CO 454 Course Notes

Scheduling

Joseph Cheriyan • Spring 2022 • University of Waterloo

Table of Contents

1 Introduction to Scheduling

1.1 Examples of Scheduling Problems

To begin, we'll first introduce some examples of scheduling problems.

Example 1.1

Suppose there are n students who need to consult an advisor AI machine M about their weekly timetables at the start of the semester. The amount of time needed by M to advise student j, where $j \in \{1, \dots, n\}$, is denoted by p_j.

If the students meet with M in the order $1, 2, \dots, n$, then student 1 completes their meeting with M at time $C_1 = p_1$, student 2 completes their meeting with M at time $C_2 = p_1 + p_2$, and in general, student j completes their meeting with M at time $C_j = p_1 + \cdots + p_j$. We call C_j the **completion time** of student j.

Now, suppose that there are $n = 3$ students. We can associate each student with a job. Assume that the processing times are $p_1 = 10$, $p_2 = 5$, and $p_3 = 2$. Then for the schedule $1, 2, 3$, the completion times are $C_1 = 10$, $C_2 = 15$, and $C_3 = 17$, for an average completion time of $\frac{10+15+17}{3} = 14$. On the other hand, the ordering $2, 3, 1$ has average completion time $\frac{5+7+17}{3} = \frac{29}{3}$.

Our objective in this case is to minimize the average completion time $\frac{1}{n}\sum_{j=1}^{n} C_j$. Notice that this is equivalent to just minimizing the sum of the completion times $\sum_{j=1}^{n} C_j$.

We pose scheduling problems as a triplet $(\alpha \mid \beta \mid \gamma)$, as we will detail in Section 1.2. This example can be denoted by $(1 \parallel \sum C_j)$.

Example 1.2

Alice is preparing to write the graduation exams at the KW School of Magic (KWSM). The exam is based on n books $B_1, \dots, B_n$ that can be borrowed from the library of KWSM, and these books are not available anywhere else. Alice estimates that she needs p_j days of preparation for the book B_j, but unfortunately, there is a due date d_j for returning B_j to the library. The library charges a late fee of \$1 per day for each overdue book. The goal is to find a sequence for returning the books (after completing the preparation for each book) that minimizes her late fees.

Suppose that Alice picks the sequence $B_1, \dots, B_n$ for returning the books. For this particular sequence, we let T_j denote the late fees for B_j, and let C_j denote the day in which Alice completes her studies from B_j. Notice that $T_j = \max(0, C_j - d_j)$, so $T_j = 0$ if Alice completes B_j by day d_j, and $T_j = C_j - d_j$ otherwise.

More concretely, suppose there are $n = 4$ books with the following preparation times and due dates.

j	1	2	3	4
p_j	4	6	8	12
d_j	10	12	9	15

For the sequence B_1, B_2, B_3, B_4, we find that $T_1 = 0$, $T_2 = 0$, $T_3 = 18 - 9 = 9$, and $T_4 = 30 - 15 = 15$, which gives $\sum T_j = 24$. On the other hand, Alice could return the books in the sequence B_3, B_1, B_2, B_4, and we can see that $T_3 = 0$, $T_1 = 12 - 10 = 2$, $T_2 = 18 - 12 = 6$, and $T_4 = 30 - 15 = 15$. In this case, we obtain $\sum T_j = 23$.

For the triplet notation $(\alpha \mid \beta \mid \gamma)$, we can denote this problem by $(1 \parallel \sum T_j)$.

Example 1.3

A bus containing n UW students has arrived at the entry point of Michitania. There is a sequence of three automatic checks for each visitor, which are

- a passport scan (M_1),
- a temperature scan (M_2),
- a facial scan and photo (M_3).

There are three well-separated machines M_1, M_2, M_3 located in a broad lane, and machine M_i applies the i-th scan. Each student S_j who gets off the bus is required to visit the machines in the sequence M_1, M_2, M_3. Viewing each student S_j as a job j, observe that j consists of the three operations $(1, j)$, $(2, j)$, and $(3, j)$, with a chain of precedence constraints among the operations given by $(1, j) \to (2, j) \to (3, j)$. In particular, the operation $(2, j)$ cannot start until $(1, j)$ is completed, and $(3, j)$ cannot start until $(2, j)$ is completed.

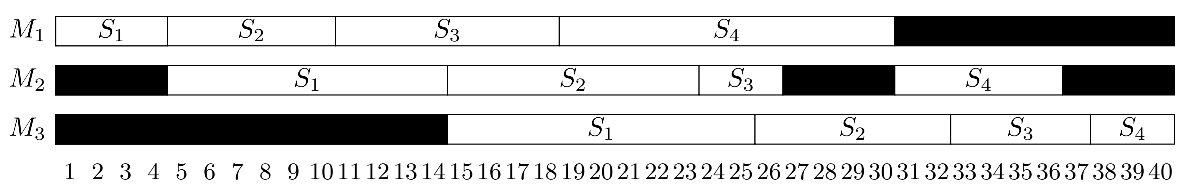
We denote the time required for machine M_i to process student S_j by p_{ij}.

The goal is to find a schedule such that the checks for all students are completed as soon as possible. Consider a fixed schedule which has the same sequence of students $S_1, S_2, \ldots, S_n$ for all three machines. Let C_{ij} denote the time when the scan of S_j is completed on M_i. Let $C_{\max}$ denote the maximum completion time of the students on M_3; that is, $C_{\max} = \max_{j \in \{1,\ldots,n\}} C_{3j}$. We often refer to $C_{\max}$ as the **makespan** of the schedule. The goal is to find a schedule which minimizes $C_{\max}$.

For instance, suppose that there are $n = 4$ students with the following processing times.

	$j=1$	$j=2$	$j=3$	$j=4$
$i=1$	4	6	8	12
$i=2$	10	9	4	6
$i=3$	11	7	5	3

Using the schedule S_1, S_2, S_3, S_4 for all three machines, we can determine that $C_{\max} = \max(C_{24}, C_{33}) + p_{34} = \max(36, 37) + 3 = 40$. We can visualize these values via a Gantt chart, like below. The black bars indicate that the machine is idling.



Notice that M_2 had to idle because M_1 did not finish processing S_4 yet.

Using the triplet notation $(\alpha \mid \beta \mid \gamma)$, this problem is denoted by $(F_3 \parallel C_{\max})$, where 3 denotes the number of machines and F refers to a "flow shop".

Note that this problem does not require the students to visit M_1, M_2, M_3 in the same sequence. For example, the students could visit M_1 in the sequence $S_1, S_2, S_3, S_4, \ldots, S_n$, while visiting M_2 in the sequence $S_2, S_1, S_4, S_3, \ldots, S_n$. It may seem "obvious" that there exists an optimal schedule such that the jobs visit each of the machines in the same sequence; this statement is in fact false when there are at least 4 machines, but is true when there are at most 3 machines.

1.2 Notation and Framework

The examples above are examples of scheduling problems and illustrate the issue of allocating limited resources over time in order to optimize an objective function. We remark here that different objectives can lead to different solutions and so a "universally best" schedule may not exist. It should also be clear from the above examples that scheduling problems appear in a wide range of fields, and ideally, we should have a unified theory for studying them. We now describe a general framework and notation that captures most (but not all) scheduling problems.

Any scheduling problem is associated with a finite set of tasks or jobs and a finite set of resources or machines. The set of jobs is denoted by J, and we use n to denote $|J|$; moreover, we write $J = \{1, 2, \ldots, n\}$. Similarly, the set of machines is denoted by M, and we use m to denote $|M|$.

At any point in time, a single machine can process at most one job.

Most scheduling problems can be described with a triplet $(\alpha \mid \beta \mid \gamma)$. The first term α is the **machine environment** and contains a single entry. This field describes the resources that are available for the completion of various tasks. The second term β denotes the various constraints on the machines and the jobs that must be respected by the schedule. The third term γ is the objective function for the scheduling problem to be minimized; a particular feasible schedule is optimal if it has the smallest value for γ among all feasible schedules.

Each job $j \in J$ may have one or more of the following pieces of data associated with it.

- **Processing Time** (p_{ij}) The time taken by machine i to process job j is denoted by p_{ij}. In many scheduling problems, the processing time of job j is independent of the machine, and in such cases the processing time (of job j on any machine) is denoted simply by p_j.
- **Release Time** (r_j) In several scheduling problems, a job j is only available for processing after time r_j. This is called the release time of the job.
- **Due Date** (d_j) This is the planned time when a job j should be completed. In several scheduling problems, a job is allowed to complete after its due date, but such jobs incur a penalty for the violation.
- **Weight** (w_j) The weight of a job j denotes the relative worth of j with respect to the other jobs. Usually, w_j is a coefficient in the objective function γ, for instance $\sum w_j C_j$. As we will see later, introducing weights can often lead to added complexity in the scheduling problem.

Machine Environment (α) The possible machine environments in our course are as follows.

- **Single Machine Environment** ($\alpha = 1$) In this case, we have only one machine. Although this appears to be a rather special case, the study of single-machine problems leads to many techniques useful for more general cases.
- **Identical Parallel Machines** ($\alpha = P$) In this case, we have m identical machines and any job can run on any machine. Each job has the same processing time on each machine. When the number of machines is constant, the number of machines is appended after the letter P. For example, if there are $m = 2$ machines, then we write $\alpha = P_2$.
- **Uniform Speed Parallel Machines** ($\alpha = Q$) In this case, we have m machines and any job can run on any machine. Each machine i has a speed s_i. The time taken to process job j on machine i is then p_j/s_i.
- **Unrelated Parallel Machines** ($\alpha = R$) In this case, we have m machines and any job can run on any machine. However, each job j takes time p_{ij} on machine i, and the p_{ij}'s are unrelated to each other. For instance, machine i could have $p_{ij} > p_{ij'}$, but machine ℓ could have $p_{\ell j} < p_{\ell j'}$.
- **Open Shop** ($\alpha = O$) The following three machine environments fall in the shop scheduling framework. In this framework, each job j consists of m operations and a job is said to be completed if and only if

all the operations of the job are completed. Furthermore, each operation takes place on a dedicated machine. Thus, each job needs to visit each machine before completion. Some of the operations may have zero processing times.

In the open shop environment, the jobs can visit the m machines in any order. There are no restrictions with regard to the routing of each job; that is, the "scheduler" is allowed to determine a route for each job and different jobs may have different routes.

- **Job Shop** ($\alpha = J$) In a job shop, each job comes with a specified order in which the m operations of the job are processed by the m machines. In other words, each job has a specified routing (a sequence for the m machines), and it follows this routing to visit the m machines.

- **Flow Shop** ($\alpha = F$) In a flow shop, each job visits the m machines in the same fixed order, which is assumed to be $\{1, 2, \ldots, m\}$; that is, all jobs have the same routing.

  We can view a flow shop in a different perspective. Each job j visits the m machines in the same sequence, and after the operation of j on one machine is completed, the job enters the "queue" of the next machine in the sequence. Moreover, whenever a machine completes an operation, the machine picks another operation from its queue, and processes that.

  In some applications, each machine is required to process the jobs in the order the jobs enter the machine's queue. Such schedules are called **FIFO schedules**, and such flow shops are called **permutation flow shops**; these additional constraints are indicated in the β field with $\beta = prmu$.

Side Constraints (β) The side constraints capture the various restrictions on the scheduling problem. We note that there could be more than one side constraint. Some of the side constraints relevant to our course are as follows.

- **Release Dates** ($\beta = r_j$) Unless specified, we assume that all jobs are available from the beginning.

- **Setup Times** ($\beta = s_{jk}(i)$) In some applications, a setup time is required on machine i after the completion of job j and before the start of the next job k. For example, the machine i may need to be cleaned and recalibrated between jobs j and k. Unless mentioned otherwise, these setup times are assumed to be zero.

- **Precedence Constraints** ($\beta = \text{prec}$) In some applications, job k cannot be processed until another job j is completed. Such constraints are called precedence constraints. We assume that these constraints are not cyclical; that is, we do not have a situation where job j precedes job k, job k precedes job ℓ, and job ℓ precedes job j. One represents the precedence constraints via a directed acyclic graph (DAG) where the nodes represent the various jobs, and an arc from node j to node k represents the constraint "j precedes k" (that is, k cannot start until j is completed).

- **Preemption** ($\beta = \text{prmp}$) Informally speaking, if a job allows preemption, then the "scheduler" is allowed to interrupt the processing of the job (preempt) at any point in time and put a different job on the machine instead. The amount of processing a preempted job already has received is not lost.

  Consider a job j that consists of a single operation. Job j does not allow preemption if all of the processing of j must occur on one machine in one contiguous time period; otherwise, the job allows preemption (in which case the job could be processed by two or more distinct machines, or it could be processed by one machine over two or more non-contiguous time periods). By default, we assume that preemption is not allowed.

Objective (γ) The objective function specifies the optimality criterion for choosing among several feasible schedules. There is a large list of possible objective functions, depending on the applications. Before we get into the objective functions, we will introduce some definitions.

- Given a job j, the **completion time** of job j in a schedule S, denoted C_j^S, is the time when all operations of the job have been completed. We drop the superscript S when the schedule is clear from the context.

- When jobs have due dates, the **lateness** of a job j, denoted L_j, is the difference between the completion time and the due date. That is, we have

$$L_j = C_j - d_j.$$

  Note that if $L_j < 0$, then the job j is not late.

- A closely related measure is the **tardiness** of a job, which is the maximum of 0 and the lateness of the job. Therefore, we have

$$T_j = \max(0, L_j) = \max(0, C_j - d_j).$$

- Finally, we use U_j to indicate whether a job j is completed after the due date. If a job j has $C_j > d_j$, then we say that the job is **tardy** and we have $U_j = 1$; otherwise, we have $U_j = 0$.

We now discuss a few fundamental objective functions that are most relevant to the course.

- **Makespan** ($\gamma = C_{\max}$) Find a schedule which minimizes the maximum completion time $C_{\max} := \max_{j \in J} C_j$.

- **Total Weighted Completion Time** ($\gamma = \sum w_j C_j$) Find a schedule which minimizes the weighted average time taken for a job to complete. When all weights are 1, we instead write $\sum C_j$ in the field. This measure is also called the **flow time** or the **weighted flow time**.

