Deterministic Scheduling: Algorithms and Complexity

Script[1] of the lecture "Scheduling" taught by Britta Peis and Katharina Eickhoff

RWTH Aachen, Chair of Management Science

Summer Term 2022

[1]Thanks to Prof. Dr. Nicole Megow and Dr. Veerle Tan Timmermans for allowing us to use some of their lecture notes. Thanks to Niklas Rieken for creating the pictures and other work on the script.

Contents

Chapter 1

Introduction to Scheduling

In this chapter we give a short introduction to the wide field of deterministic scheduling problems. Scheduling problems can be used to model all kinds of situations where limited resources must be allocated over time to a set of tasks.

1.1 Introduction

Scheduling theory covers an enourmous variety of problem types, all of which having in common that a set of activities (tasks or jobs) need to be assigned to a limited set of resources (or machines) over time. For some types of such scheduling problems, efficient algorithms exist, while other types are known to be notourosly hard (NP-hard) and it is widely believed that an efficient algorithm will never exist, not even in a thousand years. And, of course, there is still a large amount of problem types for which we just don't know yet, whether they belong to the class of efficiently solvable or NP-hard problems. We will provide you with a notion of efficient algorithms and NP-hard problems later in this course. The overall goal of this course is to provide you with a basic understanding of algorithmic techniques that turn out to be useful for solving scheduling problems, and we will teach you how to classify scheduling problems into efficiently tractable and NP-hard problems.

Definition 1 ([Graham et al., 1979])**.** Sequencing and scheduling is concerned with the optimal allocation of activities (*tasks* or *jobs*) to scarce resources over time.

In general, a scheduling problem consists out of a set of *jobs* $J = \{J_1, \ldots, J_n\}$ (or $J = \{1, \ldots, n\}$, for short), a set of resources, typically called *machines*, $M = \{M_1, \ldots, M_m\}$ (or $M = \{1, \ldots, m\}$, for short), and an objective function which typically depends on the completion times of the jobs. Each job $j \in J$ needs to be executed on one or more machines. We assume that all jobs $j \in J$ have a fixed *processing time* p_{ij} for each machine $i \in M$, which denotes the processing time of job j if it is entirely processed on machine i. We assume that the processing time does not change over time, and that it is independent by when the job is processed or by when the processing of other jobs occurs. There might also be other job characteristics such as *release dates*, *deadlines*, *due dates*, and *precedence constraints*, as we will describe below in more detail. But, independent of the characteristics of the job, we always assume that at any given point in time *no machine can process more than one job and no job can be processed by more than one machine simultaneously.*

In this course, we restrict our considerations to *deterministic scheduling problems*, in which all data are known precisely. That is, we assume that there will be no uncertainty about the processing requirements of a job, or about other job characteristics such as release dates or deadlines. Otherwise, we would be dealing with *stochastic* scheduling problems, involving probability and queueing theory as tools.

The output of a scheduling procedure is a *schedule*, which specifies exactly which job, if any, each machine works on at each point in time. In any schedule, there is a well defined *completion time* C_j for each job $j \in J$, denoting the latest point in time when job j is processed on a machine. Typically, the objective function which the scheduler aims to minimize, depends upon

the completion times of the schedule. Schedules can be represented by *Gantt charts*, which were developed by Henry Gantt in 1910 to illustrate project schedules. The horizontal axis indicates time and each band is identified with a machine. The intervals in which a machine is assigned to no job are hatched; such periods are known as *idle time*. An example of a Gantt chart can be found in Figure 1.1.

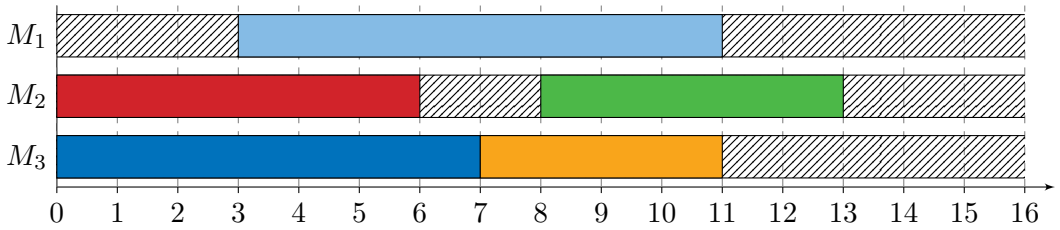


Figure 1.1: A Gantt chart for three machines and five jobs.

Let's consider three examples of real-world scheduling problems.

Example (Gate assignments in airports)**.** In an airport one can find dozens of gates, in which hundreds of planes arrive and depart each day. These flights operate under a certain schedule, that specifies the arrival time and departure time at each gate. When constructing a schedule, one also needs to take the service time at the gate into account. The task is to assign planes to gates while minimizing objectives such as work for airline personnel and airplane delays.

Example (Staff scheduling in hospitals)**.** In a hospital, staff does not only work during normal business hours, but also at night and during the weekends. Due to labor agreements and law, not too many consecutive night shifts can be done by one staff member. Staff members can submit preferences for certain evenings or weekends. The task is to create a schedule such that there is enough staff present at all times, satisfies the labor and law agreements, is fair and takes the preferences of the staff into account.

Example (Scheduling tasks in a CPU)**.** Given a central processing unit (CPU) and a set of tasks that need to be processed on the CPU, we need to allocate CPU time to tasks with the objective of minimizing the average time a task spends in the system.

If the processing time of job j does not depend on the machine, we just write p_j. Next to a processing time, a job can have more relevant information. For example, a job j might have:

- A *release date* r_j, thus one can only start processing this job after time r_j.
- A *due date* d_j, i.e. a time where job j *should* be completed.
- A *deadline* $\bar{d}_j$, i.e. a time where job j *must* be completed.
- A *weight* w_j, which gives some information on the profit or importance of the job.

Recall that, for a given schedule, the *completion time* of job j is denoted by C_j. One possible problem instance can be:

Example. Assume we are given three jobs that need to be processed on two machines. Each job needs to be finished on machine 1 before it can be processed on machine 2. Each job has a processing time p_{ij} on each machine, a weight and a due date.

	$p_{1,j}$	$p_{2,j}$	w_j	d_j
J_1	2	3	3	6
J_2	1	4	1	9
J_3	3	1	5	4

Find a schedule that minimizes the sum of weighted tardiness $\sum_j w_j T_j$. The *tardiness* is defined as $T_j := \max\{0, C_j - d_j\}$.
Note that it is never beneficial to leave the machine unnecessarily idle in between jobs. Thus, we only need to determine in which order the jobs need to be processed on which machine. For this specific problem, the optimal solution is illustrated in the Gantt chart in Figure 1.2. At the moment you do not have any algorithms at your disposal, and you can only find such a solution by trying all possible schedules.

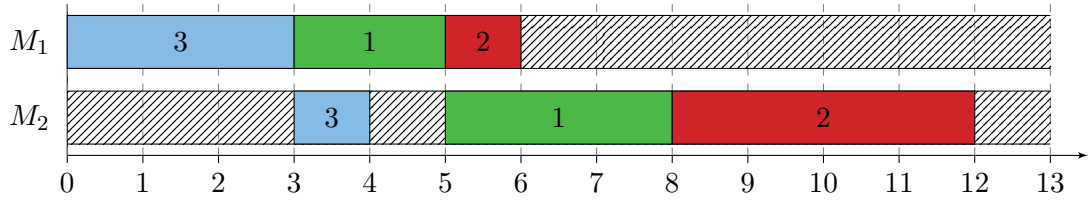


Figure 1.2: Optimal solution for Example 1.1.

Under this schedule, where we schedule the jobs in order $(3, 1, 2)$, we see that $C_1 = 8, C_2 = 12$ and $C_3 = 4$. Thus, $T_1 = 8 - 6 = 2$, $T_2 = 12 - 9 = 3$ and $T_3 = 4 - 4 = 0$. Hence,

$$\sum_j w_j T_j = 3 \cdot 2 + 1 \cdot 3 + 5 \cdot 0 = 9.$$

In this course we study many different scheduling problems and discuss algorithms for the problem at hand. We study state of the art solution methods and give you tools to help you to design and verify algorithms on your own. Though scheduling models are motivated by real-world application, we focus in this course on the theoretical foundations.

1.2 Classification of Scheduling Problems

In this section, we introduce the 3-field notation $\alpha \,|\, \beta \,|\, \gamma$ which is commonly used to classify scheduling problems. In the $\alpha \,|\, \beta \,|\, \gamma$-notation,

- α describes the machine environment,
- β contains the job characteristics, and
- γ describes the objective function.

A field may contain more than one entry, but can also stay empty. We discuss all three fields and their most common entries in more detail below. For further details, we refer the interested reader to chapter 1 in the Pinedo's standard text book on scheduling [Pinedo, 2012].

1.2.1 Machine Environment (the α-field)

A machine environment always contains exactly one value. There are two types of machine environments. First of all, there are the environments where jobs only need to be processed on one machine.

$\alpha = 1$ There is a single machine.

$\alpha = P$ There are multiple machines that run in parallel with the same speed.

$\alpha = Q$ There are multiple *uniform* (also called *related*) machines that run in parallel with different speeds $s_1, \ldots, s_m$. The processing time of job j on machine i is then $p_{ij} = p_j / s_i$.

$\alpha = R$ There are multiple *unrelated* parallel machines and the processing time for a job j on machine i is p_{ij}.

Then, there are also the *shop models*, where each job consists of of mutliple operations that all have to be processed on specified machines.

$\alpha = O$ In an *open shop* each job consists of exactly m (the number of machines) operations (one for each machine) with some processing requirements p_{ij}. The order in which these operations are completed is arbitrary. Note that if one wants to model a problem that includes a job where certain operations do not make sense, one can simply set $p_{ij} = 0$. (For instance, vehicle inspection for cars and motorbikes: There might be some operation *check windows*, which do not make sense for motorbikes.)

$\alpha = F$ This environment is called *flow shop*. There are multiple machines in series, and each job needs to be processed on each machine. All jobs follow the same route: first machine 1, then machine 2, etc. A special form of the flow shop is the *permutation flow shop*. In this model, jobs also need to be processed in the same order on each machine.

$\alpha = J$ In a *job shop* each job is given its individual route to follow. A job does not need to be processed on each machine. In a *recirculation job shop* jobs can even visit machines more than once.

We write Pm, Qm, Rm, Fm, Jm and Om when the number of machines m is fixed.
Lastly, in a *flexible shop problem* there are stages instead of machines, and per stage there are multiple parallel machines.

1.2.2 Job Characteristics (the β-field)

The set of job characteristics can be empty, but may also contain more than one value. Well-know characteristics are:

$r_j \in \beta$ If job j is assigned a release time r_j, then job j may not start before time r_j. If a job does not have a release time, then it may start at any time.

$d_j \in \beta$ If job j is assigned a due date d_j, then we try to schedule job j such that $C_j \leq d_j$. Note that a due date is not strict, and we are allowed to process job j after the due date. Though, this often results in a penalty in the objective function.

$\bar{d}_j \in \beta$ A more restrictive due date is called *deadline* – if a job j is not completed at time $\bar{d}_j$, the schedule becomes infeasible.

***pmtn* $\in \beta$** If *pmtn* (*preemption*) is a job characteristic, this means that jobs may be interrupted for processing and resumed later even on a different machine. When *pmtn* is not a job characteristic, the processing of an operation might not be interrupted.

***prec* $\in \beta$** If *prec* is a job characteristic, then certain *precedence* relations between certain pairs of jobs are present. Whenever there is a precedence relation indicating that a job j must precede a job k, we write $j \to k$, or $j \prec k$. Note that precedence constraints are *transitive* in the sense that $i \prec j$ and $j \prec k$ implies $i \prec k$. With precedence constraints present, in a feasible schedule, a job may not start before all its predecessors have been finished. These relations are typically represented by an acyclic digraph, called *comparability graph*, whose vertices represent the jobs, and an arc (j, k) indicating that $j \prec k$. The transitive reduction which is obtained by iteratively deleting transitive arcs in a comparability graph, is called *Hasse diagram*.

1.2.3 Objective Function (the γ-field)

Objective γ is mostly a minimization objective. Before we state popular objective functions, we first introduce extra notation.

- C_j denotes the completion time of job j.

- $F_j := C_j - r_j$ denotes the *flow time* of job j
- $L_j := C_j - d_j$ denotes the *lateness* of job j. The lateness can be negative!
- $T_j := \max\{0, L_j\}$ is the *tardiness* of job j.
- U_j is a *unit penalty* for tardy jobs. Thus, $U_j = 1$ if $T_j > 0$ and $U_j = 0$ if $T_j = 0$.

For each of these variables, one can try to minimize the maximal occurrence of the value. For example, one can try to minimize $C_{\max} := \max_j C_j$ the maximal completion time, also called *makespan*. Alternatively, one can try to minimize the sum of (weighted) completion times, $\sum_k (w_j)C_j$. Similar, one can try to minimize, $F_{\max}$, $L_{\max}$, $T_{\max}$, or $\sum_j (w_j)F_j$, $\sum_j (w_j)L_j$, $\sum_j (w_j)T_j$, and $\sum_j (w_j)U_j$. Note that minimizing the (weighted) sum $\sum(w_j)C_j$ (or $\sum(w_j)F_j$, $\sum(w_j)L_j$) is equivalent to minimizing the (weighted) average completion time (or flow time, lateness, respectively), since this involves only a multiplication by $\frac{1}{n}$, which is independent of the schedule.

1.3 Example Problems

With all possibilities for α, β and γ one can create an incredible huge amount of different scheduling problems. Let's consider just a few examples.

Example. Consider the single machine scheduling problem $1 \,|\, prec, r_j \,|\, C_{\max}$. This means that we are given a single machine and a finite set of jobs $\{1, \ldots, n\}$. Each job is endowed with a processing time p_j and a release time r_j. Furthermore, we are given a set of precedence constraints $j \to k$, meaning that job k can only start after job j is finished. The objective is to find a schedule that minimizes the makespan $C_{\max}$.
Assume there is one machine that needs to process four jobs. The processing times of the jobs are $p = (2, 1, 2, 1)$ and release dates are $r = (4, 2, 1, 4)$. The precedence constraints are depicted in Figure 1.3.

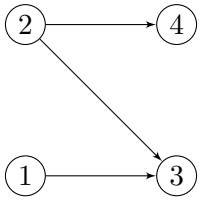
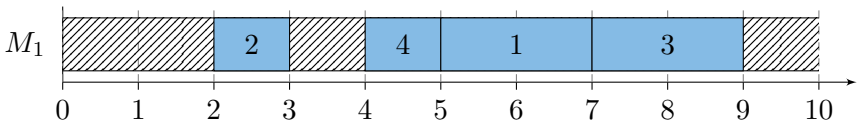


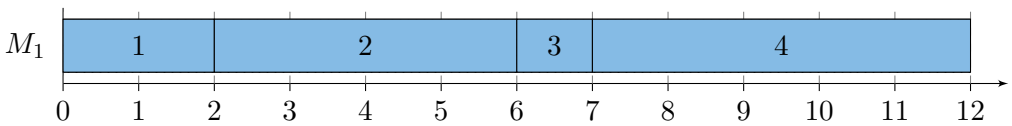
Figure 1.3: Precedence constraints on the four jobs.

A feasible schedule might look as follows:



In this particular schedule $C_{\max} = 9$.

Example. We consider an instance of $1 \,|\, | \sum w_j C_j$ with 4 jobs, processing times $p = (2, 4, 1, 5)$, and weights $w = (2, 2, 3, 4)$. Then the objective is to create a schedule that minimizes $\sum_j w_j C_j$. We first make the observation that, for this particulare problem, it does not make sense to leave the machine idle at any time. Hence, an order, also called permutation, of the jobs completely describes the schedule. Consider, for example, the schedule $\pi = (1, 2, 3, 4)$.



For this particular schedule, we obtain that

$$\sum w_j C_j = 2 \cdot 2 + 2 \cdot 6 + 3 \cdot 7 + 4 \cdot 12 = 85.$$

But how do we know whether or not this schedule is optimal?

In the next chapter, we study single machine scheduling problems and discuss some well-known algorithms.

Chapter 2

Single Machine Scheduling

We discuss three single machine scheduling problems and algorithms that, independent of the input, always output an optimal schedule for the problem at hand. In Section 2.1, we show that the *weightest shortest processing time first (WSPT)* rule optimally solves $1\,|\,|\sum w_jC_j$. In Section 2.2, we prove that the *earliest due date first (EDF)* rule is optimal for $1\,|\,|\,L_{\max}$. In Section 2.4, we show that a modified version of EDF can be used to solve $1\,|\,prec\,|\,L_{\max}$. In Section 2.5, we present and analyze Lawler's algorithm for solving $1\,|\,prec\,|\,L_{\max}$. Finally, in Section 2.6, we discuss the Moore-Hodgson algorithm for solving $1\,|\,|\sum U_j$.

2.1 WSPT rule for $1\,|\,|\sum w_jC_j$

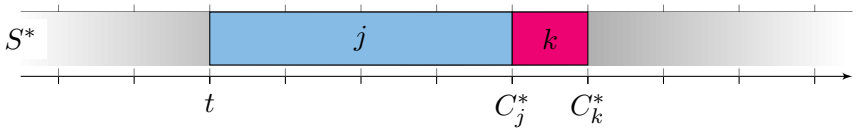
In the single machine scheduling problem $1\,|\,|\sum w_jC_j$, we are given a finite set of jobs $J = \{1,\dots,n\}$ where each job $j \in J$ has a processing time $p_j > 0$ and a weight $w_j \geq 0$. The task is to find a schedule that minimizes the weighted sum of completion times $\sum_{j\in J} w_jC_j$. Note that there always exists an optimal schedule without idle time in between jobs. Hence, the problem reduces to finding an optimal ordering (i.e. a permutation of the jobs) according to which the jobs are scheduled on the machine.

Definition 2. The *weighted shortest processing time (WSPT)* rule schedules jobs in non-increasing order of ratio w_j/p_j.

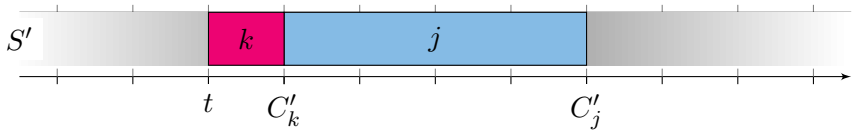
The WSPT rule is also called Smith's rule or the Photographer's rule (small and important people should be in front). Note that when all weights are equal to 1, i.e., $w_j = 1$ for all $j \in J$, we order jobs in non-decreasing order of processing time p_j (SPT rule).

