A Scalable Sweep Algorithm for the *cumulative* Constraint

Arnaud Letort[1,⋆], Nicolas Beldiceanu[1], and Mats Carlsson[2]

[1] TASC team, (EMN-INRIA,LINA) Mines de Nantes, France
{arnaud.letort,nicolas.beldiceanu}@mines-nantes.fr
[2] SICS, P.O. Box 1263, SE-164 29 Kista, Sweden
matsc@sics.se

Abstract. This paper presents a sweep based algorithm for the *cumulative* constraint, which can operate in filtering mode as well as in greedy assignment mode. Given n tasks, this algorithm has a worst-case time complexity of $O(n^2)$. In practice, we use a variant with better average-case complexity but worst-case complexity of $O(n^2 \log n)$, which goes down to $O(n \log n)$ when all tasks have unit duration, i.e. in the bin-packing case. Despite its worst-case time complexity, this algorithm scales well in practice, even when a significant number of tasks can be scheduled in parallel. It handles up to 1 million tasks in one single *cumulative* constraint in both Choco and SICStus.

1 Introduction

In the 2011 Panel of the Future of CP [6], one of the identified challenges for CP was the need to handle large scale problems. Multi-dimensional bin-packing problems were quoted as a typical example [10], particularly relevant in the context of cloud computing. Indeed the importance of bin-packing problems was recently highlighted in [12] and is part of the topic of the 2012 Roadef Challenge [13].

Till now, the tendency is to use dedicated algorithms and metaheuristics [17] to cope with large instances. Following the line of research initiated with the *geost* constraint [2], our main objective is to provide global constraints that can handle a significant sub-problem while scaling well in a traditional CP solver. Typically, filtering algorithms focus on having the best possible deductions [9,20,21], rather than on scalability issues. This explains why all existing papers on *cumulative* [9,16,20,21] and *bin-packing* [18,7,14] usually focus on small size problems (i.e., typically less than 200 tasks up to 10000 tasks) but leave open the scalability issue. Like what was already done for the *geost* constraint, which handles up to 2 million boxes, our goal is to come up with a lean filtering algorithm for *cumulative*. In order to scale well in terms of memory, we design a lean filtering algorithm, which can also be turned into a greedy algorithm that benefits from the filtering of the lean filtering algorithm while fixing tasks. This approach

⋆ Partially founded by the SelfXL project (contract ANR-08-SEGI-017).

M. Milano (Ed.): CP 2012, LNCS 7514, pp. 439–454, 2012.

allows to avoid the traditional memory bottleneck problem of CP solvers due to trailing or copying data structures [15], while still benefitting from filtering. Moreover, like for *geost* our lean filtering algorithm and its derived greedy assignment mode are compatible in the sense that they can both be used at each node of the search tree, i.e., first call the greedy mode for trying to find a solution and, if that doesn't work, use the filtering mode to restrict the variables and continue the search.

This paper focuses on the *cumulative* constraint, originally introduced in [1] for modeling resource scheduling problems:

$$cumulative([s_0, \ldots, s_{n-1}], [d_0, \ldots, d_{n-1}], [e_0, \ldots, e_{n-1}], [h_0, \ldots, h_{n-1}], limit)$$

where $[s_0, \ldots, s_{n-1}]$, $[e_0, \ldots, e_{n-1}]$ are non-empty lists of domain variables,[1] and $[d_0, \ldots, d_{n-1}]$, $[h_0, \ldots, h_{n-1}]$ are lists of non-negative integers and *limit* is a non-negative integer. The *cumulative* constraint holds if (1-2) are true:

$$\forall t \in [0, n-1] \ : s_t + d_t = e_t \tag{1}$$

$$\forall i \in \mathbb{N} \ : \sum_{\substack{t \in [0,n-1]: \\ i \in [s_t, e_t)}} h_t \leq limit \tag{2}$$

Section 2 recalls the 2001 sweep algorithm for *cumulative* [3] and provides a critical analysis of its major bottlenecks. Then, Section 3 presents the new sweep based filtering algorithm and its greedy mode. Section 4 evaluates its implementations in both Choco [19] and SICStus [5] and compares them with the 2001 implementations [3] in both systems, as well as to a dedicated *bin-packing* constraint used in Entropy [8].

2 A Critical Analysis of the 2001 Sweep Algorithm

The algorithm is based on the *sweep* idea, which is widely used in computational geometry [4]. In constraint programming, sweep was used for implementing the *non-overlapping* constraint [2] as well as the *cumulative* constraint [3].

In 2 dimensions, a plane *sweep* algorithm solves a problem by moving a vertical line from left to right. The algorithm uses two data structures:

- The *sweep-line status*, which contains some information related to the current position δ of the vertical line.
- The *event point series*, which holds the events to process, ordered in increasing order according to the abscissa.

The algorithm initializes the sweep-line status for the starting position of the vertical line. Then the line "jumps" from event to event; each event is handled and inserted or removed from the sweep-line status. In our context, the sweep-line

[1] A *domain variable* v is a variable that ranges over a finite set of integers; $\underline{v}$ and $\overline{v}$ respectively denote the minimum and maximum value of variable v.

scans the time axis in order to build a pessimistic cumulated resource consumption profile (PCRCP) and to perform checks and pruning according to this profile and to *limit*. So the algorithm is a sweep variant of the *timetable* method [11]. Before defining the notion of PCRCP let us first introduce a running example that will be used throughout for illustrating the different algorithms.

