Chapter 2

Basic Manœvers: scheduling a single server

2.1 The model and its assumptions

It's time to roll our sleeves up and do the first steps in scheduling beginning our studies with the classical single-processor problem. The scenario is quite a simple one and may be characterized by the following set of assumptions:

- There is a only one processor or server, may it be a machine, an employee, or any facility which provides some well defined service to *customers*. Traditionally, customers are also called *jobs* or *tasks* and the provision of service is called *processing* of a job.
- The processor can handle only one job at a time.
- A number n of jobs is available for service at time $t = 0$. Each job requires only one operation. Thus, initially there is a bulk of jobs forming a highly unstructured *waitingline* and it is up to the scheduler to bring some order into it.
- The processing times of the jobs are known predetermined integers and denoted by $p_1, \ldots, p_n$, each being non-negative.
- With each job we associate a non-negative integer-valued weight w_i and a *due date* d_i.
- The processor is continuously available, so in this simple model we exclude the possibility of machine breakdowns or other random and non-random events which restrict the availability of the processor.

At a first sight these assumptions seem to be very restrictive, so restrictive that this model may hardly be of any practical significance. But this is not so. Let us discuss these assumptions first.

The first two assumptions are statements about *system architecture*, we are dealing with a single server system and this server is not capable of *multitasking*. Typical examples of single processor systems are one-man businesses in crafts and trade or personal services. But we may also think of more complex production and service systems consisting of several single server units where each may be analyzed separately for various reasons. One good reason for an

isolated analysis of a single-server unit may be that it acts as a bottleneck in some system severely dominating other production units.

And last, but not least, the single processor system is an important training ground to become acquainted with the basic ideas and methods of scheduling. That's why this chapter is entitled *basic manœvers*.

Our third assumption needs a little bit of explanation: in many real-world service systems jobs arrive at the system one after the other, or in groups of varying size. The time instant of an arrival of a job is called its *release date*, usually denoted by r_i. In this chapter we assume that all release dates are zero. We do this for sake of simplicity. Indeed, systems where jobs have different release dates are pretty difficult to analyze, therefore we postpone their discussion to Chapter 7. Still, the assumption that all jobs are available for service at time $t = 0$ holds good in many situations.

Think of a company where service or production orders are coming in over night by email. In the morning a dispatcher makes a list of jobs to be done next for each of the employees. Another example comes from emergency medicine: consider a bus accident. The emergency physician arriving with the first ambulance is confronted with several injured people at once. In this critical situation several decisions have to be made within a very short time, notably, which casualty should be treated first.

Another example is *gated service*: a stream of jobs is arriving at a service station and stopped by a gate. From time to time the gate opens and gives access to, say, n jobs. Then the gate closes and remains closed until these n jobs have been served. In the mean time new jobs arrive and form a waitingline in front of the gate. Such systems are not uncommon in communication systems or in traffic control, e.g. when highways have tunnels and incoming traffic is separated into blocks of cars for reasons of safety.

The job weights serve mainly two purposes in this chapter. They may represent *holding cost* per job and unit of time. Alternatively, quite often these weights are the result of a *ranking* or *scoring* reflecting the relative importance or priority of a particular job compared to others. A typical example is the process of *triage* in emergency medicine. When medical personnel arrives at the site of a mass-casualty accident or a disaster like an earth quake, it may be necessary to assign casualties priorities for further treatment.

Due dates are special: these are *time instances* at which jobs *should be finished*. If we explicitly take into account due dates then weights are no longer interpreted as holding cost but as cost incurred by a job not finishing on time. Due dates will be the major topic in sections 2.5 and following.

2.2 The schedule and its outcome

First, let us label our jobs by giving them numbers. It will be convenient to denote the set of Jobs by $J = \{1, 2, \ldots, n\}$ with J_i denoting job number i.

A *schedule* is an assignment of a certain number of *time slots* or *slices* of processor time $A_i(1), A_i(2), \ldots$ to each job J_i, such that

- the processor works exclusively during such a slot on that job to which this slot has been assigned.
- These time slots do not overlap. Remember, the processor can handle only one job at a time.
- The total length of all slots assigned to job J_i is as least as large than p_i, the processing time of J_i.

An assignment of slots which conforms to these assumptions is called a *feasible schedule*. Observe that there are infinitely many feasible schedules for our problem.

Let us give an example.

▼ Example 2.1

Consider a single processor problem with three jobs having processing times $p_1 = 9$, $p_2 = 5$ and $p_3 = 4$. A possible feasible schedule may be this one:

$$\begin{array}{llll} \text{Job 1:} & A_1(1) = [6,9], & A_1(2) = [13,17], & A_1(3) = [18,20] \\ \text{Job 2:} & A_2(1) = [11,13], & A_2(2) = [22,25] & \\ \text{Job 3:} & A_3(1) = [0,6] & & \end{array}$$

The *Gantt-chart* of this schedule is quite informative, see Figure (2.1).

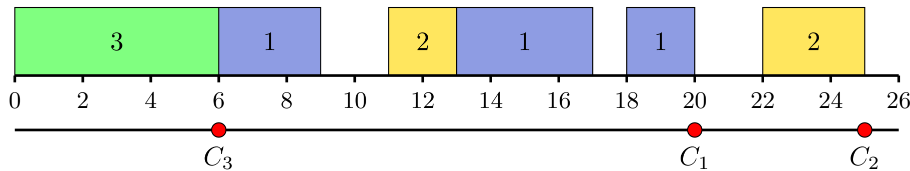


Figure 2.1

This example leads us to some important definitions.

Completion Times. The completion time C_i of job J_i is defined as the time instant when processing of job J_i is ultimately finished so that this job can be delivered. In our example we have $C_1 = 20, C_2 = 25$ and $C_3 = 6$. The completion times of jobs are data generated by the schedule, indeed, the most important ones. More precisely, completion times are *functions of the schedule*.

Makespan. This is denoted by $C_{\max}$ and is defined as the time when the last job finishes. In other words:

$$C_{\max} = \max(C_1, C_2, \ldots, C_n). \tag{2.1}$$

In our example we have $C_{\max} = 25$. By its definition $C_{\max}$ is a *function of the completion times* produced by a schedule.

Inserted Idle Time. When looking at Figure 2.1 you will notice that there are *holes* in the Gantt chart. Actually, we can identify three such holes, the intervals $(9, 11)$, $(17, 18)$ and $(20, 22)$. During these time intervals the processor is doing nothing, it is *idle* and the corresponding intervals are called *inserted idle time (IIT). IIT* may be a result of particular circumstances like a machine breakdown, then it is called *forced IIT*. This temporal unavailability of the processor due to technical conditions is excluded by our assumptions, since we have assumed that the processor is permanently available. However, *IIT* may also be *unforced*, which means that it has been decided deliberately by the scheduler to keep the machine down for a while and therefore keep jobs waiting. This raises immediately the question: *Why should we do this? Does it make any sense?* These questions are legitimate, since normally we want our work to be finished as soon as possible.

Preemption. Figure 2.1 shows another interesting feature of this schedule: Even if we disregard *IIT*, the processing of jobs J_1 and J_2 appears to be discontinuous in the sense that work on job J_1 is interrupted, then the processor works on job J_2, then it returns to job J_1 to finish it. Later it resumes processing of job J_2. Such interruptions of service are called *preemption*. There are several ways preemption is implemented in production and service systems. The two most important ones are:

- *Preemptive-Resume.* In this model the processing of a job can be continued at the point where it has been suspended, there is no loss of processing time. A typical example is downloading a large file from a web server. In order to avoid *starvation* of other concurrent client requests the server will interrupt the download process from time to time and provide service to other clients. Later it will return to a suspended job and continue transmitting data.
- *Preemptive-Repeat.* Here once processing of a job has been interrupted it has to be *restarted from scratch*. Thus this interruption leads to a total loss of processing time and other resources so far used by a job. For example in metallurgy it is often necessary to purify metals by melting them at very high temperatures in electric ovens. If a breakdown of electric power supply occurs before the metal has achieved the required level of purity it will cool down very quickly and the whole process of heating must be restarted once the problem has been fixed.

In Example 2.1 we used a preemptive-resume policy.

As with *IIT*, the question arises: *Is it necessary to preempt jobs?*

The answers to these questions depend on another very important question: *What is a good schedule?*

2.3 Performance Measures

To assess the quality of a schedule it is common to transform the the n-dimensional vector of completion times onto a 1-dimensional utility scale by *a performance measure*. The latter is a function f of the completion times $C_i, i = 1, 2, \dots, n$:

$$z = f(C_1, C_2, \dots, C_n). \tag{2.2}$$

f is called a *regular performance measure*, it

- f has to be *minimized*.
- f is an *increasing* function of each of the completion times C_i. More formally:

$$f(C_1, C_2, \dots, C_i + \delta, \dots, C_n) \geq f(C_1, C_2, \dots, C_n),$$

  for any $\delta > 0$ and any $i = 1, 2, \dots, n$. In other words, if any or some completion times increase, then f also increases or at least stays at the same value.

Remark. In this book we use a less pedantic language as it is common in scheduling. We say: a sequence of numbers a_i is increasing, when

$$a_1 \leq a_2 \leq \dots \leq a_n,$$

and we say, this sequence is *strictly increasing*, when

$$a_1 < a_2 < \dots < a_n,$$

Regular performance measures play a predominant role in scheduling theory. Most performance measures discussed in this book are regular. Actually, we have already seen one, the makespan $C_{\max}$. Indeed,

$$C_{\max} = \max(C_1, C_2, \dots, C_n)$$

is an increasing function in each completion time C_i. And of course, from a management point of view, it makes sense to find a schedule that minimizes $C_{\max}$, because we want our work to be done as soon as possible. The smaller $C_{\max}$, the earlier the processor will be free for the next bulk of jobs to be processed. Thus by minimizing $C_{\max}$ we maximize the *throughput*, the number of jobs finished per unit of time and thereby we optimize the *utilization* of our production facility.

There are also non-regular measures, we will learn about one of them in Section 2.4.4. A more detailed discussion of these measures will follow in Chapter 7.

Now we arrived at a position where it is possible to answer the question whether it makes any sense to insert idle time or to preempt jobs. It turns out that in our single processor model with all jobs available at time zero we need not consider schedules with inserted idle time or preemption when assessing the quality by a regular performance measure.

Removing inserted idle time will reduce at least one completion time and therefore the performance measure may be reduced in value, at least it cannot become larger. See Figure 2.2 shows the schedule of Example 2.1 without inserted idle time. In the new schedule the completion times of jobs J_1 and J_2 are reduced

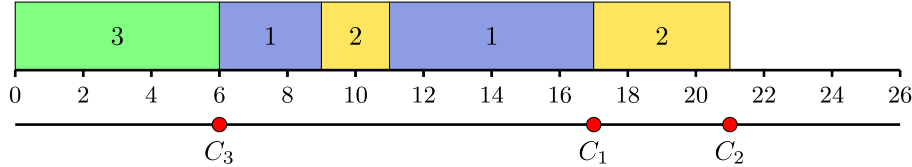


Figure 2.2

to $C_1 = 17$ and $C_2 = 21$ while still $C_3 = 6$.

To remove preemption we interchange the first block of job J_2 and the second block of job J_1, the result is shown in Figure 2.3. This new schedule reduces

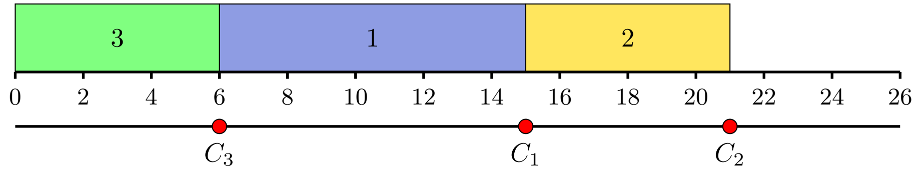


Figure 2.3

once more the completion time of J_1 to $C_1 = 15$.

At this point another interesting observation can be made. The original schedule of Example 2.1 required in total the specification of six time slots. After removal of inserted idle time and preemption the schedule in Figure 2.3 requires only *the specification of the order* in which jobs are processed.

Thus we may specify this schedule as a simple list of job labels:

$$S = [3, 1, 2]$$

But this is just a *permutation* of the labels $1, 2$ and 3.

