Dynamic programming for a scheduling problem

Giovanni Righini

O.R. Complements

UNIVERSITÀ DEGLI STUDI DI MILANO

Motivation

In industrial production plants, the production lots (jobs) must typically traverse several working phases and it may be convenient or necessary to reorganize their sequence.

Problems:

- find the optimal set of changes of a given sequence in output from phase 1, to produce a feasible sequence in input to phase 2;
- find a feasible set of changes of a given sequence in output from phase 1, to optimize the outcome of phase 2.

Our setting: jobs extracted from the initial sequence are kept in a stack of finite capacity until they are re-inserted.

Model: data and variables

Data:

- a sequence N of jobs, numbered from 1 to n; $[i, \ldots, j]$ indicates a generic subsequence in it;
- a processing time p_i for each job $i \in N$;
- a due date d_i for each job $i \in N$;
- a weight w_i for each job $i \in N$;
- the capacity of the stack, S.

Definition. A move (i, j) consists of deleting job $i \in N$ and reinserting it just after all jobs of the subsequence $[i, \ldots, j]$.

Variables.
$x_{ij} \in \{0, 1\}$ $\forall i < j \in N$: $x_{ij} = 1 \Leftrightarrow$ move (i, j) is done.

Model: constraints

Constraints:

- LIFO constraints and no multiple moves:

$$x_{ij} + x_{kh} \leq 1 \quad \forall i, j, k, h \in N : ((i = k) \wedge (j \neq h)) \vee (i < k \leq j < h).$$

- Stack capacity constraints:

$$\sum_{i,j \in N: i \leq k, j > k} x_{ij} \leq S \;\; \forall k = 1, \ldots, n-1.$$

Only sequential moves and nested moves are allowed.

Proposition 1. *Necessary and sufficient condition for two sequential moves (i, j) and (k, h) to be compatible is that $i < j < k < h$ (or viceversa $k < h < i < j$).*

Proposition 2. *Necessary and sufficient condition for two nested moves (i, j) and (k, h) to be compatible is that $i < k < h \leq j$ (or viceversa $k < i < j \leq h$).*

Sequential and nested moves

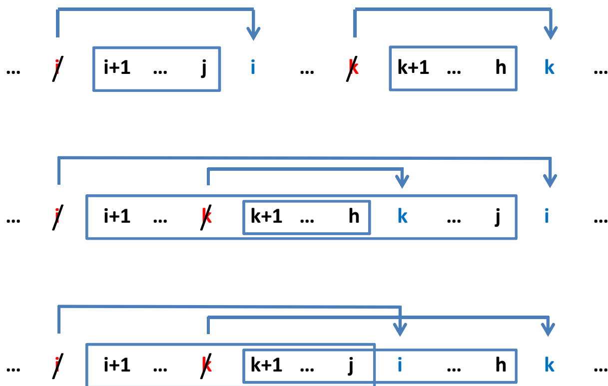

Figura: Sequential, nested and incompatible moves (i, j) and (k, h).

Model: objectives

- minimum total weighted completion time,
- minimum maximum lateness,
- minimum number of late jobs,
- minimum weighted number of late jobs.

Lateness (delay with respect to the due date):

$$L_i = c_i - d_i \quad \forall i \in N,$$

where c_i is the completion time of job $i \in N$.

Notation

Start time and completion time in the initial sequence:

$$s_i = \sum_{k=1}^{i-1} p_k \qquad e_i = \sum_{k=1}^{i} p_k \quad \forall i \in N.$$

All start and completion times can be computed in $O(n)$ time.

The duration δ_{ij} of each subsequence $[i, \ldots, j]$ can be computed in $O(n^2)$ time: $\delta_{ij} = e_j - s_i$.

Levels of nested moves:

- a move that does not contain nested moves is at level 1;
- a move containing moves at level up to $\ell - 1$ is a move at level ℓ.

Triple (i, j, ℓ): move on $[i, \ldots, j]$ at level ℓ.

Stack capacity: $\ell \in [1, \ldots, S]$.

Total weighted completion time

Observation 1. The completion time of each job preceding i and following j in the initial sequence is not affected by the move (i, j, ℓ).

