Linear-Time Filtering Algorithms for the Disjunctive Constraint

Hamed Fahimi, Claude-Guy Quimper
Université Laval
Department of Computer science and Software engineering
hamed.fahimi.1@ulaval.ca , claude-guy.quimper@ift.ulaval.ca

Abstract

We present three new filtering algorithms for the DISJUNCTIVE constraint that all have a linear running time complexity in the number of tasks. The first algorithm filters the tasks according to the rules of the time tabling. The second algorithm performs an overload check that could also be used for the CUMULATIVE constraint. The third algorithm enforces the rules of detectable precedences. We introduce the new data structure *time line* for the last two algorithms. The time line provides some constant time operations that were previously implemented in logarithmic time by the Θ-tree data structure. Experiments show that these new algorithms are competitive even for a small number of tasks and outperform existing algorithms as the number of tasks increases.

Disjunctive constraint

Let I = {$A_1,\ldots,A_n$} be a set of tasks with unknown starting times s_i, and known processing times p_i. The constraint DISJUNCTIVE([$s_1,\ldots,s_n$]) is satisfied, if for all pairs of tasks i and j ($i \neq j$)

$$s_i + p_i \leq s_j \text{ or } s_j + p_j \leq s_i$$

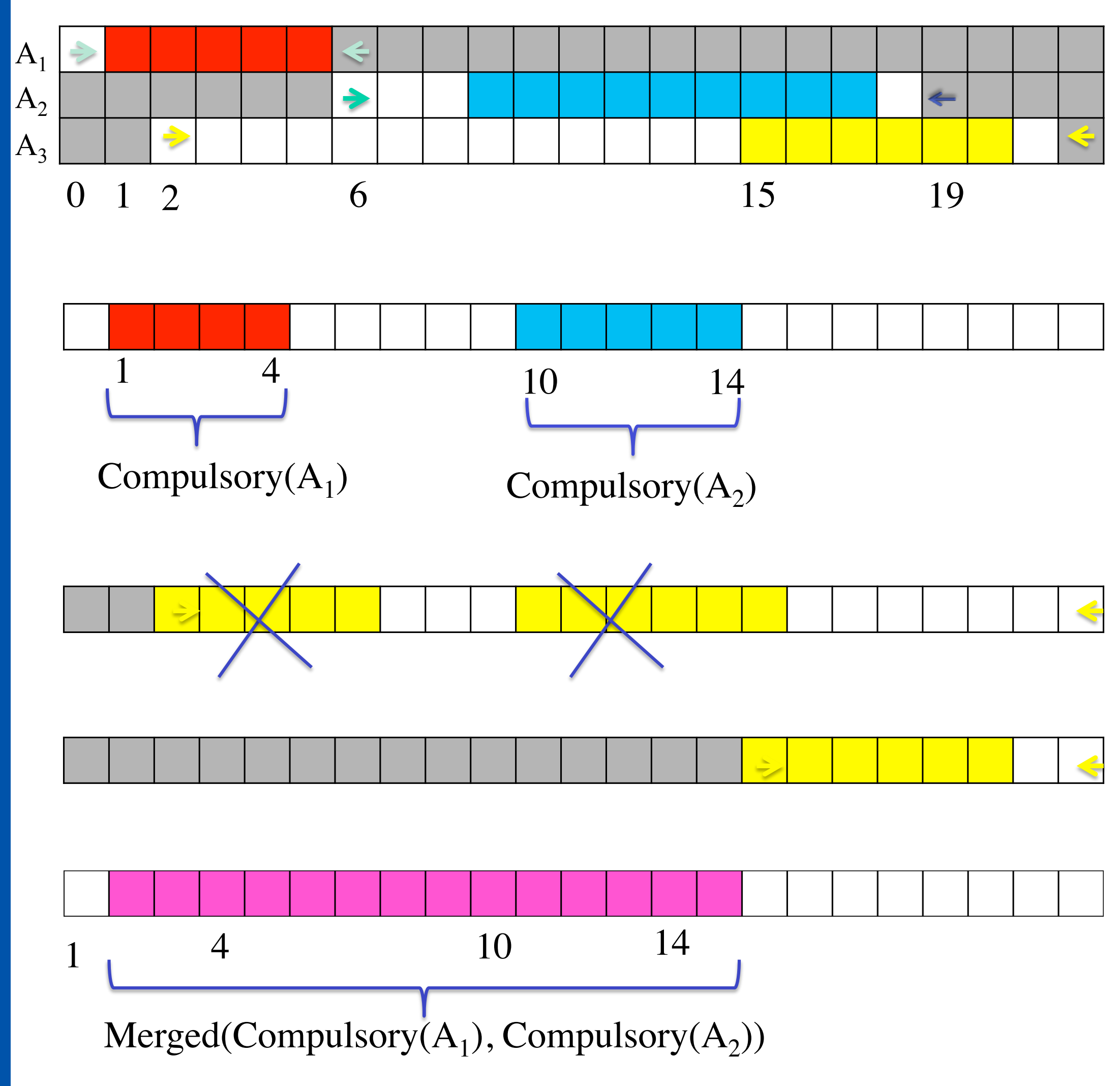
Time-tabling

This rule exploits the fact that a task i must execute within its *compulsory part* [lst_i , ect_i), if $\text{lst}_i < \text{ect}_i$. Consequently, if there exists a task i with $\text{lst}_i < \text{ect}_i$ and there exists a task j that satisfies $\text{ect}_j > \text{lst}_i$, then j has to execute after i.

$$\text{lst}_i < \text{ect}_i \wedge \text{lst}_i < \text{ect}_j \Rightarrow \text{est}'_j = \max(\text{est}_j, \text{ect}_i)$$

Strategy of the algorithm: The algorithm sorts the tasks in non-decreasing order of the processing times. While iterating through the sorted tasks, it jumps over the tasks which have compulsory parts and schedules at the earliest possible time point. Afterwards, the union find merges the traversed compulsory parts. This operation reduces the number of jumps and leads to a linear time running after processing all the tasks.

