Efficient Edge-Finding on Unary Resources with Optional Activities

Revised and Extended Version

Sebastian Kuhnert

Humboldt-Universität zu Berlin, Institut für Informatik
Unter den Linden 6, 10099 Berlin, Germany
kuhnert@informatik.hu-berlin.de

Abstract. Unary resources play a central role in modelling scheduling problems. Edge-finding is one of the most popular techniques to deal with unary resources in constraint programming environments. Often it depends on external factors if an activity will be included in the final schedule, making the activity optional. Currently known edge-finding algorithms cannot take optional activities into account. This paper introduces an edge-finding algorithm that finds restrictions for enabled *and* optional activities. The performance of this new algorithm is studied for modified job-shop and random-placement problems.

Keywords: constraint-based scheduling, global constraints, optional tasks and activities, unary resources

1 Unary Resources with Optional Activities

Many everyday scheduling problems deal with allocation of resources: Schools must assign rooms to lectures, factories must assign machines to tasks, train companies must assign tracks to trains. These problems are of high combinatorial complexity: For most of them there is no known polynomial time algorithm to find an optimal solution. Constraint programming offers a way to solve many instances of these problems in acceptable time [1].

Often an activity on a resource must be finished before a deadline and cannot be started before a release time. In this paper the following formalisation of scheduling problems is used: An **activity** (or task) i is described by its duration p_i and its earliest and latest starting and completion times (abbreviated as est_i, lst_i, ect_i, lct_i, respectively) as shown in Fig. 1. In constraint programming systems the start time is usually stored as a variable ranging from est_i to lst_i. The duration is constant in most applications; in this case the completion times can be derived as $\mathrm{ect}_i = \mathrm{est}_i + \mathrm{p}_i$ and $\mathrm{lct}_i = \mathrm{lst}_i + \mathrm{p}_i$. Otherwise let p_i denote the minimum duration for the purposes of this paper.[1]

[1] The values of p_i are used to restrict the domains of other variables. Using larger values than the minimum duration might cause unwarranted restrictions.

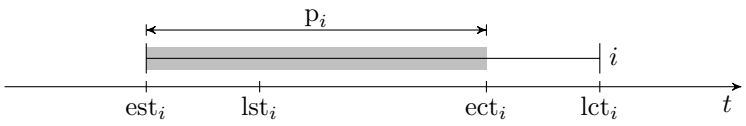


Fig. 1. Attributes of an activity i

Often sets Θ of activities are considered. The notions of production time and earliest starting time easily generalise to this case:

$$p_\Theta := \sum_{i \in \Theta} p_i \qquad\qquad est_\Theta := \min\{est_i \mid i \in \Theta\}$$

For $\Theta = \emptyset$ define $p_\emptyset := 0$ and $est_\emptyset := -\infty$.

A resource is called **unary resource**, if it can process only one activity at once and tasks cannot be interrupted. Now consider the following problem: Is there a valid schedule for a unary resource that includes all activities of a given set T, i. e. can the activities be assigned pairwise disjoint time periods satisfying the constraints given by the respective est_i, lst_i, p_i, ect_i and lst_i? See Fig. 2 for a sample instance. When the set T of activities grows, the number of possibilities to order them grows exponentially; it is long known that the unary resource scheduling problem is NP-complete [4, p. 236]. In constraint programming, the number of considered activity orderings is reduced in the following way: For each possible ordering of activities i and j (i before j or j before i), *restrictions* for the placement of other activities are derived before any other ordering decision is made. The process of searching for such restrictions is called *filtering*. Whenever a restriction is found, values can be removed from the domains of the variables describing the restricted activity. The goal of filtering is to reduce the number of ordering choices and to make the overall process of finding a solution faster. For this reason filtering algorithms should run fast and find as many restrictions as possible. On the other hand each restriction must be *justified*, i. e. there must not exist any solution with the removed values. In some cases filtering can find so many restrictions that the domain of a variable becomes empty, i. e. no possible schedule remains. This is called an *inconsistency* and causes backtracking in the search for possible orderings.

This work deals with **optional activities** on unary resources. These are tasks for which it is not yet known if they should be included in the final schedule:

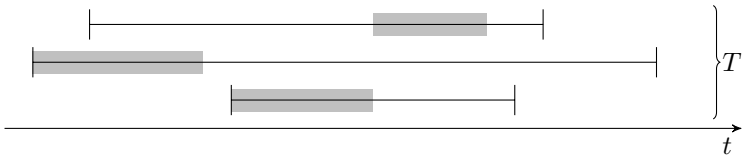


Fig. 2. A valid schedule for the unary resource with the tasks in T

E. g. when one specific task has to be performed once but several resources could do it. In this case optional activities are added to each resource, with the additional constraint that exactly one of them should be included in the final schedule.

Optional activities are modelled by adding another attribute: The variable status_i can take the values 1 for enabled and 0 for disabled. The domain $\{0, 1\}$ thus indicates an optional task. Additionally, let T_{enabled}, T_{optional} and T_{disabled} denote the partition of T into the enabled, optional and disabled tasks.

Optional activities entail additional difficulties for constraint filtering algorithms: As optional activities might become disabled later, they may not influence any other activity, because the resulting restrictions would not be justified. However, it is possible to detect if the inclusion of an optional activity causes an overload on an otherwise non-overloaded unary resource. In this case it is possible to disable this optional task. Doing this as often as possible while still maintaining fast filtering times is a desirable way to speed up the overall search process, because fewer ordering choices have to be enumerated later.