Example 1. Consider four tasks t_0, t_1, t_2, t_3 which have the following *start*, *duration*, *end* and *height*:

- $t_0 : s_0 = 0,\ d_0 = 1,\ e_0 = 1,\ h_0 = 3,$
- $t_1 : s_1 \in [0, 2],\ d_1 = 2,\ e_1 \in [2, 4],\ h_1 = 3,$
- $t_2 : s_2 \in [2, 4],\ d_2 = 3,\ e_2 \in [5, 7],\ h_2 = 3,$
- $t_3 : s_3 \in [5, 7],\ d_3 = 1,\ e_3 \in [6, 8],\ h_3 = 3,$

subject to the constraint $\mathit{cumulative}([s_0, s_1, s_2, s_3], [d_0, d_1, d_2, d_3], [e_0, e_1, e_2, e_3], [h_0, h_1, h_2, h_3],\ 5)$. Since task t_0 starts at instant 0 and since t_1 cannot overlap t_0 without exceeding the resource limit 5, the earliest start of t_1 is adjusted to 1. Since task t_1 occupies interval $[2, 3)$ and since t_1 and t_2 also cannot overlap for the same reason, the earliest start of t_2 is adjusted to 3. Since task t_2 occupies interval $[4, 6)$ and since t_2 and t_3 also cannot overlap, the earliest start of t_3 is adjusted to 6. The purpose of the sweep algorithm is to perform such filtering in an efficient way. □

Given a set of tasks $\mathcal{T}$, the PCRCP of the set $\mathcal{T}$ consists of the aggregation of the compulsory parts of the tasks in $\mathcal{T}$, where the *compulsory part* of a task is the intersection of all its feasible instances. On the one hand, the height of the compulsory part of a task t at a given time point i is defined by h_t if $i \in [\overline{s_t}, \underline{e_t})$ and 0 otherwise. On the other hand, the height of the PCRCP at a given time point i is given by $\sum_{\substack{t \in \mathcal{T},\\ i \in [\overline{s_t}, \underline{e_t})}} h_t$.

Continuation of Example 1 (Compulsory Part of a Task). Task t_0 and t_2 initially have a non-empty compulsory part: task t_0 uses 3 resource units on interval $[0, 1)$, while task t_2 uses 3 resource units on interval $[4, 5)$. After reaching the fixpoint, task t_1 also has a non-empty compulsory part: task t_1 uses 3 resource units on interval $[2, 3)$ while the compulsory part of t_2 now occupies interval $[4, 6)$. □

Event Point Series. In order to build the PCRCP and to prune the start of the tasks, the sweep algorithm considers the following types of events:

- *Profile events* for building the PCRCP correspond to the latest starts and the earliest ends of the tasks for which the latest start is strictly less than the earliest end (i.e. the start and the end of a non-empty compulsory part).
- *Pruning events* for recording the tasks to prune, i.e. the not yet fixed tasks that intersect δ.

Table 1 (top) describes the different types of events, where each event corresponds to a quadruple $\langle$*event type*, *task generating the event*, *event date*, *available space increment*$\rangle$. These events are sorted by increasing date.

Continuation of Example 1 (Generated Events). The following events are generated and sorted by increasing date: $\langle SCP, 0, 0, -3\rangle$, $\langle PR, 1, 0, 0\rangle$, $\langle ECP, 0, 1, 3\rangle$, $\langle PR, 2, 2, 0\rangle$, $\langle SCP, 2, 4, -3\rangle$, $\langle ECP, 2, 5, 3\rangle$, $\langle PR, 3, 5, 0\rangle$. □

Table 1. Event types for the 2001 sweep (top) and the dynamic sweep (bottom) with corresponding condition for generating them. The last event attribute is only relevant for event types *SCP*, *ECP* and *ECPD*.

Generated Events (2001 algo.)	**Conditions**
$\langle SCP, t, \overline{s_t}, -h_t\rangle$ and $\langle ECP, t, \underline{e_t}, +h_t\rangle$	$\overline{s_t} < \underline{e_t}$
$\langle PR, t, \underline{s_t}, 0\rangle$	$\underline{s_t} \neq \overline{s_t}$

New Events	**Events (2001 algo.)**	**Conditions**
$\langle SCP, t, \overline{s_t}, -h_t\rangle$	$\langle SCP, t, \overline{s_t}, -h_t\rangle$	$\overline{s_t} < \underline{e_t}$
$\langle ECPD, t, \underline{e_t}, +h_t\rangle$	$\langle ECP, t, \underline{e_t}, +h_t\rangle$	$\overline{s_t} < \underline{e_t}$
$\langle CCP, t, \overline{s_t}, 0\rangle$		$\overline{s_t} \geq \underline{e_t}$
$\langle PR, t, \underline{s_t}, 0\rangle$	$\langle PR, t, \underline{s_t}, 0\rangle$	$\underline{s_t} \neq \overline{s_t}$

Sweep-Line Status. The sweep-line maintains three pieces of information:

- The current sweep-line position δ, initially set to the date of the first event.
- The amount of available resource at instant δ, denoted by *gap*, i.e., the difference between the resource limit and the height of the PCRCP.
- A list of tasks $\mathcal{T}_{prune}$, recording all tasks that potentially can overlap δ.

The sweep algorithm first creates and sorts the events wrt. their date. Then, the sweep-line moves from one event to the next event, updating *gap* and $\mathcal{T}_{prune}$. Once all events at δ have been handled, the sweep algorithm tries to prune all tasks in $\mathcal{T}_{prune}$ wrt. *gap* and interval $[\delta, \delta')$ where δ' is the next sweep-line position, i.e. the date of the next event. More precisely, given a task $t \in \mathcal{T}_{prune}$ such that $h_t > gap$, the interval $[\delta - d_t + 1, \delta')$ is removed from the start of task t.

Continuation of Example 1 (Illustrating the 2001 Sweep Algorithm). The sweep algorithm reads the two events $\langle SCP, 0, 0, -3\rangle$, $\langle PR, 1, 0, 0\rangle$ and sets *gap* to $5-3$ and $\mathcal{T}_{prune}$ to $\{t_1\}$. During a first sweep, the compulsory part of task t_0 (see Part (A) of Figure 1) permits to prune the start of t_1 since the *gap* on $[0, 1)$ is strictly less than h_1.The pruning of the earliest start of t_1 during the first sweep causes the creation of a compulsory part for task t_1 which is not immediately used to perform more pruning (see Part (B)). It is necessary to wait for a second sweep to take advantage of this new compulsory part to adjust the earliest start of task t_2. This last adjustment causes the extension of the compulsory part of t_2 on $[4, 6)$ (see Part (C)). A third sweep adjusts the earliest start of task t_3 which cannot overlap t_2. A fourth and last sweep is performed to find out that the fixpoint was reached (see Part (D)). □

Weakness of the 2001 Sweep Algorithm

① `[Too static]` The potential increase of the PCRCP during a single sweep is not dynamically taken into account. In other words, creations and extensions of compulsory parts during a sweep are not immediately used to perform more pruning while sweeping. Example 1 illustrates this point since the sweep needs to be run four times before reaching its fixpoint.

