Constraints (2014) 19:243–269
DOI 10.1007/s10601-013-9157-z

A quadratic edge-finding filtering algorithm for cumulative resource constraints

Roger Kameugne · Laure Pauline Fotso · Joseph Scott · Youcheu Ngo-Kateu

Published online: 22 November 2013

Abstract The cumulative scheduling constraint, which enforces the sharing of a finite resource by several tasks, is widely used in constraint-based scheduling applications. Propagation of the cumulative constraint can be performed by several different filtering algorithms, often used in combination. One of the most important and successful of these filtering algorithms is edge-finding. Recent work by Vilím has resulted in a $\mathcal{O}(kn \log n)$ algorithm for cumulative edge-finding (where n is the number of tasks and k is the number of distinct capacity requirements), as well as a new related filter, timetable edge-finding, with a complexity of $\mathcal{O}(n^2)$. We present a sound $\mathcal{O}(n^2)$ filtering algorithm for standard cumulative edge-finding, orthogonal to the work of Vilím; we also show how this algorithm's filtering may be improved by incorporating some reasoning from extended edge-finding, with no increase in complexity. The complexity of the new algorithm does not strictly dominate

A preliminary version of this paper was published in the proceedings of CP 2011 [8].

R. Kameugne
Department of Mathematics, Higher Teachers' Training College, University of Maroua,
P.O Box 1045 Maroua, Cameroon

R. Kameugne (✉)
Department of Mathematics, Faculty of Sciences, University of Yaoundé I,
PO Box 812, Yaoundé, Cameroon
e-mail: rkameugne@yahoo.fr; rkameugne@gmail.com

L. P. Fotso · Y. Ngo-Kateu
Department of Computer Sciences, Faculty of Sciences, University of Yaoundé I,
PO Box 812, Yaoundé, Cameroon

L. P. Fotso
e-mail: lpfotso@ballstate.bsu.edu

Y. Ngo-Kateu
e-mail: mireille_youcheu@yahoo.fr

J. Scott
Department of Information Technology, Computing Science Division, Uppsala University,
Box 337, SE-751 05 Uppsala, Sweden
e-mail: joseph.scott@it.uu.se

previous edge-finders for small k, and it sometimes requires more iterations to reach the same fixpoint; nevertheless, results from Project Scheduling Problem Library benchmarks show that in practice this algorithm consistently outperforms earlier edge-finding filters, and remains competitive with timetable edge-finding, despite the latter algorithm's generally stronger filtering.

Keywords Constraint-based scheduling · Edge-finding · Global constraints · Cumulative resource · Task intervals

1 Introduction

In the cumulative scheduling problem (CuSP), a finite set of tasks or activities must be scheduled on a resource of limited capacity. Each task requires a fixed and constant amount of the resource, and takes a fixed amount of time to complete. Each task has an earliest allowable start time, or *release date*, and a latest allowable completion time, or *deadline*. A CuSP instance is solved by determining a start time for each task, so that both the time and resource constraints are satisfied. Task scheduling is non-preemptive; that is, once a task is started, it is processed without interruption. The CuSP may be viewed as a relaxation of the resource-constrained project scheduling problem (RCPSP), a more general problem in which tasks may require several resources, and in which valid start times are constrained by an acyclic network of precedence constraints among the tasks. RCPSP solutions are further constrained to have a makespan (i.e., the time at which all tasks are completed) less than some fixed value, often called the horizon.

In constraint programming, a CuSP instance is often modeled with a specialized global constraint, CUMULATIVE [1]. CuSP is an NP-Complete problem [2], so propagation of CUMULATIVE is normally based on one or more relaxations of the problem, for which polynomial time filtering algorithms are known. Perhaps the most popular of these filtering algorithms is edge-finding. Given a set of tasks, T, edge-finding reduces the range of possible start times for each task $i \in T$ by deducing new ordering relations between i and some set of tasks $\Omega \subset T$, such that $i \notin \Omega$. Tightening the release date (i.e., the lower bound of possible start times) for i requires locating a set Ω such that if any task in Ω ends later than the end time of i, then the resulting schedule will be infeasible. For example, Fig. 1 shows a CuSP of 6 tasks sharing a resource of capacity 3. Task F has a release date of $r_F = 0$, yet it is apparent that there is no feasible schedule in which F has a start time of 0, as the set of tasks $\Omega = \{A, B, C, D, E\}$ require too much of the available capacity between the times 1 and 8 for F to be scheduled within that time frame as well. In fact, in any feasible schedule, all tasks in Ω must end before the end of F.

Based on this previously unknown precedence, a new lower bound for the start time of F may be deduced. It may be that more than one set could be selected for Ω; in this case, the Ω that justifies the maximum new lower bound should be located. Continuing our example, we see that the set $\{B, E\} \subset \Omega$ also consists of tasks which must end before the end of F in all schedules; however, knowing that Ω precedes F only justifies a new release date of $r'_F = 4$, while the knowledge that $\{B, E\}$ precedes F leads to a stronger release date of $r''_F = 6$. This process is repeated for the deadline (i.e., the upper bound of possible end times, directly related to the upper bound of possible start times by the fixed duration of the task) of each task. For the remainder of this paper we focus solely

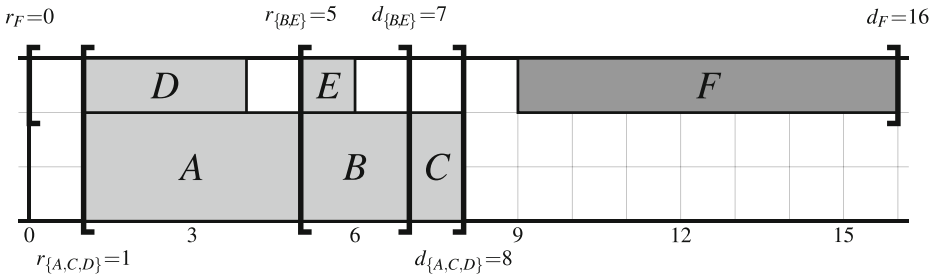

Fig. 1 A scheduling problem of 6 tasks sharing a resource of capacity $C = 3$

on the algorithm for updating release dates, as the deadline algorithm is symmetric and easily derived.

1.1 Related work

For disjunctive scheduling (i.e., scheduling on a resource that allows the execution of only one task at a time) on a set of n tasks, there are well-known $\mathcal{O}(n \log n)$ edge-finding algorithms [4, 16]. In cumulative scheduling, where tasks may have different capacity requirements and multiple tasks may execute at the same time, edge-finding is more challenging. In early work, Nuijten [11] gave a cumulative edge-finding algorithm of $\mathcal{O}(kn^2)$ complexity, where k is the number of distinct capacity requirements among the tasks; Baptiste et al. [3] subsequently refined the complexity to $\mathcal{O}(n^2)$. In both of these algorithms, the Ω that justified the maximal update to the release date of i was identified by considering the amount of the resource not used by the tasks of Ω during the interval of time those tasks would be scheduled. This quantity is often called the *slack* of a set of tasks; the idea of these algorithms was that the task set with the minimum slack would yield the maximum update.

Mercier and Van Hentenryck [10] demonstrated that the relationship between minimum slack and maximum update, while correct for disjunctive edge-finding, does not always hold for cumulative edge finding. The previous algorithms, while correct in the sense that they did not perform any unwarranted pruning, were shown to be incomplete: they failed to deduce some of the pruning justified by the edge-finding rule. Using a dynamic programming approach, they provide a complete algorithm, at the cost of increasing the complexity to $\mathcal{O}(kn^2)$.

The next major breakthrough in complexity came from Vilím [17]. In prior work, he had used a customized data structure, called a Θ-tree, to perform disjunctive edge-finding in $\mathcal{O}(n \log n)$ time [16]. Vilím [17] generalizes that work to the cumulative case, resulting in an $\mathcal{O}(kn \log n)$ algorithm for cumulative edge-finding. More recently, [19] provides an $\mathcal{O}(n^2)$ filtering algorithm for a new relaxation of CUMULATIVE called timetable edge-finding. This relaxation is stronger (although not strictly stronger) than edge-finding; however, the filtering algorithm does not always make all the pruning deductions justified by the relaxation, leading to a higher complexity in those instances.

1.2 Contribution

In this paper, we present an $\mathcal{O}(n^2)$ cumulative edge-finding algorithm, orthogonal to the approach in [19]. Our algorithm is motivated by the observation that the minimum slack approach of [3, 11] is correct in many cases. For a given task i, there exists some set of

tasks Ω that would justify a stronger update to the release date of i than any other set; if the earliest release date of any task in Ω is less than or equal to the release date of i, then Ω will have a lower slack than any other such task set (see Theorem 2). For task sets where all tasks in Ω have a release date greater then the release date of i, instead of slack we consider *density*: intuitively, the density is the average resource usage of the tasks over the time interval. We show that if any of these task sets would justify an update to the bound of i, then the set with the maximum density will also justify an update (also Theorem 2). Similar to the algorithm in [19], in the latter case it is possible that the update made in the first iteration of our algorithm is not the strongest possible update; however, the strongest update is guaranteed to be found on subsequent iterations, and we argue that in practice the overall effect on the complexity of the algorithm is minimal (see Section 7).

The paper proceeds as follows. Section 2 defines the cumulative scheduling problem, and the notations used in the paper. Section 3 defines the standard edge finding rules, and Section 4 provides dominance properties of these rules. Section 5 presents the new edge-finding algorithm and a proof of its correctness is given in Section 6. Section 7 discusses the overall complexity of the algorithm. Section 8 describes further improvement by incorporating some ideas from the extended edge-finding. Section 9 compares this algorithm with related edge-finding algorithms. Finally, in Section 10, the performance of this new algorithm is studied on benchmarks from the Project Scheduling Problem Library (PSPLib [12] and Baptiste and Le Pape BL set [2]).

A preliminary version of this paper appeared in the proceedings of CP 2011 [8]. Here we extend the presentation as follows: (i) several proofs have been expanded and clarified, (ii) several additional examples have been provided, (iii) the filtering power of the algorithm is enhanced by incorporating some filtering of the extended edge-finding rule, (iv) more complete experimental results are provided, and (v) we make a more explicit comparison with the timetable edge-finding algorithm [19], including a demonstration that this new rule does not subsume the traditional edge-finding rule.

2 Cumulative scheduling problem

Definition 1 (**CuSP**) A Cumulative Scheduling Problem (CuSP) is defined by a set T of tasks to be performed on a resource of capacity C. Each task $i \in T$ must be executed without interruption over p_i units of time between an earliest start time r_i (release date) and a latest end time d_i (deadline). Moreover, each task requires a constant amount of resource c_i. All times are assumed to be integers. A solution of a CuSP is a schedule that assigns a starting time s_i to each task i such that:

$$\forall i \in T : r_i \leq s_i \leq s_i + p_i \leq d_i \tag{1}$$

$$\forall \tau : \sum_{i \in T,\ s_i \leq \tau < s_i + p_i} c_i \leq C \tag{2}$$

The inequalities in (1) ensure that each task is assigned a feasible start and end time, while (2) enforces the resource constraint. An example of a CuSP is given in Fig. 1.

We define the energy of a task i as $e_i = c_i \cdot p_i$. This notation, along with that of earliest start and latest completion time, may be extended to sets of tasks as follows:

$$r_\Omega = \min_{j \in \Omega} r_j, \quad d_\Omega = \max_{j \in \Omega} d_j, \quad e_\Omega = \sum_{j \in \Omega} e_j \tag{3}$$

where Ω is a non-empty set of tasks.