Let us pause for a moment[1] and review, what we have found out so far:

- If we allow for inserted idle time and preemption, then there is an infinite number of possible schedules for any problem instant of our model.
- This set of schedules contains a *finite subset* determined by all possible permutations of job labels. These schedules do not have preemption and inserted idle time. They are called *permutation schedules* and there are $n!$ of them, as there are $n!$ possibilities to arrange n distinguishable items in different ways.

[1] So, let us insert some idle time!

- The permutation schedules form a *dominant set* in the sense that with respect to any regular performance measure there cannot exist a better non-permutation schedule.
- So, whatever regular performance measure we consider, our search for an optimal schedule may be restricted to the dominant set of permutation schedules. Thus all that matters is the *order* in which jobs are processed. Note that we have reduced an optimization problem with an infinite number of feasible solutions to one with only a finite number.

Remark: The reader may recall our bicycle-example from Chapter 1. There we mentioned two *busy rules* formulated by the company's management. These rules simply forbid inserted idle time and job preemption. In the single machine setting discussed in this chapter those rules can be justified. But the bicycle example is a very different scheduling problem: there is more than one processor and jobs are dependent. So we may suspect that the bicycle problem could be solved easier when we allow for inserted idle time and/or preemption.

When dealing with permutation schedules we shall always represent a particular permutation as an *ordered list*. For example, when $S = [3, 1, 5, 2, 4]$, then we process job J_3 followed by J_1, etc. It will be also convenient to use the special notation[2]

$$(i) = \text{number of job in position } i \text{ of } S, \tag{2.3}$$

Again, when $S = [3, 1, 5, 2, 4]$, then

$$J_{(1)} = J_3,\ J_{(2)} = J_1,\ J_{(3)} = J_5,\ J_{(4)} = J_2,\ J_{(5)} = J_4$$

Unfortunately, the $(\cdot)$-operator is somewhat clumsy, typographically as well as for reading. Therefore, from time to time we will deliberately *renumber* jobs so that $(i) = i$. The reader will be indicated when we do so, thus there will be no danger of confusion.

As remarked earlier in this section, *completion times* are the most important data generated by a schedule. Thus we need some way to calculate the C_i for a given permutation schedule S. This is pretty easy. Because jobs are never interrupted and there is no inserted idle time, the completion times can be determined by a simple recurrence formula:

$$C_{(i)} = C_{(i-1)} + p_{(i)}, \qquad C_{(0)} = 0 \tag{2.4}$$

The reasoning behind this formula is that: the first job in the schedule starts processing immediately, its processing time being $p_{(1)}$, so it finishes at

$$C_{(1)} = p_{(1)}.$$

Once having completed job $J_{(1)}$ the processor continues without delay with the second job in the schedule, $J_{(2)}$. It will be finished at time

$$C_{(2)} = C_{(1)} + p_{(2)}$$

[2]Technically speaking, this notation refers to the *inverse* of a permutation.

Continuing this argument yields the recurrence (2.4) which can be solved explicitly:

$$\begin{aligned} C_{(1)} &= p_{(1)} \\ C_{(2)} &= C_{(1)} + p_{(2)} &&= p_{(1)} + p_{(2)} \\ C_{(3)} &= C_{(2)} + p_{(3)} &&= p_{(1)} + p_{(2)} + p_{(3)} \\ &\dots \\ C_{(n)} &= C_{(n-1)} + p_{(n)} &&= p_{(1)} + p_{(2)} + \ldots + p_{(n)}. \end{aligned}$$

Thus we have found that the completion times are simply the *cumulative sums* of the processing times, added in the order determined by a schedule S:

$$C_{(k)} = p_{(1)} + p_{(2)} + \ldots + p_{(k)} = \sum_{\ell=1}^{k} p_{(\ell)} \tag{2.5}$$

▼ Example 2.2

Suppose there are 5 jobs to be processed, so that $J = \{1, 2, 3, 4, 5\}$. The processing times (in minutes) are given by:

J	1	2	3	4	5
p	8	3	10	5	4

Let $S = [3, 1, 5, 2, 4]$, then by (2.5) we obtain:

S	3	1	5	2	4
p	10	8	4	3	5
C	10	18	22	25	30

Correspondingly, we have found that

$$C_1 = 18, \quad C_2 = 25, \quad C_3 = 10, \quad C_4 = 30, \quad C_5 = 22$$

Note also that this schedule S has makespan $C_{\max} = 30$.

Closely related to completion time is the *waiting time* W_j of a job. It is defined as the time a job has actually to wait before its service can commence. Since in our simple model all jobs are available at $t = 0$, we have

$$W_j = C_j - p_j \tag{2.6}$$

Thus there is also a very simple formula for the calculation of waiting times which follows from (2.5):

$$W_{(k)} = p_{(1)} + p_{(2)} + \ldots + p_{(k-1)} = \sum_{\ell=1}^{k-1} p_{(\ell)}, \quad \text{for } k > 1 \tag{2.7}$$

with $W_{(1)} = 0$, obviously, since the first job in any schedule need not wait at all.

Now the stage is prepared to study some important performance measures in more detail. One measure, however, can be disposed off quickly, the *makespan* $C_{\max}$. For any permutation schedule

$$C_{\max} = C_{(n)} = \sum_{i=1}^{n} p_{(i)} = \sum_{i=1}^{n} p_i \tag{2.8}$$

Thus $C_{\max}$ is independent of any schedule S, since the value of a sum does not depend on the order of summation. $C_{\max}$ is also called the *load* of the processor and, as remarked earlier, it is mostly used to measure the *utilization* of service facilities in more complex systems. As we have only one processor, there is no inserted idle time and all jobs are available at time zero the processor's utilization is always 100 %.

The maximum waiting time $W_{\max}$, however, depends on the schedule, in particular on the processing time of the job put onto the last position in S:

$$W_{\max} = C_{\max} - p_{(n)}. \tag{2.9}$$

2.3.1 Exercises

1. Consider $n = 10$ jobs with processing times

J	1	2	3	4	5	6	7	8	9	10
p	12	2	8	16	12	15	14	13	15	16

Determine for the schedule $S = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ completion times, waiting times, makespan and maximum waiting time. Calculate arithmetic mean, median and standard deviation of the completion times.

2. (continued) How do these statistics change, if you consider instead of S the schedule $S' = [2, 3, 1, 5, 8, 7, 6, 9, 4, 10]$?

3. Given n jobs, find all schedules minimizing $W_{\max}$.

4. Define a performance measure

$$f(C_1, \ldots, C_n) = \sum_{i=1}^{n} w_i(1 - e^{-rC_i}),$$

where the w_i are non-negative job weights and r denotes a discount rate. Show that f is a regular measure of performance.

5. Let $d > 0$ be a constant. Is

$$f(C_1, \ldots, C_n) = \sum_{i=1}^{n} |C_i - d|$$

a regular performance measure?

Hint: Use the fact that the function $g(x) = |x - a|$ is differentiable everywhere except for $x = a$. Find its derivative $g'(x)$.

2.4 Total completion time

2.4.1 The *WSPT*-Rule

When developing mathematical models to solve real-world problems, it is always a good idea to talk to practitioners like dispatchers and production managers, since those people are solving scheduling problems all the time. If not to optimum they have often available surprisingly easy to use rules. Two *Golden Rules of Scheduling* are these:

- *Rule 1:* Avoid unnecessary waits of your customers.
- *Rule 2:* Keep your inventory low.

Our intuition says that these two rules must be somehow related, and indeed, this is the case, we will find out later in this section.

Let us first deal with Rule 1, i.e., with waiting times. To keep them low we could try to find a job ordering which minimizes $W_{\max}$. This is very easy, see Exercise 2.3.1.3.

Much more interesting is to schedule jobs in such a way that *average waiting time* is minimized. This is just the arithmetic mean of waiting times:

$$\overline{W} = \frac{1}{n}\sum_{i=1}^{n} W_{(i)}.$$

If weights w_i are given and interpretable as cost of waiting per unit time, then the *average cost of waiting* is defined as

$$\overline{W}_w = \frac{1}{n}\sum_{i=1}^{n} w_{(i)} W_{(i)}$$

Because of (2.6):

$$\overline{W} = \frac{1}{n}\sum_{i=1}^{n}[C_{(i)} - p_{(i)}] = \frac{1}{n}\sum_{i=1}^{n} C_{(i)} - \frac{1}{n}\sum_{i=1}^{n} p_i$$

Observe that the second sum on the right hand side equals the *mean processing time*. It is independent of the ordering of jobs because processing times are given data. This sum has always the same value $C_{\max}$ for each schedule. That's why we could write p_i instead of $p_{(i)}$. Therefore minimizing $\overline{W}$ is equivalent to minimizing average completion time. And the same is true if we want to minimize average cost of waiting $\overline{W}_w$. In other words, we are seeking a schedule such that

$$\mathcal{C} = \sum_{i=1}^{n} C_{(i)} \to \min \qquad \text{or} \tag{2.10}$$

$$\mathcal{C}_w = \sum_{i=1}^{n} w_{(i)} C_{(i)} \to \min \tag{2.11}$$

These two objectives give rise to what is known as *total (weighted) completion time problem* or briefly as $\mathcal{C}$-problem and $\mathcal{C}_w$-problem, in standard scheduling notation denoted by

$$1||\sum C_i \qquad \text{and} \qquad 1||\sum w_i C_i. \tag{2.12}$$

How to solve these optimization problems?

As a first step towards solution we define a very important function now:

$$N(S,t) := \text{number of jobs in the system at time } t$$

Note that $N(S,t)$ is a function of the schedule. For given t different schedules S will yield different values of $N(S,t)$.

$N(S,t)$ is a step function that starts at time $t=0$ at height n and has jumps of size -1 at times $C_{(i)}$, therefore the widths of the steps are $p_{(i)}$. At time $t = C_{\max}$ the graph of $N(S,t)$ reaches zero.

▼ **Example 2.2** (continued).

J	1	2	3	4	5
p	8	3	10	5	4

We have already determined completion times for the schedule $S = [3,1,5,2,4]$:

$$C_{(1)} = C_3 = 10, \quad C_2 = C_1 = 18, \quad C_{(3)} = C_5 = 22$$
$$C_{(4)} = C_2 = 25, \quad C_{(5)} = C_4 = 30$$

The graph of $N(S,t)$ corresponding to S is shown in Figure 2.4.

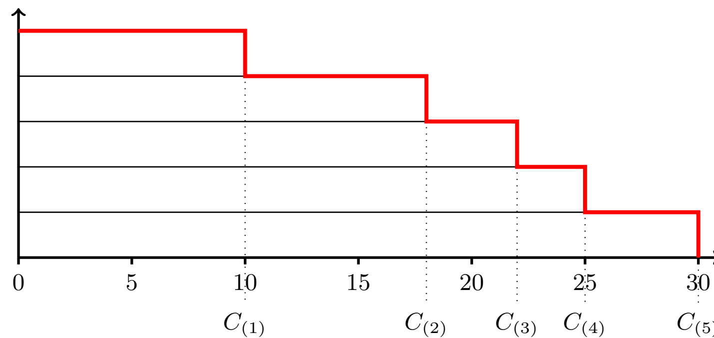


Figure 2.4

Particularly interesting is the *area* under $N(S,t)$:

$$N(S) = \int_0^{C_{\max}} N(S,t)\,\mathrm{d}t \tag{2.13}$$

As you can see from Figure 2.4, this area is composed by rectangles having height one and width equal to completion times $C_{(i)}$. Thus we have the alternative representation:

$$N(S) = \sum_{i=1}^{n} C_{(i)} = \mathcal{C}. \tag{2.14}$$

For Example 2.2 we find

$$\mathcal{C} = 10 + 18 + 22 + 25 + 30 = 105.$$

(2.14) tells us that a schedule which minimizes $\mathcal{C}$ must also minimize the area $N(S)$. But the latter is easy! Let us add straight line segments to the graph of $N(S,t)$ joining the kinks, see Figure 2.5. This connecting curve gives us the

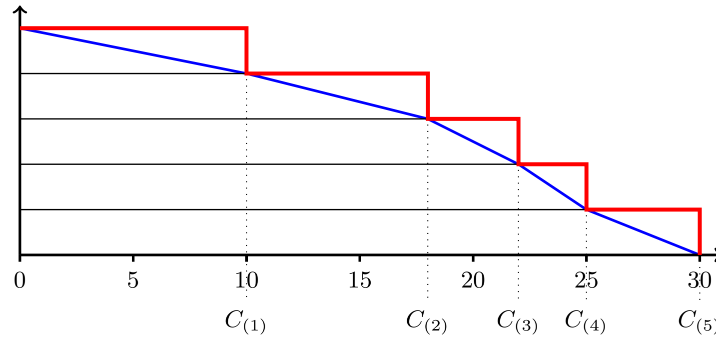


Figure 2.5

clue: *Order jobs in such a way that this curve is convex!* Or, equivalently: *Order jobs by increasing values of their processing times.* This will guarantee that N has minimum value.

