Graph-Based Reformulation of the Process Scheduling Problem: A Comparative Study Between Linear Programming Models and Shortest Path Algorithms

October 2025

Abstract

This work explores a graph-based reformulation of the classical process scheduling problem, originally modeled as a linear programming optimization similar to the knapsack problem. The proposed approach maps each scheduling instance onto a directed acyclic graph (DAG), where each feasible scheduling configuration corresponds to a path from a source to a sink node. In this framework, maximizing the total priority of scheduled processes is equivalent to solving a *Longest Path Problem (LPP)*.

To enable the use of specific shortest-path algorithms, the problem is inverted into a *Shortest Path Problem (SPP)* by negating edge weights, allowing the application of both a modified Label Correcting algorithm and the Reaching Method, a label-setting approach. Computational tests compare the performance of these algorithms with CPLEX-based models for both the LPP and the original scheduling formulation.

The results confirm that while CPLEX guarantees optimal solutions, graph-based approaches—particularly the Reaching Method—achieve comparable accuracy with significantly lower computational costs. However, directly solving the scheduling problem as a linear knapsack formulation remains the most efficient strategy overall. The study highlights the practical value of shortest-path algorithms in optimization domains not inherently graph-based.

Keywords

Process scheduling, graph optimization, linear programming, shortest path problem, CPLEX, label correcting algorithms, reaching method

1. Introduction to the problem

The problem addressed in this work is inspired by the operation of modern electronic computers, where multiple processes compete for computational resources and must share the CPU, as they cannot use it simultaneously. Each process typically has an estimated execution time and an associated priority, while the CPU provides a certain number of cores, each capable of executing only one process at a time.

The goal of process scheduling is to optimally allocate processes to the available cores, seeking to maximize the total priority value of the processes executed within a given time limit.

Process scheduling can be formulated as a **Linear Programming (LP)** problem, solvable through methods such as the **Simplex algorithm**, using solvers like **CPLEX**. However, it is also possible to construct a **graph GGG** based on the problem data, in such a way that the optimal solution to a **Longest Path Problem (LPP)** on GGG coincides with the optimal solution of the original scheduling problem.

In this study, graphs will be constructed from specific instances of the process scheduling problem to solve both a **Longest Path Problem** and its corresponding **Shortest Path Problem (SPP)** using CPLEX and two well-known algorithms designed for SPPs: the **Reaching Method** and a **Modified Label Correcting algorithm**. The results obtained from these algorithms will then be compared with those from the original scheduling formulation.

The algorithms were chosen considering that the resulting shortest path problems will have **negative edge weights** and will be **acyclic**. Moreover, the study focuses on processors with a **single core**, since when multiple cores are involved, the solution can no longer be represented as a single linear path.

2. Theoretical background

2.1. Original process scheduling problem

It is well known that the process scheduling problem closely resembles the **Knapsack Problem** and can be formalized as follows, in order to be solved using classical **Linear Programming (LP)** algorithms:

Let:

- n: number of processes to be executed
- t_j: execution time required by process j
- p_j: priority or profit of process j
- T_{lim}: maximum execution time available for the CPU

With the following decision variables:

- $X_j \in \{0, 1\}$: equals 1 if process j is assigned to the core, and 0 otherwise.

The **objective function** aims to maximize the total priority value of the processes executed within the CPU time limit:

$$\boldsymbol{max} \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{xj} * \boldsymbol{pj}$$

Subject to the following **constraints**, which ensure compliance with the total available CPU time and the binary nature of the decision variables:

$$\sum_{\boldsymbol{j=1}}^{\boldsymbol{n}} \boldsymbol{tj} * \boldsymbol{xj} \leq \boldsymbol{Tlim}$$

$$\boldsymbol{xj} \in \{\mathbf{0}, \mathbf{1}\}, \qquad \forall \boldsymbol{j} \in \{\mathbf{1}, \dots, \boldsymbol{n}\}$$

As anticipated, this formulation allows an efficient construction of a **CPLEX optimization model** capable of finding the optimal solution to the scheduling problem.

