Chapter 6. Machine scheduling

Section 6.1. Problem definition

A company is specialized in drilling holes in hard materials like steel; it has m identical machines available to perform this process. An important client has an assignment for the company that needs to be performed as soon as possible. The assignment consists of drilling holes in a set of n pieces of material; in piece j ($j = 1, \ldots, n$), the client wants p_j holes to be drilled. Since the drilling of the holes is much more time-consuming than all side-activities, we may assume that it takes p_j units of time to process piece j. The planner of the company has the task to plan the use of the machines such that the client gets his processed pieces as quickly as possible; they form one truck-load, so we are not concerned with intermediate completion times: only the completion time of the last piece counts. He has to take into account the side-constraints that the machine can only drill one hole at a time and that the drilling process cannot be stopped before all p_j holes are drilled in piece j.

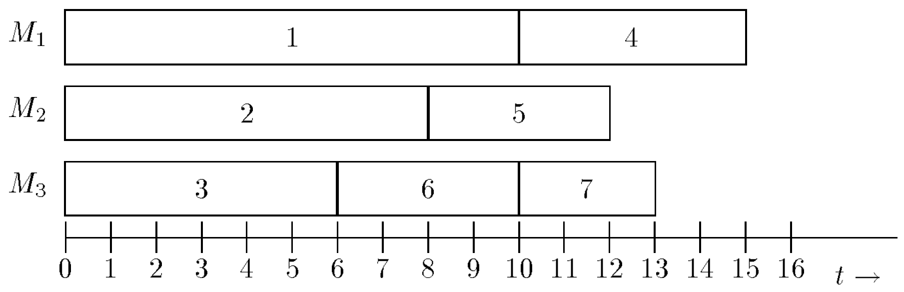
This is a typical problem in machine scheduling. It is modeled as follows. A set of n jobs has to be processed; for the processing of the jobs, there are m identical machines available from time zero onwards that can handle only one job at a time. The processing of job j ($j = 1, \ldots, n$) takes an *uninterrupted* period of p_j units of time, and it has to be executed by a single machine. We are looking for a feasible schedule, that is, an allocation of each job j to a time interval of length p_j on a machine such that no two jobs are processed by the same machine at the same time. Given a feasible schedule, we denote the *completion time* of job j ($j = 1, \ldots, n$) by C_j. The objective is to find a schedule in which the latest job finishes as soon as possible, i.e., we want to minimize the maximum completion time $C_{max} = \max_{j=1,\ldots,n} C_j$.

Example

There are 3 machines available for processing 7 jobs. The processing times are given by the following table.

j	1	2	3	4	5	6	7
p_j	10	8	6	5	4	4	3

A feasible schedule is visualized by a so-called Gantt-chart as follows.



M_1, M_2 and M_3 denote the machines. The jobs 1 and 4 are processed on machine M_1; jobs 2 and 5 are processed on M_2; jobs 3, 6, and 7 are processed on M_3. Machine M_1 finishes after 15 time units, M_2 after 12 time units, and M_3 after 13 time units. Hence, we have that $C_{max} = C_4 = 15$ in this schedule. A better schedule is obtained by swapping jobs 4 and 5 on the first two machines; this schedule finishes after 14 units of time. Since the total processing time of all jobs amounts to 40 and each processing time is integral, we know that the schedule with $C_{max} = 14$ is optimal.

Section 6.2. Minimizing the maximum completion time $C_{\max}$

Complexity

If there is only one machine available, then the problem is trivially solvable, since all jobs must be processed on the same machine. All we have to do is take care that the machine is never idle until all jobs have been processed.

If there are two or more machines available, then the problem becomes hard, since it contains the partition problem as a special case, which we show now. We take an arbitrary instance of partition and convert this instance to an equivalent instance of the 2-machine scheduling problem.

Consider any instance of partition: suppose that it has t integral weights $u_1, \ldots, u_t$ and $U = \frac{1}{2}\sum_{j=1}^{t} u_j$. We construct the following instance of the 2-machine scheduling problem: there are $n = t$ jobs with processing times $p_j = u_j$ $(j = 1, \ldots, t)$.

Theorem 6.1.
There exists a feasible schedule with $C_{\max} \leq U$ if and only if the partition problem has an affirmative answer.

