Variants of Multi-Resource Scheduling Problems with Equal Processing Times

Hamed Fahimi, Claude-Guy Quimper

Université Laval, Canada
hamed.fahimi.1@ulaval.ca, claude-guy.quimper@ift.ulaval.ca

Abstract. We tackle the problem of non-preemptive scheduling of a set of tasks of duration p over m machines with given release and deadline times. We present a polynomial time algorithm as a generalization to this problem, when the number of machines fluctuates over time. Further, we consider different objective functions for this problem. We show that if an arbitrary function cost $c_i(t)$ is associated to task i for each time t, minimizing $\sum_{i=1}^{n} c_i(s_i)$ is NP-Hard. Further, we specialize this objective function to the case that it is merely contingent on the time and show that although this case is pseudo-polynomial in time, one can derive polynomial algorithms for the problem, provided the cost function is monotonic or periodic. Finally, as an observation, we mention how polynomial time algorithms can be adapted with the objective of minimizing maximum lateness.

1 Introduction

We explore several variants of the problem of scheduling, without preemption, tasks with equal processing times on multiple machines while respecting release times and deadlines. More formally, we consider n tasks and m identical machines. Task i has a release time r_i and deadline $\bar{d}_i$. All tasks have a processing time p. Without loss of generality, all parameters r_i, $\bar{d}_i$ and p are positive integers. Moreover, we consider the time point $u_i = \bar{d}_i - p + 1$, by starting at which a task i oversteps its deadline. We denote $r_{\min} = \min_i r_i$ the earliest release time and $u_{\max} = \max_i u_i$ the latest value u_i. A solution to the problem is an assignment of the starting times s_i which satisfies the following constraints

$$r_i \leq s_i < u_i \qquad \forall\, i \in \{1, \ldots, n\} \tag{1}$$

$$|\{i : t \leq s_i < t + p\}| \leq m \qquad \forall\, t \in [r_{\min}, u_{\max}) \tag{2}$$

The completion time of a task C_i is equal to $s_i + p$. From (1), we obtain $C_i \leq \bar{d}_i$.

Following the notations of [9], this problem is denoted $Pm \mid r_j; p_j = p; \bar{d}_j \mid \gamma$ where γ is an objective function. The problem is sometimes reformulated by dividing all time points by p, resulting in tasks with unit processing times [13, 14]. However, this formulation does not make the problem easier to solve, as release times and deadlines lose their integrality. Without this integrality, greedy algorithms, commonly used to solve the problem when $p = 1$, become incorrect. Indeed, when the greedy scheduling algorithms choose to start a task i, they assume that no other tasks arrive until i is completed. This assumption does not hold if release times can take any rational value.

We explore several variations of this scheduling problem. Firstly, we solve the problem when the number of machines fluctuates over time. This models situations where there are fewer operating machines during night shifts or when fewer employees can execute tasks during vacation time or

holidays. Then, we consider the problem with different objective functions. For an arbitrary function $c_i(t)$ associated to task i that maps a time point to a cost, we prove that minimizing $\sum_{i=1}^{n} c_i(s_i)$ is NP-Hard. This function is actually very general and can encode multiple well known objective functions. We study the case where all tasks share the same function $c(t)$. This models the situation where the cost of using the resource fluctuates with time. This is the case, for instance, with the price of electricity. Executing any task during peak hours is more expensive than executing the same task during a period when the demand is low. We show that minimizing $\sum_{i=1}^{n} c(t)$ can be done in pseudo-polynomial time and propose improvements when $c(t)$ is monotonic or periodic. The periodicity of the cost function is a realistic assumption as high and low demand periods for electricity have a predictable periodic behavior. Finally, we point out how the problem is solved in polynomial time with the objective of minimizing maximum lateness.

The paper is divided as follows. Section 2 presents a brief survey on existing algorithms related to the scheduling problem, the basic terminology and notations used in this paper, and the objective functions of interest. Section 3 solves the case where the number of machines fluctuates at specific times and shows how to adapt an existing algorithm for this case, while preserving polynomiality. Section 4 shows that minimizing $\sum_{i=1}^{n} c_i(s_i)$ is NP-Hard. Sections 5 and 6 consider a unique cost function $c(t)$ that is either monotonic or periodic and present polynomial time algorithms for these cases. Finally, as an additional remark, we show how to adapt a polynomial time algorithms for minimizing maximum lateness.

2 Literature Review, Framework and Notations

2.1 Related Work

Simons [13] presented an algorithm with time complexity $O(n^3 \log\log(n))$ that solves the scheduling problem. It is reported [9] that it minimizes both the sum of the completion times $\sum_j C_j$, and the latest completion time $C_{\max}$ (also called the makespan). Simons and Warmth [14] further improved the algorithm complexity to $O(mn^2)$. Dürr and Hurand [4] reduced the problem to a shortest path in a digraph and designed an algorithm in $O(n^4)$. This led López-Ortiz and Quimper [11] to introduce the idea of the scheduling graph. By computing the shortest path in this graph, one obtains a schedule that minimizes both $\sum_j C_j$ and $C_{\max}$. Their algorithm runs in $O(n^2 \min(1, p/m))$.

There exist more efficient algorithms for special cases. For instance, when there is only one machine ($m = 1$) and unit processing times ($p = 1$), the problem is equivalent to finding a matching in a convex bipartite graph. Lipski and Preparata [10] present an algorithm running in $O(n\alpha(n))$ where α is the inverse of Ackermann's function. Gabow and Tarjan [5] reduce this complexity to $O(n)$ by using a restricted version of the union-find data structure.