By convention, if Ω is the empty set, $r_\Omega = +\infty$, $d_\Omega = -\infty$, and $e_\Omega = 0$. Throughout the paper, we assume that for any task $i \in T$, $r_i + p_i \leq d_i$ and $c_i \leq C$, otherwise the problem has no solution. We let $n = |T|$ denote the number of tasks, and $k = |\{c_i, i \in T\}|$ denote the number of distinct capacity requirements.

Clearly, if there exists a set of tasks $\Omega \subseteq T$ which cannot be scheduled in the window from r_Ω to d_Ω without exceeding the capacity, then the CuSP has no feasible solution. *Overload checking* algorithms typically enforce the following relaxation of this feasibility condition, which may be computed in $\mathcal{O}(n \log n)$ time [18, 20]:

Definition 2 (**E-Feasibility**) [10] A problem is E-feasible if $\forall \Omega \subseteq T,\ \Omega \neq \emptyset$

$$C(d_\Omega - r_\Omega) \geq e_\Omega \ . \tag{4}$$

Clearly a CuSP that fails the E-feasibility condition cannot have a feasible solution. In sequel, we consider only E-feasible CuSPs.

3 The edge-finding rule

The main idea of edge-finding is to identify a set of tasks $\Omega \subset T$ and a task $i \notin \Omega$ such that, in any solution, all the tasks in Ω end before the end of i; following [17], we denote this relationship $\Omega \ll i$. Once an appropriate Ω and i have been located, the earliest start time of i can be adjusted using the following rule:

$$\Omega \ll i \Longrightarrow r_i \geq r_\Theta + \left\lceil \frac{1}{c_i} \operatorname{rest}(\Theta, c_i) \right\rceil \tag{5}$$

for all $\Theta \subseteq \Omega$ such that $\operatorname{rest}(\Theta, c_i) > 0$, where

$$rest(\Theta, c_i) = \begin{cases} e_\Theta - (C - c_i)(d_\Theta - r_\Theta) & \text{if } \Theta \neq \emptyset \\ 0 & \text{otherwise} \end{cases} . \tag{6}$$

The condition $\operatorname{rest}(\Theta, c_i) > 0$ states that the total energy e_Ω that must be scheduled in the window $[r_\Omega, d_\Omega)$ is strictly larger than the energy that could be scheduled without making any start time of i in that window infeasible. The proof of these results can be found in [3, 11].

It remains to define what tasks and sets of tasks satisfy the condition $\Omega \ll i$. Proposition 1 provides conditions under which all tasks of a set Ω of an E-feasible CuSP end before the end of a task i.

Proposition 1 *Let Ω be a set of tasks and let $i \notin \Omega$ be a task of an E-feasible CuSP of capacity C.*

$$e_{\Omega \cup \{i\}} > C\left(d_\Omega - r_{\Omega \cup \{i\}}\right) \Rightarrow \Omega \ll i \ , \tag{EF}$$

$$r_i + p_i \geq d_\Omega \ \Rightarrow \ \Omega \ll i \ . \tag{EF1}$$

Proof (EF) is the traditional edge-finding rule; proof can be found in [3, 11]. The addition of (EF1), proposed in [17], strengthens the edge-finding rule; the proof follows trivially from the fact that $r_i + p_i \geq d_\Omega$ implies that task i ends after all tasks in the set Ω. □

Example 1 In the example shown in Fig. 1, the rule (EF) correctly detects $\Omega \ll F$ for $\Omega = \{A, B, C, D, E\}$. Using the set $\Theta = \Omega$ in formula (5), shows that the release date of F

may be updated to 5; however, allowing $\Theta = \{B, E\}$ instead yields an updated bound of 6. A value of $\Omega = \{B, E\}$ would not meet the edge-finding condition in (EF); the set $\{A, B, C, D, E\}$ is needed to detect the precedence condition.

Example 2 Figure 2 illustrates the purpose of rule (EF1). Task I cannot end before $r_I + p_I = 5$; since $d_{\{G,H\}} = 4$, clearly $\{G, H\}$ must end before the end of I in any feasible schedule. Due to (5), $\{G, H\} \ll I$ implies that r_I can be updated from 0 to 3; however, (EF) fails to detect $\{G, H\} \ll I$. The rule (EF1), on the other hand, leads to the correct updating of r_I.

Combining (EF) and (EF1) with (5) gives us a formal definition of an edge-finding algorithm as it is studied in [8, 17]:

Definition 3 (**Specification of a complete edge-finding algorithm**) An edge-finding algorithm receives as input an E-feasible CuSP, and produces as output a vector of updated lower bounds for the release times of the tasks $\langle LB_1, \ldots, LB_n \rangle$, where:

$$LB_i = \max\left(r_i, \max_{\Omega \subseteq T \setminus \{i\} | \alpha(\Omega, i)} \left(\max_{\Theta \subseteq \Omega | \text{rest}(\Theta, c_i) > 0} \left(r_\Theta + \left\lceil \frac{1}{c_i} \text{rest}(\Theta, c_i) \right\rceil \right) \right)\right) \tag{7}$$

with

$$\alpha(\Omega, i) \stackrel{def}{=} \left(C\left(d_\Omega - r_{\Omega \cup \{i\}}\right) < e_{\Omega \cup \{i\}}\right) \quad \vee \quad (r_i + p_i \geq d_\Omega) \tag{8}$$

and

$$\text{rest}(\Theta, c_i) = \begin{cases} e_\Theta - (C - c_i)\,(d_\Theta - r_\Theta) & \text{if } \Theta \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

4 Dominance properties of the rules

Clearly an edge-finder cannot efficiently consider all sets $\Theta \subseteq \Omega \subset T$ to update a task i. In order to reduce the number of sets which must be considered, we first consider the following definition:

Definition 4 (**Task Intervals**) (After [5]) Let $L, U \in T$. The task interval $\Omega_{L,U}$ is the set of tasks

$$\Omega_{L,U} = \left\{ j \in T \mid r_L \leq r_j \wedge d_j \leq d_U \right\}. \tag{10}$$

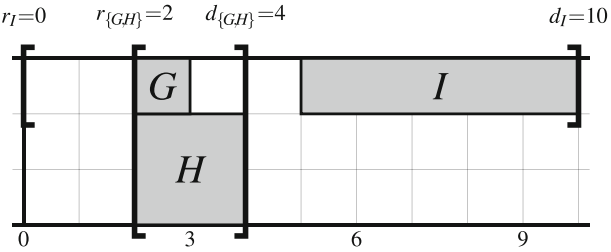

Fig. 2 Three tasks to be scheduled on a resource of capacity $C = 3$

It is demonstrated in [10] that an edge-finder that only considers sets $\Omega \subseteq T$ and $\Theta \subseteq \Omega$ which are also task intervals can be complete. Furthermore, we can reduce the number of intervals that must be checked according to the following propositions:

Proposition 2 *(After* [10]*) Let i be a task and Ω, Θ be two task sets of an E-feasible CuSP with $\Theta \subseteq \Omega$. If the edge-finding rule* (EF) *applied to task i with pair (Ω, Θ) allows to update the earliest start time of i then*

(i) *there exists four tasks L, U, ℓ, u such that $r_L \leq r_\ell < d_u \leq d_U < d_i \wedge r_L \leq r_i$*
(ii) *the edge-finding rule* (EF) *applied to task i with the pair $\left(\Omega_{L,U}, \Theta_{\ell,u}\right)$ allows at least the same update of the earliest start time of task i.*

Proposition 2 only covers edge finding with the rule (EF); we now extend this dominance property to additionally cover the rule (EF1):

Proposition 3 *Let i be a task and Ω, Θ be two task sets of an E-feasible CuSP with $\Theta \subseteq \Omega$. If the edge-finding rule (EF1) applied to task i with pair (Ω, Θ) allows to update the earliest start time of i then*

(i) *there exists four tasks L, U, ℓ, u such that $r_L \leq r_\ell < d_u \leq d_U < d_i$*
(ii) *the edge-finding rule* (EF1) *applied to task i with the pair $\left(\Omega_{L,U}, \Theta_{\ell,u}\right)$ allows at least the same update of the earliest start time of task i.*

Proof The sets Ω and Θ are not empty since $\Theta \subseteq \Omega$ and $rest(\Theta, c_i) > 0$. Therefore, there exists four tasks $L, U, \ell, u \in T$, such that $r_L = r_\Omega$, $d_U = d_\Omega$, $r_\ell = r_\Theta$ and $d_u = d_\Theta$ (if there are more tasks with this property, we choose arbitrarily). We have $r_L \leq r_\ell < d_u \leq d_U$ since $\Theta \subseteq \Omega$ and $\Theta \neq \emptyset$. If $d_i \leq d_\Omega$ then $r_i + p_i = d_i$ (since $r_i + p_i \leq d_i$ and $r_i + p_i \geq d_\Omega$), in which case the bounds of i cannot be updated, contradicting the assumption that (Ω, Θ) updates i; therefore $d_i > d_\Omega$. The inclusion $\Theta \subseteq \Theta_{\ell,u}$ implies that $rest(\Theta_{\ell,u}, c_i) > 0$ (since $rest(\Theta, c_i) > 0$) and

$$r_\ell + \frac{1}{c_i} rest\left(\Theta_{\ell,u}, c_i\right) \geq r_\Theta + \frac{1}{c_i} rest\left(\Theta, c_i\right) \quad . \tag{11}$$

Therefore, the rule (EF1) applied to task i with the pair $(\Omega_{L,U}, \Theta_{\ell,u})$ allows at least the same update of the earliest start time of task i as (Ω, Θ) would. □

5 Edge-finding with slack and density

In this section, we present a quadratic edge-finding algorithm that reaches the same fixpoint as the well known edge-finding algorithm proposed by Vilím [17]. Let Ω be a set of tasks and i be a task not belong in Ω. The main idea of the algorithm is that once the relation $\Omega \ll i$ is discovered, then it is not necessary to iterate over all subsets Θ of Ω. It is enough to consider only (1) the subset with minimum slack and $r_\Theta \leq r_i$ and (2) the subset with maximum density and $r_\Theta > r_i$. If r_i can be improved, then one of these two subsets will always justify an update, although not necessarily the strongest update on the first iteration. Further iterations of the algorithm must locate the strongest update, however. Proof of these claims follow in Sections 6 and 7.

5.1 Minimum slack

We start by considering the incomplete $\mathcal{O}(n^2)$ edge-finding algorithm [3, Algorithm 8]. In this algorithm, as in [11], the set Θ for the inner maximization of Definition 3 is identified by locating the task intervals with the minimum *slack*, as given by the following definitions.