2.2. Relationship with the Longest Path Problem

Starting from the previously described formulation, as described in [1], it is possible to construct a **directed acyclic graph (DAG)** with $\boldsymbol{n+2}$ layers, defined as follows:

- The first layer contains only a source node $\boldsymbol{s}$.
- The last layer contains only a sink node $\boldsymbol{t}$.
- The intermediate layers correspond to the $\boldsymbol{n}$ processes.

In particular, for the layers associated with the n processes:

- The layer corresponding to process j represents the progression through the first j processes and contains $T_{lim}+1$ nodes: $j^0, j^1, \ldots, j^{Tlim}$.
- Node j^k represents a **state** in which processes $1, 2, \ldots, j$ considered so far have consumed k units of CPU time in total.
- Each node j^k has at most **two outgoing edges**, representing the decisions to include or exclude process $j+1$ from execution, with associated costs p_{j+1} and 0, respectively.
- An edge corresponding to the inclusion of a new process exists only if the remaining available CPU time is sufficient.
- All nodes in the layer associated with the last process are connected to the sink node t with **zero-cost edges**.

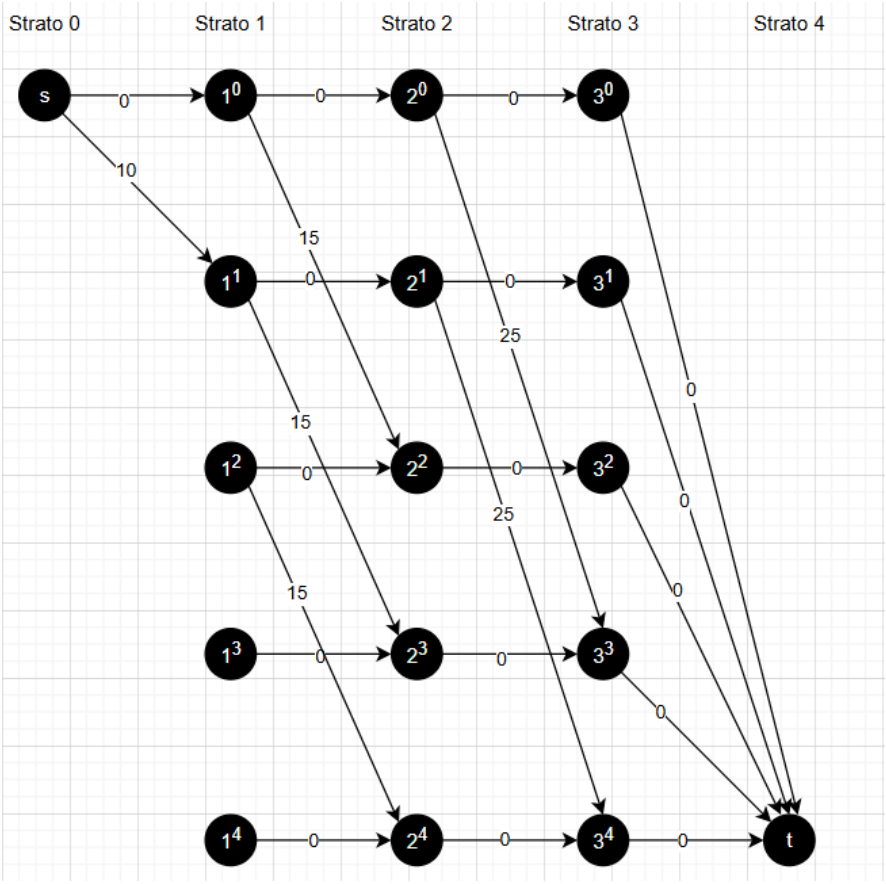
A graph structured in this way is particularly relevant to our objective, since **each path from s to t** defines a feasible solution to the original scheduling problem, with a **path cost equal to the total priority** of the processes assigned to the CPU. Similarly, **each feasible scheduling solution** corresponds to a directed path from s to t.

Consequently, finding the **optimal solution** to the process scheduling problem—i.e., maximizing the total priority of the processes executed within the CPU time limit—is equivalent to finding the **longest path from s to t** in the constructed DAG, given that the edge costs are determined based on process priorities.