Baptiste proves that in general, if the objective function can be expressed as the sum of n functions f_i of the completion time C_i of each task i, where f_i's are non-decreasing and for any pair of jobs (i, j) the function $f_i - f_j$ is monotonous, the problem can be solved in polynomial time. Note that the assumption holds for several objectives, such as the weight sum of completion times $\sum w_i C_i$. A variant of the problem exists when the tasks are allowed to miss their deadlines at the cost of a penalty. Let $L_j = C_j - d_j$ be the *lateness* of a task j. The problem of minimizing the maximum lateness $L_{\max} = \max_j L_j$ (denoted $P \mid r_i, p_i = p \mid L_{max}$) is polynomial [15] and the special case for one machine and unit processing times (denoted $1 \mid r_j, p_j = p \mid L_{max}$) is solvable in $O(n \log n)$ [8].

Möhring et al. [12] study the case where no release times or deadlines are provided and the processing times are not all equal. For the case that their problem is not resource-constrained, they

consider the objective of minimizing costs per task and per time, as it is considered in this paper. They establish a connection between a minimum-cut in an appropriately defined directed graph and propose a mathematical programming approach to compute both lower bounds and feasible solutions. The minimum-cut problem is the dual of the maximum flow problem that will be used in this paper.

Bansal and Pruhs [3] consider the problem with preemptive tasks, a single machine, no deadlines, and distinct processing times. Tasks incur a cost depending on their completion time. They introduce an approximation for the general case as well as an improved approximation algorithm for the case that all release times are identical.

2.2 Objective Functions

Numerous objective functions can be optimized in a scheduling problem. We first consider *minimizing costs per task and per time*, in which case executing a task i at time t costs $c(i,t)$ and we aim to minimize the sum of costs, i.e. $\sum_{i,t} c(i,s_i)$. Such an objective function depends on the release time of the task. In the industry, that can be used to model a cost that increases as the execution of a task is delayed. For instance, $c(i,t) = t - r_i$. Then, we consider *minimizing task costs per time*, in which case executing any task at time t costs $c(t)$ and we want to minimize $\sum_i c(s_i)$. An alternative common objective function is to minimize the sum of the completion times. In the context where the tasks have equal processing times, a solution that minimizes the sum of the completion times necessarily minimizes the sum of the starting time. We consider these two objectives equivalent. Finally, we consider minimizing the maximum lateness $L_{\max} = \max_i L_i$.

2.3 Network Flows

Consider a digraph $\overrightarrow{N} = (V,E)$ where each arc $(i,j) \in E$ has a flow capacity u_{ij} and a flow cost c_{ij}. There is one node $s \in V$ called the source and one node $t \in V$ called the sink. A *flow* is a vector that maps each edge $(i,j) \in E$ to a value x_{ij} such that the following constraints are satisfied.

$$0 \leq x_{ij} \leq u_{ij} \tag{3}$$

$$\sum_{j \in V} x_{ji} - \sum_{j \in V} x_{ij} = 0 \qquad \forall i \in V \setminus \{s,t\} \tag{4}$$

The min-cost flow satisfies the constraints while minimizing $\sum_{(i,j) \in E} c_{ij} x_{ij}$.

A matrix with entries in $\{-1,0,1\}$ which has precisely one 1 and one -1 per column is called a *network matrix*. If A is a network matrix, the following linear program identifies a flow.

$$\text{Maximize } c^T x, \text{ subject to } \begin{cases} Ax = b \\ x \geq 0 \end{cases} \tag{5}$$

There is one node for each row of the matrix in addition to a source node s and a sink node t. Each column in the matrix corresponds to an edge $(i,j) \in E$ where i is the node whose row is set to 1 and j is the node whose column is set to -1. If $b_i > 0$ we add the edge (i,t) of capacity b_i and if $b_i < 0$ we add the edge (s,i) of capacity $-b_i$ [16].

The residual network with respect to a given flow x is formed with the same nodes V as the original network. However, for each edge (i,j) such that $x_{ij} < u_{ij}$, there is an edge (i,j) in the

residual network of cost c_{ij} and residual capacity $u_{ij} - x_{ij}$. For each edge (i, j) such that $x_{ij} > 0$, there is an edge (j, i) in the residual network of cost $-c_{ij}$ and residual capacity x_{ij}.

To our knowledge, the *successive shortest path algorithm* is the state of the art, for this particular structure of the network, to solve the min-cost flow problem. This algorithm successively augments the flow values y_{ij} of the edges along the shortest path connecting the source to the sink in the residual graph. Let $N = \max_{(i,j)\in E} |c_{ij}|$ be the greatest absolute cost and $U = \max_{i\in V} b_i$ be the largest value in the vector b. To compute the shortest path, one can use Goldberg's algorithm [6] with a time complexity of $O(|E|\sqrt{|V|}\log N)$. Since at most $|V|U$ shortest path computations are required, this leads to a time complexity of $O(|V|^{1.5}|E|\log(N)U)$.

2.4 Scheduling Graph

López-Ortiz and Quimper [11] introduced the *scheduling graph* which holds important properties. For instance, it allows to decide whether an instance is feasible, i.e. whether there exists at least one solution. The graph is based on the assumption that it is sufficient to determine how many tasks start at a given time. If one knows that there are h_t tasks starting at time t, it is possible to determine which tasks start at time t by computing a matching in a bipartite convex graph (see [11]).