- **Maximum Lateness** ($\gamma = L_{\max}$) Find a schedule which minimizes the maximum lateness of a job; that is, we want to minimize $L_{\max} := \max_{j \in J} L_j$.

- **Weighted Number of Tardy Jobs** ($\gamma = \sum w_j U_j$) Find a schedule that minimizes the weighted number of tardy jobs. When all weights are 1, we instead write $\sum U_j$ in the field.

Observe that all of the above objective functions are non-decreasing in $C_1, \ldots, C_n$; in other words, if we take two schedules S and S', and the completion times are such that $C_j^S \le C_j^{S'}$ for all j, then the objective value of S is at most the objective value of S'. Such objective functions or performance measures are said to be **regular**.

PROPOSITION 1.4

The performance measure $\sum U_j$ is regular.

PROOF. Consider two schedules S and S' with $C_j^S \le C_j^{S'}$ for all jobs $j \in J$. Note that if $C_j^S > d_j$ for a job j, then $C_j^{S'} > d_j$ as well. Hence, if $U_j^S = 1$ for a job j, then $U_j^{S'} = 1$ too. It follows that $\sum U_j^S \le \sum U_j^{S'}$, so we conclude that $\sum U_j$ is regular. □

EXERCISE 1.5

Prove that all of the above performance measures are regular.

Not all performance measures are regular. For instance, there are scheduling problems where each job j has a time window $[a_j, b_j]$ and the job needs to be processed in that particular time window. Let $X_j = 0$ if the job is processed in that time window, and $X_j = 1$ otherwise. Is the performance measure $\sum X_j$ regular?

A **nondelay schedule** is a feasible schedule in which no machine is kept idle while a job or operation is waiting for processing. Notice that in Example 1.3, M_2 had a forced delay at the beginning because it had to wait for M_1 to finish processing S_1. We will discuss this later on, but introducing deliberate idleness can yield a more optimal schedule than a nondelay one.

1.3 Dynamic Programming

There are a few algorithmic paradigms that have specific uses.

- **Greedy.** Process the input in some order, myopically making irrevocable decisions.
- **Divide-and-conquer.** Break up a problem into independent subproblems. Solve each subproblem. Combine solutions to the subproblems to form a solution to the original problem.
- **Dynamic programming.** Break up a problem into a series of overlapping subproblems; combine solutions to smaller subproblems to form a solution to larger subproblem.

We will focus on dynamic programming. This has many applications, such as to AI, operations research, information theory, control theory, and bioinformatics.

Here, we will give a scheduling problem which we can solve by way of dynamic programming. Suppose that each job j has release date r_j, processing time p_j, and has weight $w_j > 0$. We will call two jobs **compatible** if they don't overlap. The goal is to find a maximum weight subset of mutually compatible jobs.

One can consider the jobs in ascending order of the finish time $r_j + p_j$. We add a job to the subset if it is compatible with the previously chosen jobs. This greedy algorithm is correct if all weights are 1, but it can fail spectacularly otherwise. For instance, consider a scenario where there are two jobs 1 and 2 with the same start time. Suppose that $p_2 > p_1$ and $w_2 > w_1$. Then the greedy algorithm would pick job 1 as it has a smaller finish time, even though picking job 2 would yield a larger weight.

For convention, we will assume that the jobs are listed in ascending order of finish time $r_j + p_j$. We define ℓ_j to be the largest index $i < j$ such that job i is compatible with job j.

We define $\mathrm{OPT}(j)$ to be the maximum weight for any subset of mutually compatible jobs for the subproblem consisting of only the jobs $1, 2, \ldots, j$. Our goal now is to find $\mathrm{OPT}(n)$.

Here's something obvious we can say about $\mathrm{OPT}(j)$: either job j is selected by $\mathrm{OPT}(j)$, or it is not.

- When job j is not selected by $\mathrm{OPT}(j)$, we easily notice that $\mathrm{OPT}(j)$ is the same as $\mathrm{OPT}(j-1)$.
- When job j is selected by $\mathrm{OPT}(j)$, we collect the profit w_j and notice that all the jobs in $\{\ell_j+1, \ldots, j-1\}$ are incompatible with j, so $\mathrm{OPT}(j)$ must include an optimal solution to the subproblem consisting of the remaining compatible jobs $\{1, \ldots, \ell_j\}$.

Therefore, we can deduce that

$$\mathrm{OPT}(j) = \max(w_j + \mathrm{OPT}(\ell_j), \mathrm{OPT}(j-1)).$$

Moreover, we notice that job j belongs to the optimal solution if and only if the first of the options above is at least as good as the second; in other words, job j belongs to an optimal solution on the set $\{1, 2, \ldots, j\}$ if and only if

$$w_j + \mathrm{OPT}(\ell_j) \geq \mathrm{OPT}(j-1). \tag{1.1}$$

These facts form the first crucial component on which a dynamic programming solution is based: a recurrence equation that expresses the optimal solution (or its value) in terms of the optimal solutions to smaller subproblems. We can now give a recursive algorithm to compute $\mathrm{OPT}(n)$, assuming that we have already sorted the jobs by finishing time and computed the values ℓ_j for each j.

```
ComputeOpt(j)
    if j = 0 then:
        return 0
    else:
        return max(w_j + ComputeOpt(ℓ_j), ComputeOpt(j − 1))
    endif
```

Unfortunately, if we implemented this as written, it would take exponential time to run in the worst case, since the tree will widen very quickly due to the recursive branching. However, we are not far from reaching a polynomial time solution.

We employ a trick called **memoization**. We could store the value of `ComputeOpt` in a globally accessible place the first time we compute it and simply use the precomputed value for all future recursive calls. We make use of an array $M = (M_0, M_1, \ldots, M_n)$ where each M_j will be initialized as empty, but will hold the value of `ComputeOpt`(j) as soon as it is determined.

```
MComputeOpt(j)
    if j = 0 then:
        return 0
    else if M_j is not empty then:
        return M_j
    else:
        M_j = max(w_j + ComputeOpt(ℓ_j), ComputeOpt(j − 1))
        return M_j
    endif
```

Notice that the running time of `MComputeOpt`(n) is $O(n)$, assuming that the input intervals are sorted by their finishing times. Indeed, every time the procedure invokes the recurrence and issues two recursive calls to `MComputeOpt`, it fills in a new entry, and thus increases the number of filled in entries by 1. Since M only has $n+1$ entries, there are at most $O(n)$ calls to `MComputeOpt`.

Now, we typically don't just want the value of an optimal solution; we also want to know what the solution actually is. We could do this by extending `MComputeOpt` to keep track of an optimal solution along with the value by maintaining an additional array which holds an optimal set of intervals. But naively enhancing the code to maintain these solutions would blow up the running time by a factor of $O(n)$, as writing down a set takes $O(n)$ time.

Instead, we can recover the optimal solution from values saved in the array M after the optimum value has been computed. From the observation we made for the inequality (1.1), we can get a pretty simple procedure which "traces back" through the array M to find the set of intervals in an optimal solution.

```
FindSolution(j)
    if j = 0 then:
        Output nothing
    else:
        if w_j + M_{ℓ_j} ≥ M_{j−1} then:
            Output j together with the result of FindSolution(ℓ_j)
        else:
            Output the result of FindSolution(j − 1)
        endif
    endif
```

We see that `FindSolution` calls itself recursively on strictly smaller values, so it makes a total of $O(n)$ recursive calls. It spends constant time for each call, so as a result, `FindSolution` returns an optimal solution in $O(n)$ time.

Note that the values ℓ_j can be computed by means of a binary search which takes $O(\log n)$ time, so the total running time between using `MComputeOpt` and `FindSolution` is $O(n \log n)$.

2 Single Machine Models

Single machine models are very important, as they are relatively simple and can be viewed as a special case of all other environments. We will analyze various single machine models in detail, such as the total weighted completion time, as well as some due date related objectives in the assignments. One observation that we can make for single machine models is that when a problem is non-preemptive and the objective is regular, finding an optimal schedule boils down to finding a sequence of jobs.

2.1 Total Weighted Completion Time

Before we begin, we should say something about interchange arguments, which are commonplace in scheduling. Suppose we have two different sequences for the same set of jobs, say

1. a reference sequence $R = r_1, r_2, \ldots, r_n$, and
2. an adversary sequence $A = a_1, a_2, \ldots, a_n$,

satisfying $\{r_1, \ldots, r_n\} = \{a_1, \ldots, a_n\}$ but $R \neq A$.

Proposition 2.1

There exists an adjacent pair of items in A, say a_i and a_{i+1}, such that a_{i+1} precedes a_i in R.

Proof. Assume no such pair exists, so every adjacent pair a_i and a_{i+1} of items in A is such that a_i precedes a_{i+1} in R as well. Then the only way for R to have exactly n jobs with a_i preceding a_{i+1} for all $i \in \{1, \ldots, n-1\}$ is for R to be the sequence $a_1, a_2, \ldots, a_n$. This is a contradiction with our assumption that $R \neq A$. □

We can now give an example of an interchange argument. A so-called **adjacent pairwise interchange** uses Proposition 2.1 to obtain two adjacent items which can be swapped.

Consider the problem $(1 \parallel \sum C_j)$, where there are n jobs with processing times $p_1, \ldots, p_n$. The **Shortest Processing Time first (SPT) rule** says to put the shortest processing times first.

Theorem 2.2

The SPT rule is optimal for $(1 \parallel \sum C_j)$.

Proof. Assume for simplicity that the processing times $p_1, \ldots, p_n$ are distinct. Suppose there is a schedule S that does not satisfy the SPT rule which is optimal. There exist two adjacent jobs, say k followed by ℓ, such that $p_k > p_\ell$, and using adjacent pairwise interchange, we can obtain a new schedule S' by swapping k and ℓ.

Note that all completion times are the same in S and S' except for C_k and C_ℓ. Suppose that t is the starting time of job k in S. Then in S, we have $C_k^S = t + p_k$ and $C_\ell^S = t + p_k + p_\ell$. On the other hand, in S', we have $C_k^{S'} = t + p_k + p_\ell$ and $C_\ell^{S'} = t + p_\ell$. We see that $C_\ell^S = C_k^{S'}$, so subtracting the objectives yields

$$\sum C_j^S - \sum C_j^{S'} = C_k^S - C_\ell^{S'} = p_k - p_\ell > 0.$$

This means that S' has a better objective value than S, contradicting the optimality of S. □

More generally, we can consider the total weighted completion time $(1 \parallel \sum w_j C_j)$. This problem gives rise to the **Weighted Shortest Processing Time first (WSPT) rule**, and according to this rule, the jobs are placed in decreasing order of w_j/p_j.

Theorem 2.3

The WSPT rule is optimal for $(1 \parallel \sum w_j C_j)$.

Proof. Again, we apply an interchange argument. Suppose that there is a optimal schedule S that is not WSPT. Then there must exist two adjacent jobs, say job k followed by job ℓ, such that

$$\frac{w_k}{p_k} < \frac{w_\ell}{p_\ell}.$$

Using adjacent pairwise interchange, obtain a new schedule S' by swapping the jobs k and ℓ. As before, all completion times are the same in S and S' except for C_k and C_ℓ. Suppose that job k starts processing at time t in S. Under S, the total weighted completion time for jobs k and ℓ is

$$w_k(t + p_k) + w_\ell(t + p_k + p_\ell),$$

whereas under S', it is equal to

$$w_k(t + p_\ell + p_k) + w_\ell(t + p_\ell).$$

Then subtracting the objective of S from the objective of S' yields the quantity

$$w_\ell p_k - w_k p_\ell,$$

which is positive due to the assumption that $w_k/p_k < w_\ell/p_\ell$. This contradicts the optimality of S. $\square$

The computation time needed to order the jobs according to the WSPT rule is the time required to sort the jobs according to the ratio of the two parameters. This takes $O(n \log n)$ time since this is the time it takes to perform a simple sort. Since the SPT rule is a special case of the WSPT rule with all weights equal to 1, it also requires $O(n \log n)$ time.

How is the minimization of the total weighted completion time affected by precedence constraints? Consider the simplest form of precedence constraints which take the form of parallel chains. This problem can still be solved by a relatively simple and very efficient (polynomial time) algorithm. This algorithm is based on some fundamental properties of scheduling with precedence constraints.

Consider two chains of jobs. The first chain consists of jobs $1, \ldots, k$, and the second chain consists of jobs $k+1, \ldots, n$. The precedence constraints are then $1 \to 2 \to \cdots \to k$ and $k+1 \to k+2 \to \cdots \to n$.

The next lemma is based on the assumption that if the scheduler decides to start processing jobs of one chain, then they have to complete the entire chain before they are allowed to work on jobs of the other chain. Which of the two chains should be processed first?

Lemma 2.4

If we have

$$\frac{\sum_{j=1}^{k} w_j}{\sum_{j=1}^{k} p_j} > \frac{\sum_{j=k+1}^{n} w_j}{\sum_{j=k+1}^{n} p_j},$$

then it is optimal to process the chain of jobs $1, \ldots, k$ before the chain of jobs $k+1, \ldots, n$.

Proof. We proceed by contradiction. Under the sequence $1, \ldots, k, k+1, \ldots, n$, the total completion time is

$$w_1 p_1 + \cdots + w_k \sum_{j=1}^{k} p_j + w_{k+1} \sum_{j=1}^{k+1} p_j + \cdots + w_n \sum_{j=1}^{n} p_j,$$

while under the sequence $k+1, \dots, n, 1, \dots, k$, it is

$$w_{k+1}p_{k+1} + \cdots + w_n \sum_{j=k+1}^{n} p_j + w_1 \left(\sum_{j=k+1}^{n} p_j + p_1 \right) + \cdots + w_k \sum_{j=1}^{n} p_j.$$

Using the inequality

$$\frac{\sum_{j=1}^{k} w_j}{\sum_{j=1}^{k} p_j} > \frac{\sum_{j=k+1}^{n} w_j}{\sum_{j=k+1}^{n} p_j},$$

the total weighted completion time of the first sequence is less than that of the second. The result follows. □

An interchange between two adjacent chains of jobs is usually referred to as an **adjacent sequence interchange**. Such an interchange is a generalization of an adjacent pairwise interchange.