Proof.
First, suppose that the partition problem has an affirmative answer, that is, there is a set $S \subset \{1, \ldots, t\}$ such that $\sum_{j \in S} u_j = U$. From this solution we construct the following solution to the 2-machine scheduling problem. The jobs corresponding to S are executed by machine 1 and the other jobs are executed by machine 2. Hence, we have that $\sum_{j \in S} p_j = \sum_{j \in S} u_j = U$ and $\sum_{j \notin S} p_j = \sum_{j \notin S} u_j = U$. This implies that both machines are finished at time U, and therefore $C_{\max} = U$.

Conversely, suppose that there exists a solution to the machine scheduling problem with $C_{\max} \leq U$. Since the total processing time amounts to $2U$, we have that both machines must finish exactly at time U. Define S as the set containing the indices corresponding to jobs that are executed by machine 1; this set S constitutes a solution to the partition problem, since $\sum_{j \in S} u_j = \sum_{j \in S} p_j = U$. □

Note that the size of the instance of the two-machine problem is polynomially bounded in the size of the instance of the partition problem (in fact, both sizes are equal); hence, we can state that the two-machine problem is as hard as the partition problem.

It is fairly easy to see that the problem with $m = 2$ is a special case of the problem with $m > 2$. For each instance $(n, p_1, \ldots, p_n)$ of the 2-machine problem, we construct an instance of the m-machine problem by adding $m - 2$ jobs of length $\frac{1}{2}\sum_{j=1}^{n} p_j = P$. Clearly, the following theorem holds.

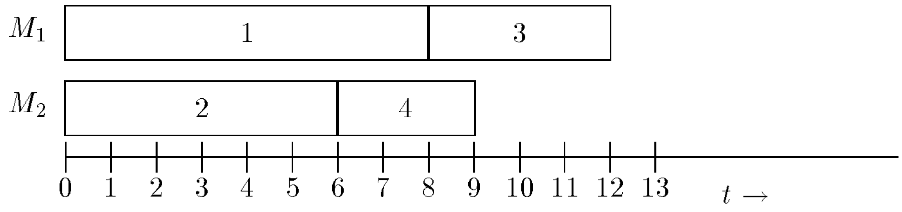
Theorem 6.2.
There is a solution of the 2-machine problem with $C_{\max} = P$ if and only if there is a solution with $C_{\max} = P$ of the m-machine problem (with $m > 2$). □

Since there is little hope for finding a polynomial algorithm to solve the 2-machine problem, we will resort to solving this problem by implicit enumeration by means of dynamic programming and to applying approximation algorithms to find a hopefully good solution in polynomial time.

A dynamic programming algorithm for the 2-machine problem

The idea behind our dynamic programming algorithm is the following. We introduce boolean variables $f_k(t_1, t_2)$ that are to signal whether it is possible to process jobs $1, \ldots, k$ such that the machines are ready at times t_1 and t_2, respectively. Since it makes no sense to leave a machine idle as long as there are unprocessed jobs available, this boils down to the question whether there exists a partition of the first k jobs such that the workload of machine 1 and 2 amounts to no more than t_1 and t_2, respectively.

Example



Hence, $f_4(12, 10) =$ true, $f_4(9, 12) = f_4(12, 9) =$ true, and $f_k(11, 9) =$ false.

The calculation of the values $f_k(t_1, t_2)$ starts with the initialization $f_0(t_1, t_2) =$ true if $t_1, t_2 \geq 0$ and $f_0(t_1, t_2) =$ false, otherwise. In the k-th iterative step, the values $f_k(t_1, t_2)$ are calculated from the values $f_{k-1}(t_1, t_2)$ by the recursion

$$f_k(t_1, t_2) = \{f_{k-1}(t_1 - p_k, t_2) \vee f_{k-1}(t_1, t_2 - p_k)\}.$$

Finally,

$$C_{\max} = \min\{t | f_n(t, t) = \text{true}\}.$$

The number of variables amounts to $n \times (\sum_{j=1}^{n} p_j)^2$. Since each variable can be calculated by an elementary operation involving only two variables, the number of elementary steps performed by the algorithm is of the same size.