▼ **Example 2.2** (continued).

Ordering jobs by increasing values of processing times gives

$$p_2 \le p_5 \le p_4 \le p_1 \le p_3.$$

It follows that we should process jobs according to the schedule $S^* = [2, 5, 4, 1, 3]$. This schedule produces completion times

S	2	5	4	1	3
p	3	4	5	8	10
C	3	7	12	20	30

Thus the area under $N(S,t)$ equals $N(S) = 3 + 7 + 12 + 20 + 30 = 72$, and this is also the minimum value of total completion time $\mathcal{C}$, see Figure 2.6 for an illustration.

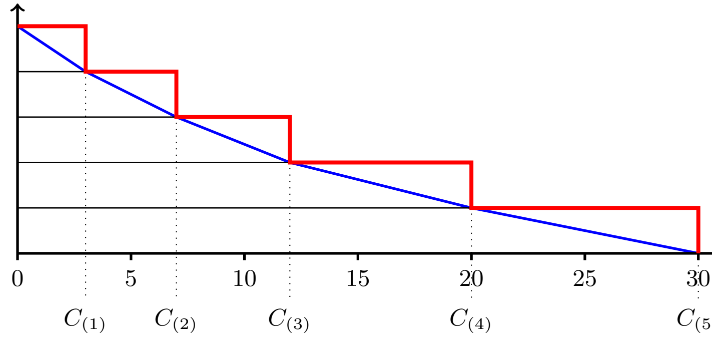


Figure 2.6

Of course this geometrical reasoning does not provide a rigorous proof, however, it gave us an *idea* how to find an optimal schedule. A similar geometrical argument may also be formulated for the $\mathcal{C}_w$-problem, see Exercise 2.4.4.3. It suggests to order jobs by increasing values of their ratios p_i/w_i.

In fact, this is one of the most famous results of scheduling theory, *Smith's WSPT-Rule*, (Smith, 1956), *weighted shortest processing times first.*

Theorem 2.4.1 (Smith's *WSPT*-Rule)**.** *The schedule S^* minimizing total weighted completion time is determined by ordering jobs by increasing values of the ratios p_i/w_i, i.e.*

$$S^* : \quad \frac{p_{(1)}}{w_{(1)}} \leq \frac{p_{(2)}}{w_{(2)}} \cdots \leq \frac{p_{(n)}}{w_{(n)}}.$$

If weights are equal to one, then order jobs by increasing values of processing times p_i to obtain the schedule minimizing total completion time:

$$S^* : \quad p_{(1)} \leq p_{(2)} \cdots \leq p_{(n)}.$$

This is called SPT-Rule. For both rules ties in the ordering may be broken arbitrarily.

Proof. Our proof will be based on a classical and extremely useful argument, the *interchange of adjacent jobs* in a schedule.

Let $S = [A, i, j, B]$ be a schedule with the following properties:

- A and B are two, possibly empty, sets of jobs.
- There is a pair of jobs i and j such that j follows immediately job i, but this pair is not in *WSPT*-order, i.e., $p_i/w_i > p_j/w_j$, or equivalently

$$p_i w_j > p_j w_i. \tag{2.15}$$

Form a new schedule S^* in which i and j have been interchanged, i.e.,

$$S^* = [A, j, i, B].$$

Suppose the last job in set A finishes at time t. If A is empty, then $t = 0$. The completion times of jobs i and j under schedule S are:

$$\begin{aligned} C_i(S) &= t + p_i \\ C_j(S) &= t + p_i + p_j, \end{aligned}$$

whereas under schedule S^* these are:

$$\begin{aligned} C_j(S^*) &= t + p_j \\ C_i(S^*) &= t + p_i + p_j. \end{aligned}$$

The completion times of all other jobs will remain unchanged, $C_k(S) = C_k(S^*)$ for all $k \neq i, j$. Next, we calculate the total weighted completion times under S and S^*:

$$\mathcal{C}_w(S) = \sum_{k\in A} w_k C_k(S) + w_i C_i(S) + w_j C_j(S) + \sum_{k\in B} w_k C_k(S), \tag{2.16}$$

$$\mathcal{C}_w(S^*) = \sum_{k\in A} w_k C_k(S^*) + w_j C_j(S^*) + w_i C_i(S^*) + \sum_{k\in B} w_k C_k(S^*). \tag{2.17}$$

Now we claim that S^* is *strictly better* that S in the sense that

$$\mathcal{C}_w(S) - \mathcal{C}_w(S^*) > 0.$$

To see this, we form the difference of (2.16) and (2.17) which yields:

$$\begin{aligned} \mathcal{C}_w(S) - \mathcal{C}_w(S^*) &= w_i C_i(S) + w_j C_j(S) - w_j C_j(S^*) - w_i C_i(S^*) \\ &= w_i(t + p_i) + w_j(t + p_i + p_j) - \\ &\quad - w_j(t + p_j) - w_i(t + p_i + p_j) \\ &= p_i w_j - p_j w_i > 0 \quad \text{because of (2.15)} \end{aligned} \tag{2.18}$$

Thus, putting any pair of adjacent jobs which are not in *WSPT*-order into *WSPT*-order *strictly improves* the total weighted completion time. Now, any finite sequence of numbers can be put into ascending order by pair-wise interchanges of adjacent elements. Therefore, starting with an arbitrary schedule S a sequence of pair-wise interchanges will improve the performance measure $\mathcal{C}_w$ step by step until S is in *WSPT*-order. This proves Smith's Rule. □

Keep your inventory low! This was a second Golden Rule of scheduling. Intuitively we expect that this objective is somehow related to waiting times of jobs. Indeed, while a job is waiting for service it is part of an *in-process inventory stock* which incurs holding cost. Recall (2.13), the function $N(S, t)$ gives us the number of jobs in system at any time $0 \le t \le C_{\max}$ for a particular schedule S. Now the *First Mean Value Theorem* of integral calculus states: the mean $\bar{f}$ of a continuous function $f(t)$ over the interval (a, b) is just

$$\bar{f} = \frac{1}{b - a} \int_a^b f(t)\,\mathrm{d}t. \tag{2.19}$$

It follows from the mean value theorem and (2.14) that the mean number of jobs in the system is given by

$$\overline{N}(S) = \frac{1}{C_{\max}} \int_0^{C_{\max}} N(S,t)\,\mathrm{d}t = \frac{1}{C_{\max}}\,\mathcal{C},$$

which may be rewritten as:

$$\overline{N}(S) = \frac{n}{C_{\max}} \cdot \overline{C}, \qquad \overline{C} = \frac{1}{n}\mathcal{C}\,. \tag{2.20}$$

But *SPT* minimizes $\mathcal{C}$ and hence it minimizes automatically $\overline{N}(S)$ thereby guaranteeing a minimum mean level of in-process inventory. Equation (2.20) therefore establishes the required relation between inventory stock and waiting times of jobs. It simply tells us that the mean number of jobs in the system is proportional to mean time a job spends in the system. The factor of proportionality, let's call it λ,

$$\lambda = \frac{n}{C_{\max}},$$

has also an interesting interpretation as a *mean*: λ equals the *mean number of arrivals per unit time*. This interpretation seems to be somewhat strange in the context of our single machine model, because we assumed that all jobs are available at time zero, there are no more arrivals during processing. But this strangeness disappears if we think of our service system running over a long period of time: whenever the last job is finished, the system is ready to accept the next bulk of n customers and the whole process of service starts from scratch, over and over again. This type of reasoning leads very naturally to the concept of a *steady state* which becomes fundamentally important once we introduce *randomness* into our scheduling models. Indeed, (2.20) is a special variant of *Little's Formula* (Little, 1961) which plays an prominent role in stochastic scheduling and queueing theory.

▼ Example 2.3

There are $n = 10$ jobs with processing times and weights given by:

J	1	2	3	4	5	6	7	8	9	10
p	3	6	6	5	4	8	9	2	7	5
w	1	18	12	8	8	17	16	8	2	5

If jobs are processed in their original order, i.e. $S = [1, 2, \dots, 9, 10]$, we obtain:

S	1	2	3	4	5	6	7	8	9	10
p	3	6	6	5	4	8	9	2	7	5
w	1	18	12	8	8	17	16	8	2	5
C	3	9	15	20	24	32	41	43	50	55

This gives a total weighted completion time of

$$C_w = 1 \cdot 3 + 18 \cdot 9 + \ldots + 5 \cdot 55 = 2616.$$

In other words, the mean cost per job due to waiting in the system, processing inclusive, is about 262 €.

How much can we reduce mean costs by scheduling jobs according the WSPT-Rule?

First, calculate the ratios p_i/w_i (rounded to two decimal places):

p_i/w_i	3.00	0.33	0.50	0.63	0.50	0.47	0.56	0.25	3.50	1.00
i	1	2	3	4	5	6	7	8	9	10

Then we sort them by increasing values of the ratios, which gives:

p_i/w_i	0.25	0.33	0.47	0.50	0.50	0.56	0.63	1.00	3.00	3.50
i	8	2	6	3	5	7	4	10	1	9

The optimal schedule S^* is given by the second line of the last table:

$$S^* = [8, 2, 6, 3, 5, 7, 4, 10, 1, 9].$$

Now reorder jobs according to S^* and calculate completion times:

S	8	2	6	3	5	7	4	10	1	9
p	2	6	8	6	4	9	5	5	3	7
w	8	18	17	12	8	16	8	5	1	2
C	2	8	16	22	26	35	40	45	48	55

The total weighted completion time evaluates to

$$C_w = 8 \cdot 2 + 18 \cdot 8 + \ldots + 2 \cdot 55 = 2167,$$

this is a reduction of mean cost per job of about 17%. ▲

2.4.2 Complexity of the *WSPT*-rule, Ties

It is one of the most celebrated theorems of computer science that sorting a list of length n by comparisons of list elements requires at least a number of about $O(n \log n)$ comparisons. This is true especially for sorting algorithms which are based on the idea of *divide and conquer*, like *mergesort* or *quicksort*. But not any sorting algorithm is that fast. Sorting by *pair-wise* interchange is not very efficient. A well-known algorithm based on this idea is *bubblesort* which does exactly what we have done in the proof of 2.4.1. Its complexity is of order $O(n^2)$ and therefore is considerably slower than *quicksort* or *mergesort*. Still, we needed that only as argument in the proof. In practice, sorting is done by any of the available fast algorithms for sorting. The important point, however,

is that the single server $\mathcal{C}_w$-problem is *well-behaved*, its complexity is *polynomial* in the number n of jobs to be scheduled.

It may happen that two or more jobs have the same p_i/w_i ratio. For instance in Example 2.3 jobs 3 and 5 had the same ratio 1/2. Such *ties* do not disturb the logic, they may be broken arbitrarily. If ties occur this simply means that our scheduling problem has alternative optima and this in turn means that the scheduler has a *choice*. In Example 2.2 we found the optimal schedule

$$S^* = [8, 2, 6, 3, 5, 7, 4, 10, 1, 9],$$

but since jobs 3 and 5 had the same p_i/w_i-ratio, an alternative schedule with the same total weighted completion time would be:

$$S^* = [8, 2, 6, 5, 3, 7, 4, 10, 1, 9].$$

Alternative optima are interesting as they offer us *strategic alternatives*. These may be important if we have secondary measures of performance like measures based on *lateness* to be introduced in the next section.

2.4.3 Variability of Completion Times and Waiting Times

SPT[3] together with Little's Formula allows us to kill two birds in one shot: keep mean in-process inventory and mean waiting times of jobs at lowest possible level. This is good for the company and it is good for customers also. But there is also a downside: mean value-based measures favor short jobs and penalize long ones which may not always be desirable. This is the point when *variability* enters the scene.

Variability becomes an interesting objective, if we want to provide all customers the same treatment as far as possible in the sense that completion times and waiting times should not vary too much among jobs. Low variability is thus important for guaranteeing smooth service of jobs.