Theorem 3 ([Smith, 1956])**.** *The WSPT rule gives an optimal solution for* $1\,|\,|\sum w_jC_j$.

Proof. We prove this theorem via some interchange argument. For the sake of contradiction, suppose that S^* is an optimal schedule which is not ordered according to the WSPT rule. Thus, there exist at least two jobs j and k violating the WSPT rule. That is, j is scheduled before k, although $\frac{w_j}{p_j} < \frac{w_k}{p_k}$. Observe that, whenever such a violating pair exists, there also exists a neighbouring violating pair where j is scheduled directly before k. Among all such neighbouring violating pairs, let's consider the pair $\{j,k\}$ scheduled first under S^*.



Let's create an alternative schedule S' from S^* by swapping the positions of job j and job k.



How about the change in the objective values when going from S^* to S'? Since the completion times of the jobs in $J \setminus \{j,k\}$ remain the same, we only need to consider the completion times of j and k. Let t denote the starting time of job j under S^*. Further, let us denote by C_j^* and C_j' the completion times of job j under schedule S^* and S', respectively. Similar, C_k^* and C_k' denote the completion times of job k under S^* and S', respectively. Since $C_j^* = t+p_j$, $C_k^* = t+p_j+p_k = C_j'$, and $C_k' = t+p_k$, we derive for the difference in the objective values

$$\begin{aligned}\sum_{j' \in J} w_{j'} C_{j'}^* - \sum_{j' \in J} w_{j'} C_{j'}' &= w_j C_j^* + w_k C_k^* - w_j C_j' - w_k C_k' \\ &= w_j(t+p_j) + w_k(t+p_j+p_k) - w_j(t+p_k+p_j) - w_k(t+p_k) \\ &= -w_j p_k + w_k p_j \\ &> 0.\end{aligned}$$

This contradicts the optimality of S^*. Hence, only the WSPT rule outputs an optimal schedule. □

2.2 EDF for $1 \mid\mid L_{\max}$

In problem $1 \mid\mid L_{\max}$, we are given a finite set of jobs $J = \{1, \ldots, n\}$ where each job $j \in J$ has a processing time $p_j > 0$ and a due date $d_j \geq 0$. The task is to find a schedule that minimizes the maximum lateness $L_{\max} = \max_{j \in J} L_j$, where the lateness L_j of job j is defined as $L_j := C_j - d_j$. Observe that L_j is negative whenever job j finishes earlier than the due date d_j. As in the previous section observe that it is never beneficial to leave the machine idle, so that problem $1 \mid\mid L_{\max}$ reduces to finding an optimal order according to which the jobs are scheduled on the machine.

Definition 4. The *Earliest Due date First (EDF)* rule schedules jobs in non-decreasing order of due dates d_j.

Theorem 5 ([Jackson, 1955])**.** *The EDF rule gives an optimal solution for* $1 \mid\mid L_{\max}$.

Proof. Once more, we prove the theorem via an interchange argument. Let S^* be an optimal schedule which is not ordered according to the EDF rule. Then there exist at least two neighbouring jobs j and k, such that j is scheduled directly before k, although $d_k < d_j$. Consider an earliest such pair, and construct a new schedule S' by swapping the positions of j and k. For any job $\ell \in J$, let us denote by L_ℓ^* and L_ℓ' the lateness of job ℓ under schedule S^* and S', respectively. Recall that $L_\ell^* = C_\ell^* - d_\ell$ and $L_\ell = C_\ell' - d_\ell$, where C_ℓ^* and C_ℓ' denote the completion times of job ℓ under schedule S^* and S', respectively. How about the change in the objective value when going from S^* to S'? Observe that the lateness remains the same for all jobs $\ell \in J \setminus \{j,k\}$. Since k is shifted to the left, we clearly have $L_k' \leq L_k^* \leq L_{\max}^* := \max_{j' \in J} L_{j'}^*$. So we only need to care about the lateness of job j under the new schedule S'. However, since $C_j' = C_k^*$ and $d_k < d_j$, it follows that

$$L_j' = C_j' - d_j = C_k^* - d_j < C_k^* - d_k \leq L_{\max}^*.$$

Thus, $L_{\max}' := \max_{\ell \in J} L_\ell' \leq Ł_{\max}^*$, i.e. the maximum lateness does not increase when going from S^* to S', implying that S' is an optimal schedule as well. Iterating this procedure of swapping jobs violating EDF as long as such jobs exist, results in an optimal schedule in EDF-order. □

In Section 2.4 below, we will see how to modify EDF to cope with precedence constraints when minimizing the maximum lateness on a single machine. The subsequent Section 2.3 formally introduces precedence constraints, posets, chains, antichains, intrees and outtrees.

2.3 Precedence Constraints

Let J be a set of jobs. Precedence constraints $j \prec k$ among certain pairs of jobs in J are used to model constraints of type "job j must be completed before job k can start". Note that precedence

constraints are *transitive* in the sense that $j \prec k$ and $k \prec l$ imply $j \prec l$ for all $\{j, k, l\} \subseteq J$. A set of jobs J together with a set of precedence constraints of type $j \prec k$ forms a *partially ordered set* $(J, \prec)$, for short: *poset* $(J, \prec)$. The *comparability graph* associated to posets $(J, \prec)$ is the acyclic digraph $D = (J, A)$ whose vertices correspond to the jobs in J, and two vertices j and k are linked by an arc (j, k) whenever $j \prec k$. Usually, precedence constraints among jobs are illustrated in a *Hasse diagram*, which is obtained from the comparability graph of poset $(J, \prec)$ by iteratively deleting all transitive arcs.

Definition 6. A pair of jobs $\{j, k\}$ is called *comparable* if $j \prec k$ or $k \prec j$. Otherwise, the pair is called *incomparable*.

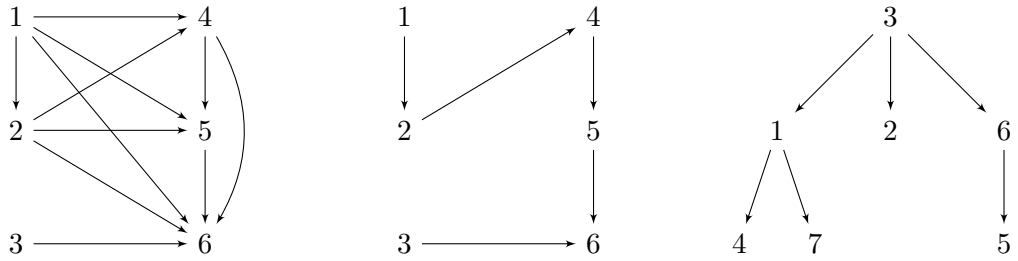
Definition 7. A *chain* in a poset $(J, \prec)$ is a subset of jobs $C \subseteq J$ such that any two jobs in C are comparable. An *antichain* is a subset $A \subseteq J$ such that no pair in A is comparable.

Definition 8. A poset $(J, \prec)$ is an *intree* if each vertex in the associated Hasse diagram has out-degree at most one. Poset $(J, \prec)$ is an *outtree* if each vertex in the associated Hasse diagram has in-degree at most one.

Example. Consider the graphs in Figure 2.1. Figure 2.1a shows the comparability graph for $(\{1, 2, 3, 4, 5, 6\}, \prec)$ with

$$\prec = \{(1,2), (1,4), (1,5), (1,6), (2,4), (2,5), (2,6), (3,6), (4,5), (4,6), (5,6)\}.$$

Every two vertices that are endpoints in one common arc are comparable. The vertices 3 and 5, for example, are incomparable, we only know that both are "smaller" ($\prec$) than 6. By removing redundant arcs in the comparability graphs i.e. those who are implicit by transitivity of $\prec$, we end up in the Hasse diagram in 2.1b. A chain can be viewed as a subset of vertices that share a directed path in a Hasse diagram. In this Hasse diagram there are two (inclusion-)maximal chains, $\{1, 2, 4, 5, 6\}$ and $\{3, 6\}$. Here, $\{3, j\}$ is an antichain for each $j \in \{1, 2, 4, 5\}$. This Hasse diagram happens to be an intree (rooted at 6). Note that this not in general the case. In 2.1c we have an outtree (that is unrelated to the previous instance) rooted at 3. We can always align the vertices of in- and outtrees by their *height* (i.e. its distance to the root) in the graph (like in 2.1c). Then it is easy to see that each *branch* (directed path from root to leaf, or vice versa) is a maximal chain while the set of all leaves is a maximal antichain.



(a) Comparability Graph (b) Hasse diagram (happens to be an intree) (c) An outtree

Figure 2.1: Illustrations of the Definitions 6, 7, and 8

2.4 Adapting EDF for $1 \,|\, prec \,|\, L_{\max}$

In this section we discuss a modification of the EDF-algorithm which constructs an optimal schedule for $1 \,|\, prec \,|\, L_{\max}$. An important observation is that whenever $d_j \leq d_k$ and $j \prec k$, then EDF respects the precedence constraints (assuming that EDF uses a tie-breaking rule respecting the precedence constraints). Unfortunately, it might occur that $d_j > d_k$ and $j \prec k$, which implies that EDF would violate this precedence constraint. The idea is to modify the due dates d_j to d'_j such that the modified due dates obey the precedence constraints in the sense that $d'_j \leq d'_k$ whenever $j \prec k$.

Observe that, if $j \prec k$, we may set

$$d'_j = \min\{d_j, d_k - p_k\}.$$

For any feasible schedule respecting the precedence constrains with completion times C it holds

$$\max_{\ell \in J}\{C_\ell - d_\ell\} = \max_{\ell \in J}\{C_\ell - d'_\ell\},$$

i.e., the objective $L_{\max}$ remains the same, since

$$\max_{\ell \in J}\{C_\ell - d_\ell\} \geq C_k - d_k \geq (C_j + p_k) - d_k = C_j - (d_k - p_k)$$

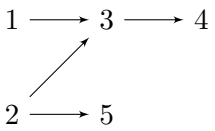
Given an instance of $1 \,|\, prec \,|\, L_{\max}$, we will solve the problem optimally by first modying the due dates according to the procedure described below, and then afterwards applying EDF to the modified due dates which gives us an optimal feasible schedule respecting the precedence constrains.

Procedure to modify due dates. Consider the Hasse diagram $D = (J, A)$ of the partially ordered set $(J, \prec)$. Modify the due dates according to the following procedure: Iteratively, while a job k of outdegree 0 exists in D, select such a job k and, for each arc $(j, k) \in A$, update d_j to $d_j = \min\{d_j, d_k - p_k\}$. When all such arcs with head k have been processed, delete them together with vertex k from the Hasse diagram D.

Corollary 9. *EDF applied to due dates modified according to the procedure described above solves* $1 \,|\, prec \,|\, L_{\max}$ *optimally.*

Example. Consider the instance of $1 \,|\, prec \,|\, L_{\max}$ given by processing times and due dates according to the following table and with precedence constraints as illustrated in the following Hasse diagram.

jobs	1	2	3	4	5
p_j	2	1	4	1	2
d_j	3	6	4	3	2



The procedure described above modifies of jobs $\{1, 2, 3\}$ as follows:

$$\begin{aligned}
d_3 &:= \min\{d_3, d_4 - p_4\} = \min\{4, 3 - 1\} = 2,\\
d_1 &:= \min\{d_1, d_3 - p_3\} = \min\{3, 2 - 4\} = -2,\\
d_2 &:= \min\{d_2, d_3 - p_3\} = \min\{6, 2 - 4\} = -2,\\
d_2 &:= \min\{d_2, d_5 - p_5\} = \min\{-2, 2 - 2\} = -2.
\end{aligned}$$

The resulting vector of modified due dates thus becomes $d' = (-2, -2, 2, 3, 2)$, implying that any of the four permutations $(1, 2, 3, 5, 4)$, $(2, 1, 3, 5, 4)$, $(1, 2, 5, 3, 4)$, or $(2, 1, 5, 3, 4)$ results in an optimal schedule of maximum lateness 7.

2.5 Lawler's Algorithm

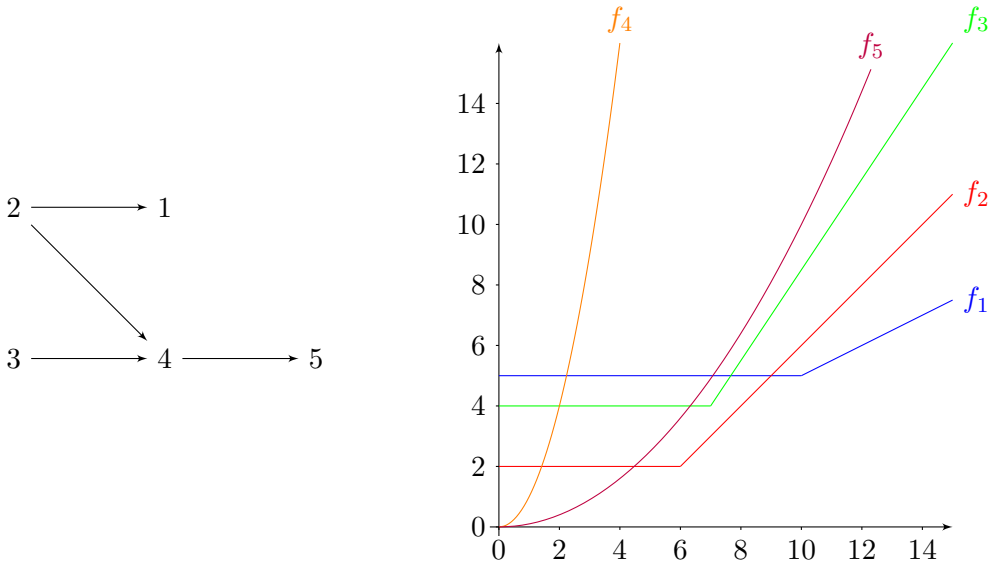
In the previous section, we studied problem $1 \,|\, prec \,|\, L_{\max}$ and showed that when we apply EDF to some modified due dates, we find an optimal schedule efficiently. In this section, we generalize the problem to $1 \,|\, prec \,|\, f_{\max}$, defined as follows.

In $1 \,|\, prec \,|\, f_{\max}$, we are given n jobs $J = \{1, \ldots, n\}$ with processing times $p_1, \ldots, p_n$, precedence constraints $(J, \prec)$ and non-decreasing cost functions $f_1, \ldots, f_n \colon \mathbb{R}_+ \to \mathbb{R}$. The task is to find a permutation π of the jobs minimizing

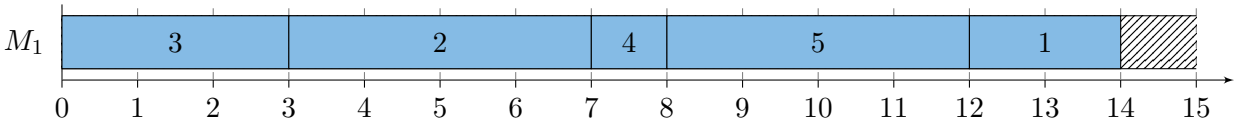
$$f_{\max} \coloneqq \max_{j \in J} f_j(C_j^\pi).$$

Note that the problem $1 \,|\, prec \,|\, L_{\max}$ is a special case of the problem $1 \,|\, prec \,|\, f_{\max}$ where all jobs have $f_j = L_j = C_j - d_j$ which is indeed a non-decreasing function.

Example. Assume we are give five jobs with processing times $p = (2, 4, 3, 1, 4)$, and precedence constraints and cost functions as illustrated in the figure below.



For this particular example, it is a feasible schedule to schedule the jobs in order $\pi = (3, 2, 4, 5, 1)$ without leaving any idle time between the jobs. This results in the following schedule.



The objective value of this schedule is:

$$f_{\max} = \max\{f_1(14), f_2(7), f_3(3), f_4(8), f_5(12)\} = f_4(8) = 8^2 = 64.$$

Note that if a function f_i is decreasing, then idle times might be beneficial. By restriction to non-decreasing functions, we know that there always exists a non-delay schedule (c.f. exercise). Hence, problem $1 \,|\, prec \,|\, f_{\max}$ reduces to finding an optimal permutation of the jobs.

Definition 10. A feasible schedule is a *non-delay* schedule if no machine is idle when a job is available for processing.

Whenever the schedule is determined by a permutation, the makespan $C_{\max} = \sum_{j \in J} p_j$ is independent of the chosen permutation.
The idea of Lawler's algorithm is to plan the jobs backwards from $C_{\max}$ to 0. At any decision point, pick the job that is 'cheapest' among those that can be scheduled according to the precedence constraints $(J, \prec)$. The pseudocode of Lawler's algorithms is as follows:

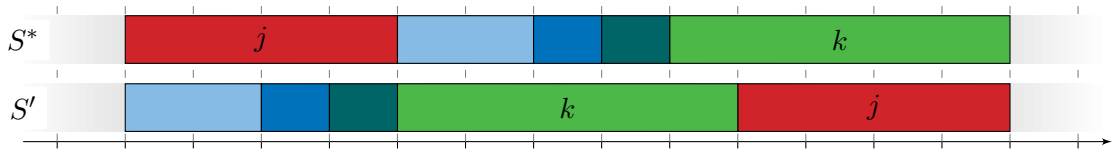
Algorithm Lawler's Algorithm:

Input: A set of n jobs J, processing time p_j and cost function f_j for each $j \in J$
Output: A feasible and optimal non-delay schedule

```
S := ∅
J' := {j ∈ J | j has no successor in J \ S}
t := Σ_{j∈J} p_j
while S ≠ J do
    Select j ∈ J' with f_j(t) = min_{k∈J'} f_k(t)
    Schedule j in interval [t − p_j, t]
    Update t := t − p_j, S := S ∪ {j} and J' := {j ∈ J | j has no successor in J \ S}
return constructed schedule
```

Theorem 11 ([Lawler, 1973]). *Lawler's Algorithm solves* $1 \,|\, prec \,|\, f_{\max}$ *optimally.*

Proof. Once more, we prove the theorem using an interchange argument. Let S^* be an optimal schedule. Consider the first iteration in which Lawler's Algorithm diverges from S^* and let t be the current decision point at the beginning of that iteration. Then Lawler's Algorithm selects some job j to finish at time t, whereas the optimal schedule S^* chooses job k to finish at time t, and we have $f_k(t) \geq f_j(t)$. Note that j is placed before k in S^*, since otherwise there would be an earlier iteration where the schedule constructed in Lawler's Algorithm differs from S^*.
We modify S^* to S' by placing j directly behind k, and shifting all jobs which are scheduled in between j and k, including k, for p_j time units to the left.