We can implement our dynamic programming algorithm in a smarter way by paying only attention to the boolean variables $f_k(t_1, t_2)$ with $t_1 + t_2 = \sum_{j=1}^{k} p_j$; given these values, we can easily obtain the values of $f_k(t_1, t_2)$ with $t_1 + t_2 > \sum_{j=1}^{k} p_j$ (in fact, we do not even need these values). The calculation of the $f_k(t_1, t_2)$ values proceeds as follows. We initialize by putting $f_0(t_1, t_2) =$ true if $(t_1, t_2) = (0, 0)$ and $f_0(t_1, t_2) =$ false, otherwise. The recursive step remains the same, and $C_{\max}$ is determined as

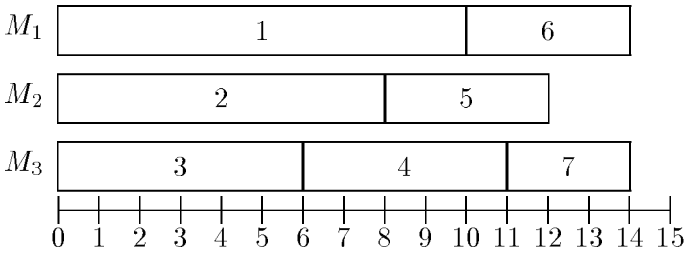
$$C_{\max} = \min\{t \geq (\sum_{j=1}^{n} p_j)/2 | f_n(t, \sum_{j=1}^{n} p_j - t) = \text{true}\}.$$

Since the number of variables has decreased to $n \times (\sum_{j=1}^{n} p_j)$, and we still can determine the value of each variable in constant time, the running time of this algorithm is $n \times (\sum_{j=1}^{n} p_j)$.

Approximation algorithms
The most common algorithms used to deal with scheduling and sequencing problems are the so-called *list scheduling* (LS) algorithms. List scheduling algorithms proceed by a two-phase greedy approach to generate solutions. In the first phase, the jobs are *listed*, that is, they are renumbered according to a certain order, for instance, to increasing processing times (SPT) or to decreasing processing times (LPT). In the second phase, the jobs are placed on the machines according to the order in which they occur in the list; the machine that is chosen to put job j on is the one with the smallest workload with respect to the jobs $1, \ldots, j-1$.

Example 7 jobs, 3 machines.

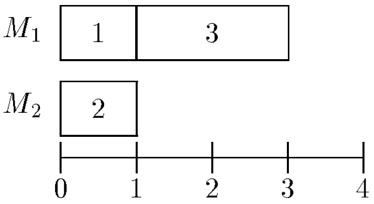
j	1	2	3	4	5	6	7
p_j	10	8	6	5	4	4	3



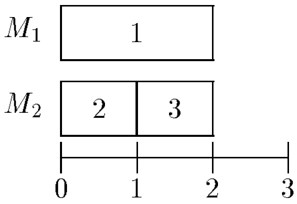
The question is of course: how bad can the solution get? For an arbitrary numbering of the jobs, the following instance is a bad one.

Example 3 jobs, 2 machines.

j	1	2	3
p_j	1	1	2



$z_{LS} = 3$



$z_{\text{opt}} = 3$

Hence, we have that $\dfrac{z_{LS}}{z_{\text{opt}}} = \dfrac{3}{2}$, and we know that we run the risk of finding a solution with an objective value that is 50% greater than necessary. The next theorem states that this is as bad as it can get.

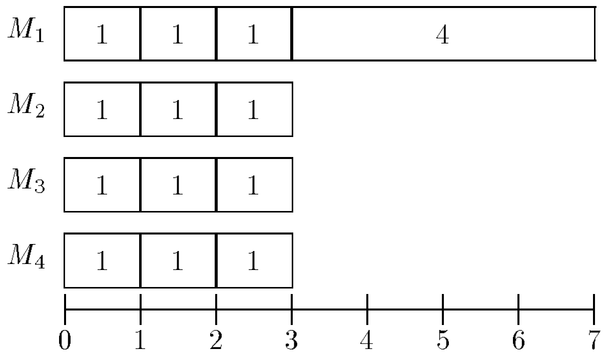
Theorem 6.3.
For any ordering of the jobs, the list scheduling algorithm finds a schedule with makespan no more than 3/2 times the length of an optimal schedule.

