A disjunctive resource would yield a Cumulative constraint with all resource requirements $r_i = 1$ and capacity $C = 1$:

The tasks cannot overlap!
Application: Job-Shop Problem

- Each color denotes a resource, with capacity 1.
- Precedence constraints (denoted ≪) on the tasks of a job.

minimize makespan

time

Each color denotes a resource, with capacity 1. Precedence constraints (denoted ≪) on the tasks of a job.
Application: Job-Shop Problem

- Each color denotes a resource, with capacity 1.
- Precedence constraints (denoted ≪) on the tasks of a job.

![Diagram showing jobs with precedence constraints and a timeline to minimize makespan]
Binary Decomposition for a Unary Resource

- Let T be a set of n tasks (aka activities) that cannot overlap, with start-time variables $s_i$ and processing-time (aka duration) parameters $p_i$.

- $\forall \ i, j \in T$ where $i < j$:
  - $b_{ij} \equiv s_i + p_i \leq s_j$
  - $b_{ji} \equiv s_j + p_j \leq s_i$
  - $b_{ij} \neq b_{ji}$ (either task i ends before task j starts, or vice-versa)

- How does this binary decomposition compare with timetable filtering for Cumulative([s_1,…,s_n],[p_1,…,p_n],[1,…,1],1)?
The binary decomposition with reified constraints is at least as strong as timetable filtering for Cumulative.

Example where the binary decomposition is *strictly* stronger:

Task A has no mandatory part: no pruning for task B with timetable filtering!
Notation and Definitions

- Let $\Omega \subseteq T$ be a subset of a set $T$ of $n$ non-overlapping tasks with start-time variables $s_i$:
  - $est_\Omega = \min \{est_j | j \in \Omega\} = \text{earliest start time of } \Omega$, with $est_j = \min D(s_j)$
  - $lct_\Omega = \max \{lct_j | j \in \Omega\} = \text{latest completion time of } \Omega$, with $lct_j = p_j + \max D(s_j)$ for parameters $p_j$
  - $p_\Omega = \sum_{j \in \Omega} p_j = \text{total processing time of } \Omega$, for parameters $p_j$

- Computing the earliest completion time of $\Omega$ or latest start time of $\Omega$ is NP-hard (as so for $\Omega=T$). We use a lower bound and upper bound instead:
  - $ect_\Omega \geq \max \{est_{\Omega'} + p_{\Omega'} | \Omega' \subseteq \Omega\}$
  - $lst_\Omega \leq \min \{lct_{\Omega'} - p_{\Omega'} | \Omega' \subseteq \Omega\}$

- By convention:
  - $est_{\emptyset} = ect_{\emptyset} = -\infty$
  - $lct_{\emptyset} = lst_{\emptyset} = +\infty$
  - $p_{\emptyset} = 0$
This failure is not captured by the binary decomposition of Disjunctive.

- \( \forall \: \Omega \subseteq T : \text{fail if } \text{est}_\Omega + p_\Omega > \text{lct}_\Omega \)

- If there exists a subset of tasks that cannot be processed within its bounds, then no solution exists.

Example:

- Take \( \Omega = \{A, B, C\}\):
  
  \[ \text{est}_\Omega = 0, \: p_\Omega = 5+5+6 = 16, \: \text{lct}_\Omega = 15, \: \text{fail because } 0+16 > 15. \]
Overload Checking: Time Complexity?

- \( \forall \Omega \subseteq T : \text{fail if } \text{est}_\Omega + p_\Omega > \text{lct}_\Omega \)

- We need to enumerate all subsets \( \Omega \) of \( T \), hence \( 2^n \) feasibility checks.

- It is impractical to embed an exponential-time algorithm into a propagator.

- We need something else…
Left cut: $\text{LCut}(T,j) = \{ i \mid i \in T \& \text{lct}_i \leq \text{lct}_j \}$ = the subset of tasks ending by the end of $j$.

In the example below: $\text{LCut}(T,C) = \{C\}$; $\text{LCut}(T,A) = \{C,A\}$; $\text{LCut}(T,B) = \{C,A,B\}$.

$\forall \Omega \subseteq T : \text{fail if } \text{est}_\Omega + p_\Omega > \text{lct}_\Omega$ can be equivalently formulated (proof omitted) as $\forall j \in T : \text{fail if } \text{ect}_{\text{LCut}(T,j)} > \text{lct}_j$.