The scheduling problem can be written as a satisfaction problem where the constraints are uniquely posted on the variables h_t. As a first constraint, we force the number of tasks starting at time t to be non-negative.

$$\forall\ r_{\min} \le t \le u_{\max} - 1 \qquad\qquad h_t \ge 0 \tag{6}$$

At most m tasks (n tasks) can start within any window of size p (size $u_{\max} - r_{\min}$).

$$\forall\ r_{\min} \le t \le u_{\max} - p \qquad \sum_{j=t}^{t+p-1} h_j \le m, \qquad \sum_{j=r_{\min}}^{u_{\max}-1} h_j \le n \tag{7}$$

Given two arbitrary (possibly identical) tasks i and j, the set $K_{ij} = \{k : r_i \le r_k \wedge u_k \le u_j\}$ denotes the jobs that must start in the interval $[r_i, u_j)$. Hence,

$$\forall\ i, j \in \{1, \ldots, n\} \qquad \sum_{t=r_i}^{u_j-1} h_t \ge |K_{ij}| \tag{8}$$

Some objective functions, such as minimizing the sum of the starting times, can also be written with the variables h_t.

$$\min \sum_{t=r_{\min}}^{u_{\max}-1} t \cdot h_t \tag{9}$$

To simplify the inequalities (6) to (8), we proceed to a change of variables. Let $x_t = \sum_{i=r_{\min}}^{t-1} h_i$, for $r_{\min} \le t \le u_{\max}$, be the number of tasks starting to execute before time t. Therefore, the

problem can be rewritten as follows.

$$\forall\ r_{\min} \leq t \leq u_{\max} - p \qquad x_{t+p} - x_t \leq m, \qquad x_{u_{\max}} - x_{r_{\min}} \leq n \tag{10}$$

$$\forall\ r_{\min} \leq t \leq u_{\max} - 1 \qquad x_t - x_{t+1} \leq 0 \tag{11}$$

$$\forall\ r_i + 1 \leq u_j \qquad x_{r_i} - x_{u_j} \leq -|K_{ij}| \tag{12}$$

These inequalities form a system of difference constraints which can be solved by computing shortest paths in what is call the *scheduling graph* [11]. In this graph, there is a node for each time point t, $r_{\min} \leq t \leq u_{\max}$ and an edge of weight k_{pq}, connecting the node q to the node p for each inequality of the form $x_p - x_q \leq k_{pq}$.

The *scheduling graph* has for vertices the nodes $V = \{r_{\min}, \ldots, u_{\max}\}$ and for edges $E = E_f \cup E_b \cup E_n$ where $E_f = \{(t, t+p) : r_{\min} \leq t \leq u_{\max} - p\} \cup \{(r_{\min}, u_{\max})\}$ is the set of *forward edges* (from inequalities (10)), $E_b = \{(u_j, r_i) : r_i < u_j\}$ is the set of *backward edges* (from inequality (12)), and $E_n = \{(t+1, t) : r_{\min} \leq t < u_{\max}\}$ is the set of *null edges* (from inequality (11)). The following weight function maps every edge $(a, b) \in E$ to a weight:

$$w(a,b) = \begin{cases} m & \text{if } a + p = b \\ n & \text{if } a = r_{\min} \wedge b = u_{\max} \\ -|\{k : b \leq r_k \wedge u_k \leq a\}| & \text{if } a > b \end{cases} \tag{13}$$

Theorem 1 shows how to compute a feasible schedule.

Theorem 1 (López-Ortiz and Quimper [11]). *Let $\delta(a, b)$ be the shortest distance between node a and node b in the scheduling graph. The assignment $x_t = n + \delta(u_{max}, t)$ is a solution to the inequalities* (10) *to* (12) *that minimizes the sum of the completion times.*

The scheduling problem has a solution if and only if the scheduling graph has no negative cycles. An adaptation [11] of the Bellman-Ford algorithm finds a schedule with time complexity $O(\min(1, \frac{p}{m})n^2)$, which is sub-quadratic when $p < m$ and quadratic otherwise.

In the next sections, we adapt the scheduling graph to solve variations of the problem.

3 Variety of machines

Consider the problem where the number of machines fluctuates over time. Let $T = [(t_0, m_0), \ldots, (t_{|T|-1}, m_{|T|-1})]$ be a sequence where t_i's are the time points at which the fluctuations occur and they are sorted in chronological order and m_i machines are available within the time interval $[t_i, t_{i+1})$. This time interval is the union of a (possibly empty) interval and an open-interval: $[t_i, t_{i+1}) = [t_i, t_{i+1} - p] \cup (t_{i+1} - p, t_{i+1})$. A task starting in $[t_i, t_{i+1} - p]$ is guaranteed to have access to m_i machines throughout its execution, whereas a task starting in $(t_{i+1} - p, t_{i+1})$ encounters the fluctuation of the number of machines before completion. Therefore, no more than $\min(m_i, m_{i+1})$ tasks can start in the interval $(t_{i+1} - p, t_{i+1})$. In general, a task can encounter multiple fluctuations of the number of machines throughout its execution. Let $\alpha(t) = \max\{t_j \in T \mid t_j \leq t\}$ be the last time the number of machines fluctuates before time t. At most $M(t)$ tasks can start at time t.

$$M(t) = \min\{m_i \mid t_i \in [\alpha(t), t + p)\}. \tag{14}$$

From (14), we conclude that no more than $\max_{t' \in [t,t+p)} M(t')$ tasks can start in the interval $[t, t+p)$. Accordingly, one can rewrite the first inequality of the constraints (10)

$$x_{t+p} - x_t \leq \max_{t \leq t' < t+p} M(t') \tag{15}$$

and update the weight function of the scheduling graph.

$$w(a,b) = \begin{cases} \max_{a \leq t' < a+p} M(t') & \text{if } a + p = b \\ n & \text{if } a = r_{\min} \wedge b = u_{\max} \\ -|\{k : b \leq r_k \wedge u_k \leq a\}| & \text{if } a \geq b \end{cases} \tag{16}$$

It remains to show how the algorithm presented in [11] can be adapted to take into account the fluctuating number of machines. This algorithm maintains a vector $d^{-1}[0..n]$ such that $d^{-1}[i]$ is the latest time point reachable at distance $-i$ from the node $u_{\max}$. In other words, all nodes whose label is a time point in the semi-open interval $(d^{-1}[i+1], d^{-1}[i]]$ are reachable at distance $-i$ from node $u_{\max}$. Let a be a node in $(d^{-1}[i+1], d^{-1}[i]]$ and consider the edge $(a, a+p)$ of weight $w(a, a+p)$. Upon processing this edge, the algorithm updates the vector by setting $d^{-1}[i - w(a,b)] \leftarrow \max(d^{-1}[i - w(a,b)], b)$, i.e. the rightmost node accessible at distance $-i + w(a,b)$ is either the one already found, or the node $a+p$ that is reachable through the path to a of distance $-i$ followed by the edge $(a, a+p)$ of distance $w(a, a+p)$.