Example



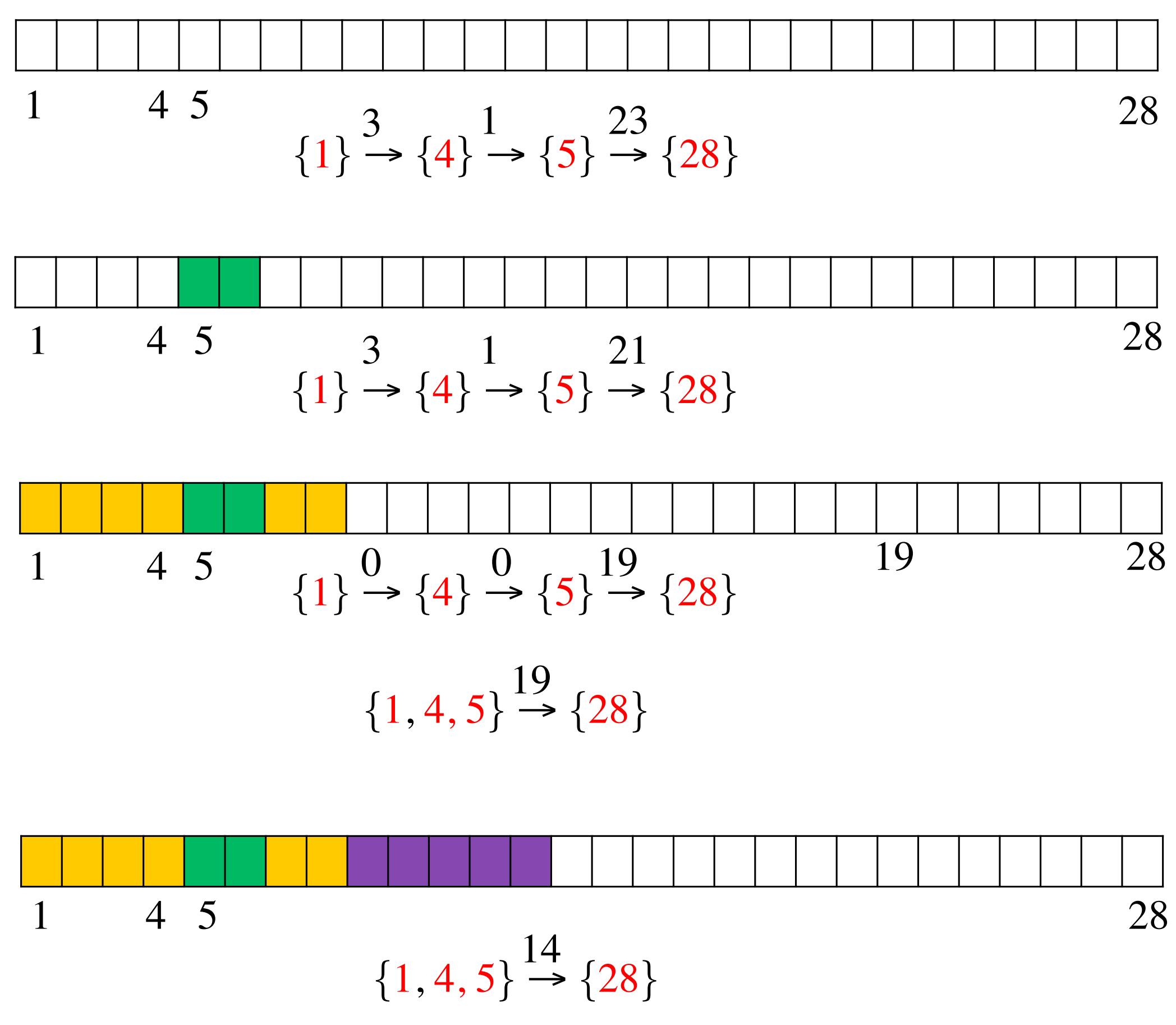
The time line data structure

Time line is a line with markers for important time points. These time points are the release times of the tasks and one time point that is late enough. Between each two consecutive time points, there is a capacity that denotes the amount of time that the resource is available through. Initially, the capacities are equal to the difference between the consecutive time points. The tasks are scheduled one by one. Adding a task runs in constant time.

After scheduling, the capacities decrease. Once a capacity between two time points equals null, the corresponding time points are merged by the Union-Find and the earliest completion time is computed in constant time.

Example

est_i	lct_i	p_i
5	8	2
1	10	6
4	15	5


1 4 5 28
$\{1\} \xrightarrow{3} \{4\} \xrightarrow{1} \{5\} \xrightarrow{23} \{28\}$
1 4 5 28
