Processing times filtering for the Cumulative constraint *

Yanick Ouellet[1] and Claude-Guy Quimper[1]

Université Laval, Québec, Canada
yanick.ouellet@ulaval.ca
claude-guy.quimper@ift.ulaval.ca

Abstract. The Cumulative constraint and its many propagators are useful to solve scheduling problems with Constraint Programming. However, most propagators only filter the starting variables of the tasks. We propose a novel algorithm to filter the upper bound of the processing times. We also show how to explain the filtering done by our algorithm. This allows its use with the powerful lazy clause generation technique.

Keywords: Constraint Programming · Cumulative constraint · Processing times · Lazy Clause Generation.

1 Introduction

The Cumulative constraint is a useful tool to model scheduling problems using Constraint Programming. An impressive amount of work has been done to develop and improve various filtering algorithms for that constraint. Recently, the lazy clause generation technique [8] encouraged further work to improve the Cumulative via explanation generation [11, 12].

However, most filtering algorithms for the Cumulative filter only the starting time and ending time variables of the tasks. To the authors' best knowledge, there is only one algorithm, by Vilím [15], that filters the processing times. In this paper, we propose a new algorithm that complements Vilím's one. Our algorithm produces its filtering by considering the compulsory parts of the tasks, as does the Time-Tabling rule. We also show how to explain the filtering done.

We begin by presenting the Cumulative constraint and the lazy clause generation method. We then introduce our algorithm and show how to generate explanations from its filtering. We conclude with a discussion on future work.

2 Background

2.1 The Cumulative constraint

Let $\mathcal{I}$ be a set of n task indices. A task i has an *earliest starting time* est_i, a *latest completion time* lct_i, a *processing time* p_i and a *height* h_i. The task must

* Supported by FRQNT and Mitacs.

be scheduled during p_i consecutive units of time, between its est_i and lct_i. During that time, the task consumes h_i units of a resource. From these parameters, one can compute the *earliest completion time* $\text{ect}_i = \text{est}_i + p_i$ and the *latest starting time* $\text{lst}_i = \text{lct}_i - p_i$.

A task is said to have a *compulsory part* $[\text{lst}_i, \text{ect}_i)$ if $\text{ect}_i > \text{lst}_i$. A task i will necessarily be executing throughout its compulsory part, no matter when the task starts. Figure 1 is an example of a task with a compulsory part. It has an est of 0, a processing time of $p = 6$, an ect of 6, a lct of 10, a lst of 4 and a height of 2. The compulsory part in gray spans the interval $[4, 6)$ for which we are sure that the task will execute.

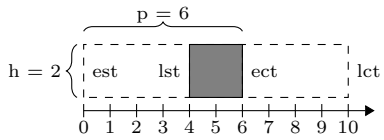

Fig. 1. Task with a compulsory part in the interval $[4, 6]$.

The Cumulative enforces that the amount of resource consumed by all tasks in execution at a given time t must be less than or equal to the capacity C of the resource. In other words, we have $\forall_t \sum_{i \in \mathcal{I}: S_i \leq t < E_i} h_i \leq C$, where S_i and E_i are the starting time and ending time variables of a task. These variables are subject to $S_i + p_i = E_i$

In the classic version of the Cumulative constraint, the only decision variables are the starting times. The processing times, the heights and the capacities are constant. However, in some cases, it may be useful to have one or more of these parameters as variables.

Suppose, for example, that the timepoints represent days, that the tasks are jobs done by workers and that the resource is the amount of workers available. One might want to allow the workers to do some amount of overtime each day. In such a case, some tasks will take fewer days to complete since more hours are worked every day. The model would try to minimize the amount of overtime. Thus, tasks should have processing time variables in order to let the solver shorten a task in time by means of working overtime.

In cases where the processing times or the heights are variables, it is possible to apply the same checker and filtering algorithms as for the traditional Cumulative. However, such algorithms must only consider the lower bounds of these variables. For instance, if we have a task with $\text{est}_i = 0, \text{lct}_i = 4, p_i \in [4, 10]$, any filtering algorithm simply use $p_i = 4$.

