Linear-Time Filtering Algorithms for the Disjunctive Constraint

Hamed Fahimi
Claude-Guy Quimper
Université Laval

Claude-Guy.Quimper@ift.ulaval.ca
hamed.fahimi.1@ulaval.ca

July 2014
Consider a set of $n$ tasks, with known parameters: The release time ($r_i$); The deadline ($d_i$); The processing time ($p_i$); and the unknown starting times $[s_1, ..., s_n]$. 
Consider a set of $n$ tasks, with known parameters: The \textit{release time}(r_i); The \textit{deadline}(d_i); The \textit{processing time}(p_i); and the unknown starting times $[s_1, \ldots, s_n]$. 

![Diagram showing disjunctive constraint with tasks A1, A2, A3, release times, deadlines, and processing times]
Consider a set of $n$ tasks, with known parameters: The release time ($r_i$); The deadline ($d_i$); The processing time ($p_i$); and the unknown starting times $[s_1, \ldots, s_n]$.

**Constraint:**

\[
\text{DISJUNCTIVE}([s_1, \ldots, s_n]) \iff s_i + p_i \leq s_j \text{ or } s_j + p_j \leq s_i
\]
Consider a set of $n$ tasks, with known parameters:
The release time($r_i$); The deadline($d_i$); The processing time($p_i$); and the unknown starting times $[s_1, \ldots, s_n]$.

**Constraint:**

$$\text{DISJUNCTIVE}([s_1, \ldots, s_n]) \iff s_i + p_i \leq s_j \text{ or } s_j + p_j \leq s_i$$

- A feasible schedule!
Disjunctive Constraint

• It is NP-Complete to determine whether there exists a solution to the Disjunctive constraint.

• It is NP-Hard to filter out all values that do not lead to a solution.

• Nonetheless, there exist rules that detect in polynomial time some filtering of the domains of the tasks.

• Our goal is to improve some of these existing filtering algorithms for this constraint.
• We aim at designing algorithms with linear complexity.

• To achieve this goal, we assume that sorting can be done with a linear time algorithm, say *radix sort*.
• If \( lst_i < ect_i \) for a task, then the interval \([lst_i, ect_i)\) is called the *compulsory part* of \( i \).
Time-Tabling

• If \( lst_i < ect_i \) for a task, then the interval \([lst_i, ect_i]\) is called the *compulsory part* of \( i \).

• The Time-Tabling technique filters the domains which are in conflict with the compulsory parts of the tasks.
The Time-Tabling technique filters the domains which are in conflict with the compulsory parts of the tasks.

- If $\text{lst}_i < \text{ect}_i$ for a task, then the interval $[\text{lst}_i, \text{ect}_i)$ is called the *compulsory part* of $i$.

- The Time-Tabling technique filters the domains which are in conflict with the compulsory parts of the tasks.
The Time-Tabling technique filters the domains which are in conflict with the compulsory parts of the tasks.

- If $\text{lst}_i < \text{ect}_i$ for a task, then the interval $[\text{lst}_i, \text{ect}_i)$ is called the compulsory part of $i$.

- The Time-Tabling technique filters the domains which are in conflict with the compulsory parts of the tasks.

![Diagram of Time-Tabling technique with filtering and compulsory parts indicated.](image-url)
Time-Tabling

• If \( \text{lst}_i < \text{ect}_i \) for a task, then the interval \([\text{lst}_i, \text{ect}_i)\) is called the *compulsory part* of \( i \).

• The Time-Tabling technique filters the domains which are in conflict with the compulsory parts of the tasks.
**Time-Tabling**

- Ouellet & Quimper presented an algorithm for Time-Tabling on a generalized case in $O(n \log(n))$.

- We took advantage of Union-Find to achieve an algorithm that admits a linear time implementation for the Disjunctive case.
The strategy of our algorithm
The strategy of our algorithm

Compulsory(A$_1$)

Compulsory(A$_2$)
The strategy of our algorithm

Merged(Compulsory(A₁), Compulsory(A₂))
The domain of $A_3$ after filtering.

- The strategy of our algorithm

- Compulsory($A_1$)
- Compulsory($A_2$)
- Merged(Compulsory($A_1$), Compulsory($A_2$))
This is a data structure that keeps track of when the resource is executing a task.

It is initialized with an empty set of tasks $\Theta = \emptyset$.