Definition 2.5

Let $1 \to 2 \to \cdots \to k$ be a chain. Let ℓ^* satisfy

$$\frac{\sum_{j=1}^{\ell^*} w_j}{\sum_{j=1}^{\ell^*} p_j} = \max_{\ell \in \{1,\dots,k\}} \left(\frac{\sum_{j=1}^{\ell} w_j}{\sum_{j=1}^{\ell} p_j} \right).$$

The ratio on the left-hand side is called the **ρ-factor** of the chain $1, \dots, k$ and is denoted by $\rho(1, \dots, k)$. The job ℓ^* is referred to as the **determining job** of the chain.

More generally, we now assume that the scheduler does not have to fully complete chains immediately; they can process some jobs of one chain (while adhering to the precedence constraints), switch over to another chain, and revisit the original chain later. If the total weighted completion time is the objective function, then the following result holds.

Lemma 2.6

For a chain of jobs $1 \to 2 \to \cdots \to k$, suppose ℓ^* is a determining job. Then there exists an optimal schedule that processes the jobs $1, \dots, \ell^*$ consecutively, without processing any jobs of any other chains.

Proof. We proceed by contradiction. Suppose that under the optimal sequence, the processing of the subsequence $1, \dots, \ell^*$ is interrupted by a job, say v, from another chain. That is, the optimal sequence contains the subsequence $1, \dots, u, v, u+1, \dots, \ell^*$; call this subsequence S. It suffices to show that either with subsequence $v, u+1, \dots, \ell^*$ which we denote S', or subsequence $1, \dots, u, v$, which we denote S'', the total weighted completion time is less than with subsequence S. If it is not less with the first subsequence, then it has to be less with the second and vice versa. From Lemma 2.4, it follows that if the total weighted completion time with S is less than with S', then

$$\frac{w_v}{p_v} < \frac{w_1 + w_2 + \cdots + w_u}{p_1 + p_2 + \cdots + p_u}.$$

Lemma 2.4 also tells us that if the total weighted completion time with S is less than with S'', then

$$\frac{w_v}{p_v} > \frac{w_{u+1} + w_{u+2} + \cdots + w_{\ell^*}}{p_{u+1} + p_{u+2} + \cdots + p_{\ell^*}}.$$

If job ℓ^* is the determining job for the chain $1, \dots, k$, then by definition of ℓ^*, we have

$$\frac{w_1 + \cdots + w_u + w_{u+1} + \cdots + w_{\ell^*}}{p_1 + \cdots + p_u + p_{u+1} + \cdots + p_{\ell^*}} > \frac{w_{u+1} + w_{u+2} + \cdots + w_{\ell^*}}{p_{u+1} + p_{u+2} + \cdots + p_{\ell^*}}.$$

Noting that $(a+c)/(b+d) > a/b$ implies $c/d > a/b$, we obtain

$$\frac{w_{u+1}+w_{u+2}+\cdots+w_{\ell^*}}{p_{u+1}+p_{u+2}+\cdots+p_{\ell^*}} > \frac{w_1+w_2+\cdots+w_u}{p_1+p_2+\cdots+p_u}.$$

If S is better than S'', this means that

$$\frac{w_v}{p_v} > \frac{w_{u+1}+w_{u+2}+\cdots+w_{\ell^*}}{p_{u+1}+p_{u+2}+\cdots+p_{\ell^*}} > \frac{w_1+w_2+\cdots+w_u}{p_1+p_2+\cdots+p_u}.$$

Therefore, S' is better than S. The same argument applies if the interruption of the chain is caused by more than one job. □

Intuitively, Lemma 2.6 makes sense. The condition of the lemma implies that the ratios of the weight divided by the processing time of the jobs in the string $1,\dots,\ell^*$ must be increasing in some sense. If one had already decided to start processing a string of jobs, it makes sense to continue processing the string until job ℓ^* is completed without processing any other job in between. Our two lemmas contain the basis for a simple algorithm that minimizes the total weighted completion time when the precedence constraints take the form of chains.

Algorithm 2.7: Total Weighted Completion Time and Chains

Whenever the machine is freed, select among the remaining chains the one with the highest ρ-factor. Process this chain without interruption up to and including the job that determines its ρ-factor.

We illustrate the use of this algorithm with an example.

Example 2.8

Consider the two chains $1 \to 2 \to 3 \to 4$ and $5 \to 6 \to 7$. The weights and processing times of the jobs are as follows.

j	1	2	3	4	5	6	7
w_j	6	18	12	8	8	17	18
p_j	3	6	6	5	4	8	10

The ρ-factor of the first chain is $(6+18)/(3+6)$ and is determined by job 2. The ρ-factor of the second chain is $(8+17)/(4+8)$ and is determined by job 6. Since $24/9$ is larger than $25/12$, jobs 1 and 2 are processed first. The ρ-factor of the remaining part of the first chain is $12/6$ and is determined by job 3. This is less than $25/12$, so we process jobs 5 and 6 next. The ρ-factor of the remaining part of the second chain is $18/10$ and is determined by job 7. Hence, job 3 follows job 6. The w_j/p_j ratio of job 7 is higher than that of job 4, so job 7 follows job 3 and job 4 goes last.

2.2 Branch and Bound

Before we discuss more about single machine models, we make a diversion and introduce branch and bound. Branch and bound is a divide-and-conquer approach for solving integer programs. While enumerating all solutions can work, it is just too slow in most cases when there are many variables. Branch and bound starts off by enumerating, but it cuts out a lot of the enumeration whenever possible.

To illustrate this point, suppose we go with a complete enumeration approach with n binary variables $x_1,\dots,x_n \in \{0,1\}$. Then there are 2^n different possibilities in the solution set, which of course blows up the runtime as n grows large.

The idea of branch and bound is as follows: if we can't solve the original integer program, maybe we can divide it into smaller ones and solve them! Picking a variable x_k (which we call the branching variable) and a value $a \notin \mathbb{Z}$, we can divide the IP into two subproblems by adding the constraint $x_k \leq \lfloor a \rfloor$ to one and $x_k \geq \lceil a \rceil$ to the other. This process partitions the feasible region, and we note that any solution with $x_k = a$ could not possibly be a solution to these two subproblems. For instance, suppose we have the IP

$$\begin{array}{rl} \max & 3x_1 + 4x_2 \\ \text{s.t.} & 2x + 2x_2 \leq 3 \\ & x \geq 0 \text{ and integer.} \end{array}$$

Then we can branch on x_2 with $a = 1.5$ to get two subproblems

$$\begin{array}{rl} \max & 3x_1 + 4x_2 \\ \text{s.t.} & 2x + 2x_2 \leq 3 \\ & x_2 \leq 1 \\ & x \geq 0 \text{ and integer,} \end{array} \qquad \begin{array}{rl} \max & 3x_1 + 4x_2 \\ \text{s.t.} & 2x + 2x_2 \leq 3 \\ & x_2 \geq 2 \\ & x \geq 0 \text{ and integer.} \end{array}$$

Suppose that we have an integer program IP_1 which we split (branched) into two subproblems IP_2 and IP_3. Then the optimal solution to IP_1 is the best between the optimal solutions to IP_2 and IP_3.

But the problem is that each subproblem can still be too hard to solve. As we recall, IPs are hard to solve in general. Thus, we can consider the LP relaxation of the IPs, which drops the integrality constraints. We know how to solve these efficiently, such as by using the Simplex algorithm from CO 250.

We remark that for knapsack problems (where we require $x_i \in \{0, 1\}$ for all $i \in \{1, \ldots, n\}$), we do not even need to use the Simplex algorithm to solve their LP relaxations. We can use the greedy algorithm which puts items into the knapsack in decreasing order of value per weight unit.

Recall that any solution that is feasible for the IP is also feasible for its LP relaxation. Equivalently, if the LP relaxation is infeasible, then so is the IP.

Letting z^*_{IP} be the optimal value of the IP and z^*_{LP} be the optimal value of the LP relaxation, we see that $z^*_{\text{IP}} \leq z^*_{\text{LP}}$. Moreover, if the LP relaxation of an IP has an optimal solution x^* which happens to be integral, then x^* is also an optimal solution to the IP, because this would imply $z^*_{\text{LP}} \leq z^*_{\text{IP}}$ and so $z^*_{\text{IP}} = z^*_{\text{LP}}$.

We make one more observation: if we know a feasible solution for the original IP with value $\bar{z}$ and the LP relaxation of a subproblem IP_i has an optimal solution with value $z^*_{\text{LP}_i}$ with $z^*_{\text{LP}_i} \leq \bar{z}$, then the optimal solution to IP_i will be less than $\bar{z}$.

We now introduce some terminology for branch and bound.

- The branch and bound process is typically represented as a tree (called the **branch and bound tree**), with each subproblem corresponding to a node of the tree (called a **branch and bound node**).
- If the LP relaxation of a node is infeasible, we say that the node is **pruned by infeasibility**.
- If the bound from the LP relaxation of a node is worse than our current best solution, we say that the node was **pruned by bound**.
- If the LP relaxation has an integral solution, we say that the node was **pruned by optimality**.
- If the node was not pruned, we must **branch** on it; that is, divide it into two smaller subproblems.

Now, we are ready to state the branch and bound algorithm. Note that there are further speedups that could be made to the algorithm as written. One way is for the branch and bound algorithm to have heuristics that "intelligently" choose the best variable to branch on. One can also round down to further improve bounds, or adding so-called valid inequalities.

Algorithm 2.9: Branch and Bound

1. Initially, set $z_{\text{best}} = -\infty$ and x_{best} to be undefined.
2. If all subproblems have been explored, then stop. If x_{best} is still undefined, then the IP is infeasible. Otherwise, x_{best} is the optimal solution to the IP with value z_{best}.
3. Solve the LP relaxation LP_k of a subproblem IP_k that has not been explored yet.
   (a) Declare IP_k as explored.
   (b) If LP_k is infeasible, do not branch. This node is pruned by infeasibility. Go back to the start of step 3.
   (c) Let $z^*_{\text{LP}_k}$ be the optimal value and $x^*_{\text{LP}_k}$ be the optimal solution to LP_k.
   (d) If $z^*_{\text{LP}_k} \leq z_{\text{best}}$, then do not branch. Any solution to the current subproblem cannot be better than the one we already have. This node is pruned by bound. Go back to the start of step 3.
   (e) If $x^*_{\text{LP}_k}$ is integral, then do not branch. We have found the optimal solution to the subproblem IP_k. If $z^*_{\text{LP}_k} > z_{\text{best}}$, then set $z_{\text{best}} = z^*_{\text{LP}_k}$ and $x_{\text{best}} = x^*_{\text{LP}_k}$. This node is now pruned by optimality. Go back to the start of step 3.
   (f) Pick j such that the j-th component of $x^*_{\text{LP}_k}$ is equal to a value a which is fractional (and x_j is required to be an integer in IP_k).
   (g) Create two new subproblems: IP_k with the additional constraint $x_j \leq \lfloor a \rfloor$, and IP_k with the additional constraint $x_j \geq \lceil a \rceil$.

2.3 Maximum Lateness

Recall that the input to the problem $(1 \parallel L_{\max})$ consists of n jobs, each with a processing time p_j and a due date d_j. The goal is to find a sequence of jobs which minimizes $L_{\max} = \max_{j \in \{1,\ldots,n\}} L_j$, where $L_j = C_j - d_j$.

The **Earliest Due Date (EDD) rule** places the jobs in increasing order of the due dates. We will prove on Assignment 2 that the EDD rule is in fact optimal for $(1 \parallel L_{\max})$; this can be done using an interchange argument.

Now, we discuss a generalization of $(1 \parallel L_{\max})$ problem, which is $(1 \mid \text{prec} \mid h_{\max})$. In this setting, we have n jobs, and associated with each job is a processing time p_j and a non-decreasing cost function h_j. The goal is to find a schedule which minimizes $h_{\max} = \max_{j \in \{1,\ldots,n\}} h_j(C_j)$. We can see this is indeed a generalization of $(1 \parallel L_{\max})$, since the lateness $L_j = C_j - d_j$ is non-decreasing.

An easy observation to make is that the completion time of the last job always occurs at the makespan $C_{\max} = \sum p_j$, and this is independent of the schedule. The following algorithm, called the **Lowest Cost Last (LCL) algorithm**, makes use of this observation by going backwards starting from time $\sum p_j$.

Let S be the set of already scheduled jobs, which is initially $S = \varnothing$. The jobs in S are processed during the time interval
$$\left[C_{\max} - \sum_{j \in S} p_j, C_{\max} \right].$$

Let $\hat{J}$ be the set of all jobs whose successors have been scheduled; this is initially all jobs without any successors. The set $\hat{J}$ is typically referred to as the set of schedulable jobs. Notice that $\hat{J} \subseteq S^c = \{1, \ldots, n\} \setminus S$. Moreover, we let t denote the time of completion of the next job, and initially set it to $t = \sum p_j$.

We now state the LCL algorithm, give an simple example of it in action, then prove that it yields an optimal schedule for $(1 \mid \text{prec} \mid h_{\max})$.

Algorithm 2.10: Lowest Cost Last

1. If $\hat{J} \neq \varnothing$, then stop. Otherwise, continue.
2. Select $j \in \hat{J}$ such that $h_j(t) = \min_{k \in \hat{J}} h_k(t)$.
3. Schedule j so that it completes at time t.
4. Add j to S, delete j from $\hat{J}$, and update $\hat{J}$ according to the precedence constraints.
5. Set $t \leftarrow t - p_j$.

Example 2.11

Consider the following instance of $(1 \parallel h_{\max})$ with $n = 3$ (so there are no precedence constraints).

j	1	2	3
p_j	2	3	5
h_j	$1 + C_j$	$1.2C_j$	10

Initially, we have $\hat{J} = \{1, 2, 3\}$ and $t = \sum p_j = 10$. Since $h_3(10) < h_1(10) < h_2(10)$, job 3 goes last.

Next, we have $\hat{J} = \{1, 2\}$ and $t = 10 - 5 = 5$. Then $h_1(5) = h_2(5) = 6$, so either job 1 or 2 could go last. This yields the two optimal schedules $1, 2, 3$ and $2, 1, 3$.

Theorem 2.12

The LCL algorithm yields an optimal schedule for $(1 \mid \text{prec} \mid h_{\max})$.