For example, consider a scheduling problem on a single-core processor with $T_{lim}=4$ and three available processes characterized by the following parameters:

j	1	2	3
p_j	10	15	25
t_j	1	2	3

The resulting DAG structure is as follows:



We have n+2 layers: the n layers corresponding to the processes, each containing Tlim+1 nodes representing the time already used, plus the source and sink layers.

The **edges** represent the choices of executing or skipping a process, each with an associated cost. Every directed path from ***s*** to ***t*** in the graph corresponds to a feasible solution of the original scheduling problem.

The **optimal solution** to the original problem consists in selecting processes **1 and 3**, which maximize the total priority with an **objective function value of 35**.

The corresponding path in the graph is $s \rightarrow 1^1 \rightarrow 2^1 \rightarrow 3^4 \rightarrow t$, representing the inclusion of process 1 (edge $s \rightarrow 1^1$), the exclusion of process 2 (edge $1^1 \rightarrow 2^1$), and the inclusion of process 3 (edge $2^1 \rightarrow 3^4$).

By analyzing the graph, it is easy to see that $s \rightarrow 1^1 \rightarrow 2^1 \rightarrow 3^4 \rightarrow t$ is also the **longest path** between ***s*** and ***t***, as described above.

Solving the scheduling problem in this way is reminiscent of **dynamic programming**, since each node represents a state summarizing past decisions, and transitions between states correspond to decisions with associated costs (which, in this case, must be maximized).

2.3. Longest Path Problem Formulation

The search for a longest path from **s** to **t** in a graph can also be formulated as a **linear programming problem**, using the same set of constraints as a standard **Shortest Path Problem (SPP)**, but with an opposite objective.
While the SPP aims to minimize the total cost along the path, the **Longest Path Problem (LPP)** aims to **maximize** it.

The **objective function** is therefore:

$$\boldsymbol{max} \sum_{\boldsymbol{(i,j)\in A}} \boldsymbol{xij} * \boldsymbol{cij}$$

Where the edge costs in the DAG correspond to the decision costs of including or excluding processes in the CPU assignment. These edge costs are either zero or equal to the priority values associated with the processes.

The **flow conservation constraints** depend on the node type:

$$\sum_{\boldsymbol{(i,j)\in\delta+(i)}} \boldsymbol{xij} - \sum_{\boldsymbol{(i,j)\in\delta-(i)}} \boldsymbol{xji} = \mathbf{1} \qquad \boldsymbol{se\ i = s}$$

$$\sum_{\boldsymbol{(i,j)\in\delta+(i)}} \boldsymbol{xij} - \sum_{\boldsymbol{(i,j)\in\delta-(i)}} \boldsymbol{xji} = \mathbf{-1} \qquad \boldsymbol{se\ i = t}$$

$$\sum_{\boldsymbol{(i,j)\in\delta+(i)}} \boldsymbol{xij} - \sum_{\boldsymbol{(i,j)\in\delta-(i)}} \boldsymbol{xji} = \mathbf{0} \qquad \boldsymbol{\forall i \in N - \{s,t\}}$$

Here, $\boldsymbol{\delta^+}$ and $\boldsymbol{\delta^-}$ represent the sets of outgoing and incoming edges of node *i*, respectively.

From the nature of the path search from **s** to **t**, we expect that:

- One selected edge leaves **s**;
- One selected edge enters **t**;
- All other nodes have either one selected incoming and one selected outgoing edge, or no edges at all.

The **decision variables** are binary, taking the value 1 if the corresponding edge is included in the selected path and 0 otherwise:

$$\boldsymbol{xij} \in \{\mathbf{0}, \mathbf{1}\}, \qquad \forall(\boldsymbol{i}, \boldsymbol{j}) \in \boldsymbol{A}$$

A Longest Path Problem formulated in this way can be solved using the **Simplex method**, and therefore by many standard linear programming solvers, including **IBM ILOG CPLEX**, as will be demonstrated in the following sections.