Example, when $j=B$: $\text{LCut}(T,B) = \{A,B,C\}$, hence fail as $\text{ect}_{\text{LCut}(T,B)} \geq 16 > 15 = \text{lct}_B$.
Computing $\text{ect}_\Omega$ Efficiently is the Key!

- Recall the lower bound: $\text{ect}_\Omega \geq \max \{\text{est}_\Omega' + p_\Omega' \mid \Omega' \subseteq \Omega\}$.
- How to compute the bound efficiently, with $\Omega$ being $\text{LCut}(T,j)$ for each $j$ in $T$?
- We can use a data structure called a $\Theta$-tree.
- A $\Theta$-tree for a set $\Omega$ of tasks is
  - a balanced binary tree,
  - whose leaf nodes correspond to the tasks of $\Omega$,
  - whose internal nodes have intermediate values, and
  - whose root node has the lower bound on $\text{ect}_\Omega$. 
Let \( \text{Leaves}(v) \) = the set of leaf nodes at or below node \( v \).

Total processing time at or below node \( v \): \( \Sigma P_v = \sum_{j \in \text{Leaves}(v)} p_j \)

Let \( \text{ect}_v = \max \{\text{est}_{\Omega'} + p_{\Omega'} | \Omega' \subseteq \text{Leaves}(v)\} \leq \text{ect}_{\text{Leaves}(v)} \).
When is this correct?

When the tasks in the leaves are sorted by increasing est_i.

\[ \sum P_v = \sum P_{\text{left}(v)} + \sum P_{\text{right}(v)} \]

\[ \text{ect}_v = \max(\text{ect}_{\text{right}(v)}, \text{ect}_{\text{left}(v)} + \sum P_{\text{right}(v)}) \]

Update rule for each non-leaf v:

\[ \Sigma P_{\text{root}} = 25 \]  
\[ \text{ect}_{\text{root}} = 45 \]

\[ \Sigma P_{ab} = 11 \]  
\[ \text{ect}_{ab} = 31 \]

\[ \Sigma P_{cd} = 14 \]  
\[ \text{ect}_{cd} = 44 \]

\[ \Sigma P_a = 5 \]  
\[ p_a = 5 \]  
\[ \Sigma P_a = 5 \]  
\[ \text{ect}_a = 5 \]

\[ \text{est}_a = 0 \]

\[ \Sigma P_b = 6 \]  
\[ p_b = 6 \]  
\[ \Sigma P_b = 6 \]  
\[ \text{ect}_b = 31 \]

\[ \text{est}_b = 25 \]

\[ \Sigma P_c = 4 \]  
\[ p_c = 4 \]  
\[ \Sigma P_c = 4 \]  
\[ \text{ect}_c = 34 \]

\[ \text{est}_c = 30 \]

\[ \Sigma P_d = 10 \]  
\[ p_d = 10 \]  
\[ \Sigma P_d = 10 \]  
\[ \text{ect}_d = 42 \]

\[ \text{est}_d = 32 \]

Time complexity to compute \( \text{ect}_{\text{root}} \):  \( O(n \log n) \)
To remove task \(i\) from a \(\Theta\)-tree: set \(\Sigma P_i = 0\) and \(ect_i = -\infty\).

Update rule for each non-leaf \(v\):

\[
\begin{align*}
\Sigma P_v &= \Sigma P_{\text{left}(v)} + \Sigma P_{\text{right}(v)} \\
ect_v &= \max(ect_{\text{right}(v)}, ect_{\text{left}(v)} + \Sigma P_{\text{right}(v)})
\end{align*}
\]

Thus we can initialize with all \(n\) tasks in removed status and then insert them one by one: \(O(n \log n)\) time total, since each insertion takes \(O(\log n)\) time.

\(\Theta\)-Tree: Incremental Update

Removed task: it takes \(O(\log n)\) time to insert it.
### Θ-Tree: Time Complexities

For $n$ tasks:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>init(${1..n}$)</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>insert($i$)</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>remove($i$)</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>extinct</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Overload Checker in $O(n^2 \log n)$ Time

Overload checking rule:
\[ \forall j \in T : \text{fail if } \text{ect}_{\text{LCut}(T,j)} \geq \text{ect}_{\text{root}} > lct_j \]

Can we iterate on $\{1..n\}$ in a specific order such that $\text{LCut}(T,j) = \text{LCut}(T,j-1) \cup \{ j \}$?
If yes, then incremental $\Theta$-tree:
one $O(\log n)$-time insertion at a time.
Overload Checker in $O(n \log n)$ Time