Proof.
Consider an optimal schedule. Let C^i_{opt} denote the workload of machine i ($i = 1, 2$) in this schedule; without loss of generality, we assume that $z_{\text{opt}} = C^1_{\text{opt}} \geq C^2_{\text{opt}}$. Analogously, C^i_{LS} denotes the workload of machine i ($i = 1, 2$) in the solution found by Algorithm LS; again, we suppose without loss of generality that $z_{LS} = C^1_{LS} \geq C^2_{LS}$.
In our proof, we make use of two obvious lower bounds on z_{opt}; these are $z_{\text{opt}} \geq p_{\max} = \max_{1 \leq j \leq n} p_j$ and $z_{\text{opt}} \geq \frac{1}{2} \sum_{j=1}^{n} p_j$. Now consider the schedule obtained by Algorithm LS; let job i be the last job scheduled on M_1. We have that $p_i \geq C^1_{LS} - C^2_{LS}$; otherwise, machine M_2 would have been idle before job i started, and hence would have been selected by Algorithm

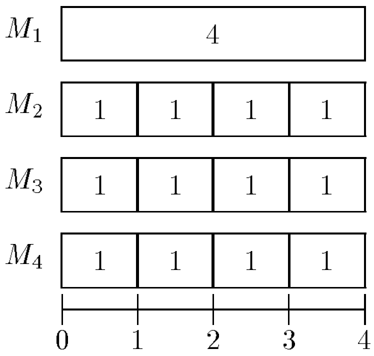
LS for processing job i. This implies that

$$z_{LS} = C^1_{LS} = \tfrac{1}{2}(C^1_{LS} + C^2_{LS}) + \tfrac{1}{2}(C^1_{LS} - C^2_{LS})$$
$$\leq \tfrac{1}{2}\sum_{j=1}^{n} p_j + \frac{1}{2}p_i \leq z_{\text{opt}} + \frac{1}{2}z_{\text{opt}} = \frac{3}{2}z_{\text{opt}}. \qquad \square$$

It can be shown that in case of m machines Algorithm LS always finds a schedule with makespan no more than $2 - \frac{1}{m}$ times the length of an optimal schedule, independent of the ordering of the jobs. The example below shows that this bound is tight for $m = 4$; it contains $m^2 - m$ jobs of length 1 and one job of length m. Note that in the picture below the length of the job is put inside the box representing it instead of its number.



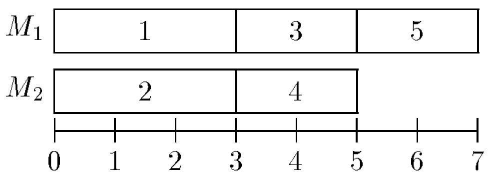
$z_{LS} = 7$



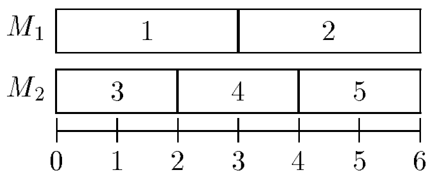
$z_{\text{opt}} = 4$

From the example above, one might get the idea that it is possible to improve the worst-case bound by starting with the jobs with large processing times, that is, by ordering the jobs in the list according to decreasing processing times. A bad instance for the list scheduling algorithm in case the jobs are ordered according to decreasing processing times is the following one.

j	1	2	3	4	5
p_j	3	3	2	2	2



$z_{LS} = 7$



$z_{\text{opt}} = 6$

The next theorem shows that this is as bad as it can get in case of two machines.

Theorem 6.4.
The list scheduling algorithm in which the jobs are assigned to the machines in LPT order always finds a schedule with length at most $\frac{7}{6}$ times the length of an optimal schedule.

Proof.

Let z_{opt} denote the makespan of an optimal schedule; let C^i_{LPT} $(i = 1, 2)$ denote the workload of machines 1 and 2. Without loss of generality, let $z_{LPT} = C^1_{LPT} \geq C^2_{LPT}$; let i be the job that is processed last on M_1. We reduce the instance by deleting the jobs $i+1, \ldots, n$: this does not increase z_{opt} and it does not decrease $z_{LPT} = C^1_{LPT}$, and hence, it does not decrease the ratio z_{LPT}/z_{opt}. This implies that if we prove the bound for the reduced instance then it certainly holds for the original instance. Due to the choice of machine 1 to put job i on, we have that $p_i \geq C^1_{LPT} - C^2_{LPT}$. We further make use in our proof of the obvious inequalities $z_{\text{opt}} \geq \frac{1}{2}\sum_{j=1}^i p_j$ and $p_i \leq \frac{1}{i}\sum_{j=1}^i p_j$ (since i is the smallest job from among the first i jobs).

