Disjunctive Scheduling
(version of 23 November 2021)
Disjunctive Resource, aka Unary Resource

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

The activities cannot overlap!
Job-Shop Problem

- Color = resource, with capacity 1.
- Precedence constraints (denoted ≪) on the activities of a job.

minimize makespan

\[ \text{job 1:} \quad \text{job 2:} \quad \text{job 3:} \quad \text{job 4:} \quad \text{job 5:} \quad \text{job 6:} \]

\[ \ll \ll \ll \ll \ll \ll \]

\[ \text{time} \]

\[ \text{disjunctive} \]
Job-Shop Problem

- Color = resource, with capacity 1.
- Precedence constraints (denoted $\ll$) on the activities of a job.

\[
\begin{align*}
\text{job 1:} & \\
\text{job 2:} & \\
\text{job 3:} & \\
\text{job 4:} & \\
\text{job 5:} & \\
\text{job 6:} & \\
\end{align*}
\]

minimize makespan
• Let $T$ be a set of $n$ activities that cannot overlap.

• $\forall i, j \in T$ where $i < j$:
  
  • $b_{ij} \equiv s_i + d_i \leq s_j$
  
  • $b_{ji} \equiv s_j + d_j \leq s_i$

• $b_{ij} \neq b_{ji}$ (either $i$ ends before $j$ starts, or vice-versa)

• How does this binary decomposition compare with timetable filtering for Cumulative($[s_1,\ldots,s_n],[d_1,\ldots,d_n],[1,\ldots,1],1$)?
• The binary decomposition with reified constraints is at least as strong as timetable filtering for Cumulative.

• Example where the binary decomposition is strictly stronger:

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

- Let $\Omega \subseteq T$ be a subset of a set $T$ of non-overlapping activities:
  - $est_\Omega = \min \{est_j \mid j \in \Omega\} =$ earliest starting time of $\Omega$
  - $lct_\Omega = \max \{lct_j \mid j \in \Omega\} =$ latest completion time of $\Omega$
  - $p_\Omega = \sum_{j \in \Omega} p_j =$ total processing time (aka duration) of $\Omega$

- Computing earliest (latest) completion (start) time of $\Omega$ is NP-hard. We use a lower (upper) bound instead:
  - $ect_\Omega = \max \{est_{\Omega'} + p_{\Omega'} \mid \Omega' \subseteq \Omega\}$
  - $lst_\Omega = \min \{lct_{\Omega'} - p_{\Omega'} \mid \Omega' \subseteq \Omega\}$

- By convention:
  - $est_\emptyset = ect_\emptyset = -\infty$
  - $lst_\emptyset = lct_\emptyset = +\infty$
  - $p_\emptyset = 0$
Overload Checking = a feasibility check

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

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

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

- $\forall \Omega \subseteq T : \text{est}_\Omega + p_\Omega > \text{lct}_\Omega \Rightarrow \text{fail}$

- We need to enumerate all subsets $\Omega$ of $T$, hence $2^{|T|}$ checks.

- It is not very practical to embed an algorithm of exponential time complexity in a propagator.

- We need something else…
Overload Checking: Try and improve efficiency

- **Left cut**: \( \text{LCut}(T,j) = \{ i \mid i \in T \& \text{lct}_i \leq \text{lct}_j \} \).  
  Example: \( \text{LCut}(T,A) = \{A,C\} \).

- \( \forall \Omega \subseteq T : \text{est}_\Omega + p_\Omega > \text{lct}_\Omega \Rightarrow \text{fail} \) can be reformulated as

\[
\forall j \in T : \text{ect}_{\text{LCut}(T,j)} > \text{lct}_{\text{LCut}(T,j)} \Rightarrow \text{fail}
\]

that is:

\[
\forall j \in T : \text{ect}_{\text{LCut}(T,j)} > \text{lct}_j \Rightarrow \text{fail}
\]

This can be proven to be equivalent. What do we gain?