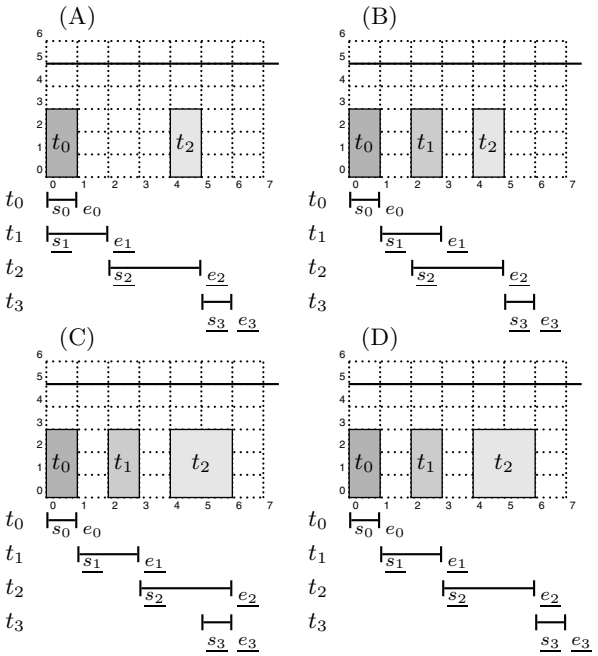

Fig. 1. Parts (A), (B), (C) and (D) respectively represent the earliest positions of the tasks and the PCRCP, of the initial problem described in Example 1, after a first sweep, after a second sweep and after a third sweep

② `[Often reaches its worst-case time complexity]` The worst-case time complexity of the 2001 sweep algorithm is $O(n^2)$ where n is the number of tasks. This complexity is often reached in practice when most of the tasks can be placed everywhere on the time line. The reason is that it needs at each value of δ to systematically re-scan all tasks that overlap δ. Profiling the 2001 implementation indicates that the sweep algorithm spends up to 45% of its overall running time scanning the list of potential tasks to prune.

③ `[Creates holes in the domains]` The 2001 sweep algorithm removes intervals of consecutive values from domain variables. This is a weak point, which prevents handling large instances since a variable cannot just be compactly represented by its minimum and maximum values.

④ `[Does not take advantage of bin-packing]` For instances where all tasks have duration one, the worst time complexity $O(n^2)$ is left unchanged.

3 The Dynamic Sweep Algorithm

This section presents our contribution, a new sweep algorithm that handles the four performance issues of the 2001 sweep algorithm raised at the end of Sect.2, i.e., points ① to ④. We first introduce some general design decisions of the new sweep algorithm as well as the property the algorithm maintains, and then

describe it in a similar way the 2001 original sweep algorithm was presented in Sect. 2. We first present the new *event point series*, then the new *sweep-line status*, and the overall algorithm. Finally we prove that the property initially introduced is maintained by the new algorithm and give its complexity in the general case as well as in the case where all task durations are fixed to one.

The first difference from the 2001 sweep is that our algorithm only deals with domain bounds, which is a good way to reduce the memory consumption for the representation of domain variables (see Point ③ of Sect. 2).[2] Consequently, we need to change the 2001 algorithm, which creates holes in the domain of task origins. The new sweep algorithm filters the task origins in two distinct sweep stages. A first stage, called *sweep_min*, tries to adjust the earliest starts of tasks by performing a sweep from left to right, and a second stage tries to adjust the latest ends by performing a sweep from right to left. The greedy mode of the new sweep algorithm will be derived from *sweep_min*, in the sense that it takes advantage of the propagation performed by *sweep_min* and fixes the start of tasks rather than adjusting them. W.l.o.g, we focus from now on the first stage *sweep_min* since the second stage is completely symmetric.

As illustrated by Example 1, the 2001 sweep algorithm needs to be re-run several times in order to reach its fixpoint (i.e., 4 times in our example). This is due to the fact that, during one sweep, restrictions on task origins are not immediately taken into account. Our new algorithm, *sweep_min*, dynamically uses these deductions to reach its fixpoint in one single sweep. To deal with this aspect, our new sweep algorithm introduces the concept of *conditional events*, i.e., events that are created while sweeping over the time axis.

We first give the property that holds when *sweep_min* reaches its fixpoint. This property will be proved at the end of this section.

Property 1. Given a *cumulative* constraint with its set of tasks $\mathcal{T}$ and its resource limit *limit*, *sweep_min* ensures that:

$$\forall t \in \mathcal{T}, \forall i \in [\underline{s_t}, \underline{e_t}) \ : \ h_t + \sum_{\substack{t' \in \mathcal{T} \setminus \{t\}: \\ i \in [\underline{s_{t'}}, \underline{e_{t'}})}} h_{t'} \leq \mathit{limit} \tag{3}$$

Property 1 ensures that, for any task t of the *cumulative* constraint, one can schedule t at its earliest start without exceeding the resource limit wrt. the PCRCP for the tasks of $\mathcal{T} \setminus \{t\}$. We now present the different parts of the new sweep algorithm.

3.1 Event Point Series

In order to address point ① `[Too static]` of Sect. 2, *sweep_min* should handle the extension and the creation of compulsory parts caused by the adjustment of earliest starts of tasks in one single sweep. We therefore need to modify the

[2] Note that most Operation Research scheduling algorithms only adjust the earliest start and latest ends of tasks.

events introduced in Table 1. The bottom part presents the events of *sweep_min* and their relations with the events of the 2001 algorithm.

