and O the set of optional activities, with $R \cup O = T$ and $R \cap O = \emptyset$.

To model optional activities, an additional boolean variable v_i is used to represent the fact that the activity i is used by the machine. We define $R = \{i \in T : v_i\}$. The unary resource with transition times constraint involving optional activities imposes the following relation:

$$\forall i, j \in T : i \neq j \wedge v_i \wedge v_j \implies (c_i + tt_{i,j} \leq s_j) \vee (c_j + tt_{j,i} \leq s_i) \quad \text{(URTTO)}$$

Precedence Graph The precedence graph $G = \langle T, E \rangle$ is a data structure [9, 16] used to maintain the precedences between activities of a given resource. In this graph, each vertex represents a given activity, and there is a directed edge from a vertex i to vertex j if and only if the activity i precedes the activity j, i.e., $c_i + tt_{i,j} \leq s_j$. In [7], the authors describe propagation rules for the precedence graph while taking optional activities into account.

One can use the precedence graph to make search decisions by adding edges in order to impose precedences between activities. A recent CP approach [19] demonstrated experimentally that branching on the precedences can be effective[3], using smart search techniques based on a domain/weighted-degree heuristic, rather than sophisticated propagators.

Finally, from a filtering perspective, additional precedences can be detected by computing the transitive closure of the graph.

Bounds of a set of activities Ω The earliest starting time of an activity 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 i is denoted ect_i and its latest completion time is denoted lct_i. The domain of c_i is hence the interval $[ect_i..lct_i]$. These definitions can be extended to a set of activities 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 $p_\Omega = \sum_{j \in \Omega} p_j$ to be the sum of the processing times of the activities in Ω. While one can directly compute $est_\Omega = \min\{est_j : j \in \Omega\}$ and $lct_\Omega = \max\{lct_j | j \in \Omega\}$, it is NP-hard [30] to compute the exact values of ect_Ω and lst_Ω. Instead, one usually computes a lower bound for ect_Ω and an upper bound for lst_Ω, as we will see in this paper.

[3] The approach of [19] does not actually use a precedence graph structure explicitly, but reify the precedence constraints and branch on the associated boolean variables.

3 Global Filtering Rules and Propagation Algorithms

This section first describes the inference rules of the unary resource without transition times. Those rules are then extended in order to handle transition times. We also describe the different algorithms in order to compute them efficiently. The data structures required by the algorithms are described in Section 4.

3.1 Filtering Rules for the Unary Resource

The filtering rules presented in [30] 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). They are valid for the general definition of ect_Ω of the earliest completion time of a set of activities $\Omega \subseteq T$. However, since the computation of its exact value is NP-hard, their implementation relies on an efficient computation of a lower bound ect_Ω^{LB0}, defined as:

$$ect_\Omega^{LB0} = \max_{\Omega' \subseteq \Omega} \{est_{\Omega'} + p_{\Omega'}\} \tag{2}$$

To define the different rules, we use the notation ect_Ω although ect_Ω^{LB0} is used in practice, as we will use a stronger lower bound under the presence of transition times later in this paper. Each rule has a symmetric counterpart for lst_Ω that can easily be retrieved from the given definitions[4].

Overload Checking This rule tries to detect an inconsistency given the current domains. Intuitively, for a set of regular activities $\Omega \subseteq R$, if the earliest completion time is found to be larger than the latest completion time, an infeasibility is detected. Additionally, if Ω is extended with an optional activity i such that there would be an inconsistency, we know that the activity cannot be executed by the machine. Formally, we have:

$$\forall \Omega \subseteq R, \forall i \in (T \setminus \Omega) : ect_{\Omega \cup \{i\}} > lct_{\Omega \cup \{i\}} \implies \neg v_i \tag{OC}$$

Notice that if $i \in R$ and $v_i = \textbf{false}$, the constraint is infeasible.

Detectable Precedences This rule detects new precedences between pairs of activities. The reasoning uses the set of activities $DPrec(R, i)$ that can be detected as preceding a given activity i based on the current domains. It is defined as:

$$DPrec(R, i) = \{j \neq i \in R : ect_i > lst_j\} \tag{DPrec}$$

[4] Practical implementation apply the symmetric rules by applying the original rules on *mirrors* of the original activities. The mirror activity m of an activity i is modeled with the variables $s_m = -c_i$, $c_m = -s_i$ and $p_m = p_i$.

The inference rule states that the earliest start time of an activity i must at least be the earliest completion time of the set of activities that are detected as preceding i, that is $\mathit{DPrec}(R,i)$. Formally:

$$\forall i \in T : v_i \implies est_i \leftarrow \max(est_i, ect_{\mathit{DPrec}(R,i)}) \tag{DP}$$

Notice that only the activities known to be running on the resource can be used to update other activities, hence the use of $\mathit{DPrec}(R,i)$ and not $\mathit{DPrec}(T,i)$. On the contrary, all activities (including optionals) can be updated.

Precedences that do not belong to $\mathit{DPrec}(R,i)$ but that must be respected are called *non-detectable precedences* [30]. They originate from the problem itself or branching decisions. Non-detectable precedences are not enforced by the rule DP but with binary propagators or a propagator based on a precedence graph.

Not-Last When a given activity i has a latest starting time that is strictly smaller than the earliest completion time of a set of regular activities Ω, this activity cannot be scheduled as the last one of the set $\Omega \cup \{i\}$. Its latest completion time can therefore be reduced to the maximum latest start time of the activities in Ω:

$$\forall \Omega \subseteq R, \forall i \in (T \setminus \Omega) : v_i \wedge ect_\Omega > lst_i \implies lct_i \leftarrow \min(lct_i, \max_{j \in \Omega} lst_j) \tag{NL}$$

Edge Finding The rule detects new edges in the precedence graph: if adding an activity i to a set of activities Ω leads to an earliest completion larger than the latest completion of the set, then the activity i must succeed the activities in Ω:

$$\forall \Omega \subseteq R, \forall i \in (T \setminus \Omega) : v_i \wedge ect_{\Omega \cup \{i\}} > lct_\Omega \implies est_i \leftarrow \max(est_i, ect_\Omega)) \tag{EF}$$

Update of Domains of Optional Activities Except in the case of the Overload Checking rule, the domain of an optional activity is updated only once it is known to be running on the resource (i.e., $v_i = \mathbf{true}$). However, the inference about the domain of this activity *if it was running on the resource* can be useful to other inference rules. Therefore, the domain is not updated until $v_i = \mathbf{true}$, but the inference on the domain *if the activity runs on the resource* is saved internally and used by all inference rules. Example 1 illustrates an example where this is beneficial.

Example 1 *Let us consider 4 activities, as represented in Figure 4. Green activities are regular activities, while A_3 is optional. If the DP rule was applied to the set $\{A_1, A_2, A_3\}$ and A_3 was a regular activity, est_3 would be updated to 9 (see the red bracket). A_3 is optional, so we only save this update internally. If the OC rule is applied to the set $\{A_3, A_4\}$ with $est_3 = 9$ instead of $est_3 = 6$, one can deduce that $v_3 =$* **false**.

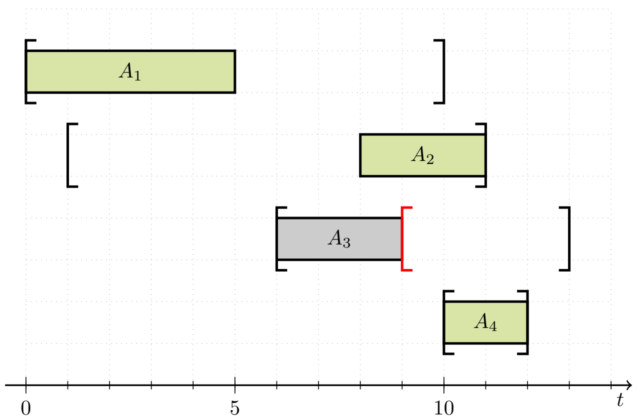


Fig. 4 The inference that can be made on optional activities must be communicated to other inference rules. The activity A_3 is optional and the others are regular. The DP rule applied to the set $\{A_1, A_2, A_3\}$ leads to $est_3 = 9$. Applying the OC rule the set $\{A_3, A_4\}$ with that information allows inferring $v_3 = \textbf{false}$.

Filtering Limitation due to Transition Times Under the presence of transition times, the rules can be improved, as illustrated in Example 2. In the next section, we strengthen the lower bound of ect_Ω so that it takes the transition times into account.

Example 2 *Consider a set of 3 regular activities $\Omega = \{1, 2, 3\}$ as shown in Figure 5. Consider also, for simplicity, that all pairs of activities from Ω have the same transition time $tt_{i,j} = 3\,\forall i, j \in \{1, 2, 3\}$. The OC rule detects a failure when $ect_\Omega^{LB0} > lct_\Omega$. The lower bound is:*

$$ect_\Omega^{LB0} = est_\Omega + \sum_{i \in \Omega} p_i = 0 + 5 + 5 + 3 = 13$$

As we have $lct_\Omega = \max_{i \in \Omega}\ lct_i = lct_2 = 17$, the OC rule from [30], combined with the transition times binary decomposition (Equation (URTTO)), 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.