- For example, take \( j = B \),
  with \( \text{LCut}(T,B) = \{A,B,C\} \) = subset of activities ending by the end of \( B \):

\[
\text{ect}_{\text{LCut}(T,B)} = 16 > \text{lct}_{\text{LCut}(T,B)} = 15 = \text{lct}_B \text{ (the red equality is true by definition)}.
\]
Computing $\text{ect}_\Omega$ efficiently is the key!

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

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

Let \( \text{ect}_v = \text{ect}_{\text{Leaves}(v)} = \max \{ \text{est}_{\Omega'} + p_{\Omega'} | \Omega' \subseteq \text{Leaves}(v) \} \)

\[ \begin{aligned}
\Sigma P_{\text{root}} &= 25 \\
\text{ect}_{\text{root}} &= 45 \\
\Sigma P_{ab} &= 11 \\
\text{ect}_{ab} &= 31 \\
\Sigma P_{cd} &= 14 \\
\text{ect}_{cd} &= 44 \\
\text{est}_a &= 0 \\
p_a &= 5 \\
\Sigma P_a &= 5 \\
\text{ect}_a &= 5 \\
\text{est}_b &= 25 \\
p_b &= 6 \\
\Sigma P_b &= 6 \\
\text{ect}_b &= 31 \\
\text{est}_c &= 30 \\
p_c &= 4 \\
\Sigma P_c &= 4 \\
\text{ect}_c &= 34 \\
\text{est}_d &= 32 \\
p_d &= 10 \\
\Sigma P_d &= 10 \\
\text{ect}_d &= 42 \\
\end{aligned} \]

How to compute these values recursively from values stored in the left and right children only?
When is this correct?

When the activities in the leaves are sorted by increasing est.

Time complexity to compute ect_root?

Update rule for each non-leaf v:

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

\[ \text{ect}_v = \max(\text{ect}_{\text{right}(v)}, \text{ect}_{\text{left}(v)} + \Sigma P_{\text{right}(v)}) \]
To remove activity $i$ from a $\Theta$-tree: set $\Sigma P_i = 0$ and $\text{ect}_i = -\infty$.

**Update rule for each non-leaf $v$:**

$$\Sigma P_v = \Sigma P_{\text{left}(v)} + \Sigma P_{\text{right}(v)}$$

$$\text{ect}_v = \max(\text{ect}_{\text{right}(v)}, \text{ect}_{\text{left}(v)} + \Sigma P_{\text{right}(v)})$$

Thus we can start with all $n$ activities in removed status and insert them one by one: $O(n \log n)$ time, since each insertion takes $O(\log n)$ time.

Removed activity: it takes $O(\log n)$ time to insert it.

---

**Example**

- $\Sigma P_{\text{root}} = 21, 25$
- $\text{ect}_{\text{root}} = 42, 45$
- $\Sigma P_{ab} = 11$
- $\text{ect}_{ab} = 31$
- $\Sigma P_{cd} = 10, 14$
- $\text{ect}_{cd} = 42, 44$
- $\text{est}_a = 0$
  - $p_a = 5$
  - $\Sigma P_a = 5$
  - $\text{ect}_a = 5$
- $\text{est}_b = 25$
  - $p_b = 6$
  - $\Sigma P_b = 6$
  - $\text{ect}_b = 31$
- $\text{est}_c = 30$
  - $p_c = 4$
  - $\Sigma P_c = 0, 4$
  - $\text{ect}_c = -\infty, 34$
- $\text{est}_d = 32$
  - $p_d = 10$
  - $\Sigma P_d = 10$
  - $\text{ect}_d = 42$
### Θ-Tree: Time Complexities

For $n$ activities:

<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>next</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Overload checking rule:
\[ \forall j \in T : \text{ect}_{\text{LCut}(T,j)} > \text{lct}_j \Rightarrow \text{fail} \]

Overload Checker taking \(O(n^2 \log n)\) time

\[
\text{OverloadCheckInefficient}(T=\{1..n\}) \{ \\
\quad \text{for } (j \leftarrow \{1..n\}) \{ \\
\quad \quad \Theta \leftarrow \Theta\text{-Tree}(\text{LCut}(T,j)) \quad \text{// } O(n \log n) \text{ time} \\
\quad \quad \text{if } (\Theta.\text{ect} > \text{lct}_j) \{ \quad \text{// } O(1) \text{ time} \\
\quad \quad \quad \text{throw InconsistencyException} \\
\quad \quad \} \\
\quad \} \\
\}
\]