A substantial amount of work has been done to improve filtering of the Cumulative constraint. Filtering algorithms include Time-Tabling [1], Edge-Finder [7], Time-Tabling-Edge-Finder [14], Not-First/Not-Last [4, 10] and Energetic Reasoning [6, 9].

Most of these algorithms only filter the starting time and ending time variables. When the processing times or the heights are variables, it should be beneficial to filter them. However, it is only possible to filter the upper bound of those variables. Indeed, if a solution satisfies the CUMULATIVE for a processing time p_i, it also satisfies the constraint for the processing time $p_i - 1$. The same reasoning applies for the height of the tasks. In this paper, we focus only on filtering the processing times.

2.2 Lazy clause generation

Recently, a new technique, called lazy clause generation, has been introduced by Ohrimenko et al.[8]. The innovation behind that technique is to combine the strengths of SAT solvers and Constraint Programming. The approach uses the powerful nogood learning capabilities of the SAT solvers, combined with the specialized filtering algorithms of global constraints propagators.

To achieve that Ohrimenko et al. propose to represent the domain of integer variables as both Boolean variables and normal integer variables. An integer variable X can be encoded by having each Boolean variable representing a certain aspect of the domain of that variable. For instance, the Boolean variable $[\![X = 5]\!]$ indicates that the integer X has a value of 5. Similarly, $[\![X \geq 5]\!]$ means that X has a value of at least 5 and $\neg[\![X \geq 5]\!]$ means that X has a value of less than 5.

In addition, constraints must be added to ensure these Boolean variables represent valid integers. An example of such constraint is $[\![X = 5]\!] \Rightarrow [\![X \geq 5]\!]$. Since the number of such constraints and variables is high in real-world problems, Ohrimenko et al. propose to generate them only when needed in the search, instead of generating them all at the beginning. This is what gives the name lazy clauses generation. The details of this representation is not the focus of this paper, more comprehensive information can be found in [8].

During the search, the SAT solver uses unit propagation and nogoods learning as usual, using the Boolean variables. When it branches on a variable, the global constraint propagators are called. Instead of filtering as usual, they *explain* the filtering they would have done in a normal CP solver. An explanation is a SAT clause representing the filtering, and the reason behind that filtering. The reason is a subset of the domain of the variables that caused the filtering.

Suppose we have two variables $X \in [0, 10], Y \in [5, 10]$ and the constraint $X \geq Y$. A propagator could detect that the lower bound of X must be filtered to 5. The reason behind that is the lower bound of the domain of Y. The propagator could generate the explanation $[\![Y \geq 5]\!] \Rightarrow [\![X \geq 5]\!]$. This explanation is not immediately added to the SAT solver. Instead, the search continues until a global propagator produces a failure. When this happens, that propagator generates a clause explaining the failure. For example, if we have $X \in [0, 4], Y \in [5, 10]$ and the same constraint as previously, the propagator generates the explanation $[\![X \leq 4]\!] \wedge [\![Y \geq 5]\!] \Rightarrow \mathbf{fail}$. At that point, the solver combines that explanation and the explanations of the previously filtered variables to generate a nogood. A nogood is a clause that is added to the SAT solver. Its goal is to prevent the

solver from doing the same choices as those that led to a failure. This process is explained in more details by Stuckey [13].

The main challenges with this technique is to adapt the existing propagators to generate explanations. It is not trivial since the explanations need to be as small and reusable as possible to be able to significantly reduce the search space.

Schutt et al. had great successes in explaining the Overload Check, the Time-Tabling, the Edge-Finding and the Time-Tabling-Edge-Finding algorithms [11, 12] of the Cumulative constraint. It allowed them to close some instances of the classic PSPLIB [5] benchmark and improve the best known upper bounds of other instances of that benchmark.

