Linear-Time Filtering Algorithms for the Disjunctive Constraint

Hamed Fahimi
Université Laval
Département d'informatique et de génie logiciel
hamed.fahimi.1@ulaval.ca

Claude-Guy Quimper
Université Laval
Département d'informatique et de génie logiciel
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. The two last algorithms use a new data structure that we introduce and that we call the time line. This data structure provides many 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.

Introduction

Constraint programming offers many ways to model and solve scheduling problems. The DISJUNCTIVE constraint allows to model problems where the tasks cannot be executed concurrently. The CUMULATIVE constraint models the problems where a limited number of tasks can execute simultaneously. With these constraints come multiple filtering algorithms that prune the search space. Since these algorithms are called thousands of times during the search, it is essential to design them for efficiency. Data structures can contribute to the efficiency of these algorithms.

For the constraint ALL-DIFFERENT, that is a special case of the constraints DISJUNCTIVE and CUMULATIVE, (Puget 1998) proposed an $O(n \log n)$ filtering algorithm. The factor $\log n$ comes from the operations achieved on a balanced tree of depth $\log n$. This algorithm was outperformed by linear time algorithms (López-Ortiz *et al.* 2003; Mehlhorn & Thiel 2000) that both use union find data structures to achieve equivalent operations. More recently, (Vilím 2007) proposed new data structures called Θ-tree and Θ-λ tree that are balanced trees of depth $\log n$. These data structures led to filtering algorithms for the DISJUNCTIVE and CUMULATIVE constraints including filtering algorithms based on overload check, not-first / not-last, detectable precedences (Vilím, Barták, & Čepek 2004), edge-finder (Vilím 2009), extended-edge-finder, and time-table-extended-edge-finder (Ouellet & Quimper 2013). These algorithms have a $\log n$ factor in their running time complexities that originates from the balanced tree. We propose to modify some of these algorithms using a union find data structure, as it was done for the ALL-DIFFERENT, to obtain linear time filtering algorithms.

The paper is divided as follows. We review the disjunctive constraint as well as three filtering techniques: time tabling, overload check, and detectable precedences. We introduce a new algorithm for time-tabling. We propose the time line data structure. We present an overload check and an algorithm applying the detectable precedences rules using the new time line data structure. Finally, we show experimental results and conclude.

The Disjunctive Constraint

We consider the scheduling problem where n tasks (denoted $\mathcal{I} = \{1, \ldots, n\}$) compete to be executed, one by one, without interruption, on the same resource. Each task has an earliest starting time $\text{est}_i \in \mathbb{Z}$, a latest completion time $\text{lct}_i \in \mathbb{Z}$, and a processing time $p_i \in \mathbb{Z}^+$. From these properties, one can compute the latest starting time $\text{lst}_i = \text{lct}_i - p_i$ and the earliest completion time $\text{ect}_i = \text{est}_i + p_i$. Let S_i be the starting time of task i with domain $\text{dom}(S_i) = [\text{est}_i, \text{lst}_i]$. The constraint DISJUNCTIVE$([S_1, \ldots, S_n])$ is satisfied when $S_i + p_i \leq S_j \vee S_j + p_j \leq S_i$ for all pairs of tasks $i \neq j$. A solution to the DISJUNCTIVE constraint is a solution to the disjunctive scheduling problem.

It is NP-complete to decide whether the DISJUNCTIVE constraint is satisfiable and therefore it is NP-hard to enforce bounds consistency on this constraint. However, when $p_i = 1$, the constraint becomes the ALL-DIFFERENT constraint and bounds consistency can be achieved in linear time (López-Ortiz *et al.* 2003). When $p_i = p_j$ for all $i, j \in \mathcal{I}$, the constraint becomes an INTER-DISTANCE constraint and bounds consistency can be achieved in quadratic time (Quimper, López-Ortiz, & Pesant 2006).

The earliest starting time, the latest completion time, and the processing time can be generalized to a set of tasks.

$$\text{est}_\Omega = \min_{i \in \Omega} \text{est}_i \qquad \text{lct}_\Omega = \max_{i \in \Omega} \text{lct}_i \qquad p_\Omega = \sum_{i \in \Omega} p_i$$

The earliest completion time (latest starting time) of a set of tasks can be approximated as follows. This corresponds to

the relaxation where the tasks can be preempted.

$$\mathrm{ect}_\Omega = \max_{\emptyset \subset \Theta \subseteq \Omega} (\mathrm{est}_\Theta + p_\Theta) \quad \mathrm{lst}_\Omega = \min_{\emptyset \subset \Theta \subseteq \Omega} (\mathrm{lct}_\Theta - p_\Theta)$$

For empty sets, we have $\mathrm{ect}_\emptyset = \mathrm{lct}_\emptyset = -\infty$ and $\mathrm{est}_\emptyset = \mathrm{lst}_\emptyset = \infty$.