Definition 5 Let Ω be a task set of an E-feasible CuSP. The *slack* of the task set Ω, denoted SL_Ω, is given by:

$$SL_\Omega = C(d_\Omega - r_\Omega) - e_\Omega.$$

Definition 6 Let i and U be two tasks of an E-feasible CuSP. $\tau(U, i)$ is a task depending on the tasks U and i, where $r_{\tau(U,i)} \leq r_i$ and that defines the task interval with the *minimum slack*: for all $L \in T$ such that $r_L \leq r_i$,

$$C(d_U - r_{\tau(U,i)}) - e_{\Omega_{\tau(U,i),U}} \leq C(d_U - r_L) - e_{\Omega_{L,U}} \quad .$$

For a given task i, the algorithm detects $\Omega \ll i$ by computing $SL_\Omega < e_i$ for all $\Omega = \Omega_{\tau(U,i),U}$ such that $d_U < d_i$ and $r_{\tau(U,i)} \leq r_i$. Furthermore, if the interval $\Theta_{\ell,u}$ that yields the strongest update to r_i is such that $r_\ell \leq r_i$, then $\Theta_{\ell,u}$ will be the interval of minimum slack. This is the situation shown in Fig. 3. Rather than determine r_ℓ, the new algorithm computes a potential update to r_i using $\text{rest}(\Omega, c_i)$. However, if the strongest updating interval $\Theta_{\ell,u}$ has $r_i < r_\ell$, then $\Theta_{\ell,u}$ need not be the interval of minimum slack. As shown in [10], the cumulative edge-finding algorithm in [3] is incomplete because it fails to consider all of the task intervals $\Theta_{\ell,u} \subseteq \Omega$ that could result in the strongest update.

5.2 Maximum density

Since slack does not yield a reliable determination of the correct $\Theta_{\ell,u}$ whenever $r_i < r_\ell$, the task intervals in this range must be evaluated by another criteria. For this case, we introduce the notion of interval *density*. Intuitively, slack is the amount of the resource not used by the tasks in $\Theta_{\ell,u}$ in the time from r_ℓ to d_u; in contrast, density is the average resource requirement over that same time span.

Definition 7 Let Θ be a task set of an E-feasible CuSP. The *density* of the task set Θ, denoted $Dens_\Theta$, is given by:

$$Dens_\Theta = \frac{e_\Theta}{d_\Theta - r_\Theta}.$$

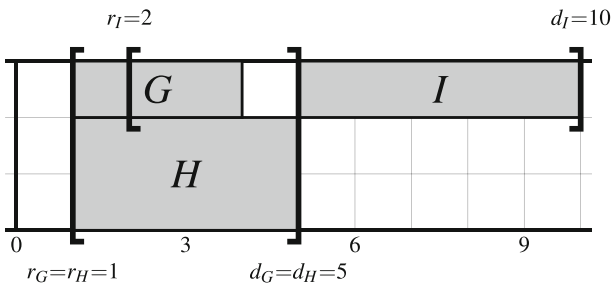

Fig. 3 Three tasks to be scheduled on a resource of capacity $C = 3$

Where the most interesting task intervals previously were those with the *minimum* slack, here we are interested in the interval with the *maximum* density.

Definition 8 Let i, u be two tasks of an E-feasible CuSP. $\rho(u, i)$ is a task depending on the tasks u and i, where $r_i < r_{\rho(u,i)}$ and that defines the task interval with the *maximum density*: for all tasks $\ell \in T$ such that $r_i < r_\ell$,

$$\frac{e_{\Theta_{\ell,u}}}{d_u - r_\ell} \leq \frac{e_{\Theta_{\rho(u,i),u}}}{d_u - r_{\rho(u,i)}} \ .$$

Algorithm 1: QUAD: Edge-finding in $\mathcal{O}(n^2)$ time and $\mathcal{O}(n)$ space

Require: T is an array of tasks
Ensure: A lower bound LB'_i is computed for the release date of each task i

for $i \in T$ **do**
  $LB'_i := r_i, \quad Dupd_i := -\infty, \quad SLupd_i := -\infty;$
for $U \in T$ *by non-decreasing deadline* **do**
  $Energy := 0, \quad maxEnergy := 0, \quad r_\rho := -\infty;$
  for $i \in T$ *by non-increasing release dates* **do**
    if $d_i \leq d_U$ **then**
      $Energy := Energy + e_i;$
      if $\left(\frac{Energy}{d_U - r_i} > \frac{maxEnergy}{d_U - r_\rho}\right)$ **then**
        $maxEnergy := Energy, \quad r_\rho := r_i \ ;$
    else
      $rest := maxEnergy - (C - c_i)(d_U - r_\rho);$
      if $(rest > 0)$ **then**
        $Dupd_i := \max(Dupd_i, r_\rho + \left\lceil \frac{rest}{c_i} \right\rceil);$
    $E_i := Energy \ ;$
  $minSL := +\infty, r_\tau := d_U \ ;$
  for $i \in T$ *by non-decreasing release date* **do**
    if $(C(d_U - r_i) - E_i < minSL)$ **then**
      $r_\tau := r_i, \quad minSL := C(d_U - r_\tau) - E_i \ ;$
    if $(d_i > d_U)$ **then**
      $rest' := c_i(d_U - r_\tau) - minSL \ ;$
      if $(r_\tau \leq d_U \wedge rest' > 0)$ **then**
        $SLupd_i := max(SLupd_i, r_\tau + \left\lceil \frac{rest'}{c_i} \right\rceil);$
      if $(r_i + p_i \geq d_U \vee minSL - e_i < 0)$ **then**
        $LB'_i := \max(LB'_i, Dupd_i, SLupd_i);$
for $i \in T$ **do**
  $r_i := LB'_i;$

If the strongest updating interval $\Theta_{\ell,u}$ has $r_i < r_\ell$, then it is always possible to strengthen r_i by considering the interval $\Theta_{\rho(u,i),u}$; proof of this statement follows in Theorem 2. The update calculated based on $\Theta_{\rho(u,i),u}$ may not be as strong as the update that would be justified by $\Theta_{\ell,u}$, although in most cases these two intervals will be the same; however, when $\Theta_{\rho(u,i),u} \neq \Theta_{\ell,u}$ it is always possible to calculate a new bound for r_i, and $\Theta_{\ell,u}$ and $\Theta_{\rho(u,i),u}$ are guaranteed to converge on a subsequent iteration (see Theorem 3). For a full discussion, see Section 7.

Example 3 Consider the scheduling problem shown in Fig. 1. We can update r_F using a task interval of maximum density, $\Theta_{\rho(u,i),u}$ where $d_u \leq d_\Omega$. For $u \in \{A, C, D\}$, the interval of maximum density is $\Theta_{A,D} = \{A, B, C, D, E\}$, which has a density of $18/7 \approx 2.6$.

Using (5) with $\Theta_{A,D}$ shows that we can strengthen the release date of F to $r_F \geq 5$. For $u \in \{B, E\}$, however, the interval of maximum density is $\Theta_{B,E} = \{B, E\}$, which has a density of $5/2 = 2.5$. Using (5) with $\Theta_{B,E}$ yields a new release date for F of $r_F \geq 6$, which is in fact the strongest update we can make.

Algorithm 1 computes the intervals of minimum slack and maximum density for all tasks $i \in T$, in $\mathcal{O}(n^2)$ time. The outer loop (line 3) selects, in the order of non-decreasing deadlines, the tasks $U \in T$ which form the possible upper bounds of the task intervals. The next step is to locate the lower bound ℓ for each task i, such that $\Theta_{\ell,U}$ yields the strongest correct update. As ℓ is unknown, either the maximum density interval or the minimum slack interval might yield a better update. Consequently, the algorithm calculates both, as follows:

1. The first inner loop (line 5) selects the tasks $i \in T$ that comprise the possible lower bounds for the task intervals, in non-increasing order by release date. If $d_i \leq d_U$, then the energy and density of $\Omega_{i,U}$ are calculated; if the new density is higher than $\Omega_{\rho(U,i),U}$, $\rho(U, i)$ becomes i. If $d_i > d_U$, then instead the potential update $Dupd_i$ to the release date of i is calculated, based on the current $\rho(U, i)$. This potential update is stored only if it is greater than the previous potential update value calculated for this task using the maximum density.
2. The second inner loop (line 16) selects i in non-decreasing order by release date. The energies stored in the previous loop are used to compute the slack of the current interval $\Omega_{i,U}$. If the slack is lower than that of $\Omega_{\tau(U,i),U}$, $\tau(U, i)$ becomes i. For any task with a deadline greater than d_U a new potential update $SLupd_i$ for the task's release date is calculated using $\tau(U, i)$ and it is checked to see if it meets either edge-finding criteria (EF) or (EF1). This potential update is stored only if it is greater than the previous potential update value calculated for this task using the minimum slack.

At the next iteration of the outer loop, $\rho(U, i)$ and $\tau(U, i)$ are re-initialized.

6 Proof of correctness

Before showing that Algorithm 1 is correct, let us prove some properties of its inner loops.

Proposition 4 *For each task i, Algorithm 1 calculates a potential update $Dupd_i$ to r_i based on the task interval of maximum density such that*

$$Dupd_i = \max_{\substack{U:\, d_U < d_i \\ \text{rest}(\Theta_{\rho(U,i),U},\, c_i) > 0}} \left(r_{\rho(U,i)} + \left\lceil \text{rest}(\Theta_{\rho(U,i),U}, c_i) \cdot \frac{1}{c_i} \right\rceil \right) . \tag{12}$$

Proof Let $i \in T$ be any task. Each choice of $U \in T$ in the outer loop (line 3) starts with the values $r_\rho = -\infty$ and $maxEnergy = 0$. The inner loop at line 5 iterates through all tasks $i' \in T$ (T sorted in non-increasing order of release date). For any task $i' \in T$ such that $r_{i'} \geq r_i$, if $d_{i'} \leq d_U$, then $i' \in \Theta_{i',U}$, so $e_{i'}$ is added to $Energy$ (line 7). Hence $Energy = e_{\Theta_{i',U}}$ at each iteration. The test on line 8 ensures that r_ρ and $maxEnergy = e_{\Theta_{\rho,U}}$ are updated to reflect $\rho(U, i)$ for the current task interval $\Theta_{i',U}$. Therefore, at the ith iteration of the

inner loop, if $d_i > d_U$ then line 11 computes $\text{rest}(i, U) = \text{rest}(\Theta_{\rho(U,i),U}, c_i)$. The potential update value is computed on line 13:

$$Dupd_i' = \begin{cases} r_\rho + \left\lceil \text{rest}(i, U) \cdot \frac{1}{c_i} \right\rceil & \text{if } \text{rest}(i, U) > 0 \\ -\infty & \text{otherwise.} \end{cases} \tag{13}$$

Note that on line 13, $Dupd_i$ only gets the new value from (13) if $Dupd_i'$ is larger than the previous value of $Dupd_i$. Since the outer loop selects the task U in non-decreasing order by d_U, we have:

$$Dupd_i = \max_{\substack{U : d_U < d_i \\ \text{rest}(i,U) > 0}} r_\rho + \left\lceil \text{rest}(i, U) \cdot \frac{1}{c_i} \right\rceil . \tag{14}$$

Hence formula (12) holds and the proposition is correct. □

Proposition 5 *For each task i, Algorithm 1 calculates a potential update $SLupd_i$ to r_i based on the task interval of minimum slack such that*

$$SLupd_i = \max_{\substack{U : d_U < d_i \\ r_{\tau(U,i)} \le d_U \\ \text{rest}(\Omega_{\tau(U,i),U}, c_i) > 0}} \left(r_{\tau(U,i)} + \left\lceil \text{rest}(\Omega_{\tau(U,i),U}, c_i) \cdot \frac{1}{c_i} \right\rceil \right) . \tag{15}$$

