A Scalable Sweep Algorithm for the *cumulative* and *bin-packing* constraints

Arnaud Letort

TASC team, (EMN-INRIA,LINA) Mines de Nantes, France
arnaud.letort@mines-nantes.fr

Abstract. This paper presents a new sweep based algorithm for the *cumulative* constraint that combines filtering with a greedy mode. The algorithm has a worst case complexity of $\mathcal{O}(n^2 \log n)$ in the context of cumulative and $\mathcal{O}(n\ log\ n)$ in the context of bin-packing, where n is the number of tasks (items). It can handle up to 256000 tasks (items) in less than 15 minutes within a single *cumulative* constraint.

Keywords: cumulative, bin-packing, filtering, greedy

1 Introduction

In the 2011 Panel of the Future of CP [4], one of the identified challenges for CP was the need to handle large scale problems. A typical problem class given as an example were problems taken from the cloud and more specially multi-dimensional bin-packing problems [7]. Indeed the importance of bin-packing problems was recently highlighted in [8] and is the topic of the 2012 Roadef Challenge [9]. Till now, dedicated algorithms and metaheuristics were used to deal with large instances. Our main objective is to provide an efficient generic solution to solve such problems with a CP solver. Typically, filtering algorithms focus on having the best possible deductions [6], [11] and often reach their worst time complexity and use a lot of memory which prevent scalability.
The *Cumulative Scheduling Problem* (CuSP) is a satisfaction problem involving a unique resource with a given capacity and a set of tasks. The goal is to find a start time for each task such that the amount of resource needed is always less than or equal to the capacity of the resource. Nowadays, many real world scheduling problems are hard to solve just because of the number of tasks involved. To solve them, we need to design fast and light propagation algorithms that can handle hundreds thousands of tasks (items) within a single constraint.
Originally, the *cumulative* constraint was introduced in [1] to tackle resource scheduling problems. The sweep algorithm introduced in [3] performs a pruning for this constraint, based on the aggregation of compulsory parts (i.e. the *compulsory part* of a task is the intersection of all its feasible instances).
In this paper, we introduce a new filtering algorithm for the *cumulative* constraint based on a dynamic sweep which brings the following contributions :

– from a practical point of view, a substantial gain in performance compared to the original sweep algorithm [3].

– a better worst case complexity on bin packing problems ($\mathcal{O}(n\ log\ n)$).
– a greedy propagation mode for large instances that is directly derived from the filtering, combining a classical greedy algorithm with a filtering algorithm.

Section 2 recalls the original sweep algorithm for the *cumulative* constraint, highlighting its weak points. Then, section 3 presents key elements of our new sweep based filtering algorithm and its greedy propagation mode.

2 The Original Sweep Algorithm

The sweep algorithm was originally introduced as a generic pruning technique in [2]. In 2 dimensions, the sweep moves a vertical line, called the sweep line, from left to right over discrete locations called event points. The algorithm uses two data structures:

– A data structure called the *sweep-line* status, which contains information related to the current position δ of the sweep line : (1) the height of compulsory parts *sum_height* , (2) the list of tasks *prune*, i.e. the tasks which can overlap the current sweep-line position.
– An array named the *event point series*, which holds the events to process, sorted in increasing order according to the abscissa. These events correspond to the start and the end of compulsory parts as well as to the earliest starts of tasks.

After the initialization of the two data structures, the sweep-line reads events in increasing order and incrementally updates the sweep-line status. In our context, the *cumulative* constraint, the sweep-line scans the time axis in order to build the cumulated profile and to perform a pruning of the task origins w.r.t. this profile and the capacity *capa* of the resource.

At each position δ of the sweep-line, all events associated to this position are read and processed. As a consequence, we know the height of the cumulated profile on the interval $[\delta, \delta')$ where δ' is the next sweep line position. Then, all tasks overlapping that interval (i.e. tasks in *prune*) are scanned and pruned if their height is strictly greater than the available resource.

The sweep algorithm removes intervals of consecutive values from domain variables. This is a first weakness that prevents handling large instances since a variable cannot just be compactly represented by its minimum and maximum values. The second weakness is that it needs to rescan all tasks which overlap the current position of the sweep line each time the sweep line moves, which often leads in practice to a quadratic time complexity.

Saturation In order to reach the fixpoint at each node of the search tree, the sweep algorithm is successively re-run until no pruning occurs any more. This is due to the fact that the potential increase of the cumulated profile during a single sweep is not dynamically taken into account. In other words, creations and extensions of compulsory parts during a sweep are not directly used to perform more pruning while sweeping.

Complexity The complexity of the original sweep algorithm [3] is $\mathcal{O}(n^2)$ where n is the number of tasks. This complexity is often reached in practice when tasks can be placed everywhere on the time line. Note that the complexity of the algorithm for the bin packing case (i.e. all tasks duration are set to 1) is left unchanged.