To efficiently proceed to this update, the algorithm evaluates the function $w(a, a+p)$ in two steps. The first step transforms T in a sequence $T' = [(t'_0, m'_0), (t'_1, m'_1), \ldots]$ such that $M(t) = m'_i$ for every $t \in [t'_i, t'_{i+1})$. The second step transforms the sequence T' into a sequence $T'' = [(t''_0, m''_0), (t''_1, m''_1), \ldots]$ such that $w(t, t+p) = m''_i$ for all $t \in [t''_i, t''_{i+1})$. Interestingly, both steps execute the same algorithm.

To build the sequence T', one needs to iterate over the sequence T and find out, for every time window $[t, t+p)$, the minimum number of available machines inside that time window. If a sequence of consecutive windows such as $[t, t+p), [t+1, t+p+1), [t+2, t+p+2), \ldots$ have the same minimum number of available machines, then only the result of the first window is reported. This is a variation of the *minimum on a sliding window problem* [7] where an instance is given by an array of numbers $A[1..n]$ and a window length p. The output is a vector $B[1..n-p+1]$ such that $B_i = \min\{A_i, A_{i+1}, \ldots, A_{i+p-1}\}$. The algorithm that solves the minimum on a sliding window problem can be slightly adapted. Rather than taking as input the vector A that contains, in our case, many repetitions of values, it can simply take as input a list of pairs like the vector T and T' which indicate the value in the vector and until which index this value is repeated. The same compression technique applies for the output vector. This adaptation can be done while preserving the linear running time complexity of the algorithm.

Once computed, the sequence T' can be used as input to the *maximum on a sliding window problem* to produce the final sequence T''. Finally, the algorithm 1 simultaneously iterates over the sequence T'' and the vector d^{-1} to relax the edges in $O(|T| + n)$ time. Since relaxing forward edges occurs at most $O(\min(1, \frac{p}{m})n)$ times [11], the overall complexity to schedule the tasks is $O(\min(1, \frac{p}{m})(|T| + n)n)$.

4 General Objective Function

We prove that minimizing costs per task and per time, i.e. $\sum_{i,t} c_i(s_i)$ for arbitrary functions $c_i(t)$ is NP-Hard. We proceed with a reduction from the InterDistance constraint [2]. The predicate

Algorithm 1: RelaxForwardEdges($[(t''_1, m''_1), \ldots, (t''_{|T''|}, m''_{|T''|})], d^{-1}[0..n], p$)

```
t ← r_min, i ← n, j ← 0
while i > 0 ∨ j < |T''| do
    if i − m''_j > 0 then d^{-1}[i − m''_j] ← max(d^{-1}[i − m''_j], t + p)
    if j = |T''| ∨ i ≥ 0 ∧ d^{-1}[i − 1] < m''_{j+1} then
        i ← i + 1
        t ← d^{-1}[i]
    else
        j ← j + 1
        t ← t''_j
```

InterDistance($[X_1, \ldots, X_n], p$) is true if and only if $|X_i - X_j| \geq p$ holds whenever $i \neq j$. Let $S_1, \ldots, S_n$ be n sets of integers. Deciding whether there exists an assignment for the variables $X_1, \ldots, X_n$ such that $X_i \in S_i$ and InterDistance($[X_1, \ldots, X_n], p$) hold is NP-Complete [2]. We create one task per variable X_i with release time $r_i = \min(S_i)$, latest starting time $u_i = \max(S_i)$, processing time p, and a cost function $c_i(t)$ equal to 0 if $t \in S_i$ and 1 otherwise. There exists a schedule with objective value $\sum_{i,t} c_i(s_i) = 0$ iff there exists an assignment with $X_i \in S_i$ that satisfies the predicate InterDistance, hence minimizing $\sum_{i,t} c_i(s_i)$ is NP-Hard.

The NP-hardness of this problem motivates the idea of studying specializations of this objective function in order to seek if polynomial time algorithms can be derived.

5 Monotonic Objective Function

Let $c(t) : \mathbb{Z} \to \mathbb{Z}$ be an increasing function, i.e. $c(t) + 1 \leq c(t + 1)$ for any t. We prove that a schedule that minimizes $\sum_i s_i$ also minimizes $\sum_i c(s_i)$. Theorem 1 shows how to obtain a solution that minimizes $\sum_i s_i$. Lemma 1 shows that this solution also minimizes other objective functions. Recall that h_t is the number of tasks starting at time t.

Lemma 1. *The schedule obtained with Theorem 1 minimizes $\sum_{a=t}^{u_{\max}-1} h_a$ for any time t.*

Proof. Let $(a_1, a_2), (a_2, a_3), \ldots, (a_{k-1}, a_k)$, with $a_1 = u_{\max}$ and $a_k = t$, be the edges on the shortest path from $u_{\max}$ to t in the scheduling graph. By substituting the inequalities (10) to (12), we obtain $\delta(u_{\max}, t) = \sum_{i=1}^{k-1} w(a_i, a_{i+1}) \geq \sum_{i=1}^{k-1} (x_{a_{i+1}} - x_{a_i}) = x_t - x_{u_{\max}}$. This shows that the difference $x_t - x_{u_{\max}}$ is at most $\delta(u_{\max}, t)$ for any schedule. It turns out that by setting $x_t = n + \delta(u_{\max}, t)$, the difference $x_t - x_{u_{\max}} = \delta(u_{\max}, t) - \delta(u_{\max}, u_{\max}) = \delta(u_{\max}, t)$ reaches its maximum and therefore, $x_{u_{\max}} - x_t = \sum_{a=t}^{u_{\max}-1} h_a$ is maximized. □