There are many ways to measure variability. The most prominent measure is the mean squared deviation of completion times about their mean, the *variance*

$$\operatorname{var}[C(S)] = \frac{1}{n}\sum_{i=1}^{n}[C_{(i)} - \overline{C}(S)]^2, \tag{A}$$

where $\overline{C}(S)$ is the mean completion time under S. Note that $\operatorname{var}[C(S)]$ is a function of the schedule. Thus it seems natural to look for a schedule S which minimizes $\operatorname{var}[C(S)]$. However, the problem is now more complicated than just minimizing total completion time, because $\operatorname{var}[C(S)]$ is *not* a regular performance measure. Thus increasing the completion time of a particular job may *decrease* variance. Find an example (two jobs are sufficient)!

[3] In this section we consider only situations where job weights $w_i = 1$ for all i.

In a similar vein we might want to minimize variance of waiting times:

$$\mathrm{var}[W(S)] = \frac{1}{n}\sum_{i=1}^{n}[W_{(i)} - \overline{W}(S)]^2. \tag{B}$$

Intuitively we expect that a schedule S which minimizes (A) should also minimize (B) because $W_{(i)} = C_{(i)} - p_{(i)}$, recall (2.6). But interestingly, this is not the case, it is in a certain sense just the *contrary*, as we shall see in a moment.

From elementary statistics it is known that the variance can be written as

$$\mathrm{var}[C(S)] = \frac{1}{n}\sum_{i=1}^{n} C_{(i)}^2 - \overline{C}^2(S). \tag{C}$$

It is not difficult to show that SPT minimizes not only total completion time but more generally it minimizes also $\sum_{i=1}^{n} C_{(i)}^{\alpha}$ for any real $\alpha > 0$. Thus SPT is capable of minimizing each term in (C) separately, but unfortunately it *does not* minimize (C) as a whole. Indeed, it has been proved by Kubiak (1993) that minimization of (A) is NP-hard. So, that are bad news, and these are also surprising news in light of the apparent simplicity of the SPT-rule.

Although there is no known efficient way to minimize the variance of completion times, some really remarkable properties of this measure have been discovered. Some of these will be discussed now very briefly.

Let us define the *antithesis* S' of a schedule S: the job assigned to position i in S is assigned to position $n - i + 1$ in S'. More formally, if

$$S = [i_1, i_2, \ldots, i_n] \implies S' = [i_n, i_{n-1}, \ldots, i_2, i_1].$$

Thus S' is obtained simply by reversing S. To fix notation, if (i) denotes the position under S, then $(i)' = (n - i + 1)$ is the position under S'. For instance

$$S = [3, 2, 4, \underline{5}, 1] \implies S' = [1, \underline{5}, 4, 2, 3],$$

Indeed, $(2)' = 5 = (5 - 2 + 1) = (4)$, as the reader may verify.

Merten and Muller (1962) having been the first to study the variance problem proved the amazing relation

$$\mathrm{var}[C(S)] = \mathrm{var}[W(S')]. \tag{2.21}$$

From (2.21) it follows that if a schedule S minimizes the variance of completion times, then it is not difficult to prove that its antithesis S' minimizes the variance of waiting times *et vice versa*. This is really counter intuitive, because if S is a SPT schedule and thus minimizes total completion time, then its antithesis S' which results from ordering processing times by *decreasing values* maximizes total completion time. The antithesis of an SPT-schedule is also called an LPT-schedule or *longest processing times first*-schedule.

Another remarkable property of the variance problem is the *shape* of an optimal schedule: it is V-shaped, as has been shown by Eilon and Chowdhury (1977). This means the following: if $p_m = \min p_i$, then a V-shaped schedule S is of the form $S = [A, m, B]$, where

- A and B are possibly empty sets of jobs;
- the jobs in A are ordered by *decreasing values* of processing times;
- the jobs in B are ordered by *increasing values* of processing times;

Note that SPT and LPT are both special cases of V-shaped schedules. For SPT set A is empty, for LPT set B is empty.

▼ Example 2.4

Consider 8 jobs with processing times:

J	1	2	3	4	5	6	7	8
p	8	10	7	6	9	4	3	5

One possible V-shaped schedule is $S = [2, 3, 4, 8, 7, 6, 1, 5]$, see Figure 2.7. ▲

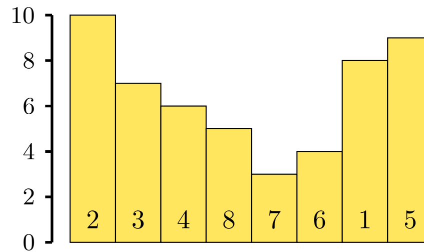


Figure 2.7: A V-shaped schedule

Given n jobs, the number of V-shaped schedules equals 2^n which is considerably less than $n!$. But the class of optimal schedules can be narrowed even further. It has been proved by Schrage (1975): there exists an optimal schedule minimizing $\text{var}[C(S)]$ in which *the longest job comes first.*

Based on this observation and the fact that an optimal schedule must be V-shaped we may devise a *heuristic* to obtain an approximate solution for the variance problem. This heuristic is very similar to the *Verified Spiral Algorithm* due to Ye et al. (2007), almost the same algorithm has been given earlier by Kanet (1981). Its idea is really simple:

- Order jobs by increasing values of processing times, i. e. jobs are indexed so that $p_1 \leq p_2 \leq \ldots \leq p_n$.
- Form a partial schedule $S = [n, 1]$.
- Select the last unscheduled job and insert it into S *before* or *after* job 1, depending on whether the chosen position yields the smaller variance of the partial schedule.
- Continue in this way until all jobs are scheduled.

Algorithm 2.1

VERIFIED SPIRAL ALGORITHM
Input: a set J of n jobs in SPT-order
Output: a schedule S and $\mathrm{var}[C(S)]$

begin
  $S := [n, 1]$, $U = J - \{1, n\}$
  $k := n - 2$

  while $k > 0$ **do**
    begin
      $j := U[k]$
      Form S_1 by inserting j *before* 1 in S
      Form S_2 by inserting j *after* 1 in S
      Calculate completion times $C_i(S_1), i = 1, \ldots, n$
      Calculate completion times $C_i(S_2), i = 1, \ldots, n$
      $v_1 := \mathrm{var}[C(S_1)]$
      $v_2 := \mathrm{var}[C(S_2)]$
      if $v_1 \leq v_2$ **then** $S := S_1$
        else $S := S_2$
      $k := k - 1$
    end

  $v = \mathrm{var}[C(S)]$

  return $[S, v]$
end

The VS-Algorithm is very fast, its time complexity is only of order $O(n \log n)$. But unfortunately, there are no known theoretical results about the accuracy of this heuristic, though numerical experiments carried out by Ye et al. (2007) suggest that it produces small relative errors on average.

▼ Example 2.5

We are given 6 jobs with processing times

J	1	2	3	4	5	6
p	1	2	2	3	3	20

We initialize first:

$$S = [6, 1], \quad U = [2, 3, 4, 5], \quad k = 4.$$

1. Iteration: $j = U[4] = 5$

$$\begin{aligned} &S_1 = [6, 5, 1] \qquad C(S_1) = [20, 23, 24] \\ &v_1 \doteq 2.89 \\ &S_2 = [6, 1, 5] \qquad C(S_2) = [20, 21, 24] \\ &v_2 \doteq 2.89 \end{aligned}$$

Since $v_1 = v_2$ we may take either of S_1 and S_2 to become S in the next iteration. Let choose S_1, so that $S = [6, 5, 1]$, and put $k = 3$.

2. Iteration: $j = U[3] = 4$

$$\begin{aligned} &S_1 = [6, 5, 4, 1] \qquad C(S_1) = [20, 23, 26, 27] \\ &v_1 \doteq 7.50 \\ &S_2 = [6, 5, 1, 4] \qquad C(S_2) = [20, 23, 24, 27] \\ &v_2 \doteq 6.25 \end{aligned}$$

Thus $S = S_2 = [6, 5, 1, 4]$ and $k = 2$.

3. Iteration: $j = U[2] = 3$

$$\begin{aligned} &S_1 = [6, 5, 3, 1, 4] \qquad C(S_1) = [20, 23, 25, 26, 29] \\ &v_1 \doteq 9.04 \\ &S_2 = [6, 5, 1, 3, 4] \qquad C(S_2) = [20, 23, 24, 26, 29] \\ &v_2 \doteq 9.04 \end{aligned}$$

Again $v_1 = v_2$, so take $S = S_1 = [6, 5, 3, 1, 4]$ and put $k = 1$.

4. Iteration: $j = U[1] = 2$

$$\begin{aligned} &S_1 = [6, 5, 3, 2, 1, 4] \qquad C(S_1) = [20, 23, 25, 27, 28, 31] \\ &v_1 \doteq 12.56 \\ &S_2 = [6, 5, 3, 1, 2, 4] \qquad C(S_2) = [20, 23, 25, 26, 28, 31] \\ &v_2 \doteq 12.25 \end{aligned}$$

Now $S = S_2$ and the algorithm stops here with $S = [6, 5, 3, 1, 2, 4]$ and

$$C(S) = [20, 23, 25, 26, 28, 31], \quad \text{var}[C(S)] = 12.25$$

which happens to be an optimal solution, as can be shown by complete enumeration of all $2^6 = 64$ possible schedules.

Let us draw a picture to illustrate what we have accomplished, see Figure 2.8. We can see very clearly in the left part of the graphic that variation of completion times is considerably smaller than for the SPT schedule. Indeed, the the variance of the SPT-schedule equals 100.14 compared to $v = 12.25$. But for that we have to pay a price. The VS-schedule produces total completion time of 153, for SPT we have only 59.

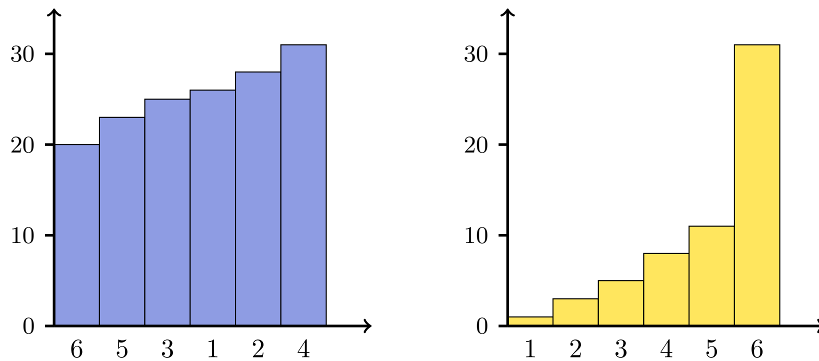


Figure 2.8: The VS schedule compared with SPT

Let us also have a look at waiting times. The optimal schedule S results in waiting times and corresponding variance:

$$W = [0, 20, 23, 25, 26, 28], \quad \text{var}[W(S)] \doteq 88.89.$$

However, it is the *antithesis* $S' = [4, 2, 1, 3, 5, 6]$ which minimizes variance of waiting times. In fact, our calculations yield $C(S') = [3, 5, 6, 8, 11, 31]$ and

$$W(S') = [0, 3, 5, 6, 8, 11], \quad \text{var}[W(S)] \doteq 12.25,$$

in accordance with (2.21). By the way, SPT yields a variance of waiting times of 14.89. ▲

2.4.4 Exercises

1. Consider the problem $1||\sum C_i$ with data

J	1	2	3	4	5	6
p	2	2	1	1	4	4

Find all optimal schedules. How many are there?

2. Our company has six big customers, A, B, C, D, E and F. Last night production orders came in:

Customer	A	B	C	D	E	F
Sales last year	125 000	108 000	46 000	270 000	83 000	75 000
number of parts ordered	600	450	1150	360	825	120

Machine time to produce a part is 0.5 minutes, parts are produced on a single machine.

- Find a schedule minimizing average waiting time of customers.
- Do the same but assign privileges to customers with high sales. What is a reasonable way to define privileges?
- Suppose that customer C orders 7500 parts instead of 1150. How sensitive are your results with respect to this change?

3. Define a function $N_w(S,t)$ analogous to $N(S,t)$ defined on page 13 taking care of job weights.

- How can this function be interpreted?
- Elaborate a geometric argument based on $N_w(S,t)$ for the WSPT-rule.

4. Show that SPT minimizes $\sum_{i=1}^n C_i^\alpha$ for $\alpha > 0$. What about the case $\alpha < 0$?

Hint: Use an interchange argument.

5. The strategy *Longest Processing Times First* (*LPT*-Rule) schedules jobs by putting them into descending order of processing times. Show that LPT-sequencing maximizes total completion time.