- The event type $\langle SCP, t, \overline{s_t}, -h_t \rangle$ for the start of compulsory part of task t is left unchanged. Note that, since *sweep_min* only adjusts earliest starts, the start of a compulsory part (which corresponds to a latest start) can never be extended to the left.
- The event type $\langle ECP, t, \underline{e_t}, h_t \rangle$ for the end of the compulsory part of task t is converted to $\langle ECPD, t, \underline{e_t}, h_t \rangle$ where D stands for *dynamic*. The date of such event corresponds to the earliest end of t (also the end of its compulsory part) and may increase due to the adjustment of the earliest start of t.
- A new event type $\langle CCP, t, \overline{s_t}, 0 \rangle$, where *CCP* stands for *conditional compulsory part*, is created for each task t that does not have any compulsory part. At the latest, once the sweep-line reaches position $\overline{s_t}$, it adjusts the earliest start of t. Consequently the conditional event can be transformed into an *SCP* and an *ECPD* events, reflecting the creation of compulsory part.
- The event type $\langle PR, t, \underline{s_t}, 0 \rangle$ for the earliest start of t is left unchanged.

On the one hand, some of these events have their dates modified (see *ECPD*). On the other hand, some events create new events (see *CCP*). Consequently, rather than just sorting all events initially, we insert them by increasing date into a heap called h_{events}.

Continuation of Example 1 (New Generated Events for sweep_min*).* The following events are generated and sorted according to their date: $\langle SCP, 0, 0, -3 \rangle$, $\langle PR, 1, 0, 0 \rangle$, $\langle \mathbf{ECPD}, \mathbf{0}, \mathbf{1}, \mathbf{3} \rangle$, $\langle \mathbf{CCP}, \mathbf{1}, \mathbf{2}, \mathbf{0} \rangle$, $\langle PR, 2, 2, 0 \rangle$, $\langle SCP, 2, 4, -3 \rangle$, $\langle \mathbf{ECPD}, \mathbf{2}, \mathbf{5}, \mathbf{3} \rangle$, $\langle PR, 3, 5, 0 \rangle$, $\langle \mathbf{CCP}, \mathbf{3}, \mathbf{7}, \mathbf{0} \rangle$. □

3.2 Sweep-Line Status

The sweep-line maintains the following pieces of information:

- The current sweep-line position δ, initially set to the date of the first event.
- The amount of available resource at instant δ, denoted by *gap*, i.e., the difference between the resource limit and the height of the PCRCP.
- Two heaps $h_{conflict}$ and h_{check} for partially avoiding point ② `[Often reaches its worst-case time complexity]` of Sect. 2. W.l.o.g. assume that the sweep-line is at its initial position and that we handle an event of type *PR* (i.e., we try to find out the earliest possible start of a task t).
  - If the height of task t is strictly greater than the available gap at δ, we know that we have to adjust the earliest start of t. In order to avoid re-checking each time we move the sweep-line whether or not the gap is big enough wrt. h_t, we say that t is in conflict with δ. We insert task t in the heap $h_{conflict}$, which records all tasks that are in conflict with δ, sorted by increasing height, i.e. the top of the heap $h_{conflict}$ corresponds to the smallest value. This order is induced by the fact that, if we need to adjust the earliest start of a task t, all earliest task starts with a height greater than or equal to h_t also need to be adjusted.

- If the height of task t is less than or equal to the available gap at δ, we know that the earliest start of task t could be equal to δ. But to be sure, we need to check Property 1 for t (i.e., $\mathcal{T} = \{t\}$). For this purpose we insert t in the heap h_{check}, which records all tasks for which we currently check Property 1. Task t stays in h_{check} until a conflict is detected (i.e., h_t is greater than the available gap, and t goes back in $h_{conflict}$) or until the sweep-line passes instant $\delta + d_t$ (and we have found a feasible earliest start of task t wrt. Property 1). In the heap h_{check}, tasks are sorted by decreasing height, i.e. the top of the heap h_{check} corresponds to the largest value, since if a task t is not in conflict with δ, all other tasks of h_{check} of height less than or equal to h_t are also not in conflict with δ.
  In the following, *empty(h)* returns *true* if the heap h is empty, *false* otherwise. Function *get_top_key(h)* returns the key of the top element in the heap h. We introduce an array of integers *mins*, which stores for each task t in h_{check} the value of δ when t was added into h_{check}.

3.3 Algorithm

The *sweep_min* algorithm performs one single sweep over the event point series in order to adjust the earliest start of the tasks wrt. Property 1. It consists of a main loop, a filtering part and a synchronization part. This last part is required in order to directly handle the deductions attached to the creation or increase of compulsory parts in one single sweep. In addition to the heaps h_{check} and $h_{conflict}$ we introduce an array of booleans *evup* for which the t^{th} entry indicates whether events related to the compulsory part of t were updated or not. It is set to true once we have found the final values of the start and end of the compulsory part of t. We introduce a list *newActiveTasks*, which records all tasks that have their *PR* event at δ. The primitive *adjust_min_start*(t, v) adjusts the minimum value of the start variable of task t to value v.

Main Loop. The main loop (Algorithm 1) consists of:

- `[INITIALIZATION]` (lines 3 to 5). The events are generated and inserted into h_{events} according to the conditions given in Table 1. The h_{check} and $h_{conflict}$ heaps are initialized as empty heaps. The list *newActiveTasks* is initialized as an empty list. δ is set to the date of the first event.
- `[MAIN LOOP]` (lines 7 to 24). For each date the main loop processes all the corresponding events. It consists of the following parts:
  - `[HANDLING A SWEEP-LINE MOVE]` (lines 9 to 16). Each time the sweep-line moves, we update the sweep-line status (h_{check} and $h_{conflict}$) wrt. the new *active tasks*, i.e. the tasks for which the earliest start is equal to δ. All the new active tasks that are in conflict with δ in the PCRCP are added into $h_{conflict}$ (lines 9 and 10). For tasks that are not in conflict we check whether the sweep interval $[\delta, \delta')$ is big enough wrt. their durations. Tasks for which the sweep interval is too small are added into h_{check} (line 10).

```
 1: function sweep_min(n, s_[0..n-1], s̄_[0..n-1], e_[0..n-1], d_[0..n-1], h_[0..n-1]) : boolean
 2: [INITIALIZATION]
 3: h_events ← generation of events wrt. n, s_t, s̄_t, d, e_t and h and Table 1.
 4: h_check, h_conflict ← ∅; newActiveTasks ← ∅
 5: δ ← get_top_heap(h_events); δ' ← δ; gap ← limit
 6: [MAIN LOOP]
 7: while ¬empty(h_events) do
 8:    [HANDLING A SWEEP-LINE MOVE]
 9:    if δ ≠ δ' then
10:       while ¬empty(newActiveTasks) do
11:          extract first task t from newActiveTasks
12:          if h_t > gap then add ⟨h_t, t⟩ in h_conflict
13:          else if d_t > δ' − δ then add ⟨h_t, t⟩ in h_check; mins_t ← δ
14:          else evup_t ← true
15:       if ¬filter_min(δ, δ', gap) then return false
16:       δ ← δ'
17:    [HANDLING CURRENT EVENT]
18:    δ ← synchronize(h_events, δ)
19:    extract ⟨type, t, δ, dec⟩ from h_events
20:    if type = SCP ∨ type = ECPD then gap ← gap + dec
21:    else if type = PR then newActiveTasks ← newActiveTasks ∪ {t}
22:    [GETTING NEXT EVENT]
23:    if empty(h_events) ∧ ¬filter_min(δ, +∞, gap) then return false
24:    δ' ← synchronize(h_events, δ)
25: return true
```