3.2 Extending the Filtering Rules with Transition Times

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:

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

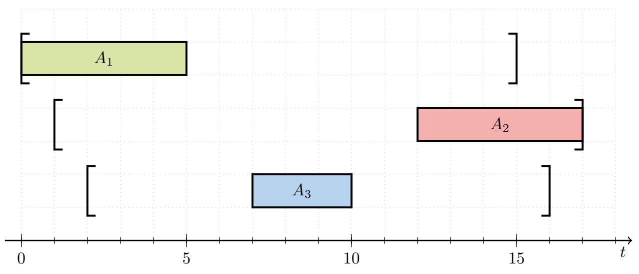


Fig. 5 Example illustrating the missed failure detection of OC when not considering transition times.

A lower bound for ect_Ω that considers transition times can then be defined as:

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

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

For each possible subset of cardinality $k \in [0..|T|]$, we compute the smallest transition time permutation 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 of the transition graph (see Section 2), which is also NP-hard as the Traveling Salesman Problem can be reduced to this problem when $k = |T|$. Since one has to solve $|T|$ NP-hard problems in pre-computation (one for each cardinality k), we proposed in [14] various lower bounds to achieve the computation in polynomial time. They are described in Section 5. Notice that we have $\underline{tt}(0) = \underline{tt}(1) = 0$.

Our final lower bound formula for the earliest completion time of a set of activities, making use of pre-computed lower-bounds on transition times, is:

$$ect_\Omega^{LB2} = \max_{\Omega' \subseteq \Omega} \left\{ est_{\Omega'} + p_{\Omega'} + \underline{tt}(|\Omega'|) \right\} \tag{5}$$

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

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

Limitation An important limitation of this approach arises in the context of *sparse* transition matrices, which typically occurs when activities are grouped in families (see Section 2). Indeed, when there exists a node-distinct path with K zero-transition edges, we have: $\underline{tt}(k) = 0\ \forall k \in [0..K+1]$. The pruning achieved by the propagator is then equivalent to the one of the original algorithms from Vilím [30], which has been shown to perform poorly when transition times are involved (see [14]). This is illustrated in the next example.

Example 3 *Consider again the three activities $\Omega = \{1, 2, 3\}$ shown in Figure 5 with activity 1 belonging to family F_1, activity 2 to family F_2, and activity 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. Since 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 [14] is unable to detect the failure.*

To cope with this limitation, we will use a stronger lower bound by counting the number of different families present in a set Ω of activities instead of the cardinality of Ω. This amounts to find the shortest node-distinct $(k-1)$-edge path in the family transition graph (see Section 2) instead of the transition graph. 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:

$$ect_{\Omega}^{LB3} = \max_{\Omega' \subseteq \Omega} \{est_{\Omega'} + p_{\Omega'} + \underline{tt}(|F_{\Omega'}|)\} \tag{6}$$

where $F_{\Omega} = \{F_i \mid i \in \Omega\}$. The term $\underline{tt}(|F_{\Omega'}|)$ in Equation (6) is pre-computed using the same lower bounds as before, but using $tt^{\mathcal{F}}$ instead of tt. Notice that if $tt = tt^{\mathcal{F}}$, we have $ect_{\Omega}^{LB2} = ect_{\Omega}^{LB3}$.

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

Proof The family transition graph induced by $tt^{\mathcal{F}}$ is isomorphic to a subgraph of the transition graph induced by tt and any (shortest) path induced by $tt^{\mathcal{F}}$ has a corresponding valid path induced by tt. Moreover, a shortest path of exactly k edges induced by tt has a length that is at most equal to a shortest path of exactly k edges induced by $tt^{\mathcal{F}}$.

□

3.3 Adapting the Algorithms

We adapt the original algorithms of [30] in order to consider transition times. Most of the modifications actually impact the underlying Θ-tree and Θ-Λ-tree data structures (described in Section 4), hence the algorithms are similar to the original ones. In our opinion, this is a strength of our approach. The algorithms described in this section apply the rules given in Section 3.1. As mentioned in Section 3.1, counterparts of those rules can be applied using the same algorithms on *mirror* activities. Importantly, one must also transpose the transition matrix.

Notation We denote by ect_{Θ}^{*} a lower bound of ect_{Θ}^{LB3} that will be used by the different algorithms. We describe in Section 4.1 the Θ-tree data structure that is used to compute this value. Moreover, following Vilím's notation, we will use a specific set of *gray activities* $\Lambda \subseteq T$ such that $\Lambda \cap \Theta = \emptyset$. For a given set Θ, this set is used to evaluate how ect_Θ would evolve *if* one of the gray activities of Λ were to be added to the set Θ. Formally, we are interested in computing

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

If $\exists i \in \Lambda : ect_{\Theta \cup \{i\}} > ect_\Theta$, we say the gray activity i is *responsible* for the value $ect_{\Theta \cup \{i\}}$. Responsible activities are used in the Overload Checking and the Edge-Finding algorithms, described in this section. We discuss how to find the responsible activity in Section 4.2. Once more, we actually compute a lower bound of $\overline{ect}_{(\Theta,\Lambda)}$, written $\overline{ect}^*_{(\Theta,\Lambda)}$. Section 4.2 describes the Θ-Λ-tree data structure, used to compute this value efficiently.

Overload Checking The checker (see Algorithm 3.1) goes over each activity in non-decreasing order of lct_i. For each activity, if it is not yet known if it will be executed by the resource (verified by checking the size of the domain of the variable v_i in line 3), it is added to the set Λ (line 4) and the next activity is considered. If the activity has to run on the resource, it is added to the set Θ. The OC rule is then applied: if the earliest completion time of the current set Θ is larger than the latest completion time of the activity i we just added to Θ, the activity i cannot be executed on the machine. Since i is not optional, a feasible schedule cannot be found (see lines 7-9). The current optional activities in Λ are then possibly updated in lines 10-14: as long as it is possible to find an optional activity o such that adding it to Θ would lead to an overload, it is inferred that o cannot be executed by the machine, and o is removed from Λ.

Algorithm 3.1: Overload Checker

```
1  $(\Theta, \Lambda) \leftarrow (\emptyset, \emptyset)$
2  for $i \in T$ in non-decreasing order of $lct_i$ do
3      if $|D(v_i)| > 1$ then
4          $\Lambda \leftarrow \Lambda \cup \{i\}$                    /* $i$ is still optional. */
5      else if $v_i$ then
6          $\Theta \leftarrow \Theta \cup \{i\}$       /* $i$ is known to be used by the machine. */
7          if $ect^*_\Theta > lct_i$ then
8              return $\bot$                          /* Infeasibility detected. */
9          end
10     end
11     while $\overline{ect}^*_{(\Theta,\Lambda)} > lct_i$ do
12         $o \leftarrow$ optional (gray) activity responsible for $\overline{ect}^*_{(\Theta,\Lambda)}$
13         $v_o \leftarrow$ false                  /* $o$ cannot run on the machine. */
14         $\Lambda \leftarrow \Lambda \backslash \{o\}$
15     end
16 end
```

Detectable Precedences Algorithm 3.2 describes how the DP inference rule can be applied. It first sorts the regular activities by non-decreasing order of latest start time and insert them into a queue Q (line 2)[5]. Then, it traverses all the activities (including optional ones as they can be updated): for each activity i, as long as its earliest completion time is strictly larger than the latest start time of the first activity j in Q, j is removed from the queue and added to the set Θ. Once this is done, Θ is the set $DPrec(R, i)$ (see DPrec), and we can apply the DP rule (line 9). Moreover, as transition times are involved, the minimal transition from any family $F_j \in F_\Theta$ to the family F_i can also be added as it was not taken into account in the computation of ect^*_Θ. This transition is the minimal one from any family $F_j \in F_\Theta$ to F_i, because we do not know which activity will be just before i in the final schedule. The detectable precedence update rule becomes:

$$est'_i \leftarrow \max\left\{est'_i, ect^*_\Theta + \min_{f \in F_\Theta} tt^{\mathcal{F}}_{f,F_i}\right\} \tag{DPUR}$$

Notice that the value $\min\limits_{f \in F_\Theta} tt^{\mathcal{F}}_{f,F_i}$ can only be available in $\mathcal{O}(1)$ if it was precomputed for any subset of families, which is exponential in $|\mathcal{F}|$ and therefore problematic if there are many families. It can also be computed in linear time, but it would increase the time complexity of the overall algorithm. In practice, the implementation can make use of the minimum transition from *any* family $f \in \mathcal{F} \setminus F_i$ *if* $F_i \notin F_\Theta$, and 0 otherwise.

When no transition times are involved, *detected* precedences are all eventually *propagated*, i.e., i precedes j if and only if $est_j \geq ect_i$ and $lct_i \leq lst_j$ (see [30]). In our case, this is not guaranteed: if a precedence is detected for a given pair of activities i and j, it is not ensured that after propagation we will have $est_j \geq ect_i + tt_{i,j}$ and $lct_i \leq est_j - tt_{i,j}$. The reason is that ect^*_Θ uses a lower bound on the transition times in Θ. One must therefore rely on branching (e.g., on the precedence graph) to ensure a given detected precedence is completely propagated.