Proof Let $i \in T$ be any task. Each choice of U in the outer loop (line 3) starts with the values $r_\tau = d_U$ and $minSL = +\infty$ (line 15). The inner loop at line 16 iterates through all tasks $i' \in T$ (T sorted in non-decreasing order of release dates). For every task $i' \in T$, $e_{\Omega_{i',U}}$ has already been computed in the first loop and stored as $E_{i'}$ (line 14); this is used to compute the slack of $\Omega_{i',U}$. If $SL_{\Omega_{i',U}} < minSL$, the values $r_\tau = r_{i'}$ and $minSL = C(d_U - r_{i'}) - e_{\Omega_{i',U}}$ are updated to reflect $\tau(U, i)$ for the current task interval $\Omega_{i',U}$. At the ith iteration, if $d_i > d_U$, then line 20 computes $rest'(i, U) = \text{rest}(\Omega_{\tau(U,i),U}, c_i)$. The potential update value is computed on line 22:

$$SLupd_i' = \begin{cases} r_\tau + \left\lceil \text{rest}'(i, U) \cdot \frac{1}{c_i} \right\rceil & \text{if } r_\tau \le d_U \wedge \text{rest}'(i, U) > 0 \\ -\infty & \text{otherwise} . \end{cases} \tag{16}$$

Note that on line 22, $SLupd_i$ only gets the new value from (16) if $SLupd_i'$ is larger than the previous value of $SLupd_i$. Since the outer loop selects the task U in non-decreasing order by d_U we have:

$$SLupd_i = \max_{\substack{U : d_U \le d_i \wedge r_\tau \le d_U \\ \text{rest}'(i,U) > 0}} r_\tau + \left\lceil \text{rest}'(i, U) \cdot \frac{1}{c_i} \right\rceil . \tag{17}$$

Hence formula (15) holds and the proposition is correct. □

We now provide a proof that the edge-finding condition (EF) can be checked using minimum slack.

Theorem 1 *For any task $i \in T$ and set of tasks $\Omega \subseteq T \setminus \{i\}$,*

$$e_{\Omega \cup \{i\}} > C(d_\Omega - r_{\Omega \cup \{i\}}) \quad \vee \quad r_i + p_i \ge d_\Omega \tag{18}$$

if and only if

$$e_i > C(d_U - r_{\tau(U,i)}) - e_{\Omega_{\tau(U,i),U}} \quad \vee \quad r_i + p_i \geq d_U \tag{19}$$

for some task $U \in T$ *such that* $d_U < d_i$, $d_U = d_\Omega$ *and* $\tau(U, i)$ *as specified in Definition 6.*

Proof Let $i \in T$ be any task. It is obvious that (EF1) can be checked by $r_i + p_i \geq d_U$ for all tasks $U \in T$ with $d_i > d_\Omega$. In the rest of the proof, we focus on the rule (EF). We start by demonstrating that (18) implies (19). Assume there exists a subset $\Omega \subseteq T \setminus \{i\}$ such that $C(d_\Omega - r_{\Omega\cup\{i\}}) < e_\Omega + e_i$. By (EF), $\Omega \lessdot i$. By Proposition 2, there exists a task interval $\Omega_{L,U} \lessdot i$, such that $d_i > d_U$ and $r_L \leq r_i$. By Definition 6 we have

$$C(d_U - r_{\tau(U,i)}) - e_{\Omega_{\tau(U,i),U}} \leq C(d_U - r_L) - e_{\Omega_{L,U}} \tag{20}$$

$$C(d_U - r_{\tau(U,i)}) \leq e_{\Omega_{\tau(U,i),U}} + C(d_U - r_L) - e_{\Omega_{L,U}} \tag{21}$$

Using the fact that

$$C(d_U - r_L) < e_{\Omega_{L,U}} + e_i, \tag{22}$$

it follows that

$$C(d_U - r_{\tau(U,i)}) < e_{\Omega_{\tau(U,i),U}} + (e_{\Omega_{L,U}} + e_i) - e_{\Omega_{L,U}} \tag{23}$$

$$C(d_U - r_{\tau(U,i)}) < e_{\Omega_{\tau(U,i),U}} + e_i. \tag{24}$$

Now we show that (19) implies (18). Let $U \in T$ such that $d_U < d_i$, and $\tau(U, i) \in T$, be tasks that satisfy (19). By the definition of task interval, $d_i > d_U$ implies $i \notin \Omega_{\tau(U,i),U}$. Since $r_{\tau(U,i)} \leq r_i$, we have

$$e_{\Omega_{\tau(U,i),U}} + e_i > C(d_U - r_{\tau(U,i)}) \geq C(d_{\Omega_{\tau(U,i),U}} - r_{\Omega_{\tau(U,i),U}\cup\{i\}}). \tag{25}$$

Hence, (18) is satisfied for $\Omega = \Omega_{\tau(U,i),U}$. □

Proposition 5 has shown that $\tau(U, i)$ and the minimum slack are correctly computed by the loop at line 16. Combined with Theorem 1 this justifies the use of $minSL - e_i < 0$ on line 23 to check (EF), where the condition (EF1) is also checked. Thus, for every task i Algorithm 1 correctly detects the sets $\Omega \subseteq T \setminus \{i\}$ for which rules (EF) and (EF1) demonstrate $\Omega \lessdot i$.

A complete edge-finder would always choose the set Θ for each task i that yielded the strongest update to the bound of i. In the following theorem, we demonstrate that our algorithm has the slightly weaker property of soundness; that is, the algorithm updates the bounds correctly, but might not always make the strongest adjustment to a bound on the first iteration.

Theorem 2 *For every task* $i \in T$*, and given the strongest lower bound* LB_i *as specified in Definition 3, Algorithm 1 computes some lower bound* LB'_i*, such that* $r_i < LB'_i \leq LB_i$ *if* $r_i < LB_i$*, and* $LB'_i = r_i$ *if* $r_i = LB_i$.

Proof Let $i \in T$ be any task. LB'_i is initialized to r_i. Because the value LB'_i is only updated by $\max(Dupd_i, SLupd_i, LB'_i)$ (line 24) after each detection, it follows that $LB'_i \geq r_i$. If the equality $LB_i = r_i$ holds, then no detection is found by Algorithm 1, and thus $LB'_i = r_i$ holds from the loop at line 25. In the rest of the proof, we assume that $r_i < LB_i$. By Propositions 2 and 3, there exist two task sets $\Theta_{\ell,u} \subseteq \Omega_{L,U} \subseteq T \setminus \{i\}$ such that $r_L \leq r_\ell < d_u \leq d_U < d_i$ and $r_L \leq r_i$, for which the following holds:

$$\alpha(\Omega_{L,U}, i) \quad \wedge \quad LB_i = r_{\Theta_{\ell,u}} + \left\lceil \frac{1}{c_i} \mathrm{rest}(\Theta_{\ell,u}, c_i) \right\rceil. \tag{26}$$

As demonstrated by Proposition 5 and Theorem 1, Algorithm 1 correctly detects the edge-finding condition; it remains only to demonstrate the computation of update values. Since (EF) and (EF1) use the same inner maximization, the following two cases hold for both rules:

1. $r_i < r_\ell$: Here we prove that the update can be made using the task interval of maximum density. According to Definition 8, we have

$$\frac{e_{\Theta_{\ell,u}}}{d_u - r_\ell} \leq \frac{e_{\Theta_{\rho(u,i),u}}}{d_u - r_{\rho(u,i)}}. \tag{27}$$

Since $(\Omega_{L,U}, \Theta_{\ell,u})$ allows the update of the release date of task i, we have $\text{rest}(\Theta_{\ell,u}, c_i) > 0$. Therefore,

$$\frac{e_{\Theta_{\ell,u}}}{d_u - r_\ell} > C - c_i. \tag{28}$$

By relations (27) and (28), it follows that $\text{rest}(\Theta_{\rho(u,i),u}, c_i) > 0$; furthermore, $r_i < r_{\rho(u,i)}$ implies

$$r_{\rho(u,i)} + \left\lceil \frac{1}{c_i} \text{rest}\left(\Theta_{\rho(u,i),u}, c_i\right) \right\rceil > r_i. \tag{29}$$

According to Proposition 4, line 13 of Algorithm 1 computes

$$Dupd_i = r_{\rho(u,i)} + \left\lceil \frac{1}{c_i} \text{rest}(\Theta_{\rho(u,i),u}, c_i) \right\rceil > r_i. \tag{30}$$

Therefore, after the detection condition is fulfilled at line 23, the release date of task i is updated to

$$LB_i' = \max(Dupd_i, SLupd_i) \geq Dupd_i > r_i. \tag{31}$$

2. $r_\ell \leq r_i$: Here we prove that the update can be made using the task interval of minimal slack. By Definition 6, we have:

$$C(d_u - r_{\tau(u,i)}) - e_{\Theta_{\tau(u,i),u}} \leq C(d_u - r_\ell) - e_{\ell,u}. \tag{32}$$

Adding $-c_i(d_u - r_{\tau(u,i)}) - c_i \cdot r_{\tau(u,i)}$ to the left hand side and $-c_i(d_u - r_\ell) - c_i \cdot r_\ell$ to the right hand side of (32) we get

$$-c_i \cdot r_{\tau(u,i)} - \text{rest}(\Theta_{\tau(u,i),u}, c_i) \leq -c_i \cdot r_\ell - \text{rest}(\Theta_{\ell,u}, c_i). \tag{33}$$

Therefore,

$$r_{\tau(u,i)} + \frac{1}{c_i} \text{rest}(\Theta_{\tau(u,i),u}, c_i) \geq r_\ell + \frac{1}{c_i} \text{rest}(\Theta_{\ell,u}, c_i) \tag{34}$$

and

$$r_{\tau(u,i)} + \frac{1}{c_i} \text{rest}(\Theta_{\tau(u,i),u}, c_i) > r_i \tag{35}$$

since $r_\ell + \frac{1}{c_i}\text{rest}(\Theta_{\ell,u}, c_i) > r_i$. From inequality (35), it follows that

$$\text{rest}(\Theta_{\tau(u,i),u}, c_i) > 0 \tag{36}$$

since $r_{\tau(u,i)} \leq r_i$. According to Proposition 5, the value

$$SLupd_i = r_{\tau(u,i)} + \frac{1}{c_i} \text{rest}(\Theta_{\tau(u,i),u}, c_i) > r_i \tag{37}$$

is computed by Algorithm 1 at line 22. Therefore, after the detection condition is fulfilled at line 23, the updated release date of task i satisfies $LB_i' > r_i$.

Hence, Algorithm 1 is sound. □

7 Overall complexity

According to Theorem 2, Algorithm 1 will always make some update to r_i if an update is justified by the edge-finding rules, although possibly not always the strongest update. As there are a finite number of updating sets, Algorithm 1 must reach the same fixpoint as other correct edge-finding algorithms. Theorem 3 illustrates the circumstances under which Algorithm 1 finds the strongest update immediately; when those circumstances do not apply, we show that Algorithm 1 requires at most $n - 1$ propagations.