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 → incremental \(\Theta\)-tree
→ one \(O(\log n)\)-time insertion at a time
→ each update in \(O(\log n)\) time
Observation:

- Let $T = \{1..n\}$ be ordered such that $lct_1 \leq \ldots \leq lct_n$.
- Then $LCut(T,1) \subseteq LCut(T,2) \subseteq \ldots \subseteq LCut(T,n) = T$: all activities 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}) // est₁ ≤ \ldots ≤ estₙ; 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
        }
    }
}
```
Detectable Precedences = a filtering rule

• Both A and B must end before C starts, denoted \{A,B\} \ll C:

\[
\begin{array}{c}
\text{A} \\
\text{B} \\
\text{C}
\end{array}
\]

• We say that a precedence \( j \ll i \) is detectable if \( \text{est}_i + p_i > \text{lct}_j - p_j \)
that is if \( \text{ect}_i > \text{lst}_j \): activity \( j \) cannot start after activity \( i \) ends.

• Set of all activities with detectable precedence before \( i \): \( \text{DPrec}(T,i) = \{ j \mid j \in T \setminus \{i\} \quad \& \quad \text{est}_i + p_i > \text{lct}_j - p_j \} \).

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

- \( \text{DPrec}'(T,i) = \{ j : j \in T \ & \ \text{est}_i + p_i > \text{lct}_j - p_j \} \). Note that activity \( i \) is \textit{not} always actually in \( \text{DPrec}'(T,i) \).

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

- Let \( T = \{1..n\} \) be ordered such that
  - \( \text{est}_1 + p_1 \leq \text{est}_2 + p_2 \leq \ldots \leq \text{est}_n + p_n \)

- Then: \( \text{DPrec}'(T,1) \subseteq \text{DPrec}'(T,2) \subseteq \ldots \subseteq \text{DPrec}'(T,n) \)

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

- Note that \( \text{DPrec}'(T,n) \) is \textit{not} necessarily \( T \): \textit{not} necessarily all activities are eventually inserted into the initialised \( \Theta \)-tree.
Detectable Precedences: Filtering Algorithm

\[ \text{est}_{i+p_i} \]

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

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

$est_{i+p_{i}}$

$DPrec'$

$lct_{j-p_{j}}$
Detectable Precedences: Filtering Algorithm

$\text{est}_{i+p_i}$

$\text{dct}_{j-p_j}$

DPrec'
Detectable Precedences: Filtering Algorithm

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

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

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

```plaintext
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 = another filtering rule

- \( \forall \Omega \subset T \) non-empty strict subset of \( T \), \( \forall i \in T \setminus \Omega \):
  
  \[ \text{est}_\Omega + p_\Omega > \text{lct}_i - p_i \Rightarrow \text{lct}_i \leftarrow \min(\text{lct}_i, \max \{\text{lct}_j - p_j \mid j \in \Omega\}) \]  
  \( \text{(NL)} \)

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

  ![Diagram showing activities A, B, and C with time slots 0 to 15, illustrating the Not-Last filtering rule.]

  It is impossible to have \( \{A, B\} \ll C \), so \( C \) must end before \( A \) or \( B \) (or both): 
  
  \[ \text{lct}_C \leftarrow \min(\text{lct}_C, \max\{\text{lct}_B - p_B, \text{lct}_A - p_A\}) \].

- Again, we need to find a way to enumerate the \( \Omega \) in a nested way.
Not-Last Rule

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

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

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

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

- The existence of a subset $\Omega$ can be tested as
  
  $\exists \text{ect}_{\text{NLSet}(T,i)} > \text{lct}_i - p_i$

- The problem is that we then do not have a subset $\Omega$ for filtering. But do we really need it?
  