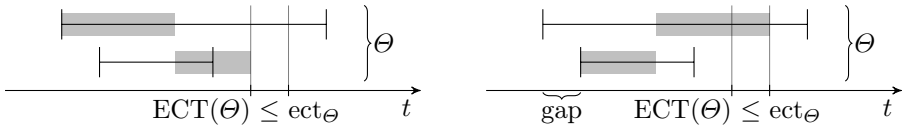
Several standard filtering algorithms for unary resources have been extended for optional activities by Vilím, Barták and Čepek [7]. To the best of the author's knowledge, no edge-finding algorithm has been proposed so far that considers optional activities. This paper aims to fill this gap.

2 Edge-Finding

There are two variants of edge-finding: One restricts the earliest starting times, the other restricts the latest completion times of the tasks on a unary resource. This paper only presents the former one, as the latter is symmetric to it.

To state the edge-finding rule and the algorithm, the following notational abbreviations are needed. Given a set Θ of n tasks, $\text{ECT}(\Theta)$ is a lower bound of the earliest completion time of all tasks in Θ:

$$\text{ECT}(\Theta) := \max_{\Theta' \subseteq \Theta} \{\text{est}_{\Theta'} + \text{p}_{\Theta'}\} \tag{1}$$



(a) $\text{ECT}(\Theta)$ assumes this invalid schedule is possible

(b) The best possible schedule includes a gap, resulting in $\text{ect}_\Theta > \text{ECT}(\Theta)$

Fig. 3. $\text{ECT}(\Theta)$ can underestimate the real earliest completion time ect_Θ

The definition of $\mathrm{ECT}(\Theta)$ ignores gaps that may be necessary for a valid schedule, as illustrated in Fig. 3. This simplification results in weaker domain restrictions, but makes the computation of $\mathrm{ECT}(\Theta)$ feasible in the first place.[2]

Vilím has shown how the value $\mathrm{ECT}(\Theta)$ can be computed using a so-called Θ-tree [6], that has $\mathcal{O}(\log n)$ overhead for adding or removing tasks from Θ and offers constant time access to this value in return.

Later, extended this data structure was extended by Vilím, Barták and Čepek to Θ-Λ-trees. They retain the time complexity and include up to one task from an additional set Λ (disjoint from Θ), such that $\mathrm{ECT}(\Theta, \Lambda)$, the lower bound of the earliest completion time, becomes maximal [7]:

$$\mathrm{ECT}(\Theta, \Lambda) := \max_{j \in \Lambda} \{\mathrm{ECT}(\Theta \cup \{j\})\} \overset{(1)}{=} \max_{j \in \Lambda} \left\{ \max_{\Theta' \subseteq \Theta \cup \{j\}} \{\mathrm{est}_{\Theta'} + \mathrm{p}_{\Theta'}\} \right\} \quad (2)$$

The purpose of this extension is to choose, for a given set Θ, one activity $j \in \Lambda$ such that $\mathrm{ECT}(\Theta \cup \{j\})$ becomes maximal. This is needed to efficiently find all applications of the edge-finding rule given below.

To handle optional activities, another extension is needed: The maximum (lower bound of the) earliest completion time for the tasks in Θ plus exactly one task from Ω plus up to one task from Λ (where Θ, Ω and Λ are pairwise disjoint):

$$\mathrm{ECT}(\Theta, \Omega, \Lambda) := \max_{o \in \Omega} \left\{ \max_{j \in \Lambda} \left\{ \max_{\Theta' \subseteq \Theta \cup \{j\}} \left\{ \mathrm{est}_{\Theta' \cup \{o\}} + \mathrm{p}_{\Theta' \cup \{o\}} \right\} \right\} \right\} \quad (3)$$

As the algorithm presented below calculates the third argument to $\mathrm{ECT}(\Theta, \Omega, \Lambda)$ dynamically,[3] this value cannot be pre-computed using a tree structure. Instead, a loop over all tasks is needed, yielding $\mathcal{O}(n)$ time complexity. This loop iterates over all tasks $j \in T$, pre-sorted by their earliest starting time est_j, and keeps track of the earliest completion time of the tasks processed so far. To achieve this, a fact about partitions of task sets is used:

Definition 1 (Partition at an earliest starting time). *Given a set T of tasks, a pair of sets (L, R) is called* partition of T at the earliest starting time t_0, *iff (L, R) is a partition of T (i. e. $L \cup R = T$, $L \cap R = \emptyset$) that fulfils the following two conditions: $\forall l \in L : \mathrm{est}_l \leq t_0$ and $\forall r \in R : \mathrm{est}_r \geq t_0$.*

Proposition 1 (Calculation of ECT[4]). *Given a set T of tasks, and a partition (L, R) of T at an arbitrary earliest starting time, the following holds:*

$$\mathrm{ECT}(T) = \max \left\{ \mathrm{ECT}(R), \mathrm{ECT}(L) + \sum_{j \in R} \mathrm{p}_j \right\}$$

[2] If the exact earliest completion time ect_Θ of Θ could be calculated in polynomial time, the NP-complete problem "is there a valid schedule for this set of tasks" could be decided in polynomial time as well.

[3] The membership in Λ is not stored in memory, but decided by evaluating a condition. This also applies to one occurrence of $\mathrm{ECT}(\Theta, \Lambda)$.

[4] This is a variant of a proposition proven by Vilím, Barták and Čepek [7, p. 405] that was originally stated for left and right subtrees in a Θ-tree. The proof given there directly carries over to this proposition.