Theorem 3 *Let $i \in T$ be any task of an E-feasible CuSP. Let $\Theta \subseteq T \setminus \{i\}$ be a set used to perform the maximum adjustment of r_i by the edge-finding rule. Let $\rho(u, i)$ be a task as given in Definition 8, applied to $i, u \in T$ with $d_u = d_\Theta$. Then Algorithm 1 performs the strongest update of r_i in the following number of iterations:*

1. *If $r_\Theta \leq r_i$, then on the first iteration,*
2. *If $r_i < r_\Theta$ then:*
   (a) *If $r_i < r_\Theta \leq r_{\rho(u,i)}$, then also on the first iteration,*
   (b) *If $r_i < r_{\rho(u,i)} < r_\Theta$, then after at most $n - 1$ iterations.*

Proof Given $i \in T$, let $\Theta \subseteq T \setminus \{i\}$ be a task set used to perform the maximum adjustment of r_i by the edge-finding rule. Let $\rho(u, i)$ be the task of Definition 8 applied to i and $u \in T$ with $d_u = d_\Theta$.

1. Assume $r_\Theta \leq r_i$. By Propositions 1 and 2, and the proof of the second item of Theorem 2, formula (34) holds. By Proposition 5, when Algorithm 1 considers u in the outer loop and i in the second inner loop, it sets

$$Dupd_i = r_{\tau(u,i)} + \left\lceil \frac{1}{c_i} \text{rest}(\Theta_{\tau(u,i),u}, c_i) \right\rceil \geq r_\Theta + \left\lceil \frac{1}{c_i} \text{rest}(\Theta, c_i) \right\rceil. \tag{38}$$

   As the adjustment value of Θ is maximal, r_i is updated to $r_\Theta + \left\lceil \frac{1}{c_i} \text{rest}(\Theta, c_i) \right\rceil$.
2. Assume $r_i < r_\Theta$. We analyze two subcases:
   (a) $r_i < r_\Theta \leq r_{\rho(u,i)}$: According to definition of task $\rho(u, i)$, we have

$$\frac{e_\Theta}{d_\Theta - r_\Theta} \leq \frac{e_{\Theta_{\rho(u,i),u}}}{d_u - r_{\rho(u,i)}}. \tag{39}$$

   Removing $\frac{e_{\Theta_{\rho(u,i),u}}}{d_\Theta - r_\Theta}$ from each side of (39) and multiplying each side by $(d_\Theta - r_\Theta)$ (which is positive), we get

$$e_\Theta - e_{\Theta_{\rho(u,i),u}} \leq \frac{e_{\Theta_{\rho(u,i),u}}}{d_u - r_{\rho(u,i)}} (r_{\rho(u,i)} - r_\Theta). \tag{40}$$

   As the problem is E-feasible, we have

$$\frac{e_{\Theta_{\rho(u,i),u}}}{d_u - r_{\rho(u,i)}} \leq C. \tag{41}$$

   Combining inequalities (40) and (41) gives

$$Cr_\Theta + e_\Theta \leq Cr_{\rho(u,i)} + e_{\Theta_{\rho(u,i),u}}. \tag{42}$$

Obviously, inequality (42) is equivalent to

$$r_\Theta + \left\lceil \frac{1}{c_i}\text{rest}(\Theta, c_i) \right\rceil \leq r_{\rho(u,i)} + \left\lceil \frac{1}{c_i}\text{rest}(\Theta_{\rho(u,i),u}, c_i) \right\rceil. \tag{43}$$

Proposition 4 shows that when Algorithm 1 considers u in the outer loop and i in the first inner loop, it sets

$$Dupd_i = r_{\rho(u,i)} + \left\lceil \frac{1}{c_i}\text{rest}(\Theta_{\rho(u,i),u}, c_i) \right\rceil \geq r_\Theta + \left\lceil \frac{1}{c_i}\text{rest}(\Theta, c_i) \right\rceil. \tag{44}$$

As the adjustment value of Θ is maximal, r_i is updated to $r_\Theta + \left\lceil \frac{1}{c_i}\text{rest}(\Theta, c_i) \right\rceil$.

(b) $r_i < r_{\rho(u,i)} < r_\Theta$: Let

$$\Theta^k_{\leq i} := \{j,\ j \in T \wedge r_j \leq r_i \wedge d_j \leq d_\Theta\}$$
$$\Theta^k_{>i} := \{j,\ j \in T \wedge r_j > r_i \wedge d_j \leq d_\Theta\} \tag{45}$$

be sets of tasks defined at the k$^{\text{th}}$ iteration of Algorithm 1. If the maximum adjustment is not found after this iteration, then at least one task is moved from $\Theta^k_{>i}$ to $\Theta^k_{\leq i}$. Indeed, if at the k$^{\text{th}}$ and k + 1$^{\text{th}}$ iteration we have $\Theta^k_{\leq i} = \Theta^{k+1}_{\leq i}$ and $\Theta^k_{>i} = \Theta^{k+1}_{>i}$, and the maximum adjustment is not found, then the tasks $\tau(u,i) \in \Theta^k_{\leq i} = \Theta^{k+1}_{\leq i}$ (Definition 6) and $\rho(u,i) \in \Theta^k_{>i} = \Theta^{k+1}_{>i}$ (Definition 8) are the same for both iterations. Therefore, at the k+1th iteration, no new adjustment is found, yet the maximum adjustment of the release date of task i is not reached, contradicting the soundness of Algorithm 1. Hence, the maximum adjustment of the release date of task i is reached after at most $\left|\Theta^1_{>i}\right| \leq n - 1$ iterations.

□

Example 4 Figure 4 illustrates case 2(b) of the proof of Theorem 3. The application of the edge-finding rule to task I and sets $\Omega = \{A, B, D\}$, $\Theta = \{D\}$ allows the update of r_I from 0 to 5, since

$$e_\Omega + e_I = 17 > C(d_\Omega - r_{\Omega\cup\{I\}}) = 15 \qquad \text{and} \qquad \text{rest}(\Theta, c_I) = 1 \tag{46}$$

together imply

$$r_I = r_\Theta + \left\lceil \frac{1}{c_I}\text{rest}(\Theta, c_i) \right\rceil = 5. \tag{47}$$

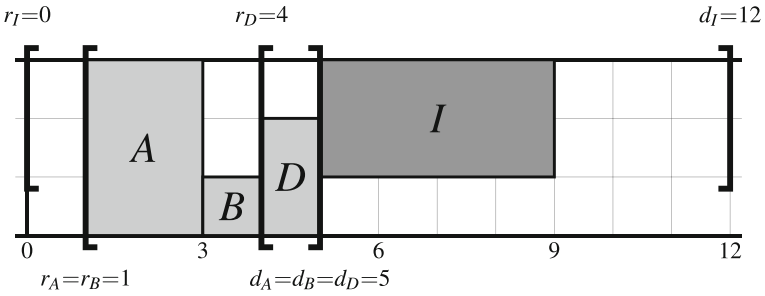

Fig. 4 An example of four tasks to be scheduled on a resource of capacity $C = 3$: The first run of Algorithm 1 update r_I from 0 to 4 and the second run from 4 to 5 which is the maximum adjustment

Using Algorithm 1, the relation $\Omega \ll I$ is detected for $U \in \{A, B, D\}$ in the outer loop (line 3). We have $r_{\rho(U,I)} = 1$ in the first inner loop (line 5) since $\Omega_{\rho(U,I),A} = \{A, B, C\}$, and $r_{\tau(U,I)} = 0$ in the second inner loop (line 16). The first update value based on maximum density is then $Dupd_I = 4$ since $rest(\Omega_{\rho(U,I),U}, c_I) = 5$, and the second potential update value based on minimum slack is $SLupd_I = 2$ since $rest(\Omega_{\tau(U,I),U}, c_I) = 4$. So after the first run of Algorithm 1, r_I is updated from 0 to 4. It is only on the second run that the maximum adjustment will be performed using $\Omega = \Theta = \{D\}$.

8 Improvement

The conjunction of the edge-finding rules (EF) and (EF1) is not sufficient to detect all cases where $\Omega \ll i$. As edge-finding is a relatively weak relaxation of the CUMULATIVE constraint, it is worth considering ways of strengthening it. Some of the missing cases can be detected by a related rule, often called the extended edge-finding rule [2, 10, 11]; however, the additional filtering power of extended edge-finding over standard edge-finding is limited, and comes at the cost of higher complexity (Mercier and Van Hentenryck provide an $\mathcal{O}(n^3)$ algorithm [10]). With a slight modification, Algorithm 1 may be extended to partially apply the extended edge-finding rule, with no increase in complexity.

Intuitively, extended edge-finding attempts to update r_i if scheduling a task i as early as possible (i.e., starting at r_i) would cause an overload in some interval $[r_\Omega, d_\Omega)$. Consider the example on Fig. 5: it is clear that I cannot be scheduled starting at 0 in any feasible schedule, and in fact for $\Omega = \{A, B\}$ we have $\Omega \ll I$. The edge-finding rule (EF), however, considers tasks A and B over the interval $[r_{\Omega\cup\{I\}}, d_\Omega)$, giving a slack of 5. Since $e_I = 4$, it appears to the edge-finding rule that I can be scheduled in this interval without causing an overload. Over the interval $[r_\Omega, d_\Omega)$ the tasks in Ω have a slack of only 2. If I was scheduled as early as possible, then the portion of I intersecting this interval would have an energy of $c_I(r_I + p_I - r_\Omega) = 3$, which is greater than the available slack, and therefore indicates an overload. The only alternative is to schedule I to end after the end of all tasks in Ω; in other words, $\Omega \ll i$. This reasoning leads to the extended edge-finding rule.

Proposition 6 *Let Ω be a set of tasks and let $i \notin \Omega$ be a task of an E-feasible CuSP of capacity C.*

$$\left(\begin{array}{c} r_i \le r_\Omega < r_i + p_i \\ \wedge \\ e_\Omega + c_i(r_i + p_i - r_\Omega) > C(d_\Omega - r_\Omega) \end{array}\right) \Longrightarrow \Omega \ll i \qquad \text{(EEF)}$$

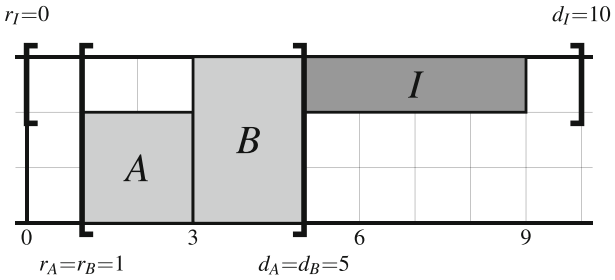

Fig. 5 Three tasks to be scheduled on a resource of capacity $C = 3$: r_I can be updated from 0 to 3

Proof The proof proceeds by contradiction. Given an E-feasible CuSP of capacity C, assume we have a set of tasks Ω and a task $i \notin \Omega$ such that

$$r_i \leq r_\Omega < r_i + p_i \tag{48}$$

and

$$e_\Omega + c_i(r_i + p_i - r_\Omega) > C(d_\Omega - r_\Omega) \tag{49}$$

Further, assume the negation of the consequence of (EEF); in other words, assume there exists a feasible solution with a task $j \in \Omega$ such that j does not end before the end of i. Therefore $r_i \leq r_j$, which implies $d_i \leq d_\Omega$. From (48) we see that i cannot end before r_Ω; therefore, some portion of i must occur within the window from r_Ω to d_Ω. The least portion of e_i that might occur in this window is $c_i(r_i + p_i - r_\Omega)$, obtained by scheduling i as early as possible. By definition, all of e_Ω also must be scheduled between r_Ω and d_Ω; so the minimum required capacity between r_Ω and d_Ω is $e_\Omega + c_i(r_i + p_i - r_\Omega)$. By (49), this required capacity must be greater than $C(d_\Omega - r_\Omega)$; in other words, the solution is not feasible, which contradicts our assumption. □

Returning to Algorithm 1, we observe that $r_i < r_{\rho(U,i)}$ is an invariant for the loop at line 5, as i is selected by non-increasing release date. Therefore, whenever $d_i > d_U$ inside this loop, whenever

$$maxEnergy + c_i(r_i + p_i - r_{\rho(U,i)}) > C(d_U - r_{\rho(U,i)}) \tag{50}$$

we must have $\Omega_{\rho(U,i),U} \ll i$. Algorithm 2 extends Algorithm 1 to detect this new condition: (50) is tested at line 13a and when satisfied line 13b updates the bound of i. Note that the adjustment in this case uses only the potential update based on density, since $\Omega_{\rho(U,i),U} \subset \Omega_{\tau(U,i),U}$.