3 The Dynamic Sweep Algorithm

We now introduce our contribution, a dynamic sweep based filtering algorithm for the *cumulative*, and a greedy mode for large instances completely relaying on this dynamic sweep.
The first major difference with the original sweep is that our algorithm only deals with variables bounds, which is a good way to reduce the memory consumption attached to the variables. The dynamic sweep algorithm tries to solve the problem by pruning lower and upper bounds of variables in two distinct sweep stages. The first stage, called *sweep_min*, tries to prune lower bounds of the start variables by performing a sweep from left to right whereas the second stage tries to prune upper bounds by performing a sweep from right to left. Without loss of generality, we focus on *sweep_min* since the other part is strictly symmetric.
As we said before, the original sweep has to be re-run to reach its fixpoint, due to the fact that during a same run, restrictions of the origins are not directly taken into account. Our *sweep_min* algorithm dynamically uses these deductions to reach its fixpoint in one single run. This permits to derive a greedy mode using the *sweep_min* part of the filtering algorithm.

The two main difficulties of *sweep_min* are the following: (1) how to handle on the fly, pruning of variables bounds during a single sweep, since adjusting the earliest start of a task may increase its compulsory part which may trigger extra deductions ? (2) how to avoid rescanning all the tasks at each position of the sweep-line ? Difficulty (1) will be addressed by introducing the notion of conditional events, while difficulty (2) will be addressed by introducing a dedicated data structure.

Data Structures In order to catch on the fly adjustments performed by the current sweep, we need a data structure to handle events. Indeed, during the sweep, for each task for which the earliest start is adjusted, we have to update its associated events. That is why, instead of a sorted array of events, we use the following data structure:

- A heap h_{events} for storing the events. We use a heap rather than an array since new events can be dynamically added during the sweep, and we always need to know the minimal event date (i.e. the next event date). A a consequence, events are recorded in the heap by their increasing date.

Once all events on a given position δ are handled and the next event date δ' is known, the filtering procedure scans each task which overlaps the interval $[\delta, \delta')$ (i.e. tasks in *prune*). For each task, it checks that its height is less or equal

than the gap (i.e. the capacity of the resource minus the height of the cumulated profile), if not, the start variable of the task is pruned. Based on the fact that, if the start variable of task t needs to be pruned on the current sweep-line position, start variable of all tasks with a greater height than h_t need to be pruned too. And symmetrically, if start variable of task t does not need to be pruned, start variable of all tasks in *prune* with a less or equal height don't need to be pruned. We introduce the two following data structures :

- A heap h_{check} for storing tasks for which the current tested position is feasible. The tasks are recorded in the heap by decreasing height.
- A heap $h_{conflict}$ for storing tasks for which we know that we will have to update their earliest start since the current tested position is infeasible. The tasks are recorded in the heap by increasing height.

These two heaps avoid a pitfall of the original sweep which consists in rescanning all tasks of *prune* at each sweep-line position.

Dynamic Events Like the original sweep algorithm, each task generates a list of events which are inserted into h_{events}. The fact that we need during the sweep to dynamically take into account the filtering performed, forces us to introduce new dynamic events. According to its type, an event points to: the earliest starting time of the task (i.e. *PR* type), the start of a compulsory part (i.e. *SCP* type), the end of a compulsory part (i.e. *ECP* type). We introduce a new event type (compared to the original sweep) called *CCP* event, standing for *Conditional Compulsory Part*. This event is generated for each task which initially has no compulsory part and is set to its latest starting time. Since *sweep_min* algorithm sweeps and prunes lower bounds of start variables, it denotes the first time point where the compulsory part of the task can start, iff the task is pruned enough.

Synchronize Data Structures The introduction of dynamic events in our algorithm brings synchronization problems between the real current state of tasks and the information stored into the data structures. For instance, before each extraction from the heap h_{events}, we must check that the top event is always valid. Since the compulsory part of a task can dynamically be extended or created, some events generated during the initialization step and always present in the heap, can be de-synchronized. There are two cases where an event is desynchronized.
The first one is the case where a task t initially without compulsory part is pruned enough to have one. When the sweep-line reads the *CPP* event of task t, it checks that t still does not have a compulsory part. If it does, it adds the two events corresponding to the start and the end of the compulsory part (i.e. *SCP* and *ECP*) in h_{events}. Since we only adjust lower bounds, the date of the *SCP* event type is the same that the de-synchronized *CCP* event.
The second case involves a task t which initially has a compulsory part. If its earliest starting time is pruned, the end of its compulsory part is pushed to the the right, so its initial *ECP* event is de-synchronized and need to be updated.

Property For a task t, s_t denotes its start variable, e_t its end variable and h_t its height. For a given variable v, $\underline{v}$ denotes its lower bound, $\overline{v}$ its upper bound. T denotes the set of tasks of the problem and C the capacity of the resource. The *sweep_min* algorithm ensures the following property:

$$\forall t \in T, \forall i \in [\underline{s_t}, \underline{e_t}) : \; h_t + \sum_{\substack{t' \in T \setminus \{t\} \\ \overline{s_{t'}} \leq i < \underline{e_{t'}}}} h_{t'} \leq C \tag{1}$$

The property ensured by the *sweep_min* algorithm is that for any task t of the problem scheduled to its earliest position, the cumulated profile of compulsory parts (including task t) does not exceed the capacity of the resource C.