When iterating over the tasks $j \in T$, let L be the set of tasks considered so far (thus $\mathrm{ECT}(L)$ is known[5]) and $R := \{j\}$ the set containing only the current activity (for which $\mathrm{ECT}(R) = \mathrm{ect}_j$ is also known). Furthermore let $L' := L \cup \{j\}$ denote the set of activities considered after the current iteration. This way L grows from the empty set until it includes all activities in T. To compute not only $\mathrm{ECT}(\Theta)$ but $\mathrm{ECT}(\Theta, \Omega, \Lambda)$ on the basis of Proposition 1, the following values must be updated in the loop for each task j that is considered:

1. (The lower bound of) the earliest completion time of the Θ-activities only, i. e. $\mathrm{ECT}(L' \cap \Theta)$:

$$ect_{L'} := \begin{cases} \max\{ect_L + \mathrm{p}_j, \mathrm{ect}_j\} & \text{if } j \in \Theta \\ ect_L & \text{otherwise} \end{cases}$$

2. (The lower bound of) the earliest completion time when including up to one Λ-activity, i. e. $\mathrm{ECT}(L' \cap \Theta, L' \cap \Lambda)$:

$$ectl_{L'} := \begin{cases} \max\{ectl_L + \mathrm{p}_j, \mathrm{ect}_j\} & \text{if } j \in \Theta \\ \max\{ectl_L, \mathrm{ect}_j, ect_L + \mathrm{p}_j\} & \text{if } j \in \Lambda \\ ectl & \text{otherwise} \end{cases}$$

3. (The lower bound of) the earliest completion time when including exactly one Ω-activity (provided there is one), i. e. $\mathrm{ECT}(L' \cap \Theta, L' \cap \Omega, \emptyset)$:

$$ecto_{L'} := \begin{cases} \max\{ecto_L + \mathrm{p}_j, \mathrm{ect}_j\} & \text{if } j \in \Theta \wedge \Omega \cap L = \emptyset \\ ecto_L + \mathrm{p}_j & \text{if } j \in \Theta \wedge \Omega \cap L \neq \emptyset \\ \max\{\mathrm{ect}_j, ect_L + \mathrm{p}_j\} & \text{if } j \in \Omega \wedge \Omega \cap L = \emptyset \\ \max\{\mathrm{ect}_j, ect_L + \mathrm{p}_j, ecto_L\} & \text{if } j \in \Omega \wedge \Omega \cap L \neq \emptyset \\ ecto_L & \text{otherwise} \end{cases}$$

4. (The lower bound of) the earliest completion time when including up to one Λ- and exactly one Ω-activity (provided there is one), i. e. $\mathrm{ECT}(L' \cap \Theta, L' \cap \Omega, L' \cap \Lambda)$:

$$ectol_{L'} := \begin{cases} \max\{ectol_L + \mathrm{p}_j, \mathrm{ect}_j\} & \text{if } j \in \Theta \wedge \Omega \cap L = \emptyset \\ ectol_L + \mathrm{p}_j & \text{if } j \in \Theta \wedge \Omega \cap L \neq \emptyset \\ \max\{\mathrm{ect}_j, ect_L + \mathrm{p}_j, ectl_L + \mathrm{p}_j\} & \text{if } j \in \Omega \wedge \Omega \cap L = \emptyset \\ \max\{\mathrm{ect}_j, ect_L + \mathrm{p}_j, ectl_L + \mathrm{p}_j, ectol_L\} & \text{if } j \in \Omega \wedge \Omega \cap L \neq \emptyset \\ \max\{ectol_L, ecto_L + \mathrm{p}_j, \mathrm{ect}_j\} & \text{if } j \in \Lambda \wedge \Omega \cap L = \emptyset \\ \max\{ectol_L, ecto_L + \mathrm{p}_j\} & \text{if } j \in \Lambda \wedge \Omega \cap L \neq \emptyset \\ ectol_L & \text{otherwise} \end{cases}$$

[5] For $L = \emptyset$ define $\mathrm{ECT}(\emptyset) := -\infty$

After the last iteration (i. e. $L' = T$) $ectol_{L'} = \mathrm{ECT}(L' \cap \Theta, L' \cap \Lambda, L' \cap \Omega) = \mathrm{ECT}(\Theta, \Omega, \Lambda)$ contains the result of the computation.

Note that the indices of ect_L, $ectl_L$, $ectl_L$ and $ectol_L$ are only added for conceptual clarity. If the values are updated in the right order (i. e. $ectol_L$ first and ect_L last), only plain variables *ect*, *ectl*, *ecto* and *ectol* (and no arrays) are needed.

Finally, one more notational abbreviation is needed to state the edge-finding rule: The set of all tasks in a set T (usually all tasks or all enabled tasks of a resource), that end not later than a given task j:

$$\Theta(j, T) := \{k \in T \mid \mathrm{lct}_k \leq \mathrm{lct}_j\} \tag{4}$$

Notice the overloading of Θ; the return value of the function Θ corresponds to value of the set Θ at the time when j is processed in the main loop of the algorithm introduced later.

Rule 1 (Edge-Finding) *For all tasks $j \in T$ and $i \in T \setminus \Theta(j, T)$ holds: If*

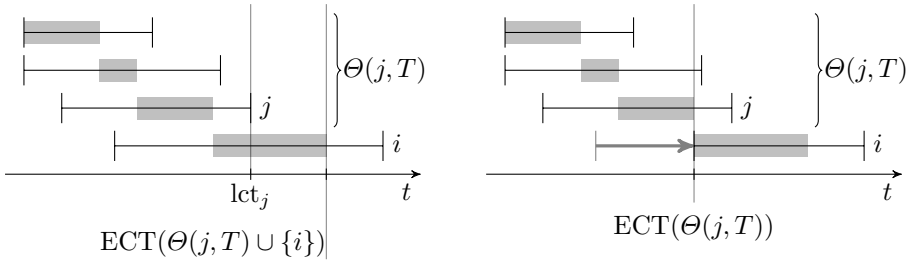
$$\mathrm{ECT}(\Theta(j, T) \cup \{i\}) > \mathrm{lct}_j \tag{5}$$

then i must be scheduled after all activities in $\Theta(j, T)$, i. e. the following restriction is justified:

$$\mathrm{est}_i \leftarrow \max\{\mathrm{est}_i, \mathrm{ECT}(\Theta(j, T))\} \tag{6}$$

This form of the edge-finding rule was introduced by Vilím, Barták and Čepek [7, p. 412ff] and proven equivalent to the more traditional form. The idea behind this rule is to detect if an activity i has to be the last one within the set $\Theta(j, T) \cup \{i\}$. This is ensured by the precondition (5) of the rule, which is illustrated in Fig. 4(a): As the latest completion time lct_j is by (4) also the latest completion time of $\Theta(j, T)$, the precondition states that $\Theta(j, T)$ must be finished before $\Theta(j, T) \cup \{i\}$ can be finished.