Not Last The NL inference rule can be applied with Algorithm 3.3, similarly to Algorithm 3.2: a queue Q is filled with regular activities[6], and all activities (regular and optional) are then traversed in non-decreasing order of latest completion time. For each activity i, activities from Q having a larger latest starting time than the latest completion time of i, are removed from the queue and added to the set Θ (line 5-8). Θ is then the set of activities with a latest starting time stricly smaller than the latest completion time of i. The NL rule can then be applied (lines 9-11). An analogous reinforcement to the DP rule due to transition times can be applied when updating lct_i (see line 10).

[5] Alternatively, as proposed in [32,31] in the case of the Edge-Finding algorithm, one could consider all activities of T and pretend the latest start time of optional activities amounts to $+\infty$. All activities (including optionals) are then inserted in Q and the rest of the algorithm remains unchanged.

[6] Or as for the Detectable Precedence algorithm, one can pretend the latest start time of optional activities amounts to $+\infty$ and insert all of them in Q.

Algorithm 3.2: Detectable Precedences

```
1  $\Theta \leftarrow \emptyset$
2  $Q \leftarrow$ queue of all regular activities $r \in R$ in non-decreasing order of $lst_r$
3  $j \leftarrow Q.peek()$
4  for $i \in T$ in non-decreasing order of $ect_i$ do
5      while $ect_i > lst_j$ do
6          $\Theta \leftarrow \Theta \cup \{j\}$
7          $Q.pop()$
8          $j \leftarrow Q.peek()$
9      end
10     $est'_i \leftarrow \max\left\{est_i, ect^*_{\Theta\setminus\{i\}} + \min_{f \in F_\Theta} tt^{\mathcal{F}}_{f,F_i}\right\}$
11 end
12 for $i \in T$ do
13     $est_i \leftarrow est'_i$
14 end
```

Algorithm 3.3: Not-Last

```
1  $lct'_i \leftarrow lct_i, \forall i \in T$
2  $\Theta \leftarrow \emptyset$
3  $Q \leftarrow$ queue of all regular activities $r \in R$ in non-decreasing order of $lst_r$
4  $j \leftarrow Q.peek()$
5  for $i \in T$ in non-decreasing order of $lct_i$ do
6      while $lct_i > lst_j$ do
7          $\Theta \leftarrow \Theta \cup \{j\}$
8          $Q.pop()$
9          $j \leftarrow Q.peek()$
10     end
11     if $ect^*_{\Theta\setminus\{i\}} > lst_i$ then
12         $lct'_i \leftarrow \min\left\{lct'_i, lst_j - \min_{f \in F_\Theta} tt^{\mathcal{F}}_{F_i,f}\right\}$
13     end
14 end
15 for $i \in T$ do
16     $lct_i \leftarrow lct'_i$
17 end
```

Edge Finding Unlike the previous algorithms, Algorithm 3.4 starts with a set Θ filled with all regular activities. We also directly fill the set Λ with the optional activities[7] so that their domain can be updated but they can never be used to update other activities (since they will not be in the set Ω in the EF rule). A queue Q of regular activities sorted in non-increasing order of latest completion time is also initialized. The algorithm traverses this queue and the activities in Θ will progressively be removed from Θ and added to the set Λ of gray activities. For each activity j popped out the queue Q, the

[7] Notice that this is equivalent to what is proposed in [32,31]. The author suggests to handle optional activities by modifying the input data rather than the algorithm: lct_o is assumed to be $+\infty$ for all optional activities $o \in O$.

algorithm first checks for an overload, before j is removed from Θ (lines 5-7). This is equivalent to what is done in Algorithm 3.1 for regular activities, so it is actually facultative. The activity j is then grayed : it is transferred from Θ to Λ. This means it is no more in the set Θ we consider, but it will be part of the activities used to infer what would happen if one of them was added to Θ. Lines 10-14 apply the EF rule to the current gray activities such that $\overline{ect}^*_{(\Theta,\Lambda)} > lct_j$: as long as adding one of the gray activities would imply an overload (i.e., condition in line 10 is verified), we identify which gray activity i is responsible for this potential overload, we update its earliest start time, and remove it from Λ. The EF rule is strengthened using transition times similarly to the DP and NL rules.

Algorithm 3.4: Edge Finding

```
1  $(\Theta, \Lambda) \leftarrow (R, O)$
2  $Q \leftarrow$ queue of all regular activities $r \in R$ in non-increasing order of $lct_r$
3  $j \leftarrow Q.peek()$
4  while $|Q| > 1$ do
5      if $ect^*_\Theta > lct_j$ then
6          return $\perp$
7      end
8      $(\Theta, \Lambda) \leftarrow (\Theta \setminus \{j\}, \Lambda \cup \{j\})$
9      $Q.pop()$
10     $j \leftarrow Q.peek()$
11     while $\overline{ect}^*_{(\Theta,\Lambda)} > lct_j$ do
12         $i \leftarrow$ gray activity responsible for $\overline{ect}^*_{(\Theta,\Lambda)}$
13         $est'_i \leftarrow \max\left\{est_i, ect^*_\Theta + \min_{f \in F_\Theta} tt^{\mathcal{F}}_{f,F_i}\right\}$
14         $\Lambda \leftarrow \Lambda \setminus \{i\}$
15     end
16 end
17 for $i \in T$ do
18     $est_i \leftarrow est'_i$
19 end
```

Precedence Graph Propagator Algorithm 3.5 uses the precedence graph data structure (see Section 2). It relies on the topological order of all known precedences (i.e., edges in the digraph) since if i precedes j in the topological order of the precedence graph, the earliest start time of i cannot be influenced by the domain of s_j and c_j. Algorithm 3.5 first builds a queue Q of activities in topological order of the precedence graph. It then traverses Q and for each activity i, it applies the pairwise rule URTTO for all its successors in the precedence graph. In addition, if a successor j of an activity i is known to be running on the resource (i.e., v_j is true), then one can use j to update the latest completion time of the activity i (see lines 6-8).

Important note. Notice that when transition times are involved, this algorithm is mandatory in order to ensure the pruning is complete: because we use a lower bound of the earliest completion time of a set of activities Θ in the other algorithms (ect^*_Θ), they are are not sufficient to ensure correctness of a given (partial) assignment of all s_i, $\forall i \in T$.

Algorithm 3.5: Precedence Graph Propagation

```
1  $Q \leftarrow$ queue of all regular activities $r \in R$ in topological order in the precedence
     graph $G$
2  while $|Q| > 1$ do
3      $i \leftarrow Q.pop()$
4      foreach successor $s$ of $i$ in $G$ do
5          $est_s \leftarrow \max\{est_s, ect_i + tt_{i,s}\}$
6          if $v_s$ then
7              $lct_i \leftarrow \min\{lct_i, lst_i - tt_{i,s}\}$
8          end
9      end
10 end
```

Complexities Section 4 describes data structures that allow to retrieve ect^*_Θ in $\mathcal{O}(1)$ while addition/removal of an activity to/from Θ are performed in $\mathcal{O}(\log(|T|){\cdot}\log(|\mathcal{F}|))$. All algorithms but the Precedence Graph have therefore a time complexity of $\mathcal{O}(|T|{\cdot}\log(|T|){\cdot}\log(|\mathcal{F}|))$. The precedence graph propagator runs in $\mathcal{O}(|T|^2)$.

4 Extending the Θ-tree and Θ-Λ-tree Data Structures

To efficiently use the sets Θ and Λ, the algorithms described in Section 3.3 rely on the so-called Θ-tree and Θ-Λ-tree data structures, introduced by Vilím. Those structures are used to compute efficiently and incrementally ect^*_Θ and $\overline{ect}^*_\Theta$ for sets of activities Θ and Λ. This section describes how those can be extended to handle (family-based) transition times.

4.1 Extended Θ-tree

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

The main value stored in a node n is the lower bound of $ect_{Leaves(n)}$, denoted ect^*_n. 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.

Without any transition times involved, Vilím has shown [30] that by defining $ect^*_n = ect^{LB0}_{Leaves(n)}$, it suffices to store additionally $p_n = p_{Leaves(n)}$. In a leaf n representing an activity i, one can compute $p_n = p_i$ and $ect^*_n = ect_i$. In an internal node n, one can compute:

$$\begin{aligned} p_n &= p_{l(n)} + p_{r(n)} \\ ect^*_n &= \max\left\{ ect^*_{r(n)}, ect^*_{l(n)} + p_{r(n)} \right\} \end{aligned}$$

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

In our case, we would like instead to define $ect^*_n = ect^{LB3}_{Leaves(n)}$ in order to take (family-based) transition times into account. However, this value cannot easily be computed incrementally, so we compute a lower bound, i.e., $ect^*_n \leq ect^{LB3}_{Leaves(n)}$. In addition to ect^*_n, one needs to store not only p_n, but also $F_n = F_{Leaves(n)}$, the set of the families of the activities in $Leaves(n)$. In a leaf n representing an activity i, one can compute $p_n = p_i$, $ect^*_n = ect_i$, and $F_n = \{F_i\}$. In an internal node n, one can compute:

$$\begin{aligned} p_n &= p_{l(n)} + p_{r(n)} \\ F_n &= F_{l(n)} \cup F_{r(n)} \\ ect^*_n &= \max \begin{cases} ect^*_{r(n)} \\ ect^*_{l(n)} + p_{r(n)} + \underline{tt}\left(\left|F_{r(n)} \setminus F_{l(n)}\right| + 1\right) \end{cases} \end{aligned}$$

Intuitively, ect^*_n is maximized either by only considering activities in $r(n)$, or by adding to $ect^*_{l(n)}$ the processing times and (a lower bound of) the transition times due to activities in $r(n)$. In the latter case, only *additional* families are counted to compute the lower bound on transition times, that is, the families that are present in the right child but not in the left one. Hence, the cardinality of the set $F_{r(n)} \setminus F_{l(n)}$ is considered. Notice we always add 1 family to the count because of the definition of $\underline{tt}(k)$ (remember $\underline{tt}(0) = \underline{tt}(1) = 0$).