Proof. We proceed by contradiction. Assume that the LCL algorithm yields a schedule S which is not optimal. Let S^* be an optimal schedule that has the longest common suffix with S; that is, S^* agrees with S at the tail end of the schedule for as long as possible.

Let j^* be the first index on the tail in S such that S and S^* disagree. Let j^{**} be the job in that same position in S^*. Then j^* occurs at some earlier point in S^*. Construct the schedule $\hat{S}$ from S^* by placing j^* immediately after j^{**} in S^*, and shift the rest of the jobs forward in the sequence.

We show that the objective value of $\hat{S}$ is at most the objective value of S^*. This will give a contradiction since $\hat{S}$ is then an optimal sequence with a longer common suffix with S than S^*, and we assumed that S^* has the longest common suffix with S^*.

Notice that the completion times of all jobs either stay the same or decrease from S^* to $\hat{S}$, except for job j^*. But the completion time of j^* in $\hat{S}$ is now less than the completion time of j^{**} in S^*. Using the fact that the functions h_j are all increasing, this completes the proof. $\square$

3 Complexity Theory

3.1 Polynomial Time Reduction

In practice, we can only solve problems that have polynomial time algorithms, since they can scale to large problems when the corresponding constants are small. We have polynomial time algorithms for shortest path, primality testing, and linear programming; in contrast, it is unlikely that there are polynomial time algorithms for longest path, factoring, and integer programming. We would like to classify problems into two categories: those that can be solved in polynomial time, and those that cannot be. But the bad news is that a huge number of fundamental problems have defied classification for decades.

We introduce the notion of **polynomial time reduction**.

Definition 3.1

We say that problem X **reduces** to problem Y in polynomial time if arbitrary instances of problem X can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to an oracle that solves problem Y. We write $X \leq_P Y$ in this scenario.

This definition allows us to do a few things.

(1) **Design algorithms.** If $X \leq_P Y$ and Y can be solved in polynomial time, then X can also be solved in polynomial time.

(2) **Establish intractability.** If $X \leq_P Y$ and X cannot be solved in polynomial time, then Y cannot be solved in polynomial time.

(3) **Establish equivalence.** If $X \leq_P Y$ and $Y \leq_P X$, then we write $X \equiv_P Y$. In this case, X can be solved in polynomial time if and only if Y can be.

The bottom line is that reductions allow us to classify problems according to relative difficulty.

We give two examples of polynomial time reductions here, and refer to Chapter 8 of Kleinberg and Tardos for many other examples.

Recall that a **literal** is a boolean variable or its negation, and a **clause** is a disjunction of literals. A propositional formula Φ is in **conjunctive normal form (CNF)** if it is a conjunction of clauses. For example, $\Phi = (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_3)$ is in CNF.

The Sat problem is as follows: given a propositional formula Φ in CNF, does it have a satisfying truth assignment? Then the 3-Sat problem is Sat where each clause contains exactly 3 literals, and each literal corresponds to a different variable. This has a key application in electronic design automation. One example of an instance of 3-Sat is

$$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4),$$

which has a satisfying truth assignment of $x_1 = \mathsf{T}$, $x_2 = \mathsf{T}$, $x_3 = \mathsf{F}$, and $x_4 = \mathsf{F}$.

The Independent-Set problem is as follows: given a graph $G = (V, E)$ and an integer k, is there a subset of k (or more) vertices such that no two are adjacent? It turns out that 3-Sat can be reduced to Independent-Set.

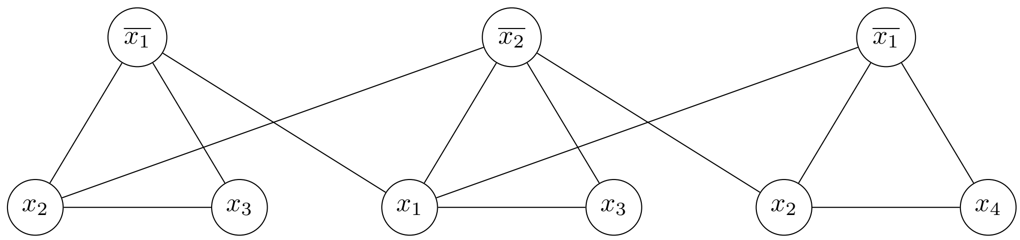
Theorem 3.2

We have 3-Sat $\leq_P$ Independent-Set.

Proof. Let Φ be an instance of 3-Sat. We will construct an instance (G, k) of Independent-Set that has an independent set of size k if and only if Φ is satisfiable.

Let G be a graph which contains 3 nodes for each clause, one for each literal. Connect the 3 literals in a clause in a triangle, and connect every literal to its negations.

For example, with $k = 3$ and $\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$ as above, we can see that G will be the following graph.



We claim that Φ is satisfiable if and only if G contains an independent set of size $k = |\Phi|$.

For the forward direction, consider any satisfying assignment for Φ. Then selecting one true literal from each clause (or triangle) will give an independent set of size k.

Conversely, let S be an independent set of size k. Then S must contain exactly one node in each triangle by construction. Set these literals to T (and the remaining literals consistently). Then all clauses in Φ are satisfied. This completes the proof. $\square$

The Vertex-Cover problem is the following: given a graph $G = (V, E)$ and an integer k, is there a subset of $\leq k$ vertices such that each edge is incident to at least one vertex in the subset?

The Set-Cover problem is the following: given a set U of elements, a collection S of subsets of U, and an integer k, are there $\leq k$ of these subsets whose union is equal to U?

Theorem 3.3

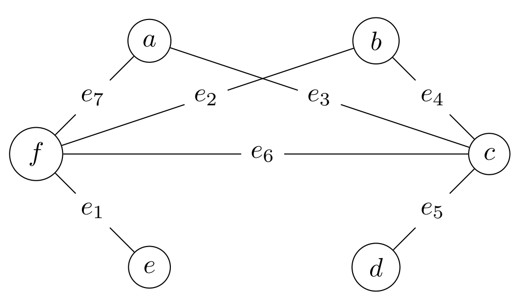
We have Vertex-Cover $\leq_P$ Set-Cover.

Proof. Given a Vertex-Cover instance with the graph $G = (V, E)$ and integer k, we construct a Set-Cover instance (U, S, k) that has a set cover of size k if and only if G has a vertex cover of size k.

We do this by setting the universe to be $U = E$, and include a subset for each node $v \in V$ by

$$S_v = \{e \in E : e \text{ incident to } v\}.$$

For example, consider the following graph G, which has a vertex cover of size 2 given by $\{c, f\}$.



Then we have universe $U = \{1, 2, 3, 4, 5, 6, 7\}$ with subsets $S_a = \{3, 7\}$, $S_b = \{2, 4\}$, $S_c = \{3, 4, 5, 6\}$, $S_d = \{5\}$, $S_e = \{1\}$, and $S_f = \{1, 2, 6, 7\}$. We can see that $U = S_c \cup S_f$, so there is a set cover of size 2.

Let us now show that $G = (V, E)$ contains a vertex cover of size k if and only if U has a set cover of size k.

For the forward direction, let $X \subseteq V$ be a vertex cover of size k in G. Then $Y = \{S_v : v \in X\}$ is a set cover of size k. Conversely, if $Y \subseteq S$ is a set cover of size k for U, then $X = \{v : S_v \in Y\}$ is a vertex cover of size k in G. □

3.2 Computational Intractability

There are three main types of problems.

- **Decision problems.** Does there *exist* a vertex cover of size $\leq k$?
- **Search problems.** *Find* a vertex cover of size $\leq k$.
- **Optimization problems.** *Find* a vertex cover of *minimum* size.

In scheduling, we are mostly concerned with optimization problems. Notice that decision problems are in some sense easier than search problems, which are then easier than optimization problems. In particular, if the decision problem is intractible, then both the search and optimization problems are intractible. Because of this, we will define **P**, **NP**, and **EXP** using decision problems. A decision problem is essentially a yes or no question; we want to answer "yes" if the solution exists and "no" if it doesn't.

Definition 3.4

We denote by **P** the set of decision problems for which there exists a polynomial time algorithm to solve it.

As we mentioned earlier, there are polynomial time algorithms for shortest path, primality testing, and linear programming, so these problems are all in **P**.

Definition 3.5

- An algorithm $C(s, t)$ is a **certifier** for the problem X if for every string s, we have $s \in X$ if and only if there exists a string t such that $C(s, t)$ returns "yes". We call the string t the **certificate** for the input s.
- We denote by **NP** the set of decision problems for which there exists a polynomial time certifier. The certifier $C(s, t)$ is a polynomial time algorithm, and the certificate t for the input s is of polynomial size.

Example 3.6

For the Sat and 3-Sat problems, the input is a propositional formula Φ in CNF. A certificate for the input Φ is an assignment of truth values to the boolean variables, and a certifier checks that each clause in Φ has at least one true literal. Thus, Sat $\in$ **NP** and 3-Sat $\in$ **NP**.

Finally, we can define **EXP**.

Definition 3.7

We denote by **EXP** the set of decision problems for which there exists an exponential time algorithm to solve it.

Now, we show that $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$.

Proposition 3.8

We have $\mathbf{P} \subseteq \mathbf{NP}$.

Proof. Consider a problem X in $\mathbf{P}$. By definition, there exists a polynomial time algorithm $A(s)$ which solves X given any input s. Then take the certificate $t = \varepsilon$ to be the empty string, and set the certifier to be $C(s,t) = A(s)$, which of course runs in polynomial time. □

Proposition 3.9

We have $\mathbf{NP} \subseteq \mathbf{EXP}$.

Proof. Let X be a problem in $\mathbf{NP}$. By definition, there exists a polynomial time certifier $C(s,t)$ for X, whose certificate t satisfies $|t| \leq p(|s|)$ for some polynomial p and any input string s. To solve instance s, we run $C(s,t)$ on all strings t with $|t| \leq p(|s|)$. Return "yes" if and only if $C(s,t)$ returns "yes" for any of these potential certificates. □

It is a known fact that $\mathbf{P} \neq \mathbf{EXP}$, so we either have $\mathbf{P} \neq \mathbf{NP}$ or $\mathbf{NP} \neq \mathbf{EXP}$, or both.

We now move on to $\mathbf{NP}$-completeness. We can think of these as the hardest problems in $\mathbf{NP}$.

Definition 3.10

A problem $Y \in \mathbf{NP}$ is called **NP-complete** if it has the property that for every $X \in \mathbf{NP}$, we have $X \leq_P Y$.

The following proposition says that if we find even one problem $\mathbf{NP}$-complete problem Y that is also in $\mathbf{P}$, then $\mathbf{P} = \mathbf{NP}$. Note that $\mathbf{P} = \mathbf{NP}$ is a famous conjecture, so we have of course not found one yet. It is commonly agreed upon that $\mathbf{P} \neq \mathbf{NP}$, but it is still an open problem.

Proposition 3.11

Suppose that Y is $\mathbf{NP}$-complete. Then $Y \in \mathbf{P}$ if and only if $\mathbf{P} = \mathbf{NP}$.

Proof. For the backwards direction, notice that if $\mathbf{P} = \mathbf{NP}$, then $Y \in \mathbf{P}$ since $Y \in \mathbf{NP}$. On the other hand, suppose $Y \in \mathbf{P}$. Consider any problem $X \in \mathbf{NP}$. Since $X \leq_P Y$, we have $X \in \mathbf{P}$. This implies that $\mathbf{NP} \subseteq \mathbf{P}$, and so $\mathbf{P} = \mathbf{NP}$. □

The following proposition gives us a recipe for proving that a problem Y is $\mathbf{NP}$-complete.

1. Show that $Y \in \mathbf{NP}$.
2. Choose an $\mathbf{NP}$-complete problem X and prove that $X \leq_P Y$.

Proposition 3.12

If X is $\mathbf{NP}$-complete, $Y \in \mathbf{NP}$, and $X \leq_P Y$, then Y is also $\mathbf{NP}$-complete.

Proof. Consider any problem $W \in \mathbf{NP}$. Then $W \leq_P X$ by the definition of $\mathbf{NP}$-completeness and $X \leq_P Y$ by assumption, so by transitivity, we obtain $W \leq_P Y$. Since $W \in \mathbf{NP}$ is arbitrary, we have that Y is $\mathbf{NP}$-complete. □

We now give some examples of **NP**-complete problems. It is useful to know them as many scheduling problems are **NP**-complete, and we can verify that they are indeed **NP**-complete via reductions.

Theorem 3.13: Cook 1971, Levin 1973

The problem Sat is **NP**-complete.

Example 3.14

The following two problems are **NP**-complete.

- Partition: Given n positive integers $s_1, \dots, s_n$ and $b = \frac{1}{2}\sum_{j=1}^{n} s_j$, does there exist a subset $J \subseteq I = \{1, \dots, n\}$ such that
$$b = \sum_{j \in J} s_j = \sum_{j \in I \setminus J} s_j?$$
- 3-Partition: Given $3t$ positive integers $s_1, \dots, s_{3t}$, and an integer b satisfying $\frac{b}{4} < s_j < \frac{b}{2}$ for all $j \in \{1, \dots, 3t\}$ and $\sum_{j=1}^{3t} s_j = t \cdot b$, do there exist t pairwise disjoint three-element subsets $S_j \subseteq \{1, \dots, 3t\}$ such that
$$b = \sum_{j \in J_i} s_j$$
for all $j \in \{1, \dots, t\}$?

Next, we define what it means to be **NP**-hard.

Definition 3.15

An **NP**-hard problem is one such that every problem in **NP** reduces to it in polynomial time.

Intuitively, **NP**-hard problems are "at least as hard as the hardest problems in **NP**". These problems are not necessarily in **NP**, and they do not have to be decision problems. For instance, given an **NP**-complete decision problem, the optimization problem corresponding to it is **NP**-hard. The **NP**-hard problems which are in **NP** are in fact **NP**-complete.

There are two categories of **NP**-hard problems.

- **NP-hard in the ordinary sense.** We can reduce a known **NP**-hard problem to this problem using a polynomial time algorithm, and we can find an optimal solution with an algorithm of pseudo-polynomial time complexity.
- **NP-hard in the strong sense.** We can reduce a known **NP**-hard problem to this problem using a polynomial time algorithm, even if the size of the largest parameter is polynomial in the input size of the problem.