  No, we accept to relax the filtering:

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

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

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

- \( \text{ect}_{\text{NLSet}(T,i)} > \text{lct}_i - p_i \Rightarrow \text{lct}_i \leftarrow \max \{\text{lct}_j - p_j \mid j \in \text{NLSet}(T,i)\} \) (NL’)

- Rule NL’ can filter less than rule NL, but the fixpoint is the same.
• Recall: \( \text{NLSet}(T,i) = \{ j \mid j \in T \setminus \{i\} \land \text{lct}_j - \text{p}_j < \text{lct}_i \} \).

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

• Let \( \text{NLSet}'(T,i) = \{ j \mid j \in T \land \text{lct}_j - \text{p}_j < \text{lct}_i \} \).

  Note that \( i \) is always in \( \text{NLSet}'(T,i) \).

• Let \( T = \{1..n\} \) be ordered such that \( \text{lct}_1 \leq \text{lct}_2 \leq \ldots \leq \text{lct}_n \):

  then \( \text{NLSet}'(T,1) \subseteq \text{NLSet}'(T,2) \subseteq \ldots \subseteq \text{NLSet}'(T,n) = T \):

  all activities are eventually inserted into the initialised \( \Theta \)-tree.

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

\[ lct_i \]

\[ lct_{j-p_j} \]

NLSet'}
Not-Last: Filtering Algorithm

$$\text{lct}_i$$

$$\text{lct}_{j-p_j}$$

NLSet'
Not-Last: Filtering Algorithm

\[ \text{lct}_i \]

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

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

NLSet'

lct_i

lct_{j-p_j}
Not-Last: Filtering Algorithm

NotLast(T={1..n}) {
    lct’_i ← lct_i, \forall i \in 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
    \Theta ← \Theta-Tree.init({1..n}) // O(n log n) time

    for (i ← T_{lct}) {
        while (lct_i > lct_k-p_k) {
            \Theta.insert(k) // O(log n) time
            j ← k // lct_j-p_j = max \{lct_\Omega - p_\Omega : \Omega \subseteq NLSet(T,i)\}
            k ← ite.next()
        }
        if (ect_{\Theta \backslash i} > lct_i-p_i) { // O(log n) time
            lct’_i ← min(lct_i, lct_j-p_j)
        }
    }
    lct_i ← lct’_i, \forall i \in T
}
Edge Finding

- \( \forall \Omega \subseteq T, \forall i \in T \setminus \Omega \text{ } = \text{arbitrary non-empty subset of } T \)

- \( \text{est}_{\Omega \cup i} + p_{\Omega \cup i} > \text{lct}_{\Omega} \Rightarrow \Omega \ll i \Rightarrow \text{est}_i \leftarrow \max \{\text{est}_i, \text{ect}_\Omega\} \) (EF)

- i must be scheduled after the set \( \Omega \)

impossible to schedule \{A,B,C,D\} before \( \text{lct}_{\{B,C,D\}} \) thus we must have \( \{B,C,D\} \ll A \)
• Reformulation of EF for easier implementation

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

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

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

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

• Implementation using Θ-tree considering j and i wrt LCut(T,j)

- Θ = LCut(T,j)
- Θ-Tree.insert(i), check if ect_Θ > lct_j
- Θ.remove(i)

O(log n) for testing one (i,j)
O(n^2 \log n) overall => too slow!
\(\Theta-\Lambda\)-Tree = generalization of \(\Theta\)-Tree

- \(\text{ect}(\Theta-\Lambda) = \max(\{\text{ect}_\Theta\}, \{\text{ect}_{\Theta|i} : i \in \Lambda\})\)
  - earliest completion time if at most one gray activity used

- New values stored in the nodes (in addition to \(\Sigma P_v \) & \(\text{ect}_v\))
  - \(\Sigma P_v = \max \{p_{\Theta'} | \Theta' \subseteq \text{Leaves}(v) \& |\Theta' \cap \Lambda| \leq 1\}\)
  - \(\text{ect}_v = \text{ect}_{\text{Leaves}(v)} = \max \{\text{est}_{\Theta'} + p_{\Theta'} | \Theta' \subseteq \text{Leaves}(v) \& |\Theta' \cap \Lambda| \leq 1\}\)