Even though it is NP-Hard to filter the DISJUNCTIVE constraint, there exist pruning rules that can be enforced in polynomial time. All the rules we present aim at delaying the earliest starting time of the tasks. To advance the latest completion time, one can create the symmetric problem where task i is transformed into a task i' such that $\mathrm{est}_{i'} = -\mathrm{lct}_i$, $\mathrm{lct}_{i'} = -\mathrm{est}_i$, and $p_{i'} = p_i$. Delaying the earliest completion time in the symmetric problem prunes the latest completion time in the original problem.

Time-tabling The *time-tabling* rule exploits the fact that a task i must execute within the semi-open time interval $[\mathrm{lst}_i, \mathrm{ect}_i)$ when $\mathrm{lst}_i < \mathrm{ect}_i$. This interval is called the *compulsory part*. The compulsory part of a task makes the resource unavailable for the execution of the other tasks. Consequently, if there exists a task i such that $\mathrm{lst}_i < \mathrm{ect}_i$ and there exists a task j that satisfies $\mathrm{ect}_j > \mathrm{lst}_i$, then j must execute after i.

$$\mathrm{lst}_i < \mathrm{ect}_i \wedge \mathrm{lst}_i < \mathrm{ect}_j \Rightarrow \mathrm{est}'_j = \max(\mathrm{est}_j, \mathrm{ect}_i) \quad (1)$$

Several algorithms apply the time-tabling rules (Le Pape 1988; Beldiceanu & Carlsson 2002; Beldiceanu, Carlsson, & Poder 2008; Letort, Beldiceanu, & Carlsson 2012; Ouellet & Quimper 2013). Most of them were designed for the CUMULATIVE constraint but can also be used for the more restrictive case of the DISJUNCTIVE constraint. The fastest algorithm by (Ouellet & Quimper 2013) runs in $O(n \log n)$ time.

Overload check The *overload check* does not filter the search space, but detects an inconsistency and triggers a backtrack during the search process. The overload fails when it detects a set of task $\Omega \subseteq \mathcal{I}$ for which the earliest completion time exceeds the latest completion time.

$$\mathrm{ect}_\Omega > \mathrm{lct}_\Omega \Rightarrow \text{Fail} \quad (2)$$

(Vilím 2004) proposes an algorithm that runs in $O(n \log n)$. (Wolf & Schrader 2005) propose an overload check for the CUMULATIVE constraint.

Detectable precedences The precedence relation $i \ll j$ indicates that the task i must execute before task j. The precedence can also be established between sets of tasks, for instance, $\Omega \ll i$ indicates that the task i must execute after all tasks in Ω. When $\mathrm{ect}_i > \mathrm{lst}_j$ holds, we say that the precedence $j \ll i$ is *detectable*. The technique of *detectable precedences* consists of finding, for a task i, the set of tasks $\Omega_i = \{j \in \mathcal{I} \setminus \{i\} \mid \mathrm{ect}_i > \mathrm{lst}_j\}$ for which there exists a detectable precedence with i. Once this set is discovered, one can delay the earliest starting time of i up to ect_{Ω_i}.

$$\mathrm{est}'_i = \max(\mathrm{est}_i, \mathrm{ect}_{\{j \in \mathcal{I} \setminus \{i\} \mid \mathrm{ect}_i > \mathrm{lst}_i\}}) \quad (3)$$

(Vilím 2002) proposed this filtering technique that he later improved in (Vilím 2004) to obtain an algorithm with a running time complexity of $O(n \log n)$.

Preliminaries

Let $\mathcal{I}_{\mathrm{est}}$, $\mathcal{I}_{\mathrm{lct}}$, $\mathcal{I}_{\mathrm{ect}}$, $\mathcal{I}_{\mathrm{lst}}$ $\mathcal{I}_p$ be the ordered set of tasks $\mathcal{I}$ sorted by est, lct, ect, lst, and processing times. We assume that all time points are encoded with w-bit integers and that these sets can be sorted in linear time $O(n)$. This assumption is supported by the fact that a word of $w = 32$ bits is sufficient to encode all time points, with a precision of a second, within a period longer than a century. This is sufficient for most industrial applications. An algorithm such as radix sort can sort the time points in time $O(wn)$ which is linear when w is constant. In practice, the filtering algorithms are called multiple times during the search process and a constant number of tasks are modified between each call. In such cases, algorithms as simple as insertion sort can resort the tasks in linear time.