In fact, it turns out that 3-Partition (as defined in Example 3.14) is **NP**-hard in the strong sense, while Partition is only **NP**-hard in the ordinary sense.

For our purposes, we can show that a scheduling problem is **NP**-hard in the ordinary sense if Partition (or a similar problem) can be reduced to this problem with a polynomial time algorithm, and there is an algorithm with pseudo-polynomial time complexity which solves the scheduling problem.

On the other hand, we can show that a scheduling problem is **NP**-hard in the strong sense if 3-Partition (or a similar problem) can be reduced to this problem with a polynomial time algorithm.

3.3 NP-completeness of the Knapsack Problem

In the KNAPSACK problem, we are given n objects and a knapsack. Each item i has value $v_i > 0$ and weight $w_i > 0$. The knapsack has weight limit W. The goal is to fill the knapsack as to maximize the total value.

EXAMPLE 3.16

Consider the following instance of the KNAPSACK problem.

i	1	2	3	4	5
v_i	1	6	18	22	28
w_i	1	2	5	6	7

Then the set $\{3, 4\}$ has total weight 11 and value 40.

First, we formulate the KNAPSACK problem as a decision problem and give a proof that it is **NP**-complete. Given a set X, weights $w_i \geq 0$, values $v_i \geq 0$, a weight limit W, and a target value V, is there a subset $S \subseteq X$ such that $\sum_{i\in S} w_i \leq W$ and $\sum_{i\in S} v_i \geq V$? We can see that KNAPSACK is in **NP** because given a subset $S \subseteq X$, it is easy to add up the weights and values in order to verify that they satisfy the desired bounds.

The SUBSET-SUM problem is the following: given n natural numbers $w_1, \ldots, w_n$ and an integer W, is there a subset that adds up to exactly W?

We know that SAT is **NP**-complete, and we will show that

$$\text{SAT} \leq_P \text{3-SAT} \leq_P \text{SUBSET-SUM} \leq_P \text{KNAPSACK}.$$

It is not hard to see that these intermediate problems are in **NP**, so they will also be **NP**-complete.

PROPOSITION 3.17

We have SAT $\leq_P$ 3-SAT.

PROOF. Consider an instance Φ of SAT. We construct an instance Φ' of 3-SAT by dealing with each clause C in Φ as follows:

(1) If C already has 3 literals, leave it alone.

(2) If C has fewer than 3 literals, just duplicate one of the literals. For example, if $C = (x_1 \vee \overline{x_2})$, then replace the clause by $(x_1 \vee x_1 \vee \overline{x_2})$.

(3) Suppose that $C = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_n)$ where $n > 3$. Create new variables λ_i and replace clause C with the clauses

$$(\ell_1 \vee \ell_2 \vee \lambda_1) \wedge (\overline{\lambda_1} \vee \ell_3 \vee \lambda_2) \wedge (\overline{\lambda_2} \vee \ell_4 \vee \lambda_3) \wedge \cdots \wedge (\overline{\lambda_{n-4}} \vee \ell_{n-2} \vee \lambda_{n-3}) \wedge (\overline{\lambda_{n-3}} \vee \ell_{n-1} \vee \ell_n).$$

We show that this replacement does not affect whether the formula is satisfiable. Suppose that Φ were satisfiable. Let A be an assignment of truth values that makes Φ true. Then for every clause C in Φ, at least one literal is true. Let ℓ_i be a true literal in C. Set $\lambda_j = \mathsf{T}$ for $j = 1, \ldots, i-2$ and $\lambda_j = \mathsf{F}$ for $j = i-1, \ldots, n-3$. Then it can be checked that all the clauses in (3) have a true literal, so Φ' is satisfiable.

Conversely, suppose that Φ' is satisfiable. Take an assignment of truth values so that Φ' is true. We claim that at least one of the ℓ_i is true. Assume towards a contradiction that they are all false. The first clause $(\ell_1 \vee \ell_2 \vee \lambda_1)$ means that λ_1 must be true. Following this train of thought means that λ_i is true for all $i = 1, \ldots, n-3$. But then the last clause $(\overline{\lambda_{n-3}} \vee \ell_{n-1} \vee \ell_n)$ is false, which is a contradiction. $\square$

PROPOSITION 3.18

We have 3-SAT $\leq_P$ SUBSET-SUM.

PROOF. This reduction is fairly tricky. Take an instance Φ of 3-SAT, and suppose that it has n variables $x_1, \ldots, x_n$ and m clauses $C_1, \ldots, C_m$. For each variable x_i, we construct numbers t_i and f_i of $n+m$ digits.

(1) The i-th digit of t_i and f_i is equal to 1.

(2) For $n+1 \leq j \leq n+m$, the j-th digit of t_i is equal to 1 if x_i is in clause C_{j-n}.

(3) For $n+1 \leq j \leq n+m$, the j-th digit of f_i is equal to 1 if $\overline{x_i}$ is in clause C_{j-n}.

(4) All other digits of t_i and f_i are 0.

For example, if $\Phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee x_3)$, then t_i and f_i are as follows.

	$i=1$	$i=2$	$i=3$	$j=1$	$j=2$	$j=3$	$j=4$
t_1	1	0	0	1	0	0	1
f_1	1	0	0	0	1	1	0
t_2	0	1	0	1	0	1	0
f_2	0	1	0	0	1	0	1
t_3	0	0	1	1	1	0	1
f_3	0	0	1	0	0	1	0

Next, for each clause C_j, we construct numbers x_j and y_j also of $n+m$ digits.

(1) The $(n+j)$-th digit of x_j and y_j is equal to 1.

(2) All other digits of x_j and y_j are 0.

For the same formula Φ as above, x_j and y_j are as follows.

	$i=1$	$i=2$	$i=3$	$j=1$	$j=2$	$j=3$	$j=4$
x_1	0	0	0	1	0	0	0
y_1	0	0	0	1	0	0	0
x_2	0	0	0	0	1	0	0
y_2	0	0	0	0	1	0	0
x_3	0	0	0	0	0	1	0
y_3	0	0	0	0	0	1	0
x_4	0	0	0	0	0	0	1
y_4	0	0	0	0	0	0	1

Finally, we construct a sum number W of $n+m$ digits by letting the j-th digit of W be 1 for $j = 1, \ldots, n$, and 3 for $j = n+1, \ldots, n+m$. This gives us our SUBSET-SUM instance with natural numbers t_i, f_i, x_j, y_j and desired sum W.

We show that Φ is satisfiable if and only if a solution to the SUBSET-SUM instance exists. For the forward direction, consider an assignment of truth values that makes Φ true. Take t_i if x_i is true and f_i if x_i is false. Take x_j if the number of true literals in C_j is at most 2, and take y_j if the number of true literals in C_j is 1. Then the sum of these numbers will yield W.

For example, consider our above formula $\Phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee x_3)$ which is satisfied by taking $x_1 = x_2 = x_3 = \mathsf{T}$.

	$i=1$	$i=2$	$i=3$	$j=1$	$j=2$	$j=3$	$j=4$
t_1	1	0	0	1	0	0	1
t_2	0	1	0	1	0	1	0
t_3	0	0	1	1	1	0	1
x_2	0	0	0	0	1	0	0
y_2	0	0	0	0	1	0	0
x_3	0	0	0	0	0	1	0
y_3	0	0	0	0	0	1	0
x_4	0	0	0	0	0	0	1
W	1	1	1	3	3	3	3

Conversely, suppose that a solution to the Subset-Sum problem exists. Set $x_i = \mathsf{T}$ if t_i is in the subset, and $x_i = \mathsf{F}$ if f_i is in the subset. Notice that exactly one number per variable must be in the subset, for otherwise one of the first n digits of the sum is greater than 1. This means that this assignment is consistent. Moreover, at least one variable number corresponding to a literal in a clause must be in the subset. Otherwise, one of the next m digits would be smaller than 3. In particular, each clause in Φ is satisfied. ☐

Proposition 3.19

We have Subset-Sum $\leq_P$ Knapsack.

Proof. Given an instance $(\{u_1, \ldots, u_n\}, U)$ of Subset-Sum, create a Knapsack instance by taking $v_i = w_i = u_i$ and $V = W = U$. The conditions in the Knapsack problem are then $\sum_{i \in S} u_i \leq U$ and $\sum_{i \in S} u_i \geq U$, and thus $\sum_{i \in S} u_i = U$. So there is a solution to the Subset-Sum instance if and only if there is a solution to the Knapsack instance. ☐

3.4 Dynamic Programming and FPTAS for the Knapsack Problem

We will first discuss two dynamic programming approaches to the Knapsack problem.

- **Approach 1.** Let $\mathrm{OPT}(i, w)$ be the maximum value subset of items $1, \ldots, i$ with weight limit w.

  If OPT does not select item i, then OPT selects the best of $1, \ldots, i-1$ up to the weight limit w. On the other hand, if OPT does select item i, then there is a new weight limit $w - w_i$, and OPT selects the best of $1, \ldots, i-1$ up to the weight limit $w - w_i$. Thus, we can recursively compute

$$\mathrm{OPT}(i, w) = \begin{cases} 0, & \text{if } i = 0, \\ \mathrm{OPT}(i-1, w), & \text{if } w_i > w, \\ \max\{\mathrm{OPT}(i-1, w), v_i + \mathrm{OPT}(i-1, w - w_i)\}, & \text{otherwise.} \end{cases}$$

  We see that this approach computes the optimal value in $O(nW)$ time, which is not polynomial in the input size; however, it is polynomial in the input size if the weights are small integers.

- **Approach 2.** Let $\mathrm{OPT}(i, v)$ be the minimum weight of a knapsack for which we can obtain a solution of value $\geq v$ using a subset of items $1, \ldots, i$. Notice that the optimal value is the largest value v such that $\mathrm{OPT}(n, v) \leq W$.

  If OPT does not select item i, then OPT selects the best of $1, \ldots, i-1$ that achieves value $\geq v$. On the other hand, if OPT does select item i, then it consumes weight w_i, and the remaining items need to achieve value $\geq v - v_i$. In particular, OPT selects the best of $1, \ldots, i-1$ that achieves value $\geq v - v_i$.

Therefore, we can recursively compute

$$\mathrm{OPT}(i,v) = \begin{cases} 0, & \text{if } v \le 0, \\ \infty, & \text{if } i = 0 \text{ and } v > 0, \\ \min\{\mathrm{OPT}(i-1,v), w_i + \mathrm{OPT}(i-1, v - v_i)\}, & \text{otherwise.} \end{cases}$$

We see that this approach computes the optimal value in $O(n^2 v_{\max})$ time, where $v_{\max}$ is the maximum of the item values. Indeed, it is easy to see that the optimal value V^* is bounded above by $nv_{\max}$. There is one subproblem for each item and for each value $v \le V^*$, and it takes $O(1)$ time for each subproblem. Again, this is not polynomial in the input size, but it is polynomial time if the values are small integers.

In fact, both these dynamic programming approaches are pseudo-polynomial time algorithms for Knapsack, so the Knapsack problem is **NP**-complete in the ordinary sense.

We now work towards an approximation scheme for solving Knapsack instances.

Definition 3.20

- Let Π be an optimization problem, and let I be an instance of Π with optimal value OPT. An **approximation scheme** for Π is an algorithm such that given I together with an error parameter $\varepsilon > 0$, returns a (feasible) solution whose objective value is $\ge (1-\varepsilon)\mathrm{OPT}$ if Π is a maximization problem, or $\le (1+\varepsilon)\mathrm{OPT}$ if Π is a minimization problem.
- An approximation scheme is called a **polynomial time approximation scheme (PTAS)** if for every fixed $\varepsilon > 0$, the running time of the algorithm is polynomial in the size of the instance $|I|$. Note that the running time of a PTAS may depend on $1/\varepsilon$ and may grow rapidly respect to it, but for a PTAS, only the running time with respect to $|I|$ is relevant.
- A **fully polynomial time approximation scheme (FPTAS)** is a PTAS that is also polynomial in $1/\varepsilon$. In other words, it is polynomial in both $1/\varepsilon$ and the size of the instance $|I|$.

We can give an FPTAS for the Knapsack problem by making use of our second dynamic programming approach above. Recall that it runs in time $O(n^2 v_{\max})$, so if $v_{\max}$ is polynomial in n, then the dynamic programming approach is also polynomial in n. Therefore, a rough sketch of the algorithm would be as follows:

(1) Round all the item values to lie in a smaller range.

(2) Run the second dynamic programming approach on the rounded instance.

(3) Return the optimal items from the rounded instance.

Now, consider step (1) where we round the values. Let $v_{\max}$ be the largest value in the original instance. Given a precision factor $0 < \varepsilon \le 1$, let $\theta = \varepsilon v_{\max}/(2n)$ be the scaling factor we apply to the values. Define the scaled values by

$$\hat{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil,$$

and the values obtained by rescaling to the original problem by

$$\bar{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil \theta.$$

An observation we can make is that optimal solutions to the problem using $\bar{v}$ are equivalent to optimal solutions to the problem using $\hat{v}$. Therefore, an optimal solution using $\hat{v}$ is nearly optimal for the original instance. Since $\hat{v}$ is small, using our dynamic programming algorithm is fast.

Theorem 3.21

If S is a solution found by the rounding algorithm and S^* is any other feasible solution, then

$$(1+\varepsilon)\sum_{i\in S} v_i \geq \sum_{i\in S^*} v_i.$$

Proof. Let S^* be any feasible solution satisfying the weight constraints. Then we have

$$\begin{aligned}
\sum_{i\in S^*} v_i &\leq \sum_{i\in S^*} \bar{v}_i && \text{(always rounding up)}\\
&\leq \sum_{i\in S} \bar{v}_i && \text{(solve rounded up instance optimally)}\\
&\leq \sum_{i\in S} (v_i+\theta) && \text{(rounding by at most } \theta\text{)}\\
&\leq \sum_{i\in S} v_i + n\theta && \text{(since } |S|\leq n\text{)}\\
&= \sum_{i\in S} v_i + \frac{1}{2}\varepsilon v_{\max}. && \text{(since } \theta=\varepsilon v_{\max}/(2n)\text{)}
\end{aligned}$$

Notice that this argument applies to the subset S^* containing only the item of largest value, which means

$$v_{\max} \leq \sum_{i\in S} v_i + \frac{1}{2}\varepsilon v_{\max} \leq \sum_{i\in S} v_i + \frac{1}{2} v_{\max},$$

where the last inequality is because $0<\varepsilon\leq 1$. Rearranging the above gives

$$v_{\max} \leq 2\sum_{i\in S} v_i,$$

and we can complete the above chain of inequalities to get

$$\sum_{i\in S^*} v_i \leq \sum_{i\in S} v_i + \frac{1}{2}\varepsilon v_{\max} \leq (1+\varepsilon)\sum_{i\in S} v_i.$$

□

Thus, we can conclude that our above algorithm is an FPTAS for the Knapsack problem.