The resulting restriction (6), which is illustrated in Fig. 4(b), ensures that i is not started until all activities in $\Theta(j, T)$ can be finished.



(a) The rule is applicable for j and i: $\mathrm{ECT}(\Theta(j, T) \cup \{i\}) > \mathrm{lct}_j$

(b) After the application est_i is at least $\mathrm{ECT}(\Theta(j, T))$

Fig. 4. An application of the edge-finding rule

Besides the edge-finding rule, the presented edge-finding algorithm makes use of the following overload rule, that can be checked along the way without much overhead:

Rule 2 (Overload) *For all $j \in T$ holds: If $\mathrm{ECT}(\Theta(j,T)) > \mathrm{lct}_j$ then an overload has occurred, i. e. it is not possible to schedule all activities in T without conflict.*

This rule is illustrated in Fig. 5. The intuition for the overload rule is that the tasks in $\Theta(j,T)$ cannot possibly be finished before they have to.

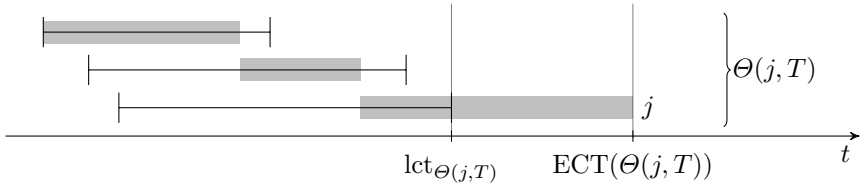


Fig. 5. An application of the overload rule

2.1 Edge-Finding Algorithm

Now to the central algorithm of this article: Listing 1 shows a program that finds all applications of the edge-finding rule to the set of enabled tasks and furthermore disables optional tasks whose inclusion would cause an overload (i. e. est_i could be raised to a value greater than lst_i for some task i).

Throughout the repeat-loop the set Θ is updated to reflect $\Theta(j, T_{\mathrm{enabled}})$ and Ω is updated to reflect $\Theta(j, T_{\mathrm{optional}})$ – exceptions are only allowed if multiple tasks have the same lct value. As j iterates over Q, lct_j decreases and j is removed from Θ or Ω and added to Λ.

Lines 15 to 25 handle enabled activities j and correspond closely to the algorithm by Vilím, Barták and Čepek [7, p. 416], with the only change being the handling of $\mathrm{status}_i = \mathrm{optional}$ in lines 20 to 24: If $\mathrm{ECT}(\Theta) > \mathrm{lst}_i$ (line 21) holds, the restriction $\mathrm{est}'_i \leftarrow \mathrm{ECT}(\Theta)$ (line 24) would cause an inconsistency as no possible start time for i remains. In this case the activity i is set to disabled (line 22), which fails for enabled activities and is the proper restriction for optional ones.

There are several more additions to handle optional activities: Firstly, the case $\mathrm{status}_j = \mathrm{optional}$ is handled. It follows the scheme of the enabled case, bearing in mind that j is optional: The overload-condition $\mathrm{ECT}(\Theta) > \mathrm{lct}_j$ on line 15 carries over to $\mathrm{ECT}(\Theta \cup \{j\}) > lct_j$ on line 39, where no immediate failure is generated but the optional activity j which would cause the overload is disabled.

If j is optional, the condition $\mathrm{ECT}(\Theta, \Lambda) > lct_j$ of the while-loop on line 18 can only result in the disabling of j as no optional activity may influence others. For this, no while-loop is required and a simple if-condition suffices. It must however take care to choose an appropriate $i \in \Lambda$ that leads to j being disabled. This is achieved by requiring $i \in T_{\mathrm{enabled}}$ and $\mathrm{lst}_i < \mathrm{ECT}(\Theta \cup \{j\})$ in the condition on line 40.

Listing 1. Edge-finding algorithm.

```
1  input: activities T on a unary resource
2         // read-write access to est_i, lst_i, lct_i, status_i is assumed for all i ∈ T
3  for i ∈ T do // cache changes to est_i: changing it directly would mess up Θ-Λ-trees
4      est'_i ← est_i
5
6  (Θ, Ω, Λ) ← (T_enabled, T_optional, ∅) // initialise tree(s)
7  Q ← queue of all non-disabled j ∈ T \ T_disabled in descending order of lct_j
8  j ← Q.first // go to the first task in Q
9  repeat
10     if status_j ≠ disabled then // move j to Λ
11         (Θ, Ω, Λ) ← (Θ \ {j}, Ω \ {j}, Λ ∪ {j})
12     Q.dequeue; j ← Q.first // go to the next task in Q
13
14     if status_j = enabled then
15         if ECT(Θ) > lct_j then
16             fail // because overload rule (Rule 2) applies
17
18         while ECT(Θ, Λ) > lct_j do
19             i ← the Λ-activity responsible for ECT(Θ, Λ)
20             if ECT(Θ) > est_i then // edge-finding rule is applicable
21                 if ECT(Θ) > lst_i then // inconsistency detected
22                     status_i ← disabled // fails if i is enabled
23                 else // apply edge-finding rule
24                     est'_i ← ECT(Θ) // ECT(Θ) > est_i already ensured
25             Λ ← Λ \ {i} // no better restriction for i possible [7, p. 416]
26
27         while ECT(Θ, Ω) > lct_j do // overload rule applies
28             o ← the Ω-activity responsible for ECT(Θ, Ω)
29             status_o ← disabled
30             Ω ← Ω \ {o}
31
32         while Ω ≠ ∅ and ECT(Θ, Ω, Λ'(o)) > lct_j do // Λ'(o) is defined in (7)
33             // edge-finding rule detects overload
34             o ← the Ω-activity responsible for ECT(Θ, Ω, ...)
35                                 // already used in line 32 with that meaning
36             status_o ← disabled
37             Ω ← Ω \ {o}
38     else if status_j = optional then
39         if ECT(Θ ∪ {j}) > lct_j // overload rule applicable ...
40         or ECT(Θ ∪ {j}, {i ∈ T_enabled \ Θ | lst_i < ECT(Θ ∪ {j})}) > lct_j
41         then // ... or edge-finding rule detects overload
42             status_j ← disabled
43             Ω ← Ω \ {j} // no more restrictions for j possible
44 until the end of Q is reached
45
46 for i ∈ T do
47     est_i ← est'_i // apply cached changes
```