The new algorithms we present rely on the Union-Find data structure. The function $\texttt{UnionFind}(n)$ initializes n disjoint sets $\{0\}, \{1\}, \ldots, \{n-1\}$ in $O(n)$ steps. The function $\texttt{Union}(a, b)$ merges the set that contains element a with the set that contains the element b. The functions $\texttt{FindSmallest}(a)$ and $\texttt{FindGreatest}(a)$ return the smallest and greatest element of the set that contains a. These three functions run in $O(\alpha(n))$ steps, where α is Ackermann's inverse function. (Cormen *et al.* 2001) present how to implement this data structure using trees. The smallest and greatest element of each set can be stored in the root of these trees. This implementation is the fastest in practice. However, we use this data structure in a very specific context where the function $\texttt{Union}(a, b)$ is called only when $\texttt{FindGreatest}(a) + 1 = \texttt{FindSmallest}(b)$. Such a restriction allows to use the Union-Find data structure as presented by (Gabow & Tarjan 1983) that implement the functions $\texttt{Union}(a, b)$, $\texttt{FindSmallest}(a)$ and $\texttt{FindGreatest}(a)$ in constant time. This implementation is the fastest in theory, but not in practice due to a large hidden constant.

Time-Tabling

We present a linear time algorithm that enforces the time-tabling rule. Like most time-tabling algorithms, this new algorithm is not idempotent. However, it provides some guaranties on the level of filtering it achieves. Consider the set of compulsory parts $\mathcal{F} = \{[\mathrm{lst}_i, \mathrm{ect}_i) \mid i \in \mathcal{I} \wedge \mathrm{lst}_i < \mathrm{ect}_i\}$. Consider a task $j \in \mathcal{I}$. The algorithm guarantees that after the filtering occurs, the interval $[\mathrm{est}'_j, \mathrm{ect}'_j)$ does not intersect with any intervals in $\mathcal{F}$. However, the pruning of est_j to est'_j might create a new compulsory part $[\mathrm{lst}_j, \mathrm{ect}'_j)$ that could cause some filtering in a further execution of the algorithm.

Algorithm 1 proceeds in three steps, each of them is associated to a for loop. The first for loop on line 1 creates the vectors l and u that contain the lower bounds and upper bounds of the compulsory parts. The compulsory parts $[l[0], u[0]), [l[1], u[1]), \ldots, [l[m-1], u[m-1])$ form a sequence of sorted and disjoint semi-open intervals such that each of them is associated to a task i that satisfies $\mathrm{lst}_i < \mathrm{ect}_i$. If two compulsory parts overlap, the algorithm, on line 2, returns *Inconsistent*. When processing the task i

Algorithm 1: `TimeTabling` ($\mathcal{I}$)

$m \leftarrow 0, k \leftarrow 0, l \leftarrow [\,], u \leftarrow [\,], r \leftarrow [\,]$;
$\text{est}'_i \leftarrow \text{est}_i, \forall\, i \in \mathcal{I}$;
1 **for** $i \in \mathcal{I}_{\text{lst}}$ **do**
  if $\text{lst}_i < \text{ect}_i$ **then**
    if $m > 0$ **then**
2   **if** $u[m-1] > \text{lst}_i$ **then return** Inconsistent;
3   **else** $\text{est}'_i \leftarrow \max(\text{est}'_i, u[m-1])$;
    l.append(lst_i);
    u.append($\text{est}'_i + p_i$);
    $m \leftarrow m + 1$;
if $m = 0$ **then return** Consistent;
4 **for** $i \in \mathcal{I}_{\text{est}}$ **do**
  while $k < m \wedge \text{est}_i \geq u[k]$ **do**
    $k \leftarrow k + 1$;
  $r[i] \leftarrow k$;
$s \leftarrow$ `UnionFind`(m);
5 **for** $i \in \mathcal{I}_p$ **do**
  if $\text{ect}_i \leq \text{lst}_i$ **then**
    $c \leftarrow r[i]$;
    first_update $\leftarrow$ True;
6   **while** $c < m \wedge \text{est}'_i + p_i > l[c]$ **do**
      $c \leftarrow s$.`FindGreatest`(c);
      $\text{est}'_i \leftarrow \max(\text{est}'_i, u[c])$;
      if $\text{est}'_i + p_i > \text{lct}_i$ **then return** Inconsistent;
      if $\neg$*first_update* **then** s.`Union`$(r[i], c)$;
      first_update $\leftarrow$ False;
      $c \leftarrow c + 1$;
return Consistent;

that has a compulsory part $[l[k], u[k])$, the algorithm makes sure, on line 3 that the task i starts no earlier than $u[k-1]$, so that the tasks that have a compulsory part are all filtered.