4 More Single Machine Models

4.1 Maximum Lateness with Release Dates

We have already seen in Section 2.3 that there is a polynomial time algorithm to solve $(1 \parallel L_{\max})$ instances. This was the Earliest Due Date (EDD) rule, which placed the jobs in increasing order of the due dates. We also saw that this was a special case of $(1 \mid \text{prec} \mid h_{\max})$, for which there was also an efficient algorithm, namely Lowest Cost Last (LCL).

But what if we introduce release dates to the $(1 \parallel L_{\max})$ problem? It turns out that this generalization, without preemption, is significantly harder than the problem where all jobs are available at time 0. Moreover, the optimal schedule is not necessarily a non-delay schedule. It can be advantageous in this case to keep the machine idle before the release of a new job.

Theorem 4.1

The problem $(1 \mid r_j \mid L_{\max})$ is strongly **NP**-hard.

Proof. This proof is based on the fact that 3-Partition (as described in Example 3.14) reduces to $(1 \mid r_j \mid L_{\max})$. Suppose that we are given integers $a_1, \ldots, a_{3t}, b$ such that $\frac{b}{4} < a_j < \frac{b}{2}$ and $\sum_{j=1}^{3t} a_j = t \cdot b$. We construct an instance of $(1 \mid r_j \mid L_{\max})$ with $n = 4t - 1$ as follows:

- For $j = 1, \ldots, t-1$, we set $r_j = jb + (j-1)$, $p_j = 1$, $d_j = jb + j$.
- For $j = t, \ldots, 4t-1$, we set $r_j = 0$, $p_j = a_{j-t+1}$, and $d_j = tb + (t-1)$.

Notice that a schedule with $L_{\max} \leq 0$ exists if and only if every job $j \in \{1, \ldots, t-1\}$ can be processed between r_j and $d_j = r_j + p_j$. This can be done if and only if the remaining jobs can be partitioned over the t intervals of length b, which can be done if and only if 3-Partition has a solution. □

The $(1 \mid r_j \mid L_{\max})$ problem is important because it often appears as a subproblem in heuristic procedures for flow shop and job shop problems. A branch and bound procedure $(1 \mid r_j \mid L_{\max})$ can be constructed as follows. The branching process may be based on the fact that schedules are developed starting from the beginning of the schedule. There is a single node at level 0 which is the top of the tree. At this node, no job has been put into any position of the sequence yet. There are n branches going down to n nodes at level 1. Each node at this level has a specific job put into the first position of the schedule. Then at each node, there are $n-1$ jobs remaining whose position in the schedule has yet to be determined. Hence, there are $n-1$ arcs emanating from each node at level 1 to level 2, and there are $(n-1) \times (n-2)$ nodes at level 2. We could continue in this way to enumerate all possible schedules.

However, it is not necessary to consider every remaining job as a possible candidate for the next position. If the jobs $j_1, \ldots, j_{k-1}$ are scheduled as the first $k-1$ jobs at a node at level $k-1$, then we only need to consider job j_k if

$$r_{j_k} < \min_{\ell \in J} \left(\max(t, r_\ell) + p_\ell \right),$$

where J denotes the set of jobs not yet scheduled and t denotes the time job j_k is supposed to start. The reason for this condition is because if job j_k does not satisfy this inequality, then selecting the job which minimizes the right-hand side instead of j_k does not increase the value of $L_{\max}$. Therefore, the branching rule is fairly easy.

There are several ways in which bounds for nodes can be obtained. One easy lower bound for a node at level $k-1$ can be established by scheduling the remaining jobs J according to the *preemptive* EDD rule which is known to be optimal for $(1 \mid r_j, \text{prmp} \mid L_{\max})$, and thus provides a lower bound for the problem at hand. If a preemptive EDD rule results in a non-preemptive schedule, then all nodes with a higher lower bound may be disregarded.

Example 4.2

Consider the following instance of $(1 \mid r_j \mid L_{\max})$ with $n = 4$ jobs.

j	1	2	3	4
p_j	4	2	6	5
r_j	0	1	3	5
d_j	8	12	11	10

At level 1 of the search tree, there are four nodes, namely $(1, *, *, *)$, $(2, *, *, *)$, $(3, *, *, *)$, and $(4, *, *, *)$. Notice that we may discard the nodes $(3, *, *, *)$ and $(4, *, *, *)$ immediately. Indeed, we have $\min_{\ell \in \{1,2,3,4\}}(\max(t, r_\ell) + p_\ell) = 3$ with $\ell = 2$, and we see that $r_3 \geq 3$ and $r_4 \geq 3$.

Computing a lower bound for node $(1, *, *, *)$ according to the preemptive EDD rule results in a schedule where job 3 is processed during the interval $[4, 5]$, job 4 is processed during $[5, 10]$, job 3 again during $[10, 15]$, and job 2 during $[15, 17]$. Then $L_{\max} = 5$ for this schedule, and so 5 is a lower bound for the node $(1, *, *, *)$. A similar computation shows that a lower bound for the node $(2, *, *, *)$ is 7.

Consider now the node $(1, 2, *, *)$ at level 2. The lower bound for this node is 6 and is determined by the non-preemptive schedule $1, 2, 4, 3$. Next, looking at the node $(1, 3, *, *)$ at level 2, the lower bound is 5 and is determined by the non-preemptive schedule $1, 3, 4, 2$. Since the lower bound for node $(1, *, *, *)$ is 5 and the lower bound for $(2, *, *, *)$ is greater than 5, it follows that the schedule $1, 3, 4, 2$ is optimal.

The problem $(1 \mid r_j, \text{prec} \mid L_{\max})$ can be handled in a similar way. From an enumeration point of view, it is even easier than the problem without precedence constraints since many schedules can be ruled out immediately.

4.2 Number of Tardy Jobs

Recall that $U_j = 0$ if the job is timely and $U_j = 1$ if the job is late. The goal of the problem $(1 \parallel \sum U_j)$ is to minimize the number of tardy jobs. This objective may at first appear somewhat artificial and seems to be of no practical interest. However, in the real world, it is a performance measure that is often monitored. It is equivalent to the percentage of on time shipments.

Notice that it does not matter how late a job is; the only determining factor is if it is late or not. A solution to this problem can be represented as a partition of the jobs into sets S_1 and S_2, where S_1 is the set of jobs meeting their due dates in Earliest Due Date (EDD) order, and S_2 is the set of late jobs in an arbitrary order (since the amount of lateness is irrelevant).

Lemma 4.3

Let OPT denote the optimal value for a given instance of $(1 \parallel \sum U_j)$. If the sequence given by the EDD rule has a late job, then OPT ≥ 1.

Proof. Let k be the first late job in the EDD sequence. Then we have

$$C_k = \sum_{i \in [k]} p_i > d_k = \max_{i \in [k]} d_i.$$

Consider any schedule S. Let ℓ be the last of the jobs in $[k]$ in S. Then the completion time of ℓ in S is at least $\sum_{i \in [k]} p_i > d_\ell$, so ℓ is a late job in S. $\square$

Algorithm 4.4: Moore-Hodgson

1. Enumerate the jobs in EDD order.
2. Set $S_1 \leftarrow \varnothing$ and $t \leftarrow 0$.
3. for $j = 1$ to n do:
   - Set $S_1 \leftarrow S_1 \cup \{j\}$ and $t \leftarrow t + p_j$.
   - if $t > d_j$ then:
     - Find a job k with the largest p_k value in S_1.
     - Set $S_1 \leftarrow S_1 \setminus \{k\}$ and $t \leftarrow t - p_k$.
   - endif

   endfor

The principle of the Moore-Hodgson algorithm is that we schedule the jobs by the EDD rule, and when a job gets late, we rescue the situation by throwing out the job with the highest processing time. All removed jobs are considered late, and the remaining ones are timely. This algorithm runs in $O(n \log n)$ time.

Example 4.5

Consider the following instance of $(1 \parallel \sum U_j)$ with $n = 5$ jobs, where the jobs are already in EDD order.

j	1	2	3	4	5
p_j	7	8	4	6	6
d_j	9	17	18	19	21

We can initially schedule jobs 1 and 2, which will both be timely. However, once we schedule job 3, we see that it will be late, since $t = 19 > 18 = d_3$.

Job 1	Job 2	Job 3
1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19

Therefore, we toss out job 2 which has the highest processing time of $p_2 = 8$ and continue. We can schedule job 4 just fine, but job 5 will be late with $t = 23 > 21 = d_5$.

Job 1	Job 3	Job 4	Job 5
1 2 3 4 5 6 7	8 9 10 11	12 13 14 15 16 17	18 19 20 21 22 23

This time, we toss out job 1 since it has the highest processing time of $p_1 = 7$. Then we obtain $S_1 = \{3, 4, 5\}$ and $S_2 = \{1, 2\}$, so OPT $= 2$ for this instance.

The following lemma is the key to proving that the Moore-Hodgson algorithm is optimal for $(1 \parallel \sum U_j)$. We will assume that the jobs are already scheduled in EDD.

Lemma 4.6

Suppose there is at least one late job in the EDD sequence $1, \ldots, n$. Let k be the first late job in the EDD sequence, and let m be the first job rejected by the Moore-Hodgson algorithm. Then there is an optimal schedule which rejects m.

PROOF. Consider any optimal schedule π. Let $R_\pi \subseteq [n]$ denote the subset of rejected (late) jobs, and let $A_\pi = [n] \setminus R_\pi$ denote the set of timely jobs. By Lemma 4.3, we may assume that π schedules the jobs of A_π in EDD order, followed by the jobs of R_π in arbitrary order.

If $m \in R_\pi$, we are done. So suppose that $m \notin R_\pi$. By Lemma 4.3, there is a job $r \in [k]$ other than m that has been rejected. Consider the schedule σ that sequences the jobs in $A_\sigma = (A_\pi \setminus \{m\}) \cup \{r\}$ in EDD order first, followed by the jobs in $[n] \setminus A_\sigma = (R_\pi \setminus \{r\}) \cup \{m\}$ in arbitrary order. We will show that σ schedules all jobs in A_σ on time. This will mean that σ is an optimal schedule since $|R_\sigma| = |R_\pi|$, and by construction, σ rejects m.

First, the jobs in $A_\sigma \cap [k-1]$ are completed on time since the EDD rule completes all jobs in $[k-1]$ on time. Next, if $k \in A_\sigma$, then its completion time is at most

$$\sum_{i \in [k] \setminus \{m\}} p_i \leq \sum_{i \in [k-1]} p_i \leq d_{k-1} \leq d_k,$$

where the first inequality follows from the choice of m by the Moore-Hodgson algorithm which ensures that $p_m = \max_{i \in [k]} p_i \geq p_k$. Finally, compared to the schedule π, the completion times of the jobs in $A_\sigma \setminus [k] = A_\pi \setminus [k]$ have been changed in the new schedule σ by $p_r - p_m \leq 0$, so these jobs are also on time. ☐

THEOREM 4.7

The Moore-Hodgson algorithm gives an optimal schedule for $(1 \parallel \sum U_j)$.

PROOF. We proceed by induction on OPT over all instances of the problem.

If an instance has OPT $= 0$, then by Lemma 4.3, the Moore-Hodgson algorithm outputs the EDD sequence with no late jobs.

Suppose now that I is an instance with $\text{OPT}(I) \geq 1$ late jobs. Let I' be obtained from I by deleting the first job m which is rejected by the Moore-Hodgson algorithm. By Lemma 4.6, we have $\text{OPT}(I') = \text{OPT}(I) - 1$. By the induction hypothesis, the algorithm finds an optimal schedule S' for I'. The algorithm for I outputs the schedule S such that S is the same as S' except that job m is added at the end as a rejected job. Clearly, S has at most $\text{OPT}(I') + 1 = \text{OPT}(I)$ rejected jobs and hence is optimal. ☐

4.3 Total Tardiness

In practice, minimizing the number of tardy jobs $\sum U_j$ cannot be the only objective to measure how due dates are being met. Only minimizing the number of late jobs will force some jobs to have to wait for an unacceptably long time to complete. If we instead minimize the total tardiness, it is less likely that the wait for any given job will be unacceptably long.

The problem $(1 \parallel \sum T_j)$ has received an enormous amount of attention in literature. For many years, its computational complexity remained open; its **NP**-hardness was only established recently in 1990. Since $(1 \parallel \sum T_j)$ is **NP**-hard in the ordinary sense, it allows for a pseudo-polynomial time algorithm based on dynamic programming. The algorithm is based on two preliminary results.

LEMMA 4.8

If $p_j \leq p_k$ and $d_j \leq d_k$, then there exists an optimal schedule in which job j is scheduled before job k.

PROOF. We leave the proof as an exercise. This is a standard interchange argument. ☐

This type of result is useful when an algorithm has to be developed for a problem that is **NP**-hard. Such a result, often referred to as a **dominance result** or **elimination criterion**, often allows one to disregard a fairly large number of sequences. Such a dominance result may also be thought of as a set of precedence constraints on the jobs. The more precedence constraints created through such dominance results, the easier the problem becomes.

In the following lemma, the sensitivity of an optimal sequence to the due dates is considered. We consider two problem instances, both of which have n jobs with processing times $p_1, \ldots, p_n$. The first instance has due dates $d_1, \ldots, d_n$. Let k be a fixed job, and let $\hat{C}_k$ be the latest completion time of job k among all optimal schedules. Then we set the due dates for the second instance to be

$$\hat{d}_j = \begin{cases} d_j, & \text{if } j \neq k, \\ \max(d_k, \hat{C}_k), & \text{if } j = k. \end{cases}$$

In particular, we are only changing one piece of data, namely the due date of job k.