Secondly, the case $\text{status}_j = \text{enabled}$ is extended with two more while-loops. They realise overload-detection and edge-finding for optional activities that are still contained in Ω. The set $\Lambda'(o)$, which is used in the condition of the second while-loop (line 32), is defined as follows (with T_{enabled} and Θ taken from the context where $\Lambda'(o)$ is evaluated):

$$\Lambda'(o) := \Big\{ i \in T_{\text{enabled}} \setminus \Theta \;\Big|\; \text{lst}_i < \text{ECT}(\Theta \cup \{o\}) \\ \wedge\, \nexists\, i_2 \in T_{\text{enabled}} \setminus \Theta : \Big(\text{lst}_{i_2} \geq \text{ECT}(\Theta \cup \{o\}) \\ \wedge\, \text{ECT}(\Theta \cup \{i_2\}) > \text{ECT}(\Theta \cup \{i\}) \Big) \Big\} \quad (7)$$

Like for the algorithm by Vilím, Barták and Čepek, the algorithm in Listing 1 must be repeated until it finds no further restrictions to make it idempotent. This issue is addressed by Listing 2 in Sect. 3 below.

Proposition 2 (Correctness). *The algorithm presented in Listing 1 is correct, i. e. all restrictions are justified.*

Proof. All restrictions are applications of the edge-finding and overload rules (Rules 1 and 2). It remains to be shown that the preconditions of the rules are met each time they are applied. This directly follows from the way the algorithm updates its data structures. As a simple example, consider the application of the overload rule in line 16. Remember that $\Theta = \Theta(j, T_{\text{enabled}})$. With the conditions in the preceding lines $j \in T_{\text{enabled}}$ and the precondition $\text{ECT}(\Theta(j, T_{\text{enabled}})) > \text{lct}_j$ is fulfilled.

For a more complicated case consider the disabling of the task o in line 36: This restriction is based on the edge-finding rule and that o is an optional activity. Remember that $\Theta = \Theta(j, T_{\text{enabled}})$ and $o \in \Omega = \Theta(j, T_{\text{optional}})$. Define $T' := T_{\text{enabled}} \cup \{o\}$. The activity i chosen from $\Lambda'(o)$ satisfies $i \in T_{\text{enabled}} \setminus \Theta$ by (7). This implies $i \in T' \setminus \Theta$. It holds that

$$\text{ECT}(\Theta, \Omega, \Lambda'(o)) = \text{ECT}(\Theta(j, T') \cup \{i\}) \; .$$

Together with the condition on the enclosing while loop this means that the precondition of the edge-finding rule is satisfied. From (7) also follows that $\text{lst}_i < \text{ECT}(\Theta \cup \{o\}) = \text{ECT}(\Theta(j, T'))$. This implies that the restriction of the edge-finding rule, if carried out, would lead to i having no possible start times left, i. e. an inconsistency would occur. As o is the only optional activity involved in this inconsistency, o cannot be included in a correct schedule and disabling it is correct.

The proofs for the other restrictions are similar. □

2.2 Complexity

Proposition 3 (Time Complexity). *The time complexity of the edge-finding algorithm presented in Listing 1 is $\mathcal{O}(n^2)$, where n is the number of activities on the unary resource.*

Proof. To see this, observe that each loop is executed at most n times: The two small copy-loops in lines 3–4 and 46–47 directly loop over all tasks, the main loop in lines 14–44 loops over the subset of non-disabled tasks and the three inner loops in lines 18–25, 27–30 and 32–37 each remove a task from Λ or Ω. As each task is added to these sets at most once, each loop is executed at most n times.

Sorting the tasks by lct (line 7) can be done in $\mathcal{O}(n \log n)$. All other lines (this includes all lines within the loops) bear at most $\mathcal{O}(n)$ complexity. Most are $\mathcal{O}(1)$, the more complicated ones are $\mathcal{O}(\log n)$ or $\mathcal{O}(n)$, as discussed for (1) to (3) at the beginning of Sect. 2.

Only the calculation of $\mathrm{ECT}(\Theta, \Omega, \Lambda'(o))$ in line 32 needs special consideration, as o already references the task chosen from Ω. The definition of $\Lambda'(o)$ in (7) makes it possible to calculate it in the following way:

- When calculating $ectl_{L'}$ take *all* activities from $T_{\text{enabled}} \setminus \Theta$ in account, as it is not yet known which o will be chosen.
- For the cases with $j \in \Omega$ during the calculation of $ectol_{L'}$, consider $ectl_L + \mathrm{p}_j$ for calculating the maximum only if the activity i chosen for $ectl_L$ satisfies $\mathrm{lst}_i < \mathrm{ECT}(\Theta \cup \{j\})$.
- For the cases with $j \in \Lambda$ during the calculation of $ectol_{L'}$, consider $ecto_L + \mathrm{p}_j$ for the maximum only if the activity o chosen for $ecto_L$ satisfies $\mathrm{lst}_j < \mathrm{ECT}(\Theta \cup \{o\})$.