2.3 Vilím's algorithm

Vilím introduced a filtering algorithm [15] for the processing time and the height variables. His algorithm filters the upper bound of those variables to the maximum amount such that the Overload Check [16] does not fail.

3 Processing times filtering

We propose a new algorithm that filters the upper bound of the processing time domains. Our algorithm is similar to the time-tabling in the sense that we only consider the compulsory parts of the tasks.

Consider that we want to filter the upper bound of the processing time of task i. We know that we can schedule task i only in intervals where at least h_i units of resources are available at each timepoint in the interval.

We define a *fixed aggregate* for task i as an interval $[l, u)$ in which, at any timepoint, the sum of the height of the compulsory part of the tasks other than i is greater than $C - h_i$:

$$\forall_{l \leq t < u} \sum_{j \in \mathcal{I} \setminus \{i\} : \mathrm{lst}_j \leq t < \mathrm{ect}_j} h_j > C - h_i$$

Similarly, we define a *hole* for task i as an interval that is not a fixed aggregate. In a hole, the compulsory parts of tasks other than i do not prevent i from executing. In other words, an interval $[l, u)$ is a hole if and only if the following equation holds.

$$\forall_{l \leq t < u} \sum_{j \in \mathcal{I} \setminus \{i\} : \mathrm{lst}_j \leq t < \mathrm{ect}_j} h_j \leq C - h_i$$

Figure 2 is an example of a task i with an est of 0, a lct of 10, a height of 2 and a processing time with a domain of $[2, 10]$. The capacity of the resource is 3. The compulsory part of all tasks except i are represented in gray. In this example, there are two holes, $[0, 2)$ and $[4, 8)$. They are represented by thick blue

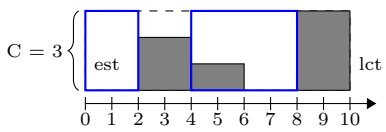

Fig. 2. Example where a task with a height of 2 would have the upper bound of its processing time filtered to 4. The grey parts represent the fixed part of other tasks and the rectangles with thick blue borders are holes.

borders. The other intervals are fixed aggregates. The largest hole has a length of 4. Thus, the upper bound of the processing time of i would be filtered to 4.

The idea behind our filtering algorithm is to use the length of the holes to filter the upper bound of the processing times. We distinguish two cases: when task i does not have a compulsory part and when i has a compulsory part.

If task i does not have a compulsory part, it can be executed in any hole long enough to contain its processing time. These holes must be included in $[\text{est}_i, \text{lct}_i)$. Note that there cannot be a fixed aggregate during the execution of the task since it would cause an overload of the resource. Hence, the length of the largest hole in $[\text{est}_i, \text{lct}_i)$ is an upper bound on the processing time of task i.

If task i has a compulsory part, it must be executing during that time. Since the CUMULATIVE does not allow preemption, the fraction of the processing time of i that is not a compulsory part must be scheduled immediately before or after the compulsory part, or both. Let l_a be the smallest time point such that $[l_a, \text{lst}_i)$ is a hole and let u_b be the largest time point such that $[\text{ect}_i, u_b)$ is a hole. Task i can execute for at most $u_b - l_a$ units of time.

The filtering algorithm we propose uses a datastructure called the profile, inspired by the one used in [3]. This datastructure is composed of several vectors. The first vector, T, contains the lst and ect of all tasks, sorted in non-decreasing order. The two vectors `lstToTimepoint` and `ectToTimepoint` map a task with the index of its lst and ect in T. We can build these vectors using a process similar to the merge procedure of the merge sort algorithm. The fourth vector `compulsoryparts`, contains, at index t, the sum of the height of the compulsory parts of all tasks in the interval $[T[t], T[t+1])$. Since compulsory parts begin at a lst and end at an ect, we know that the height cannot change between two timepoints. To compute that vector, we begin by iterating over all tasks. For each task j, we add h_j to `compulsoryparts` at the timepoint lst_j, where the compulsory part of j begins. We remove h_j from `compulsoryparts` at the timepoint ect_j, where the compulsory part ends. We then compute the partial sum of `compulsoryparts`, such that `compulsoryparts`$[t]$ = `compulsoryparts`$[t]$ + `compulsoryparts`$[t-1]$ for all $1 < t \leq |$`compulsoryparts`$|$.