Lemma 4.9

Any optimal sequence with respect to the second instance with due dates $\hat{d}_1, \ldots, \hat{d}_n$ is also optimal for the first instance with due dates $d_1, \ldots, d_n$.

Proof. We first introduce some notation. Let $\hat{S}$ be an optimal sequence such that job k has largest completion time $\hat{C}_k$ among all optimal sequences. For an arbitrary sequence S, we let $V(S)$ be the objective value according to the first instance where job k has due date d_k, and $V'(S)$ be the objective value according to the second instance where job k has due date $\hat{d}_k$.

Let S' be any optimal sequence for the second instance. Then we have $V'(\hat{S}) \geq V'(S')$ since S' is optimal for the second instance. Moreover, we observe that

$$V'(\hat{S}) = V(\hat{S}) - (\hat{C}_k - d_k),$$

or equivalently, $V(\hat{S}) - V'(\hat{S}) = \hat{C}_k - d_k$. Finally, when we switch objectives, the most improvement we can get is $\hat{C}_k - d_k$, so we deduce that

$$\begin{aligned} V(S') &\leq V'(S') + (\hat{C}_k - d_k) \\ &= V'(S') + V(\hat{S}) - V'(\hat{S}) \\ &\leq V(\hat{S}). \end{aligned}$$

This means that S' is also optimal for the first instance, so we are done. □

Lemma 4.9 is quite a technical result, and it requires the knowledge of all optimal sequences for the first instance. For now, we will assume that the jobs are scheduled in EDD order so that $d_1 \leq \cdots \leq d_n$, and job k is such that $p_k = \max(p_1, \ldots, p_n)$. In particular, this is the job with the k-th smallest due date and largest processing time. It follows from Lemma 4.8 that there exists an optimal sequence in which jobs $1, \ldots, k-1$ all appear in some order before job k. Of the remaining $n-k$ jobs, some may appear before job k while some may appear after job k. The following lemma focuses on the placement of these remaining jobs.

Lemma 4.10

There is an integer δ with $0 \leq \delta \leq n-k$ such that there is an optimal sequence S in which job k is preceded by all jobs j with $j \leq k+\delta$ and followed by all jobs j with $j > k+\delta$.

Proof. We only give a brief sketch here. In some optimal schedule, the jobs $1, \ldots, k-1$ are positioned before job k due to the dominance criterion from Lemma 4.8. From Lemma 4.9, we can increase d_k such that some other jobs $k+1, \ldots, k+\delta$ are positioned before job k due to the dominance criterion. □

Lemma 4.10 tells us an optimal sequence looks like the concatenation of three subsets of jobs, namely

(i) jobs $1, \ldots, k-1, k+1, \ldots, k+\delta$ in some order, followed by

(ii) job k, followed by

(iii) jobs $k+\delta+1, \ldots, n$ in some order.

The completion time of job k under this schedule is

$$C_k(\delta) = \sum_{j \le k+\delta} p_j.$$

For the entire sequence to be optimal, it is clear that the first and third subsets must be optimally sequenced within themselves. This is the heart of a dynamic programming approach that determines an optimal sequence for a larger set of jobs after having determined optimal sequences for proper subsets of the larger set.

We define $J(j, \ell, k)$ to be the set of all jobs in the set $\{j, \ldots, \ell\}$ with processing time at most p_k, but $k \notin J(j, \ell, k)$. Note that here, k does not necessarily have to be the job with highest processing time. Then, we define $V(J(j, \ell, k), t)$ to be the total tardiness of the jobs $J(j, \ell, k)$ in an optimal sequence that starts at time t. We now state the dynamic programming algorithm.

Algorithm 4.11: Dynamic Programming for Total Tardiness

The initial conditions are $V(\varnothing, t) = 0$ and $V(\{j\}, t) = \max(0, t + p_j - d_j)$ for all jobs j and $t \ge 0$. We have the recursive relation

$$V(J(j, \ell, k), t) = \min_{\delta}\{V(J(j, k'+\delta, k'), t) + \max(0, C_{k'}(\delta) - d_{k'}) + V(J(k'+\delta+1, \ell, k'), C_{k'}(\delta))\},$$

where job k' is such that $p_{k'} = \max_{j' \in J(j,\ell,k)} p_{j'}$. The optimal value is then given by $V(\{1, \ldots, n\}, 0)$.

Note that there are at most $O(n^3)$ subsets of jobs $J(j, \ell, k)$ and $\sum p_j$ possible values of t. This means that there are $O(n^3 \sum p_j)$ recursive equations. Each recursion takes $O(n)$ time, so the total running time of the dynamic programming approach is $O(n^4 \sum p_j)$, which is pseudo-polynomial.

Example 4.12

We run our dynamic programming algorithm on an example. Consider the following instance of $(1 \parallel \sum T_j)$ with $n = 5$ jobs.

j	1	2	3	4	5
p_j	121	79	147	83	130
d_j	260	266	266	336	337

We see that job 3 has the largest processing time, giving us $0 \le \delta \le 5 - 3 = 2$. Then we have

$$V(\{1,2,3,4,5\}, 0) = \min \begin{cases} V(J(1,3,3),0) + 81 + V(J(4,5,3),347), & \text{for } \delta = 0, \\ V(J(1,4,3),0) + 164 + V(J(5,5,3),430), & \text{for } \delta = 1, \\ V(J(1,5,3),0) + 294 + V(\varnothing,560), & \text{for } \delta = 2. \end{cases}$$

We see that $V(J(1,3,3),0) = 0$ via the sequences $1,2$ and $2,1$. The dominance rule tells us that the sequences $1,2,4$ and $2,1,4$ are optimal for the jobs $J(1,4,3)$ with $V(J(1,4,3),0) = 0$, and similarly, the sequences $1,2,4,5$ and $2,1,4,5$ are optimal for $J(1,5,3)$ with $V(J(1,5,3),0) = 76$.

On the other hand, we have

$$V(J(4,5,3),347) = (347+83-336) + (347+83+130-337) = 317$$

for the sequence 4, 5, and $V(J(5,5,3),430) = 430+130-337 = 223$.

Putting everything together, we obtain

$$V(\{1,2,3,4,5\},0) = \min\{0+81+317, 0+164+223, 76+294+0\} = \min\{398,387,370\} = 370.$$

The sequences that attain this value are 1, 2, 4, 5, 3 and 2, 1, 4, 5, 3.

Since $(1 \parallel \sum T_j)$ is **NP**-hard, we cannot hope to find a polynomial time algorithm to solve arbitrary instances of it. However, we can give an FPTAS to obtain a solution that is close to optimal, using a similar approach as we did for the Knapsack problem in Section 3.4.

First, we will give some lower and upper bounds for total tardiness.

- Recall that the EDD rule produces a schedule with optimal maximum lateness. In particular, this also produces a schedule with optimal maximum tardiness.
- Clearly, at least one job in a schedule has the maximum tardiness. Then the total tardiness of an optimal schedule is at least the optimal maximum tardiness.
- The total tardiness of the EDD schedule is at least the optimal maximum tardiness, but at most n times the optimal maximum tardiness.

Let $\sum T_j(\text{EDD})$ denote the total tardiness under the EDD schedule, and let $T_{\max}(\text{EDD}) = \max(T_1, \dots, T_n)$ be the maximum tardiness under the EDD schedule. For an optimal schedule OPT, we have

$$T_{\max}(\text{EDD}) \le \sum T_j(\text{OPT}) \le \sum T_j(\text{EDD}) \le n \cdot T_{\max}(\text{EDD}).$$

Let $V(J,t)$ be the minimum total tardiness of the job subset J assuming that processing begins at time t. There is a time t^* such that $V(J,t) = 0$ for all $t \le t^*$ and $V(J,t) > 0$ for all $t > t^*$. Then we have $V(J, t^* + \delta) \ge \delta$ for all $\delta \ge 0$. By executing the pseudo-polynomial dynamic programming approach described in Algorithm 4.11, one only has to compute $V(J,t)$ for

$$\max\{0, t^*\} \le t \le t^* + n \cdot T_{\max}(\text{EDD}).$$

Substituting $\sum p_j$ in the overall running time of the dynamic programming algorithm by $n \cdot T_{\max}(\text{EDD})$ yields a new bound of $O(n^5 \cdot T_{\max}(\text{EDD}))$.

Now, rescale the processing times and due dates via $p'_j = \lfloor p_j/K \rfloor$ and $d'_j = d_j/K$ for some factor $K > 0$. Let S be an optimal schedule for the rescaled problem. We let $\sum T^*_j(S)$ denote the total tardiness of the schedule S with respect to the processing times $Kp'_j \le p_j$ and due dates d_j. Let $\sum T_j(S)$ denote the total tardiness of the schedule S with respect to the original processing times p_j and due dates d_j. From the fact that $Kp'_j \le p_j < K(p'_j + 1)$, it follows that

$$\sum T^*_j(S) \le \sum T_j(\text{OPT}) \le \sum T_j(S) < \sum T^*_j(S) + \frac{Kn(n+1)}{2}.$$

This chain of inequalities implies that

$$\sum T_j(S) - \sum T_j(\text{OPT}) < \frac{Kn(n+1)}{2}.$$

If K is chosen such that

$$K = \frac{2\varepsilon}{n(n+1)} \cdot T_{\max}(\text{EDD}),$$

then we obtain

$$\sum T_j(S) - \sum T_j(\mathrm{OPT}) < \varepsilon \cdot T_{\max}(\mathrm{EDD}).$$

Moreover, for this choice of K, the time bound $O(n^5 \cdot T_{\max}(\mathrm{EDD})/K)$ becomes $O(n^7/\varepsilon)$, making our approximation scheme fully polynomial. We summarize the FPTAS as follows.

Algorithm 4.13: Total Tardiness FPTAS

(1) Apply EDD in order to determine $T_{\max}(\mathrm{EDD})$. If $T_{\max}(\mathrm{EDD}) = 0$, then $\sum T_j(\mathrm{OPT}) = 0$ and EDD gives an optimal schedule, so stop. Otherwise, set

$$K = \frac{2\varepsilon}{n(n+1)} \cdot T_{\max}(\mathrm{EDD}).$$

(2) Rescale the processing times and due dates via $p'_j = \lfloor p_j/K \rfloor$ and $d'_j = d_j/K$.

(3) Apply Algorithm 4.11 to the rescaled data.

4.4 Weighted Number of Late Jobs

The generalization of the problem $(1 \parallel \sum U_j)$ with additional weights is known to be **NP**-hard. This can be shown via a reduction from 3-Partition. A popular heuristic for this problem is the WSPT rule, but even that can perform very badly. As usual, we cannot hope to find a polynomial time algorithm which solves $(1 \parallel \sum w_j U_j)$.

Example 4.14

Consider the following instance of $(1 \mid d_j = d \mid \sum w_j U_j)$ with $n = 3$ jobs and equal due dates.

j	1	2	3
p_j	11	9	90
d_j	100	100	100
w_j	12	9	89

Then the WSPT rule fails: it gives the schedule $1, 2, 3$ which has objective value 89. The optimal schedules for this example are $2, 3, 1$ and $3, 2, 1$, which both have objective value 12.

In fact, the Knapsack problem is equivalent to the special case $(1 \mid d_j = d \mid \sum w_j U_j)$ where all due dates are equal. Indeed, we can think of the due date d as the weight of the knapsack, the processing times p_j as the weight of the individual items, and the weights w_j as their values. We then want to minimize the value of the items we are throwing out of the knapsack, which is the same as maximizing the value of the items inside the knapsack.

In this section, our goal is to come up with some dynamic programming approaches to solve $(1 \parallel \sum w_j U_j)$ in pseudo-polynomial time. We then construct an FPTAS using one of the dynamic programming schemes.

First, we will introduce some notation. Consider an instance of $(1 \parallel \sum w_j U_j)$ where all weights $w_j \geq 0$ are integer. We assume that the jobs are indexed by the EDD sequence.

- We call S a **feasible** set of jobs if every job in S can be completed on time starting at time 0.
- We define $w(S) = \sum_{j \in S} w_j$ and $p(S) = \sum_{j \in S} p_j$.

Approach 1. We define $T(i, \hat{w})$ to be the minimum completion time of a feasible subset S of $[i] = \{1, \dots, i\}$ with weight $\geq \hat{w}$. We can think of $\hat{w}$ as a "target" weight. In other words, $T(i, \hat{w})$ is the minimum completion time of a chosen subset of the jobs $\{1, \dots, i\}$ such that all jobs in the chosen subset are timely (that is, there are no late jobs in an EDD ordering of the chosen subset) and moreover, the total weight of the jobs in the chosen subset is $\geq \hat{w}$.

We set $T(i, \hat{w}) = \infty$ if no such feasible set exists, $T(0, 0) = 0$, and $T(0, \hat{w}) = \infty$ for all $\hat{w} > 0$. Then the recurrence relation is given by

$$T(i, \hat{w}) = \begin{cases} \min\{T(i-1, \hat{w}), p_i + T(i-1, \hat{w} - w_i)\}, & \text{if } p_i + T(i-1, \hat{w} - w_i) \leq d_i, \\ T(i-1, \hat{w}), & \text{otherwise.} \end{cases}$$

In the computed table for $T(i, \hat{w})$, there are $n+1$ rows corresponding to $i = 0, \dots, n$ where n is the number of jobs, and $\sum w_j + 1$ corresponding to the potential total weights $\hat{w} = 0, \dots, \sum w_j$. From the table, we can then find the optimal solution to the $(1 \parallel \sum w_j U_j)$ instance in $O(\sum w_j)$ time.

We can actually do slightly better. If w^* is the maximum value of w such that $T(i, w^*)$ is finite, then we can compute the table for $T(i, \hat{w})$ in $O(nw^*)$ time and find the optimal solution from the table in $O(w^*)$ time. So the dynamic programming algorithm runs in time $O(nw^*)$.

Approach 2. This is the approach we will use for our FPTAS. We define $Q(i, \hat{w})$ to be the minimum completion time of a feasible subset S of $[i]$ such that $w([i] \setminus S) \leq \hat{w}$. In other words, $Q(i, \hat{w})$ is the minimum completion time of a chosen subset of the jobs $\{1, \dots, i\}$ such that all of the jobs in the chosen subset are timely (that is, there are no late jobs in an EDD ordering of the chosen subset) and moreover, the sum of the weights of the remaining jobs in $\{1, \dots, i\}$ is $\leq \hat{w}$.