Before we prove this lower bound is correct, let us prove in Lemma 2 a property of the function $\underline{tt}(k)$.

Lemma 2 $\forall i \in [0..|T|], k \in [0..i] : \underline{tt}(i) \geq \underline{tt}(k) + \underline{tt}(i - k + 1)$

Proof The optimal path p^{opt} in the transition graph leading to the value $\underline{tt}(i)$ can be split in two subpaths:

- $p^{opt}_{[1..k]}$ with $k-1$ edges. Its total length is greater than or equal to $\underline{tt}(k)$ (as $\underline{tt}(k)$ is the minimum), the length of the optimal path with $k-1$ edges.
- $p^{opt}_{[k..i]}$ with $i-k$ edges. Its total length is greater than or equal to $\underline{tt}(i-k+1)$, the length of the optimal path with $i-k$ edges.

Therefore $\underline{tt}(n) = p^{opt}_{[1..k]} + p^{opt}_{[k..i]} \geq \underline{tt}(k) + \underline{tt}(i-k+1)$.

□

Lemma 3 $\forall$ *node n in a Θ-tree*: $ect^*_n \leq ect^{LB3}_{Leaves(n)}$

Proof By induction. If n is a leaf representing activity i, then $ect^*_n = ect_i = ect^{LB3}_{\{i\}}$. Otherwise, our induction hypothesis is that $ect^*_{l(n)} \leq ect^{LB3}_{Leaves(l(n))}$ and $ect^*_{r(n)} \leq ect^{LB3}_{Leaves(r(n))}$. Let us call $\Omega^{LB3} \subseteq Leaves(n)$ the optimal set to compute $ect^{LB3}_{Leaves(n)}$. For space reasons, we write $L(\Omega)$ to denote $Leaves(\Omega)$.

One can consider two cases:

- $ect^*_n = ect^*_{r(n)}$. We have $ect^*_{r(n)} \leq ect^{LB3}_{L(r(n))}$ (by induction) and $ect^{LB3}_{L(r(n))} \leq ect^{LB3}_{L(n)}$ (by definition). Therefore, $ect^*_n \leq ect^{LB3}_{L(n)}$.
- $ect^*_n = ect^*_{l(n)} + p_{r(n)} + \underline{tt}\big(|F_{r(n)} \setminus F_{l(n)}| + 1\big)$. Then, we have:

$$\begin{aligned}
ect^*_n &\leq ect^{LB3}_{L(l(n))} + p_{r(n)} + \underline{tt}\big(|F_{r(n)} \setminus F_{l(n)}| + 1\big) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{(by induction)} \\
&= \max_{\Omega_l \subseteq L(l(n))} \{est_{\Omega_l} + p_{\Omega_l} + \underline{tt}(|F_{\Omega_l}|)\} + p_{r(n)} + \underline{tt}\big(|F_{r(n)} \setminus F_{l(n)}| + 1\big) \\
&= \max_{\Omega_l \subseteq L(l(n))} \{est_{\Omega_l} + p_{\Omega_l \cup L(r(n))} + \underline{tt}(|F_{\Omega_l}|) + \underline{tt}\big(|F_{r(n)} \setminus F_{l(n)}| + 1\big)\} \\
&= \max_{\Omega_l \subseteq L(l(n))} \{est_{\Omega_l \cup L(r(n))} + p_{\Omega_l \cup L(r(n))} \\
&\qquad\qquad + \underline{tt}(|F_{\Omega_l}|) + \underline{tt}\big(|F_{r(n)} \setminus F_{l(n)}| + 1\big)\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{(since } est_{\Omega_l} = est_{\Omega_l \cup L(r(n))}\text{)} \\
&\leq \max_{\Omega_l \subseteq L(l(n))} \{est_{\Omega_l \cup L(r(n))} + p_{\Omega_l \cup L(r(n))} + \underline{tt}\big(|F_{\Omega_l \cup L(r(n))}|\big)\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{(by Lemma 2)} \\
&\leq ect^{LB3}_{L(n)} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{(by definition)}
\end{aligned}$$

□

Complexity We use bit sets to represent the set of families in each node. The space complexity of the Θ-tree is therefore $\mathcal{O}(|T| \cdot |\mathcal{F}|)$. The set operations we use are *union*, *intersection*, *difference* and *cardinality*. Using bit sets and assuming $|\mathcal{F}| \leq 64$,, the three former ones are $\mathcal{O}(1)$ and the latter one is $\mathcal{O}(\log(|\mathcal{F}|))$ with a *binary population count*[8] [34]. The time complexity of insertion and deletion of an activity in the Θ-tree is therefore $\mathcal{O}(\log(|T|) \cdot \log(|\mathcal{F}|))$.

Example 4 *Let us consider the activities presented in Figure 6 (left). The family transition matrix $tt^{\mathcal{F}}$ is given in Figure 6 (center). The pre-computed values of $\underline{tt}(k)$ are reported in Figure 6 (right). Figure 7 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 since we have $ect^*_\Theta = 75 \leq ect^{LB3}_\Theta = 80 \leq ect_\Theta = 85$.*

[8] Some processors also have a dedicated machine instruction.

	1	3	2	4
est	0	15	25	30
p	10	10	20	25
F	1	2	3	3

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

$tt(k)$	k
0	0
1	0
2	5
3	15

Fig. 6 Four activities and their families (left), transition times for the families (center), and pre-computed lower bounds for the transition times (right).

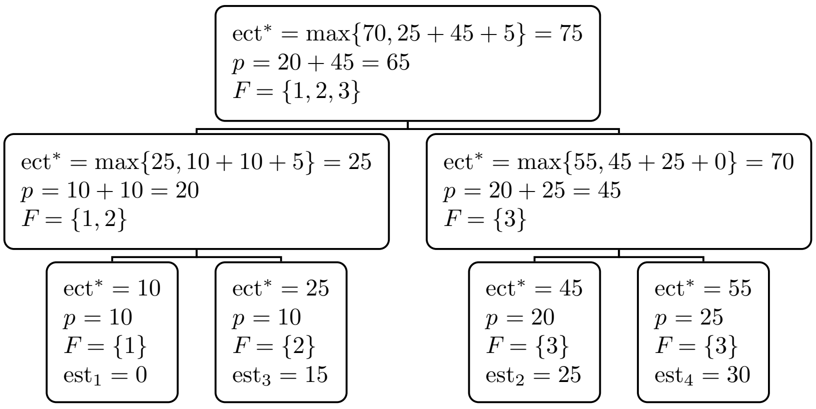


Fig. 7 A Θ-tree when all activities of Figure 6 are inserted.

4.2 Extended Θ-Λ-tree

Algorithms 3.1 and 3.4 require an extension of the original Θ-tree, called Θ-Λ-tree [30]. 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^*_n, a lower bound to the ect of Θ, a Θ-Λ-tree also aims at computing $\overline{ect}^*_n$, which is a lower bound to $\overline{ect}_{(\Theta,\Lambda)}$, the largest ect obtained by including *one* activity from Λ into Θ:

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

In addition to p_n, ect^*_n, Vilím's original Θ-Λ-tree also maintains $\overline{p}_n$ and $\overline{ect}^*_n$, respectively corresponding to p_n and ect^*_n, *if* a single gray activity $i \in \Lambda$ in the sub-tree rooted at n maximizing $ect_{Leaves(v) \cup \{i\}}$ *was* included.

Our extension to the Θ-Λ-tree is similar to the one outlined in Section 4.1 for the Θ-tree: in addition to the previous values, each node also stores $\overline{p}_n$ and $\overline{F}_n$ in order to compute the lower bound $\overline{ect}^*_n$.