2.4. Shortest Path Problem and Label Correcting Algorithms

Once the graph has been generated from the process scheduling data, it is possible — in addition to solving a Longest Path Problem — to **invert the sign of the edge costs** and thus solve a **Shortest Path Problem (SPP)** using specific algorithms.

This operation is valid in our case because the graph is **acyclic**. Therefore, even with negative edge costs, there will be no negative cycles that could make the objective function unbounded (i.e., indefinitely improvable).

However, care must be taken when selecting algorithms, since some of them — such as the well-known **Dijkstra's algorithm** — cannot be applied to graphs containing negative edge costs, as in the DAG constructed from our scheduling problem.

Shortest Path Problems can be solved using two main algorithmic families: **label setting** and **label correcting**, which differ in how they manage node "distance labels," representing upper bounds on the distances from the source node.

- **Label setting algorithms**:
  At each step, they select a temporary node (whose distance label has not yet been proven optimal), confirm it as permanent (its distance label is now proven to be optimal), and update the other distance labels accordingly.

- **Label correcting algorithms**:
  These algorithms, instead, repeatedly consider all temporary labels until no further improvements can be made.

2.4.1. Modified Label Correcting Algorithm

The first algorithm used to solve the SPP in this study is the **Modified Label Correcting algorithm**.

In general, label correcting algorithms iteratively identify an edge that violates the optimality condition:

$$\boldsymbol{d(j) \leq d(i) + cij, \qquad \forall j \in N}$$

and then correct **d(j)**, the distance label of node ***j***, which represents the current upper bound of its distance from the source node.

When the following inequality holds:

$$\boldsymbol{d(j) > d(i) + cij, \qquad \forall j \in N}$$

it means that the current distance label of ***j*** can be improved by selecting ***i*** as its predecessor.

The Modified Label Correcting algorithm proceeds as follows:

1. Initialize all distance labels to infinity.
2. Set the distance label of the source node ***s*** to 0 and its predecessor to *null*.
3. Create a list **LIST**, initially containing only ***s***.
4. While **LIST** is not empty:
   - Remove an element ***i*** from **LIST**.
   - For each outgoing edge ***(i, j)***, if the optimality condition is violated (i.e., $\mathbf{d(j) > d(i) + c_{ij}}$):
     - Update $\mathbf{d(j) = d(i) + c_{ij}}$.
     - Set the predecessor of ***j*** to ***i***.
     - If ***j*** is not already in **LIST**, insert it.

Label correcting algorithms such as this one can be applied to graphs with **arbitrary edge costs** and even with cycles (as long as they are non-negative).
However, they generally exhibit **high computational complexity**: for the Modified Label Correcting algorithm, in the worst case, it can reach $\mathbf{O(n \cdot m \cdot C)}$,
where n is the number of nodes, m the number of edges, and C the maximum possible path cost in the optimal solution.

2.5. Label Setting algorithms

Label setting algorithms generally have **significantly lower computational costs**, but they **cannot be applied in all cases**.

The second algorithm used to solve the Shortest Path Problem is the **Reaching Method**, a label setting algorithm applicable to **acyclic graphs** with **arbitrary edge costs**.

The Reaching Method requires a **topological ordering** of the graph's nodes (which always exists in acyclic graphs like ours).
We define an array order such that:

$$\boldsymbol{order(i) < order(j), \qquad \forall (i,j) \in A}$$

Once a topological order is obtained, the Reaching Method processes nodes one by one, following that order.
At each step, it selects a node i and updates the distance labels of all adjacent nodes j if reaching j through i provides a better path.

The algorithm proceeds as follows:

1. Perform a **topological sort** of the nodes.
2. Initialize the **distance label** of each node to infinity.
3. Set the **distance label** and **predecessor** of the source node s to 0.
4. For each node i in topological order:
    - For each adjacent node j:
        - If $\mathbf{d(j) > d(i) + c_{ij}}$ then:
            - Update $\mathbf{d(j) = d(i) + c_{ij}}$
            - Set the predecessor of j to i.

To obtain a topological ordering, the algorithm leverages the property that **acyclic graphs always contain at least one node with indegree equal to 0**.
Thus, at each iteration:

1. While the graph is not empty:
    - Select a node ***i*** with **indegree = 0**.
    - Insert ***i*** into the topological order.
    - Remove ***i*** and all its incident edges from the graph.