6. The *total discounted weighted completion time* is defined by

$$f(C_1, C_2, \ldots, C_n) = \sum_{i=1}^n w_i(1 - e^{-rC_i}).$$

Here r denotes a discount rate, usually close to zero. Prove the *WDSPT*-rule: schedule jobs in increasing order of

$$\frac{1 - e^{-rp_j}}{w_j e^{-rp_j}}.$$

Hint: Use an interchange argument.

7. The *stretch* of a job J_i is defined by $s_i = C_i/p_i$. It is used to measure the quality of service a customer gets. The ratio behind this measure is: customers estimate the speed of service by $1/s_i$. If s_i is high then customers get the impression of being served by a slow processor. Find a scheduling rule which minimizes total stretch $\sum_{i=1}^n s_i$.

8. Let S' be the antithesis to a schedule S. Show that

$$\sum_{i=1}^n C_{(i)} + \sum_{i=1}^n C_{(i)'} = (n+1)\sum_{i=1}^n p_i.$$

Recall the meaning of the notations (i) and $(i)'$, see page 20.

9. Let S' be the antithesis to a schedule S. Show that

$$\sum_{i=1}^n W_{(i)} + \sum_{i=1}^n W_{(i)'} = (n-1)\sum_{i=1}^n p_i.$$

10. Let S' be the antithesis to a schedule S. Show that

$$\mathrm{var}[C(S)] = \mathrm{var}[W(S')].$$

11. Consider the scheduling problem $1|p_j = 1|\sum C_i$. The β-field $p_j = 1$ means that all jobs have the same processing time $p_j = 1$. What is the optimal value of $\sum_{i=1}^{n} C_i$? What is the optimal value of var$[C]$?

12. Consider a single processor problem with objective to minimize variance of total completion time. Show that there exists an optimal schedule in which the longest job comes first. Is this condition necessary for optimality?

13. Construct a numerical example showing that var$[C(S)]$ is not a regular performance measure.

2.5 Lateness

2.5.1 Due dates and delivery times

In this section we introduce one more job parameter, *due dates*. To each job i we assign an integer-valued due date d_i which equals the time when job i *should* be finished. Due dates are not the same as *deadlines*, the latter meaning that when a job cannot be finished before its deadline $\tilde{d}_i$ then the corresponding order becomes obsolete, the customer is no longer interested in this service. Due dates, however, may be missed without rendering an order obsolete, although this incurs some cost.

In realistic production environments due dates are set be the customers or are the result of negotiations between customer and producer. But due dates may also be determined by technical considerations. For instance, in complex multistage production processes it is important that jobs do not complete too late, because other jobs of a subsequent stage would otherwise have to wait, they are *blocked* and this blocking deteriorates the system, in extreme cases it may even may lead to a *deadlock* of the whole system. However, there are also situations where it makes sense to consider due dates as decision variables, thus becoming part of the optimization process, as we shall see in Chapter 7.

The *lateness* L_i of job i is defined as

$$L_i = C_i - d_i, \tag{2.22}$$

and measures the deviation of the actual completion time of a job from a planned point in time. Job weights w_i no longer reflect holding costs as in the last section but costs which are incurred whenever a job misses its due date. Therefore it is quite natural to organize production in such a way that these costs are as small as possible.

We shall also allow *negative due dates*, these may be interpreted as *delivery times*. Suppose that $d_i < 0$, then we put $q_i = -d_i$ and (2.22) becomes

$$L_i = C_i + q_i. \tag{2.23}$$

Hence L_i equals the total time C_i it takes to produce a certain part plus the transportation time q_i required to convey the finished part to the customer. It is customary to call L_i in this case *completion-delivery time*.

Including delivery times in our model has an important effect on *model structure*. Now we have a two-stage process: the first stage is production of parts on one machine, the second stage handles deliveries. In contrast to the production stage there are no resource constraints in the delivery state. For every finished part there is an employee or some appropriate facility which takes care of delivery. It is perfectly OK, that deliveries are done in parallel. Thus technically speaking: the delivery process is carried out by several *parallel processors* whose number is sufficiently large so that no part has to wait for its delivery.

From (2.22) and (2.23) we can see that L_i is a *linear function* of C_i, see Figure 2.9 for an illustration. In the sequel we will concentrate mainly on the case of

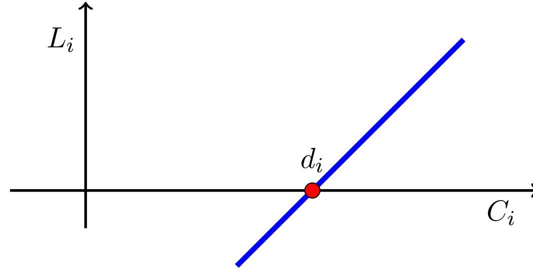


Figure 2.9

non-negative due dates, thus on the lateness L_i of job.

Obviously, L_i may be positive, zero or negative. The following terminology is common:

- If $L_i = 0$, job i is *on time*.
- If $L_i > 0$, job i finishes too late, it is *tardy*.
- If $L_i < 0$, job i is completed before it is due, it is called *early*.

Accordingly we define

$$\begin{aligned} \text{earliness of job } i\text{:} \quad & E_i = \max[0, d_i - C_i] \\ \text{tardiness of job } i\text{:} \quad & T_i = \max[0, C_i - d_i]. \end{aligned}$$

Observe that $E_i \geq 0$ and $T_i \geq 0$. If one of these quantities is positive, the other must be zero, so always $E_i \cdot T_i = 0$. Therefore, lateness may be expressed alternatively as

$$L_i = T_i - E_i. \tag{2.24}$$

2.5.2 Total weighted lateness

This performance measure reflects the total cost resulting from jobs missing their due dates. Total weighted lateness L_w is defined by

$$\mathcal{L}_w = \sum_{i=1}^{n} w_i L_i. \tag{2.25}$$

$\mathcal{L}_w$ is a *regular* performance measure, i.e., increasing in the completion times C_i, as can be seen from (2.22) or Figure 2.9. The corresponding scheduling model is $1\,||\sum w_i L_i$.

Finding a schedule that minimizes $\mathcal{L}_w$ is easy, just use the *WSPT*-rule.

Theorem 2.5.1. *A schedule S which minimizes total weighted lateness $\mathcal{L}_w$ is found by ordering jobs by increasing values of the ratios p_i/w_i.*

Proof. By definition (2.22),

$$\begin{aligned}\mathcal{L}_w &= \sum_{i=1}^{n} w_i L_i = \sum_{i=1}^{n} w_i (C_i - d_i)\\ &= \sum_{i=1}^{n} w_i C_i - \sum_{i=1}^{n} w_i d_i.\end{aligned} \tag{2.26}$$

But *WSPT* minimizes the first sum in (2.26). The second sum, however, is constant under any job ordering because the d_i and w_i are given data. Therefore, a schedule minimizing $\sum w_i C_i$ automatically minimizes $\mathcal{L}_w$. □

Remarks.

- This result is quite remarkable insofar as *WSPT* does not take care of the due dates at all.
- Nothing changes when we consider delivery times. The performance measure analogue to (2.25) is total weighted completion-delivery time:

$$\mathcal{D}_w = \sum_{i=1}^{n} w_i (C_i + q_i), \tag{2.27}$$

  which is also minimized by ordering jobs according to the *WSPT*-rule.

▼ Example 2.6

Consider $n = 10$ jobs with data given by:

J	1	2	3	4	5	6	7	8	9	10
p	5	4	7	10	12	9	6	3	9	4
w	4	1	8	2	3	2	3	1	5	4
d	20	45	15	11	34	25	40	62	62	50

Ordering jobs by $WSPT$ results in an optimal schedule S:

S	3	10	1	9	7	8	2	5	6	4
p	7	4	5	9	6	3	4	12	9	10
w	8	4	4	5	3	1	1	3	2	2
d	15	50	20	62	40	62	45	34	25	11
C	7	11	16	25	31	34	38	50	59	69
L	-8	-39	-4	-37	-9	-28	-7	16	34	58

with $\mathcal{L}_w = -251$. In other words, the mean cost due to lateness of this schedule equals:

$$\bar{\mathcal{L}}_w = \frac{1}{10} \sum_{i=1}^{10} w_i L_i = -25.1 \; .$$

Observe that this schedule produces 7 early and 3 tardy jobs. Moreover, job 4 has maximum lateness of 58, so it is pretty late!

▼ **Example 2.7** Processing times and delivery times of $n = 5$ jobs are as follows:

J	1	2	3	4	5
p	8	12	4	10	3
q	10	10	28	12	14

We assume that once an employee has delivered a job she is immediately available for handling the next delivery, if any. In this scenario the delivery times also include the time the employee requires to travel back from the customer.

Applying $WSPT$ yields an optimal schedule with evaluation:

S	5	3	1	4	2
p	3	4	8	10	12
q	14	7	28	12	10
C	3	7	15	25	37
$C+q$	17	14	43	37	47

Thus we find that total completion-delivery time equals

$$\mathcal{D} = \sum_{i=1}^{5} (C_i + q_i) = 158.$$

What is the minimum number of employees needed to deliver each job without any delay? This question is intimately connected with a fascinating *coloring problem* in graph theory. We shall hear more about it in the next chapter. For the moment it is sufficient to use a seemingly unsophisticated *ad hoc* approach. Let's just make a picture, see Figure 2.10. By simple inspection we find that the minimum number of employees needed equals two. Of course, for a larger number of jobs, an *algorithm* is required, of course.

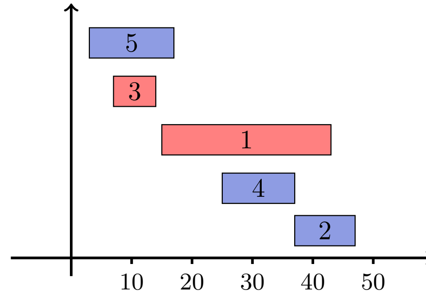


Figure 2.10

2.5.3 Maximum lateness

Example 2.6 suggests another interesting performance measure, *maximum lateness* which is defined as

$$\text{maximum lateness:} \quad L_{\max} = \max_i L_i = \max_i (C_i - d_i) \tag{2.28}$$

Note that $L_{\max}$ is a regular performance measure, since it will never decrease when any of the completion times is increased. Therefore it seems quite natural to look for a schedule S which yields the *smallest possible value* for $L_{\max}$:

$$\text{find an } S\text{, such that:} \quad L_{\max} \to \min \tag{2.29}$$

A schedule minimizing maximum lateness can be found very easily (Jackson, 1955).

Theorem 2.5.2 (Jackson's earliest due date rule)**.** *To find an optimal schedule S for (2.29) order jobs according to increasing values of their due dates:*

$$S: \quad d_{(1)} \leq d_{(2)} \leq \cdots \leq d_{(n)} \qquad \textit{EDD-Rule} \tag{2.30}$$

Before we prove this classical result let us consider an example.

▼ Example 2.8

The job data are the same as in Example 2.6, but without weights:

J	1	2	3	4	5	6	7	8	9	10
p	5	4	7	10	12	9	6	3	9	4
d	20	45	15	11	34	25	40	62	62	50

Sorting jobs by increasing values of their due dates, calculating completion times and lateness values of the resulting *EDD*-schedule S yields:

S	4	3	1	6	5	7	2	10	8	9
p	10	7	5	9	12	6	4	4	3	9
d	11	15	20	25	34	40	45	50	62	62
C	10	17	22	31	43	49	53	57	60	69
L	-1	2	2	6	9	9	8	7	-2	7

We find that the *EDD*-schedule yields a maximum lateness of $L_{\max} = 9$ attained by jobs 5 and 7.

The reader may check that scheduling jobs by the *SPT*-rule which minimizes average lateness will give a value of $L_{\max} = 46$, more than five times of optimal value!