With that datastructure, we can filter the upper bound of the processing times of the tasks. We begin by processing each task i individually. For each task without a compulsory part, we note that a hole begins at est_i. We then process each timepoint t in the interval $[\text{est}_i, \text{lct}_i)$ in non-decreasing order. If `compulsoryparts` is greater than $C - h_i$ at timepoint t (ie. `compulsoryparts`$[t] >$

$C-h_i$), it means that the hole ends and a fixed aggregate begins. We note the size of the hole. We continue to examine timepoints until we have $\texttt{compulsoryparts}[t] \leq C-h_i$, at which point another hole begins. We repeat the process until all timepoints in $[\text{est}_i, \text{lct}_i)$ have been processed. The filtered upper bound of the processing time corresponds to the length of the greatest hole we found. Note that we can stop the process early if we find a hole with a length greater than the current upper bound of the processing time. In that case, no filtering is needed.

For tasks with compulsory parts, we begin at the timepoint lst_i and process earlier timepoints one by one until we find a fixed aggregate or we are at a timepoint smaller that est_i. This give us the length of the left hole. The process for the right hole is symmetric. We then add the length of both holes to the length of the compulsory part $(\text{ect}_i - \text{lst}_i)$ to compute the filtered upper bound.

Our algorithm has a $O(n^2)$ complexity, where n is the number of tasks. Indeed, the profile datastructure contains at most $2n$ timepoints, one for each ect and lct. Hence, the algorithm needs to examine $O(n)$ timepoints for each task to filter, giving us the complexity of $O(n^2)$ for all tasks.

4 Generating explanations

Our algorithm needs to generate explanations indicating the reasons behind the filtering it produces. In this section, we assume that we want to explain the filtering of the upper bound of the processing time of task i to a new value x.

We first add the following literals for the current est and lct of task i: $[\![S_i \geq \text{est}_i]\!] \wedge [\![E_i \leq \text{lct}_i]\!]$. We have these literals since the filtering algorithm searches for holes only in the interval $[\text{est}_i, \text{lct}_i)$. There might be a hole of greater length outside that interval that becomes available if the est decreases or the lct increases, as shown on Figure 3. This means that our explanations need to have the form $[\![S_i \geq \text{est}_i]\!] \wedge [\![E_i \leq \text{lct}_i]\!] \wedge ... \Rightarrow [\![P_i \leq x]\!]$. It might be possible to extend these literals to make the explanation more reusable. Note that we do not need to include the lower bound of the processing time of task i in the left-hand side of the explanation since our algorithm does not consider it when filtering.

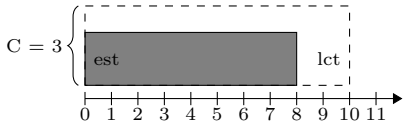

Fig. 3. Example where a task with a height of 2 and an lct of 10 would have the upper bound of its processing time filtered to 2. However, if its lct increases to 11, the hole of length 2 becomes of length 3. The filtered value of 2 is no longer valid.

There are many ways to explain the fixed aggregates, each with varying degree of simplicity and reusability. We begin by presenting naive explanations and then proceed to show how to improve on that.