Adapting the rules for the Θ-Λ-tree requires caution when families are involved. In [30] and [14], 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 responsible gray activity (i.e., the gray activity maximizing $\overline{ect}^*$ at the root node) is located. Hence, when a node n is updated, one first update $\overline{ect}^*_n$ with the rule:

$$\overline{ect}^*_n = \max \begin{cases} \overline{ect}^*_{l(n)} + p_{r(n)} + \underline{tt}\big(|F_{r(n)} \setminus \overline{F}_{l(n)}| + 1\big) & \text{(Case A)} \\ ect^*_{l(n)} + \overline{p}_{r(n)} + \underline{tt}\big(|\overline{F}_{r(n)} \setminus F_{l(n)}| + 1\big) & \text{(Case B)} \\ \overline{ect}^*_{r(n)} & \text{(Case C)} \end{cases}$$

Case A occurs when it is (locally) considered that the gray responsible activity that maximizes $\overline{ect}^*_n$ is among $Leaves(l(n))$. Cases B and C correspond to the opposite case (i.e., the responsible activity is among $Leaves(r(n))$). Depending on which value gets assigned to $\overline{ect}^*_n$, the values $\overline{F}_n$ and $\overline{p}_n$ of the node n are updated, as follows:

$$\overline{F}_n = \begin{cases} \overline{F}_{l(n)} \cup F_{r(n)} & \text{if (Case A)} \\ F_{l(n)} \cup \overline{F}_{r(n)} & \text{otherwise} \end{cases}$$
$$\overline{p}_n = \begin{cases} \overline{p}_{l(n)} + p_{r(n)} & \text{if (Case A)} \\ p_{l(n)} + \overline{p}_{r(n)} & \text{otherwise} \end{cases}$$

If a leaf n represents an activity i, then we simply have $\overline{ect}^*_n = ect_i$, $\overline{p}_n = p_i$, and $\overline{F}_n = \{F_i\}$. The rules for p_n, ect_n, and F_n are as presented in Section 4.1, but one must also define, for a gray leaf n, $ect^*_n = -\infty$, $p_n = 0$, and $F_n = \emptyset$.

Example 5 *Let us reconsider the activities from Figure 6. Figure 8 illustrates a Θ-Λ-tree where all activities have been inserted, but where activities 3 and 4 have been grayed. Notice that the activity 4 is the gray responsible one (since* $70 > 25 + 20 + 5$*) and therefore* $\overline{p} = 55$ *and* $\overline{F} = \{F_1, F_3\}$ *in the root node.*

As for the extended Θ-tree introduced in Section 4.1, the time complexity for the insertion and the deletion of an activity is $\mathcal{O}(\log(|T|) \cdot \log(|\mathcal{F}|))$. Table 1 summarizes the complexities of all operations on the Θ-Λ-tree.

Operation	**Time Complexity**
$(\Theta, \Lambda) \leftarrow (\emptyset, \emptyset)$	$\mathcal{O}(1)$
$(\Theta, \Lambda) \leftarrow (R, O)$	$\mathcal{O}(\|T\| \cdot \log(\|T\|) \cdot \log(\|\mathcal{F}\|))$
$(\Theta, \Lambda) \leftarrow (\Theta \setminus \{i\}, \Lambda \cup \{i\})$	$\mathcal{O}(\log(\|T\|) \cdot \log(\|\mathcal{F}\|))$
$\Theta \leftarrow \Theta \cup \{i\}$	$\mathcal{O}(\log(\|T\|) \cdot \log(\|\mathcal{F}\|))$
$\Lambda \leftarrow \Lambda \setminus \{i\}$	$\mathcal{O}(\log(\|T\|) \cdot \log(\|\mathcal{F}\|))$
$ect^*_\Theta > lct_i$	$\mathcal{O}(1)$
$\overline{ect}^*_{(\Theta,\Lambda)} > lct_i$	$\mathcal{O}(1)$

Table 1 Worst-case time complexities of operations on the Θ-Λ-tree.

4.3 Strengthening ect^*_Θ and $\overline{ect}^*_{(\Theta,\Lambda)}$

The value ect^*_Θ is a lower bound for ect^{LB3}_Θ. One can actually strengthen the value computed with the Θ-tree to get a value closer to ect^{LB3}_Θ. An idea from

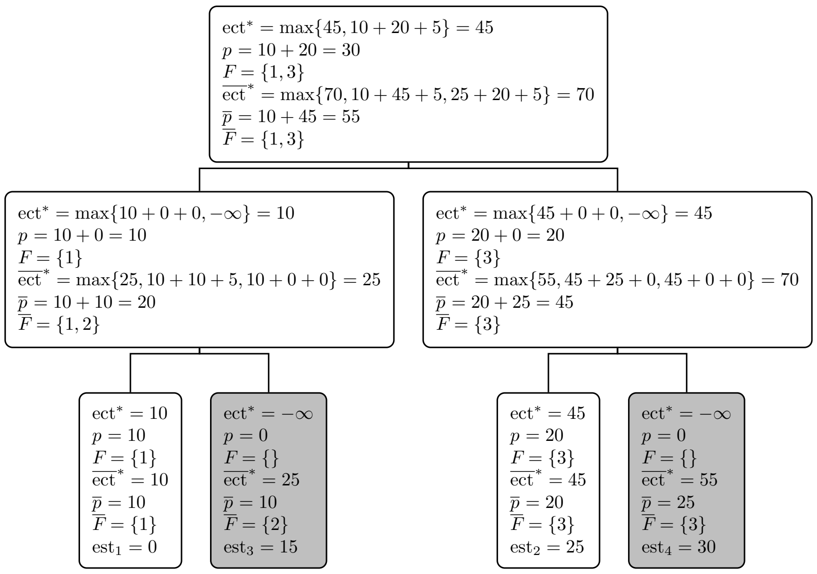


Fig. 8 A Θ-Λ-tree when all activities of Figure 6 are inserted and activities 3 and 4 are gray.

[10,33] that is also used in [3] is to pre-compute the exact minimum total transition time for every subset of families[9].

For a subset of families $\mathcal{F}' \subseteq \mathcal{F}$, let $tt(\mathcal{F}')$ denote the minimum total transition time used for any activity set Θ such that $F_\Theta = \mathcal{F}'$. Assuming $tt(F_\Theta)$ is accessible in $\mathcal{O}(1)$, each time we access to the value ect^*_Θ in the algorithms of Section 3.3, we can also compute

$$ect^{tsp}_\Theta = est_\Theta + p_\Theta + tt(F_\Theta)$$

without changing the complexity of the algorithms. The value $tt(F_\Theta)$ must be precomputed for all subsets of families, so this is tractable only if there are few families[10] as it requires solving many Traveling Salesman Problems of increasing sizes. Moreover, it is necessary to store $2^{|\mathcal{F}|}$ integers in an array. One can then use the bit set representation of a given set $\mathcal{F}' \subseteq \mathcal{F}$ as an index in the array in order to access the value in $\mathcal{O}(1)$. The value est_Θ can be easily maintained in the Θ-tree, and the values p_Θ and F_Θ can be obtained in $\mathcal{O}(1)$ in the root node of the Θ-tree.

The value ect^{tsp}_Θ can be larger than ect^*_Θ because it uses $tt(F_\Theta)$ instead of $\underline{tt}(|F_\Theta|)$. This typically occurs when $ect^{tsp}_\Theta = ect^{LB3}_\Theta$. On the contrary, ect^{tsp}_Θ might be smaller than ect^*_Θ since ect^{tsp}_Θ always considers all activities in Θ but never a subset $\Theta' \subset \Theta$. Yet, ect^*_Θ can rely on a subset $\Theta' \subset \Theta$ such that $est_\Theta + p_\Theta < est_{\Theta'} + p_{\Theta'}$. Hence, the algorithms in Section 3.3 should use

[9] The approach can also be used for sets of activities. The description focuses here on families since it was initially used in the context of family-based transition times.

[10] Typically maximum 10.

the maximum of those two values instead of ect^*_Θ in order to strengthen the filtering.

One can also consider the family of the updated activity: similarly to $tt\,(\mathcal{F}')$, let us write $tt\,(F_i \to \mathcal{F}')$ the minimum transition time when the processing starts with some activity of the family $F_i \in \mathcal{F}'$, and $tt\,(\mathcal{F}' \to F_i)$ when it completes with an activity of the family $F_i \in \mathcal{F}'$. 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 $\Theta(|\mathcal{F}|^2 \cdot 2^{|\mathcal{F}|})$ and requiring $\Theta(|\mathcal{F}| \cdot 2^{|\mathcal{F}|})$ of memory. For instance, for $tt\,(F_i \to \mathcal{F}')$, one defines:

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

In the case of Detectable Precedences, Equation DPUR finally becomes:

$$est'_i \leftarrow \max\left\{ est'_i,\, ect^*_\Theta + \min_{f \in F_\Theta} tt^{\mathcal{F}}_{f,F_i},\, est_\Theta + p_\Theta + tt\,(F_\Theta \to F_i) \right\}$$

The same idea can be used to strengthen $\overline{ect}^*_{(\Theta,\Lambda)}$:

$$\overline{ect}^{tsp}_{(\Theta,\Lambda)} = \min\{est_\Theta, est_r\} + p_{\Theta \cup \{r\}} + tt\left(F_{\Theta \cup \{r\}}\right)$$

where r is the gray responsible activity (see line 11 in Section 3.4). A subtle point is that the responsible activity r is not accessible from the Θ-Λ-tree as for $\overline{ect}^*_{(\Theta,\Lambda)}$, so we should iterate over all $r' \in \Lambda$ to maximize $\overline{ect}^{tsp}_{(\Theta,\Lambda)}$. We therefore use the responsible activity of $\overline{ect}^*_{(\Theta,\Lambda)}$ to compute $\overline{ect}^{tsp}_{(\Theta,\Lambda)}$.