To be able to access $\mathrm{ECT}(\Theta \cup \{o\})$ in constant time for all $o \in \Omega$, three arrays $ectAfter_o$, $pAfter_o$ and $ectBefore_o$ can be used, where after/before refers to the position of o in the list of tasks sorted by est_j and the three values consider only tasks from Θ. They can be computed based on Proposition 1 in linear time by looping over the tasks pre-sorted by est_j. It holds:

$$\mathrm{ECT}(\Theta \cup \{o\}) = \max\{ectAfter_o, \mathrm{ect}_o + pAfter_o, ectBefore_o + \mathrm{p}_o + pAfter_o\}$$

Thus the overall time complexity of the algorithm is $\mathcal{O}(n^2)$. □

Proposition 4 (Space Complexity). *The space complexity of the edge-finding algorithm presented in Listing 1 is $\mathcal{O}(n)$, where n is the number of activities on the unary resource.*

Proof. The algorithm uses only tree and list structures. Both have linear memory requirements. □

2.3 Optimality

For efficient restriction of the domains it is necessary that as many applications of the edge-finding rule as possible are found by the algorithm, while it is equally crucial that it offers fast runtimes.

Proposition 5 (Optimality). *The algorithm presented in Listing 1 finds all applications of the edge-finding rule to enabled activities and disables almost all optional activities that would cause an overload that is detectable by the edge-finding rule.*

Proof. First consider all applications of the edge-finding rule that involve only enabled tasks. Let therefore $j \in T_{\text{enabled}}$ and $i \in T_{\text{enabled}} \setminus \Theta(j, T_{\text{enabled}})$, such that $\text{ECT}(\Theta(j, T_{\text{enabled}}) \cup \{i\}) < \text{lct}_j$. As argued above, the set Θ at some point takes the value of $\Theta(j, T_{\text{enabled}})$ and i is contained in Λ. Then in line 20 i will be chosen (possibly after other i' that feature a larger $\text{ECT}(\Theta \cup \{i'\})$ have been chosen and removed from Λ) and est_i will be adjusted or an overload detected.

If an optional activity $o \in T_{\text{optional}}$ is involved, things are a bit more complicated. If the edge-finding rule is applicable for the set $T' := T_{\text{enabled}} \cup \{o\}$, the following cases can occur:

Case 1: $o \notin \Theta(j, T') \cup \{i\}$
The optional activity o is not involved and the edge-finding rule is equally applicable to the set T_{enabled}. The resulting restriction is found by the algorithm as argued above.

Case 2: $o = i$
This case is handled together with non-optional i in lines 18 to 25.

Case 3: $o = j$
In this case est_i may not be adjusted as the optional activity o may not influence the non-optional i. However, if est_i can be adjusted to a value greater than lst_i, the inclusion of o would cause an overload and o can be disabled. This is done in lines 38 to 43.

Case 4: $o \in \Theta(j, T') \setminus \{j\}$
Like in the previous case the only possible restriction is disabling o if it causes an overload. This case is handled in lines 32 to 37. For optimal results $\Lambda'(o)$ would have to be defined slightly differently, omitting the condition

$$\nexists\, i_2 \in T_{\text{enabled}} \setminus \Theta : \Big(\text{lst}_{i_2} \geq \text{ECT}(\Theta \cup \{o\}) \wedge \text{ECT}(\Theta \cup \{i_2\}) > \text{ECT}(\Theta \cup \{i\}) \Big).$$

However, this would destroy the $\mathcal{O}(n^2)$ complexity of the algorithm. As this case never occurs for any of the problem instances measured in the next section, omitting these restrictions appears to be a good choice. □

3 Comparison with Other Edge-Finding Algorithms

The presented algorithm has been implemented for the Java constraint programming library `firstcs` [5] and compared to other algorithms. The following algorithms were considered:

New algorithm: The algorithm presented in Listing 1. It finds almost all possible applications of the edge-finding rule in $\mathcal{O}(n^2)$ time.

Simplified algorithm: Similar to the previous algorithm but without the loop in lines 32 to 37. This saves one of the two most expensive computations, though the asymptotic time complexity is still $\mathcal{O}(n^2)$. Fewer restrictions are found.

Active-only algorithm: The algorithm presented by Vilím, Barták and Čepek [7, p. 416]. It runs at $\mathcal{O}(n \log n)$ time complexity but takes no optional tasks into account.[6]

Iterative algorithm: Run the active-only algorithm once on T_{enabled} and then for each $o \in T_{\text{optional}}$ on $T_{\text{enabled}} \cup \{o\}$, adjusting or disabling o only. This algorithm finds all restrictions to optional activities at the cost of $\mathcal{O}(n^2 \log n)$ runtime.

To make these (and the other filtering algorithms, taken from [7]) idempotent, the fix-point technique of Listing 2 was used.

Listing 2. Fix-point iteration

```
1  repeat
2      repeat
3          repeat
6              filter based on detectable precedences
7          until no more restrictions are found
8          filter with not-first/not-last rule
9      until no more restrictions are found
10     filter with the edge-finding algorithm to be benchmarked
11 until no more restrictions are found
```

For some benchmarks mentioned later, additional overload checking was performed by adding the following lines at the beginning of the innermost repeat-loop:

```
4              if overload detected then
5                  fail
```

For now, this additional overload checking is skipped as all the studied edge-finding algorithms include applications of the overload rule. This way the comparison between the algorithms is more meaningful, as not only the differences in the way the edge-finding rule is applied, but also those in the way the overload rule is applied contribute to the results.