Observation:
– Let $T = \{1..n\}$ be ordered such that $lct_1 \leq \ldots \leq lct_n$. Observe the example at page 9.
– Then $\text{LCut}(T,1) \subseteq \text{LCut}(T,2) \subseteq \ldots \subseteq \text{LCut}(T,n) = T$: all tasks are eventually inserted.

```java
OverloadCheckEfficient(T={1..n}) {
    T ← sortAZ([1..n],sortKey = lct) // O(n log n) time
    Θ ← Θ-Tree.init({1..n}) // est$_1 \leq \ldots \leq$ est$_n$; O(n log n)
    for (j ← T) {
        Θ.insert(j) // O(log n) time
        // invariant: Θ contains LCut(T,j)
        if (Θ.ect > lct$_j$) { // O(1) time
            throw InconsistencyException
        }
    }
}
```
Both A and B must end before C starts, denoted \( \{A,B\} \ll C \) (for \( A \ll C \) & \( B \ll C \)):

- We say that \( j \ll i \) is a detectable precedence if \( \text{est}_i + p_i > \text{lct}_j - p_j \) that is if \( \text{lst}_j < \text{ect}_i \): task j must end before task i starts.

- For a task i, the set of all tasks j with detectable precedence \( j \ll i \) is \( \text{DPrec}(T,i) = \{ j \mid j \in T \setminus \{i\} \land \text{est}_i + p_i > \text{lct}_j - p_j \} \), as \( \text{lst}_i < \text{ect}_i \) can happen.

- Filtering: \( \text{est}_i \leftarrow \max(\text{est}_i , \text{ect}_{\text{DPrec}(T,i)}) \), for all \( i \in T \).
Detectable Precedences: Iterating on Tasks

- $\text{DPrec}'(T,i) = \{ j : j \in T \& \text{ est}_i + p_i > \text{ lct}_j - p_j \}$.
  Note that task $i$ is not always actually in $\text{DPrec}'(T,i)$:
  For example at page 17: $\text{DPrec}'(T,A) = \emptyset$; $\text{DPrec}'(T,B) = \emptyset$; $\text{DPrec}'(T,C) = \{A,B\} \neq T = \{A,B,C\}$.

- Hence: $\text{DPrec}(T,i) = \text{DPrec}'(T,i) \setminus \{i\}$.

- Let $T = \{1..n\}$ be ordered (by $\text{ect}_i$) such that
  - $\text{est}_1 + p_1 \leq \text{est}_2 + p_2 \leq ... \leq \text{est}_n + p_n$
  - Then: $\text{DPrec}'(T,1) \subseteq \text{DPrec}'(T,2) \subseteq ... \subseteq \text{DPrec}'(T,n)$

- This is exactly what we are looking for: an order to consider the tasks $i$ of $T$ such that the set of detectable precedences is growing monotonically, as this is very important for computing all $\text{ect}_{\text{DPrec}}(T,i)$ efficiently and incrementally with a $\Theta$-tree.

- Note that $\text{DPrec}'(T,n)$ is not necessarily equal to $T$ (see the counterexample above): not necessarily all tasks are eventually inserted into the initialized $\Theta$-tree.
Detectable Precedences: Filtering Algorithm

\[ \text{est}_i + p_i \]

\[ \text{lct}_j - p_j \]

\[ \text{DPrec}' \]
Detectable Precedences: Filtering Algorithm

est\(_i + p_i\)

lct\(_j - p_j\)

A

B

C

D

DPrec'
Detectable Precedences: Filtering Algorithm

\[ \text{est}_i + p_i \]

\[ \text{lct}_j - p_j \]

\[ \text{DPrec}' \]
Detectable Precedences: Filtering Algorithm

\[ \text{est}_i + p_i \]

\[ \text{lct}_j - p_j \]

\[ \text{DPrec}' \]
Detectable Precedences: Filtering Algorithm

\[ \text{est}_i + p_i \]

\[ lct_j - p_j \]

\[ \text{DPrec}' \]
Detectable Precedences: $O(n \log n)$ time

DetectablePrecedence(T={1..n}) {
  T_{lst} ← sortAZ([1..n], sortKey = lct-p) // $O(n \log n)$
  T_{ect} ← sortAZ([1..n], sortKey = est+p) // $O(n \log n)$
  ite ← iterator(T_{lst})
  j ← ite.next() // candidate precedence of i
  Θ ← Θ-Tree.init({1..n}) // $O(n \log n)$ time
  for (i ← T_{ect}) {
    while (est_i+p_i > lct_j-p_j) {
      Θ.insert(j) // $O(\log n)$ time
      if (ite.hasNext()) {j ← ite.next()} else {break}
    }
    est'_i ← max(est_i, ect_{Θ\i}) // $O(\log n)$ time
  }
  est_i ← est'_i, ∀i∈T
}