$$\begin{aligned} z_{LPT} = C^1_{LPT} &= \tfrac{1}{2}(C^1_{LPT} + C^2_{LPT}) + \tfrac{1}{2}(C^1_{LPT} - C^2_{LPT}) \\ &\leq \tfrac{1}{2}\sum_{j=1}^{i} p_j + \frac{1}{2}p_i \leq z_{\text{opt}} + \frac{1}{2}(\frac{1}{i}\sum_{j=1}^{i} p_j) \\ &= z_{\text{opt}} + \tfrac{1}{i}(\tfrac{1}{2}\sum_{j=1}^{i} p_j) \leq z_{\text{opt}} + \frac{1}{i}z_{\text{opt}} = (1 + \frac{1}{i}) \times z_{\text{opt}}. \end{aligned}$$

Therefore, the theorem holds for $i \geq 6$.

For $i \leq 4$, it is a matter of simple checking to show that the LPT ordering leads to an optimal schedule. We are left with checking the case $i = 5$. We may assume that job 5 finishes last; otherwise, we can delete job 5 from the instance without decreasing the worst-case ratio. Since one of the machines, say M_1, processes at least three jobs, the workload of M_1 amounts to at least $3p_5$, which implies that $p_5 \leq \frac{1}{3}z_{\text{opt}}$. Along the lines used in the proof above, we can prove that the theorem holds for $i = 5$, too. □

Section 6.3. Minimizing the sum of completion times $\sum C_j$

Minimizing the average completion time $\frac{1}{n}\sum_{j=1}^n C_j$ is a natural objective when jobs are identified with customers waiting in a queue. Of course, this is equivalent to minimizing $\sum_{j=1}^n C_j$, since n is given. If there is only one machine available for processing the jobs, that is, $m = 1$, the problem is solved by a simple sorting of the processing times.

Theorem 6.5.
If there is only one machine available, then $\sum_{j=1}^n C_j$ is minimized by scheduling the jobs in order of nondecreasing processing times.

Proof.
We will give two proofs: the first one is meant to just prove the theorem, whereas the second one is meant to provide some insight.

Proof one:
Consider any optimal order; suppose to the contrary that the jobs are not scheduled in order of nondecreasing processing times. Then there are two adjacent jobs, say jobs j and k, such that $p_j > p_k$, whereas job j precedes job k. If we exchange the two jobs, then C_j increases by p_k and C_k decreases by p_j. This interchange decreases the value of the objective function by $p_j - p_k > 0$, since the completion times of all other jobs remain the same. This contradiction proves the theorem.

Proof two:
Suppose that the jobs are numbered according to their appearance on the machine. We then have that

$$\begin{aligned}\sum_{j=1}^{n} C_j &= C_1 + C_2 + \ldots + C_n \\ &= p_1 + (p_1 + p_2) + \ldots + (p_1 + p_2 + \ldots + p_n) \\ &= np_1 + (n-1)p_2 + \ldots + p_n.\end{aligned}$$

We see that the processing time of the job that occupies the kth position in the schedule is counted $(n+1-k)$ times. Hence, we can minimize the value of the objective function by scheduling the jobs in order of nondecreasing processing times. $\square$

If there are two or more machines available, then it is easily shown that the processing time of the job that is scheduled on the kth position on machine M_i is counted $(m_i + 1 - k)$ times in the value of the objective function, where m_i denotes the number of jobs that are executed by M_i. This observation leads to the following theorem.

Theorem 6.6.
In any optimal schedule, each machine executes either $\lfloor n/m \rfloor$ or $\lceil n/m \rceil$ jobs.

Proof.
Suppose to the contrary that there exists an optimal schedule in which some machine executes at least two jobs more than some other machine; without loss of generality, we suppose that these machines are M_1 and M_2, which implies that $m_1 \geq m_2 + 2$. Let job i be the job that is executed first by M_1. We move job i from the first position on M_1 to the first position on M_2, and we evaluate the change of the objective function. In the new value of the objective function, p_i is counted only $m_2 + 1$ times instead of m_1 times. Since the number of times that the processing times of all other jobs are counted in the new value of the objective function has not changed, moving job i has decreased the value of the objective function by $(m_1 - m_2 - 1)p_i$; this clearly contradicts the optimality of the original schedule. Therefore, in any optimal schedule each machine executes at most one job more or less than any other machine; this implies the division of the jobs over the machines stated in the theorem. $\square$

Now that we have characterized the number of jobs executed by each machine, we can assign the jobs to the positions on the machines. Since the processing time of any job that is processed last on any machine is counted only once in the value of the objective function, it is optimal to assign the m longest jobs to these positions; it does not matter to which machine they are assigned, as long as they are executed last. Similarly, it is optimal to assign the next m longest jobs to the last but one positions on the machines, and so on.