The search was performed using a modified resource-labelling technique. If optional tasks are allowed, the value of enabled_i has to be decided during the labelling as well. Usually it is decided at each level in the search tree in which order the corresponding pair of tasks $i, j \in T$ is to be executed. These two decisions were combined, so that the following alternatives are considered at each level: (a) both enabled with i before j, (b) both enabled with i after j, (c) only i enabled, (d) only j enabled or (e) both disabled.

[6] Although this algorithm is presented in an article mentioning optional activities in its title, it does not derive any restrictions from optional tasks. The reason it is listed in that paper is probably that it shares the data structure used, namely Θ-Λ-trees, with the other algorithms introduced there.

3.1 The Job-Shop Scheduling Problem

For benchmarking, *-alt* variants [7, p. 423] of job-shop problems [3] were used: For each job, exactly one of the fifth and sixth task must be included for a solution. All instances mentioned in this paper have 10 jobs each consisting of 10 tasks. This results in 10 machines with 10 tasks each. For the *-alt* instances 2 tasks per machine are initially optional on average. The last three instances contain no optional tasks and are included for reference. All times are measured in milliseconds and include finding a solution for the known optimal make-span and proving that this make-span is indeed optimal.

Table 1 shows the results. The number of backtracking steps required is equal for the new, the simplified and the iterative algorithm. So the new algorithm looses none of the possible restrictions, even if the loop in lines 32 to 37 is left out. When comparing the runtimes, the overhead of this loop is clearly visible: The simplified variant saves time, as it has lower overhead.

The active-only algorithm needs more backtracking steps because it finds less restrictions. Often the lower complexity compensates for this when it comes to the runtimes and the active-only variant is slightly better than the simplified one. The largest relative deviations is the other way around, though: For orb02-alt the active-only algorithm is almost 30% slower than the simplified one, for la19-alt it even needs 90% more time. These are also the instances with the largest difference in backtracking steps. It obviously depends on the inner structure of the job-shop instances (or the problems in general), if the higher time complexity of the newly proposed algorithms is justified by the backtracking steps saved by the additionally found restrictions.

Table 1. Runtime in milliseconds and number of backtracks for the different algorithms and job-shop-scheduling instances

	new		**simplified**		**active-only**		**iterative**	
instance	**time**	**bt**	**time**	**bt**	**time**	**bt**	**time**	**bt**
abz5-alt	4843	5859	4484	5859	4515	6441	4964	5859
orb01-alt	55747	56344	53662	56344	49361	56964	57013	56344
orb02-alt	7800	7265	7394	7265	9590	10610	7609	7265
orb07-alt	81856	99471	79063	99471	78309	104201	79786	99471
orb10-alt	136	85	125	85	121	85	132	85
la16-alt	7269	8294	6886	8294	6593	8841	7241	8294
la17-alt	46	9	31	9	35	11	31	9
la18-alt	26780	26846	25147	26846	24877	29039	25897	26846
la19-alt	2566	2022	2406	2022	4609	4632	2574	2022
la20-alt	62	53	63	53	55	53	62	53
abz5	5863	5107	5863	5107	5587	5107	5570	5107
orb01	29612	21569	29706	21569	28198	21569	28336	21569
orb02	10144	7937	10187	7937	9644	7937	9687	7937

The runtime of the iterative algorithm is mostly better than the new, but worse than the simplified algorithm.[7] As the tested instances all have relatively few optional activities per machine[8] (which benefits the active-only and iterative algorithms), it is interesting how the new algorithm performs for problems with more activities and a higher proportion of optional ones.

3.2 The Random Placement Problem

One such problem is the random placement problem [2], which can be solved using alternative resource constraints which in turn can be implemented using single resources with optional activities [8]. The runtimes of the instances provided on `http://www.fi.muni.cz/~hanka/rpp/` that are solvable in less than ten minutes are shown in Fig. 6.[9]

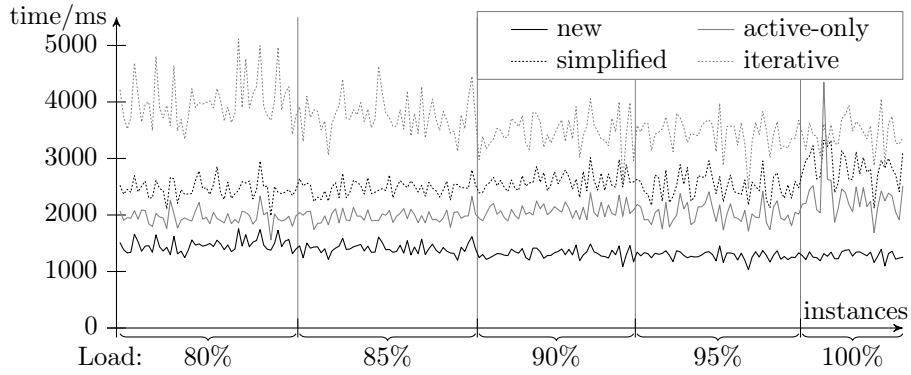


Fig. 6. Runtimes for instances of the Random-Placement-Problem, without additional overload checking

It is obvious that the new algorithm is the fastest in this case. Surprisingly the simplified algorithm is worse than the active-only one: Obviously it finds no or not enough additional restrictions. The power of the new algorithm thus derives from the loop omitted in the simplified one.[10] The iterative algorithm is the slowest, though it finds as many restrictions as the new one. The reason is the higher asymptotic time complexity, which seems to be prohibitive even

[7] An exception are the three last instances, which include no optional tasks.

[8] As mentioned above, the machines each have 10 tasks, 20% are optional on average.

[9] All algorithms need 0 backtracking steps, with the exception of the active-only algorithm for one instance with 100% load. This instance is responsible for the one peak going above even the iterative algorithm.