The second for loop on line 4 creates a vector r that maps a task i to the compulsory part whose upper bound is the smallest one to be greater than est_i. We therefore have the relation $u[r[i]-1] \leq \text{est}_i < u[r[i]]$.

The third for loop on line 5 filters the tasks that do not have a compulsory part. The tasks are processed by non-decreasing order of processing times. Line 6 checks whether $\text{est}'_i + p_i > l[r[i]]$. If so, then the time-tabling rule applies and the new value of est'_i is pruned to $u[c]$. The same task is then checked against the next compulsory part $[l[r[i]+1], u[r[i]+1])$ and so on. Suppose that a task is filtered both by the compulsory part $[l[c], u[c])$ and the compulsory part $[l[c+1], u[c+1])$. Since we process the tasks by non-decreasing order of processing time, any further task that is filtered by the compulsory part $[l[c], u[c])$ will also be filtered by the compulsory part $[l[c+1], u[c+1])$. The algorithm uses a Union-Find data structure to keep track that these two compulsory parts are *glued* together. The next task j that satisfies $\text{est}'_j + p_j > l[c]$ will be filtered to $u[c+1]$ in a single iteration. The Union-Find data structure can union an arbitrary long sequence of compulsory parts.

Theorem 1 *Algorithm 1 enforces the time-tabling rule in $O(n)$ steps.*

Proof: Each of the two first for loops iterate through the tasks once and execute operations in constant time. Each time the while loop on line 6 executes more than once, the Union-Find data structure merges two compulsory parts. This can occur at most n times. □

The Time Line Data Structure

We introduce the *time line* data structure. This data structure is initialized with an empty set of tasks $\Theta = \emptyset$. It is possible to add, in constant time, a task to Θ and to compute, in constant time, the earliest completion time ect_Θ. (Vilím 2004) proposes the Θ-tree data structure that supports the same operations. It differs in two points from the time line. Inserting a task in a Θ-tree requires $O(\log n)$ steps. Removing a task from a Θ-tree is done in $O(\log n)$ steps while this operation is not supported in a time line. The time line is therefore faster than a Θ-tree but can only be used for algorithms where the removal of a task is not needed.

The data structure is inspired from (López-Ortiz *et al.* 2003). We consider a sequence $t[0..|t|-1]$ of unique time points sorted in chronological order formed by the earliest starting times of the tasks and a sufficiently large time point, for instance $\max_{i \in \mathcal{I}} \text{lct}_i + \sum_{i=1}^{n} p_i$. The vector $m[0..n-1]$ maps a task i to the time point index such that $t[m[i]] = \text{est}_i$. The time points, except for the last one, have a capacity stored in the vector $c[0..|t|-2]$. The capacity $c[a]$ denotes the amount of time the resource is available within the semi-open time interval $[t[a], t[a+1])$ should the tasks in Θ be scheduled at their earliest starting time with preemption. Initially, since $\Theta = \emptyset$, the resource is fully available and $c[a] = t[a+1] - t[a]$. A Union-Find data structure s is initialized with $|t|$ elements. This data structure will maintain the invariant that a and $a+1$ belong to the same set in s if and only if $c[a] = 0$. This will allow us to quickly request, by calling s.`FindGreatest`(a), the earliest time point no earlier than $t[a]$ with a positive capacity. Finally, the data structure has an index e which is the index of the latest time point whose capacity has been decremented. Algorithm 2 initializes the components t, m, c, s, and e that define the time line data structure.

Algorithm 2: `InitializeTimeline` ($\mathcal{I}$)

$t \leftarrow [\,], c \leftarrow [\,]$;
for $i \in \mathcal{I}_{\text{est}}$ **do**
  if $|t| = 0 \vee t[|t|-1] \neq \text{est}_i$ **then** t.append(est_i);
  $m[i] \leftarrow |t| - 1$;
t.append($\max_i \text{lct}_i + \sum_{i=1}^{n} p_i$);
for $k = 0..|t|-2$ **do** $c[k] \leftarrow t[k+1] - t[k]$;
$s \leftarrow$ `UnionFind`$(|t|)$;
$e \leftarrow -1$;

The data structure allows to schedule a task i over the time line at its earliest time and with preemption. The value $m[i]$ maps the task i to the time point associated to the earliest starting time of task i. Algorithm 3 iterates through the time

intervals $[t[m[i]], t[m[i]+1]), [t[m[i]+1], t[m[i]+2]), \ldots$ and decreases the capacity of each interval down to 0 until a total of p_i units of capacity is decreased. Each time a capacity $c[k]$ reaches zero, the union-find merges the index k with $k+1$ which allows, in the future, to skip arbitrarily long sequences of intervals with null capacities in constant time.