Theorem 2. *The schedule of Theorem 1 minimizes $\sum_{i=1}^{n} c(s_i)$ for any increasing function $c(t)$.*

Proof. Consider the following functions that differ by their parameter a.

$$c_a(t) = \begin{cases} c(t) & \text{if } t < a \\ c(a) + t - a & \text{otherwise} \end{cases} \tag{17}$$

The function $c_a(t)$ is identical to $c(t)$ up to point a and then increases with a slope of one. As a base case of an induction, the schedule described in Theorem 1 minimizes $\sum_{i=1}^{n} s_i$ and therefore minimizes $\sum_{i=1}^{n} c_{r_{\min}}(s_i)$. Suppose that the algorithm minimizes $\sum_{i=1}^{n} c_a(s_i)$, we prove that it also minimizes $\sum_{i=1}^{n} c_{a+1}(s_i)$. Consider the function

$$\Delta_a(t) = \begin{cases} 0 & \text{if } t \leq a \\ c(a+1) - c(a) - 1 & \text{otherwise} \end{cases} \tag{18}$$

and note that $c_a(t) + \Delta_a(t) = c_{a+1}(t)$. For all t, since $c(t+1) - c(t) \geq 1$, we have $\Delta_a(t) \geq 0$.

If $c(a+1) - c(a) = 1$ then $\Delta_a(t) = 0$ for all t and therefore $c_a(t) = c_{a+1}(t)$. Since the algorithm returns a solution that minimizes $\sum_{i=1}^{n} c_a(s_i)$, it also minimizes $\sum_{i=1}^{n} c_{a+1}(s_i)$.

If $c(a+1) - c(a) > 1$, a schedule minimizes the function $\sum_{i=1}^{n} \Delta_a(s_i)$ if and only if it minimizes the number of tasks starting after time a. From Lemma 1, the schedule described in Theorem 1 achieves this. Consequently, the algorithm minimizes $\sum_{i=1}^{n} c_a(s_i)$, it minimizes $\sum_{i=1}^{n} \Delta_a(s_i)$, and therefore, it minimizes $\sum_{i=1}^{n} c_a(s_i) + \sum_{i=1}^{n} \Delta_a(s_i) = \sum_{i=1}^{n} c_{a+1}(s_i)$.

By induction, the algorithm minimizes $\sum_{i=1}^{n} c_\infty(s_i) = \sum_{i=1}^{n} c(s_i)$. □

If the cost $c(t)$ function is decreasing, i.e. $c(t) - 1 \geq c(t+1)$, it is possible to minimize $\sum_{i=1}^{n} c(t)$ by solving a transformed instance. For each task i in the original problem, one creates a task i with release time $r'_i = -u_i$ and latest starting time $u'_i = -r_i$. The objective function is set to $c'(t) = -c(t)$ which is an increasing function. From a solution s'_i that minimizes $\sum_{i=1}^{n} c'(s'_i)$, one retrieves the original solution by letting $s_i = -s'_i$.

6 Periodic Objective Function

6.1 Scheduling problem as a network flow

Theorem 1 shows that computing the shortest paths in the scheduling graph can minimize the sum of the completion times. We show that computing, in pseudo-polynomial time, a flow in the scheduling graph can minimize $\sum_{i=1}^{n} c(s_i)$ for an arbitrary function $c(t)$.

The objective function (9) can be modified to take into account the function c. We therefore minimize $\sum_{t=r_{\min}}^{u_{\max}-1} c(t)h_t$. After proceeding to the change of variables $x_t = \sum_{i=r_{\min}}^{t-1} h_i$, we obtain $\sum_{t=r_{\min}}^{u_{\max}-1} c(t)(x_{t+1} - x_t)$ which is equivalent to

$$\text{maximize } c(r_{\min})x_{r_{\min}} - \sum_{t=r_{\min}+1}^{u_{\max}-1} \left(c(t) - c(t-1)\right) x_t - c(u_{\max}-1)x_{u_{\max}}$$

We use this new objective function with the original constraints of the problem given by equations (10) to (12). This results in a linear program of the form $\max\{\boldsymbol{c}^T x \mid A\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{x} \lneqq 0\}$ which has for dual $\min\{\boldsymbol{b}^T y \mid A^T \boldsymbol{y} = \boldsymbol{c}, y \geq 0\}$. Note that every row of matrix A has exactly one occurrence of value 1, one occurrence of the value -1, and all other values are null. Consequently, A^T is a *network matrix* and the dual problem $\min\{\boldsymbol{b}^T y \mid A^T \boldsymbol{y} = \boldsymbol{c}, y \geq 0\}$ is a min-cost flow problem.

Following Section 2.3, we reconstruct the graph associated to this network flow which yields the scheduling graph augmented with a source node and a sink node. An edge of capacity $c(r_{\min})$ connects the node $r_{\min}$ to the sink. An edge of capacity $c(u_{\max} - 1)$ connects the source node to the node $u_{\max}$. For the nodes t such that $r_{\min} < t < u_{\max}$, an edge of capacity $c(t-1) - c(t)$ connects

i	r_i	u_i
1	4	8
2	1	4
3	1	6
4	1	9
5	1	6

$m = 2$
$p = 2$
$c(t) = t \bmod 3$

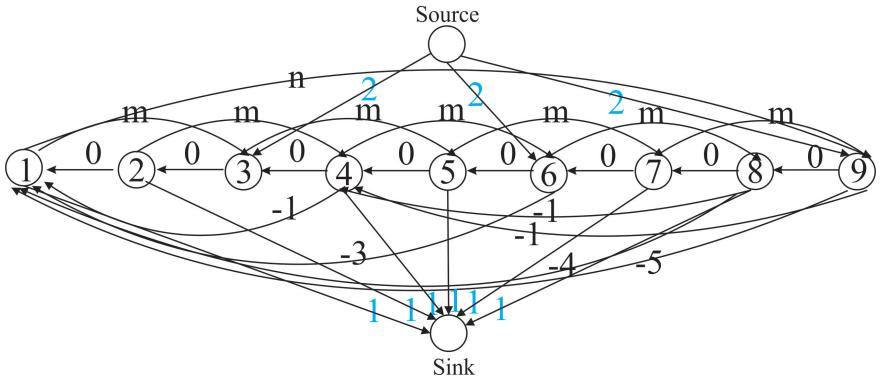

Fig. 1. A network flow with 5 tasks. The cost on the forward, backward, and null edges are written in black. These edges have unlimited capacities. The capacities of the nodes from the source and to the sinks are written in blue. These edges have a null cost.