The computational complexity of the Reaching Method is **linear in the number of edges**.
However, when including the **topological sorting step**, the total complexity becomes **O(n + m)**, where ***n*** is the number of nodes and ***m*** is the number of edges.

For acyclic graphs with arbitrary edge costs, it is also possible to use the **Pulling Method**, which processes nodes ***j*** instead of ***i*** in topological order.
However, since in our case adjacency lists are used to represent the graph data, the **Reaching Method** is more suitable — as it does not require computing inverse adjacency lists or maintaining explicit predecessor sets.

3. Algorithm implementation

3.1. Graph generation

A dedicated function was developed to **generate a graph from the scheduling problem data**, namely the **core time** and the **set of processes**, following the theoretical specifications introduced in the previous sections.

The graph is constructed as follows:

- A number of **layers** equal to the number of processes ***n*** is created.
- Each layer contains $\mathbf{T_{lim}}$**+1 nodes**.
- **Edges** are created to represent the **inclusion decisions** for each process, with costs either equal to **zero** or to the **priority** of the corresponding process.
- **Source** and **sink** nodes are then created, corresponding to the final two layers of the graph.

Each edge is assigned a **name** beginning with ***yes*** or ***no***, in order to easily trace back the **decisions** made by the algorithms after execution.
The edge **costs** are stored under the attribute *priority*, for consistency with the formulation of the original problem.

Nodes are represented as **pairs (i, k)**, where ***i*** denotes the **process layer** and ***k*** represents the **time units already consumed** in the corresponding state.
Each edge stores information about its **incident nodes**, **priority value**, and **corresponding decision**.

```
#Crea il grafo a partire dai dati del problema di scheduling
archi = []
for i in range(n ):        #tanti strati quanti processi per adesso
    for k in range(T + 1):        #nodi per strato, rappresentanti k istanti consumati
        if i < n:                    #creazione nodi
            dur = processes[i]["t"]
            pr = processes[i]["p"]
            nm = processes[i]["nome"]

            #arco scelta di non eseguire il processo
            archi.append({"from": (i, k), "to": (i + 1, k), "priority": 0, "dec": f"no_{nm}"})

            #arco scelta di eseguire il processo, se il tempo residuo lo permette
            if k + dur <= T:
                archi.append({
                    "from": (i, k), "to": (i+1, k + dur),
                    "priority": pr, "dec": f"yes_{nm}"
                })

# aggiunge source e sink e li collega ai nodi opportuni
source, sink = "s", "t"
archi.append({"from": source, "to": (0, 0), "priority": 0, "dec": "start"})
for k in range(T + 1):
    archi.append({"from": (n, k), "to": sink, "priority": 0, "dec": f"end_{k}"})
```

For the **Shortest Path algorithms**, the **edge cost** is set to the **negative of the process priority**.

3.2. CPLEX model creation

As the first approach to solving the Longest Path Problem (LPP) on the graphs generated in the previous step, a CPLEX model was implemented. This method was primarily chosen to compare the optimal solutions obtained from the Linear Programming (LP) formulation on the graph with those from the original process scheduling problem.

Using CPLEX for this type of problem is generally disadvantageous, mainly due to both the execution time of the model and the significant overhead involved in its construction.

The model generation follows the theoretical LPP formulation discussed earlier.

First, the **objective function** was defined to maximize the costs (i.e., process priorities) associated with the selected edges in the path:

```
#Obiettivo: massimizza la somma delle priorità
model.maximize(model.sum(e["priority"] * x[(e["from"], e["to"], e["dec"])] for e in archi))
```

Then, the **flow conservation constraints** were added for the source, sink, and all intermediate nodes. Recall that the source node must have exactly one selected outgoing edge, the sink must have exactly one selected incoming edge, and every other node must have an equal number of selected incoming and outgoing edges:

```
#Vincoli di flow conservation
# uscita da source
model.add_constraint(sum(x[k] for k in x if k[0] == source) == 1)

# ingresso a sink
model.add_constraint(sum(x[k] for k in x if k[1] == sink) == 1)

# ogni altro nodo
nodes = set([e["from"] for e in archi] + [e["to"] for e in archi])
nodes.discard(source)
nodes.discard(sink)          #senza contare source e sink
for node in nodes:
    model.add_constraint(
        sum(x[k] for k in x if k[1] == node) == sum(x[k] for k in x if k[0] == node),
        ctname: f"flow_{node}"
    )
```

The constraints were explicitly named according to the corresponding incoming and outgoing flows, allowing for straightforward inspection of the model structure.