Algorithm 3: `ScheduleTask` (i)

```
ρ ← p_i;
k ← s.FindGreatest(m[i]);
while ρ > 0 do
    Δ ← min(c[k], ρ);
    ρ ← ρ − Δ;
    c[k] ← c[k] − Δ;
    if c[k] = 0 then
        s.Union(k, k + 1);
        k ← s.FindGreatest(k);
e ← max(e, k);
```

Let Θ be the tasks that were scheduled using Algorithm 3, then Algorithm 4 computes in constant time the earliest completion time ect_Θ.

Algorithm 4: `EarliestCompletionTime` ()

return t[e + 1] - c[e]

Example 1 Consider three tasks whose parameters $(\mathrm{est}_i, \mathrm{lct}_i, p_i)$ are in $\{(4, 15, 5), (1, 10, 6), (5, 8, 2)\}$. Initializing the time line produces the structure $\{1\} \overset{3}{\to} \{4\} \overset{1}{\to} \{5\} \overset{23}{\to} \{28\}$ where the numbers in the sets are time points and numbers on the arrows are capacities. After scheduling the first task, the capacity between the time points 4 and 5 becomes null and the union-find merges both time points into the same set. The structure becomes $\{1\} \overset{3}{\to} \{4, 5\} \overset{19}{\to} \{28\}$. After scheduling the second task, the time line becomes $\{1, 4, 5\} \overset{16}{\to} \{28\}$ and after scheduling the last task, it becomes $\{1, 4, 5\} \overset{14}{\to} \{28\}$. The earliest completion time is given by $28 - 14 = 14$.

Theorem 2 *Algorithm 2 runs in $O(n)$ amortized time while Algorithm 3 and Algorithm 4 run in constant amortized time.*

Proof: Let c_i be the capacity vector after the ith call to an algorithm among Algorithm 2, Algorithm 3, and Algorithm 4. We define a potential function $\phi(i) = |\{k \in 0..|t| - 2 \mid c_i[k] > 0\}|$ that is equal to the number of positive components in the vector c_i. Prior to the initialization of the time line data structure, we have $\phi(0) = 0$ since the capacity vector is not even initialized and in all time, we have $\phi(i) \geq 0$. After the initialization, we have $\phi(1) = |t| - 1 \leq n$. The two for loops in Algorithm 2 execute $n + |t| - 1 \leq 2n \in O(n)$ times. Therefore, the amortized complexity of the initialization is $O(n) + \phi(1) - \phi(0) = O(n)$.

Suppose the while loop in Algorithm 3 executes a times. There are at least $a - 1$ and at most a components in the capacity vector that are set to zero hence $a - 1 \leq \phi(i) - \phi(i-1) \leq a$. The amortized complexity of Algorithm 3 is therefore $a + \phi(i) - \phi(i-1) \leq a - (a-1) \in O(1)$.

Algorithm 4 executes in constant time and does not modify the capacity vector c which implies $\phi(i) = \phi(i-1)$. The amortized complexity is therefore $O(1) + \phi(i) - \phi(i-1) = O(1)$. □

Overload check

The overload check, as described by (Vilím 2004), can be directly used with a time line data structure rather than a Θ-tree. One schedules the tasks, using Algorithm 3, in non-decreasing order of latest completion times. If after scheduling a task i, Algorithm 4 returns an earliest completion time greater than lct_i, then the overload check fails. The total running time complexity of this algorithm is $O(n)$. The proof of correctness is identical to Vilím's.

The overload check can be adapted to the CUMULATIVE constraint with a resource of capacity C. One can transform the task i of height h_i into a task i' with $\mathrm{est}_{i'} = C\,\mathrm{est}_i$, $\mathrm{lct}_{i'} = C(\mathrm{lct}_i + 1) - 1$, and $p_{i'} = h_i p_i$. The overload check fails on the original problem if and on if it fails on the transformed model. The transformation preserves the running time complexity of $O(n)$.

Detectable Precedences

We introduce a new algorithm to enforce the rule of detectable precedences. One cannot simply adapt the algorithm in (Vilím, Barták, & Čepek 2004) for the time line data structure as it requires to temporarily remove a task among the scheduled tasks which is an operation the time line cannot do. Algorithm 5 applies the rules of the detectable precedences in linear time using the time line data structure.