the source node to node t whenever $c(t-1) > c(t)$ and an edge of capacity $c(t)-c(t-1)$ connects the node t to the sink node whenever $c(t-1) < c(t)$. All other edges in the graph (forward, backward, and null edges) have an infinite capacity. Figure 1 illustrates an example of such a graph.

The computation of a min-cost flow gives rise to a solution for the dual problem. To convert the solution of the dual to a solution for the primal (i.e. an assignment of the variables x_t), one needs to apply a well known principle in network flow theory [1]. Let $\delta(a, b)$ be the shortest distance from node a to node b in the *residual graph*. The assignment $x_t = \delta(u_{\max}, t)$ is an optimal solution of the primal. The variable x_t is often called *node potential* in network theory.

Consider a network flow of $|V|$ nodes, $|E|$ edges, a maximal capacity of U, and a maximum absolute cost of N. The successive shortest path algorithm computes a min-cost flow with $O(|V|U)$ computations of a shortest path that each executes in $O(|E|\sqrt{|V|}\log N)$ time using Goldberg's algorithm [6]. Let $\Delta c = \max_t |c(t) - c(t-1)|$ be the maximum cost function fluctuation and $H = u_{\max} - r_{\min}$ be the horizon. In the scheduling graph, we have $|V| \in O(H)$, $|E| \in O(H + n^2)$, $N \in O(n)$, and $U = \Delta c$. Therefore, the overall running time complexity to find a schedule is $O((H - p + n^2)(H)^{3/2}\Delta c \log n)$.

6.2 Periodic objective function formulated as a network flow

In many occasions, one encounters the problem of minimizing $\sum_{i=1}^{n} c(s_i)$ where $c(s_i)$ is a periodic function, i.e. a function where $c(t) = c(t + W)$ for a period W. Moreover, within a period, the function is increasing. An example of such a function is the function $c(t) = t \bmod 7$. If all time points correspond to a day, the objective function ensures that all tasks are executed at their earliest time in a week. In other words, it is better to wait for Monday to start a task rather than executing this task over the weekend. In such a situation, it is possible to obtain a more efficient time complexity than the algorithm presented in the previous section.

Without loss of generality, we assume that the periods start on times kW for $k \in \mathbb{N}$ which implies that the function $c(t)$ is only decreasing between $c(kW - 1)$ and $c(kW)$ for some $k \in \mathbb{N}$. In the network flow from Section 6.1, only the time nodes kW have an incoming edge from the source.

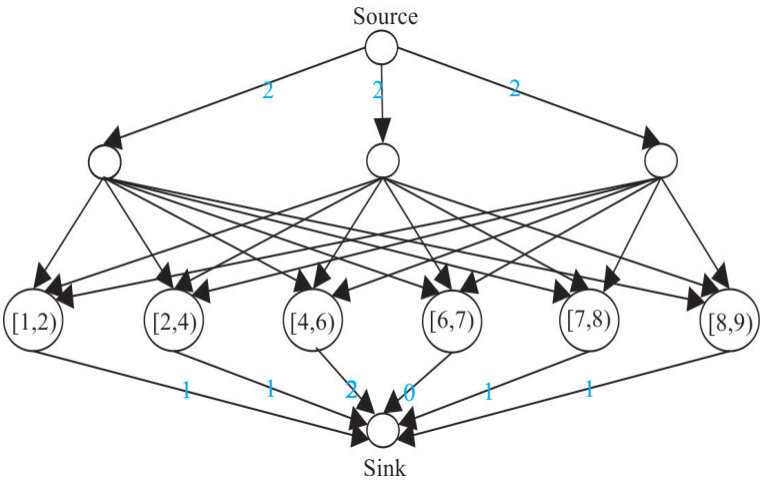

Fig. 2. The compressed version of the graph on Figure 1.

We use the algorithm from [11] to compute the shortest distance from every node kW to all other nodes. Thanks to the null edges, distances can only increase in time, i.e. $\delta(kW, t) \leq \delta(kW, t+1)$, and because of the edge $(r_{\min}, u_{\max})$ of cost n and the nonexistence of negative cycles, all distances lie between $-n$ and n. Therefore, the algorithm outputs a list of (possibly empty) time intervals $[a^k_{-n}, b^k_{-n}), [a^k_{-n+1}, b^k_{-n+1}), \ldots, [a^k_n, b^k_n)$ where for any time $t \in [a^k_d, b^k_d)$, $\delta(kW, t) = d$. The min-cost flow necessarily pushes the flow along these shortest paths. We simply need to identify which shortest paths the flow follows.