4.1 Naive explanation

The easiest way to produce an explanation is to include literals for the current domain of all tasks that have a compulsory part in $[\text{est}_i, \text{lct}_i)$. These tasks are the ones that produce the fixed aggregates. For each of those tasks, we need to include a literal for the lower bound and upper bound of its starting time and one literal for the lower bound of its processing time. If the starting time or the processing time become smaller than their current lower bounds or if the starting time becomes greater than its current upper bound, the compulsory part of the task decreases. This could create new holes that have not been considered by the filtering algorithm. The resulting explanation is as follows.

$$[\![S_i \geq \text{est}_i]\!] \wedge [\![E_i \leq \text{lct}_i]\!] \wedge \bigwedge_{j \in F} \left([\![S_j \geq \text{est}_j]\!] \wedge [\![S_j \leq \text{lct}_j]\!] \wedge [\![P_j \geq p_j]\!] \right) \Rightarrow [\![P_i \leq x]\!]$$

The set F is the set of all tasks that have a compulsory part in $[\text{est}_i, \text{lct}_i)$.

It is possible to improve the quality of these naive nogoods with a relatively simple adjustment. We know that, at a given timepoint, the task being filtered can either be executed or not. If that timepoint is part of a hole, the task can be executed. If it is part of a fixed aggregate, the task cannot be executed. Hence, the height of the fixed aggregate only needs to be greater than $C - h_i$. Let $h_{\texttt{FA}}$ be the height of the fixed aggregate. We have $h_{\texttt{FA}} > C - h_i$. Thus, we can remove tasks from the fixed aggregate as long as the previous equation holds.

To generate an explanation with this method, we proceed as follows. We identify each fixed aggregate and process them individually. For each fixed aggregate `FA`, we sort the tasks in that aggregate by height. Then, we process the tasks in non-increasing order of height. For each task, we add it to the explanation and add its height to $h_{\texttt{FA}}$. We stop as soon as we have $h_{\texttt{FA}} > C - h_i$.

By proceeding this way, we remove as many tasks from the explanation as possible, compared to the naive explanation. For each task removed in that manner, three literals are removed from the explanation.

4.2 Extended explanation

It is possible to further generalize our explanation. Instead of generating literals representing the current domain of the tasks in the fixed aggregates, we can extend the literals to make them more general. Sometimes, a fraction of the compulsory part of a task is not relevant in the fixed aggregate. This can happen if the compulsory part crosses the interval $[\text{est}_i, \text{lct}_i)$ or if the compulsory part is in part in a hole and in part in a fixed aggregate (see Figure 4). In such cases, we can produce a literal with only the relevant fragment of the compulsory part.

Suppose we want to generate literals for the fixed aggregate $[l, u)$. For task j in that fixed aggregate, we generate the following literal for the lower bound of the starting time: $[\![S_j \geq \min(\text{est}_j, u - p_j)]\!]$. The second option in the minimum $(u - p_j)$ represents the case where the compulsory part crosses the fixed aggregate by the right. The segment $[u, \text{ect}_j)$ is not in the fixed aggregate and we can remove

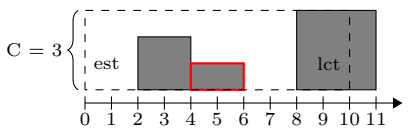

Fig. 4. Example where a task of height 2 has the upper bound of its processing time filtered to 4. The grey parts represent the compulsory part of other tasks. The compulsory part with a thick red border in the interval $[4, 6)$ is in a hole and thus is not relevant. The part of the fixed aggregate in the interval $[10, 11)$ is not relevant since it is after the lct of the task to be filtered.

it from the explanation. Hence, we can reduce the earliest completion time of j to u. Since the literal is a lower bound on the starting time, we have to convert the earliest completion time to its equivalent earliest starting time using $u - p_j$.

We can proceed similarly for the literal representing the upper bound of the starting time. We use the literal $[\![S_j \leq \max(\text{lst}_j, l)]\!]$. The second part of the maximum represents the case where the compulsory part crosses the fixed aggregate by the left. In such a case, the segment $[\text{lst}_j, l)$ is not relevant and we can generalize the literal by removing it.