Suppose that the problem has no tasks with a compulsory part, i.e. $\mathrm{ect}_i \leq \mathrm{lst}_i$ for all task $i \in \mathcal{I}$. The algorithm simultaneously iterates over all the tasks i in non-decreasing order of earliest completion times and on all the tasks k in non-decreasing order of latest starting times. Each time the algorithm iterates over the next task i, it iterates (line 2) and schedules (line 3) all tasks k whose latest starting time lst_k is smaller than the earliest completion time ect_i. Once the while loop completes, the set of scheduled tasks is $\{k \in \mathcal{I} \setminus \{i\} \mid \mathrm{lst}_k < \mathrm{ect}_i\}$. We apply the detectable precedence rule by pruning the earliest starting time of task i up to the earliest completion time of the time line (line 4).

Suppose that there exists a task k with a compulsory part, i.e. $\mathrm{ect}_k > \mathrm{lst}_k$. This task could be visited in the while loop before being visited in the main for loop. We do not want to schedule the task k before it is filtered. We therefore call the task k the *blocking task*. When a blocking task k is encountered in the while loop, the algorithm waits to encounter the same task in the for loop. During this waiting period, the filtering of all tasks is postponed. A postponed task i necessarily satisfies the conditions $\mathrm{lst}_k < \mathrm{ect}_i \leq \mathrm{ect}_k$ and $\mathrm{ect}_i < \mathrm{lst}_i$ and therefore the precedence $k \ll i$ holds. When the for loop reaches the blocking task k, it filters the blocking task, schedules the blocking task, and filters the postponed tasks. The blocking task and the set of postponed

Algorithm 5: `DetectablePrecedences` ($\mathcal{I}$)

```
   InitializeTimeline (I)
   j ← 0
   k ← I_lst[j]
   postponed_tasks ← ∅
   blocking_task ← null
1  for i ∈ I_ect do
2      while j < |I| ∧ lst_k < ect_i do
           if lst_k ≥ ect_k then
3              ScheduleTask (k)
           else
               if blocking_task ≠ null then
                   return Inconsistent
               blocking_task ← k
           j ← j + 1
           k ← I_lst[j]
       if blocking_task = null then
4          est'_i ← max(est_i,
                        EarliestCompletionTime())
       else
           if blocking_task = i then
               est'_i ← max(est_i,
                            EarliestCompletionTime())
               ScheduleTask (blocking_task)
               for z ∈ postponed_tasks do
5                  est'_z ← max(est_z,
                                EarliestCompletionTime())
               blocking_task ← null
               postponed_tasks ← ∅
           else
               postponed_tasks ← postponed_tasks ∪ {i}
   for i ∈ I do est_i ← est'_i
```

tasks are reset. It is not possible to simultaneously have two blocking tasks since their compulsory parts would overlap, which is inconsistent with the time-tabling rule.

Example 2 Figure 1 shows a trace of the algorithm. The for loop on line 1 processes the tasks $\mathcal{I}_{\text{ect}} = \{1, 2, 3, 4\}$ in that order. For the two first tasks 1 and 2, nothing happens: the while loop is not executed and no pruning occurs as no tasks are scheduled on the time line. When the for loop processes task 3, the while loop processes three tasks. The while loop processes the task 2 which is scheduled on the time line. When it processes task 4, the while loop detects that task 4 has a compulsory part in $[14, 18)$ making task 4 the blocking task. Finally, the while loop processes task 1 which is scheduled on the time line. Once the while loop completes, the task 3 is not filtered since there exists a blocking task. Its filtering is postponed until the blocking task is processed. Finally, the for loop processes the task 4. In this iteration, the while loop does not execute. Since task 4 is the blocking task, it is first filtered to the earliest completion time computed by the time line data structure ($\text{est}'_4 \leftarrow 13$). Task 4 is then scheduled on the time line. Finally, the postponed task 3 is filtered to the earliest completion time computed by the time line data structure ($\text{est}'_3 \leftarrow 19$).

i	est_i	lct_i	p_i	ect_i	lst_i	postponed_tasks	est'_i
1	0	19	4	4	15	$\emptyset$	0
2	2	22	9	11	13	$\emptyset$	2
3	9	30	7	16	23	$\{3\}$	19
4	12	20	6	18	14	$\emptyset$	13

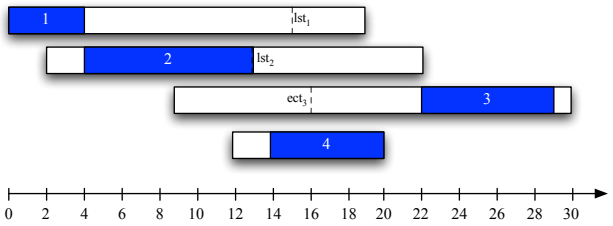

Figure 1: The tasks $\mathcal{I}_{\text{ect}} = \{1, 2, 3, 4\}$ and the visual representation of a solution to the DISJUNCTIVE constraint. The algorithm `DetectablePrecedences` prunes the earliest starting times $\text{est}'_3 = 19$ and $\text{est}'_4 = 13$.