Algorithm 1. False if a resource overflow is detected, true otherwise.

Then *filter_min* (see Alg. 2) is called to update h_{check} and $h_{conflict}$ and to adjust the earliest start of tasks for which a feasible position was found.

- `[HANDLING CURRENT EVENT]` (lines 18 to 21). Conditional events (*CCP*) and dynamic events (*ECPD*) at the top of h_{events} are processed (see Alg. 3). The top event is extracted from the heap h_{events}. Depending of its type (i.e., *SCP* or *ECPD*), the gap of the available resource is updated, or (i.e., *PR*), the task is added into the list of new active tasks.
- `[GETTING NEXT EVENT]` (lines 23 to 24). If there is no more event in h_{events}, *filter_min* is called in order to empty the heap h_{check}, which may generate new compulsory part events.

The Filtering Part. Algorithm 2 processes tasks in h_{check} and $h_{conflict}$ in order to adjust the earliest start of the tasks. The main parts of the algorithm are:

- `[CHECK RESOURCE OVERFLOW]` (line 3). If the available resource *gap* is negative on the sweep interval $[\delta, \delta')$, Alg. 2 returns false meaning a failure (i.e. the resource capacity limit is exceeded).
- `[UPDATING TOP TASKS OF` h_{check} `]` (lines 5 to 11). All tasks in h_{check} of height greater than the available resource *gap* are extracted.

```
 1: function filter_min(δ, δ', gap) : boolean
 2: [CHECK RESOURCE OVERFLOW]
 3: if gap < 0 then return false
 4: [UPDATING TOP TASKS OF h_check ]
 5: while ¬empty(h_check) ∧ (empty(h_events) ∨ get_top_key(h_check) > gap) do
 6:    extract ⟨h_t, t⟩ from h_check
 7:    if δ ≥ s̄_t ∨ δ − mins_t ≥ d_t ∨ empty(h_events) then
 8:       adjust_min_start(t, mins_t)
 9:       if ¬evup_t then update events of the compulsory part of t; evup_t ← true
10:    else
11:       add ⟨h_t, t⟩ in h_conflict
12: [UPDATING TOP TASKS OF h_conflict ]
13: while ¬empty(h_conflict) ∧ get_top_key(h_conflict) ≤ gap do
14:    extract ⟨h_t, t⟩ from h_conflict
15:    if δ ≥ s̄_t then
16:       adjust_min_start(t, s̄_t)
17:       if ¬evup_t then update events of the compulsory part of t; evup_t ← true
18:    else
19:       if δ' − δ ≥ d_t then
20:          adjust_min_start(t, δ)
21:          if ¬evup_t then update events of the compulsory part of t; evup_t ← true
22:       else
23:          add ⟨h_t, t⟩ in h_check; mins_t ← δ
24: return true
```

Algorithm 2. Tries to adjust earliest starts of tasks in h_{check} and $h_{conflict}$ wrt. the sweep interval $[\delta, \delta')$ and the available resource *gap* and returns false if a resource overflow is detected, true otherwise.

- A first case to consider is when task t has been in h_{check} long enough (i.e. $\delta - mins_t \geq d_t$, line 7), meaning that the task is not in conflict on interval $[mins_t, \delta)$, whose size is greater than or equal to d_t. Consequently, we adjust the earliest start of task t to value $mins_t$.
- A second case to consider is when δ has passed the latest start of task t (i.e. $\delta \geq \overline{s_t}$, line 7). That means task t was not in conflict on interval $[mins_t, \delta)$ either, and we can adjust its earliest start to $mins_t$.
- A third case is when there is no more event in the heap h_{events} (i.e. empty(h_{events}), line 7). It means that the height of the PCRCP is equal to zero and we need to empty h_{check}.
- Otherwise, the task is added into $h_{conflict}$ (line 11).

– `[UPDATING TOP TASKS OF` $h_{conflict}$ `]` (lines 13 to 23). All tasks in $h_{conflict}$ that are no longer in conflict with δ are extracted. If δ has passed the latest start of task t, we know that t cannot be scheduled before its latest position. Otherwise, we compare the duration of t with the sweep interval and decide whether to adjust the earliest start of t or to add it into h_{check}.

```
 1: function synchronize(h_events, δ) : integer
 2: [UPDATING TOP EVENTS]
 3: repeat
 4:    if empty(h_events) then return −1
 5:    sync ← true; ⟨date, t, type, dec⟩ ← consult top event of h_events
 6:    [PROCESSING DYNAMIC EVENT]
 7:    if type = ECPD ∧ ¬evup_t then
 8:       if t ∈ h_check then update event date to mins_t + d_t
 9:       else update event date to s̄_t + d_t
10:       evup_t ← true; sync ← false;
11:       [PROCESSING CONDITIONAL EVENT]
12:    else if type = CCP ∧ ¬evup_t ∧ date = δ then
13:       if t ∈ h_check ∧ mins_t + d_t > δ then
14:          add ⟨SCP, t, δ, −h_t⟩ and ⟨ECPD, t, mins_t + d_t, h_t⟩ into h_events
15:       else
16:          add ⟨SCP, t, δ, −h_t⟩ and ⟨ECPD, t, ē_t, h_t⟩ into h_events
17:       evup_t ← true; sync ← false;
18: until sync
19: return date
```

Algorithm 3. Checks that the event at the top of h_{events} is updated and returns the date of the next event or *null* if h_{events} is empty.