Algorithm 2: IQUAD: Additional detection from extended edge-finding

```
5   for i ∈ T by non-increasing release dates do
6       if d_i ≤ d_U then
⋮           ⋮
10      else
11          rest := maxEnergy − (C − c_i)(d_U − r_ρ);
12          if (rest > 0) then
13              Dupd_i := max(Dupd_i, r_ρ + ⌈rest/c_i⌉);
13a         if (maxEnergy + c_i(r_i + p_i − r_ρ) > C(d_U − r_ρ)) then
13b             LB'_i := max(LB'_i, Dupd_i);
14      E_i := Energy;
```

9 Discussion

9.1 Lazy evaluation

According to Theorem 3, there are some cases in which Algorithm 1 will not make the strongest update justified by the edge-finding rule in a single iteration, although it will

always make this update in a subsequent iteration. We argue that, in practice, the possibility of our algorithm using multiple iterations to find the strongest bound is not significant.

First, filtering algorithms for edge-finding are not idempotent. The pruning of the start times for any task is dependent on the upper and lower bounds of the start times of the remaining tasks. An adjustment to the release date or deadline of any task may result in additional pruning for the remaining tasks; therefore, additional propagations are always required after any adjustment, if only to recognize when the propagator is at fixpoint. Second, in actual cumulative problems, there are typically a relatively small number of task sets that could be used to update the start time of a given task, so the number of propagations should not normally approach the worst case. This claim is borne out by the experimental observations reported in Section 10.4.

Other recent filtering algorithms for cumulative, such as not-first/not-last [7, 14], also rely on precisely this sort of "lazy" propagation. It should be noted that this is quite a different situation from the incomplete filtering of the $\mathcal{O}(n^2)$ edge-finding algorithm in [3]; in the counter-example provided from [10], some pruning justified by edge-finding is simply not performed by the earlier algorithms. By Theorem 3, we know that this will never be the case for our algorithm.

Finally, it should be noted that the only other known $\mathcal{O}(n^2)$ algorithm for edge-finding, timetable edge-finding, also depends upon "lazy" propagation [19].

9.2 Timetable edge-finding

The pruning performed by the timetable edge-finding algorithm of [19] is similar to the pruning justified by the conjunction of (EF) and (EF1), but it is not identical; timetable edge-finding borrows reasoning from both the timetable and energetic reasoning filtering algorithms in order to make a stronger deduction than standard edge-finding. Nevertheless, timetable edge-finding is not *strictly* stronger than edge-finding, as claimed in [19].

Example 5 (*Timetable edge-finding counterexample*) Consider the CuSP instance of Fig. 6. When applying the standard edge-finding rule to this instance, the release date of task I can be adjusted using the set $\{A, B, C\}$. For $\Omega = \{A, B, C\}$, we have

$$e_{\Omega\cup\{I\}} = 16 > C(d_\Omega - r_{\Omega\cup\{I\}}) = 15 \tag{51}$$

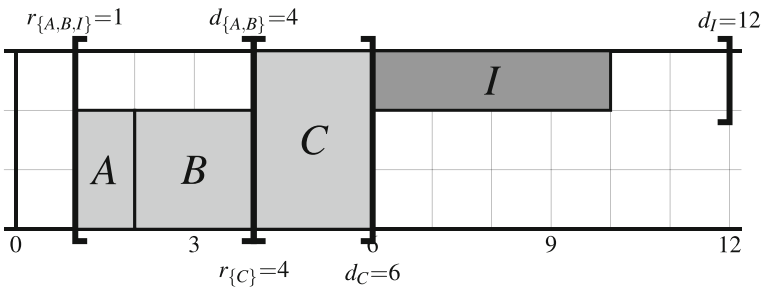

Fig. 6 Three tasks to be scheduled on a resource of capacity $C = 3$

In other words, the standard edge-finding condition detects $\Omega \ll I$. Applying the adjustment rule with $\Theta = \{C\}$, we get $rest(\Theta, c_I) = 2$, and a new release date of:

$$r_\Theta + \left\lceil \frac{rest(\Theta, c_I)}{c_I} \right\rceil = 6. \tag{52}$$

Applying the timetable edge-finding rule on this instance results in no update to r_I. The problem arises from task C, which has no free part, only a required part. The edge-finding portion of the timetable edge-finding rule considers only the free parts of the tasks; in this case, $\Omega \subseteq T^{EF} \setminus \{I\} = \{A, B\}$, where T^{EF} denotes the set of tasks with non-empty free parts [19]. Clearly no Ω satisfying this condition justifies an update to r_I, so no adjustment is performed by the timetable edge-finding rule. Nor is this adjustment performed by the improved version of the timetable edge-finding rule [19].

The timetable edge finding rule only considers task intervals Ω where time bounds are tasks with non-empty free parts. This restriction unfortunately reduces the filtering power of the rule. In practice, edge-finding and related rules must be combined with some other filter that reasons on required parts, most often a variant of the timetable filtering algorithm [2]. Since a timetable algorithm would perform the pruning missed by the timetable edge-finding algorithm in the example in Fig. 6, this observation does not invalidate timetable edge-finding. Nevertheless, this counterexample clearly demonstrates that the claim of [19] that timetable edge-finding strictly dominates edge-finding is incorrect. The reasoning on required parts incorporated into timetable edge-finding strengthens the rule, but it is not complete, and a timetable filter is still required.

10 Experimental results

Experiments were carried out on two benchmark suites of RCPSP instances. The first, PSPLib [9, 12], comprises the three data sets J30, J60, and J90 of 30, 60, and 90 tasks, respectively. Each set was generated with 48 combinations of various parameter settings, making 10 random instances for each combination. The second suite, BL, has 40 instances of highly cumulative problems (i.e., problems in which many tasks can run concurrently) each of 20 or 25 tasks, and is due to Baptiste and Le Pape [2].

All tests were performed on a 3.07 GHz Intel Core i7 processor running OpenSUSE Linux. The code was implemented in C++, using the Gecode constraint programming toolkit, version 3.7.3 [6], and compiled with GCC 4.5.1. The Gecode library includes a propagator for the global constraint CUMULATIVE,

We compared several edge-finding algorithms. We implemented Algorithm 1 from this paper (QUAD); the dynamic programming, $\mathcal{O}(kn^2)$ algorithm for standard edge-finding of [10] (MVH); and the $\mathcal{O}(n^2)$ timetable edge-finding algorithm from [19] (TTEF). Gecode's propagator for the global constraint CUMULATIVE is implemented as a sequence of filtering algorithms for timetabling, overload checking, and edge-finding [13], the latter using the Θ-tree algorithm from [17]. For our tests, we kept the two other filters in place, substituting alternate edge-finding algorithms for the built-in. Wherever possible, Gecode utility functions were used to implement auxiliary functionality (e.g., task sorting), so as to limit the effect of uneven optimization among the algorithms.

Each instance from the PSPLib sets includes tasks to be scheduled over 4 resources, while instances from the BL suite share 3 resources; each resource was modeled with a single CUMULATIVE constraint. Additionally, as these instances are RCPSP problems, the

start times of the tasks are constrained by a precedence graph; precedence relations between pairs of tasks were enforced with linear constraints.

To compare the speed of the four algorithms, we analyzed the time required to find the optimal solution for each instance, where optimality is defined as the solution with the minimal makespan. Starting with the provided horizon as an upper bound, we used branch and bound search to minimize the makespan. Each test was run three times, with the best result reported; any search taking more than 300 seconds was counted as a failure.

Detailed results are available at http://dx.doi.org/10.6084/m9.figshare.736454.

10.1 CPU time

Table 1 provides a comparison of the four algorithms using a dynamic branching strategy: variable selection was based on a combination of domain size and the number of constraints each variable occurred in; values were taken from the smallest range for domains with multiple ranges, or the lesser half of the domain when only one range existed [13].

QUAD was able to find the optimal solution before timeout for the highest number of instances in each test suite. Additionally, on the J30 suite QUAD had the best runtime in more instances than the other algorithms. In J60, J90, and the BL tests, however, TTEF performed the best in general; furthermore, the performance gap between QUAD and TTEF appears to widen in favor of TTEF as the proportion of more challenging instances increases.

Comparing the runtimes of QUAD and THETA, we find that QUAD was faster in every instance. Recall that, while the former is independent of k (the number of distinct capacity requirements among the tasks), THETA has a complexity of $\mathcal{O}(kn \log n)$, so we would expect the comparative performance of QUAD to increase along with the value of k. Figure 7 demonstrates exactly that; each instance tested had either 3 (BL) or 4 (J30, J60, J90) resources, each with a different k-value, so the plot compares runtime with the mean k over all resources for each instance. Furthermore, the BL and PSPLib suites are plotted separately, as the two suites were generated with different basic parameters, resulting in somewhat different performance [2].

Comparing the runtime of TTEF and QUAD is less straightforward. TTEF was faster in a majority of test cases, but a close examination of the distribution of the speedup, shown in Fig. 8a, reveals that the runtimes were extremely close in almost all instances. Note that the distribution of speedups is skewed to the left; in a large number of instances, QUAD had a slightly faster runtime, but when instead TTEF was faster, it tended to be faster by a larger margin. Given that these two algorithms have different pruning strengths, and therefore generate differently shaped search trees, it is possible that in some cases a weaker algorithm will

Table 1 Number of instances in which each algorithm found the optimal solution (solve), did so in the fastest time (time), and generated the smallest search tree (nodes), using dynamic branching

	J30			J60			J90			BL		
	Solve	Time	Nodes	Solve	Time	Nodes	Solve	Time	Nodes	Solve	Time	Nodes
MVH	377	1	332	334	1	325	322	1	314	38	0	10
THETA	381	0	333	336	0	325	322	0	314	40	0	12
QUAD	386	250	333	336	117	325	324	61	314	40	15	12
TTEF	378	136	364	327	219	316	317	263	310	39	25	29

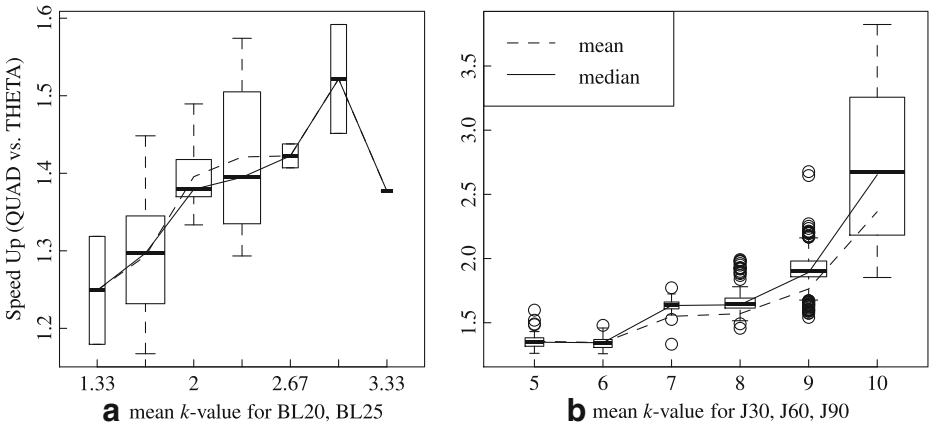

Fig. 7 Proportional speed up of QUAD over THETA for PSPLib instances, as a function of mean k-value over all resources in the instance. Box widths are proportional to square-root of number of instances for each k-value; whisker lengths represent highest/lowest datum within 1.5 times the interquartile range, with outliers individually represented

happen to lead to a faster solution, especially when dynamic branching is used; the effects of static branching on the comparative runtime of these algorithms is discussed below.