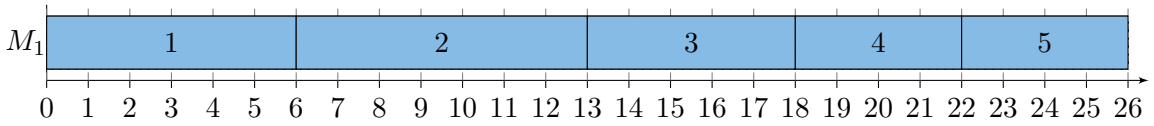
For all jobs other than job j, the completion time either reduces or remains the same when going from S^* to S'. Thus, for all such jobs $j' \in J \setminus \{j\}$, value $f_{j'}(\cdot)$ does not increase. Moreover, since $f_j(t) \leq f_k(t)$, the new $f_{\max}$-value under S' cannot be larger than the $f_{\max}$-value under S^*. Thus, S' must also be an optimal schedule. Repeating this procedure until S^* coincides with Lawler's schedule proves that Lawler's algorithm solves $1 \,|\, prec \,|\, f_{\max}$ optimally. □

We already observed above that $1 \,|\, prec \,|\, L_{\max}$ is a special case of $1 \,|\, prec \,|\, f_{\max}$. Recall that we learned in the previous lecture that a modified version of EDF solves $1 \,|\, prec \,|\, L_{\max}$ optimally. In the exercises, we ask you to think about advantages of the modified EDF algorithm versus Lawler's algorithm.

2.6 The Moore-Hodgson Algorithm

In an instance of $1 \,|\, | \sum U_j$ we are given a set of n jobs $J = \{1, \ldots, n\}$, where each job $j \in J$ has a processing time p_j and a due date d_j. Our task is to minimize the number of tardy jobs $\sum_{j \in J} U_j$, where U_j equals 1 whenever $C_j > d_j$ and 0 otherwise.

Example. Consider an instance of $1 \,|\, | \sum U_j$ where we are given 5 jobs, with processing times $p = (6, 7, 5, 4, 4)$ and due dates $d = (8, 14, 16, 17, 18)$. In the earliest due-date first schedule, which minimizes the maximum lateness, we obtain the following schedule:



In this schedule job 1 and 2 are early in the sense that they are finished before their due date. In contrast, jobs 3,4 and 5 are too late ("tardy") as they finish after their respective due dates. Thus, the schedule above has an objective value of $\sum_{j=1}^{5} U_j = 3$.

The EDF schedule we constructed in the previous example is not an optimal schedule for this instance, as there exists an optimal schedule with only two tardy jobs. Moore and Hodgson (1968) designed an algorithm which is guaranteed to return a schedule minimizing the number of tardy jobs.
The idea of the Moore-Hodgson Algorithm is to sequentially schedule the jobs in EDF order, but as soon as a job would finish too late, the job with the largest processing time among those considered so far is moved to the end of the schedule. The structure of the resulting optimal schedule is as follows. Let $E \subseteq J$ and $L \subseteq J$ denote the jobs that are on time (or early) and too late, respectively. Thus, $J = E \cup L$. All jobs in E are scheduled before all jobs in L, in the EDF order. The jobs in L are scheduled after the jobs in E have been scheduled, in arbitrary

order. Note that the selection of the set $E \subseteq J$ of early jobs completely describes the solution of $1 \mid\mid \sum U_j$. The pseudocode of the algorithm by Moore and Hodgson is given below:

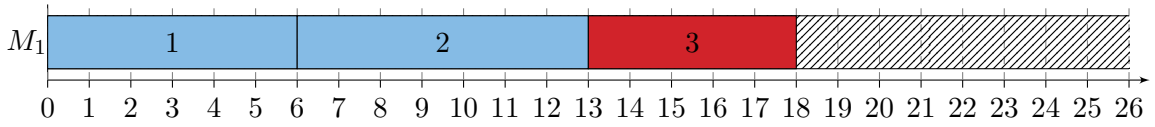
Algorithm Moore-Hodgson Algorithm:

Input: A set of n jobs J, processing time p_j and due dates d_j for each $j \in J$
Output: A schedule minimizing the number of late jobs

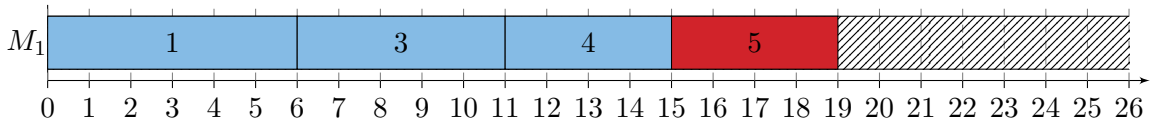
$E := \varnothing$
$\lambda := 0$
Sort and re-index jobs such that $d_1 \leq \cdots \leq d_n$
for $j := 1$ **to** n **do**
  if $\lambda + p_j \leq d_j$ **then**
    $E := E \cup \{j\}$
    $\lambda := \lambda + p_j$
  else
    Select job $k \in E \cup \{j\}$ with $k \in \arg\max_{\ell \in E \cup \{j\}} \{p_\ell\}$
    $E := (E \cup \{j\}) \setminus \{k\}$
    $\lambda := \lambda - p_k + p_j$
Schedule the jobs in E according to EDF, and the remaining jobs $L = J \setminus E$ in any order after E.

Example. Consider the instance for $1 \mid\mid \sum U_j$ as described in the previous example and find an optimal schedule using the Moore-Hodgson Algorithm. Note that the jobs are already indexed according to their due date.

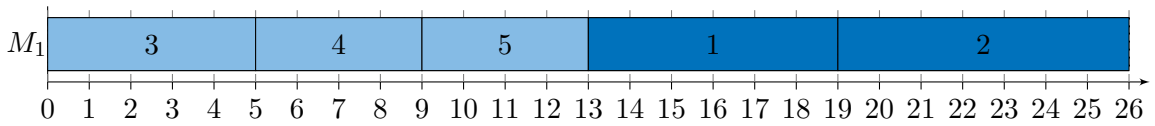
We add the jobs one by one to the schedule until a job finishes late. Note that we can schedule jobs 1 and 2 without a problem, but job 3 would arrive late.



We drop the job with the largest processing time in E from the schedule. Thus, we remove job 2 and add job 3. Again, we can add job 4 without a problem, but job 5 will be late.



Thus, we delete the job with largest processing time (job 1) from the schedule and add job 5. The jobs that got deleted in earlier iterations will be appended at the end in arbitrary order.



Theorem 12. *The Moore-Hodgson Algorithm solves problem* $1 \mid\mid \sum U_j$ *optimally.*

Proof. We omit the proof, but refer the interested reader to any standard scheduling textbook (e.g. Chapter 3 in the book *Scheduling* by Pinedo [Pinedo, 2012], or Chapter 5 in *Elements of Scheduling* `https://elementsofscheduling.nl`). □

Chapter 3

Asymptotic Running Time Analysis and Complexity Theory in a Nutshell

In this chapter, we give a short introduction into asymptotic running time analysis of algorithms and complexity theory.

3.1 Optimization and decision problems

In its most general form, we define an optimization problem as follows:

Definition 13. Given a feasibility region S and an objective function $f\colon S \to \mathbb{R}$, and the task is to find an $s \in S$ minimizing $f(s)$. That is, an *optimization problem* has the form

$$\begin{aligned} &\text{minimize } f(s) \\ &\text{subject to } s \in S. \end{aligned}$$

Note that this definition covers maximization problems as well, since any optimization problem of type $\max_{s\in S} f(s)$ can be turned into the equivalent minimization problem $\min_{s\in S} g(s)$, where $g(s) \coloneqq -f(s)$, and where the objective value of the first problem is equal to the objective value of the latter problem multiplied by -1, and vice versa.
For every optimization problem, there is an associated decision problem:

Definition 14. For an optimization problem $\min_{s\in S} f(s)$ the *associated decision problem* with target value k asks whether there exists an $s \in S$ with $f(s) \leq k$.

Clearly, any algorithm which solves an optimization problem can also be used to solve the associated decision problem. For the opposite direction observe that whenever there exists an algorithm which solves a decision problem (and even finds a solution $s \in S$ with $f(s) \leq k$), then the associated optimization problem to find the minimal value $f(s)$ for $s \in S$ (and a solution s^* with $s^* = \operatorname{argmax}_{s\in S} f(s)$) can be solved using the algorithm for the decision problem coupled with a binary search procedure to find the minimal target value k so that the answer to the decision problem is "yes". Note that we assume throughout the entire course that all input data is rational, and so (by scaling arguments) we might even assume all input data to be integral.

3.1.1 Examples of optimization and decision problems

We give two examples of optimization problems and their associated decision problems:

Example (Shortest path problem)**.** Here, the optimization problem can be described as follows: We are given a directed graph $G = (V, A)$ with vertex set V and arc set A, each arc $(u, v) \in A$ has a length $d_{u,v}$, and we are given two designated vertices, namely a start vertex s and a destination vertex t. The task is to find an s-t-path $P \subseteq A$ of minimal length $d(P) = \sum_{(u,v)\in P} d_{uv}$.
For the associated decision problem with target value $k \in \mathbb{R}$, the task is to decide whether there exists an s-t-path $P \subseteq A$ of length $d(P) \leq k$.

Example (Weighted sum of completion times on a single machine)**.** Consider the optimization problem $1 \mid \mid \sum w_j C_j$. In an instance of this problem, we are given a set of n jobs $J = \{1, \ldots, n\}$ together with processing times p_j and weights w_j for $j \in J$, and the task is to find a schedule of the jobs on one machine which minimizes the weighted sum of completion times $\sum_{j \in J} w_j C_j$. In the associated decision problem, in addition to the input data as described above, we are given a target value $k \in \mathbb{R}$, and the task is to decide whether there exists a schedule of the jobs on one machine such that $\sum_{j \in J} w_j C_j \leq k$.

3.2 Running time of algorithms

The *input size* of a problem is the number of bits needed to represent the input. Determining the exact input size of a problem is a very delicate task and needs knowledge on the difference between unary and binary encoding. For our purpose it is enough to know that the input size of a scheduling problem is approximately the number of jobs plus the number of machines, often denoted by n and m respectively.

Definition 15. Given an algorithm A, we denote by $T_A(n)$ the *running time* (the number of elementary operations) of algorithm A in the worst case on an input size of n.

There are many different elementary operations. Examples are: adding, subtracting, multiplication, division, comparing two numbers or swapping two numbers.

Example. Consider the task of finding a specific element in a sorted list or array. Let algorithm A be the algorithm that searches through all entries (from the first to the last entry of the list) until it has found the correct element, or detected that no such element exists in the list. In the worst case, this algorithm searches through the entire list. If the list contains n elements, we thus need n comparisons in the worst case.
Alternatively, one could use algorithm B, a binary search algorithm. This algorithm starts in the middle of the list. Either this is the item we are looking for, or we can dismiss roughly half of the remaining elements. Hence, in the worst case, we need $\log n$ comparisons.

The worst-case running time is usually expressed by using the Landau notation that disregards constant factors.

Definition 16. The worst-case running time function $T_A \colon \mathbb{N} \to \mathbb{N}$ of algorithm A is in $\mathcal{O}(f(n))$ for some function $f \colon \mathbb{N} \to \mathbb{N}$ if there exists a $c > 0$ and an $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$ it holds that $T_A(n) \leq cf(n)$.

Example. The running time function $T_A(n) = 30n^2 + 15n + 1$ is in $\mathcal{O}(n^2)$ (*Landau-O*, *Big-Oh*). Take, for example, $c = 40$ and $n_0 = 2$, then $30n^2 + 15n + 1 \leq 40n^2$ for all $n \geq 2$.

In the literature we find the following two notations, and you can use either one of them: $T_A(n) = \mathcal{O}(f(n))$ or $T_A(n) \in \mathcal{O}(f(n))$.
Where the Landau-O notation is used de decribe an upper bound on $T_A(n)$, the Landau-Omega notation is used to denote a lower bound on $T_A(n)$

Definition 17. The worst-case running time function $T_A \colon \mathbb{N} \to \mathbb{N}$ of algorithm A is in $\Omega(f(n))$ for some function $f \colon \mathbb{N} \to \mathbb{N}$ if there exists a $c > 0$ and an $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$ it holds that $T_A(n) \geq cf(n)$.

Example. For $T_A(n) = 30n^2 + 15n + 1$, it holds that $T_A(n) \in \Omega(n^2)$, as we can take $c = 30$ and $n_0 = 1$. Then, $30n^2 + 15n + 1 \geq 30n^2$ for all $n \geq 1$.

The Landau-Theta is used whenever $T_A(n) \in O(f(n))$ and $T_A(n) \in \Omega(f(n))$:

Definition 18. The worst-case running time function $T_A \colon \mathbb{N} \to \mathbb{N}$ of algorithm A is in $\Theta(f(n))$ for some function $f \colon \mathbb{N} \to \mathbb{N}$ if there exists $c_1, c_2 > 0$ and $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$ it holds that $c_1 f(n) \geq T_A(n) \geq c_2 f(n)$.

Example. Consider again the running time function $T_A(n) = 30n^2 + 15n + 1$. We have seen that $T_A(n) \in O(n^2)$. As we also know that $T_A(n) \in \Omega(n^2)$, we have $T_A(n) \in \Theta(n^2)$ since we can pick $c_1 = 40$, $c_2 = 30$, $n_0 = \max\{1, 2\} = 2$.

3.2.1 Enumeration

The brute-force approach is to enumerate all possible solutions in feasibility region S and select the best $s \in S$. Unfortunately, this approach takes in most cases way too long, as set S is often very large. Below, we provide a table that states the running times of several functions, assuming that the computer can do one million elementary operations per second.
Here, ✓ means that the running time is less than 1 second, and an ✗ means the running time is more than 10^{25} years. Furthermore, s is for seconds, m for minutes, h for hours, d for days and y for years.

	$f(n)$					
n	n	$n \log n$	n^2	n^3	2^n	$n!$
10	✓	✓	✓	✓	✓	$4s$
30	✓	✓	✓	✓	$18m$	$10^{25}y$
50	✓	✓	✓	✓	$36y$	✗
100	✓	✓	✓	$1s$	$10^{17}y$	✗
1.000	✓	✓	$1s$	$18m$	✗	✗
10.000	✓	✓	$2m$	$12d$	✗	✗
100.000	✓	$2s$	$3h$	$32y$	✗	✗
1.000.000	$1s$	$20s$	$12d$	$31.710y$	✗	✗
	great	good	okay...	not great	uh-oh	

3.3 Complexity Classes: P vs. NP

The complexity of a given problem is determined by the worst-case running time $T_A(n)$.

Definition 19. Algorithm A has a *polynomial running time* if

$$T_A(n) \in O(n^d)$$

for some constant degree $d \in \mathbb{N}$. In this case, we say that A is a *polynomial time algorithm* or that A is an *efficient* algorithm.

Definition 20. The complexity class P consists all decision problems which can be solved with a polynomial time algorithm.

Intuitively, this means P is the class of somewhat *easy* problems. There are many problems known to be in P. Well known problems that belong to this class are:

- Find a shortest s-t-path in a graph.
- $1 \,||\, \sum w_j C_j$
- $1 \,|\, prec \,|\, f_{\max}$

The complexity class NP is the class of problems for which there exists a *non-deterministic polynomial time* algorithm.

Definition 21. A decision problem belongs to class NP if for any given yes-instance and a certificate (whose length is upper-bounded by a polynomial), there exists a polynomial time algorithm to verify that this is indeed a yes-instance.

The difference between P and NP is the following. A problem is in P if there is a polynomial time algorithm that *solves* the problem. A problem is in NP if there exists a polynomial time algorithm that can *verify* a yes-instance.

Example. Consider an instance $1 \mid r_j \mid \sum C_j$ where we need to decide whether or not there exists a feasible schedule with $\sum C_j \leq k$ for some $k \in \mathbb{N}$. For any given schedule we can check within polynomial time whether or not it is feasible and $\sum C_j \leq k$, hence, this decision problem is in NP.

Example. Consider an instance $1||\sum w_jC_j$ where we need to decide whether or not there exists a feasible schedule with $\sum w_jC_j \leq k$ for some given $k \in \mathbb{N}$. For any given schedule we can check within polynomial time whether or not it is feasible and $\sum w_jC_j \leq k$, hence, this decision problem is in NP.

Example. If we ask whether there exists a value k such that for exactly half of the feasible non-delay schedules it holds $\sum w_jC_j \leq k$, we cannot verify a yes-instance in polynomial time. If we have given a yes-instance (i.e., a value k such that $\sum w_jC_j \leq k$ holds for exactly half of the feasible non-delay schedules) we have to check for all non-delay schedules whether $\sum w_jC_j \leq k$, but there are $n!$ (number of all possible permutations of the jobs) such schedules. Since $n!$ grows exponential, we cannot verify this in polynomial time. Thus this problem is not in NP.

Clearly $\mathsf{P} \subseteq \mathsf{NP}$. But, we do not know whether or not $\mathsf{P} = \mathsf{NP}$. Though it is widely believed that $\mathsf{P} \neq \mathsf{NP}$, there is currently no proof which supports this conjecture. If someone is able to prove or disprove this conjecture he or she earns $1.000.000, as it belongs to the Millenium-Problems, stated by the Clay Mathematics Institute.

3.4 Polynomial time reductions

In this section, we study polynomial time reductions, a.k.a. Karp reductions, which provides us with tools to compare the complexity of two problems, and to show that a certain problem under consideration belongs to the "hardest problems" w.r.t. a complexity class. We recommend to read Chapter 2 in *Elements of Scheduling* (see `https://elementsofscheduling.nl`) for more details.

A decision problem P is polynomial time reducible to decision problem P' if the existence of a polynomial time algorithm for P' implies that there exists a polynomial time algorithm for P. A formal definition is the following:

Definition 22. A decision problem P is *polynomial time reducible* to decision problem P' if there exists a polynomial time algorithm which transforms any given input instance $\mathcal{I}_P$ of P into an input instance $\mathcal{I}_{P'}$ of P' such that the answer to $\mathcal{I}_P$ for P is *yes* if and only if the answer to $\mathcal{I}_{P'}$ for P' is *yes*. When P is polynomial time reducible to P', we write $P \leq_p P'$.