Finally, the **binary constraints** were imposed on the decision variables associated with the edges, each named according to its incident nodes and the corresponding scheduling decision:

```
#Variabili binarie per arco
x = {}
for e in archi:
    key = (e["from"], e["to"], e["dec"])
    x[key] = model.binary_var(name=f"x_{e['dec']}_{e['from']}_{e['to']}")
```

Given a process scheduling instance and its corresponding graph, the optimal solutions obtained through CPLEX and through the Longest Path formulation on the generated graph are expected to coincide.

3.3. Modified Lael Correcting

A more efficient way to solve this problem is to invert the sign of the edge costs in the graph, transforming it into a shortest path problem that can be solved using well-established and efficient algorithms.
The choice of algorithms has been discussed earlier and is motivated by the presence of negative edge costs in the acyclic graph derived from the scheduling problem.

The first algorithm used to solve the resulting Shortest Path Problem is the **Modified Label Correcting** method, implemented according to the specifications described in the theoretical section.

After constructing the graph structure, the algorithm initializes the **distance labels**, **predecessors**, the **source node distance**, and the **LIST** that temporarily stores the nodes to be processed:

```
#Inizializza
d = {nodo: float('inf') for nodo in N}
pred = {nodo: None for nodo in N}
d[source] = 0

LIST = deque([source])
```

The main loop of the algorithm then iteratively updates the LIST, checks the **optimality conditions** for the edges, corrects the **distance labels**, and updates the **predecessors** when an improvement is found:

```
while LIST:
    i = LIST.popleft()
    for j, cij in A[i]:
        if d[j] > d[i] + cij:
            d[j] = d[i] + cij
            pred[j] = i
            if j not in LIST:
                LIST.append(j)

return d[sink], pred
```

Finally, the algorithm returns:

- The **distance of the sink node**, which corresponds to the value of the objective function in the original scheduling problem.
- The **list of predecessors**, which identifies the path found by the algorithm.

3.4. Reaching method

Although the **Modified Label Correcting** algorithm is significantly more efficient than the CPLEX model for solving the Longest Path Problem, the most effective way to handle shortest path problems—when possible—is to use **label setting algorithms**.
The second proposed algorithm is therefore the **Reaching Method**, implemented according to its theoretical specifications.

After constructing the graph, a **topological sorting** procedure is executed. This procedure computes the initial indegree of each node, uses a list to track the nodes with indegree equal to zero, and at each iteration removes a node from the list, adds it to the topological order, and updates the indegrees of its adjacent nodes:

```
# Calcolo indegree
inDegree = defaultdict(int)
for i in A:
    for j, _ in A[i]:
        inDegree[j] += 1
    if i not in inDegree:
        inDegree[i] = 0  # anche nodi senza archi entranti

#ordinamento topologico
LIST = [node for node in N if inDegree[node] == 0]
order = []

while LIST:
    i = LIST.pop(0)
    order.append(i)
    for j, _ in A[i]:
        inDegree[j] -= 1
        if inDegree[j] == 0:
            LIST.append(j)
```

Once the topological order is obtained, the algorithm initializes the **distance labels** and **predecessors**:

```
#inizializza
d = {node: float('inf') for node in N}
pred = {node: None for node in N}
d[source] = 0
```

Finally, the main part of the algorithm updates the distance labels of the nodes following the topological order, checking for potential improvements:

```
for i in order:
    for j, cij in A[i]:
        if d[j] > d[i] + cij:
            d[j] = d[i] + cij
            pred[j] = i

return d[sink], pred
```

As in the previous method, the algorithm returns:

- The **optimal distance of the sink node**.
- The **list of predecessors**, which identifies the path found by the algorithm.