- Update rule
  - \(\Sigma P_v = \max \{\Sigma P_{\text{left}(v)} + \Sigma P_{\text{right}(v)}, \Sigma P_{\text{left}(v)} + \Sigma P_{\text{right}(v)}\}\)
  - \(\text{ect}_v = \max \{\text{ect}_{\text{right}(v)}, \text{ect}_{\text{left}(v)} + \Sigma P_{\text{right}(v)}, \text{ect}_{\text{left}(v)} + \Sigma P_{\text{right}(v)}\}\)
Example

- Θ-Λ-Tree: Θ={a,b,d} Λ={c}

```
SigmaP = 21
ect = 44
SigmaP = 26
```

```
SigmaP = 11
ect = 34
SigmaP = 11
```

```
est_a = 0
p_a = 5
SigmaP_a = 5
ect_a = 5
SigmaP_a = 5
```

```
est_b = 25
p_b = 9
SigmaP_b = 9
ect_b = 34
SigmaP_b = 9
```

```
est_c = 30
p_c = 5
SigmaP_c = 0
ect_c = -\infty
SigmaP_c = 5
```

```
est_d = 32
p_d = 10
SigmaP_d = 10
ect_d = 42
SigmaP_d = 10
```
For each node $v$ we can also compute the gray activity 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:**
  - $\text{resp}_{\Sigma P}(v) = \text{resp}_{\Sigma P}($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 $\text{ect}_v = \text{ect}_{\text{left}(v)} + \Sigma P_{\text{right}(v)}$
### Complexities

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\emptyset, \Lambda) := (\emptyset, \emptyset)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>$(\emptyset, \Lambda) := (T, \emptyset)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>$(\emptyset, \Lambda) := (\emptyset \setminus {i}, \Lambda \cup {i})$</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{ect}(\Theta - \Lambda) > \text{lct}_j) \{
    i \leftarrow \text{resp}_{\text{ect}}(\Theta - \Lambda)
    \text{est}_i \leftarrow \max\{\text{est}_i, \text{ect}_\Theta\}
    \Lambda \leftarrow \Lambda \setminus i \text{ // } O(\log n)
\}

Retrieve the activity of \Lambda responsible
Edge Finding Algorithm

EdgeFinding(T={1..n}) {
  \((\Theta, \Lambda) = (T, \emptyset) \quad // \quad O(n \log n) \text{ time} \)
  \(T_{lct} \leftarrow \text{sortZA}([1..n], \text{sortKey} = \text{lct}) \quad // \quad O(n \log n) \text{ time} \)
  \(\text{ite} \leftarrow \text{iterator}(T_{lct})\)
  \(j = \text{ite.next}()\)
  \(\text{while (ite.hasNext())} \{\}
    \quad \text{if (ect}_\Theta > \text{lct}_j) \text{ throw InconsistencyException} \quad // \quad \text{overload} \)
    \quad \((\Theta, \Lambda) = (\Theta \setminus j, \Lambda \cup j) \quad // \quad O(\log n) \text{ time} \)
    \quad \(j \leftarrow \text{ite.next}()\)
    \(\text{while (ect}(\Theta - \Lambda) > \text{lct}_j) \{ \quad // \quad O(1) \text{ time} \)
      \quad \(i \leftarrow \text{resp}_{\text{ect}}(\Theta - \Lambda)\)
      \quad \text{est}_i \leftarrow \max\{\text{est}_i, \text{ect}_\Theta\}\)
      \quad \Lambda \leftarrow \Lambda \setminus i \quad // \quad O(\log n) \text{ time}\)
    \}
  \}
}
None of the algorithms above is idempotent.

According to Petr Vilím (see next slide), the following order for fixpoint computation is very efficient:

Putting it all together
Most of the notation, examples, … come from Petr Vilím’s PhD thesis ([http://vilim.eu/petr/disertace.pdf](http://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 $O(n^3)$.

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