We have an idea to generalize even further our explanations. We know that the less fixed aggregates there are (and the smaller they are), the fewer literals we have to include in the explanation. Reducing the size and quantity of fixed aggregate is equivalent to increasing the size and quantity of holes. Fortunately, we can easily do the latter. Recall that our filtering algorithm filters the processing time by finding the greatest holes. This means that we can extend the smaller holes by increasing their size up to the size of the largest hole, which is the upper bound of the processing time. It is also possible to create holes in fixed aggregates. However, in both of these cases, we must be careful to keep a fixed aggregate of at least one unit between holes.

Several strategies could be used to generate and extend holes. One could, for instance, randomly extend or create holes. It could also be possible to create holes that would allow to remove tasks from the explanation. This could come with a complexity cost. Experiments with various strategies will be required.

5 Conclusion

We introduced a novel algorithm to filter the upper bound of the processing times for the Cumulative constraint. We also showed how to generate explanations from the filtering done.

However, this is still a work in progress. The algorithm has been implemented in C++ using the Chuffed solver [2], but experimentations are needed to confirm its usefulness. More work will likely be needed to improve our explanation and experiments with the different ways to generate them. We also have plans to improve our filtering algorithm from $O(n^2)$ to $O(n \log n)$ using a tree datastructure.

References

1. Beldiceanu, N., Carlsson, M.: A new multi-resource cumulatives constraint with negative heights. CP **2470**, 63–79 (2002)
2. Chu, G.: Improving combinatorial optimization. Ph.D. thesis, Department of Computing and Information Systems, University of Melbourne (2011)
3. Gingras, V., Quimper, C.G.: Generalizing the edge-finder rule for the cumulative constraint. In: IJCAI. pp. 3103–3109 (2016)
4. Kameugne, R., Fotso, L.P.: A cumulative not-first/not-last filtering algorithm in $O(n^2log(n))$. Indian Journal of Pure and Applied Mathematics **44**(1), 95–115 (2013)
5. Kolisch, R., Sprecher, A.: PSPLIB-a project scheduling problem library: Or software-orsep operations research software exchange program. European journal of operational research **96**(1), 205–216 (1997)
6. Lopez, P., Esquirol, P.: Consistency enforcing in scheduling: A general formulation based on energetic reasoning. In: 5th International Workshop on Project Management and Scheduling (PMS'96) (1996)
7. Mercier, L., Hentenryck, P.V.: Edge finding for cumulative scheduling. INFORMS Journal on Computing **20**(1), 143–153 (2008)
8. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation via lazy clause generation. Constraints **14**(3), 357–391 (Jan 2009). https://doi.org/10.1007/s10601-008-9064-x, http://dx.doi.org/10.1007/s10601-008-9064-x
9. Ouellet, Y., Quimper, C.G.: A $O(n \log^2 n)$ checker and $O(n^2 \log n)$ filtering algorithm for the energetic reasoning. In: International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research. pp. 477–494. Springer (2018)
10. Schutt, A., Wolf, A., Schrader, G.: Not-first and not-last detection for cumulative scheduling in $O(n^3 \log n)$. In: INAP. pp. 66–80. Springer (2005)
11. Schutt, A., Feydy, T., Stuckey, P.J.: Explaining time-table-edge-finding propagation for the cumulative resource constraint. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings. pp. 234–250 (2013)
12. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Explaining the cumulative propagator. Constraints **16**(3), 250–282 (Aug 2010). https://doi.org/10.1007/s10601-010-9103-2, https://doi.org/10.1007/s10601-010-9103-2
13. Stuckey, P.J.: Lazy clause generation: Combining the power of SAT and CP (and MIP?) solving. Lecture Notes in Computer Science pp. 5–9 (2010)
14. Vilım, P.: Timetable edge finding filtering algorithm for discrete cumulative resources. Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems **6697**, 230–245 (2011)
15. Vilím, P.: Max energy filtering algorithm for discrete cumulative resources. In: International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems. pp. 294–308. Springer (2009)
16. Wolf, A., Schrader, G.: $O(n \log n)$ overload checking for the cumulative constraint and its application. In: INAP. vol. 4369, pp. 88–101. Springer (2005)