Theorem 3 *The algorithm* `DetectablePrecedences` *runs in linear time.*

Proof: The for loop on line 1 processes each task only once, idem for the while loop. Finally, a task can be postponed only once during the execution of the algorithm and therefore line 5 is executed at most n times. Except for `InitializeTimeline` and the sorting of $\mathcal{I}_{\text{ect}}$ and $\mathcal{I}_{\text{lst}}$ that are executed once in $O(n)$ time, all other operations execute in amortized constant time. Therefore, `DetectablePrecedences` runs in linear time. □

Experimental Results

We experimented with the job-shop and open-shop scheduling problems where n jobs, consisting of a set of non-preemptive tasks, execute on m machines. Each task executes on a predetermined machine with a given processing time. In the job-shop problem, the tasks belonging to the same job execute in a predetermined order. In the open-shop problem, the number of tasks per job is fixed to m and the order in which the tasks of a job are processed is not provided. In both problems, the goal is to minimize the makespan.

We model the problems with a starting time variable $S_{i,j}$ for each task j of job i. We post a DISJUNCTIVE constraint over the starting time variables of tasks running on the same machine. For the job-shop scheduling problem, we add the precedence constraints $S_{i,j} + p_{i,j} \leq S_{i,j+1}$. For the open-shop scheduling problem, we add a DISJUNCTIVE constraint among all tasks of a job. We use the benchmark provided by (Taillard 1993) that includes 82 and 60 instances of the job-shop and open-shop problems.

We implemented our algorithms in Choco 2.1.5 and, as a point of comparison, the overload check and the detectable precedences from (Vilím 2004) as well as the time tabling algorithm from (Ouellet & Quimper 2013). For the six algorithms, we sort the tasks using the function `Arrays.sort`

	Overload Check			Detectable Precedences			Time-Tabling		
n	Θ-tree (ms)	time line (ms)	bt	Θ-tree (ms)	time line (ms)	bt	Ouellet et al. (ms)	Union-Find (ms)	bt
10	11420	**10716**	142843	7559	**7519**	6803	18652	**15545**	154202
20	7751	**7711**	377305	17311	**14847**	322384	11313	**8902**	140229
30	9606	**9412**	443407	13326	**11109**	136142	11772	**8984**	139346
40	4433	**4112**	5969	19098	**16493**	115986	9551	**7205**	62901
50	5904	**5299**	34454	14895	**12012**	65043	3487	**2871**	3082
60	6150	**5250**	27491	7816	**6952**	3995	6300	**5107**	2612
70	5508	**4737**	17894	5425	**4495**	1514	5505	**3940**	22766
80	28800	**26236**	201453	5915	**4942**	481	2965	**2148**	317
90	31480	**29461**	174305	10016	**7993**	32318	3708	**2939**	509
100	48686	**46104**	262883	9879	**8156**	2360	7393	**5564**	1190

Table 1: Random instances with n tasks. Times are reported in milliseconds. Algorithms implementing the same filtering technique lead to the same number of backtracks (bt).

$n \times m$	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

Table 2: Open-shop with n jobs and m tasks per job. Ratio of the cumulative number of backtracks between all instances of size $n \times m$ after 10 minutes of computations. OC: our overload check vs. Vilím's. DP: our detectable precedences vs Vilím's. TT: Our time tabling vs Ouellet et al.

provided by Java 1.7. All experiments were run on an Intel Xeon X5560 2.667GHz quad-core processor. We used the impact based search heuristic with a timeout of 10 minutes. Each filtering algorithm is individually tested, i.e. we did not combine the filtering algorithms. For the few instances that were solved to optimality within 10 minutes, the two filtering algorithms of the same technique, whether it is overload check, detectable precedences, or time tabling, produce the same number of backtracks since they achieve the same filtering. To compare the algorithms, we sum up, for each instance of the same size, the number of backtracks achieved within 10 minutes and we report the ratio of these backtracks between both algorithms. A ratio greater than 1 indicates that our algorithm explores a larger portion of the search tree and thus is faster. We also ran a Student's t-Test on all instances to verify whether the new algorithms are faster. Table 2 and Table 3 present the results.