There are $c(kW-1) - c(kW)$ units of flow that must circulate from node kW and $c(t) - c(t-1)$ units of flows that must arrive to node t, for any t that is not a multiple of W. In order to create a smaller graph with fewer nodes, we aggregate time intervals where time points share common properties. We consider the sorted set S of time points a^k_i and b^k_i. Let t_1 and t_2 be two consecutive time points in this set. All time points in the interval $[t_1, t_2)$ are at equal distance from the node kW, for any $k \in \mathbb{N}$. The amount of units of flow that must reach the sink from the nodes in $[t_1, t_2)$ is given by

$$\sum_{j=t_1}^{t_2-1} \max(c(j) - c(j-1), 0) = \\ c(t_2 - 1) - c(t_1 - 1) + \left(\left\lfloor \frac{t_2 - 1}{W} \right\rfloor - \left\lceil \frac{t_1}{W} \right\rceil + 1 \right) (c(W-1) - c(0)) \tag{19}$$

Consequently, we create a graph, called the *compressed graph*, with one source and one sink node. There is one node for each time point kW for $\frac{r_{\min}}{W} \leq k \leq \frac{u_{\max}}{W}$. There is an edge between the source node and a node kW with capacity $c(kW-1) - c(kW)$. For any two consecutive time points t_1, t_2 in S there is a time interval node $[t_1, t_2)$. An edge whose capacity is given by equation (19) connects the interval node $[t_1, t_2)$ to the sink. Finally, a node kW is connected to an interval node $[t_1, t_2)$ with an edge of infinite capacity and a cost of $\delta(kW, t_1)$. Figure 2 shows the compressed version of the graph on figure 1.

Computing a min-cost flow in this network simulates the flow in the scheduling graph. Indeed, a flow going through an edge $(kW, [t_1, t_2))$ in the compressed graph is equivalent, in the scheduling graph, to a flow leaving the source node, going to the node kW, going along the shortest path from node kW to a time node $t \in [t_1, t_2)$, and reaching the sink.

Theorem 3. *To every min-cost flow in the compressed graph corresponds a min-cost flow in the scheduling graph.*

Proof. Let G be the scheduling graph and G' be the compressed graph. Let Y' denote a min-cost flow in G'. We show how to obtain a min-cost flow Y in G whose cost is the same as the cost of Y'.

Consider an edge $e_j = (kW, [t_1, t_2))$ in G' which conveys a positive amount of flow, say f. In the scheduling graph G, it is possible to push f units of flow along the shortest paths from kW to the nodes within the interval $[t_1, t_2)$. It suffices to see how one can retrieve Y from Y', presuming it is initially null. This is done by considering all incoming flows to $[t_1, t_2)$ and manage to spread them over the edges of G. We start with the node t_1 and consider the shortest path P from kW to t_1 in G. The amount of flow that can be incremented is the minimum between f and the amount of flow that t_1 can receive. Then, we increment the amount of flow on the extended path in G, which connects the source to P and connects P to the sink.

If the capacity of t_1 is reached, we decrement f by the amount of flow which was consumed and we move to the next node in the interval. Now, there remains f units of flow for the nodes within the interval $[t_1 + 1, t_2)$. By repeating the same instruction for the rest of the nodes in $[t_1, t_2)$ and for every edge in G' that carries a positive amount of flow, we obtain the flow Y. It is guaranteed that all the flow can be pushed to the nodes in $[t_1, t_2)$ as the sum of the capacities of the edges that connect a node in $[t_1, t_2)$ to the sink in G is equal to the capacity of the edges between $[t_1, t_2)$ and the sink in the G'.

Furthermore, the flow Y satisfies the capacities since the capacities on the edges adjacent to the source in G are the same as those in G'. Moreover, the capacities were respected for the nodes adjacent to the sink. The cost of Y is the same as Y' since the paths on which the flow is pushed in Y have the same cost as the edges in the compressed graph.

We prove that Y is optimal, i.e. it is a min-cost flow. Each unit of flow in a min-cost flow in G leaves from the source to a node kW and necessarily traverses along the shortest path going to a node t and then reaches the sink. Note that the edges on the shortest path have unlimited capacities. The question is therefore on which shortest path does each unit of flow travel? This is exactly the question that the flow in the compressed graph answers. □

In what follows, RG and RG', stand for the residual graph of the scheduling graph G and the residual compressed graph G'.

Lemma 2. *Let t be a node in the residual scheduling graph RG and $[t_i, t_{i+1})$, such that $t_i \leq t < t_{i+1}$, be a node in the residual compressed scheduling graph. The distance between node kW and t in RG is equal to the distance between kW and $[t_i, t_{i+1})$ in RG'.*

Proof. We show that for any path P' in the residual compressed graph RG', there is a path P in the residual graph RG that has the same cost. From Lemma 3, we know that for a flow in the compressed graph G', there is an equivalent flow in the original graph G. Consider a path P' from a node kW to an interval node $[t_i, t_{i+1})$. By construction of the compressed graph, for each edge of this path corresponds a path of equal cost in the residual graph RG'. Consequently, there is a path in G' that goes from node kW to any node $t \in [t_i, t_{i+1})$ with the same cost as the path going from kW to $[t_i, t_{i+1})$ in G.

Consider a path P in the residual graph RG going from a node k_1W to a node t. Suppose that this path contains exactly one edge in RG that is not in G. We denote this edge (a, b) and the path P can be decomposed as follows: $k_1W \rightsquigarrow a \to b \rightsquigarrow t$. The edge (a, b) appears in the residual graph RG because there is a positive amount of flow circulating on a shortest path $S : k_2W \rightsquigarrow b \to a \rightsquigarrow u$ to which the reversed edge (b, a) belongs. Let Q be the following path in the residual graph RG: $k_1W \rightsquigarrow u \rightsquigarrow a \to b \rightsquigarrow k_2W \rightsquigarrow t$. Let l be the function that evaluates the cost of a path. We prove that Q has a cost that is no more than P and that it has an equivalent in the residual compressed graph RG'.

$$\begin{aligned} l(Q) &= l(k_1W \rightsquigarrow u \rightsquigarrow a \to b \rightsquigarrow k_2W \rightsquigarrow t) \\ &\leq l(k_1W \rightsquigarrow a \rightsquigarrow u \rightsquigarrow a \to b \rightsquigarrow k_2W \rightsquigarrow t) \end{aligned}$$

In the residual graph, the paths $a \rightsquigarrow u$ and $u \rightsquigarrow a$ have opposite costs, hence $l(a \rightsquigarrow u \rightsquigarrow a) = 0$.