5 Lower Bounds on the Minimum Total Transition of a Set of Activities

In this section, we describe different lower bounds [14] for Equation 4, recalled hereafter:

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

For each k, one has to find the shortest node-distinct $(k{-}1)$-edge path between any two nodes of the (family) transition graph (see Section 2), which is NP-hard as the Traveling Salesman Problem can be reduced to this problem when $k = |T|$. Even though $\underline{tt}(k)$ is to be precomputed, it is desirable to have polynomial precomputation, which justifies the use of the lower bounds explained in this section. A more detailed description can be found in [13], we summarize them here so that the paper is self-contained. Notice that the lower bounds do not dominate each other, so the final lower bound for a given cardinality k will be the maximum between the different lower bounds for this cardinality.

Minimum Weight Forest This lower bound consists of finding the set of $k-1$ edges with a minimum cost. Basically, we use Kruskal's algorithm [20] to prevent cycles in our selection. As soon as $k-1$ edges have been selected, the algorithm is stopped. The result being a minimum weight forest in the general case, it is a lower bound of our original problem since it does not ensure to obtain a simple path in the graph.

Shortest Walk A dynamic program can be used to compute a lower bound on the minimum transition in a set of cardinality k. The idea is to compute a shortest walk with $k-1$ edges in the transition graph. Formally, we define $SW(k', i)$ as the shortest walk with k' edges from any node to node i. To compute this value for all number of edges k' and every node i, we rely on the following $\mathcal{O}(k \cdot T^2)$ dynamic program:

$$\begin{aligned} SW(0,i) &= 0, \forall i \in [1..T] \\ SW(k+1,i) &= \min_j SW(k,j) + tt_{i,j}, \forall i \in [1..T] \end{aligned}$$

The lower bound for a given cardinality k is finally:

$$\min_i SW(k,i)$$

Notice this lower bound ensures the solution to be a walk in the graph but it does not prevent cycles. However, as suggested in [12], one can strengthen the bound by avoiding 1-cycles, i.e., cycles of the form $i \to j \to i$.

Minimum Assignment A lower bound based on a Minimum Assignment problem was proposed by Brucker and Thiele [10]: two sets containing all the nodes of the transition graph are constructed and a minimum assignment of k edges is searched for, that is, the edges always link an activity of one set with an activity of the other set. One can model this problem as a Minimum-Cost Maximum-Flow problem in a manner similar to the reduction of a minimum weight bipartite matching.

Lagrangian Relaxation To find the shortest simple path with k edges in the transition graph, one can add a source (node 0) and a sink node (node n + 1) to the transition graph so that the edges from the source node to all nodes (but the sink one) and the edges from the nodes (but the source one) to the sink node have a transition of zero. Then, one can solve the problem by searching for the shortest path from the source to the sink with $k+2$ edges. This can be solved with the following integer linear program:

$$
\begin{aligned}
\textbf{minimize} \quad & \sum_i \sum_j tt_{i,j} \cdot x_{i,j} \\
\textbf{such that} \quad & \sum_j x_{0,j} - \sum_j x_{j,0} = 1 \\
& \sum_j x_{n+1,j} - \sum_j x_{j,n+1} = -1 \\
& \sum_j x_{i,j} - \sum_j x_{j,i} = 0 \\
& \sum_i \sum_j x_{i,j} = k \qquad \text{(CARD)} \\
& x_{i,j} \in \{0,1\}
\end{aligned}
$$

This problem is NP-hard, therefore we solve a Lagrangian relaxation instead: we remove the edge cardinality constraint (i.e., Equation CARD) and penalize its violation in the objective function. Without the cardinality constraint, the shortest path can be computed with the Bellman-Ford algorithm [8, 24] that is also able to detect a negative cycle. If this occurs, we use a classic linear relaxation instead of using the Bellman-Ford algorithm.

Exact Shortest Path for every Subset Using the definitions given in Section 4.3, one can compute the best possible lower bound based on the cardinality of a set of activities/families. We compute the value of the shortest path for every subset, and for each cardinality k, we take the smallest shortest path of all subsets of cardinality k:

$$\underline{tt}(k) = \min_{|\mathcal{F}'|=k} tt\left(\mathcal{F}'\right)$$

The other lower bounds described before are upper bounded by this approach. However, it is not polynomial, so it can only be used for problems with a few activities/families.

6 Experimentations

We split our evaluation in two parts: first, we consider the case where there are no optional activities, which was more studied in the literature. The experiments were conducted on Job-Shop Problem with Sequence Dependent Transition Times (JSPSDTT) instances. In a second time, we consider the same problem with alternative machines, that is modeled using optional activities from the resource point of view.

Setting We used AMD Opteron processors (2.7 GHz), the Java Runtime Environment 8 and the constraint solver *OscaR* [25]. The memory consumption was limited to 4GB.

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 [27,29]. 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. 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* [15], that we built with a public web tool [28] made available to the community.[11] 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:

$$F_m(\tau) = \frac{1}{|\mathcal{I}|} \left| \left\{ i \in \mathcal{I} : \frac{time_{replay}(m,i)}{\min\limits_{m' \in M} time_{replay}(m',i)} \leq \tau \right\} \right| \qquad (7)$$

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.

A performance profile that is above the other ones in its graphical representation shows a higher performance than the others. This specific representation allows to have a global understanding of the actual performances of a propagator over a full set of instances at a glance.

Let us for example, consider a performance profile with a performance metric ratio τ representing the time needed to replay instances using a given propagator. If this performance profile has a point in (30% of instances, 2.5), it means that for 30% of the considered instances, the propagator takes at most 2.5 times as much time as the baseline model.

6.1 Experimentations without Optional Activities

6.1.1 Problem instances

We have used two sets of instances. First, we used the standard **t2ps** instances from Brucker and Thiele [10]. 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. The transition times between two families was randomly picked between 5 and 50, and duration of activities were randomly taken between 10 and 100.[12]

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

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

State-of-the-art filtering with Families

Based on the definition of $tt\,(F_i \rightarrow \{\mathcal{F}'\})$, two propagators are introduced in [3]:

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

Although the filtering obtained with these propagators can be stronger than their counterpart from [30] and our extensions, the time complexity of the propagators is quite high as compared to $\mathcal{O}(n \cdot \log(n) \cdot \log(|\mathcal{F}|))$. In addition, they do not make use of a Not-First/Not-Last rule and the pre-computation of the minimum exact transition times for every subset of family is only tractable for small (typically less than 10) values of $|\mathcal{F}|$.

6.1.2 Compared Propagators

We compare models with the following propagators for Equation (URTTO):

- *decomp*: binary decomposition of Equation (URTTO) only.
- *urtt*: propagators for URTT from [14].
- art_{ex}: propagators of [3] using exact values for $tt\,(\mathcal{F})$, $tt\,(F \rightarrow \mathcal{F})$ and $tt\,(\mathcal{F} \rightarrow F)$.
- art_{lb}: propagators of [3] adapted to make use of cardinality-based lower bounds from Section 5 for $tt\,(\mathcal{F})$, $tt\,(F \rightarrow \mathcal{F})$ and $tt\,(\mathcal{F} \rightarrow F)$.
- $urttf_{ex}$: propagators introduced in this paper making use of the exact values for $\underline{tt}(|\mathcal{F}|)$ computed with $\min_{\mathcal{F}':|\mathcal{F}'|=|\mathcal{F}|} tt\,(\mathcal{F}')$.
- $urttf_{lb}$: propagators introduced in this paper making use of lower bounds of Section 5 for $\underline{tt}(|\mathcal{F}|)$.

6.1.3 Replay Evaluation

To generate the search trees, the Conflict Ordering Search [17] was used, as it was shown to be a good search strategy for scheduling problems. The baseline model is *decomp*. The generation lasted for 300 seconds, and we enforced a timeout of 1,800 seconds for the replay. 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.).

*6.1.4 Results on the **t2ps** Instances*

Figures 9 and 10 provide the performance profiles for the time and number of backtracks, respectively. Figure 10 shows that, interestingly, $urttf_{lb}$ prunes exactly as much as $urttf_{ex}$. 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

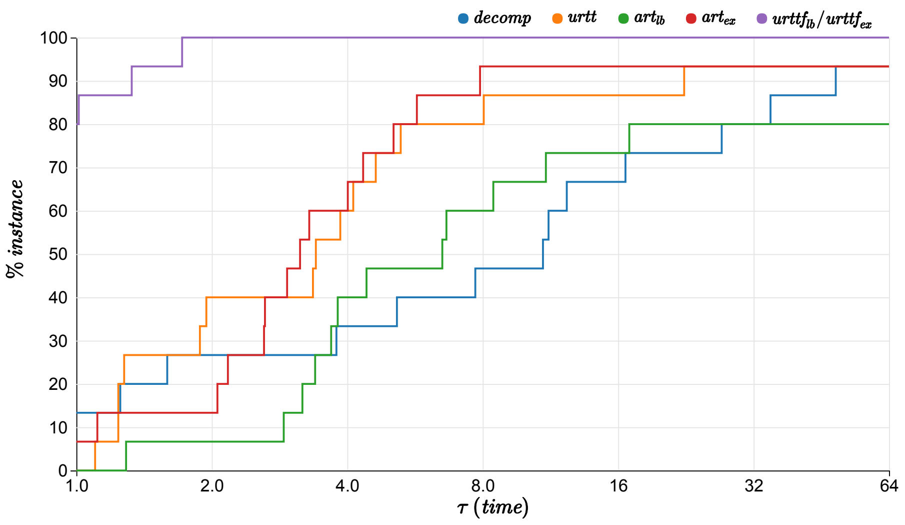


Fig. 9 Performance profiles on **t2ps** instances for the time metric.