Complexity In a cumulative problem, the overall complexity of the dynamic sweep algorithm is $\mathcal{O}(n^2 \log n)$. This worst case can be reached when tasks are often switched between h_{check} and $h_{conflict}$. This can occur when the cumulated profile consists of a succession of peaks and valleys. Despite all, the introduction of new data structures permits to avoid this worst case in practice. For the bin-packing problem, our new data structures permit to reduce the complexity 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).

Greedy Mode The main motivation for a greedy propagation mode is to handle large instances in a CP solver. This propagation mode is closely related to 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 task to its earliest start rather than adjusting it. Intuitively, the greedy is an opportunistic mode of the *sweep_min* algorithm which tries to build a solution in a single sweep.

4 Evaluation

We implement the dynamic sweep algorithm on Choco [10] and run large random instances of cumulative and bin packing problems. For cumulative problems we have compared the time needed to find a first solution with the three following algorithms, the original sweep (denoted by *sweep*) which is already present in Choco, the new dynamic sweep (denoted by *dynamic*) and the greedy mode (denoted by *greedy*). For bin packing problems, we also test a dedicated filtering algorithm (denoted by *fastbp*) coming from entropy [5], an open-source autonomous virtual machines manager. Benchmarks were run with an Intel i7 720QM processor, a memory limited to 2.5GB under Windows 7 64 bits.

We observe that the dynamic sweep algorithm is clearly faster than the original sweep algorithm in Choco. This difference is partially due to an inappropriate design of the code for large scale instances. Anyways, we can see that the greedy mode has a better scalability than the two other algorithms. For bin-packing problems, the dedicated constraint (*fastbp*) is faster than the dynamic sweep (even if it does not scale as well), but always beaten by the greedy.

# of tasks	cumulative instances			bin-packing instances			
	sweep	*dynamic*	*greedy*	*sweep*	*dynamic*	*fastbp*	*greedy*
1000	16.1	2.2	0.4	4.6	1.9	0.3	0.4
2000	168.3	6.4	0.8	34.8	6.2	0.8	0.5
4000	1284	26.3	2.1	269.1	26.0	3.4	0.6
8000	to	128.8	4.6	to	143.6	27.3	1.0
16000	to	588.2	13.9	to	605.0	175.9	2.0
32000	to	to	38.7	to	to	m.o.	6.6
64000	to	to	124.9	to	to	m.o.	23.5
128000	to	to	388.9	to	to	m.o.	83.9
256000	to	to	1309	to	to	m.o.	832.3

Table 1. Average time in seconds to find a first solution to the problem with the heuristic *first-fail*. Timeout is fixed to 1800s. (*to* stands for *timeout*, *m.o.* for *memory overflow*))

5 Conclusion

We have presented a new sweep based filtering algorithm which dynamically handle deductions during a unique sweep stage. It reaches its fixpoint in only one run which permits to design an efficient greedy mode. Future work will focus on the adaptation of this algorithm to multiple resources.

Acknowledgments The author would like to thank Nicolas Beldiceanu for his valuable contribution, Mats Carlsson for his help with the code and Sophie Demassey for providing the constraint *fastbp*.

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.: Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In: CP. pp. 377–391 (2001)
3. Beldiceanu, N., Carlsson, M.: A New Multi-Resource *cumulatives* Constraint with Negative Heights. In: CP 2002. LNCS, vol. 2470, pp. 63–79. Springer-Verlag (2002)
4. Freuder, E., Lee, J., O'Sullivan, B., Pesant, G., Rossi, F., Sellman, M., Walsh, T.: The future of cp. personal communication (2011)
5. Hermenier, F., Demassey, S., Lorca, X.: The bin-repacking scheduling problem in virtualized datacenters. In: CP'11. LNCS, Springer-Verlag, Perrugia, Italy (2011)
6. Kameugne, R., Fotso, L.P., Scott, J., Ngo-Kateu, Y.: A quadratic edge-finding filtering algorithm for cumulative resource constraints. In: CP. pp. 478–492 (2011)
7. O'Sullivan, B.: Cp panel position - the future of cp. personal communication (2011)
8. Régin, J.C., Rezgui, M.: Discussion about constraint programming bin packing models. In: AI for Data Center Management and Cloud Computing. AAAI (2011)
9. ROADEF: Challenge 2012 machine reassignment (2012), http://challenge.roadef.org/2012/en/index.php
10. Team, C.: Choco: an open source java CP library. Research report 10-02-INFO, Ecole des Mines de Nantes (2010), http://choco.emn.fr/
11. Vilím, P.: Edge finding filtering algorithm for discrete cumulative resources in o(kn log n). In: CP. pp. 802–816 (2009)