$\{1\} \xrightarrow{3} \{4\} \xrightarrow{1} \{5\} \xrightarrow{21} \{28\}$
1 4 5 19 28
$\{1\} \xrightarrow{0} \{4\} \xrightarrow{0} \{5\} \xrightarrow{19} \{28\}$
$\{1,4,5\} \xrightarrow{19} \{28\}$
1 4 5 28
$\{1,4,5\} \xrightarrow{14} \{28\}$

The earliest completion time is computed in constant time with 28-14 = 14.

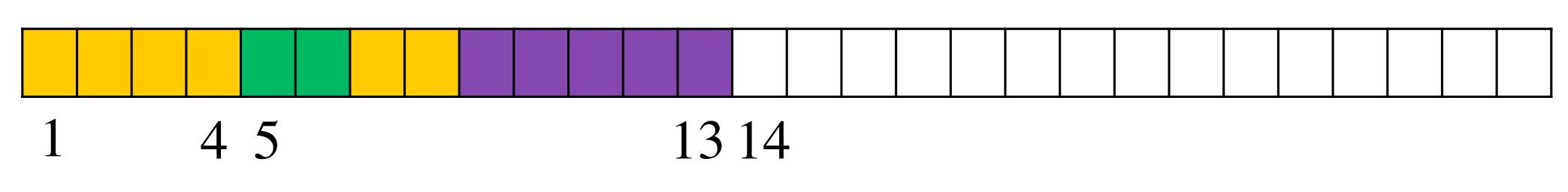
Overload check

This rule does not filter the search space. Rather, it detects an inconsistency and triggers a backtrack during the search process.

Strategy of the algorithm: The tasks are scheduled in non-decreasing order of the latest completion times with the time line. If after scheduling a task i, the earliest completion time is greater than lct_i, then the overload check fails.

$$\text{ect}_\Omega > \text{lct}_\Omega \Rightarrow \text{Fail}$$

Example
In the previous example, the overload check fails if $\text{lct}_3 = 13$.