We extend the complexity landscape that was introduced in the previous lecture with the class of *NP-hard* problems.

Definition 23. A decision problem P is *NP-hard* if $\widetilde{P} \leq_p P$ for all $\widetilde{P} \in \mathsf{NP}$.

Intuitively, an NP-hard problem is at least as hard as all other problems in NP. We call the NP-hard problems *NP-complete* if they are in NP.

Definition 24. A decision problem P is *NP-complete* if

1. $P \in \mathsf{NP}$ (*membership*), and
2. $\widetilde{P} \leq_p P$ for all $\widetilde{P} \in \mathsf{NP}$ (*hardness*).

Obviously, all NP-complete problems are NP-hard, but there are also NP-hard problems which are not NP-complete. Observe that if P is an optimization problem and its associated decision problem is NP-complete, then P is NP-hard.

Recall that it is not known yet whether or not $\mathsf{P} = \mathsf{NP}$ and note that it is sufficient to show that there exists one NP-hard problem P_0 that can be solved in polynomial time, as this implies that all problems in NP can be solved within polynomial time, because we could simply transform each instance of a problem that is in NP (including those that are hard) in polynomial time to an instance of P_0 and solve this in polynomial time.

We use polynomial time reductions to learn about the complexity of a problem. Suppose that $P \leq_p P'$, then:

- If P' is known to be in P, then also P is in complexity class P.
- If P is known to be NP-hard, then P' is NP-hard.
- If P is known to be NP-complete, and $P' \in$ NP, then P' is NP-complete.

To prove that a decision problem P' is NP-complete, we typically take the following 4 steps:

1. Show that Problem P' is in NP.
2. Choose a suitable NP-complete decision problem P. State an algorithm that, for any instance $\mathcal{I}_P$ of Problem P, creates an instance $\mathcal{I}_{P'}$ for Problem P'.
3. Show that your algorithm works in polynomial time.
4. Show that $\mathcal{I}_P$ is a yes-instance if and only if $\mathcal{I}_{P'}$ is a yes-instance.

For an overview of standard NP-complete problems see, e.g.,

- Michael Garey and David Johnson: *Computers and Intractability - A Guide to the Theory of NP-completeness*; Freeman, 1979.
- `http://cgi.csc.liv.ac.uk/~ped/teachadmin/COMP202/annotated_np.html`

3.4.1 Example reduction: $P2\,|\,|\,C_{\max}$ to 2-Partition

The 2-partition problem is a problem for which we know it is NP-complete.

2-Partition
Given: Numbers $b_j > 0$, $j \in J = \{1, \ldots, n\}$.
Find: Partition $L \cup R = J$ ($L \cap R = \varnothing$) such that $\sum_{j \in L} b_j = \sum_{j \in R} b_j$.

We show in the following theorem and it's proof, how to use this knowledge to show that makespan minimization on two parallel machines is NP-hard.

Theorem 25. *$P2\,|\,|\,C_{\max}$ is NP-hard.*

Proof. To prove this theorem we need to show that the corresponding decision problem is NP-complete. The decision problem associated with $P2\,|\,|\,C_{\max}$ is the following: *Given a $k \in \mathbb{R}$, a set N of n jobs with processing times p_j and two machines, does there exists a schedule such that $C_{max} \leq k$?* We refer to this decision problem as problem S, and to the 2-partition problem as problem P.
The rest of the proof consist out of four steps:

1. First, we need to show that S is in NP. For any *yes*-instance, a schedule is determined by an assignment of jobs to the machines. Given a set of jobs assigned to each of the two machines, we can verify within polynomial time whether or not $C_{max} \leq k$ We only have to check whether every job is processed by exactly one machine and no machine works on two jobs simultaneously. After that we only have to check whether for each of the two machines, the sum of processing times of jobs assigned to that machine is upper bounded by target value k.
2. Next, we state an algorithm that, for any instance $\mathcal{I}_P$ of P, constructs an instance $\mathcal{I}_S$ of S. Thus, given any instance $\mathcal{I}_P$ of P with numbers $b_j, (j = 1, ..., n)$, we create $\mathcal{I}_S$ as follows. We consider a scheduling problem on two parallel machines with n jobs that have processing time $b_j, (j = 1, ..., n)$. The decision question is whether or not there exists a schedule such that $C_{\max} \leq \frac{1}{2} \sum_{j=1}^{n} b_j$.

3. This algorithm works in polynomial time, as we only need to copy all numbers b_j, for $j = 1, ..., n$ and compute $k := \frac{1}{2} \sum_{j=1}^{n} b_j$.

4. Observe that any solution for P directly gives a schedule with $C_{\max} = k$. Conversely, any schedule with $C_{\max} = k$ (note that $C_{\max} \geq k$ for all schedules) is an assignment of numbers b_j in two sets L, R, for which we have $\sum_{j \in L} b_j = \sum_{j \in R} b_j$. □

Though partition is NP-complete, it can be solved in *pseudo-polynomial time*. This means that when the input is encoded in the *unary* notation, we are aware of an algorithm that runs in polynomial time with respect to the size of the unary encoding of the input.

Decision problems that are NP-complete, but for which a pseudo-polynomial time algorithm is known are called *weakly NP-complete*. If such a pseudo-polynomial algorithm does not exist, the problem is *strongly NP-complete*. Analogue, there are *weakly NP-hard* problems and *strongly NP-hard* problems.

Chapter 4

Scheduling on Multiple Machines

So far, we mainly discussed algorithms for single machine scheduling problems. In this chapter, we focus on scheduling problems on multiple machines. We start by showing that the *longest-processing-time-last (LPL)* rule is optimal for $P\,|\,|\sum_j C_j$. In Section 4.2, we show that the same problem on related machines, i.e., $R\,|\,|\sum_j C_j$, can be solved via reduction to a min cost max cardinality bipartite matching problem. Afterwards, we turn our attention to makespan minimization as objective function. Recall from the previous chapter that makespan minimization is NP-hard already on two parallel machines. However, we will see that makespan minimization on parallel machines is efficiently solvable if we allow preemption in the sense that processing of jobs can be interrupted at any time and resumed later on the same or another machine. We will see in Section 4.3 how to solve $P\,|\,pmtn\,|\,C_{\max}$ optimally with *McNaughton's Wrap-Around Rule*, and will afterwards see in Section 4.4 how to use this algorithm as a subroutine for turning an optimal solution of an LP into an optimal solution of the more general problem with release dates, i.e., for $P\,|\,pmtn, r_j\,|\,C_{\max}$. Finally, in Section 4.5, we present and analyze an algorithm for preemptive makespan minimization on unrelated machines ($R\,|\,pmtn\,|\,C_{\max}$).

4.1 LPL-algorithm for minimizing the the sum of completion times on parallel machines ($P\,|\,|\sum_j C_j$)

Consider an instance of $P\,|\,|\sum_j C_j$ with n jobs $J = \{1, \ldots, n\}$, m machines $M = \{1, \ldots, m\}$, and processing times $p_1, \ldots, p_n$. The task is to distribute all jobs over the identical machines such that the sum of completion times $\sum_{j=1}^{n} C_j$ is minimized. To achieve this goal, we will need to order the jobs beforehand by their processing times. To avoid technicalities, let's assume that $\frac{n}{m}$ is an integer. Note that this assumption can always be met by adding dummy jobs of 0-length until $\frac{n}{m} \in \mathbb{Z}$. Since there always exists a non-delay schedule, a schedule is fully described by an assignment of jobs to machines, together with an ordering of the jobs assigned to the same machine. That is, the scheduler needs to decide on an assignment of jobs to positions on the machines.

Definition 26. The *largest-processing-time-last (LPL) rule* schedules the m jobs of largest processing times to the m last positions, the m jobs of largest processing times among the remaining jobs to the m second-last positions, and so on.

Theorem 27. *$P\,|\,|\sum_j C_j$ can be solved optimally in polynomial time via the largest-processing-time-last (LPL) rule.*

Proof. As a warm-up, let's first consider the case with a single machine, i.e., where $m = 1$. Note that LPL is equivalent to SPT (= Shortest-Processing-Time-First), and so the optimality of SPT follows immediately by the optimality of Smith's rule, which schedules the jobs in order of non-increasing ratio of weight and processing time, for the case of unit weights. Alternatively, we could argue as follows. Suppose the n jobs are scheduled in order $1, 2, \ldots, n$. Then the completion

time of job $j \in [n]$ is $C_j = \sum_{r=1}^{j} p_r$, and so the objective is

$$\begin{aligned}\sum_{j=1}^{n} C_j = \sum_{j=1}^{n}\sum_{r=1}^{j} p_r &= p_1 + (p_1 + p_2) + \ldots (p_1 + p_2 + \ldots + p_n) \\ &= n \cdot p_1 + (n-1) \cdot p_2 + \ldots + 2 \cdot p_{n-1} + 1 \cdot p_n \\ &= \sum_{r=1}^{n} (n - r + 1) \cdot p_r.\end{aligned}$$

That is, the processing time of the job scheduled last contributes only once to the objective, the processing time of the job scheduled second last contributes twice, and so on. Thus, in an optimal schedule, the job with largest processing time is scheduled last, the job with second largest processing time is scheduled second last, and so on. This implies that LPL is optimal for $P\,|\,|\sum_j C_j$ with $m = 1$.

Let's now consider the general case, where m might be larger than 1. Clearly, if we already assigned the jobs to the m machines, then the subset of jobs assigned to machine i should be scheduled according to LPL, for each $i \in [m]$. But how should we find an optimal assignment of jobs to machines? A similar line of thoughts as for the case where $m = 1$ turns out to be helpful. For a given schedule, let us denote by p_{ik} the processing time of the job at *k-th last* position on machine i, for $k = 1, \ldots, n$. If less than k jobs are assigned to machine i, assign $p_{ik} = 0$. Then the objective is

$$\sum_{j=1}^{n} C_j = \sum_{i=1}^{m} \left(n \cdot p_{in} + (n-1) \cdot p_{i,n-1} + \ldots + 2 \cdot p_{i2} + 1 \cdot p_{i1}\right).$$

Thus, in an optimal schedule, the m largest jobs are assigned to the m last positions, the next m largest jobs are assigned to the m second-last positions, and so on. □

Example. Consider $P\,|\,|\sum_j C_j$ on $m = 3$ machines with 8 jobs $J = \{1, \ldots, 8\}$ and processing times given by $p = (2, 7, 5, 4, 3, 6, 2, 1)$. In the first round, the LPL-algorithm assigns the three jobs of processing times $7, 6$, and 5 to pairwise different machines, each of the three jobs is assigned to the last position on the individual machine. Afterwards, the three jobs of processing times $4, 3$, and 2 are assigned to the 2nd last positions on the three machines. For the objective function, it doesn't matter which of the three jobs is assigned to which machine. Only the position matters. In the last round, the remaining two jobs of processing times 2 and 1 are assigned to different machines, both on the first position. The Gantt chart below illustrates one (of multiple possible) optimal schedules.

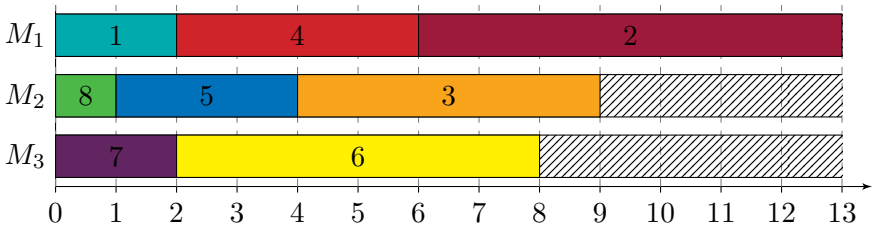


Figure 4.1: Possible Gantt chart corresponding to LPL-rule.

4.2 Solving $R\,|\,|\sum_j C_j$ via reduction to the min cost max cardinality bipartite matching problem

Consider an instance of problem $R\,|\,|\sum_j C_j$ consisting of n jobs $J = \{1, \ldots, n\}$, m machines $M = \{1, \ldots, m\}$, and processing times $p_{ij} > 0$ denoting the time job $j \in J$ requires when it is fully processed on machine $i \in M$. The task is to find a non-preemptive assignment of jobs to machines so that the sum of completion times is minimized. We already observed above that a job j scheduled last on machine i contributes p_{ij} to the objective, and a job j' scheduled

2nd-last on machine i contributes $2 \cdot p_{ij'}$ to the objective, and so on. This leads to the following observation.

Observation: Job $j \in J$ scheduled at the *kth-from-last position* on machine i contributes $k \cdot p_{ij}$ to the objective.

As a consequence, the problem to find an optimal assignment of jobs to positions on the m machines so that the objective $\sum_{j \in J} C_j$ is minimized is equivalent to the problem to find a min cost max cardinality matching in the complete bipartite graph on vertex $V = L \cup R$ where

- vertex set $L = \{v_1, \ldots, v_n\}$ contains a vertex v_j for each job $j \in J$,
- vertex set R consists of $n \cdot m$ vertices u_{ik}, where vertex u_{ik} corresponds to the kth-from-last position on machine $i \in \{1, \ldots, m\}$ for $k \in \{1, \ldots, n\}$, and
- an edge (v_j, u_{ik}) has cost $k \cdot p_{ij}$ for $j, k \in \{1, \ldots, n\}$ and $i \in \{1, \ldots, m\}$.

Theorem 28. *$R \mid\mid \sum_j C_j$ can be solved in polynomial time via reduction to a min cost max cardinality bipartite matching problem.*

Proof. By construction, a min cost bipartite max cardinality matching in the complete bipartite graph as defined above corresponds to an optimal schedule. A max cardinality matching of minimum cost in a bipartite graph $G = (L \cup R, E)$ with edge costs $c \colon E \to \mathbb{R}_+$ can be found in strongly polynomial time, for example, via reduction to a min cost flow problem in an auxiliary flow network (cf. exercises). □

4.2.1 Digression: Matchings

Let's recall the basic definitions and facts about matchings in bipartite and general graphs. Given an undirected graph, a *matching* is a subset of edges $M \subseteq E$ such that no two edges in M share a common endpoint. The MAX CARDINALITY MATCHING PROBLEM asks for a matching of maximum cardinality.

MAX CARDINALITY MATCHING
Given: Undirected graph $G = (V, E)$.
Find: Matching $M \subseteq E$ of max cardinality $|M|$.

Given costs $c \colon E \to \mathbb{R}_+$, the MIN COST MAX CARDINALITY MATCHING PROBLEM asks for a matching $M \subseteq E$ of minimum cost $\sum_{e \in M} c_e$ among all matchings M of maximal cardinality.

MIN COST MAX CARDINALITY MATCHING
Given: Undirected graph $G = (V, E)$ and edge costs $c \colon E \to \mathbb{R}_+$.
Find: Matching $M \subseteq E$ of minimum cost $\sum_{e \in M} c_e$ among all matchings M of max cardinality.

Both problems can be solved in strongly polynomial time[1] via nice combinatorial algorithms. The design and analysis of these algorithms goes beyond the scope of this course. We refer the interested reader to any standard text book on Combinatorial Optimization (e.g. Korte & Vygen *Combinatorial Optimization* [Korte and Vygen, 2011]). In the special case where the underlying graph $G = (V, E)$ is *bipartite*, i.e., where the vertex set V can be partitioned into $V = L \cup R$ with $L \cap R = \varnothing$, and where each edge has exactly one endpoint in L and in one in R the two problem above can be reduced to efficiently solvable network flow problems in auxiliary flow networks (see exercises).

[1] for details on the difference between polynomial time and strongly polynomial time algorithms, we refer the interested reader to the document "Pseudo-Polynomial Algorithms and Strong NP-Hardness" by Niklas Rieken.

4.3 Makespan minimization with preemption on parallel machines ($P \,|\, pmtn \,|\, C_{\max}$)

In this and the subsequent section, we will see that makespan minimimization on parallel machines where preemption is allowed can be solved efficiently, even with release dates.

Recall that makespan minimization on parallel machines is NP-hard, even for only two machines. However, if preemption is allowed, the problem becomes a lot easier and can be solved efficiently, even for the setting with release dates. As a warm-up, we start by investigating problem $P \,|\, pmtn \,|\, C_{max}$. The more general problem $P \,|\, pmtn, r_j \,|\, C_{max}$ with release dates will be discussed in the subsequent section.

4.3.1 Lower bounds on the optimal makespan.

Consider an instance of $P \,|\, pmtn \,|\, C_{max}$ given by n jobs $J = \{1, \ldots, n\}$ with processing times $p_j, j \in J$, that need to be scheduled on m identical machines $\{M_1, \ldots, M_m\}$. Let us have a closer look at lower bounds on the makespan for a preemptive schedule. Obviously, there exist two constraints: Firstly, the makespan can not be shorter than the total processing volume divided by the number of machines. Secondly, the longest processing time is also a lower bound on the makespan. This means, for the optimal makespan C^*_{max}, we have

$$C^*_{\max} \geq \max \left\{ \frac{1}{m} \sum_{j=1}^{n} p_j, \max_{j \in J} p_j \right\}.$$

As it turns out, the maximum of the two lower bounds $q \coloneqq \max \left\{ \frac{1}{m} \sum_{j=1}^{n} p_j, \max_{j \in J} p_j \right\}$ is in fact equal to the optimal makespan, and an optimal schedule of makespan $C^*_{\max} = q$ can be found with the following procedure, which is known under the name *McNaughton's Wrap-Around Rule*.

4.3.2 McNaughton's Wrap-Around Rule

Algorithm McNaughton's Wrap-Around Rule:

Input: A set of n jobs J, processing time p_j for each $j \in J$, m identical machines.
Output: A preemptive schedule minimizing the makespan

1 Compute $q \coloneqq \max \left\{ \frac{1}{m} \sum_{j=1}^{n} p_j, \max_{j \in J} p_j \right\}$
2 Schedule one job after the other on one virtual machine in an arbitrary order.
3 **for** $i \coloneqq 1$ **to** m **do**
4 place the ith interval, $[(i-1)q, iq]$ on the ith machine.