The new overload check seems to be faster with 10 tasks and more. However, on all instances, the p-values prevent us from drawing a conclusion about its performance.

The new algorithm of detectable precedences shows improvements on both problems especially when the number of variables increases. The p-values confirm our hypothesis. One way to explain why the ratios are greater than with the overload check is that the most costly operations in Vilím's algorithm is the insertion and removal of a task in the Θ-tree which can occur up to 3 times for each task. With the new

$n \times m$	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

Table 3: Job-shop with n jobs and m tasks per job. Ratio of the cumulative number of backtracks between all instances of size $n \times m$ after 10 minutes of computations. OC: our overload check vs. Vilím's. DP: our detectable precedences vs Vilím's. TT: Our time tabling vs Ouellet et al.

algorithm, the most costly operation is the scheduling of a task on the time line which occurs only once per task.

Student's test confirms that the new time tabling algorithm is faster. Ratios are higher than with the algorithm by (Ouellet & Quimper 2013) since the latter one was designed for the CUMULATIVE constraint.

We randomly generated large but easy instances with a single DISJUNCTIVE constraint over variables with uniformly generated domains. Unsatisfiable instances and instances solved with zero backtracks were discarded. Table 1 shows that the new algorithms are consistently faster.

Conclusion

We introduced a new data structure, called the time line. We took advantage of this data structure to present three new filtering algorithms for the disjunctive constraint that all have a linear running time complexity in the number of tasks. Moreover, the overload check can be adapted to the CUMULATIVE constraint. The new algorithms outperform the best algorithms known so far. Future works include the adaptation of the algorithms to the problems with optional tasks.

Acknowledgment

Thanks to Giovanni Won Dias Baldini Victoret for his help in the experiments.

References

Beldiceanu, N., and Carlsson, M. 2002. A new multi-resource cumulatives constraint with negative heights. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP 2002)*, 63–79.

Beldiceanu, N.; Carlsson, M.; and Poder, E. 2008. New filtering for the cumulative constraint in the context of non-overlapping rectangles. In *Proceedings of the 5th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CPAIOR 2008)*, 21–35.

Cormen, T. H.; Stein, C.; Rivest, R. L.; and Leiserson, C. E. 2001. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition.

Gabow, H. N., and Tarjan, R. E. 1983. A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the 15th annual ACM symposium on Theory of computing*, 246–251.

Letort, A.; Beldiceanu, N.; and Carlsson, M. 2012. A scalable sweep algorithm for the cumulative constraint. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP 2012)*, 439–454.

López-Ortiz, A.; Quimper, C.-G.; Tromp, J.; and van Beek, P. 2003. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, 245–250.

Le Pape, C. 1988. *Des systèmes d'ordonnancement flexibles et opportunistes*. Ph.D. Dissertation, Universit Paris IX.

Mehlhorn, K., and Thiel, S. 2000. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. *Sixth International Conference on Principles and Practice of Constraint Programming*.

Ouellet, P., and Quimper, C.-G. 2013. Time-table-extended-edge-finding for the cumulative constraint. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP 2013)*, 562–577.

Puget, J.-F. 1998. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the 15th National Conference on Artificiel Intelligence (AAAI-98) and the 10th Conference on Innovation Applications of Artificial Intelligence (IAAI-98)*, 359–366.

Quimper, C.-G.; López-Ortiz, A.; and Pesant, G. 2006. A quadratic propagator for the inter-distance constraint. In *Proc. of the 21st Nat. Conf. on Artificial Intelligence (AAAI 06)*, 123–128.

Taillard, E. 1993. Benchmarks for basic scheduling problems. *European Journal Opererational Research* 64(2):278–285.

Vilím, P.; Barták, R.; and Čepek, O. 2004. Unary resource constraint with optional activities. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, 62–76.

Vilím, P. 2002. Batch processing with sequence dependent setup times: New results. In *Proceedings of the 4th Workshop of Constraint Programming for Decision and Control (CPDC'02)*.

Vilím, P. 2004. $O(n \log n)$ filtering algorithms for unary resource constraint. In *Proceedings of the 1st International conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CPAIOR 2004)*, 335–347.

Vilím, P. 2007. *Global Constraints in Scheduling*. Ph.D. Dissertation, Charles University in Prague.

Vilím, P. 2009. Edge finding filtering algorithm for discrete cumulative resources in o(kn log n). In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, 802–816.

Wolf, A., and Schrader, G. 2005. $o(n \log n)$ overload checking for the cumulative constraint and its application. In *16th International Conference on Applications of Declarative Programming and Knowledge Management (INAP'05)*, 88–101.