▲

Proof of Theorem 2.5.2. We use an interchange argument as we did in the proof of the *WSPT*-Rule (Theorem 2.4.1). Let S be a schedule which does not conform to the *EDD*-rule, i.e.,

$$S = [A, i, j, B],$$

where A and B are two possibly empty sets of jobs, job j is scheduled immediately after i, but $d_i > d_j$. Furthermore, let

$$p(A) = \sum_{i \in A} p_i, \quad L_A = \max_{i \in A} L_i, \quad L_B = \max_{i \in B} L_i,$$

i.e., $p(A)$ denotes the sum of the process times in set A and therefore the time when job i will be started, since we need not consider inserted idle time. Now

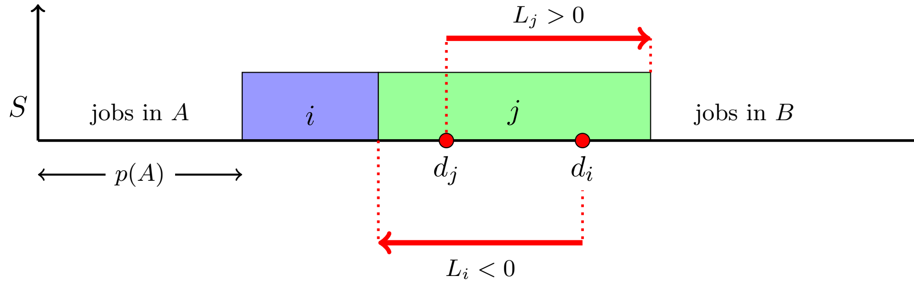


Figure 2.11: Jobs i and j not *EDD*-conforming

form a new schedule S^* by interchanging jobs i and j and calculate the lateness of i and j under S and S^*, see Figure 2.11 and Figure 2.12 for illustration. For schedule S we can read off:

$$L_i(S) = p(A) + p_i - d_i \tag{A}$$
$$L_j(S) = p(A) + p_i + p_j - d_j, \tag{B}$$

wherease for S^*:

$$L_i(S^*) = p(A) + p_j + p_i - d_i \tag{C}$$
$$L_j(S^*) = p(A) + p_j - d_j. \tag{D}$$

A comparison of (A) and (B) shows that $L_j(S) > L_i(S)$. Indeed,

$$\begin{aligned} p(A) + p_i + p_j - d_j &> p(A) + p_i - d_i \\ \implies p_j &> d_j - d_i. \end{aligned}$$

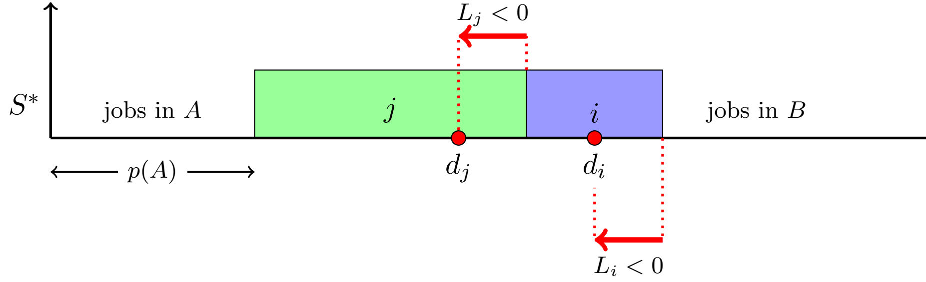


Figure 2.12: Jobs i and j *EDD*-conforming

But $d_j - d_i < 0$ by assumption and $p_j \geq 0$, so the last inequality follows. Similarly, by (B), (C) and (D):

$$L_j(S) > L_i(S^*), \quad \text{and} \quad L_j(S) > L_j(S^*). \tag{E}$$

Hence

$$L_j(S) > \max[L_i(S), L_i(S^*), L_j(S^*)].$$

Now, observe that L_A and L_B have the same values for S and S^*. Therefore

$$\begin{aligned} L_{\max}(S) &= \max[L_A, L_B, L_i(S), L_j(S)] \\ &= \max[L_A, L_B, L_j(S)] \qquad (\text{because } L_j(S) > L_i(S)) \\ &\geq \max[L_A, L_B, L_i(S^*), L_j(S^*)] \qquad (\text{by (E)}) \\ &= L_{\max}(S^*). \end{aligned}$$

Hence

$$L_{\max}(S) \geq L_{\max}(S^*). \tag{2.31}$$

□

Remarks.

- Inequality (2.31) is *not strict*. This means that even when $d_i > d_j$ it may happen that after interchanging i and j we have $L_{\max}(S) = L_{\max}(S^*)$. All we can say is, the new schedule S^* will be no worse than S. This is in contrast to the *WSPT*-rule. There we have been able to show that a pair-wise exchange always *improved* the objective function.
- Ordering jobs according to *EDD* is therefore a *sufficient* condition for minimal $L_{\max}$. It may be that there are other non-*EDD* schedules (indeed, there may be many of them), yielding the same minimum for $L_{\max}$.

  Here is an example:

J	1	2	3	4	5
p	5	4	8	2	5
d	8	9	10	9	11
C	5	9	17	19	25
L	−3	0	7	10	14

Jobs 3 and 4 are not in *EDD*-order, $L_{\max} = 14$. Interchanging jobs 3 and 4 and thereby establishing *EDD*-order yields:

S	1	2	4	3	5
p	5	4	2	8	5
d	8	9	9	10	11
C	5	9	11	19	25
L	-3	0	2	9	14

As you can see, this schedule has the same $L_{\max}$.

- Nothing changes if we consider delivery times, again the *EDD*-rule yields an optimal schedule minimizing maximum completion-delivery time

$$D_{\max} = \max_i [C_i + q_i]$$

- *EDD* requires sorting of n due dates, thus the time complexity of *EDD* is $O(n \log n)$, the same we as for *WSPT*.

2.5.4 Lawler's Algorithm

The *EDD*-rule presented in the last section follows also as a very special case from an algorithm due to Lawler (1973) which is capable of solving much more complicated scheduling problems. It is based on the powerful idea of constructing a schedule in a greedy manner *from right to left*.

Let J denote to set of all n jobs and $I \subset J$ denotes the subset of jobs still to be scheduled, i.e., those jobs which have not been assigned a position in the final schedule. Define the total processing time of set I:

$$T = \sum_{i \in I} p_i = p(I)$$

With each job we associate a monotone and increasing cost function $g_i(t)$, i.e.,

$$g_i(s) \leq g_i(t), \quad \text{if } s \leq t,$$

The function $g_i(t)$ equals the cost incurred by job i, if i completes at time t. Then the following holds:

Lemma 2.5.1. *Let $g_k(T) = \min_{i \in I} g_i(T)$. Then there exists a schedule S which minimizes the maximum of incurred cost and in which job k is the last in S.*

Proof. Let S^* be a job ordering in which job $\ell \neq k$ is last, A and B denote possibly empty sets of jobs other than ℓ and k so that

$$S^* = [A, k, B, \ell].$$

Move k to the last position thereby getting a new schedule S:

$$S = [A, B, \ell, k]$$

Now observe the following by comparing S and S^*:

- No job in S will be completed later than in S^*, except job k.
- Since by assumption the cost functions $g_i(t)$ are monotone increasing in t, no incurred cost can be greater in S than in S^*, except possibly the cost due to job k.
- But job k has been chosen in such a way that $g_k(T) \leq g_\ell(T)$.

It follows that the maximum cost of S^* is no greater than that of S. □

This lemma suggests the following Algorithm 2.2 for constructing an optimal schedule: simply find a job k which can be placed last. Remove this job from the set of jobs and apply the same rule to the remaining $n-1$ jobs, etc. It solves the scheduling problem $1\,|\,|\,g_{\max}$, where

$$g_{\max} = \max(g_1(C_1), g_2(C_2), \ldots, g_n(C_n)).$$

Algorithm 2.2

LAWLER's ALGORITHM

Input: a set J of n jobs
a set of cost functions $g_i(t), i = 1, 2. \ldots, n$

Output: an optimal schedule S and optimal $g_{\max}$.

begin
  $S := [\,]$
  $I := J$

  begin
    begin
    LOOP: $T = \sum_{i \in I} p_i$
      Find k such that $g_k(T) = \min_{i \in I} g_i(T)$
      $S := [k, S]$
      $I := I - \{k\}$
      if $I \neq \varnothing$ **then go to** *LOOP*
    end

    Calculate $C_i(S), i = 1, 2, \ldots, n$
    Calculate $g_i(C_i(S)), i = 1, 2, \ldots, n$
    $g_{\max} = \max_i(g_i(C_i))$

    return $[S, g_{\max}]$
  end
end

Remarks.

- Lawler's algorithm allows us to consider problems even with *nonlinear* cost functions, like $g_i(t) = \alpha t^2$ or something of that kind, provided, these functions are monotone increasing. The cost functions may be different for each job.
- The *EDD*-rule simply follows by using the cost functions $g_i(t) = t - d_i$. This is easy to see. In the first iteration the algorithm selects a job k for which $g_k(T) = T - d_k$ is minimal. This will be the case, if d_k has maximum value. Thus the job with largest due date is put on the last position of S, and so on.
- The *time complexity* of Lawler's Algorithm can be shown to be $O(n^2)$.

▼ Example 2.9

Consider once again the data of Example 2.8, but this time we want to find a schedule which minimizes *maximum weighted lateness* defined by $L_{\max}^w = \max_i[w_i(C_i - d_i)]$:

J	1	2	3	4	5	6	7	8	9	10
p	5	4	7	10	12	9	6	3	9	4
w	4	1	8	2	3	2	3	1	5	4
d	20	45	15	11	34	25	40	62	62	50

We apply Lawler's algorithm with cost functions $g_i(t) = w_i(t - d_i)$.

In the first iteration we have $I = [1, 2, \ldots, 10]$, $T = \sum_{i \in I} p_i = 69$ and

i	1	2	3	4	5	6	7	8	9	10
$g_i(T)$	196	24	432	116	105	88	87	7	35	76

The minimum is found for $i = 8$, thus

$$S = [\,8\,], \quad I = [\,1, 2, 3, 4, 5, 6, 7, 9, 10\,], \quad T = 66$$

Recalculation of cost functions for $T = 66$ yields

i	1	2	3	4	5	6	7	9	10
$g_i(T)$	184	21	408	110	96	82	78	20	64

The minimum is now obtained for job $i = 9$, it follows that

$$S = [\,9, 8\,], \quad I = [\,1, 2, 3, 4, 5, 6, 7, 10\,], \quad T = 57,$$

and so on. The table with complete calculations is given below.

Iter.	g_1	g_3	g_3	g_4	g_5	g_6	g_7	g_8	g_9	g_{10}	T	S
1	196	24	432	116	105	88	87	7	35	76	69	8
2	184	21	408	110	96	82	78		20	64	66	9,8
3	148	12	336	92	69	64	51			28	57	2,9,8
4	132		304	84	57	56	39			12	53	10,2,9,8
5	116		272	76	45	48	27				49	7,10,2,9,8
6	92		224	64	27	36					43	5,7,10,2,9,8
7	44		128	40		12					31	6,5,7,10,2,9,8
8	8		56	22							22	1,6,5,7,10,2,9,8
9			16	12							17	4,1,6,5,7,10,2,9,8
10			-64								7	3,4,1,6,5,7,10,2,9,8

Thus the optimal schedule is $S = [3, 4, 1, 6, 5, 7, 10, 2, 9, 8]$ with $L^w_{\max} = 27 = g_7$ in iteration 5.

As remarked above, the complexity of Lawler's Algorithm is in general $O(n^2)$ except for situations where the application of the algorithm essentially results in *sorting* jobs, e.g., *EDD* is an example which minimizes $L_{\max}$ and has complexity $O(n \log n)$, thus *EDD* is considerably faster. So the question arises whether there is an algorithm minimizing *maximum weighted lateness* $L^w_{\max}$ that is faster than $O(n^2)$. Actually, such an algorithm has been found by Hochbaum and Shamir (1989). This algorithm makes clever use of special data structures and thereby decreases complexity to $O(n \log^2 n)$, thus being roughly as fast as *EDD*.

▲

2.5.5 Exercises

1. Devise a rule yielding an optimal schedule to minimize weighted maximum lateness.

2. Using Lawler's algorithm find a simple rule to minimize weighted maximum lateness when all jobs have the same due date.

2.6 Tardiness - a first glimpse

When a job is not on time, it may be early or it may be tardy. Tardiness is positive lateness, and very often in practical applications this is what really hurts, because when a jobs needs too much time be be finished high costs may be the result.

Tardiness of a job i is defined as:

$$T_i = \max(0, C_i - d_i), \tag{2.32}$$

and unlike lateness this *is not a linear function* of the completion time C_i (see Figure 2.13). Working with nonlinear functions is generally more difficult than working with linear functions, and indeed, most scheduling problems with tardiness related objective functions are NP-hard.

But there are exceptions to this somewhat disillusioning perspective.