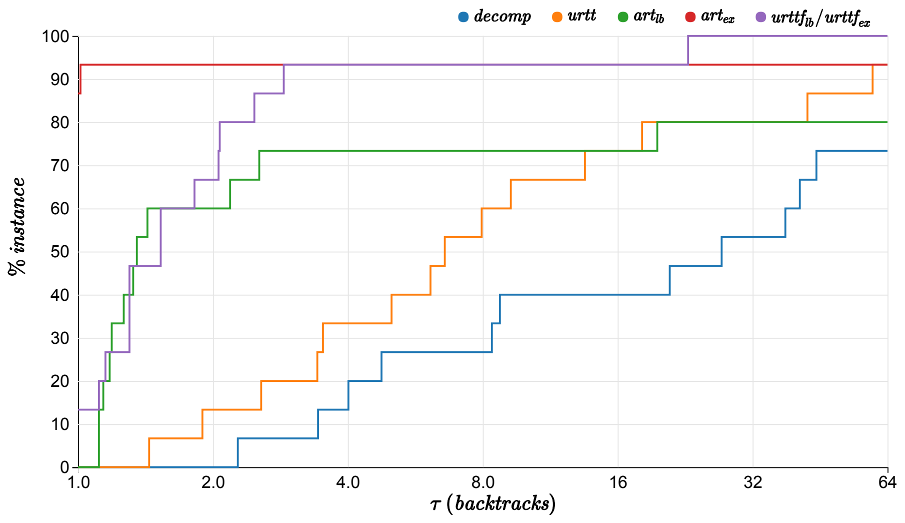


Fig. 10 Performance profiles on **t2ps** instances for the number of backtracks metric.

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 (Figure 9), our approach is the fastest for $\sim 80\%$ of the instances ($urttf_{ex}$ being here equivalent to $urttf_{lb}$, see the function in $\tau = 1$ in Figure 9). 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 art_{ex} (not more than three times), but substantially more than *urtt*. This lack of pruning as compared to art_{ex} is compensated in practice by the low time complexity. Although not reported, we tried to combine $urttf_{ex}$ and art_{ex} and the performances were close to the ones of art_{ex} alone, thus only inducing a small overhead when $urttf_{ex}$ does not provide additional pruning.

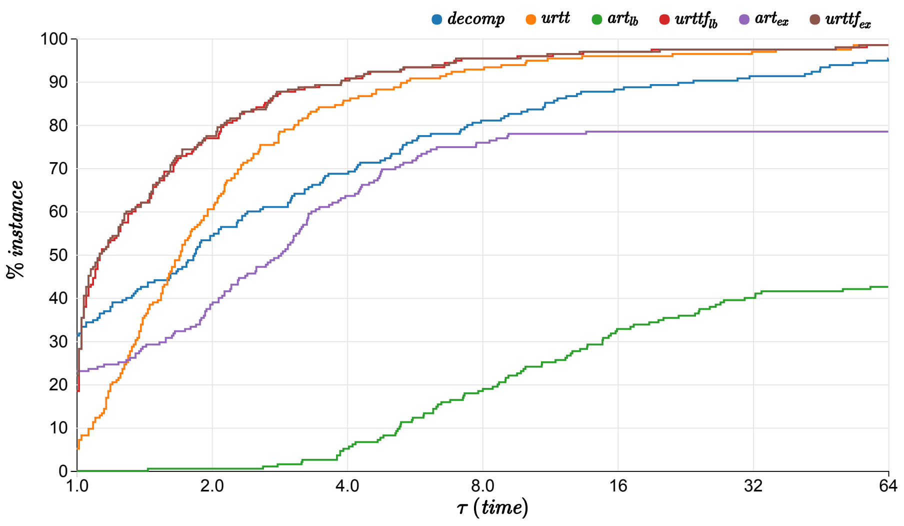


Fig. 11 Performance profiles on **uttf** instances with strictly less than 20 families for the time metric.

6.1.5 Results on the **uttf** Instances

First of all, we consider the approaches art_{ex} and $urttf_{ex}$ 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 11 and 12 provide the time performance profiles for the instances with strictly less than and with more than 20 families, respectively. Figure 11 shows that our approach still outperforms the other ones, although 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 $urttf_{ex}$ and $urttf_{lb}$ 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[13] when used with our

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

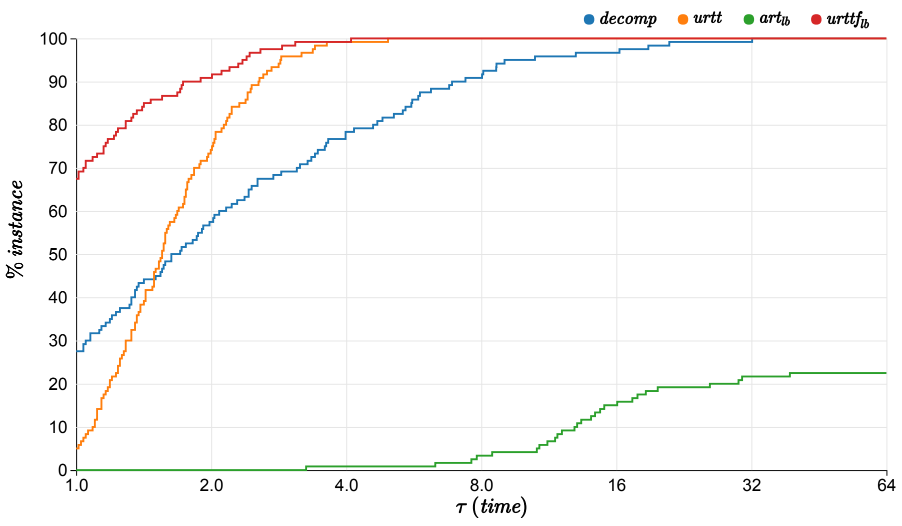


Fig. 12 Performance profiles on **uttf** instances with more than 20 families for the time metric.

propagators, which is profitable since we also target scalability in terms of number of families.

Regarding the instances with more than 20 families (Figure 12), 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 remaining instances. This teaches us that when more families are involved, our approach is both efficient and robust.

6.2 Experimentations with Optional Activities

Optional activities are typically used when modeling problems where activities can be processed on a set of a alternative resources. Hence, in order to experiment with our approach when optional activities are involved, we experimented on JSPSDTT with alternative resources. In particular, we used an approach that consists in duplicating a times the activities and the resources of an original Job Shop problem [16]. For each of the original activities, exactly one of its duplicates must then be executed on its corresponding duplicated machine. This amounts to solving the same problem as the original one, but with the additional liberty of choosing on which one of the a alternative machines an activity will be executed.

Formally, for a given activity i and a duplications, we write i_k the k^{th} duplicate of activity i. To ensure that one and only one of the alternative machines is used by the activity i, we force one and only one of the a duplicates i_k to be used by its corresponding duplicated machine:

$$\exists!\, k \in [1, a] : v_{i_k}$$

Moreover, the job precedences between activities must be respected by all duplicates, i.e., if there is a precedence between two activities i and j in the original problem, then we must have:

$$\forall k \in [1, a] \quad \forall k' \in [1, a] : k \neq k' \implies c_{i_k} \leq s_{j_{k'}}$$

Search Heuristic To our knowledge, few search heuristics are actually devoted to the presence of optional activities. For our evaluation, we used a strategy from Barták that avoids taking decisions about optional activities that will actually not be executed in the final schedule [6,11]. This is important, as it prevents the search to explore several times the exact same schedule.

The heuristic has two levels: on the first level, it decides wether an activity i is valid or not, i.e., it branches on v_i. On the left branch, it imposes $v_i = \mathit{true}$, and will then branch using the second level, as explained hereafter. On the right branch, $v_i = \mathit{false}$ is posted and the activity i will not be considered deeper in the tree. An other activity $j \neq i$ will then be considered to be branched on using the first level. In the second level, precedences between i and all activities $j : \neg(v_j = \mathbf{false})$ (i.e., still possibly running on the same resource) will be imposed, until no more precedences involving i can be decided. The first level of branching is then used with a different activity j.

To decide which activity should be branch on first, the activity with the smallest *est* is chosen (ties are breaked by smallest duration and *ect*). Finally, once all decisions have been made, one can assign all activities to their *est* since the objective is here to minimize the makespan.

Settings We generated 100 instances similar to the five small **t2ps** instances, i.e., with 10 jobs, 5 machines and 5 families. The instances are kept small because duplicating the alternatives already increase substantially the search space. The models we compared are the same ones as before, but the approach from Artigues et al., as they do not deal with optional activities. Our approach use lower bounds for $\underline{tt}(|\mathcal{F}|)$. We also consider an additional model, called urV, that uses the filtering from Vilím.

We also used the *Replay* evaluation: the generation lasted at most 300 seconds and we filtered out instances that were solved within less than a second.