This is executed at most n times

Because Θ contains DPrec'(T,i) and not DPrec(T,i): Θ.remove(i), use Θ.ect for max, Θ.insert(i).
Not-Last (NL): Another Filtering Rule

- \( \forall \Omega \subset T \) (non-empty strict subset of \( T \)) : \( \forall i \in T \setminus \Omega : \)

  if \( \text{est}_\Omega + p_\Omega > \text{lct}_i - p_i \) then \( \text{lct}_i \leftarrow \min(\text{lct}_i, \max \{\text{lct}_j - p_j \mid j \in \Omega\}) \)  

  (NL)

- Example: For \( \Omega = \{A, B\} \), task \( i = C \) cannot start last:

- Again, we need to find a way to enumerate the \( \Omega \) in a nested way.

It is impossible to have \( \{A, B\} \prec C \), so \( C \) must end before \( A \) or \( B \) (or both):

\( \text{lct}_C \leftarrow \min(\text{lct}_C, \max \{\text{lct}_B - p_B, \text{lct}_A - p_A\}) \).
Not-Last Rule

- If \( \text{est}_\Omega + p_\Omega > \text{lct}_i - p_i \) then \( \text{lct}_i \leftarrow \min(\text{lct}_i, \max \{\text{lct}_j - p_j \mid j \in \Omega\}) \) \hspace{1cm} \text{(NL)}

- If there exists a subset \( \Omega \) for which this rule actually filters (tightens \( \text{lct}_i \)), then it is a subset of \( \text{NLSet}(T,i) = \{ j \mid j \in T \setminus \{i\} \land \text{lct}_j - p_j < \text{lct}_i \} \) = tasks that start before \( i \) ends.

- Does there exist a subset \( \Omega \subseteq \text{NLSet}(T,i) \) for which the detection part of the rule (namely \( \text{est}_\Omega + p_\Omega > \text{lct}_i - p_i \)) also holds?

- Such a subset exists if and only if \( \text{ect}_{\text{NLSet}(T,i)} \geq \max \{\text{est}_{\Omega'}, p_{\Omega'} \mid \Omega' \subseteq \text{NLSet}(T,i)\} > \text{lct}_i - p_i \).

The left-hand side is a lower bound on \( \text{ect}_{\text{NLSet}(T,i)} \): this probably means that a \( \Theta \)-tree will be useful...
Let us make this more efficient!

† The existence of a subset $\Omega \subseteq \text{NLSet}(T,i)$ can be tested as

$$\text{ect}_{\text{NLSet}(T,i)} \geq \max \{ \text{est}_{\Omega'} + p_{\Omega'} \mid \Omega' \subseteq \text{NLSet}(T,i) \} > \text{lct}_i - p_i$$

† The problem is that we then do not have a subset $\Omega$ for filtering. But do we really need one? No, we accept to relax the filtering, namely only when

$$\max \{ \text{lct}_j - p_j \mid j \in \Omega \} \leq \max \{ \text{lct}_j - p_j \mid j \in \text{NLSet}(T,i) \} < \text{lct}_i$$

Because $\Omega \subseteq \text{NLSet}(T,i)$:
The advantage of this relaxation is that we do not need a $\Omega$!
Weaker Not-Last Rule

\[ \text{If } \text{est}_\Omega + p_\Omega > \text{lct}_i - p_i \text{ then } \text{lct}_i \leftarrow \min(\text{lct}_i, \max \{\text{lct}_j - p_j \mid j \in \Omega\}) \quad (\text{NL}) \]

\[ \text{If } \text{ect}_{\text{NLSet}(T,i)} > \text{lct}_i - p_i \text{ then } \text{lct}_i \leftarrow \max \{\text{lct}_j - p_j \mid j \in \text{NLSet}(T,i)\} \quad (\text{NL'}) \]

• Rule NL’ can filter less than rule NL, but the fixpoint is the same.
Recall: \( NLSet(T, i) = \{ j \mid j \in T \setminus \{i\} \ \& \ \lct_j - p_j < \lct_i \} \) = set of other tasks that start before \( i \) ends.