Observation 2. The cost variation $m(i, j)$ due to moving job i after job j is

$$m(i, j) = w_i \sum_{k=i+1}^{j} p_k - p_i \sum_{k=i+1}^{j} w_k \;\; \forall i < j \in N.$$

First term: increase of the weighted completion time of job i;
Second term: total decrease of the weighted completion times of the jobs in $[i + 1, j]$.

Observation 3. The effect of a move (i, j, ℓ) at any level $\ell > 1$ does not depend on the order of the jobs in $[i + 1, j]$.

The objective variation due to move (i, j, ℓ) is $m(i, j)$ independently on ℓ.

Dynamic programming: state

Property. The cost of sequential moves and nested moves is additive.

Sequence: optimally rearrange all subsequences $[i, j]$ using ℓ levels, from $\ell = 1$ to $\ell = S$.

State: (i, j, ℓ).

Each state has an associated cost $\mu^*(i, j, \ell)$: minimum cost (i.e. increase of the objective function) obtained by rearranging jobs in $[i, j]$ using up to ℓ levels.

Define $c(i, j, \ell)$: optimal cost of move (i, j, ℓ).

$$\begin{cases} c(i,j,1) = m(i,j) & \forall i < j \in N \\ c(i,i,\ell) = 0 & \forall i \in N, \forall \ell = 1,\dots,S \\ c(i,j,\ell) = m(i,j) + \mu^*(i+1,j,\ell-1) & \forall i < j \in N, \forall \ell = 2,\dots,S. \end{cases}$$

Dynamic programming: extension function

Extension function for each level $\ell = 1, \ldots, S$:

$$\begin{cases} \mu^*(i,i,\ell) = 0 & \forall i \in N \\ \mu^*(i,j,\ell) = & \\ = \min\{\min_{k \in [i,j-1]}\{c(i,k,\ell) + \mu^*(k+1,j,\ell)\}, c(i,j,\ell)\} & \forall i < j \in N. \end{cases}$$

Optimal final value: $\mu^*(1, n, S)$.

Computational complexity:

- computing $m(i,j)$ takes $O(n^2)$,
- computing $c(i,j,\ell)$ takes $O(1)$ for each (i,j,ℓ),
- computing $\mu^*(i,j,\ell)$ takes $O(n)$ for each (i,j,ℓ),
- the number of states (i,j,ℓ) is $O(n^2 S)$.

Hence the time complexity is $O(n^3 S)$ (bounded by $O(n^4)$).

Maximum lateness

Notation:

- $L(i)$: lateness of job i in the input sequence.
- $L(i,j)$: max lateness of the jobs in $[i,j]$ in the input sequence.

Observation 1. The completion time (and the lateness) of each job preceding i and following j in the initial sequence is not affected by the move (i,j,ℓ).

Hence, even in this case, moves involving disjoint subsequences can be evaluated independently.

However, the effect of a move (i,j,ℓ) depends on the arrangement of the jobs in $[i+1,j]$:

$$u(i,j) = \max\left\{L(i) + \sum_{k=i+1}^{j} p_k, L(i+1,j) - p_i\right\} \quad \forall i < j \in N.$$

Dynamic programming: state

Sequence: optimally rearrange all subsequences $[i, j]$ using ℓ levels, from $\ell = 1$ to $\ell = S$.

State: (i, j, ℓ).

Each state has an associated cost $\lambda^*(i, j, \ell)$: minimum max lateness of a rearrangement of jobs in $[i, j]$ using up to ℓ levels.

Define $g(i, j, \ell)$: min max lateness obtained by rearranging $[i, j]$ at level ℓ.

$$\begin{cases} g(i,i,1) = L(i) & \forall i \\ g(i,j,1) = u(i,j) & \forall i < j \\ g(i,j,\ell) = \max\{L(i) + \sum_{k=i+1}^{j} p_k, \lambda^*(i+1,j,\ell-1) - p_i\} & \forall i < j, \forall \ell \geq 2. \end{cases}$$

Dynamic programming: extension function

Extension function for each level $\ell = 1, \ldots, S$:

$$\begin{cases} \lambda^*(i,i,\ell) = L(i) & \forall i \in N \\ \lambda^*(i,j,\ell) = & \\ = \min\left\{\min_{k\in[i,j-1]}\left\{\max\{g(i,k,\ell), \lambda^*(k+1,j,\ell)\}\right\}, g(i,j,\ell)\right\} & \forall i < j \in N. \end{cases}$$

Optimal final value: $\lambda^*(1,n,S)$.