The Synchronization Part. Before each extraction or access to h_{events}, Alg. 3 checks and updates the top event and returns the next event date. The main parts of the algorithm are:

- `[UPDATING TOP EVENTS]` (lines 3 to 18). Dynamic and conditional events require to check whether the next event to be extracted by Alg. 1 needs to be updated or not. The repeat loop updates the next event if necessary until the top event is up to date.
- `[PROCESSING DYNAMIC EVENT]` (lines 7 to 10). An event of type *ECPD* must be updated if the related task t is in h_{check} or in $h_{conflict}$. If t is in $h_{conflict}$, it means that t cannot start before its latest starting time $\overline{s_t}$. Consequently, its *ECPD* event is pushed back to the date $\overline{s_t} + d_t$ (line 9). If t is in h_{check}, it means that its earliest start can be adjusted to $mins_t$. Consequently, its *ECPD* event is updated to the date $mins_t + d_t$ (line 8).
- `[PROCESSING CONDITIONAL EVENT]` (lines 12 to 17). When the sweep-line reaches the position of a *CCP* event for a task t, we need to know whether or not a compulsory part for t is created. As $evup_t$ is set to *false*, we know that t is either in h_{check} or in $h_{conflict}$. If t is in $h_{conflict}$ the task is fixed to its latest position and related events are added into h_{events} (line 16). If t is in h_{check}, a compulsory part is created iff $mins_t + d_t > \delta$ (lines 13-14).

Continuation of Example 1 (Illustrating the Dynamic Sweep Algorithm). The sweep algorithm first reads the two events $\langle SCP, 0, 0, -3\rangle$, $\langle PR, 1, 0, 0\rangle$ and sets *gap* to 2. Since the height of task t_1 is greater than the available resource *gap*, t_1 is added into

$h_{conflict}$ (see Alg. 1 line 12 and Fig. 2 Part (A)). The call of *filter_min* only checks that the gap is non-negative. Then, the sweep-line moves to the position 1, reads the event $\langle ECPD, 0, 1, +3\rangle$ and sets *gap* to 5. The call of *filter_min* with $\delta = 1$, $\delta' = 2$ and $gap = 5$ retrieves t_1 from $h_{conflict}$ and inserts it into h_{check} (see Alg. 2, line 23). In *synchronize* (called in Alg. 1 line 24), the next event $\langle CCP, 1, 2, 0\rangle$ is converted into two events $\langle SCP, 1, 2, -3\rangle$ and $\langle ECPD, 1, 3, +3\rangle$ standing for the creation of a compulsory part on interval $[2, 3)$ for the task t_1 (see Fig. 2 Part (B)). Note that the creation of the compulsory part occurs after the sweep-line position, which is key to ensuring Property 1. □

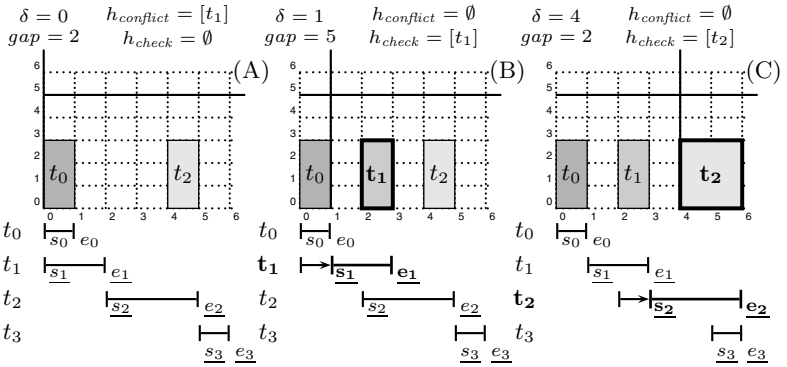

Fig. 2. Parts (A), (B) and (C) represent the earliest positions of the tasks and the PCRCP at different values of δ. Part (A) is when $\delta = 0$ just before the call of *filter_min* (Alg. 1 line 15). Part (B) is when $\delta = 1$ just after the call of *synchronize* (Alg. 1 line 24). Part (C) is when $\delta = 4$ just after the call of *synchronize* (line 24).

3.4 Correctness and Property Achieved by *sweep_min*

We now prove that after the termination of *sweep_min*(Alg. 1), Property 1 holds. For this purpose, we first introduce the following lemma.

Lemma 1. *At any point of its execution,* sweep_min*(Alg. 1) cannot generate a new compulsory part that is located before* δ*.*

Proof. Since the start of the compulsory part of a task t corresponds to $\overline{s_t}$, which is indicated by its *CCP* or *SCP* event, and since *sweep_min* only prunes earliest starts, the compulsory part of t cannot start before this event. Consequently, the latest value of δ to know whether the compulsory part of t is created is $\overline{s_t}$. This case is processed by Alg. 3, lines 12 to 17.

The end of the compulsory part of a task t corresponds to $\underline{e_t}$ and is indicated by its *ECPD* event. To handle its potential extension to the right, the earliest start of t must be found before the sweep extracts its *ECPD* event. This case is processed by Alg. 3, lines 7 to 10. □

Proof (of Property 1). Given a task t, let δ_t and min_t respectively denote the position of the sweep-line when the earliest start of t is adjusted by *sweep_min*, and the new earliest start of t. We successively show the following points:

① When the sweep-line is located at instant δ_t we can start task t at min_t without exceeding *limit*, i.e.

$$\forall t' \in \mathcal{T} \backslash \{t\}, \forall i \in [min_t, \delta_t) : \; h_t + \sum_{\substack{t' \in \mathcal{T} \backslash \{t\}: \\ i \in [\underline{s_{t'}}, \underline{e_{t'}})}} h_{t'} \leq limit$$

The adjustment of the earliest start of task t to min_t implies that t is not in conflict on the interval $[min_t, \delta_t)$ wrt. the PCRCP. Condition $get\_top\_key(h_{check}) > gap$ (Alg. 2 line 5) ensures that the adjustment in line 8 does not induce a resource overflow on $[min_t, \delta_t)$, otherwise t should have been added into $h_{conflict}$. Condition $get\_top\_key(h_{conflict}) \leq gap$ (Alg. 2 line 13) implies that task t is in conflict until the current sweep-line position δ. If $\delta \geq \overline{s_t}$ (line 15) the conflict on $[\overline{s_t}, \delta_t)$ is not "real" since the compulsory part of t is already taken into account in the PCRCP. Alg. 2 (line 20), the earliest start of task t is adjusted to the current sweep-line position, consequently the interval $[min_t, \delta_t)$ is empty.