Detectable Precedences

This technique consists of finding the following set for a task i:

$$\Omega_i = \{j \in \mathcal{I} \setminus \{i\} \mid \text{ect}_i > \text{lst}_j\}$$

Once this set is fully detected, one can delay the earliest starting time of i up to $\text{ect}_{\Omega i}$.

$$\text{est}'_i = \max(\text{est}_i, \text{ect}_{\{j \in \mathcal{I} \setminus \{i\} | \text{ect}_i > \text{lst}_i\}})$$

Strategy of the algorithm: We introduced a new algorithm to enforce the rule of detectable precedences. The algorithm simultaneously iterates over all the tasks i in non-decreasing order of ect_i and on all the tasks k in non-decreasing order of lst_k, finds the detectable precedences and filters the domains at the end of iterating through i.

```
DetectablePrecedences ($\mathcal{I}$)
   InitializeTimeline ($\mathcal{I}$)
   $j \leftarrow 0$
   $k \leftarrow \mathcal{I}_{\text{lst}}[j]$
   postponed_tasks $\leftarrow \emptyset$
   blocking_task $\leftarrow$ null
1  for $i \in \mathcal{I}_{\text{ect}}$ do
2      while $j < |\mathcal{I}| \wedge \text{lst}_k < \text{ect}_i$ do
           if $lst_k \geq \text{ect}_k$ then
3              ScheduleTask ($k$)
           else
               if blocking_task $\neq$ null then
                   return Inconsistent
               blocking_task $\leftarrow k$
           $j \leftarrow j + 1$
           $k \leftarrow \mathcal{I}_{\text{lst}}[j]$
       if blocking_task $=$ null then
4          $\text{est}'_i \leftarrow \max(\text{est}_i,$
                       EarliestCompletionTime())
       else
           if blocking_task $= i$ then
               $\text{est}'_i \leftarrow \max(\text{est}_i,$
                           EarliestCompletionTime())
               ScheduleTask (blocking_task)
               for $z \in$ postponed_tasks do
5                  $\text{est}'_z \leftarrow \max(\text{est}_z,$
                               EarliestCompletionTime())
               blocking_task $\leftarrow$ null
               postponed_tasks $\leftarrow \emptyset$
           else
               postponed_tasks $\leftarrow$ postponed_tasks $\cup \{i\}$
   for $i \in \mathcal{I}$ do $\text{est}_i \leftarrow \text{est}'_i$
```

Experiments

Here are the tables of our experiments on the benchmarks for the *open-shop problem* and *job-shop problem* with *n* jobs and m tasks per job. The numbers indicate the ratio of the cumulative number of backtracks performed by our algorithms and the number of backtracks performed by the state-of-the-art algorithms. A ratio greater than 1 means that, within 10 minutes of computations, the state-of-the-art algorithms explore a larger portion of the search tree.

$n \times m$	OC	DP	TT
4×4	0.96	1.00	1.00
5×5	1.03	1.12	1.75
7×7	1.02	1.16	2.09
10×10	1.06	1.33	2.14
15×15	1.03	1.39	2.15
20×20	1.06	1.56	2.17
p-value	0.25	8.28E-14	5.95E-14

Open-shop problem

$n \times m$	OC	DP	TT
10×5	1.07	1.27	2.11
15×5	1.02	1.35	2.27
20×5	1.00	1.55	2.12
10×10	1.01	1.25	2.18
15×10	1.26	1.42	1.97
20×10	1.00	1.47	2.14
30×10	1.08	1.56	2.36
50×10	1.05	1.48	3.18
15×15	0.95	1.48	2.16
20×15	1.04	1.61	2.13
20×20	1.09	1.46	1.71
p-value	0.17	1.41E-12	3.38E-20

Job-shop problem

Conclusion

Thanks to the constant time operation of the Union-Find data structure, we designed the new time line data structure, to speed up filtering algorithms for the Disjunctive constraint.

We came up with three faster algorithms to filter the disjunctive constraint. The following table exhibits the results.

Algorithm	**Previous complexity**	**Improved complexity**
Time-Tabling	O(n log(n)) (Ouellet & Quimper)	O(n) (Fahimi & Quimper)
Overload check	O(n log(n)) (Vilím)	O(n) (Fahimi & Quimper)
Detectable precedences	O(n log(n)) (Vilím)	O(n) (Fahimi & Quimper)