[10] This is confirmed by the number of choices needed during search: For the iterative and new algorithms it is always equal (the new one again looses no restrictions) and averagely 407, but the active-only and simplified ones both need more than 620 on average.

for relatively small resources with 50 to 100 tasks which appear in the tested random-placement instances.

So far only the fix-point algorithm without additional overload checking has been considered. This has its justification in the fact that all measured edge-finding algorithms perform overload-checking during their regular operation: By omitting the additional overload check their filtering strength contributes more to the results of the benchmark. In case of the job-shop benchmark, the additional overload checking furthermore results in slower runtimes for all algorithms. However, in case of the random placement benchmark, additional overload checking can lead to performance improvements for some edge-finding algorithms. This can be seen in Fig. 7: While the new and iterative algorithms loose 75 and 30 milliseconds, the active-only and simplified algorithms gain 990 and 1185 milliseconds averaged over the instances, overtaking the new algorithm. The additional overload checking thus can compensate for weaker performance of an edge-finding algorithm in this case.

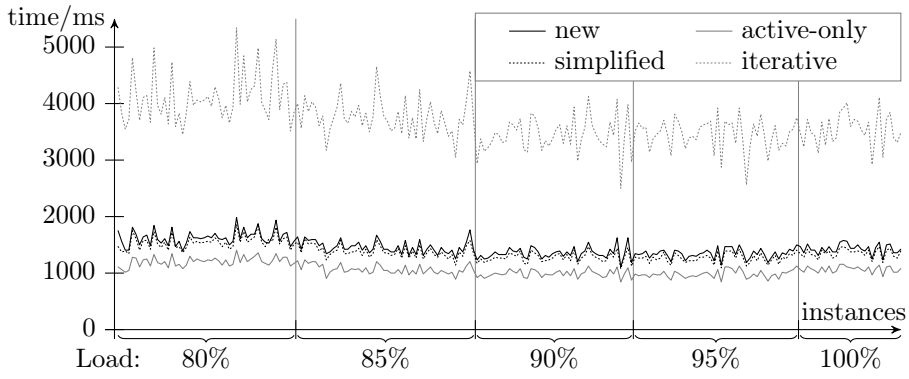


Fig. 7. Runtimes for instances of the Random-Placement-Problem, with additional overload checking

4 Summary

This paper introduces a new edge-finding algorithm for unary resources. It deals with optional activities correctly, finds all applications of the edge-finding rule to the enabled activities and disables optional activities if they cause an overload detected by the edge-finding rule. It is the first nontrivial edge-finding algorithm that derives restrictions for optional activities. It outperforms the naive, iterative implementation as soon as a larger fraction of optional activities is involved.

With $\mathcal{O}(n^2)$ asymptotic time complexity it is slower than edge-finding algorithms that cannot take optional activities into account. The additionally found restrictions offer a significant reduction of the search decisions needed to find and prove the solution of a problem. Especially for problems with many activities, of which a high proportion is optional, this can lead to faster runtimes. Performing additional overload checking can cancel this effect out, though.

Future work can study the influence of different labelling strategies when it comes to optional activities: Is it better to first establish an order of tasks and then decide which tasks should be enabled? Another possible direction for future work is studying which edge-finding algorithm yields best results depending on the problem structure, possibly implementing heuristics to decide which algorithm to use.

References

1. Philippe Baptiste, Claude Le Pape, and Wim Nuijten. 'Constraint-Based Scheduling – Applying Constraint Programming to Scheduling Problems'. Boston: Kluwer Academic Publishers, 2001. ISBN 0792374088.
2. Roman Barták, Tomáš Müller and Hana Rudová. 'A New Approach to Modeling and Solving Minimal Perturbation Problems'. In: Recent Advances in Constraints, CSCLP 2003. LNCS 3010. Berlin: Springer, 2004. ISBN 978-3-540-21834-0. Pp. 233–249.
3. Yves Colombani. 'Constraint programming: an efficient and practical approach to solving the job-shop problem'. In: Eugene C. Freuder, editor, *Principles and Practice of Constraint Programming – CP96*. LNCS 1118. Berlin: Springer, 1996. ISBN 3-540-61551-2. Pp. 149–163.
4. Michael R. Garey and David S. Johnson. 'Computers and Intractability – A Guide to the Theory of NP-Completeness'. New York: W. H. Freeman, 1979. ISBN 0-7167-1045-5.
5. Matthias Hoche, Henry Müller, Hans Schlenker and Armin Wolf. '`firstcs` – A Pure Java Constraint Programming Engine'. In: Michael Hanus, Petra Hofstedt and Armin Wolf, editors, *2nd International Workshop on Multiparadigm Constraint Programming Languages – MultiCPL'03*, 29th September 2003. URL: `http://uebb.cs.tu-berlin.de/MultiCPL03/Proceedings.MultiCPL03.RCoRP03.pdf`.
6. Petr Vilím. '$\mathcal{O}(n \log n)$ Filtering Algorithms for Unary Resource Constraint'. In: Jean-Charles and Régin Michel Rueher, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. LNCS 3011. Berlin: Springer, 2004. ISBN 978-3-540-21836-4. Pp. 335–347. URL: `http://kti.mff.cuni.cz/~vilim/nlogn.pdf`
7. Petr Vilím, Roman Barták and Ondřej Čepek. 'Extension of $\mathcal{O}(n \log n)$ Filtering Algorithms for the Unary Resource Constraint to Optional Activities'. In: *Constraints* 10.4 (2005). Pp. 403–425. URL: `http://kti.mff.cuni.cz/~vilim/constraints2005.pdf`
8. Armin Wolf, and Hans Schlenker. 'Realising the Alternative Resources Constraint'. In: Applications of Declarative Programming and Knowledge Management, INAP 2004. LNCS 3392. Berlin: Springer, 2005. ISBN 978-3-540-25560-4. Pp. 185–199.