We are looking for an order on the tasks \( i \) so as to have nested sets.

Let \( NLSet'(T, i) = \{ j \mid j \in T \ \& \ \lct_j - p_j < \lct_i \} \).

Note that \( i \) is always in \( NLSet'(T, i) \).
Hence: \( NLSet(T, i) = NLSet'(T, i) \setminus \{i\} \).

Let \( T = \{1..n\} \) be ordered such that \( \lct_1 \leq \lct_2 \leq \ldots \leq \lct_n \):
then \( NLSet'(T, 1) \subseteq NLSet'(T, 2) \subseteq \ldots \subseteq NLSet'(T, n) = T \):
all tasks are eventually inserted into the initialized \( \Theta \)-tree.

For example at page 21: \( NLSet'(T, C) = \{A, B, C\} = NLSet'(T, A) = NLSet'(T, B) = T \).

Now we have a way to compute the \( NLSet(T, i) \) incrementally when using a \( \Theta \)-tree.
Not-Last: Filtering Algorithm

lct_i

lct_j-p_j

NLS_set'
Not-Last: Filtering Algorithm

lct_i

lct_j - p_j

NLS\text{Set}'
Not-Last: Filtering Algorithm

lct_i

lct_j-p_j

NLSset'
Not-Last: Filtering Algorithm
Not-Last: Filtering Algorithm

\[ \text{lct}_i \]

\[ \text{lct}_{j-p_j} \]

\[ \text{NLS}t' \]
Not-Last: Filtering Algorithm

\begin{verbatim}
NotLast(T={1..n}) { 
  lct’_i ← lct_i, ∀i∈T
  T_{lst} ← sortAZ([1..n], sortKey = lct-p)  // O(n log n) time
  T_{lct} ← sortAZ([1..n], sortKey = lct)  // O(n log n) time
  ite ← iterator(T_{lst})
  k ← ite.next()
  j ← -1
  Θ ← Θ-Tree.init({1..n})  // O(n log n) time
  for (i ← T_{lct}) {
    while (lct_i > lct_k-p_k) {
      Θ.insert(k)  // O(log n) time
      j ← k  // lct_j-p_j = max {lct_Ω - p_Ω : Ω ⊆ NLSet(T,i)}
      k ← ite.next()
    }
    if (ect_{Θ\setminus i} > lct_i-p_i) {  // O(log n) time
      lct’_i ← min(lct_i, lct_j-p_j)
    }
  }
  lct_i ← lct’_i, ∀i∈T
}
\end{verbatim}

Θ-tree contains all NLSet'(T,i).
Edge Finding (EF): Another Filtering Rule

- \( \forall \Omega \subset T \) (non-empty strict subset of \( T \)) : \( \forall i \in T \setminus \Omega \) :
  
  \[
  \text{if } \text{est}_{\Omega_i} + p_{\Omega_i} > \text{lct}_{\Omega} \text{ then } \Omega \ll i \text{ and thus } \text{est}_i \leftarrow \max \{\text{est}_i, \text{ect}_{\Omega}\} \quad \text{(EF)}
  \]

- That is: Task \( i \) must be scheduled after the set \( \Omega \).

![Diagram showing the scheduling of tasks A, B, C, D with edge finding rule applied.](image-url)
Edge Finding

- Reformulation of EF for easier implementation

\[ \forall j \in T, \forall i \in T \setminus LCut(T, j): \]

\[ \text{ect}_{LCut(T, j)} > \text{lct}_j \Rightarrow LCut(T, j) \ll i \]

\[ \Rightarrow \text{est}_i \leftarrow \max \{\text{est}_i, \text{ect}_{LCut(T, j)}\} \quad (EF') \]

- Implementation using Θ-tree considering j and i wrt LCut(T,j)
  - \( \Theta = LCut(T, j) \)
  - \( \Theta\)-Tree.insert(i), check if \( \text{ect}_\Theta > \text{lct}_j \)
  - \( \Theta\).remove(i)

\[ LCut(T, j) = \{i \mid i \in T \& \text{lct}_i \leq \text{lct}_j\} \]

- \( O(\log n) \) for testing one \((i, j)\)
- \( O(n^2 \log n) \) overall => too slow!
**Θ-Λ-Tree = Generalization of Θ-Tree**

- **ect(Θ-Λ)** = max(\{ect_{Θ},ect_{Θ∪i } : i ∈ Λ\})
  - earliest completion time if at most one gray task used