Section 6.4. Minimizing the sum of the weighted completion times $\sum w_j C_j$

In the previous section, we have considered the problem of minimizing $\sum_{j=1}^{n} C_j$. In general, however, we will not consider each job to be equally important. We can express this difference in importance by attaching a *weight* w_j to job j $(j = 1, \ldots, n)$; the objective is now to minimize $\sum_{j=1}^{n} w_j C_j$. Usually, the weights are assumed to be positive integers.
In case of a single machine, we have that $\sum_{j=1}^{n} w_j C_j$ is minized by scheduling the jobs according to nonincreasing w_j/p_j ratios without unnecessary idle time; this is called Smith's rule.

The correctness of this rule can be proven in a fashion similar to the first proof of Theorem 6.5.

In case of more than one machine, the above result implies that given the assignment of jobs to machines we can determine an optimal schedule for this assignment by processing the jobs in order of nonincreasing w_j/p_j ratio on each machine. Unfortunately, there is no polynomial algorithm known that finds the optimal assignment of jobs to machines. In fact, we can show that partition is a special case of the problem of minimizing the sum of the weighted completion times on two machines.

Consider any instance of partition: suppose it has t integral weights $u_1, \ldots, u_t$ and $U = \frac{1}{2}\sum_{j=1}^{t} u_j$. We construct the following instance of the 2-machine scheduling problem: there are $n = t$ jobs with processing times and weights $p_j = w_j = u_j$ $(j = 1, \ldots, n)$.

Theorem 6.7
The partition problem has an affirmative answer if and only if there is a solution to the corresponding scheduling problem with value no more than $\sum_{i \leq j} u_i u_j - U^2$.

Proof.
First of all, note that the w_j/p_j ratio of all jobs is equal; this implies that the value of the objective function does not depend on the ordering of the jobs on the machines.

Consider any assignment of the jobs to the machines: suppose that the workloads assigned to M_1 and M_2 amount to $U + \delta$ and $U - \delta$, respectively. We show that the value of the objective function for the schedule corresponding to this assignment amounts to $\sum_{i \leq j} u_i u_j - U^2 + \delta^2$.
First, suppose that $\delta = U$, that is, all jobs have been assigned to M_1 and none to M_2. A straightforward calculation shows that the corresponding schedule has $\sum_{j=1}^{n} w_j C_j = \sum_{i \leq j} u_i u_j$. Now suppose that $\delta < U$, that is, M_2 processes a set S of jobs. If we move all jobs in S from M_2 to M_1 to be processed by M_1 immediately after all jobs that are not in S, then we increase the completion time of each job in S by $U + \delta$. We have just computed that the newly obtained schedule has $\sum_{j=1}^{n} w_j C_j = \sum_{i \leq j} u_i u_j$; clearly, the value of the objective function of the original schedule must have been equal to

$$\begin{aligned}
\sum_{j=1}^{n} w_j C_j &= \sum_{i \leq j} u_i u_j - (U + \delta) \sum_{j \in S} w_j \\
&= \sum_{i \leq j} u_i u_j - (U + \delta)(U - \delta) \\
&= (\sum_{i \leq j} u_i u_j - U^2) + \delta^2.
\end{aligned}$$

Proving the theorem has now become straightforward. Suppose that the partition problem has an affirmative answer; let $S \subset \{1, \ldots, t\}$ be such that $\sum_{i \in S} u_i = U$. Then we can determine a schedule with $\sum_{j=1}^{n} w_j C_j = \sum_{i \leq j} u_i u_j - U^2$ by assigning the jobs in S to M_1 and all other jobs to M_2. Conversely, a schedule with $\sum_{j=1}^{n} w_j C_j = \sum_{i \leq j} u_i u_j - U^2$ must be such that the workload on both machines is equal to U, and a subset of the weights that leads to an affirmative answer to partition is readily obtained. $\square$

Since the size of the input of the instance of the problem of minimizing $\sum_{j=1}^{n} w_j C_j$ on two machines that we constructed is polynomial in the size of the input of the instance of partition, we have that the problem of minimizing $\sum_{j=1}^{n} w_j C_j$ on two machines is at least as hard as the problem partition.