3.5. Testing

Before analyzing the performance of the algorithms and models on larger scheduling problems, it was considered appropriate to verify the **correctness** of the implementations on a small-scale dataset:

```
"Tlim": [70],  # Tempo limite per ciascun core
"processi": [
    {"nome": "processo_0", "t": 15, "p": 8},  # Processo 1: tempo 15, priorità 8
    {"nome": "processo_1", "t": 20, "p": 6},  # Processo 2: tempo 20, priorità 5
    {"nome": "processo_2", "t": 10, "p": 2},  # E cosi via
    {"nome": "processo_3", "t": 25, "p": 7},
    {"nome": "processo_4", "t": 30, "p": 6},
    {"nome": "processo_5", "t": 3, "p": 8},
    {"nome": "processo_6", "t": 18, "p": 5},
    {"nome": "processo_7", "t": 40, "p": 10},
    {"nome": "processo_8", "t": 25, "p": 3},
    {"nome": "processo_9", "t": 30, "p": 9}
]
```

First, the standard **scheduling problem** was solved using a **CPLEX model** aimed at maximizing the total priority of the processes executed within the time limit (as in **Process Scheduling Optimization: CPLEX Model and Heuristic Comparison (FCFS & SJF)**). The following results were obtained:

```
objective: 31
status: OPTIMAL_SOLUTION(2)
  x_0_0=1
  x_0_1=1
  x_0_5=1
  x_0_9=1
```

The optimal total priority value is **31**, achieved by selecting processes **0, 1, 5, and 9**.

Subsequently, the **graph** was generated from these data according to the method previously described, and the **Longest Path Problem** was solved using the CPLEX model implemented in the previous chapter. The results were:

```
objective: 31
status: OPTIMAL_SOLUTION(2)
  "x_yes_processo_0_(0, 0)_(1, 15)"=1
  "x_yes_processo_1_(1, 15)_(2, 35)"=1
  "x_no_processo_2_(2, 35)_(3, 35)"=1
  "x_no_processo_3_(3, 35)_(4, 35)"=1
  "x_no_processo_4_(4, 35)_(5, 35)"=1
  "x_yes_processo_5_(5, 35)_(6, 38)"=1
  "x_no_processo_6_(6, 38)_(7, 38)"=1
  "x_no_processo_7_(7, 38)_(8, 38)"=1
  "x_no_processo_8_(8, 38)_(9, 38)"=1
  "x_yes_processo_9_(9, 38)_(10, 68)"=1
  "x_start_s_(0, 0)"=1
  "x_end_68_(10, 68)_t"=1
```

The objective function value again equals **31**, and the selected edges correspond to the same decisions as the original scheduling model—executing processes **0, 1, 5, and 9**.

This confirms that the **conversion from the scheduling problem to the Longest Path Problem** is correct and that the model behaves as expected.

However, as previously mentioned, the **model construction times** are significantly higher for the Longest Path formulation:

Problema	Scheduling originale	LPP
Tempo costruzione	0.002371 s	0.159002 s
Tempo esecuzione	0.016 s	0.016 s

This difference increases with problem size and is clearly due to the **larger number of variables and constraints** required by the layered-graph formulation.
For this reason, it is generally more efficient to **convert the Longest Path Problem into a Shortest Path Problem** and solve it using algorithms specifically designed for that purpose.

When executing the **Modified Label Correcting algorithm** for the Shortest Path Problem derived from the scheduling data, the same results are obtained as with the previous models, confirming the **correctness of the transformations and algorithm**:

```
Priorità totale ottima: 31
Processi selezionati:
 - processo_0 (priorità 8, durata implicita)
 - processo_1 (priorità 6, durata implicita)
 - processo_5 (priorità 8, durata implicita)
 - processo_9 (priorità 9, durata implicita)
```

Likewise, running the **Reaching Method** produces the same results:

```
Priorità totale ottima: 31
Processi selezionati:
 - processo_0 (priorità 8, durata implicita)
 - processo_1 (priorità 6, durata implicita)
 - processo_5 (priorità 8, durata implicita)
 - processo_9 (priorità 9, durata implicita)
```

Regarding **execution times**, no significant advantage of the label setting algorithm is observed at this stage:

Algoritmo	LC Modificato	Reaching Method
Tempo esecuzione	0.000994 s	0.001771 s

However, this difference is expected to become more evident as the **problem size increases**.