10.2 Nodes

In the preliminary version of this paper [8], we reported a small number of test instances in which the node count differed between Θ-tree and quadratic filtering. Specifically, in approximately 1 % of all test cases, QUAD resulted in a search tree with 1–2 % fewer nodes than THETA, seemingly in contradiction of our claim that these two algorithms reach the same fixpoint. Subsequent investigation revealed that the discrepancy was due to the Gecode implementation of the Θ-tree algorithm from [17], which (in versions prior to 3.7.2) missed

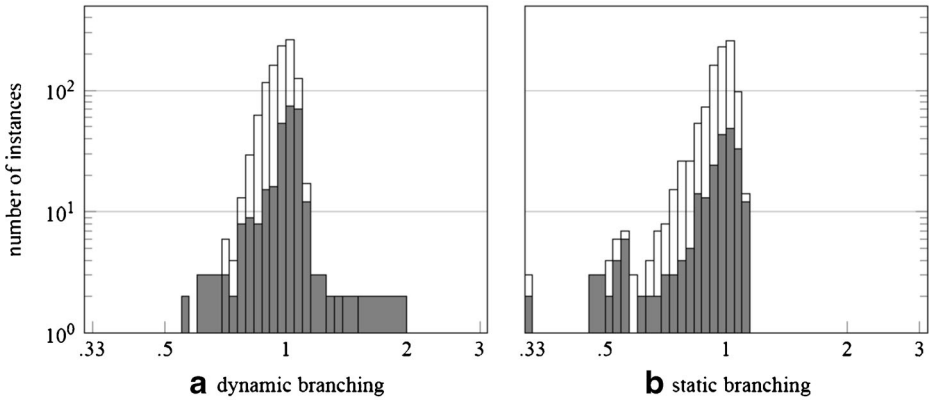

Fig. 8 Distribution of proportional speed up of QUAD vs. TTEF, with **a** dynamic, and **b** static branching. The *grey* portion of each bar represents instances pruned at least once by both algorithms; the *white* portions compare all other instances

a small amount of pruning as a result of issues omitted from [17] (for a complete description, see [15, Section 3.3.1]). Using Gecode 3.7.3, the node counts for QUAD and THETA are identical for all test cases, exactly as we would expect for two algorithms that perform the same pruning. A small difference in node counts between these two algorithms and MVH was observed, as expected; the portion of the rule we refer to as (EF1), implemented in the latter two algorithms only, accounts for slightly stronger pruning.

For a direct comparison of the number of nodes generated by QUAD and TTEF, it is most interesting to focus only on those instances where edge-finding results in some pruning of domains. To this end, we tested an additional propagator, BASE, which consisted solely of the two invariant filtering algorithms from our earlier tests (i.e., timetable and overlaod checking, but no edge-finding algorithm). In Fig. 9, we consider only those instances where either QUAD or TTEF resulted in a different node count than BASE. Note that, for the sake of space, we group the PSPLib instances by parameter settings number (1–48), and use a box and whisker plot to show the distribution of node count differences over the 10 instances in each group.

For the BL instances, both edge-finding filters introduced stronger pruning for almost all instances, which resulted in smaller search trees. For PSPLib, not only was the proportion of instances in which there was additional pruning smaller, but the reduction of nodes in those instances also tended to be smaller. Also, the generally stronger filtering of TTEF is in evidence here on the PSPLib instances; in some instances this stronger filtering resulted in a poorer branching decision, as evidenced by those instances where TTEF led to a larger search tree than BASE.

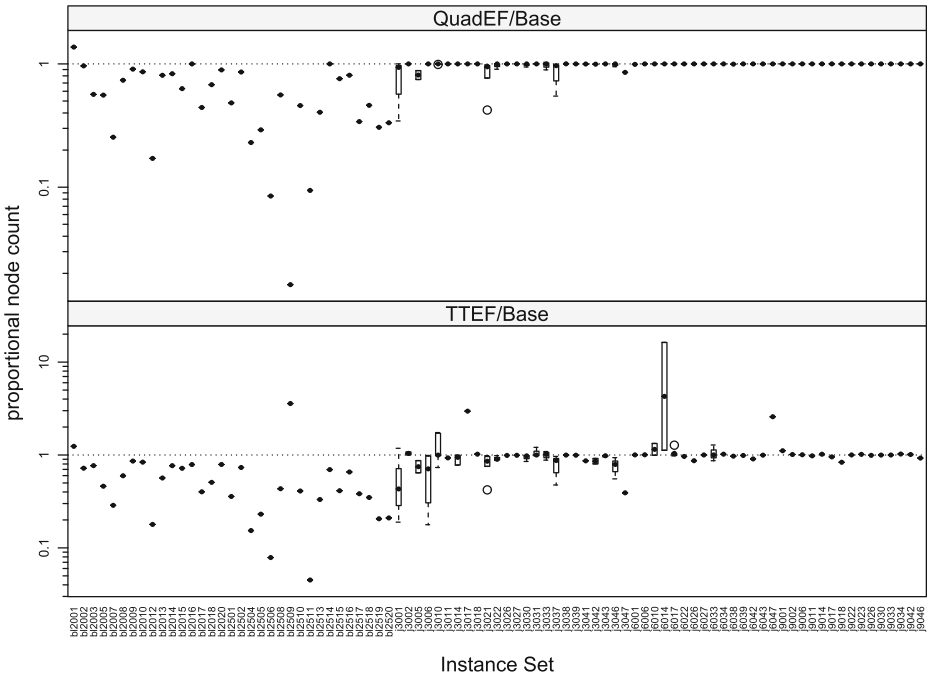

Fig. 9 Difference in node counts between QUAD and TTEF as compared to BASE. Only instances where some difference was observed are reported

10.3 Static branching

In order to minimize the effect of differing pruning strength on the shape of the search trees, we also tested the algorithms with static branching, selecting the first unassigned variable and the smallest value in the domain. Results are summarized in Table 2. For these tests, the slower MVH was discarded. Given that QUAD and THETA perform identical filtering, the choice of branching heuristic is insignificant when comparing the performance of these two algorithms; hence, our analysis focuses on the difference between QUAD and TTEF.

With static branching, TTEF generally outperformed QUAD, although differences in performance between the two algorithms were, on the whole, minimized. Table 2 shows that TTEF was able to solve 6 more instances than QUAD; on the J30 set, QUAD was more often the fastest algorithm, while on the J60, J90, and BL sets TTEF tended to be faster. On 36 instances of PSPLib and 34 instances of BL, TTEF had a search tree with a lower node count, while there were no instances for which QUAD had a lower count.

Figure 8b, shows the distribution of the comparative running times of QUAD and TTEF. Similar to the dynamic branching, the distribution of speedups is skewed, with a peak just above 1 (representing cases where QUAD was slightly faster than TTEF) and a longer tail to the left (cases where TTEF was faster by a somewhat larger margin). On the whole, TTEF appears to be faster, but by a relatively narrow margin—in almost every case, the runtime of the two algorithms differed by no more than a factor of 2, regardless of the branching strategy selected.

10.4 Number of propagations

In Section 7 we discuss the possibility that QUAD will in some cases require up to n iterations to accomplish the same pruning as a single iteration of THETA. In Section 9.1 we argue that, in practice, additional iterations should be rare.

To test the hypothesis that Algorithm 1 generally makes the strongest edge-finding inference in the first iteration, we used an instrumented version of the Gecode CUMULATIVE propagator to observe the number of times the propagator was called for finding the optimal solution to each instance. Figure 10 summarizes the results. Both QUAD and TTEF required the same number of propagations as THETA in the vast majority of instances. In the small portion of cases where a different number of propagations was recorded, the difference was quite small. In fact, the possibility of earlier stopping for the "lazy" propagators than for THETA appears, in some instances, to be advantageous, allowing simpler, faster propagators for the remaining constraints in the problem to calculate the fixpoint without reference to

Table 2 Number of instances in which each algorithm found the optimal solution (solve), did so in the fastest time (time), and generated the smallest search tree (nodes), using static branching

	J30			J60			J90			BL		
	Solve	Time	Nodes	Solve	Time	Nodes	Solve	Time	Nodes	Solve	Time	Nodes
THETA	340	0	317	311	0	301	316	0	309	33	0	1
QUAD	345	191	319	312	105	301	316	71	309	35	3	1
TTEF	348	157	343	315	211	308	316	245	314	35	32	35

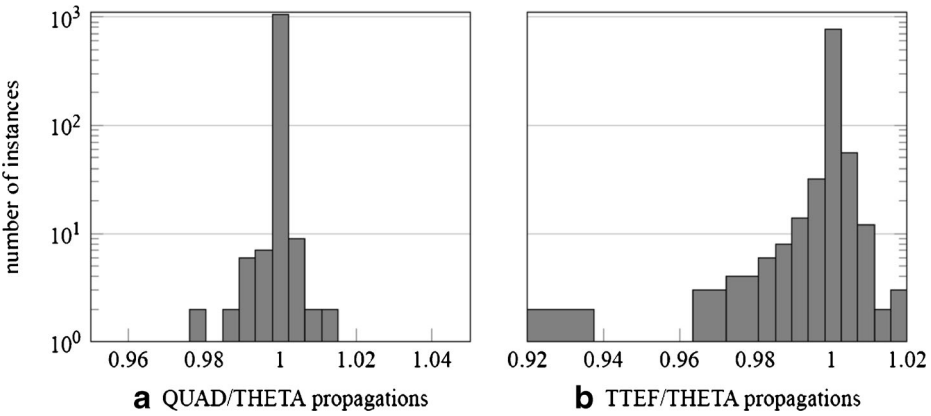

Fig. 10 Comparison of propagation count for THETA vs. **a** QUAD, and **b** TTEF. Only instances with identical node counts are reported

the CUMULATIVE propagator, as indicated by the small number of instances where THETA ultimately required the larger number of propagations.

To test this hypothese more directly, we compared the number of edge-finding iterations required to reach an individual fix point when using QUAD versus THETA, the latter being a complete edge-finder (i.e., one which always makes the strongest edge-finding inference in a single iteration). As edge-finding is not idempotent, any edge-finder will sometimes require multiple iterations before reaching a fix point. Tests were varied by number of tasks and by the maximum duration of a single task, while resource capacity and horizon were fixed; for each combination of parameters, 100,000 random E-feasible instances were generated. Propagation for a single cumulative constraint was carried out only until the first fix point was reached. To minimize the influence of the baseline filtering algorithms, the edge-finding algorithm in each case was run repeatedly until it was unable to further strengthen the domain. Each individual iteration of the edge-finding filter was counted (i.e., rather than the number of invocations of the cumulative propagator).