$$\begin{aligned} &= l(k_1W \rightsquigarrow a \to b \rightsquigarrow k_2W \rightsquigarrow t) \\ &\leq l(k_1W \rightsquigarrow a \to b \rightsquigarrow k_2W \rightsquigarrow b \rightsquigarrow t) \end{aligned}$$

In the residual graph, the paths $b \rightsquigarrow k_2W$ and $k_2W \rightsquigarrow b$ have opposite costs, hence $l(b \rightsquigarrow k_2W \rightsquigarrow b)$.

$$\begin{aligned} &= l(k_1W \rightsquigarrow a \to b \rightsquigarrow t) \\ &= l(P) \end{aligned}$$

The path Q has an equivalent in the residual compressed graph RG'. Indeed, the sub-paths $k_1W \rightsquigarrow u$ and $k_2W \rightsquigarrow t$ are edges in RG' whose cost is given by the shortest paths in G. The path $u \rightsquigarrow a \to b \rightsquigarrow k_2W$ is the reverse of path S. Since S is an edge in G' and there is a flow circulating on S, the reverse of S also appears in RG'. Consequently, the path P can be transformed into path Q that has an equivalent in the compressed residual graph. If P contains more than one edge that belongs to RG but not G, then the transformation can be applied multiple times.

Since a path in RG has an equivalent path whose cost is not greater in RG' and vice-versa, we conclude that a node kW is at equal distance from all the other nodes in either graph. □

Notice that the above lemma implies that after computing the min-cost flow in the compressed graph, one sets the value for x_t to the shortest distance between an arbitrary but fixed node kW to the interval node that contains t.

Let $H = u_{\max} - r_{\min}$ be the horizon, we need $\Theta(\frac{H}{W})$ calls to the algorithm in [11] to build the compressed graph in $O(\frac{H}{W} n^2 \min(1, \frac{p}{m}))$ time. As in Section 6.1, the successive shortest paths technique, with Goldberg's algorithm [6], computes the maximum flow. The compressed graph has $|V| \in O(\frac{H}{W} + n^2)$ nodes, $|E| \in O(\frac{H}{W} n^2)$ edges, a maximum absolute cost of $N \in O(n)$, and a maximum capacity of $U = \Delta c = C(W-1) - c(0)$. Computing the values for x_t requires an additional execution of Goldberg's algorithm on the compressed graph. The final running time complexity is $O\left(\left(\left(\frac{H}{W}\right)^{2.5} + n^5\right) \Delta c \log(n)\right)$ which is faster than the algorithm presented in the previous sections when the number of periods is small, i.e. when $\frac{H}{W}$ is bounded. In practice, there are fewer periods than tasks: $\frac{H}{W} < n$.

7 Additional Remark

Consider the case where tasks have due dates d_i and deadlines $\bar{d}_i$. One wants to minimize the maximum lateness $L_{\max} = \max_i \max(C_i - d_i, 0)$ while ensuring that tasks complete before their deadlines. To test whether there exists a schedule with maximum lateness L, one changes the deadline of all task i for $\min(\bar{d}_i, d_i + L)$. If there exists a valid schedule with this modification, then there exists a schedule with maximum lateness at most L in the original problem. Since the maximum lateness is bounded by $0 \leq L \leq \lceil \frac{np}{m} \rceil$, a well known technique consists of using the binary search that calls at most $\log(\lceil \frac{np}{m} \rceil)$ times the algorithm in [11] and achieves a running time complexity of $O(\log(\frac{np}{m})n^2 \min(1, \frac{p}{m}))$.

8 Conclusion

We studied variants of the problem of non-preemptive scheduling of tasks with equal processing times on multiple machines. We considered the problem with different objective functions and presented polynomial time algorithms. We also generalized the problem to the case that the number of machines fluctuate through the time.

References

1. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice hall, 1993.
2. K. Artiouchine and P. Baptiste. Inter-distance constraint: An extension of the all-different constraint for scheduling equal length jobs. In *Proc. of the 11th Int. Conf. on Principles and Practice of Constraint Programming*, pages 62–76, 2005.
3. Nikhil Bansal and Kirk Pruhs. The geometry of scheduling. *SIAM Journal on Computing*, 43(5):1684–1698, 2014.
4. C. Dürr and M. Hurand. Finding total unimodularity in optimization problems solved by linear programs. *Algorithmica*, 2009. DOI 10.1007/s00453-009-9310-7.
5. H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the 15th annual ACM symposium on Theory of computing*, pages 246–251, 1983.
6. A. V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing*, 24(3):494–504, 1995.
7. R. Harter. The minimum on a sliding window algorithm. Usenet article, 2001. http://richardhartersworld.com/cri/2001/slidingmin.html.
8. W. A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21(1):177–185, 1974.
9. J. Y-T. Leung. *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press, 2004.
10. W. Lipski Jr and F. P. Preparata. Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems. *Acta Informatica*, 15(4):329–346, 1981.
11. A. López-Ortiz and C.-G. Quimper. A fast algorithm for multi-machine scheduling problems with jobs of equal processing times. In *Proc. of the 28th Int. Symposium on Theoretical Aspects of Computer Science (STACS'11)*, pages 380–391, 2011.
12. Rolf H Möhring, Andreas S Schulz, Frederik Stork, and Marc Uetz. Solving project scheduling problems by minimum cut computations. *Management Science*, 49(3):330–350, 2003.
13. B. Simons. Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines. *SIAM Journal on Computing*, 12(2):294–299, 1983.

14. B. B Simons and M. K. Warmuth. A fast algorithm for multiprocessor scheduling of unit-length jobs. *SIAM Journal on Computing*, 18(4):690–710, 1989.
15. Barbara Simons. A fast algorithm for single processor scheduling. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 246–252. IEEE, 1978.
16. L. A. Wolsey and G. L Nemhauser. *Integer and combinatorial optimization*. John Wiley & Sons, 2014.