It is possible to add a task to $\Theta$ in constant time. The task will be scheduled at the earliest time as possible with preemption.

It is possible to compute the earliest completion time of $\Theta$ in constant time at any time!
### Θ-Tree and Time line comparison

<table>
<thead>
<tr>
<th>Operation</th>
<th>Θ-Tree (Vilím)</th>
<th>Time line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding a task to the schedule</td>
<td>O((\log(n)))</td>
<td>O(1)</td>
</tr>
<tr>
<td>Computing the earliest completion time</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Removing a task from the schedule</td>
<td>O((\log(n))) steps</td>
<td>Not supported !</td>
</tr>
</tbody>
</table>
• Between each two consecutive time points, there is a capacity that denotes the amount of time that the resource is available through. The capacities are initially equal to the difference between the consecutive time points.
We schedule the tasks, one by one. After scheduling, the free times will reduce.

<table>
<thead>
<tr>
<th>est&lt;sub&gt;i&lt;/sub&gt;</th>
<th>lct&lt;sub&gt;i&lt;/sub&gt;,</th>
<th>p&lt;sub&gt;i&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>6</td>
</tr>
</tbody>
</table>

{1} → {4} → {5} → {28}
We schedule the tasks, one by one. After scheduling, the free times will reduce.

<table>
<thead>
<tr>
<th>est_i</th>
<th>lct_i,</th>
<th>p_i</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>6</td>
</tr>
</tbody>
</table>

\{1\} \rightarrow \{4\} \rightarrow \{5\} \rightarrow \{28\}
Once a capacity equals null, the corresponding time points are merged by Union-Find.

\[
\begin{array}{c|c|c}
\text{est}_i & \text{lct}_i & p_i \\
5 & 8 & 2 \\
1 & 10 & 6 \\
4 & 15 & 6 \\
\end{array}
\]

\[
\{1\} \to \{4\} \to \{5\} \to \{28\}
\]
• Once a capacity equals null, the corresponding time points are merged by Union-Find.

<table>
<thead>
<tr>
<th>est_i</th>
<th>lct_i</th>
<th>p_i</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>6</td>
</tr>
</tbody>
</table>

\{1,4,5\}^{19} \rightarrow \{28\}
That allows to run a linear search over the time line for periods that have free time. This search will jump over the occupied regions in constant time.

$$\{1, 4, 5\} \rightarrow \{28\}$$
• That allows to run a linear search over the time line for periods that have free time. This search will jump over the occupied regions in constant time.

\[
\{1,4,5\}^{13} \rightarrow \{28\}
\]

\[
\begin{array}{ccc|c}
\text{est}_i & \text{lct}_i, & p_i \\
5 & 8 & 2 \\
1 & 10 & 6 \\
4 & 15 & 6 \\
\end{array}
\]
That allows to run a linear search over the time line for periods that have free time. This search will jump over the occupied regions in constant time.

The earliest completion time is computed in constant time by $28 - 13 = 15$.

\[
\{1,4,5\} \xrightarrow{13} \{28\}
\]

The earliest completion time is computed in constant time by $28 - 13 = 15$. 

<table>
<thead>
<tr>
<th>est&lt;sub&gt;i&lt;/sub&gt;</th>
<th>lct&lt;sub&gt;i&lt;/sub&gt;,</th>
<th>p&lt;sub&gt;i&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>6</td>
</tr>
</tbody>
</table>
Overload Checking

<table>
<thead>
<tr>
<th>A_1</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Using the idea of a $\Theta$-Tree, Vilím presented the following algorithm for the overload check.

1. $\Theta := \emptyset$;
2. for $j \in T$ in non-decreasing order of $lct_j$ do begin
3.   $\Theta := \Theta \cup \{j\}$;
4.   if $ect_\Theta > lct_j$ then
5.     fail; {No solution exists}
6. end;
Overload Checking

Using the idea of a $\Theta$-Tree, Vilím presented the following algorithm for the overload check.