Theorem 29 ([McNaughton, 1959])**.** *Problem $P \,|\, pmtn \,|\, C_{\max}$ can be solved optimally in polynomial time and*

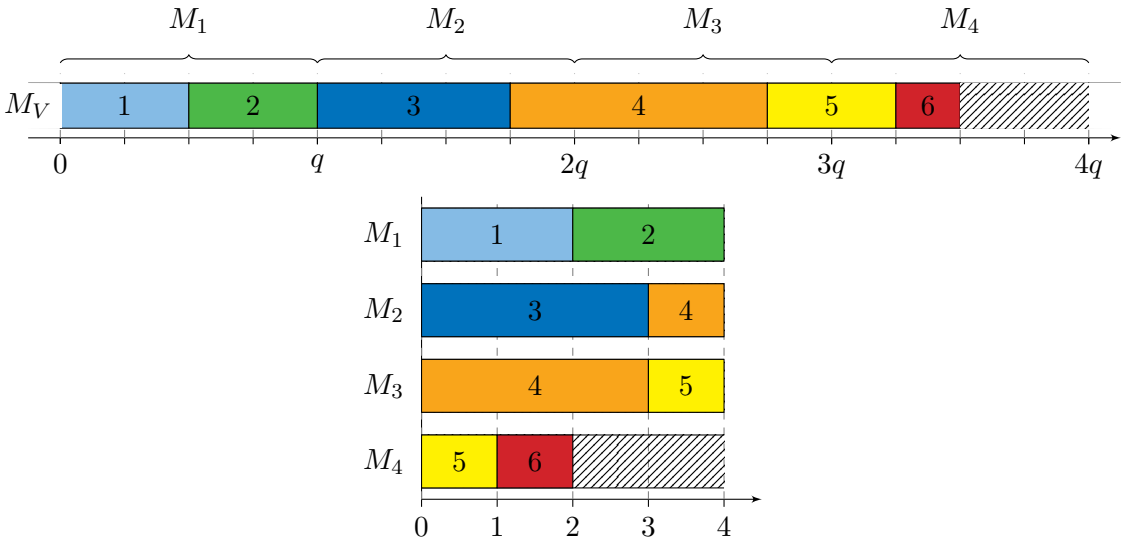
$$C^*_{max} = \max \left\{ \frac{1}{m} \sum_{j=1}^{n} p_j, \max_{j \in J} p_j \right\} =: q.$$

Proof. By the discussion above it is clear that $C^*_{max} \geq q$. It remains to show that the McNaughton Wrap-Around Rule actually achieves that bound.

Note that the rule yields at most m pieces of length q because of $q \geq \frac{1}{m} \sum_{j=1}^{n} p_j$. Thus, if the obtained schedule on m machines is feasible in the sense that no job is scheduled in parallel with itself, the makespan is equal to q.

However, because of $q \geq \max_{j \in J} p_j$, no job runs in parallel with itself because every job fits in an interval of length q. Hence, one job may end up in at most two adjacent intervals. More precisely, if a job is scheduled in $[iq, (i+1)q]$ in the one-machine schedule, the first part of it will be processed by machine M_i and the remaining part of the job by machine M_{i+1}. Moreover, the completion time of the job on machine M_{i+1} will be less than or equal to the starting time on machine M_i. As an illustration, see e.g. the orange job in the figure from the next example. □

Example. Consider the following instance with seven differently colored jobs that need to be scheduled on four machines, as illustrated in the Figure below. In the first Gantt chart, all jobs are scheduled one after each other on a single virtual machine M_V. According to McNaughton's Wrap-Around Rule, the virtual machine is afterwards "cut" into four intervals, all of length equal to q. These intervals correspond to the assignment of job-parts to the four machines.



4.4 Makespan minimization with preemption on parallel machines with release dates ($P \,|\, pmtn, r_j \,|\, C_{\max}$)

Let's now generalize the problem by allowing the jobs to have individual release dates $r_j \in \mathbb{R}_+$ for $j \in J = \{1, \ldots, n\}$. That is, we consider problem $P \,|\, pmtn, r_j \,|\, C_{\max}$. Note that McNaughton's Wrap-Around Rule won't work in general, since a preemptive schedule is feasible only if all fractions of job $j \in J$ are scheduled after or at j's release date r_j. However, as we will see below, McNaugthon's Wrap-Around Rule serves as a subroutine for solving problem $P \,|\, r_j, pmtn \,|\, C_{\max}$ efficiently.

4.4.1 LP-formulation for $P \,|\, pmtn, r_j \,|\, C_{\max}$.

We might assume that the jobs in $J = \{1, \ldots, n\}$ are ordered such that $r_1 \leq \ldots \leq r_n$. If not, we first order the jobs by non-decreasing release dates and re-index the jobs accordingly. We don't know the optimal makespan in advance, but we know that the makespan is at least $r_n + p_n$, maybe even larger. Let us introduce a variable r_{n+1} for the optimal makespan, and split the interval $[r_1, r_{n+1}]$ into n intervals

$$I_\ell := [r_\ell, r_{\ell+1}] \quad \text{for } \ell \in \{1, \ldots, n\}.$$

Beside variable r_{n+1}, we introduce n^2 variables $x_{\ell j}$ for $\ell, j \in \{1, \ldots, n\}$ to denote the fraction of p_j scheduled in interval I_ℓ. Consider the following linear program (LP).

$$\begin{aligned}
\text{minimize} \quad & r_{n+1} & & \\
\text{s.t.} \quad & \sum_{\ell=1}^{n} x_{\ell j} = p_j & & j \in J \\
& x_{\ell j} \leq r_{\ell+1} - r_\ell & & j, \ell \in J \\
& \frac{1}{m} \sum_{j=1}^{n} x_{\ell j} \leq r_{\ell+1} - r_\ell & & \ell \in J \\
& x_{\ell j} = 0 & & j, \ell \in J \text{ with } j < \ell \\
& x_{\ell j} \geq 0 & & j, \ell \in J
\end{aligned}$$

Note that the first equality together with the last non-negativity constraints require that the variables $\{x_{\ell j}\}_{\ell \in \{1,\dots,n\}}$ partitions p_j into n fractions, the second constraint ensures that each fraction $x_{\ell j}$ fits into interval $I_\ell = [r_\ell, r_{\ell+1}]$, and the third constraint ensures that the overall load of job fractions assigned to interval I_ℓ fits into the interval I_ℓ.

Theorem 30 ([Horn, 1974])**.** *The problem $P \,|\, pmtn, r_j \,|\, C_{\max}$ can be solved in polynomial time.*

Proof. Note that an optimal solution (x^*, r^*_{n+1}) of the LP above can be found in polynomial time (either by any efficient linear programming solver, or via max-flow computations as described below). Given an optimal solution (x^*, r^*_{n+1}), an optimal schedule with makespan r^*_{n+1} can be found by applying McNaughton's Wrap-Around Rule to each individual interval $I_\ell = [r_\ell, r_{\ell+1}]$ for $\ell \in \{1, \dots, n\}$ with job fractions $x_{\ell j}$ for $j \in \{1, \dots, n\}$. □

4.4.2 Solving the LP via max flow computations

Consider once more the LP-formulation for $P \,|\, pmtn, r_j \,|\, C_{\max}$ as described above. We will see that the LP can be solved via a sequence of max flow computations. The principal idea is that we search for the optimal makespan $r^*_{n+1} \in \mathbb{R}_+$ via binary search. In each iteration of such a binary search, we

- "guess" a value $\lambda := r_{n+1} \in \mathbb{R}_+$ for the optimal makespan, and
- compute a max flow x^λ in the flow network $D(\lambda)$ constructed as follows.

Construction of auxiliary flow network $D(\lambda)$: Digraph $D(\lambda)$ consists of

- one vertex $j \in J$ for each job, one vertex I_ℓ for each interval $I_\ell = [r_\ell, r_{\ell+1}], \ell \in \{1, \dots, n\}$, one source s and one sink t,
- one arc from s to each job-vertex $j \in J$ of capacity p_j,
- one arc from each job-vertex $j \in J$ to each interval vertex I_ℓ of capacity $r_{\ell+1} - r_\ell$ for each $\ell \in \{j, \dots, n\}$, and
- one arc from each interval-vertex $I_\ell, \ell \in \{1, \dots, n\}$ to t of capacity $m \cdot (r_{\ell+1} - r_\ell)$.

Observation: By construction of $D(\lambda)$, there exists a feasible schedule if and only if the optimal flow value is equal to the sum of all processing times, i.e., if and only if

$$\mathrm{val}(x^\lambda) = \sum_{j \in J} x^\lambda_{(s,j)} = \sum_{j \in J} p_j.$$

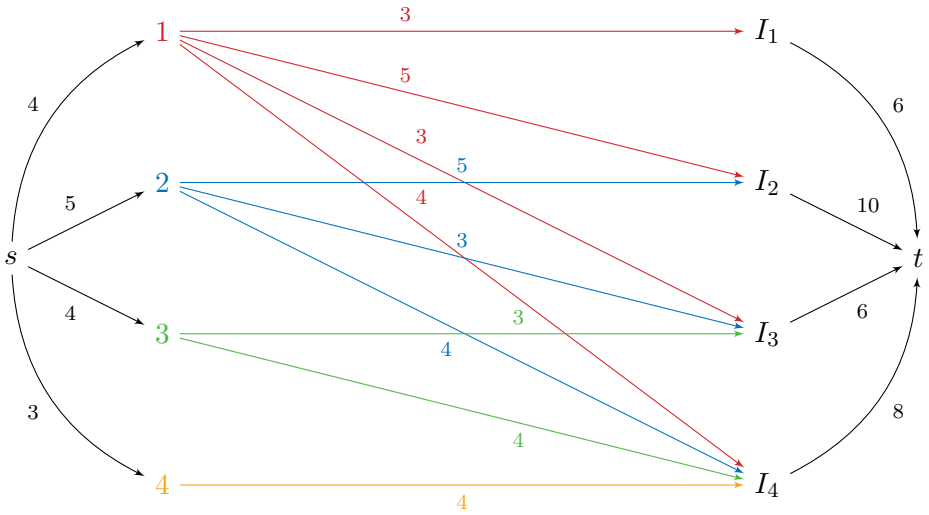
Example. Suppose we have the following 4 jobs and 2 machines.

job	1	2	3	4
p_j	4	5	4	3
r_j	0	3	8	11

Also suppose that we guessed (we usually obtain this value via binary search in the interval $[r_n + p_n, r_n + \sum p_j]$) the makespan $\lambda = r_{n+1} = 15$. We obtain the following intervals:

$$I_1 = [0, 3], \quad I_2 = [3, 8], \quad I_3 = [8, 11], \quad I_4 = [11, 15].$$

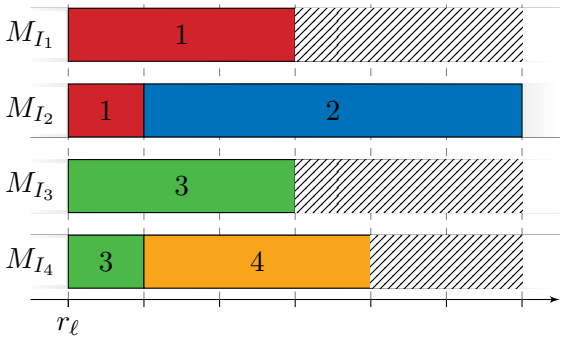
We construct the flow network $D(\lambda)$:



Via the Ford-Fulkerson Algorithm we obtain a feasible flow $f\colon E \to \mathbb{R}_+$ that fully congests all edges leaving s (we only enumerate edges from jobs to intervals here):

$$f(1, I_1) = 3, \quad f(1, I_2) = 1, \quad f(2, I_2) = 5 \quad f(3, I_3) = 3, \quad f(3, I_4) = 1, \quad f(4, I_4) = 3.$$

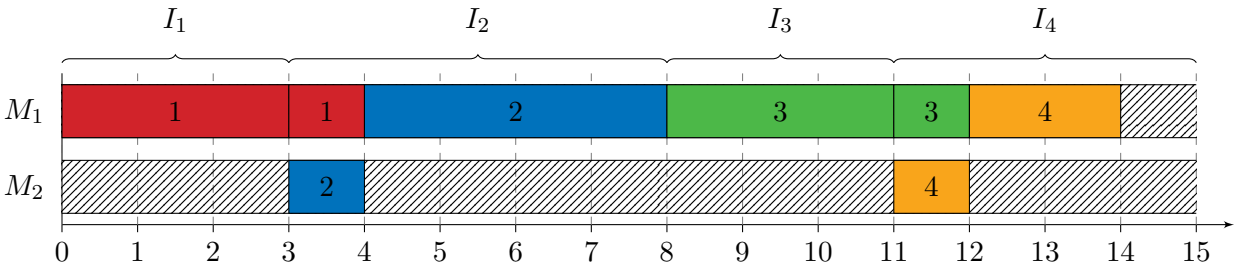
Thus, we obtain virtual machines for each interval.



We can now apply McNaughton's Wrap-Around Rule for each Interval using the following wrap points:

$$q_{I_1} = 3, \quad q_{I_2} = 5, \quad q_{I_3} = 3, \quad q_{I_4} = 3.$$

This yields to the following schedule on two machines.



As we can easily see here, the makespan is smaller than the guessed value for $\lambda = 15$. In this example, it is obvious that the optimal makespan is 14 (it cannot be lower than that, since $r_4 + p_4 = 14$ is a lower bound). In other instances we would have to apply more rounds of binary search to obtain the optimal value.

Remark: The optimal makespan $\lambda^* = r^*_{n+1}$ can in fact be found much faster using tools from *parametrized max flow*, which goes beyond the scope of this lecture.

4.5 Preemptive makespan minimization on unrelated machines ($R \,|\, pmtn \,|\, C_{\max}$)

In this and the subsequent section, we consider the problem $R \,|\, pmtn \,|\, C_{\max}$ and show that it is solvable in polynomial time. Consider an instance of problem $R \,|\, pmtn \,|\, C_{\max}$ with n jobs $J = \{1, \ldots, n\}$, m machines $M = \{1, \ldots, m\}$ and machine-dependent processing times p_{ij} denoting the processing time of job j if it is fully processed on machine i. If machine i cannot process job j at all, we model this by setting $p_{ij} = \infty$. The task is to find a preemptive schedule of jobs to machines so that each machine can only process one job at each point in time, and no job runs simultaneously on two machines. If you like, think of assigning tasks to employees over time, where the processing of a job can be interrupted at any time, and the same or an alternative employee can continue the processing at a later point in time. In the remainder of this chapter, we will present a polynomial-time algorithm for solving $R \,|\, pmtn \,|\, C_{\max}$.

Outline of the algorithm. The algorithm goes back to Lawler & Labetoulle (1978) and consists out of two steps.

(1) In a first step, we solve a linear programming relaxation of $R \,|\, pmtn \,|\, C_{\max}$ with a variable $C \geq 0$ for the optimal makespan, and variables x_{ij} to denote the fraction of each job $j \in J$ to be processed on machine $i \in M$.

(2) Afterwards, in step 2, an optimal solution (C^*, x^*) of the linear programming relaxation is turned into a feasible preemptive schedule of makespan C^* using a classical result by Birkhoff and von Neumann on decomposing a perfect fractional matching in a bipartite graph into a convex combination of integral perfect matchings.

A linear programming relaxation of $R \,|\, pmtn \,|\, C_{\max}$. Consider the following linear programming (LP) relaxation of $R \,|\, pmtn \,|\, C_{\max}$ with a variable C for the makespan, and variables $x_{ij} \geq 0$ to denote the fraction of job j that is processed on machine i. That is, job j runs $p_{ij} \cdot x_{ij}$ time units on machine i.

$$
\begin{aligned}
\text{minimize} \quad & C & & \\
\text{s.t.} \quad & \sum_{i \in M} x_{ij} = 1 & & \forall j \in J \\
& \sum_{i \in M} p_{ij} x_{ij} \leq C & & \forall j \in J \\
& \sum_{j \in J} p_{ij} x_{ij} \leq C & & \forall i \in M \\
& x_{ij} \geq 0 & & \forall i \in M, j \in J
\end{aligned}
$$

Note that this LP is in fact a relaxation of $R \,|\, pmtn \,|\, C_{\max}$, since each feasible schedule of makespan C corresponds to a feasible solution (x, C) of the LP with x_{ij} being the fraction of job j processed on i in the schedule. Now, suppose (x^*, C^*) is an optimal solution of the LP. Then, certainly, C^* is a lower bound on the optimal makespan of a preemptive schedule. However, the LP-solution (x^*, C^*) does not provide any information of the concrete schedule, i.e., the order in which a machine processes the job fractions assigned to it under x^*, and whether or not the processing of the fraction x^*_{ij} of job j that is assigned to machine i is interrupted temporarily. We will see in the proof of the following theorem how to actually convert an optimal solution (x^*, C^*) of the LP into a feasible preemptive schedule of makespan C^*.

Theorem 31 (Lawler & Labetoulle, 1978)**.** *The problem $R \,|\, pmtn \,|\, C_{\max}$ can be solved in polynomial time.*

Proof. As mentioned above, the algorithm to solve $R\,|\,pmtn\,|\,C_{\max}$ in polynomial time consists of two steps. In the first step, we solve the linear programming relaxation above. Observe that the number of variables and constraints of the LP is bounded by a polynomial in n and m, implying that an optimal solution (x^*, C^*) can be found in polynomial time.

In step 2, we take an optimal solution (x^*, C^*) of the LP and turn it into an actual schedule where no job runs simultaneously on two machines with the following procedure. First of all, without loss of generality, we might assume that $C^* = 1$. Otherwise, if $C^* \neq 1$, we scale the problem to an equivalent problem by dividing all processing times by C^*. Note that any optimal schedule for the scaled problem corresponds to an optimal solution for the original problem, and vice versa. Now, given (x^*, C^*) with $C^* = 1$, define vector y^* with values

$$y^*_{ij} \coloneqq p_{ij} \cdot x^*_{ij} \quad \text{for all } i \in M, j \in J.$$

The task is now to find a feasible preemptive schedule of makespan 1 out of y^* which processes job j on machine i for y^*_{ij} time units. Since the LP is a relaxation of $R\,|\,pmtn\,|\,C_{\max}$ with optimal objective value C^*, it follows that any feasible preemptive schedule of makespan C^* is an optimal schedule. Thus, we need to find a *timely* job-machine allocation out of y^*. Such a timely allocation can be seen as a sequence of integral job-machine matchings in the bipartite graph $G = (M \cup J, E)$ consisting of all edges $\{i, j\}$ with $p_{ij} < \infty$. To get this sequence of integral matchings in G, we use a famous result of Birkhoff and von Neumann from the 50's for decomposing a fractional perfect matching into a convex combination of integral perfect matchings. For this, we first extend the fractional matching y^* to a perfect fractional matching in some auxiliary bipartite graph as follows.