Computational complexity:

- computing $u(i,j)$ takes $O(n^2)$,
- computing $g(i,j,\ell)$ takes $O(1)$ for each (i,j,ℓ),
- computing $\lambda^*(i,j,\ell)$ takes $O(n)$ for each (i,j,ℓ),
- the number of states (i,j,ℓ) is $O(n^2 S)$.

Hence the time complexity is $O(n^3 S)$ (bounded by $O(n^4)$).

Number of late jobs

Observation 1. In general, given a subsequence $[i, j]$, the number of late jobs in it depends not only on their processing times and due dates, but also on start time of the subsequence.

Also the rearrangement of a subsequence that minimizes the number of late jobs depends on the start time of the subsequence.

Example. Sequence $[1, 2, 3]$ with $p^T = [7\ 10\ 10]$ and $d^T = [27\ 25\ 15]$.

Consider the subsequence $[2, 3]$:

- if the subsequence starts at $t = 7$ (i.e. job 1 is not moved), then
  - $[2, 3]$ has $C^T = [7\ 17\ 27]$ with one late job
  - $[3, 2]$ has $C^T = [7\ 27\ 17]$ with two late jobs
- if the subsequence starts at $t = 0$ (i.e. job 1 is moved), then
  - $[2, 3]$ has $C^T = [27\ 10\ 20]$ with one late job
  - $[3, 2]$ has $C^T = [27\ 20\ 10]$ with no late jobs.

Number of late jobs

Observation 2. The arrangement of a subsequence $[i, j]$ that minimizes the q-th largest lateness value, with $q \in \{1, \ldots, j - i + 1\}$, does not depend on the start time of the subsequence.

In general, the arrangements minimizing the q-th largest lateness value are different for different q.

However, when a subsequence is moved forward or delayed, all lateness values decrease or increase by the same amount and hence their order remains unchanged.

Idea: enumerate the number of late jobs in each subsequence.

Dynamic programming

Observation 3. The number of late jobs in a subsequence can only decrease, when the start time of the subsequence is decreased.

Observation 4. To have m late jobs in a subsequence $[i, j]$, its start time must be decreased by at least $\max^{(m+1)}_{k \in [i,j]} \{L_k\}$.

The case $m = j - i + 1$ (all jobs are late) is not meaningful.

Definition. *ℓ-rearrangement*: a feasible rearrangement of a subsequence that can be obtained by moves up to level ℓ.

State and dominance

State. For each subsequence $[i, j]$, for each $\ell = 0, \ldots, S$ and for each $m = 0, \ldots, j - i$, $s(i, j, m, \ell)$ is the minimum reduction of the starting time that is needed to have m late jobs in $[i, j]$, when $[i, j]$ can be rearranged with moves up to level ℓ.

Dominance. All ℓ-rearrangements of a subsequence $[i, j]$ that allow to obtain m late jobs in it when the subsequence is moved forward by s' are dominated by the ℓ-rearrangement of $[i, j]$ that allows to obtain the same number m of late jobs in it when the subsequence is moved forward by $s'' < s'$.

This dominance relation between rearrangements allows to avoid the combinatorial explosion implied by the binary programming formulation of the problem.

The sequence

For each subsequence $[i, j]$ and for each level $\ell = 1, \ldots, S$ we enumerate all feasible ℓ-rearrangements of $[i, j]$.

The enumeration of all feasible ℓ-rearrangements of $[i, j]$ is done by enumerating all possible moves (i, k, ℓ) with $k = i, \ldots, j$, after having enumerated all feasible ℓ-rearrangements of $[k+1, j]$, which is not affected by the move.

Therefore, the value of $s(i, j, m, \ell)$ requires the values of s

- for all subsequences $[i+1, k]$ and level $\ell - 1$;
- for all subsequences $[k+1, j]$ and level ℓ.

The sequence to follow to compute the states is given by considering

- increasing ℓ from 0 to S;
- for each given ℓ, decreasing j from n to 1;
- for each given ℓ and j, decreasing i from j to 1;
- for each given ℓ and $[i, j]$, decreasing m from $j-1$ to 0.