- New values stored in the nodes (in addition to \(ΣP_v \) & ect_v)
  - \(ΣP_v = max \{p_{Θ'} | Θ'⊆Leaves(v) & |Θ'∩Λ| ≤ 1\}\)
  - ect_v = ect_{Leaves(v)} = max \{est_{Θ'}+p_{Θ'} | Θ'⊆Leaves(v) & |Θ'∩Λ| ≤ 1\}\)

- Update rule
  - \(ΣP_v = max \{ΣP_{left(v)}+ΣP_{right(v)},ΣP_{left(v)}+ΣP_{right(v)}\}\)
  - ect_v = max \{ect_{right(v)},ect_{left(v)}+ΣP_{right(v)},ect_{left(v)}+ΣP_{right(v)}\}\)
Example

- Θ-Λ-Tree: Θ={a,b,d} Λ={c}
Responsible Tasks

- For each node $v$ we can also compute the gray task responsible for $\Sigma P_v$ or $\text{ect}_v$

- Leaf nodes:
  - $\text{resp}_{\Sigma P}(i) = i$ if $i$ is gray, undef otherwise
  - $\text{resp}_{\text{ect}}(i) = i$ if $i$ is gray, undef otherwise

- Internal nodes:
  - $\text{resp}_{\Sigma P}(v) = \text{resp}_{\Sigma P}(\text{left}(v))$ if $\Sigma P_v = \Sigma P_{\text{left}(v)} + \Sigma P_{\text{right}(v)}$
    
    $\text{resp}_{\Sigma P}(\text{right}(v))$ otherwise
  - $\text{resp}_{\text{ect}}(v) = \text{resp}_{\text{ect}}(\text{right}(v))$ if $\text{ect}_v = \text{ect}_{\text{right}(v)}$
    
    $\text{resp}_{\text{ect}}(\text{left}(v))$ if $\text{ect}_v = \text{ect}_{\text{left}(v)} + \Sigma P_{\text{right}(v)}$
    
    $\text{resp}_{\Sigma P}(\text{right}(v))$ if $\text{ect}_v = \text{ect}_{\text{left}(v)} + \Sigma P_{\text{right}(v)}$
<table>
<thead>
<tr>
<th>Operation</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\emptyset, \emptyset)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$(T, \emptyset)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>$(\emptyset \setminus {i}, \emptyset \cup {i})$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>$\Theta := \Theta \cup {i}$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>$\Lambda := \Lambda \cup {i}$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>$\Lambda := \Lambda \setminus {i}$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>$\text{ect}(\Theta, \Lambda)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$\text{ect}_\Theta$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
while (\text{est}(\Theta-\Lambda) > lct_j) 
    \begin{align*}
    i &\leftarrow \text{respect}(\Theta-\Lambda) \\
    \text{est}_i &\leftarrow \max\{\text{est}_i, \text{est}_0\} \\
    \Lambda &\leftarrow \Lambda \setminus i \quad \text{O}(\log n)
    \end{align*}

Retrieve the task of $\Lambda$ responsible
Edge-Finding Algorithm

```plaintext
EdgeFinding(T={1..n}) {
(Θ,Λ) = (T,∅) // O(n log n) time
T_{lct} ← sortZA([1..n],sortKey = lct) // O(n log n) time
ite ← iterator(T_{lct})
j = ite.next()
while (ite.hasNext()) {
    if (ect_{Θ} > lct_{j}) throw InconsistencyException // overload
    (Θ,Λ) = (Θ\j,Λ∪j) // O(log n) time
    j ← ite.next()
    while (ect(Θ-Λ) > lct_{j}) { // O(1) time
        i ← resp_{ect}(Θ-Λ)
        est_{i} ← max{est_{i},ect_{Θ}}
        Λ ← Λ\i // O(log n) time
    }
}
}
```

Executed at most n times
None of the algorithms above is idempotent.

According to Petr Vilím (see next slide), the following order for fixpoint computation is very efficient:
Most of the notation, examples, … come from Petr Vilím’s PhD thesis (https://vilim.eu/petr/disertace.pdf), where all the proofs omitted here can be found.

This thesis had a big impact on CP solvers because most of the algorithms for a disjunctive resource introduced by Petr Vilím take $O(n \log n)$ time instead of $O(n^2)$ or $O(n^3)$ time.

Petr Vilím is now working at IBM on CP Scheduling Solver.