Construction of a fractional perfect matching in an auxiliary bipartite graph. First, a *fractional matching* in a graph $G = (V, E)$ is a mapping $y\colon E \to [0, 1]$ such that for all vertices $v \in V$ the constraint $\sum_{e \in \delta(v)} y(e) \leq 1$ is satisfied. A fractional matching $y\colon E \to [0, 1]$ is called *perfect fractional matching* if all constraints are satisfied with equality, i.e., if for all vertices $v \in V$ we have $\sum_{e \in \delta(v)} y(e) = 1$. Now, given our bipartite graph G on vertex set $V = M \cup J$ where the vertices in J correspond to the jobs in $J = \{1, \dots, n\}$, and the vertex set M corresponds to the machines in $M = \{1, \dots, m\}$, we extend G to a complete bipartite graph $\widetilde{G} = (\widetilde{M} \cup \widetilde{J}, \widetilde{E})$ as follows: Set $\widetilde{J}$ is obtained by adding m dummy jobs $n + 1, \dots, n + m$ to $J = \{1, \dots, n\}$, and set $\widetilde{M}$ is obtained by adding n dummy machines $m + 1, \dots, m + n$ to $M = \{1, \dots, m\}$. Moreover, $\widetilde{E}$ contains an edge for every pair $\{i, j\}$ with $i \in \widetilde{M}$ and $j \in \widetilde{J}$. As an illustration of this construction, consider Figure 4.2.

Now, given the fractional matching y^* in G obtained from our optimal LP solution (x^*, C^*) via $y^*_{ij} = p_{ij} \cdot x^*_{ij}$ for all $j \in J$, $i \in M$, extend y^* to a fractional matching

$$\widetilde{y} = \left[\widetilde{y}_{ij}\right]_{\substack{i \in \widetilde{M} \\ j \in \widetilde{J}}}$$

in $\widetilde{G}$ as follows: Set

- $\widetilde{y}_{ij} \coloneqq y^*_{ij}$ for all $i \in M$ and $j \in J$,
- $\widetilde{y}_{m+j,j} \coloneqq 1 - \sum_{i \in M} y^*_{ij}$ for all $j \in J$,
- $\widetilde{y}_{m+k,j} \coloneqq 0$ for all $j \in J, k \in J \setminus \{j\}$,
- $\widetilde{y}_{i,n+i} \coloneqq 1 - \sum_{j \in J} y^*_{ij}$ for all $i \in M$,
- $\widetilde{y}_{i,n+k} \coloneqq 0$ for all $i \in M, k \in M \setminus \{i\}$

So far, $\widetilde{y}$ is a fractional matching in $\widetilde{G}$, but not necessarily a perfect fractional matching. However, we can easily extend $\widetilde{y}$ to a perfect fractional matching by setting

$$\widetilde{y}_{m+j,n+i} \coloneqq y^*_{ij} \quad \text{for all } i \in M, j \in J.$$

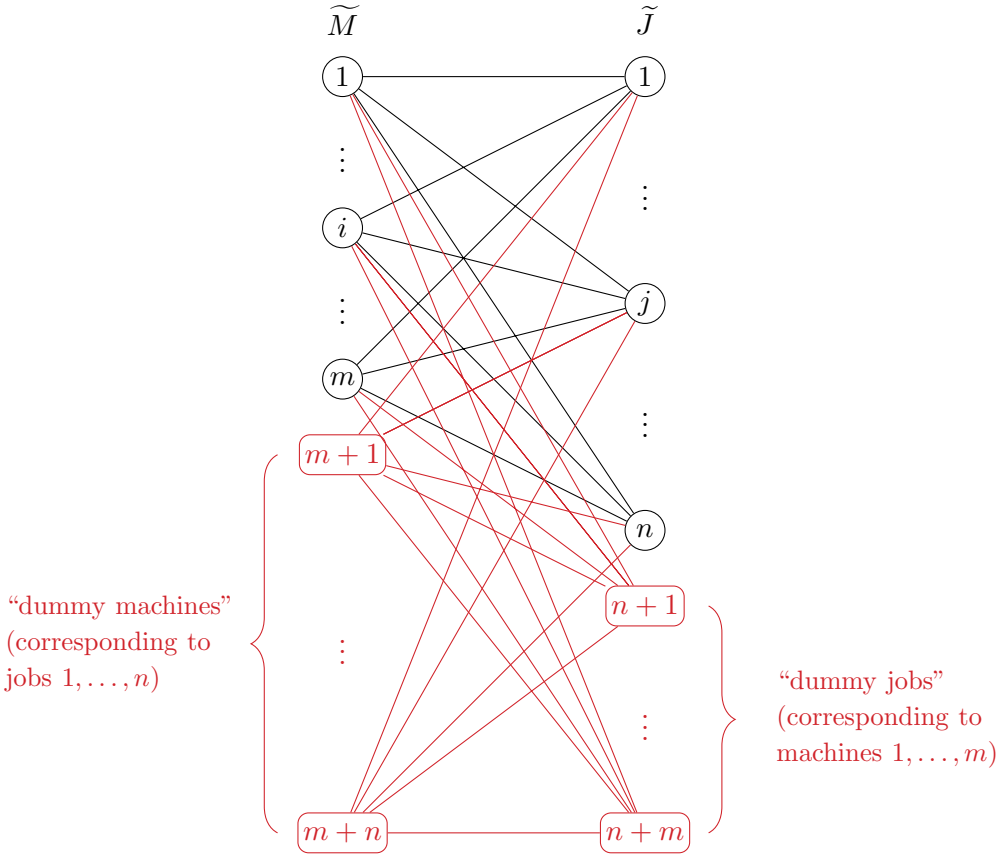


Figure 4.2: Complete bipartite graph with jobs and dummy jobs on the left and machines and dummy machines on the right.

Decomposing $\widetilde{y}$ into a timely sequence of integral perfect matchings. Now, $\widetilde{y}$ is a fractional perfect matching in auxiliary bipartite graph $\widetilde{G}$. By the Birkhoff-von Neumann Theorem (cf. Section 4.6 below), $\widetilde{y}$ can be decomposed in polynomial time into

$$\widetilde{y} = \lambda_1 \cdot \chi(M_1) + \ldots + \lambda_K \cdot \chi(M_K),$$

where each $M_k, k \in \{1,\ldots,K\}$, is an integral perfect matchings in $\widetilde{G}$ with incidence vector

$$\chi(M_k)_e = \begin{cases} 1, & e \in M_k \\ 0, & e \notin M_k. \end{cases}$$

The coefficients $\lambda_1,\ldots,\lambda_K$ are non-negative and satisfy $\sum_{i=1}^{K} \lambda_i = 1$, and $K \leq (n+m)^2$.
Now, given those integral perfect matchings M_k with coefficients λ_k, $k \in \{1,\ldots,K\}$, we construct a schedule by iteratively, for $k = 1,\ldots,K$, assigning job $j \in J$ to the machine $i \in M$ with $\{i,j\} \in M_k$ if such an edge exists (otherwise j is not assigned to any machine) for λ_k time units. Note that, in case the processing times were scaled in the preprocessing step by the factor $\frac{1}{C^*}$, in the final schedule all λ_k values, $k \in K$, need to be multiplied by C^*.
Due to the fact that all matchings M_k are integral matchings, it follows that the resulting schedule is feasible in the sense that no job is ever processed simultaneously on two machines. □

4.6 The Birkhoff-von Neumann Theorem

In this section, we prove the Birkhoff-von Neumann Theorem algorithmically. That is, we present an algorithm that decomposes a fractional perfect matching in a bipartite graph in a convex combination of integral matchings. Remember from the previous section that this algorithm is used as a subroutine to solve $R\,|\,pmtn\,|\,C_{\max}$.

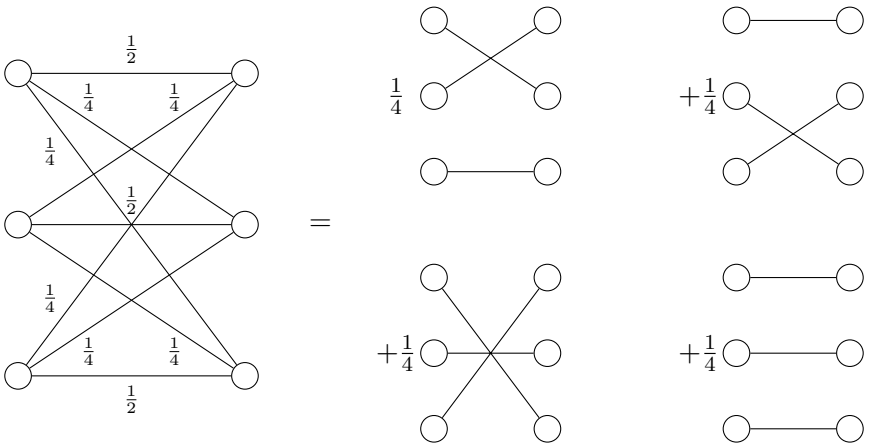


Figure 4.3: Decomposition of a fractional perfect matching as convex combination of integral perfect matching.

Let $G = (L \cup R, E)$ be a bipartite graph. Recall that $y \in \mathbb{R}^E$ is a *fractional matching* in G if

$$\begin{aligned} \sum_{i \in L} y_{ij} &\leq 1 \text{ for each } j \in R, \\ \sum_{j \in R} y_{ij} &\leq 1 \text{ for each } i \in L, \\ y_{ij} &\geq 0 \text{ for each } \{i,j\} \in E. \end{aligned}$$

If $y \in \mathbb{R}_+^E$ satisfies even $\sum_{\{i,j\} \in \delta(v)} y_{ij} = 1$ for all vertices $v \in L \cup R$, then y is a fractional *perfect* matching in G.

Theorem 32 ([Birkhoff, 1946], [von Neumann, 1953])**.** *Let $G = (L \cup R, E)$ be a bipartite graph and y be a fractional perfect matching in G. Then y can be decomposed in polynomial time into*

$$y = \lambda_1 \cdot \chi(M_1) + \ldots + \lambda_K \cdot \chi(M_K),$$

where each $M_k, k \in \{1, \ldots, K\}$, is an integral perfect matching in G with incidence vector

$$\chi(M_k)_e = \begin{cases} 1, & e \in M_k, \\ 0, & e \notin M_k. \end{cases}$$

The coefficients $\lambda_1, \ldots, \lambda_K$ are non-negative real numbers and satisfy $\sum_{k=1}^K \lambda_k = 1$, and $K \leq |E|$.

Proof. (by induction over the size of the support of y.) Let $\bar{E} \coloneqq \{e \in E \mid y(e) > 0\}$ denote the support of y, and consider the subgraph $\bar{G} = (L \cup R, \bar{E})$ of G consisting of all vertices, but only those edges in the support of y. As our induction hypothesis, we assume that the statement of the theorem holds for all fractional perfect matchings $\hat{y}$ in G of support size

$$|\hat{E}| = |\{e \in E \mid \hat{y}(e) > 0\}| < |\bar{E}|.$$

By Hall's Theorem[2], $\bar{G}$ admits a perfect integral matching $\bar{M}$, since for each set $S \subseteq L$ we have

$$|S| = \sum_{i \in S} \sum_{j \in R} y_{ij} = \sum_{i \in S} \sum_{j \in N(S)} y_{ij} \leq \sum_{j \in N(S)} \sum_{i=1}^{|L|} y_{ij} = |N(S)|,$$

where the first and last equality follow from the fact that y is a fractional perfect matching. Recall that such an integral perfect matching $\bar{M}$ in $\bar{G}$ can be found efficiently, e.g., by one max-flow computation in an auxiliary flow network.

[2]For further informations about Hall's Theorem, we refer to the notes of the discussion session or any textbook on Combinatorial Optimization (e.g. Korte & Vygen *Combinatorial Optimization* [Korte and Vygen, 2011]).

Given the fractional perfect matching y, and the integral perfect matching $\bar{M}$ in $\bar{G}$, let

$$\bar{\lambda} \coloneqq \min_{e \in \bar{M}} \{y_e\} \quad \text{and} \quad y' \coloneqq y - \bar{\lambda} \cdot \chi(\bar{M}).$$

Observe that, if $\bar{\lambda} = 1$, we are done, since $y = \bar{\lambda} \cdot \chi(\bar{M})$. Else, we have $\bar{\lambda} < 1$, and so we proceed as follows: Note that y' is a non-negative vector satisfying $\sum_{e \in \delta(v)} = 1 - \bar{\lambda}$ for all vertices $v \in L \cup R$. Since $\bar{\lambda} < 1$, we can divide any entry of y' by $1 - \bar{\lambda}$. Note that the resulting vector $\hat{y} = \frac{1}{1-\bar{\lambda}} \cdot y'$ is a fractional perfect matching in $\hat{G} = (L \cup R, \hat{E})$, where $\hat{E} = \{e \in E \mid \hat{y}_e > 0\}$, and $|\hat{E}| < |\bar{E}|$. By our induction hypthesis, we can decompose $\hat{y}$ into

$$\hat{y} = \hat{\lambda}_1 \cdot \chi(M_1) + \ldots + \hat{\lambda}_{\hat{K}} \cdot \chi(M_{\hat{K}}),$$

where each $M_k, k \in \{1, \ldots, \hat{K}\}$, is an integral perfect matching in $\hat{G}$ with incidence vector $\chi(M_k)$, the coefficents $\hat{\lambda}_1, \ldots, \hat{\lambda}_K$ are non-negative and satisfy $\sum_{k=1}^{\hat{K}} \hat{\lambda}_k = 1$, and $\hat{K} \leq |\hat{E}|$. By construction of $\hat{y}$ if follows that

$$y = (1 - \bar{\lambda}) \cdot \hat{\lambda}_1 \cdot \chi(M_1) + \ldots + (1 - \bar{\lambda}) \cdot \hat{\lambda}_{\hat{K}} \cdot \chi(M_{\hat{K}}) + \bar{\lambda} \cdot \chi(\bar{M})$$

with $\sum_{i=1}^{\hat{K}} (1 - \bar{\lambda}) \cdot \hat{\lambda}_i + \bar{\lambda} = (1 - \bar{\lambda}) + \bar{\lambda} = 1$. Since $K \coloneqq \hat{K} + 1 \leq |\hat{E}| + 1 \leq |\bar{E}| \leq |E|$, the statement of the Theorem follows. Note that all integral matchings $M_1, \ldots, M_K$, and all coefficients $\lambda_i \coloneqq (1 - \bar{\lambda}) \cdot \hat{\lambda}_i$, $i = 1, \ldots, K - 1$, and $\lambda_K = \bar{\lambda}$ can be constructed in polynomial time by applying the procedure describe above recursively. □

4.6.1 Example run of the algorithm for solving $R \,|\, pmtn \,|\, C_{\max}$

We provide a complete example of the algorithm described above for the problem $R \,|\, pmtn \,|\, C_{\max}$. We name the integral matchings that we find during this algorithm Y_1, Y_2, Y_3 in order to avoid confusion with the names of the machines M_1, M_2, M_3.

Example. Consider the following instance of $R3 \,|\, pmtn \,|\, C_{\max}$ with five jobs.

p_{ij}	1	2	3	4	5
1	5	3	2	1	3
2	4	4	1	2	2
3	4	3	4	1	2

We use an LP solver to solve the linear program constructed from these job-dependent processing times and obtain the following solution (only non-zeros):

$$\begin{array}{llll} x^*_{12} = \frac{2}{3}, & x^*_{14} = 1, & x^*_{21} = \frac{1}{4}, & x^*_{23} = 1 \\ x^*_{25} = 1, & x^*_{31} = \frac{3}{4}, & x^*_{32} = \frac{1}{3}, & C^* = 4. \end{array}$$

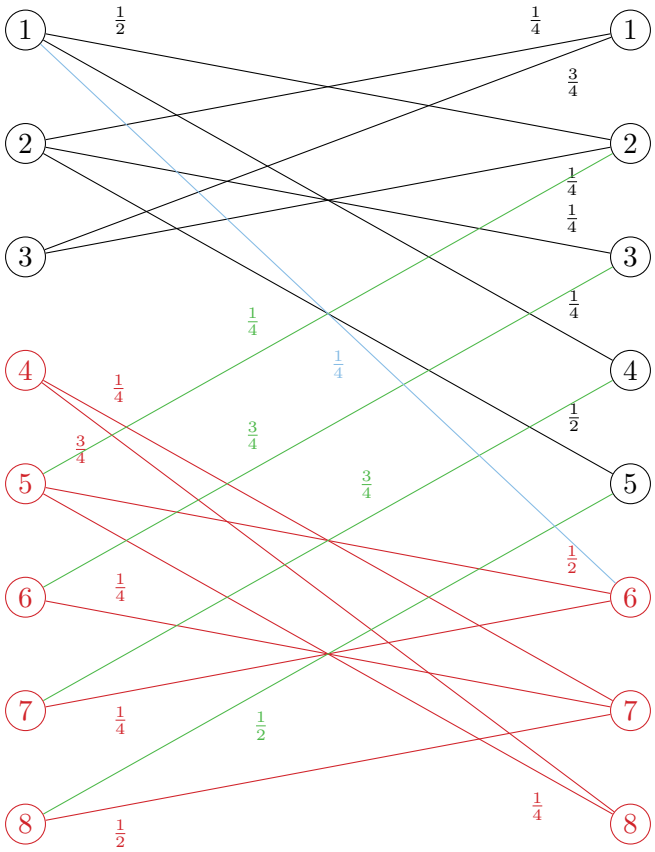
From this solution, we already know what fraction of a job j needs to be processed on each machine i (given by x^*_{ij}) and the optimal makespan (C^*). We still need to find a correct timely allocation of the jobs, that is, which job is processed at any given time t processed on which machine. We first translate the solution of the LP back into actual processing times for each machine but in order to use the Birkhoff-von Neumann Theorem, we also divide each variable by C^*. The optimal schedule we compute from that has a makespan of 1 but multiplying everything with C^* again, we obtain an optimal schedule for the original problem. More formally, the next step is to compute

$$y^*_{ij} \coloneqq \frac{p_{ij}}{C^*} \cdot x^*_{ij}$$

for all machines i and jobs j. We obtain as non-zeros:

$$\begin{array}{llll} y^*_{12} = \frac{1}{2}, & y^*_{14} = \frac{1}{4}, & y^*_{21} = \frac{1}{4}, & y^*_{23} = \frac{1}{4}, \\ y^*_{25} = \frac{1}{2}, & y^*_{31} = \frac{3}{4}, & y^*_{32} = \frac{1}{4}. & \end{array}$$