1. $\Theta := \emptyset$;
2. for $j \in T$ in non-decreasing order of $lct_j$ do begin
3.   $\Theta := \Theta \cup \{j\}$;
4.   if $ect_{\Theta} > lct_j$ then
5.     fail;  {No solution exists}
6. end;

We keep the same algorithm and only replace the $\Theta$-Tree with time line to achieve a linear time algorithm.
Let $A_i$ and $A_j$ be two tasks. If $ect_i > lst_j$, the precedence $A_j \ll A_i$ is called detectable.
• Let $A_i$ and $A_j$ be two tasks. If $ect_i > lst_j$, the precedence $A_j << A_i$ is called detectable.
Example

$\text{p}_A = 11$

$\text{p}_B = 10$

$\text{p}_C = 5$
\[ p_A = 11 \]
\[ p_B = 10 \]
\[ p_C = 5 \]

- A\ll C, B\ll C.
\( p_A = 11 \)

\( p_B = 10 \)

\( p_C = 5 \)

\( \cdot A \ll \ll C, B \ll \ll C. \)

\( \cdot \) Since \( \{A, B\} \ll \ll C \), the domain of \( C \) will be filtered to

\[ \text{est}_C \geq \text{est}_A + p_A + p_B = 21. \]
• $A \ll C$, $B \ll C$.
• Since $\{A, B\} \ll C$, the domain of $C$ will be filtered to $\text{est}_C \geq \text{est}_A + p_A + p_B = 21$.
• The domain of $C$ after filtering.
Detectable Precedences

• Vilím introduced the idea of detectable precedences and presented an algorithm in $O(n \log(n))$. 
• Vilím introduced the idea of detectable precedences and presented an algorithm in $O(n \log(n))$.

• This algorithm temporarily removes a task from the schedule, computes the earliest completion time of the set, and reinserts the task to the schedule.
Detectable Precedences

- Vilím introduced the idea of detectable precedences and presented an algorithm in $O(n \log(n))$.

- This algorithm temporarily removes a task from the schedule, computes the earliest completion time of the set, and reinserts the task to the schedule.

- The time line does not allow the removal of a task.
Detectable Precedences

• Vilím introduced the idea of detectable precedences and presented an algorithm in $O(n \log(n))$.

• This algorithm temporarily removes a task from the schedule, computes the earliest completion time of the set, and reinserts the task to the schedule.

• The time line does not allow the removal of a task.

• We modified the algorithm so that no removal of a task is required.
In order to show the advantage of the state of the art algorithms, we ran the experiments on job-shop and open-shop scheduling problems.

After 10 minutes of computations, the program halts.

The problems are not solved to optimality.

The number of backtracks that occur will be counted.

We compare two algorithms which explore the same tree in the same order.

A larger portion of the search tree will be traversed within 10 minutes with the faster algorithm.
The results of three methods on open-shop and job-shop benchmark problems with $n$ jobs and $m$ tasks per job. The numbers indicate the ratio of the cumulative number of backtracks between all instances of size $nm$ after 10 minutes of computations.

<table>
<thead>
<tr>
<th>$n \times m$</th>
<th>OC</th>
<th>DP</th>
<th>TT</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 × 4</td>
<td>0.96</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>5 × 5</td>
<td>1.03</td>
<td>1.12</td>
<td>1.75</td>
</tr>
<tr>
<td>7 × 7</td>
<td>1.02</td>
<td>1.16</td>
<td>2.09</td>
</tr>
<tr>
<td>10 × 10</td>
<td>1.06</td>
<td>1.33</td>
<td>2.14</td>
</tr>
<tr>
<td>15 × 15</td>
<td>1.03</td>
<td>1.39</td>
<td>2.15</td>
</tr>
<tr>
<td>20 × 20</td>
<td>1.06</td>
<td>1.56</td>
<td>2.17</td>
</tr>
<tr>
<td><strong>p-value</strong></td>
<td>0.25</td>
<td>8.28E-14</td>
<td>5.95E-14</td>
</tr>
</tbody>
</table>

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

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

• We came up with three faster algorithms to filter the disjunctive constraint.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Previous complexity</th>
<th>Now complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time-Tabling</td>
<td>(O(n \log(n)))</td>
<td>(O(n)) () (Fahimi &amp; Quimper)</td>
</tr>
<tr>
<td>(Ouellet &amp; Quimper)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overload check</td>
<td>(O(n \log(n)))</td>
<td>(O(n)) () (Fahimi &amp; Quimper)</td>
</tr>
<tr>
<td>Vilím</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Detectable precedences</td>
<td>(O(n \log(n)))</td>
<td>(O(n)) () (Fahimi &amp; Quimper)</td>
</tr>
<tr>
<td>Vilím</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thank you!