4. Results and conclusion

To confirm the findings obtained from the small-scale dataset and to compare the behavior of the algorithms, several tests were conducted on **randomly generated datasets**, created based on the following parameters:

NOME	SEED	PROCESSI	Tlim	Tmax PROCESSO
s1	11	50	100	6
s2	19	100	220	8
s3	23	100	270	9
s4	38	200	500	12
s5	48	200	400	10
s6	59	200	500	7
s7	65	250	500	11
s8	76	250	550	12
s9	88	250	600	12
s10	93	250	700	11

In all cases, both the two **CPLEX models** and the two **Shortest Path algorithms** correctly identified the **optimal solution** of the problem. Therefore, only **execution times** and **model construction times** (for the CPLEX models) will be compared.

As anticipated, solving a **Longest Path Problem** using a **CPLEX model** on this type of layered graph is highly inefficient. In fact, simply increasing the number of processes to **100** causes the model construction time (in seconds) for the LPP to grow dramatically compared to that of the simpler **knapsack-style model**:

DATI	CPLEX SCHEDULING	CPLEX LPP
s1	0.010377	5.659405
s2	0.006395	133.71798

This behavior can be explained by analyzing how the number of variables and constraints changes across the two formulations:

- In the **knapsack-style model**, the number of variables and constraints grows **linearly** with the number of processes ***n***.
- In the **graph-based model** for the Longest Path Problem, however, there are ***n*** layers with $(T_{lim}+1)$ nodes each, plus the source and sink nodes. Consequently, the number of variables (associated with the edges) and the number of constraints increase **linearly with nT**, making the model significantly more complex.

Due to the high **construction time** of the CPLEX model for the LPP, further testing was performed **only with the other algorithms**, leading to the following **execution times (in seconds)**:

DATI	CPLEX SCHEDULING	CPLEX LPP	LABEL CORRECTING	REACHING METHOD
s1	0.030999	0.108999	0.019673	0.020544
s2	0.031	0.75	0.168453	0.130551
s3	0.032		0.247995	0.185238
s4	0.016		1.607334	1.062372
s5	0.016		1.073571	1.025973
s6	0.016		1.25164	0.8631
s7	0.016		2.223915	1.089296
s8	0.016		2.550129	1.287466
s9	0.015		3.201146	1.317433
s10	0.016		3.538658	1.60241

As expected, the **Reaching Method** proves to be **more efficient** than the **Modified Label Correcting algorithm** as the problem size increases.
However, both algorithms remain **significantly less efficient** than solving the original **scheduling problem** directly, when formulated as a **knapsack problem**.

4.1. Conclusion

From the results of the experiments, the following conclusions can be drawn:

- **Process scheduling problems** can be effectively modeled as **Longest Path Problems**, by constructing an acyclic graph that represents all admissible solutions as a set of states and decisions.
- However, solving an **LPP directly with a CPLEX model** is generally inefficient. A better approach is to **invert the sign of the edge weights** and solve the equivalent **Shortest Path Problem** using dedicated algorithms.
- Among the tested methods, the **Reaching Method** proved to be **more efficient** than the **Modified Label Correcting algorithm**. Nonetheless, **label setting algorithms** cannot be applied in every scenario.
- Solving the **process scheduling problem** directly as a **knapsack-style CPLEX model**, without constructing any graph, **remains the most practical and efficient solution**.

Overall, the study highlights the **applicability of Shortest Path algorithms** even to problems that, at first glance, appear **unrelated to graph or network theory**.

5. References

[1] R. K. Ahuja, T. L. Magnanti, J. B. Orlin, Network Flows, Theory, Algorithms and Applications. Upper Saddle River, New Jersey, Prentice-Hall, 1993.