Next, we construct the complete bipartite auxiliary graph consisting of jobs and dummy jobs (red) on the left hand side and machines and dummy machines (red) on the right hand side. Since a lot of edges of the complete bipartite graph are not used, we only draw edges that have non-zero weight in the fractional perfect matching $\widetilde{y}$ that we construct from y^*. We use the dummy machine $m+j$ for job j as the *on hold* assignment (i.e. for intervals where no machine processes j). The corresponding edge has weight $1-\sum_i y^*_{ij}$. Similarly, we use the dummy job $n+i$ for machine i as the *standby* node (i.e. for intervals where machine i is idle). The corresponding edge has weight $1-\sum_j y^*_{ij}$. In order to cover all dummy jobs and dummy vertices completely (i.e. obtain a perfect fractional matching), we also add the assignment given by y^*_{ij} on the dummy jobs and machines (*fill-ups*). That is we have edges $\{i,j\}$ and $\{m+j, n+i\}$ with weight y^*_{ij}. Overall, we obtain the following perfect fractional matching, with black edges for actual job-machine assignments, green edges for on holds, light blue edges for idle times, red edges for fill-ups and 0-edges omitted:



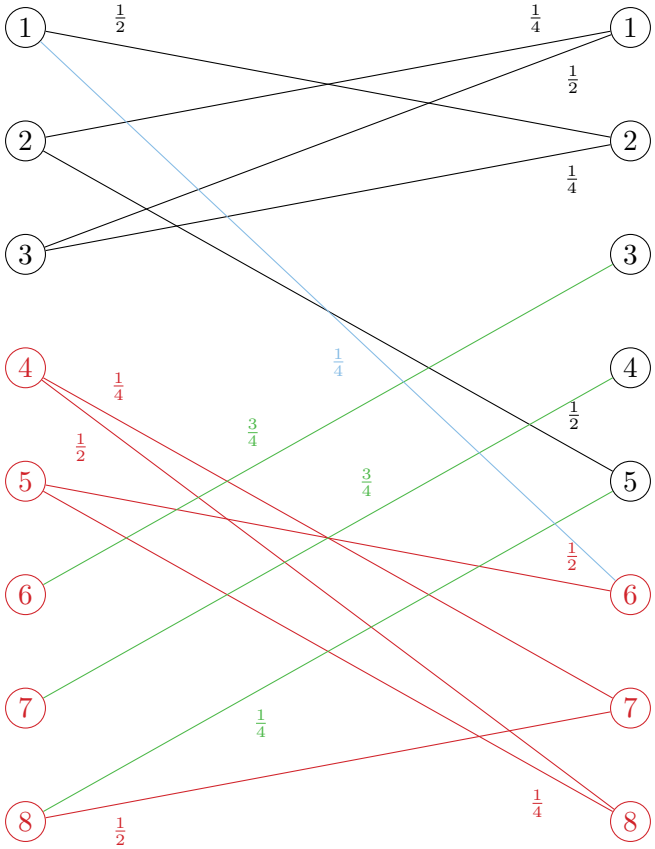
Now in each iteration k, we find an integral perfect matching Y_k in this graph (i.e. ignore weights for now) using for example the max-flow construction from earlier lectures. Moreover, we set the λ-values for each iteration to the minimum weight of all matching edges, i.e.

$$\lambda_k = \min\{w_e \mid e \in Y_k\}.$$

Let's assume we found the following matching (with the first number always representing the job or dummy job, the second number always representing the machine or dummy machine):

$$Y_1 = \{\{1,4\},\{2,3\},\{3,1\},\{4,8\},\{5,2\},\{6,7\},\{7,6\},\{8,5\}\}.$$

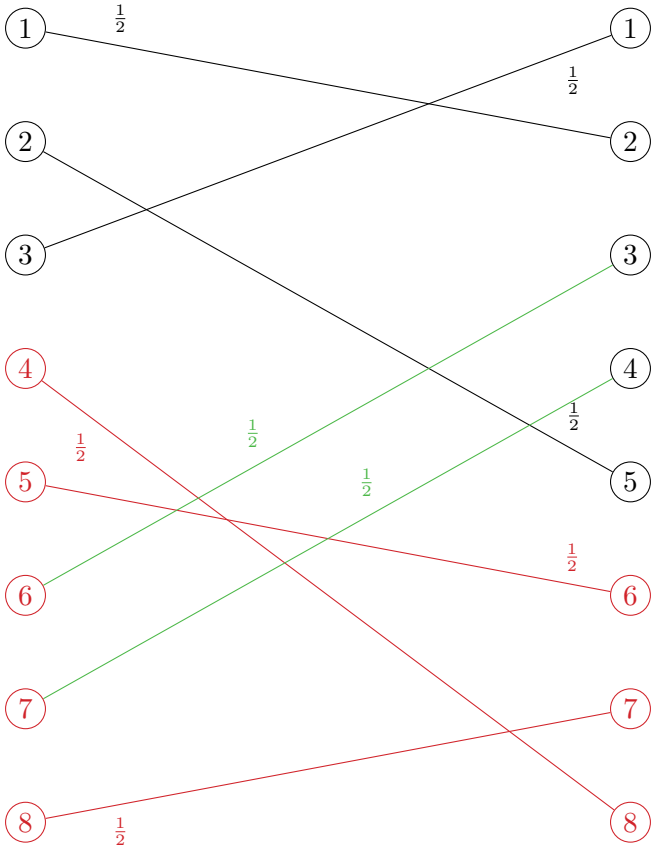
The minimum weight on all of this edges gives us $\lambda_1 = \frac{1}{4}$. Now we, reduce the weight on every matching edge by λ_1 and obtain the following graph.



We find a second matching

$$Y_2 = \{\{1,6\},\{2,1\},\{3,2\},\{4,7\},\{5,8\},\{6,3\},\{7,4\},\{8,5\}\}$$

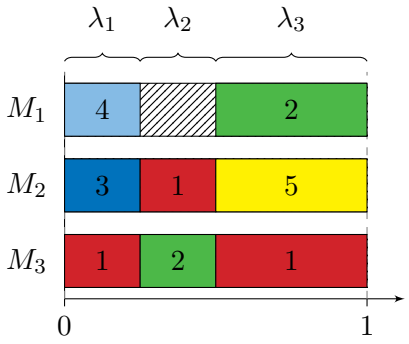
with $\lambda_2 = \frac{1}{4}$. Again reducing weight on the matching edges yields to the following graph.



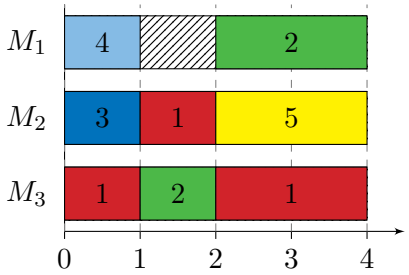
In this graph all remaining edges form a perfect matching, i.e.

$$Y_3 = \{\{1,2\},\{2,5\},\{3,1\},\{4,8\},\{5,6\},\{6,3\},\{7,4\},\{8,7\}\}$$

and $\lambda_3 = \frac{1}{2}$. After reducing, the resulting graph has only edges of weight 0. This means we are done. From the matchings Y_1, Y_2, Y_3 and the corresponding values $\lambda_1, \lambda_2, \lambda_3$, we construct an optimal schedule. Note that the order in which we apply the job-machine assignments given by the matchings is not important, but we stick with the order in that we found the matchings. We schedule each (non-dummy) job j on (non-dummy) machine i in the interval $[\sum_{\ell=1}^{k-1} \lambda_\ell, \sum_{\ell=1}^{k} \lambda_\ell]$ if $\{i, j\} \in Y_k$ for each iteration k. Matching edges that involve a dummy machine can be ignored for each iteration and if a non-dummy machine i is assigned to a dummy job, we idle machine i in this interval.



Rescaling with $C^* = 4$ from the LP-solution yields the optimal schedule of the original problem.



Chapter 5

Approximation Algorithms and List Scheduling

5.1 Approximation algorithms

Most scheduling problems belong to the class of NP-hard problems, meaning that there is no efficient (a.k.a. polynomial time) algorithm which solves the problem to optimality for any given instance of the problem, unless $\mathsf{P} = \mathsf{NP}$. How can we cope with such NP-hard problems? There are basically two ways: either we allow for algorithms whose running time cannot be bounded by a polynomial in the input size (which might be suitable for small input sizes), or we allow for algorithms that run in polynomial time, but do not necessarily return optimal solutions, but at least guarantee to return a solution that is not *too far* from an optimal solution.

Definition 33. Let $\mathcal{I}$ denote the set of all possible instances of the optimization problem under consideration. An *α-approximation algorithm* for a minimization problem is a polynomial-time algorithm that computes for any feasible input instance $I \in \mathcal{I}$ a feasible solution with cost

$$\mathrm{Alg}(I) \leq \alpha \cdot \mathrm{OPT}(I),$$

where $\mathrm{OPT}(I)$ is the cost of an optimal solution to I.
An *α-approximation algorithm* for a maximization problem is a polynomial-time algorithm that computes for any feasible input instance $I \in \mathcal{I}$ a feasible solution with cost

$$\mathrm{Alg}(I) \geq \frac{1}{\alpha} \cdot \mathrm{OPT}(I).$$

We call $\alpha \geq 1$ the *approximation factor*, or *performance ratio*.

If not stated otherwise, we restrict our considerations to minimization problems.
If we have a closer look at the definition, we are faced with the following dilemma: how can we compare against an *unknown* optimal solution for *every* instance of the problem? The key to overcome this dilemma are lower bounds on the optimal solution. A generic approach for proving that a certain polynomial-time algorithm is an α-approximation for a minimization problem is as follows:

1. Find a lower bound LB for your problem satisfying $LB(I) \leq OPT(I)$ for all $I \in \mathcal{I}$.
2. Prove that $\mathrm{Alg}(I) \leq \alpha \cdot LB(I)$ holds for the objective value of the solution returned by the algorithm for all $I \in \mathcal{I}$.

Together, 1. and 2. imply that the algorithm is an α-approximation, since

$$\mathrm{Alg}(I) \leq \alpha \cdot LB(I) \leq \alpha \cdot \mathrm{OPT}(I) \quad \forall I \in \mathcal{I}.$$

5.2 List scheduling for makespan minimization on parallel machines

In this section, we introduce a very simple algorithm, called *(basic) list scheduling*, and show that list scheduling is a $(2-\frac{1}{m})$-approximation algorithm for $Pm\,|\,|\,C_{\max}$. Moreover, we discuss the extension of the problem that allows for release dates. Recall that the problem to minimize the makespan on a set of identical machines without preemption ($P\,|\,|\,C_{\max}$) is NP-hard.

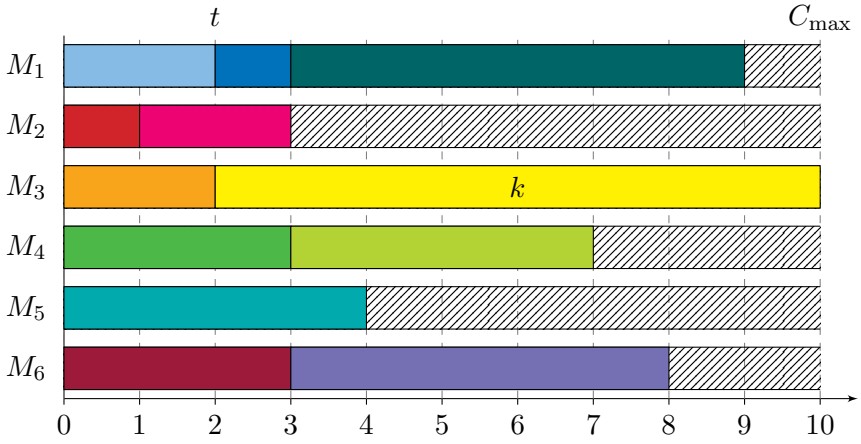
Algorithm "List scheduling" (basic version): Consider jobs iteratively in an arbitrary order. In each iteration, assign the current job to the machine with currently minimal load.

Theorem 34. *Basic list scheduling is a $(2-\frac{1}{m})$-approximation algorithm for $Pm\,|\,|\,C_{\max}$.*

Proof. It's clear that the list scheduling algorithm runs in polynomial time. Let $C_{\max}$ denote the makespan obtained by list scheduling, and $C^*_{\max}$ the optimal makespan. Note that, for any given input instance consisting of m identical machines, and n jobs $J=\{1,\dots,n\}$ with processing times $p_j, j\in J$, we have the following two lower bounds on the optimal makespan.

$$\begin{aligned} C^*_{\max} &\geq \max_{j\in J}\{p_j\}, \\ C^*_{\max} &\geq \frac{1}{m}\sum_{j\in J} p_j. \end{aligned}$$

Consider a job $k\in J$ which determines the makespan, i.e., with $C_k = C_{\max}$.



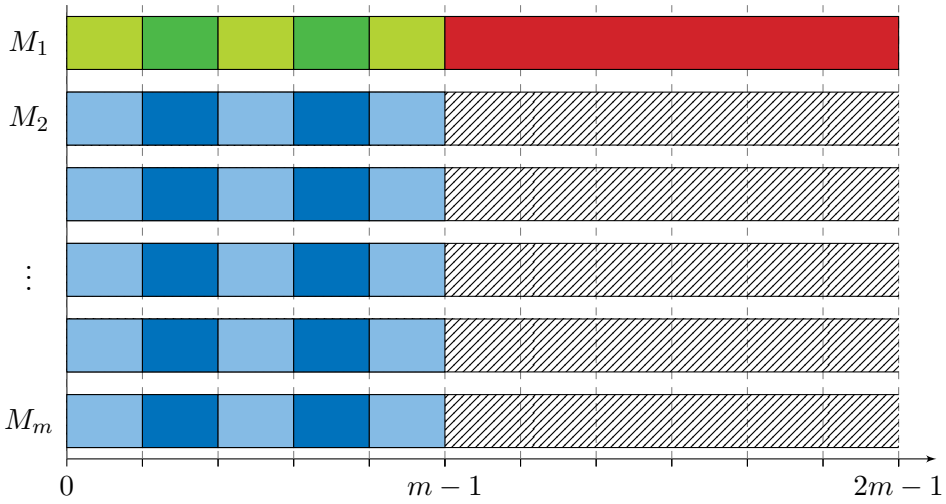
Note that our list scheduling algorithm placed job k on the currently least loaded machine with current load t of at most $\frac{1}{m}\sum_{j\in J\setminus\{k\}} p_j$. Thus,

$$\begin{aligned} C_{\max} &\leq \frac{1}{m}\sum_{j\in J\setminus\{k\}} p_j + p_k \\ &= \underbrace{\frac{1}{m}\sum_{j\in J} p_j}_{\leq C^*_{\max}} + (1-\frac{1}{m})\cdot \underbrace{p_k}_{\leq C^*_{\max}} \\ &\leq (2-\frac{1}{m})\cdot C^*_{\max}. \end{aligned}$$

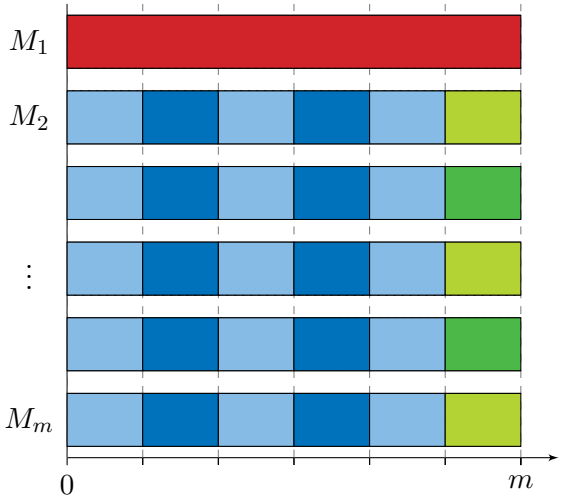
□

We note that the analysis is tight in the sense that there is indeed an instance of $Pm\,|\,|\,C_{\max}$ where basic list scheduling might return a solution that is a factor of $(2-\frac{1}{m})$ away from the optimal solution.

Example. Consider a set of $m\cdot(m-1)$ small jobs, each of which of processing time 1, and one large job of processing time m. If list scheduling assigns the jobs to the machines in an order where the large job comes last, the resulting schedule has makespan $C_{\max} = m-1+m = 2m-1$.



In contrast the optimal makespan is $C^*_{\max} = m$. Hence, $C_{\max} = (2 - \frac{1}{m}) \cdot C^*_{\max}$ on this particular instance. Note that, if jobs were ordered by non-increasing processing time (longest-processing-time-first rule), list scheduling would return an optimal solution.



A special variant of list scheduling is the *longest processing time first (LPT) rule*:

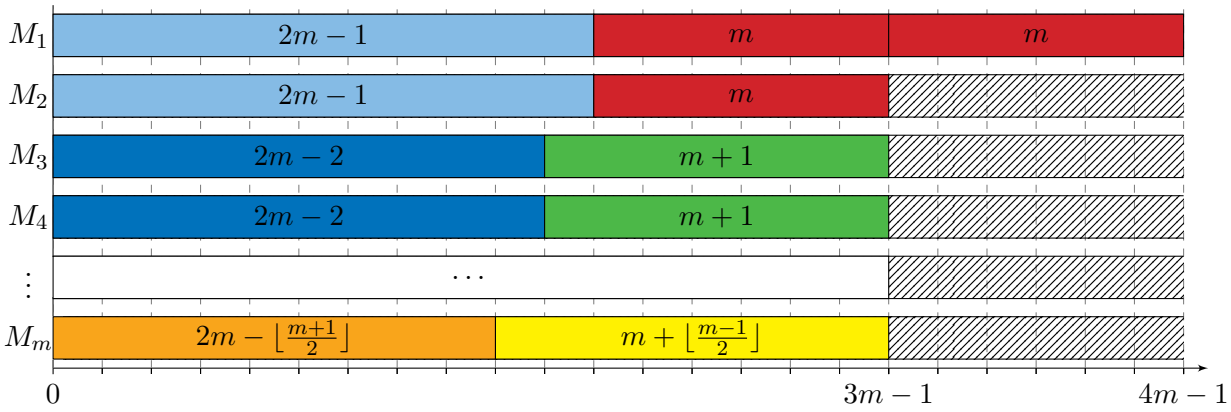
Algorithm "LPT list scheduling": Order jobs by non-increasing processing time and apply list scheduling.

