Minimizing the Number of Tardy Jobs

1|| $\sum U_j$

Example

<table>
<thead>
<tr>
<th>$j$</th>
<th>$p_j$</th>
<th>$d_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>20</td>
</tr>
</tbody>
</table>

Ideas:

- Need to choose a subset of jobs $S$ that meet their deadlines.
- Schedule the jobs that meet their deadlines in EDD order (Why?)
- Schedule the remaining jobs in an arbitrary order.

Question: How do you choose the subset?
Algorithm for $1|| \sum U_j$

- Give an incremental algorithm
- Consider jobs in deadline order
- **Invariant**: Maintain a maximum cardinality set of jobs that meet their deadlines, among such sets, choose the one with the smallest total amount of processing time.
Algorithm for $1||\sum U_j$ 

- Give an incremental algorithm
- Consider jobs in deadline order
- **Invariant**: Maintain a maximum cardinality set of jobs that meet their deadlines, among such sets, choose the one with the smallest total amount of processing time.

**Algorithm**

- Sort jobs by deadlines; $S = \emptyset$
- For each job $j$ in deadline order
  - $S = S \cup \{j\}$
  - if $j$ doesn’t meet it’s deadline when $S$ is scheduled
    * $S = S - \{\text{job in } S \text{ with largest processing time}\}$
Analysis

- Run time
  - Need to sort – \( O(n \log n) \)
  - Need to maintain the schedule for S and delete the job with largest processing time. (Maintain a set of numbers doing insert, delete and delete max operations).
Analysis

- Run time
  - Need to sort – $O(n \log n)$
  - Need to maintain the schedule for S and delete the job with largest processing time. (Maintain a set of numbers doing insert, delete and delete max operations). – Use a priority queue, each operations is $O(\log n)$ time.

Analysis: Proof by Induction. After each step $k$, let $S_k$ denote $S$.
- $S_k$ schedules a maximum sized subset of $\{1, \ldots, k\}$
- Among all such subsets $S_k$ is the one with the minimum total processing time.
Another Example

<table>
<thead>
<tr>
<th>$j$</th>
<th>$p_j$</th>
<th>$d_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>15</td>
</tr>
</tbody>
</table>
Special Case of a common deadline

- $1||\sum U_j$ is easy.
- What about $1||\sum w_jU_j$

Example

<table>
<thead>
<tr>
<th>$j$</th>
<th>$p_j$</th>
<th>$w_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>20</td>
</tr>
</tbody>
</table>

$D$ is 40.

- We are choosing a minimum weight subset of jobs that miss their deadline
- Equivalently: we are choosing a maximum weight subset of jobs that make their deadlines.
- Equivalently: Choosing a maximum weight set of jobs that fit in a “bin” of certain size.
A one constraint lp, a knapsack problem.

- If you can take objects fractionally, then the greedy algorithm \( \frac{w_j}{p_j} \) is optimal.
- What about the integral (non-preemptive case).

**Example**

<table>
<thead>
<tr>
<th>( j )</th>
<th>( p_j )</th>
<th>( w_j )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>90</td>
<td>89</td>
</tr>
</tbody>
</table>

\( D \) is 100.
Solving Knapsack Via Dynamic Programming

1. Non-polynomial. We will explicitly solve the problem for all possible values of either time or weight (in this example time.)

2. Polynomial would be polynomial in $n, m, \log W, \log D$, where $W = \max_j w_j$.

3. Running time will be polynomial in $n, m, W, D$. Called pseudopolynomial.

4. Reasonable approach when $W$ and/or $D$ is not too large.

Main Ideas:

- Parameterize solution, and define optimal solutions of a certain size in terms of solutions with smaller parameter values.

- Build up a table of solutions, eventually obtaining the solution for the desired parameter value.
DP for Knapsack: maximum weight competing by deadline

- \( f(j, t) \) will be the best way to schedule jobs \( 1, \ldots, j \) with \( t \) or less total processing time.
- Best means maximum total weight.

- What is \( f(n, D) \)?
- Maximum weight way to schedule all the jobs using at most \( D \) total processing time.
- This is the problem we want to solve.
To schedule jobs $1, \ldots, j$ using $t$ total processing time there are two cases:

- job $j$ is not scheduled.
- job $j$ is scheduled
To schedule jobs \(1, \ldots, j\) using \(t\) total processing time there are two cases:

- job \(j\) is not scheduled.
- job \(j\) is scheduled

- If \(j\) is not scheduled, then the optimal solution for \(1, \ldots, j\) is the same as the optimal solution for \(1, \ldots, j - 1\), hence \(f(j, t) = f(j - 1, t)\)

- If \(j\) is scheduled, then there are two subcases:
  - \(j\) was also scheduled using \(t - 1\) time units, hence \(f(j, t) = f(j, t - 1)\)
  - \(j\) was not scheduled when we used \(t - 1\) time units. We need to add \(j\) to the schedule, hence we have to look at the optimal schedule using \(t - p_j\) units of processing, hence: \(f(j, t) = f(j - 1, t - p_j) + w_j\).

We don’t know which case happens, so we try all and take the maximum

\[
 f(j, t) = \max\{f(j - 1, t), f(j, t - 1), f(j - 1, t - p_j) + w_j\} 
\]
Example

\[ f(j, t) = \max\{f(j - 1, t), f(j, t - 1), f(j - 1, t - p_j) + w_j\} \]

<table>
<thead>
<tr>
<th>(j)</th>
<th>(p_j)</th>
<th>(w_j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>90</td>
<td>89</td>
</tr>
</tbody>
</table>

\(D\) is 100.