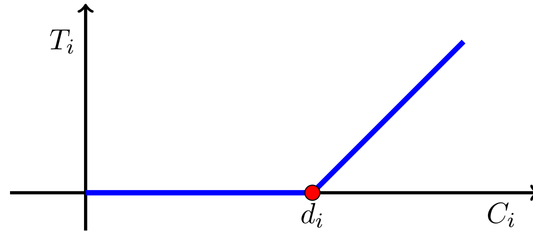


Figure 2.13: Tardiness as nonlinear function of completion time

2.6.1 Maximum Tardiness

Let us define the regular performance measure *maximum tardiness* of a schedule by

$$T_{\max} = \max_{1 \le i \le n} T_i. \tag{2.33}$$

How to find a schedule that minimizes $T_{\max}$?

Again, Lawler's algorithm provides the answer. As you can see from the definition (2.32) as well as from Figure 2.13, tardiness is an continuous monotone increasing function of the completion times. Thus for a given set J of jobs with total processing time $T = p(J)$ in each iteration of Lawler's algorithm we have to find a job k which minimizes

$$g_k(T) = \max_{i \in J}(0, T - d_i).$$

Obviously this is the job in J which has maximum due date. As a result, the algorithm will always order jobs according to increasing values of the due dates. But this is the *EDD*-rule!

Theorem 2.6.1 (Maximum tardiness)**.** *To find an optimal schedule S which minimizes maximum tardiness order jobs according to increasing values of their due dates, thus apply the EDD-rule.*

▼ **Example 2.10** Consider $n = 10$ jobs with data:

J	1	2	3	4	5	6	7	8	9	10
p	3	4	8	19	20	11	11	9	15	3
d	81	55	57	61	79	53	34	34	38	46

Ordering jobs by increasing values of due dates d_i yields an optimal schedule S which evaluates to:

S	7	8	9	10	6	2	3	4	5	1
p	11	9	15	3	11	4	8	19	20	3
d	34	34	38	46	53	55	57	61	79	81
C	11	20	35	38	49	53	61	80	100	103
T	0	0	0	0	0	0	4	19	21	22

Hence, $T_{\max} = 22$. ▲

Lawler's algorithm is applicable also when we are interested in *maximum weighted tardiness*, see the exercises below.

But what about the performance measures *total tardiness* T and *total weighted tardiness* T_w,

$$\mathcal{T} = \sum_{i=1}^{n} T_i, \qquad \mathcal{T}_w = \sum_{i=1}^{n} w_i T_i \quad ?$$

These are very important performance measures since, e.g., $\mathcal{T}_w$ yields the average cost of jobs finishing too late. Unfortunately, finding a schedule that minimizes $\mathcal{T}$ or $\mathcal{T}_w$ is NP-hard. Therefore we will postpone a discussion of the total tardiness problem to chapter 6.

2.6.2 The number of tardy jobs

The quality of work done by production managers is often assessed by the *number of tardy jobs* they are responsible for. To count the number of tardy jobs it is convenient to introduce the *Heavyside function*:

$$U_i(C_i) := U_i = \begin{cases} 1 & \text{if } C_i \geq d_i \\ 0 & \text{otherwise} \end{cases}$$

This is a step function defined for each job i and it has a single jump of size $+1$, if the completion time of i exceeds its due date (see Figure 2.7). The number

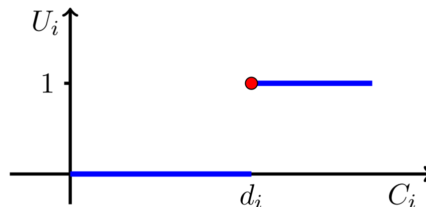


Figure 2.14: The Heaviside function for job i

of tardy jobs produced by a particular schedule is simply $\mathcal{U} = \sum_{i=1}^{n} U_i$ and its mean value $\bar{U}$ equals the *relative frequency* of tardy jobs. The corresponding scheduling problem of minimizing $\mathcal{U}$ has signature $1||\sum U_i$.

We expect that *EDD*-ordering should have an influence on this measure of performance. Indeed, if *EDD* produces no tardy job then it is optimal with respect to $\mathcal{U}$. The same is true when *EDD* results in exactly one tardy job, since EDD minimizes also $\mathcal{T}_{\max}$. But in all other cases *EDD* need not be optimal, as the following example shows.

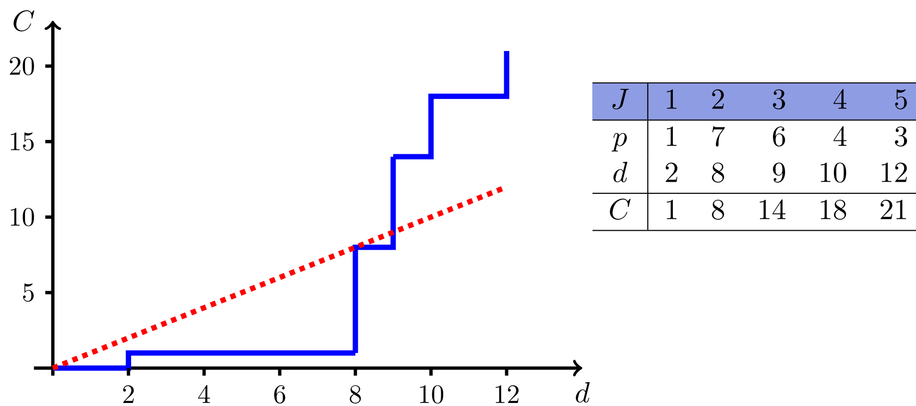
▼ **Example 2.11**

Consider $n = 5$ jobs with data given by:

J	1	2	3	4	5
p	1	7	6	4	3
d	2	8	9	10	12
C	1	8	14	18	21
T	0	0	5	8	9
U	0	0	1	1	1

Jobs are indexed in *EDD*-order and as you can see, $\sum U_i = 3$, so there are three tardy jobs. This is not optimal, as the alternative schedule $S^* = [1, 4, 5, 2, 3]$ produces two tardy jobs only. Please verify!

To get an idea of how this problem can be solved we will apply a rather simple graphical device, a *d/C-diagram*. This is a very useful tool as we shall see also in Chapter 6. The diagram consists of a coordinate system, the horizontal axis for due dates d_i, the vertical axis for completion times C_i of the *EDD*-schedule. In this coordinate system we draw the points $(d_i, C_i), i \in J$ and connect them by line segments so that we get a step function. Let us also draw the line $d = C$, see Figure 2.15.



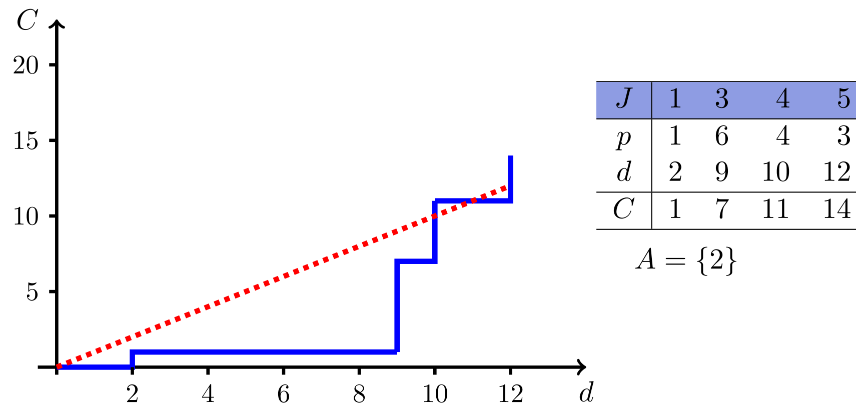
J	1	2	3	4	5
p	1	7	6	4	3
d	2	8	9	10	12
C	1	8	14	18	21

Figure 2.15: The d/C-diagram for $J = \{1, 2, 3, 4, 5\}$

Ideally, the step function should not cross the line $d = C$, but it does in our example, as you can see. This simply means that there are points with $C_i > d_i$, in other words there are tardy jobs. Only jobs 1 and 2 are on time. Now time has come to be a little bit *greedy*: let us remove the *longest job* among the on time jobs and put it aside. The longest job is 2, as it has processing time $p_2 = 7$.

What will happen? All jobs in the *EDD*-schedule following job 2 will now complete earlier by 7 time units. As a result their tardiness is reduced and it

may happen that jobs which have been tardy before are now no longer tardy after this change. Graphically, this means that we cut out one step of the step function and as a result its tail segment *moves down*. This situation is shown in Figure 2.16.

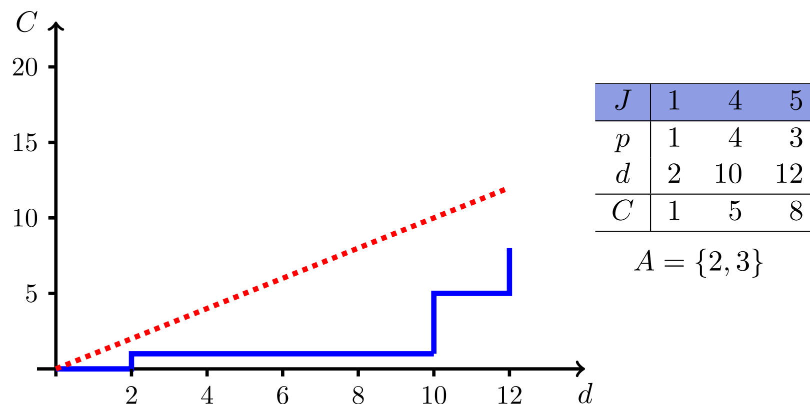


J	1	3	4	5
p	1	6	4	3
d	2	9	10	12
C	1	7	11	14

$A = \{2\}$

Figure 2.16: The d/C-diagram for $J = \{1, 3, 4, 5\}$

Still the step function crosses the line $d = C$, but the situation looks much better now. Why shouldn't we apply the same trick again? The longest job among the on time jobs is job 3. We remove it from J and put it aside, i. e., store it in a list A. And again all jobs after 3 will complete earlier and thereby become non-tardy.



J	1	4	5
p	1	4	3
d	2	10	12
C	1	5	8

$A = \{2, 3\}$

Figure 2.17: The d/C-diagram for $J = \{1, 4, 5\}$

Now the step function runs completely below the line $d = C$, there are no more tardy jobs.

This simple graphical procedure gives us the clue how the *Moore-Hodgson Algorithm* works. It is not a proof of the *correctness* of that algorithm, just a way to intuitive understanding.

Algorithm 2.3 given below does exactly what we have done above. It constructs the optimal schedule in a *left-to-right fashion*, as follows:

- Jobs to be scheduled are given in *EDD*-order.
- The algorithm maintains two lists S and A and a variable t, the completion time of the last job added to S.
- At the beginning both lists are empty and $t = 0$.
- Jobs are added one at a time to S. The completion time variable t is updated and it is checked whether the last job added is tardy. If so, the *longest job* in S, say job k, is determined. It is removed from S, added to A, and t is reduced by p_k.
- This process continues until all jobs have been considered.

The algorithm returns the lists S and A: S contains the non-tardy jobs in *EDD*-order, A is the list of tardy jobs.

The optimal rule is therefore:

- Schedule jobs in S in that order.
- Then process jobs in A in *any order*.
- The optimal value of the performance measure is $\min \sum U_i = |A|$, the number of jobs in set A.

The time complexity of the Moore-Hodgson Algorithm is $O(n \log n)$ because the hardest part is to order jobs by increasing values of due dates.

Here is a formal statement of the Algorithm.

Algorithm 2.3

MOORE-HODGSON ALGORITHM

Input: n jobs with processing times p_i
and due dates d_i, jobs in *EDD*-order.

Output: a list S of non-tardy jobs and a list A of tardy jobs.

```
begin
  S   := [ ]
  A   := [ ]
  t   := 0

  for j := 1 to n do
    begin
      S := [S, j]
      t := t + p_j
      if t > d_j then do
        begin
          find job k with maximum processing time in S
          A := [A, k]
          S := S − {k}
          t := t − p_k
        end
```

end

return $[S, A]$
end

▼ **Example 2.11** (continued).