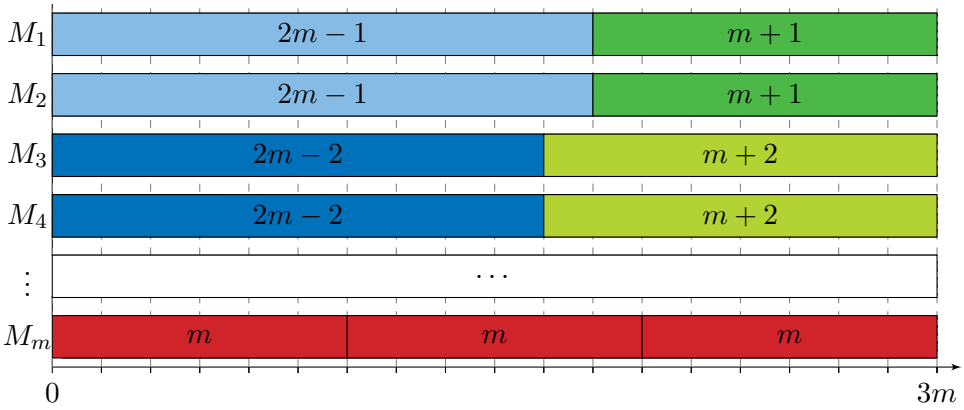
Theorem 35. *LPT list scheduling is a* $(\frac{4}{3} - \frac{1}{3m})$*-approximation algorithms for* $Pm||C_{\max}$.

Proof. The proof is omitted. For interested readers, we refer to any standard text book on scheduling (e.g., Chapter 9 in the book draft "Elements of Scheduling" on the website `https://elementsofscheduling.nl`). □

The bound in Theorem 35 is tight. There are examples where an LPT-schedule can be a factor of $(\frac{4}{3} - \frac{1}{3m})$ worse than an optimal schedule.

Example. Consider $2m+1$ jobs with processing times $2m-1, 2m-1, 2m-2, 2m-2, 2m-3, 2m-3, \ldots, m+1, m+1, m, m, m$. Then LPT returns a schedule of makespan $3m - 1 + m = 4m - 1$, while the optimal schedule has makespan $3m$ (see the two Gantt charts below).





5.3 Makespan minimization with release dates

List scheduling can easily be generalized to settings where release dates are present:

Algorithm "List scheduling with release dates": Given an arbitrary priority list of the jobs, whenever a machine becomes idle, schedule the first available job in the list among the jobs that are so far unscheduled, but already released. If no job is available, wait for the next release date.

Theorem 36. *List scheduling with release dates is a 2-approximation for* $P \,|\, r_j \,|\, C_{\max}$.

Proof. Let k be a job that finishes last, i.e., with $C_k = C_{\max}$, where $C_{\max}$ denotes the makespan of the schedule obtained by list scheduling. Let $S_k = C_k - p_k$ be the start time of job k. Notice that there is no idle time in interval $[r_k, S_k]$. Thus,

$$S_k - r_k \leq \frac{1}{m} \sum_{j \in J \setminus \{k\}} p_j \leq C^*_{\max},$$

where $C^*_{\max}$ is the optimal makespan. Since $C^*_{\max} \geq r_k + p_k$, we derive

$$\begin{aligned} C_{\max} = C_k &= S_k + p_k \\ &= \underbrace{(r_k + p_k)}_{\leq C^*_{\max}} + \underbrace{(S_k - r_k)}_{\leq C^*_{\max}} \\ &\leq 2 \cdot C^*_{\max}. \end{aligned}$$

□

A more careful analysis yields that basic list scheduling is even a $(2-\frac{1}{m})$-approximation algorithm for $Pm \,|\, r_j \,|\, C_{\max}$. We omit the proof. However, LPT provides a better performance guarantee.

Theorem 37 ([Chen and Vestjens, 1997])**.** *LPT is a* $\frac{3}{2}$*-approximation algorithm for* $P \,|\, r_j \,|\, C_{\max}$.

Proof. The proof is omitted. The interested reader is referred to the article by Chen & Vestjens, or to any standard text book on scheduling. □

We ask you in the exercises to show that the bound is indeed tight.

5.3.1 On-line makespan minimization.

Notice that LPT with release dates works as well for the *on-line* variant of $P\,|\,|\,C_{\max}$. In an on-line scheduling problem, the jobs arrive over time and only the number of machines is known in advance, but not the number of jobs or their respective processing times. Each job becomes available at its release date, which is not known in advance, and the processing time of each job becomes known at its arrival. The LPT rule now schedules at any time a machine becomes idle, an available job of longest processing time. Chen & Vestjens in their paper not only show that LPT has a performace guarantee of at most $\frac{3}{2}$, but also show that any on-line algorithm for $P\,|\,|\,C_{\max}$ will have a performance guarantee of at least 1.3473.

5.4 Makespan Minimization on Parallel Machines with Precedence Constraints

In this section, we study makespan minimization on parallel machines under precendence constraints and show that list scheduling yields a $(2-\frac{1}{m})$-approximation algorithm for $Pm\,|\,prec\,|\,C_{\max}$. Recall that precendence constraints of type $j \prec k$ among pairs of jobs $\{j,k\} \subseteq J$ require that job j must be completed before job k can start in any feasible schedule. Basic list scheduling can easily be generalized to the setting with precedence constraints:

Algorithm "List scheduling with precedence constraint"s: Take an arbitrary permutation π of the jobs. Starting at time 0, whenever a machines becomes idle, schedule the first available job in the list, where a job is *available* if it is not yet scheduled and all predecessors of which have already been completed. If none of the jobs is available, wait until the next machine becomes idle.

Remark. Problem $Pm\,|\,prec\,|\,C_{\max}$ is

- at least as hard as $Pm\,|\,|\,C_{\max}$ (thus NP-hard already for $m = 2$),
- trivially solvable for $m = 1$, and
- efficiently solvable for $m \geq n$ by the *critical path method*, described below.

5.4.1 Critical path method

The critical path method is a polynomial-time algorithm which is guaranteed to return an optimal schedule for $Pm\,|\,prec\,|\,C_{\max}$ in case there are $m \geq n$ machines available. A schedule in which each job starts at the earliest time possible in any feasible schedule is called *earliest-starttime schedule (ESS)*. An optimal schedule in which each job starts at the latest time possible in an optimal schedule is called *latest-starttime schedule (LSS)*. The two variants of critical path methods described below are guaranteed to return optimal ESS and LSS for $Pm\,|\,prec\,|\,C_{\max}$ in the case where $m \geq n$ machines are available.

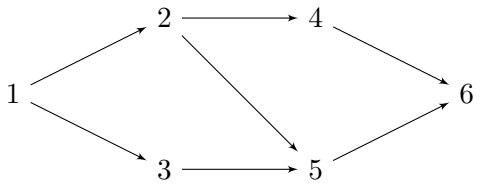
Critical path method (for ESS): Start at time 0 and schedule all jobs without predecessors on pairwise different machines. Whenever a job is completed, schedule all jobs for which all predecessors have already been completed on pairwise different machines (if there are more jobs than machines available, build a queue). This algorithm returns an *earliest-starttime schedule (ESS)*, since every job starts at the earliest possible time in any feasible schedule.

Optimality of critical path method if $m \geq n$: Note that the critical path method always returns a feasible schedule, since there are always enough machines available so that all jobs that are scheduled to start simultaneously can be scheduled on pairwise different machines. Moreover, the returned schedule is optimal since the makespan is equal to the length of a longest path in the comparability graph ("critical path"), where the length of a path is the sum of processing

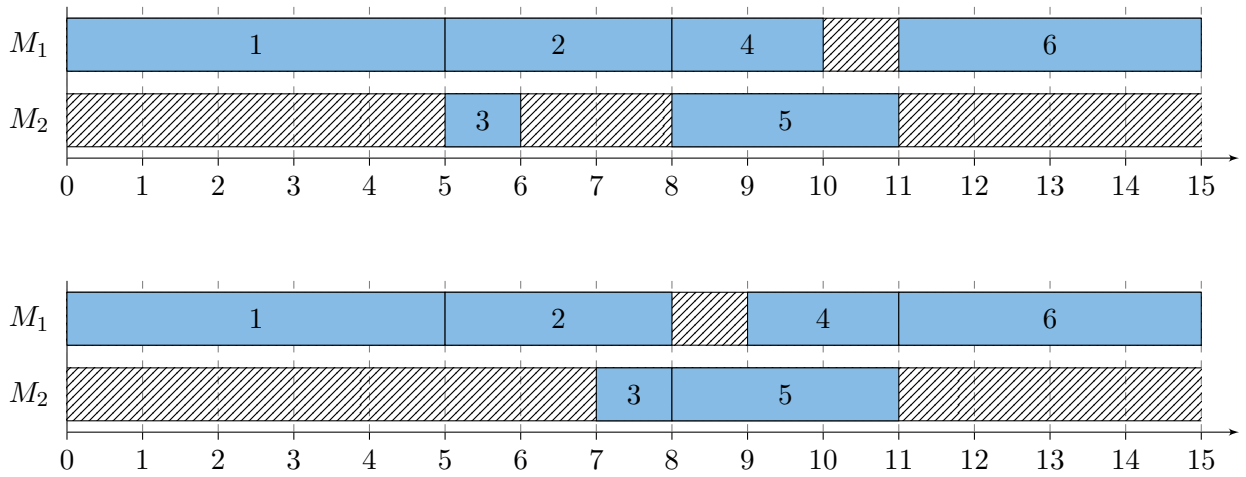
times on the path. Note that the length of a longest path in the comparability graph is a natural lower bound on the optimal makespan for $Pm \,|\, prec \,|\, C_{\max}$.
Applying the same procedure backwards, starting at the length of a longest path in the comparability graph, yields the *latest-starttime schedule (LSS)*.

Critical path method (for LSS): Starting at the length of a longest path in the comparability graph, schedule all jobs without successors on pairwise different machines backwards. Going from right to left, at any starting point of a scheduled job, schedule all jobs for which all successors have already been scheduled.

Example. Consider 6 jobs $J = \{1, \ldots, 6\}$ with processing times $p = (5, 3, 1, 2, 3, 4)$ and precedence constraints as illustrated in the following Hasse diagram.



The following two Gantt charts illustrate the earliest-starttime schedule and the latest-starttime schedule obtained by applying the critical path method forward or backward, respectively. Both schedules are in fact optimal although there are more jobs than machines. We ask you in the exercises to construct an instance where the critical path method does not yield an optimal schedule.



5.5 List scheduling for $Pm \,|\, prec \,|\, C_{\max}$ with $m < n$ machines

We show that list scheduling with precedence constraints applied to makespan minimization under precedence constraints is a $2 - \frac{1}{m}$-approximation algorithm.

Theorem 38. *List scheduling with precedence constraints is a $(2 - \frac{1}{m})$-approximation algorithm for $Pm \,|\, prec \,|\, C_{\max}$.*

Proof. Consider a job, which we call j_1, which determines the makespan in the schedule returned by our basic list scheduling algorithm. That is,

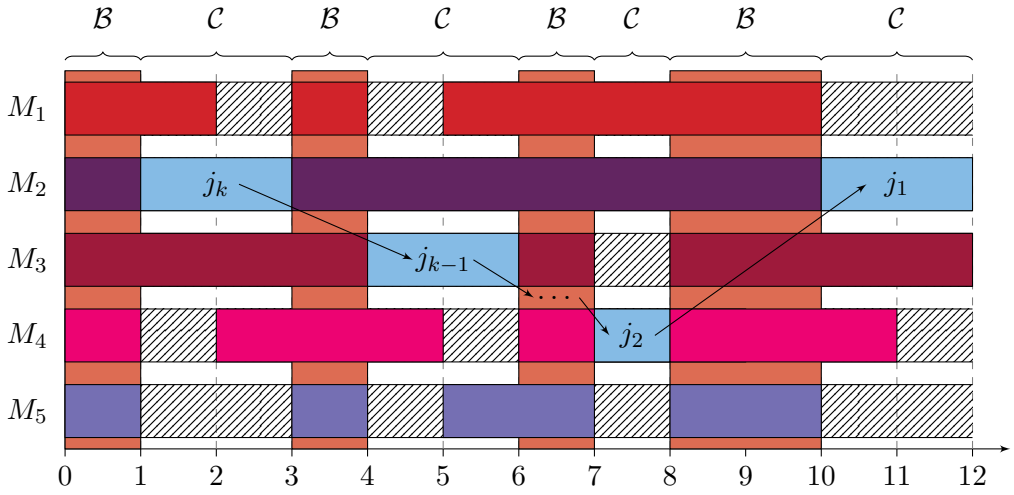
$$C_{j_1} = C_{\max}.$$

Construct a chain of jobs $j_k \prec \ldots \prec j_2 \prec j_1$ with the following procedure: Start with $i = 1$. While j_i has some predecessor, let j_{i+1} be a predecessor of j_i of latest finishing time. Note that

the chain $j_k \prec \ldots \prec j_2 \prec j_1$ constructed this way yields a natural lower bound on the optimal schedule

$$C^*_{\max} \geq \sum_{i=1}^{k} p_{j_i}.$$

Let $\mathcal{C} = \bigcup_{i=1}^{k} [S_{j_i}, C_{j_i}]$ be the union of intervals in which the jobs of our constructed chain are processed, and $\mathcal{B} = [0, C_{\max}] \setminus \mathcal{C}$.



Note that for each point in time $t \in \mathcal{B}$, all machines are busy, since otherwise one of the jobs in our constructed chain could start at time t. The makespan $C_{\max}$ can be determined by just summing up the length of all intervals in $\mathcal{C}$ and $\mathcal{B}$. For the sake of convenience let $C_{j_{k+1}} = 0$. It follows that

$$\begin{aligned} C_{\max} &= \overbrace{\sum_{i=1}^{k} (C_{j_i} - S_{j_i})}^{\text{length of } \mathcal{C}} + \overbrace{\sum_{i=1}^{k} (S_{j_i} - C_{j_{i+1}})}^{\text{length of } \mathcal{B}} \\ &\leq \sum_{i=1}^{k} p_{i_j} + \frac{1}{m} \sum_{j \in J \setminus \{j_1, \ldots, j_k\}} p_j \\ &= \frac{1}{m} \sum_{j \in J} p_j + (1 - \frac{1}{m}) \sum_{i=1}^{k} p_{i_j} \\ &\leq (2 - \frac{1}{m}) \cdot C^*_{\max}. \end{aligned}$$

□

5.6 Open shop makespan minimization

We finally consider makespan minimization for an open shop scheduling problem, i.e., problem $O\,|\,|\,C_{\max}$. Recall that in $O\,|\,|\,C_{\max}$, we are given a set of m machines and n jobs. Each job j consists of m operations o_{ij} for $i \in \{1, \ldots, m\}$ that need to be processed on the different machines, but cannot be processed simultaneously. Processing operation o_{ij} on machine i requires p_{ij} time units. Note that, if a job needs only be processed on a subset of the machines, we can model this fact by simply assigning a processing time of $p_{ij} = 0$ to all machines i on which job j needs not to be processed. The goal is to find an assignment of operations to machines minimizing the makespan. One natural way of scheduling the operations to machines is basic list scheduling, where we order the jobs arbitrarily, and, whenever a machine i becomes idle, assign the operation o_{ij} of highest priority (given by the order of the list) among the operations $\{o_{ij}\}_{j \in J}$ that have not been already scheduled on i, and can be scheduled since there is no operation of the same job that is currently processed on some other machine.

Theorem 39. *List scheduling is a 2-approximation algorithm for $O\,|\,|\,C_{\max}$.*

Proof. Clearly, the algorithm runs in polynomial time. Consider the two simple lower bounds on the optimal makespan $C^*_{\max}$:

$$\begin{aligned} C^*_{\max} &\geq \sum_{i=1}^{m} p_{ij} \quad \forall j \in \{1, \ldots, n\}, \\ C^*_{\max} &\geq \sum_{j=1}^{n} p_{ij} \quad \forall i \in \{1, \ldots, m\} \end{aligned}$$

The first lower bound describes the requirement that all operations of each job must be processed, and cannot be processed simultaneously, and the second lower bound describes the requirement that each machine needs to process all required operations. Let $C_{\max}$ denote the makespan obtained by list scheduling, and let o_{rk} denote the operation that finishes last. Note that, at any time step before $C_{\max}$, either machine r is busy by processing some job, or job k is busy being processed by some machine. Thus,

$$C_{\max} \leq \underbrace{\sum_{j=1}^{n} p_{rj}}_{\leq C^*_{\max}} + \underbrace{\sum_{i=1}^{m} p_{ik}}_{\leq C^*_{\max}} \leq 2 \cdot C^*_{\max}. \qquad \square$$

Bibliography

[Birkhoff, 1946] Birkhoff, G. (1946). Three observations on linear algebra. *Univ. Nac. Tacuman, Rev. Ser. A*, 5:147–151.

[Chen and Vestjens, 1997] Chen, B. and Vestjens, A. P. (1997). Scheduling on identical machines: How good is LPT in an on-line setting? *Operations research letters*, 21(4):165–169.

[Graham et al., 1979] Graham, R. L., Lawler, E. L., Lenstra, J. K., and Kan, A. R. (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. In *Annals of discrete mathematics*, volume 5, pages 287–326. Elsevier.

[Horn, 1974] Horn, W. (1974). Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21(1):177–185.

[Jackson, 1955] Jackson, J. R. (1955). Scheduling a production line to minimize maximum tardiness. *management science research project*.

[Korte and Vygen, 2011] Korte, B. and Vygen, J. (2011). *Combinatorial Optimization: Theory and Algorithms*. Springer.

[Lawler, 1973] Lawler, E. L. (1973). Optimal sequencing of a single machine subject to precedence constraints. *Management science*, 19(5):544–546.

[McNaughton, 1959] McNaughton, R. (1959). Scheduling with deadlines and loss functions. *Management Science*, 6(1):1–12.

[Pinedo, 2012] Pinedo, M. (2012). *Scheduling*, volume 29. Springer.

[Smith, 1956] Smith, W. E. (1956). Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3(1-2):59–66.

[von Neumann, 1953] von Neumann, J. (1953). A certain zero-sum two-person game equivalent to the optimal assignment problem. *Contributions to the Theory of Games*, 2(0):5–12.