Table 3 Comparison for QUAD versus THETA of the number of iterations required to reach fix point of the cumulative propagator, on randomly generated domains.

n	$p \leq 5$		$p \leq 10$		$p \leq 15$		$p \leq 20$	
	Diff (%)	Mean iter (#)	Diff (%)	Mean iter (#)	Diff (%)	Mean iter (#)	Diff (%)	Mean iter (#)
10	2.56	1.04	3.19	1.02	3.34	1.01	3.22	1.02
20	11.67	1.05	11.14	1.03	8.86	1.03	6.59	1.02
30	21.06	1.07	15.98	1.04	9.85	1.04	6.20	1.03
40	29.16	1.09	17.14	1.05	8.68	1.05	5.67	1.05
50	34.53	1.10	15.99	1.07	7.35	1.06	5.13	1.07

All resources had a capacity of 10 and a horizon of 200. For each combination of number of tasks (n) and maximum duration (p) are shown (*i*) the percentage of instances where *any* difference (diff) in the number of required iterations was observed, and (*ii*) the average number of iterations (mean iter) by which the two algorithms differed (excluding identical counts)

Results may be seen in Table 3. The QUAD algorithm required additional iterations most frequently when both the maximum duration of the tasks was quite low in comparison to the horizon, and the number of tasks was high. With a large number of short tasks, there is a greater chance of encountering an arrangement of tasks satisfying Theorem 3, case 2(b). When tasks are generally longer, and therefore have less flexibility in possible start times, the incidence of additional iterations drops substantially. This is significant as these instances are harder in the sense that the start times are more tightly constrained. However, the most substantial result in Table 3 is the number of additional iterations required by QUAD in those instances where the number differs from THETA. Those instances almost exclusively required only a single additional iteration (and never, in our observations, more than 2 additional iterations). This result is far below the theoretical bound on the number of iterations from Theorem 3 (i.e., $n-1$). We conclude that, in practice, the need for additional edge-finding iterations when using QUAD has only a minimal impact on performance.

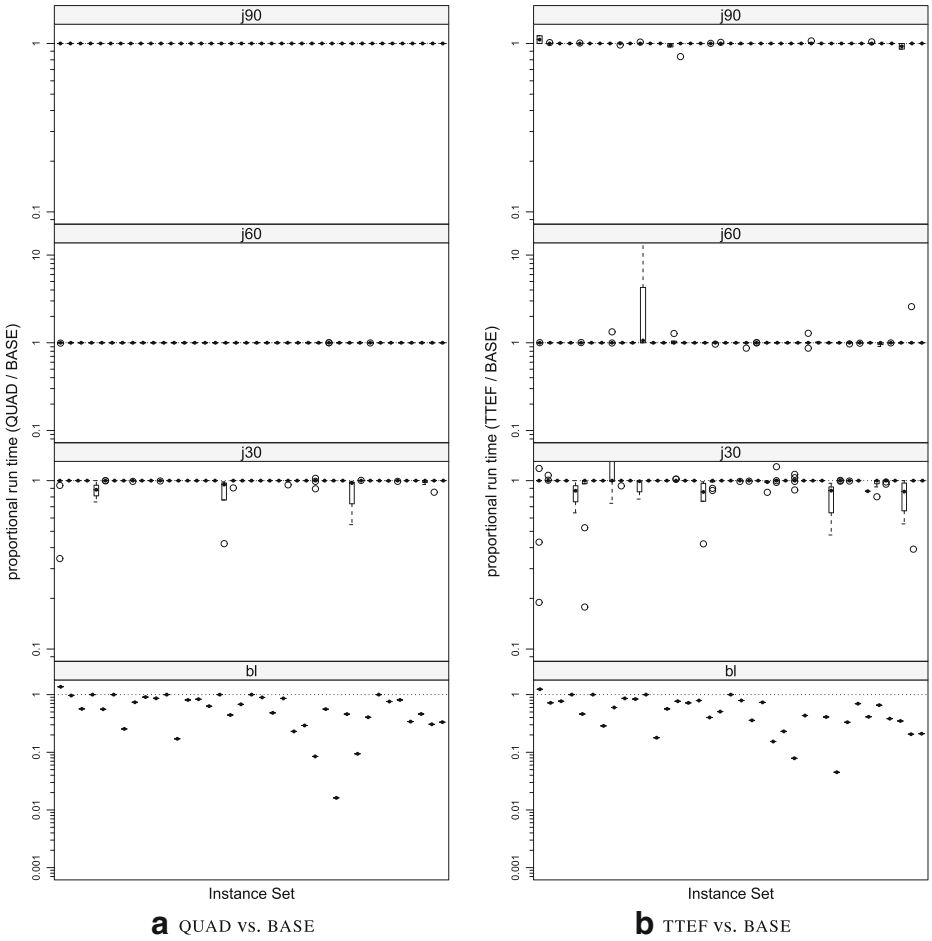

Fig. 11 Comparison of runtimes of QUAD and TTEF, versus the invariant filtering algorithms (BASE), using dynamic branching, sorted by instance. Again, PSPLib instances results are averages over the 10 instances in each parameter setting, while BL instances are reported individually

10.5 Effectiveness of Edge-Finding

Finally, we wished to evaluate the overall effectiveness of edge-finding on these two benchmark sets. Figure 11 shows runtimes for QUAD and TTEF as a proportion of the runtime for the invariant filters (BASE) on the same instances. Results indicate that edge-finding is of limited benefit at best on the tested benchmarks. Runtimes with edge-finding were improved for several J30 instances, and for almost all BL instances (unsurprisingly, given that the BL set was specifically designed to showcase the strengths of edge-finding [2]). For the J60 and J90 instances, QUAD made little, if any, difference. TTEF, on the other hand, did slightly improve runtimes for a few instances; however, it also resulted in significantly increased runtimes in a handful of instances. This last fact appears to account for those cases where QUAD outperformed TTEF on the J60 and J90 instances

11 Conclusion

We have presented an edge-finding filtering algorithm for cumulative scheduling which is quadratic in the number of tasks, improving on the complexity of known standard edge-finding algorithms. This algorithm reaches the same fix point as previous algorithms, possibly after more propagations, although we demonstrate that in practice additional propagations are rarely required. While the complexity of this new algorithm does not strictly dominate that of Vilím's $\mathcal{O}(kn \log n)$ algorithm, experimental results on a standard benchmark suite demonstrate that our algorithm is substantially faster, even in cases with very low values of k. We provide a counterexample to the claim that the standard edge-finding rule is subsumed by the timetable edge finding rule [19], and demonstrate empirically that, while timetable edge-finding is faster in most cases, it was slower on several benchmark instances.

Future work will focus on finding a complete quadratic edge-finder, a similar algorithm for extended edge-finding, and investigating the use of Θ-trees to increase the efficiency of finding maximum density.

Acknowledgments The authors would like to thank Christian Schulte, Guido Tack, and Pierre Flener for their advice and assistance. We also wish to sincerely thank our editors and anonymous referees, who provided a tremendous amount of constructive and insightful feedback. Additionally, the third author is supported by grant 2009-4384 of the Swedish Research Council (VR).

References

1. Aggoun, A., & Beldiceanu, N. (1993). Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, *17*(7), 57–73.
2. Baptiste, P., & Le Pape, C. (2000). Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints*, *5*(1), 119–139.
3. Baptiste, P., Le Pape, C., Nuijten, W.P.M. (2001). *Constraint-based scheduling: applying constraint programming to scheduling problems.* Berlin: Springer.
4. Carlier, J., & Pinson, E. (1994). Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, *78*(2), 146–161.
5. Caseau, Y., & Laburthe, F. (1994). Improved CLP scheduling with task intervals. In P. Van Hentenryck (Ed). *ICLP 1994—logic programming* (pp. 369–383). Cambridge: MIT Press.
6. Gecode. http://www.gecode.org. Accessed 30 Aug 2012.
7. Kameugne, R., & Fotso, L.P. (2013). A cumulative not-first/not-last filtering algorithm in $\mathcal{O}(n^2 \log(n))$. *Indian Journal of Pure Applied Mathematics*, 44(1), 95–115 Springer, Berlin. doi:10.1007/s13226-013-0005-z.

8. Kameugne, R., Fotso, L.P., Scott, J., Ngo-Kateu, Y. (2011). A quadratic edge-finding filtering algorithm for cumulative resource constraints. In J.H.M. Lee (Ed.), *CP 2011—principles and practice of constraint programming, LNCS* (Vol. 6876, pp. 478–492). Berlin: Springer.
9. Kolisch, R., & Sprecher, A. (1996). PSPLIB–a project scheduling library. *European Journal of Operational Research*, *96*, 205–216.
10. Mercier, L., & Van Hentenryck, P. (2008). Edge finding for cumulative scheduling. *INFORMS Journal on Computing*, *20*(1), 143–153.
11. Nuijten, W.P.M. (1994). *Time and resource constrained scheduling*. PhD thesis. Technische Universiteit Eindhoven.
12. *PSPLib—project scheduling problem library*. http://129.187.106.231/psplib/.
13. Schulte, C., Tack, G., Lagerkvist, M.Z. (2012). Modeling. In C. Schulte, G. Tack, M.Z. Lagerkvist (Eds.), *Modeling and programming with Gecode*. Corresponds to Gecode 3.7.3.
14. Schutt, A., & Wolf, A. (2010). A new $O(n^2 \log n)$ not-first/not-last pruning algorithm for cumulative resource constraints. In D. Cohen (Ed.), *CP 2010—principles and practice of constraint programming, LNCS* (Vol. 6308, pp. 445–459). Berlin: Springer.
15. Scott, J. (2010). *Filtering algorithms for discrete cumulative resources*. Masters Thesis, Uppsala University, Sweden. http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-132172.
16. Vilím, P. (2007). *Global constraints in scheduling*. PhD thesis, Charles University in Prague.
17. Vilím, P. (2009). Edge finding filtering algorithm for discrete cumulative resources in $O(kn \log n)$. In I.P. Gent (Ed.), *CP 2009—principles and practice of constraint programming, LNCS* (Vol. 5732, pp. 802–816). Berlin: Springer.
18. Vilím, P. (2009). Max energy filtering algorithm for discrete cumulative resources. In W.J. van Hoeve, & J.N. Hooker (Eds.), *CPAIOR 2009—integration of AI and OR techniques in constraint programming, LNCS* (Vol. 5547, pp. 294–308). Berlin: Springer.
19. Vilím, P. (2011). Timetable edge finding filtering algorithm for discrete cumulative resources. In T. Achterberg & J.C. Beck (Eds.), *CPAIOR 2011—integration of AI and OR techniques in constraint programming, LNCS* (Vol. 6697, pp. 230–245). Berlin: Springer.
20. Wolf, A., & Schrader, G. (2006). $O(n \log n)$ overload checking for the cumulative constraint and its application. In M. Umeda, A. Wolf, O. Bartenstein, U. Geske, D. Seipel, O. Takata (Eds.), *INAP 2005—applications of declarative programming for knowledge management, LNCS* (Vol. 4369, pp. 88–101). Berlin: Springer.