② For each value of δ greater than δ_t, *sweep_min* cannot create a compulsory part before instant δ_t. This is implied by Lemma 1, which ensures that *sweep_min* cannot generate any compulsory part before δ.

Consequently once *sweep_min* is completed, any task t can be fixed to its earliest start without exceeding the resource limit *limit*. □

Property 2. [Correctness.] For any task t, there is no feasible position before its earliest start min_t wrt. the PCRCP.

Proof. By contradiction. Given a task t, let $omin_t$ be its earliest start before the execution of *sweep_min*. If the earliest start of t is pruned during the sweep, i.e. $min_t > omin_t$, then t is in conflict at a time point in the interval $[omin_t, min_t)$ (see Alg. 2, line 11). Consequently condition $\delta - mins_t > d_t$ (Alg. 2, line 7) is false, which ensures that there is no earliest feasible position before min_t. □

3.5 Complexity

Given a *cumulative* constraint involving n tasks, the worst-case time complexity of the dynamic sweep algorithm is $O(n^2 \log n)$. First note that the overall worst-case complexity of synchronize over a full sweep is $O(n)$ since conditional and dynamic events are updated at most once. The worst-case $O(n^2 \log n)$ can be reached in the special case when the PCRCP consists of a succession of high peaks and deep, narrow valleys. Assume that one has $O(n)$ peaks, $O(n)$ valleys, and $O(n)$ tasks to switch between h_{check} and $h_{conflict}$ each time. A heap operation costs $O(\log n)$. The resulting worst-case time complexity is $O(n^2 \log n)$.[3] For bin-packing, the two heaps $h_{conflict}$ and h_{check} permit to reduce the worst-case time complexity down to $O(n \log n)$. Indeed, the earliest start of the tasks of duration one that exit $h_{conflict}$ can directly be adjusted (i.e. h_{check} is unused).

[3] Note that when the two heaps are replaced by a list where for each active task we record its status (in checking mode or in conflict mode), we get an $O(n^2)$ worst-case time complexity which, like the 2001 algorithm, is reached in practice.

3.6 Greedy Mode

The motivation for a greedy assignment mode is to handle larger instances in a CP solver. This propagation mode reuses the *sweep_min* part of the filtering algorithm in the sense that once the minimum value of a start variable is found, the greedy mode directly fixes the start to its earliest feasible value wrt. Property 1 rather than adjusting it. Then, the sweep-line is reset to this start and the process continues until all tasks get fixed or a resource overflow occurs. Thus the greedy mode directly benefits from the propagation performed while sweeping.

4 Evaluation

We implemented the dynamic sweep algorithm on Choco [19] and SICStus [5]. Benchmarks were run with an Intel i7 at 2.93 GHz processor on one single core, memory limited to 13GB under Mac OS X 64 bits.

In a first experiment, we ran random instances of cumulative and bin-packing problems. Instances were randomly generated with a density close to 0.7. For a given number of tasks, we generated two different instances with the average number of tasks overlapping a time point equal to 10 (denoted by *ttu1*) resp. 100 (denoted by *ttu2*). For cumulative problems, we compared the time needed to find a first solution using fail-first search with the 2001 sweep algorithm (denoted by *sweep*), the dynamic sweep (denoted by *dynamic*) and the greedy mode (denoted by *greedy*). For bin-packing problems, we also tested a dedicated filtering algorithm (denoted by *fastbp*) coming from Entropy [7], an open-source autonomous virtual machine manager. The Choco results are shown in Fig. 3 (top and middle). The SICStus results (omitted) paint a similar picture. We notice a significant difference from the 2001 algorithm due to an inappropriate design of the code for large instances in Choco (iterating over objects). SICStus is up to 8 times faster than Choco on *sweep* and twice as fast on *dynamic* and *greedy*. The dynamic sweep is always faster than the 2001 sweep with a speedup increasing with the number of tasks (e.g., for 8000 tasks up to 7 times in Choco and 5 times in SICStus). The dynamic sweep algorithm is also more robust than the 2001 algorithm wrt. different heuristics. For the bin-packing case (ttu2), *greedy* could handle up to 10 million tasks in one *cumulative* constraint in SICStus in 8h20m.

In a second experiment, we ran the J30 single-mode resource-constrained project scheduling benchmark suite from PSPLib [4], comparing *sweep* with *dynamic*. Each instance involves four *cumulative* constraints on 30 tasks and several precedence constraints. The variable with the smallest minimal value was chosen during search. The SICStus results are shown in Fig. 3 (bottom). The left hand scatter plot compares run times for instances that were solved to within 5% of the optimal makespan within a 1 minute time-out by both algorithms. The right hand scatter plot compares backtrack counts for instances that timed out in at least one of the two algorithms. Search trees are the same for *sweep* and *dynamic*, and a higher backtrack count means that propagation is faster, and so

[4] `http://129.187.106.231/psplib/`, (480 instances)