Applying Algorithm 2.3 yields after initialization step by step:

$$
\begin{array}{llll}
j=1: & S=[1], & A=[\,], & t=1 \\
j=2: & S=[1,2], & A=[\,], & t=8 \\
j=3: & S=[1,2,3], & A=[\,], & t=14
\end{array}
$$

$t = 14 > d_3 = 9$
$k = 2$, since $p_2 = 7$ is maximal, therefore remove job 2
$\Longrightarrow \; S = [1,3], \quad A = [2], \quad t = 7$

$$
j=4: \quad S=[1,3,4], \quad A=[2], \quad t=11
$$

$t = 12 > d_4 = 10$
$k = 3$, since $p_3 = 6$ is maximal, therefore remove job 3
$\Longrightarrow \; S = [1,4], \quad A = [2,3], \quad t = 7$

$$
j=5: \quad S=[1,4,5] \quad A=[2,3], \quad t=10
$$

STOP

Thus we have found:

- It is optimal to process first jobs $S = [1,4,5]$ in that order.
- Set $A = [2,3]$ has $|A| = 2$ elements, therefore the optimal value of the performance measure equals $\mathcal{U} = 2$
- Jobs in A can be processed in *any order*.
- Therefore we have *two optimal schedules*:

$$
S_1^* = [1,4,5,2,3] \quad \text{and} \quad S_2^* = [1,4,5,3,2]
$$

▲

Remark. It is not at all clear that Algorithm 2.3 yields an optimal schedule. This claim requires a proof, of course. We shall not give a proof here and refer the interested reader to the literature because all known proofs though not being difficult are very messy, see for instance (Moore, 1968) and (Sturm, 1970).

What about job weights?

Consider the following scheduling problem: at 0 am (say) due to bad weather conditions there are n aircrafts circling in turning loops near an airport. Aircraft i will run out of fuel in d_i minutes (from now on). It is also known to flight control that this aircraft has w_i passengers on board. Under guidance of the flight controller it will take p_i minutes to bring down aircraft i safely. The bad thing is: it has to be expected that an aircraft waiting too long for landing permission may run out of fuel and crash.

So, what to do? From a scheduling point of view, one alternative would be to minimize $\mathcal{U}$, thus schedule aircrafts so that the number of crashes is minimum. This can be done by the Moore-Hodgson Algorithm. A much better and much more reasonable alternative is to take into account the expected number of casualties. This number is

$$\mathcal{U}_w = \sum_{i=1}^{n} w_i U_i$$

Thus the appropriate performance measure is now the *weighted number* of tardy jobs $\mathcal{U}_w = \sum w_i U_i$. Unfortunately, the problem $1||\sum w_i U_i$ is a hard one, but it can be solved by means of a clever enumeration method, *dynamic programming*, which we will explore in detail in chapter 4.

2.7 Secondary Measures of Performance

Example 2.11 resulted in two optimal schedules, optimal in the sense that they minimize the number of tardy jobs. Thus our optimization problem has *multiple optima*, a situation not untypical of combinatorial optimization.

Multiple optima provide us with *strategic alternatives*. We can figure out easily what these alternatives are in the Example 2.11. For this purpose let us calculate total completion time $\mathcal{C}$ and maximum tardiness $T_{\max}$ for S_1^* and S_2^*. A trite calculation gives:

S_1^*	1	4	5	2	3
p	1	4	3	7	6
d	2	10	12	8	9
C	1	5	8	15	21
T	0	0	0	7	12

S_2^*	1	4	5	3	2
p	1	4	3	6	7
d	2	10	12	9	8
C	1	5	8	14	21
T	0	0	0	5	13

S_1^* has maximum tardiness $\mathcal{T}_{\max} = 12$, whereas for S_2^* we have $T_{\max} = 13$. Also, total completion time for S_1^* equals $\mathcal{C}(S_1^*) = 1 + 5 + 8 + 15 + 21 = 50$, for S_2^* we have $\mathcal{C}(S_2^*) = 49$.

That S_1^* produces a smaller $\mathcal{T}_{\max}$ is clear: the tardy jobs 2 and 3 are in *EDD*-order in S_1^*, not so in S_2^*. But S_1^* can be improved with respect to completion time without increasing the number of tardy jobs nor maximum tardiness. Just exchange jobs 4 and 5. You will find that now total completion time reduces to $\mathcal{C} = 49$. That is pretty close to the minimum value that results from SPT-ordering.

But why did we take job 4 in S_1^*? There is a simple reason for that: this job has the largest processing time among the non-tardy jobs, thus with respect to total completion time it is profitable to move this job to the end of the list of non-tardy jobs. Moreover, its due date is sufficiently large so that this shift will not produce more tardy jobs.

This idea can be crafted into an algorithm due to Smith[4]. Suppose that our primary objective is somehow tardiness related and we could find in some way a schedule which produces zero tardy jobs. This schedule is certainly optimal with respect to $T_{\max}$, $\sum w_i U_i$ and even total weighted tardiness $\sum w_i T_i$. Suppose also that our secondary objective is to minimize total weighted completion time. Step by step we identify jobs which when moved to the end of the schedule will not produce any tardiness. Among these jobs we always select the one with maximum processing time since moving this job to the end reduces total completion time as much as possible.

Algorithm 2.4

SMITH's SECOND RULE

Input: n jobs with processing times p_i and due dates d_i
a set J of jobs already indexed so that there are no tardy jobs.

Output: a schedule S minimizing total completion time.

begin
$S \quad := [\,]$

while $J \neq \varnothing$ **do**
begin
$T := \sum_{i \in J} p_i$
Find job $k \in J$ such that $d_k \geq T$ and
$p_k \geq p_\ell$ among all jobs $\ell \in J$ with $d_\ell \geq T$ (∗)
$S := [k, S]$
$J := J - \{k\}$
end
return S
end

▼ Example 2.12

The following set of jobs is already in *EDD*-order which results in zero tardy jobs and total completion time $C = 614$:

J	1	2	3	4	5	6	7	8	9	10
p	6	18	7	3	18	12	18	13	16	4
d	47	53	65	67	69	92	94	120	133	138
C	6	24	31	34	52	64	82	95	111	115

In the first iteration we have $T = 115$ and there are three candidates to be moved, jobs 8, 9 and 10. Job 9 has maximum processing time, so it is put at the end of the schedule and removed from the list of jobs yet to be scheduled.

[4] Unfortunately also known as Smith's Rule.

In the next round $T = 99$ and candidates for move are jobs 8 and 9. Job 8 is moved because it has larger processing time, etc. The optimal schedule finally found is:

$$S = [4, 10, 1, 3, 6, 2, 5, 7, 8, 9],$$

with total completion time to $\mathcal{C}(S) = 493$ and still all jobs non-tardy. ▲

2.8 Deadline Scheduling

Algorithm 2.4 is capable of solving another interesting problem, scheduling with *deadlines*. Deadlines are very different from due dates. If in a bulk of n jobs a job is tardy, thus completes after its due date, then costs are incurred which may be high, sometimes prohibitively high. But still it may be reasonable to accept this order.

In contrast, suppose we are to produce n parts with processing times p_i and *deadlines* $\tilde{d}_i$. If even one job misses its deadline then the whole order becomes obsolete. The corresponding scheduling models with and without job weights are denoted by

$$\mathcal{C}(\tilde{d}) : \ 1|\tilde{d}_i|\sum C_i \quad \text{and} \quad \mathcal{C}_w(\tilde{d}) : \ 1|\tilde{d}_i|\sum w_i C_i.$$

Let us consider first problem $\mathcal{C}(\tilde{d}_i)$. Indeed, it is *two problems in one*! At the top-level there is *feasibility problem*. Is it even possible to find a job ordering such that each job meets its deadline?

We already know the answer.

Lemma 2.8.1. *If the EDD-ordering results in at least one tardy job then neither for $\mathcal{C}(\tilde{d})$ nor for $\mathcal{C}_w(\tilde{d})$ a feasible schedule exists in which all jobs are on time.*

Proof. *EDD* is a sufficient condition for a schedule to have minimal maximum tardiness $T_{\max}$. There cannot be any schedule with a smaller $T_{\max}$. But, if $T_{\max} > 0$ then there must be at least one tardy job, so no feasible schedule exists for $\mathcal{C}(\tilde{d})$, and the same is true for $\mathcal{C}_w(\tilde{d})$. □

Once the existence of a feasible schedule has been established we are faced with the second problem: find a feasible schedule S which minimizes either $\mathcal{C}$ or $\mathcal{C}_w$.

This is easy for the problem $\mathcal{C}(\tilde{d})$: Algorithm 2.4 solves the problem. The situation is, however, quite different, as it is known that $\mathcal{C}_w(\tilde{d})$ is NP-hard (Lenstra, Rinnooy Kan, and Brucker, 1977). Finding an exact solution of the $\mathcal{C}_w(\tilde{d})$ problem requires special techniques which we will discuss in Chapter 4. For the moment, however, we note that a minor modification of Algorithm 2.4 yields good approximations. This modification is: in Algorithm 2.4 replace the line marked by $(*)$ by

$$p_k/w_k \geq p_\ell/p_\ell \text{ among all jobs } \ell \in J \text{ with } d_\ell \geq T$$

▼ **Example 2.13**

Given are $n = 10$ jobs with processing times and deadlines:

J	1	2	3	4	5	6	7	8	9	10
p	6	8	1	2	4	5	7	8	5	4
$\tilde{d}$	40	40	42	43	47	47	47	47	51	52

Smith's 2nd Rule yields very quickly the optimal schedule

$$\sum C_i = 221, \quad S = [3, 4, 10, 5, 6, 1, 7, 2, 8, 9].$$

Suppose now, that addition we are given also job weights

$$w = [8, 3, 4, 5, 3, 1, 8, 5, 2, 7].$$

Algorithm 2.4 suitably adapted, as described above, now yields as an approximation for the model $1|\tilde{d}_i| \sum w_i C_i$:

$$\sum C_i = 829, \quad S = [3, 4, 10, 1, 7, 5, 8, 2, 6, 9].$$

2.9 Bibliographic Notes

Most of the material in this introductory chapter is covered in one or another way in standard textbooks on scheduling. My favorite book is Baker and Trietsch (2009). The classical textbook although somewhat outdated is certainly Conway, Maxwell, and Miller (2003). Parker (1995) gives a thorough coverage which emphasizes complexity of scheduling algorithms. A rather up to date book is Pinedo (2008).

2.10 References

[1] Kenneth R. Baker and Dan Trietsch. *Principles of Sequencing and Scheduling*. Wiley Publishing, 2009.

[2] R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of Scheduling*. Dover, 2003.

[3] S. Eilon and I. E. Chowdhury. "Minimising Waiting Time Variance in the Single Machine Problem". In: *Management Science* 23.6 (1977), pp. 567–575.

[4] D. S. Hochbaum and R. Shamir. "An $O(n \log^2 n)$ algorithm for the maximum weighted tardiness problem". In: *Information Processing Letters* 31 (1989), pp. 215–219.

[5] J. R. Jackson. “Scheduling a production line to minimize maximum tardiness”. In: *Research Report, Management Science Research Project, University of California, Los Angeles* 43 (1955).

[6] J. J. Kanet. “Minimizing Variation of Flow Time in Single Machine Systems”. In: *Management Science* 27.12 (1981), pp. 1453–1459.

[7] W. Kubiak. “Completion time variance minimization on a single machine is difficult”. In: *Operations Research Letters* 14.1 (1993), pp. 49–59.

[8] J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. “Complexity of Machine Scheduling Problems”. In: *Ann. Discrete Math.* 1 (1977), pp. 343–362.

[9] J. D. C. Little. “A Proof for the Queuing Formula: $L = \lambda W$”. In: *Operations Research* 9.3 (1961), pp. 383–387.

[10] A. G. Merten and M. E. Muller. “Variance minimization in sigle machine sequencing problems”. In: *Management Science* 18.9 (1962), pp. 513–528.

[11] J. M. Moore. “An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs”. In: *Management Science* 15.1 (1968), pp. 102–109.

[12] R. G. Parker. *Deterministic Scheduling Theory.* Chapman & Hall, 1995.

[13] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems.* 3rd. Springer Publishing Company, Incorporated, 2008.

[14] L. Schrage. “Minimizing the time-in-system variance for a finite jobset”. In: *Management Science* 21.5 (1975), pp. 540–543.

[15] Wayne E. Smith. “Various optimizers for single-stage production”. In: *Naval Research Logistics Quarterly* 3.1-2 (1956), pp. 59–66. ISSN: 1931-9193.

[16] L. B. J. M. Sturm. “A Simple Optimality Proof of Moore’s Sequencing Algorithm”. In: *Management Science* 17.1 (1970), pp. 116–118.

[17] N. Ye et al. “Job scheduling methods for reducing waiting time variance”. In: *Computers & Operations Research* 34 (2007), pp. 3069–3083.