Results First, we consider the problem with two alternative resources. The results are given in Figure 13. A first observation if that urttf_{lb} is almost always the fastest and it solves all instances in $\tau < 2$, which makes our approach appealing. Interestingly, one can also see that the profiles of the other approaches are in this case quite similar. Finally, for $\sim 10\%$ of the instances, urttf_{lb} provides a speed-up of ~ 32 as compared to the other approaches (see the profiles in $\tau = 32$ in Figure 13).

Let us now consider the results (given in Figure 14) when we have three alternative resources. While our approach is still clearly the best one for similar reasons, one can now better separate *decomp*, urV, and *urtt*: urV is better than *decomp* and *urtt* is better than urV. Still, *urtt* and urV are close to each

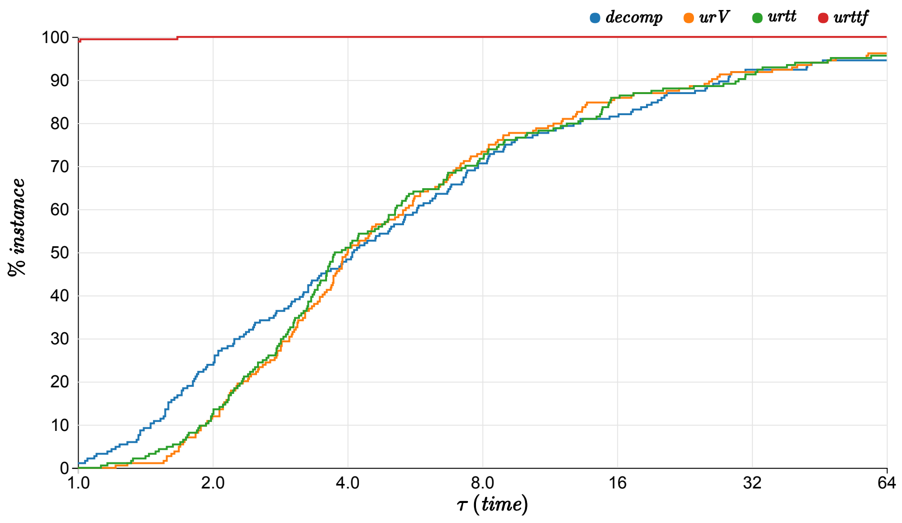


Fig. 13 Performance profiles on generated instances of the Job Shop problem with two alternative resources.

other, and tends to converge. This shows again the benefits of reasoning with families of activities.

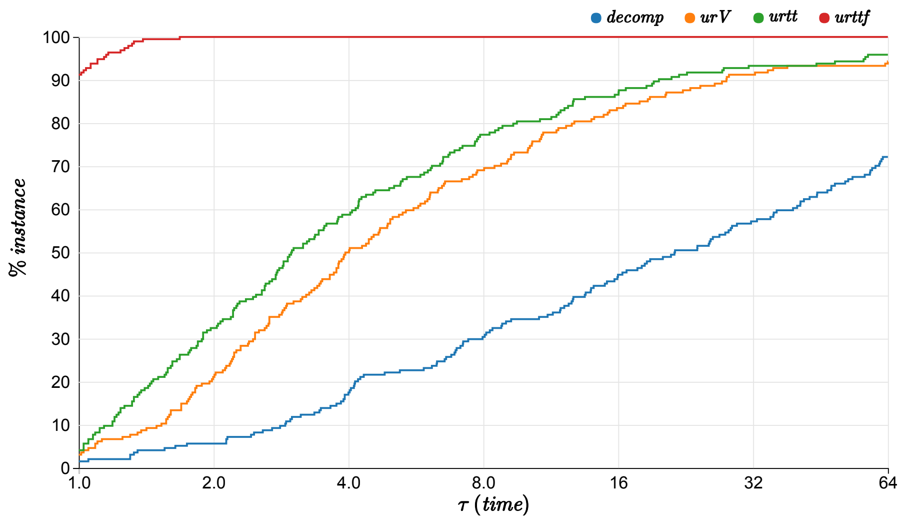


Fig. 14 Performance profiles on generated instances of the Job Shop problem with three alternative resources.

7 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 method also handles optional activities so that one can model more general problems (e.g., involving alternative resources). The original data structures and algorithms have been adapted accordingly. The approach is lightweight from both the time and space perspectives. Experiments conducted on the Job-Shop Problem with Sequence Dependent Transition Times have demonstrated that our work provides a substantial gain and is quite robust to changes in instance characteristics (e.g., number of activities and families).

We would like to consider other types of problems (e.g., the Traveling Salesman Problem with Time Windows) 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. Baptiste, P., Laborie, P., Le Pape, C., Nuijten, W.: Constraint-based scheduling and planning. In: F. Rossi, P. van Beek, T. Walsh (eds.) Handbook of Constraint Programming, chap. 22, pp. 759–797. Elsevier (2006)
5. Baptiste, P., Le Pape, C., Nuijten, W.: Constraint-based scheduling: applying constraint programming to scheduling problems, *International Series in Operations Research & Management Science*, vol. 39. Springer (2001)
6. Barták, R.: Search strategies for scheduling problems with optional activities. In: Proceedings of CSCLP 2008 Annual ERCIM Workshop on Constraint Solving and Constraint Logic Programming, Rome (2008)
7. Barták, R., Čepek, O.: Incremental propagation rules for a precedence graph with optional activities and time windows. Transactions of the Institute of Measurement and Control **32**(1), 73–96 (2010)
8. Bellman, R.: On a routing problem. Tech. rep., DTIC Document (1956)
9. Brucker, P.: Complex scheduling problems. In: Zeitschrift Oper. Res. Citeseer (1999)
10. 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)
11. Cappart, Q., Thomas, C., Schaus, P., Rousseau, L.M.: A constraint programming approach for solving patient transportation problems. In: International Conference on Principles and Practice of Constraint Programming, pp. 490–506. Springer (2018)

12. Christofides, N., Mingozzi, A., Toth, P.: State-space relaxation procedures for the computation of bounds to routing problems. Networks **11**(2), 145–164 (1981)
13. Dejemeppe, C.: Constraint programming algorithms and models for scheduling applications. Ph.D. thesis, Université catholique de Louvain, Louvain-la-Neuve (2016)
14. Dejemeppe, C., Van Cauwelaert, S., Schaus, P.: The unary resource with transition times. In: International Conference on Principles and Practice of Constraint Programming, pp. 89–104. Springer (2015)
15. Dolan, E.D., Moré, J.J.: Benchmarking optimization software with performance profiles. Mathematical programming **91**(2), 201–213 (2002)
16. Focacci, F., Laborie, P., Nuijten, W.: Solving scheduling problems with setup times and alternative resources. In: AIPS, pp. 92–101 (2000)
17. Gay, S., Hartert, R., Lecoutre, C., Schaus, P.: Conflict ordering search for scheduling problems. In: International Conference on Principles and Practice of Constraint Programming, pp. 140–148. Springer (2015)
18. 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)
19. 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)
20. Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical society **7**(1), 48–50 (1956)
21. Laborie, P.: An update on the comparison of mip, cp and hybrid approaches for mixed resource allocation and scheduling. In: International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research, pp. 403–411. Springer (2018)
22. Laborie, P., Rogerie, J.: Reasoning with conditional time-intervals. In: FLAIRS conference, pp. 555–560 (2008)
23. Laborie, P., Rogerie, J., Shaw, P., Vilím, P.: Ibm ilog cp optimizer for scheduling. Constraints **23**(2), 210–250 (2018)
24. Moore, E.: The shortest path through a maze. In: Proc. Internat. Sympos. Switching Theory (1959)
25. OscaR Team: OscaR: Scala in OR (2012). Available from `https://bitbucket.org/oscarlib/oscar`
26. Ozolins, A.: Bounded dynamic programming algorithm for the job shop problem with sequence dependent setup times. Operational Research (2018). DOI 10.1007/s12351-018-0381-6. URL `https://doi.org/10.1007/s12351-018-0381-6`
27. Van Cauwelaert, S., Lombardi, M., Schaus, P.: Understanding the potential of propagators. In: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming, pp. 427–436. Springer (2015)
28. Van Cauwelaert, S., Lombardi, M., Schaus, P.: A visual web tool to perform what-if analysis of optimization approaches. Tech. rep., UCLouvain (2016)
29. Van Cauwelaert, S., Lombardi, M., Schaus, P.: How efficient is a global constraint in practice? Constraints pp. 1–36 (2017)
30. Vilım, P.: Global constraints in scheduling. Ph.D. 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)
31. Vilım, P.: Edge finding filtering algorithm for discrete cumulative resources in o (kn log n). In: Principles and Practice of Constraint Programming-CP, vol. 5732, pp. 802–816. Springer (2009)
32. Vilím, P.: Max energy filtering algorithm for discrete cumulative resources. In: International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems, pp. 294–308. Springer (2009)
33. 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)

34. Warren, H.S.: Hacker's delight. Pearson Education (2013)
35. Wolf, A.: Constraint-based task scheduling with sequence dependent setup times, time windows and breaks. GI Jahrestagung **154**, 3205–3219 (2009)
36. 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)