The extension rule

For a given quadruple (i, j, m, ℓ), the best ℓ-rearrangement of $[i, j]$ to have m late jobs in it is obtained by comparing the effect of all possible moves (i, k, ℓ) for $k = i, \ldots, j$ (where $k = i$ represents “no move”).

For each $k = i, \ldots, j$ the resulting number of late jobs is given by three terms:

- $m_1 \in \{0, \ldots, k - i\}$ late jobs from subsequence $[i + 1, k]$ moved forward by p_i;
- $m_2 \in \{0, \ldots, j - k\}$ late jobs from subsequence $[k + 1, j]$ with the same start time;
- $m_3 \in \{0, 1\}$ late jobs from job i delayed by $\delta(i + 1, k)$.

Initialization

At level $\ell = 0$, the values $s(i, j, m, 0)$ are evaluated on the initial sequence using the lateness values of the jobs in it.

For each subsequence $[i, j]$, $\max^{(m)}_{k \in [i,j]}\{L_k\}$ indicates the m-th largest value among the lateness values of the jobs in the subsequence (m-lateness, for short).

Initialization.

$$s(i, j, m, 0) = \max^{(m+1)}_{k \in [i,j]}\{L_k\} \quad \forall\, i \leq j,\ \forall m = 0, \ldots, j - i.$$

Recursion

For each ℓ-rearrangement, we consider the ordered sequence of lateness values in it and for each number $m-1$ of late jobs, we keep the corresponding minimum m-lateness.

When move (i,k,ℓ) is done, the multiset of lateness values produced is given by

$$\mathcal{R}(i,j,k,\ell) = \bigcup_{u=0}^{k-i-1} \{s(i+1,k,u,\ell-1)-p_i\} \cup \bigcup_{u=0}^{j-k-1} \{s(k+1,j,u,\ell)\} \cup \{C_k - d_i\}.$$

By Observation 2, the optimal $(\ell-1)$-rearrangement of subsequence $[i+1,k]$ does not change when the subsequence is moved forward by p_i.

The value of the new state is computed as follows:

$$s(i,j,m,\ell) = \min_{k\in[i,j]} \left\{ \overset{(m+1)}{\max} \mathcal{R}(i,j,k,\ell) \right\}.$$

The final optimal value is given by $\min\{m : s(1,n,m,S) \leq 0\}$.

```
//Initialization//
for i = 1, ..., n do
   for j = i, ..., n do
      for m = 0, ..., j − i do
         s(i, j, m, 0) ← max^(m+1){L_i, ..., L_j}
//Recursion//
for ℓ = 1, ..., S do
   for j = 1, ..., n do
      for i = j, ..., 1 do
         for k = i, ..., j do
            R_k ← {}
            for u = 0, ..., k − i − 1 do
               SortedInsert (R_k, s(i + 1, k, u, ℓ − 1) − p_i)
            for u = 0, ..., j − k − 1 do
               SortedInsert (R_k, s(k + 1, j, u, ℓ))
            SortedInsert (R_k, C_k − d_i)
         for m = 0, ..., j − i do
            s(i, j, m, ℓ) ← min_{k=i,...,j}{max^(m+1) R_k}
Return(min{m : s(1, n, m, S) ≤ 0})
```

Complexity

The worst-case computational complexity of the initialization is $O(n^3)$ (not the bottleneck).

In the recursion there are five nested loops, yielding an overall worst-case time complexity of $O(n^4 S)$, which is bounded by $O(n^5)$ (strongly polynomial).

The implementation (not the complexity) can be improved (see Nicosia et al., 2021).

Weighted number of late jobs

In its weighted version (a weight w_j is associated with each job j) the problem is NP-hard.

Assume weights are integers. Then, instead of enumerating the number of late jobs, we enumerate the weighted number of late jobs.

The same algorithm still works, but its computational complexity is now $O(n^4 W)$, with $W = \sum_j w_j$ (pseudo-polynomial complexity).

For large weights, it is possible to prove that the complexity can be bounded by B instead of W, where $B = \sum_{q=1}^{S} \max_j^{(q)}\{p_j\}$ (see Pferschy et al., 2022).