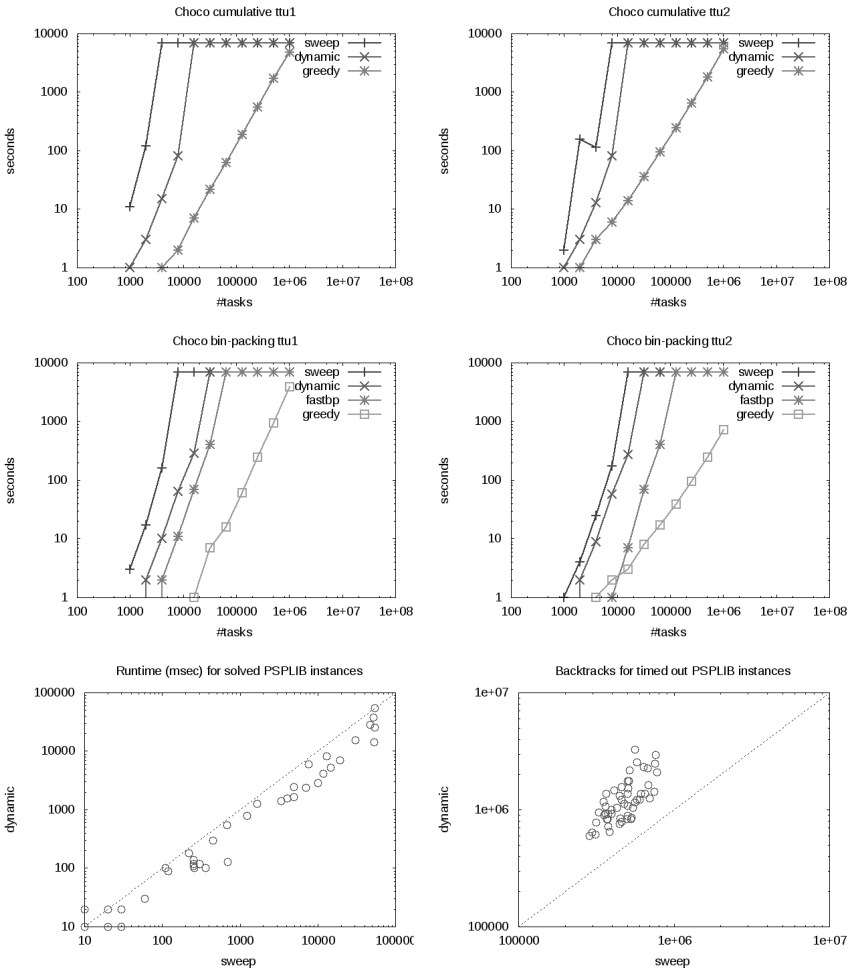

Fig. 3. Runtimes on random instances (top and middle). Comparing runtimes and backtrack counts on PSPLib instances (bottom).

both plots confirm the finding that the dynamic sweep outperforms the 2001 one by a factor up to 3, and not just for problems stated with a single constraint.

5 Conclusion

We have presented a new sweep based filtering algorithm, which dynamically handles deductions while sweeping. In filtering mode, the new algorithm is up to 8 times faster than the 2001 implementation. In assignment mode, it allows to handle up to 1 million tasks in both Choco and SICStus. Future work will focus on the adaptation of this algorithm to multiple resources.

Acknowledgments. Thanks to S. Demassey for providing the *fastbp* constraint.

References

1. Aggoun, A., Beldiceanu, N.: Extending CHIP in order to solve complex scheduling and placement problems. Mathl. Comput. Modelling 17(7), 57–73 (1993)
2. Beldiceanu, N., Carlsson, M., Poder, E., Sadek, R., Truchet, C.: A Generic Geometrical Constraint Kernel in Space and Time for Handling Polymorphic k-Dimensional Objects. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 180–194. Springer, Heidelberg (2007)
3. Beldiceanu, N., Carlsson, M.: A New Multi-resource *cumulatives* Constraint with Negative Heights. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 63–79. Springer, Heidelberg (2002)
4. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational geometry - algorithms and Applications. Springer (1997)
5. Carlsson, M., et al.: SICStus Prolog User's Manual. SICS, 4.2.1 edn. (2012), `http://www.sics.se/sicstus`
6. Freuder, E., Lee, J., O'Sullivan, B., Pesant, G., Rossi, F., Sellman, M., Walsh, T.: The future of CP. Personal communication (2011)
7. Hermenier, F., Demassey, S., Lorca, X.: Bin Repacking Scheduling in Virtualized Datacenters. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 27–41. Springer, Heidelberg (2011)
8. Hermenier, F., Lorca, X., Menaud, J.M., Muller, G., Lawall, J.: Entropy: a consolidation manager for clusters. In: VEE 2009, pp. 41–50. ACM (2009)
9. Kameugne, R., Fotso, L.P., Scott, J., Ngo-Kateu, Y.: A Quadratic Edge-Finding Filtering Algorithm for Cumulative Resource Constraints. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 478–492. Springer, Heidelberg (2011)
10. O'Sullivan, B.: CP panel position - the future of CP. Personal communication (2011)
11. Pape, C.L.: Des systèmes d'ordonnacement flexibles et opportunistes. Ph.D. thesis, Université Paris IX (1988) (in French)
12. Régin, J.C., Rezgui, M.: Discussion about constraint programming bin packing models. In: AI for Data Center Management and Cloud Computing. AAAI (2011)
13. ROADEF: Challenge 2012 machine reassignment (2012), `http://challenge.roadef.org/2012/en/index.php`
14. Schaus, P., Deville, Y.: A global constraint for bin-packing with precedences: application to the assembly line balancing problem. In: AAAI 2008, pp. 369–374. AAAI Press (2008)
15. Schulte, C.: Comparing trailing and copying for constraint programming. In: Schreye, D.D. (ed.) ICLP 1999, pp. 275–289. The MIT Press (1999)
16. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Why *Cumulative* Decomposition Is Not as Bad as It Sounds. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 746–761. Springer, Heidelberg (2009)
17. Shaw, P.: Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In: Maher, M.J., Puget, J.-F. (eds.) CP 1998. LNCS, vol. 1520, pp. 417–431. Springer, Heidelberg (1998)
18. Shaw, P.: A Constraint for Bin Packing. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 648–662. Springer, Heidelberg (2004)
19. Team, C.: Choco: an open source Java CP library. Research report 10-02-INFO, Ecole des Mines de Nantes (2010), `http://choco.emn.fr/`
20. Vilím, P.: Edge Finding Filtering Algorithm for Discrete Cumulative Resources in $O(kn \log n)$. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 802–816. Springer, Heidelberg (2009)
21. Vilím, P.: Timetable Edge Finding Filtering Algorithm for Discrete Cumulative Resources. In: Achterberg, T., Beck, J.C. (eds.) CPAIOR 2011. LNCS, vol. 6697, pp. 230–245. Springer, Heidelberg (2011)