We define $Q(0, \hat{w}) = 0$ for all $\hat{w} \geq 0$ and $Q(0, \hat{w}) = \infty$ for all $\hat{w} < 0$. Then the recurrence relation is given by

$$Q(i, \hat{w}) = \begin{cases} \min\{Q(i-1, \hat{w} - w_i), p_i + Q(i-1, \hat{w})\}, & \text{if } p_i + Q(i-1, \hat{w}) \leq d_i, \\ Q(i-1, \hat{w} - w_i), & \text{otherwise.} \end{cases}$$

In essence, this is the complement to the first approach. But why do we prefer this one instead? This is because it is slightly more efficient. We do not need to compute all of the entries of the table like in the first approach; we only need to start at $\hat{w} = 0$ and stop when we reach the minimum value w^{**} such that $Q(i, w^{**})$ is finite. Observe that this minimum value w^{**} is the same as the optimal value to the $(1 \parallel \sum w_j U_j)$ instance. In many cases, the value of w^{**} is much smaller than that of w^* above.

Our FPTAS then uses an extension of the above observation. If we can compute a lower bound LB on OPT such that $\text{LB} \leq \text{OPT} \leq n \cdot \text{LB}$, then we could replace w^{**} above by $n \cdot \text{LB}$. We design the FPTAS by scaling and rounding the weights, and then applying our second dynamic programming approach above with $Q(i, \hat{w})$. This is quite similar to the FPTAS of the Knapsack problem.

There are two key building blocks for the FPTAS for $(1 \parallel \sum w_j U_j)$.

(1) We need a lower bound LB on OPT computable in polynomial time such that $\text{LB} \leq \text{OPT} \leq n \cdot \text{LB}$.

(2) We need an algorithm (based on dynamic programming) which computes the optimal value and runs in time polynomial in n and LB.

We address (1) by solving the problem $(1 \parallel \max_j w_j U_j)$ on the same instance. This can be done by applying the LCL algorithm (Algorithm 2.10) to the functions

$$h_j(C_j) = \begin{cases} 0, & \text{if } C_j \leq d_j, \\ w_j, & \text{if } C_j > d_j. \end{cases}$$

We also have the following result, and we leave its proof as an exercise.

Proposition 4.15

For any schedule S, we have

$$\max_j w_j U_j^S \leq \sum_j w_j U_j^S \leq n \cdot \max_j w_j U_j^S.$$

Moreover, if S^* is an optimal schedule for $(1 \parallel \max_j w_j U_j)$, then

$$\max_j w_j U_j^{S^*} \leq \mathrm{OPT} \leq n \cdot \max_j w_j U_j^{S^*},$$

where OPT is the optimal value for the $(1 \parallel \sum w_j U_j)$ instance.

We set LB to be the optimal value for the $(1 \parallel \max_j w_j U_j)$ instance. Our scale and round step is straightforward. Given an error parameter $\varepsilon > 0$, let the scaling parameter be $\delta = \varepsilon \cdot \mathrm{LB}/n$. We define $w_j' = \lfloor w_j/\delta \rfloor$ for all jobs j. It is easy to see that $w_j/\delta - 1 \leq w_j' \leq w_j/\delta$.

We apply the second dynamic programming algorithm to the rounded instance with the weights w'. The running time is polynomial in n and LB' where $\mathrm{LB}' = \mathrm{LB}/\delta = n/\varepsilon$ because the optimal value of the rounded instance is $\leq n \cdot \mathrm{LB}'$.

For the rounded instance, let OPT' denote the optimal value. Let A' be the set of late jobs in an optimal schedule, so $w'(A') = \mathrm{OPT}'$. Let A^* be the set of late jobs in an optimal schedule for the original instance so that $w(A^*) = \mathrm{OPT}$.

We claim that $w(A') \leq (1+\varepsilon)\mathrm{OPT}$. This follows from applying some inequalities we have stated earlier and the fact that $w'(A') \leq w'(A^*)$ since A' is optimal for the weights w'. Indeed, we have

$$\begin{aligned}
w(A') &= \sum_{j \in A'} w_j \leq \sum_{j \in A'} (\delta w_j' + \delta) \\
&\leq \delta w'(A') + n\delta \\
&\leq \delta w'(A^*) + n\delta \\
&\leq w(A^*) + \varepsilon \cdot \mathrm{LB} \\
&\leq (1+\varepsilon) w(A^*) \\
&= (1+\varepsilon)\mathrm{OPT}.
\end{aligned}$$

Thus, we see that our algorithm is an FPTAS.

5 Advanced Single Machine Models

5.1 Total Earliness and Tardiness

One could imagine a situation where an earliness penalty is incurred. For example, if one needs to go to convocation in a different city a week from now, they wouldn't want to go immediately as they would need to pay for rent or parking. We define the earliness of a job j by $E_j = \max(0, d_j - C_j)$. Note that unlike the performance measures we have discussed so far, earliness is non-regular because it is decreasing with respect to completion time.

In this section, we cover a generalization of the total tardiness problem. We will focus on the objective

$$\sum_{j=1}^{n} E_j + \sum_{j=1}^{n} T_j.$$

Intuitively, this problem is harder than total tardiness, which is known to be **NP**-hard. Therefore, we will focus on the special case where all jobs have the same due date $d_j = d$.

An optimal schedule for this special case has some useful properties. For example, it is easily shown that there is no idleness between any two jobs for an optimal schedule, as one could close the gap by shifting jobs later if they are early, or shifting jobs earlier if they are late. Moreover, it is possible that an optimal schedule does not start processing the jobs immediately at time 0. Indeed, by the previous property, an increase of d would increase the starting time of the first job if d is sufficiently large.

We also observe that we can divide any sequence of jobs into two disjoint sets and possibly one additional job. One set consists of the jobs that are completed early so that $C_j \leq d$, and we will denote it by J_1. The other set consists of the jobs that are started late, and we will denote it by J_2. There may be another job which is started early but completed late.

Proposition 5.1

There is an optimal sequence in which one job is completed exactly at time d.

Proof. We proceed by contradiction. Suppose there is no such schedule. Then there is always one job that starts its processing before time d and completes its processing after time d. Call this job j^*. Let $|J_1|$ denote the number of jobs that are early and $|J_2|$ denote the number of jobs that are late. If $|J_1| < |J_2|$, then we can shift the entire schedule to the left in such a way that job j^* completes its processing exactly at time d. This implies that the total tardiness decreases by $|J_2|$ times the length of the shift, while the total earliness increases by $|J_1|$ times the length of the shift. Clearly, the objective value is reduced. The case where $|J_1| > |J_2|$ can be treated in a similar way.

The case where $|J_1| = |J_2|$ is somewhat special. In this case, there are many optimal schedules, of which only two satisfy the property given in the lemma. ☐

Due to Proposition 5.1, we can avoid the situation where we have the additional job in the middle altogether. In particular, there is an optimal schedule such that the first job starts at time 0 or later, the jobs in J_1 scheduled first according to Longest Processing Time (LPT), and the jobs in J_2 are scheduled last according to Shortest Processing Time (SPT). In this sense, the schedule is shaped like a "V". Note that for a schedule of the above form, say $1, \ldots, n$, the total earliness is

$$\sum_{j \in J_1} E_j = 0 \cdot p_1 + 1 \cdot p_2 + 2 \cdot p_3 + \cdots + (|J_1| - 1) \cdot p_{|J_1|},$$

while the total tardiness is

$$\sum_{j \in J_2} T_j = 1 \cdot p_{|J_1|+|J_2|} + 2 \cdot p_{|J_1|+|J_2|-1} + \cdots + |J_2| \cdot p_{|J_1|+1}.$$

For an instance in which all optimal schedules start processing the first job some time after $t = 0$ (that is, the due date d is sufficiently large), the following algorithm yields the optimal allocations of jobs to sets J_1 and J_2. We assume that the jobs are ordered such that $p_1 \geq p_2 \geq \cdots \geq p_n$.

Algorithm 5.2: Minimizing Total Earliness and Tardiness with Loose Due Date

Step 1. Assign job 1 to set J_1. Initialize $k = 2$.

Step 2. Assign job k to set J_1 and job $k+1$ to set J_2, or vice versa.

Step 3. If $k + 2 \leq n - 1$, then increase k by 2 and go back to Step 2. If $k + 2 = n$, assign job n to either set J_1 or set J_2 and stop. If $k + 2 = n + 1$, then all jobs have been assigned, so we stop.

This algorithm is somewhat flexible in its assignment of jobs to sets J_1 and J_2. It can be implemented in a way that in the optimal assignment, the total processing time of the jobs assigned to J_1 is minimized. Given the total processing time of the jobs in J_1 and the due date d, it can be verified easily whether the machine indeed must remain idle before it starts processing its first job.

If the due date d is tight and it is necessary to start processing a job at time 0, then the problem is **NP**-hard. The following heuristic which assigns the n jobs to the n positions in the sequence can be very effective, but does not always yield an optimal sequence. Again, we assume that $p_1 \geq p_2 \geq \cdots \geq p_n$.

Algorithm 5.3: Minimizing Total Earliness and Tardiness with Tight Due Date

Step 1. Initialize $\tau_1 = d$ and $\tau_2 = \sum_{j=1}^{n} p_j - d$. Set $k = 1$.

Step 2. If $\tau_1 > \tau_2$, then assign job k to the first unfilled position in the sequence and decrease τ_1 by p_k. If $\tau_1 < \tau_2$, then assign job k to the last unfilled position in the sequence and decrease τ_2 by p_k.

Step 3. If $k < n$, then increase k by 1 and go back to Step 2. If $k = n$, then stop.

We give an example of this heuristic in action.

Example 5.4

Consider the following example with 6 jobs and common due date $d = 180$.

j	1	2	3	4	5	6
p_j	106	100	96	22	20	2

Applying the heuristic yields the following results.

τ_1	τ_2	Assignment	Sequence
180	166	Job 1 placed first	$1, *, *, *, *, *$
74	166	Job 2 placed last	$1, *, *, *, *, 2$
74	66	Job 3 placed first	$1, 3, *, *, *, 2$
-22	66	Job 4 placed last	$1, 3, *, *, 4, 2$
-22	44	Job 5 placed last	$1, 3, *, 5, 4, 2$
-22	24	Job 6 placed last	$1, 3, 6, 5, 4, 2$

To end this section, we discuss some other versions of total earliness and tardiness problems.

- Consider the objective $\sum w' E_j + \sum w'' T_j$ where all due dates are equal with $d_j = d$. All jobs have exactly the same cost function, but the earliness penalty w' and the tardiness penalty w'' are not the same. All the previous properties and algorithms can be generalized relatively easily to take the difference between w' and w'' into account.

- Consider the more general objective $\sum w_j' E_j + \sum w_j'' T_j$ with $d_j = d$ for all jobs j. In this case, the shapes of the cost functions of the jobs are different. The LPT-SPT sequence described above is not necessarily optimal. The first part of the sequence must now be ordered according to Weighted Longest Processing Time first (WLPT), which is in increasing order of w_j/p_j. The second part of the sequence is ordered by Weighted Shortest Processing Time first (WSPT), which we are already familiar with.
- Consider the model with the objective function $\sum w' E_j + \sum w'' T_j$ where we do not assume that the due dates are equal. It is clear that this case is **NP**-hard as it is a more general model of the one we discussed in Section 4.3. This problem has an additional level of complexity. Because of the different due dates, it may not necessarily be optimal to process the jobs one after another without interruption; it may be necessary to have idle times between the processing of consecutive jobs. Therefore, this problem has two aspects: one concerning the sequencing of the jobs, and the other concerning the idle time between the jobs. Approaches for this problem are typically based either on dynamic programming or branch and bound. However, given a predetermined sequence, the timing of the processing of the jobs (and therefore also the idle times) can be determined in polynomial time. This can also be applicable in a more general setting, which we will describe next.
- The most general setting has objective $\sum w_j' E_j + \sum w_j'' T_j$ where the jobs have different due dates and weights. This problem is strongly **NP**-hard since it is harder than $(1 \mid\mid \sum w_j T_j)$, which is also known to be strongly **NP**-hard. But again, given a predetermined sequence of the jobs, the timing of the jobs can be determined in polynomial time. However, we will not go into detail about this.

5.2 Primary and Secondary Objectives

In practice, a scheduler is often concerned with more than one objective. For example, one might want to minimize inventory costs and meet due dates. It would then be of interest, for instance, to find a schedule that minimizes a combination of $\sum C_j$ and $L_{\max}$.

In many cases, there is more than one optimal schedule that minimizes a given objective. A scheduler may then wish to consider the set of all schedules that are optimal with respect to one objective (which we will call the primary objective), and then search within this set of schedules for the schedule that is best with regard to a secondary objective. If the primary objective is γ_1 and the secondary objective is γ_2, then such a problem is denoted by $(\alpha \mid \beta \mid \gamma_1^{(1)}, \gamma_2^{(2)})$.

We begin with the simple example $(1 \mid\mid \sum C_j^{(1)}, L_{\max}^{(2)})$ where the primary objective is the total completion time and the secondary objective is the maximum lateness. If there are no jobs with identical processing times, then there is exactly one schedule that minimizes the total completion time (namely the one obtained by SPT), so there is no freedom to minimize $L_{\max}$. If there are jobs with identical processing times, then there are multiple schedules that minimize the total completion time. A set of jobs with identical processing times is preceded by a job with a strictly shorter processing time and followed by a job with a strictly longer processing time. Jobs with identical processing times have to be processed one after another, but the key is that they may be done in any order. Then to minimize $L_{\max}$, we can put these jobs with identical processing times in EDD order. We can refer to this rule as SPT/EDD, since the jobs are first scheduled according to SPT and ties are broken according to EDD. This rule can generalized to the case where the primary objective is the total weighted completion time $\sum w_j C_j$.

Consider now the problem $(1 \mid\mid L_{\max}^{(1)}, \sum C_j^{(2)})$, which has the same two objectives with reversed priorities. We know that the EDD rule minimizes $L_{\max}$; suppose that the value of this minimum $L_{\max}$ is z. Then the original problem can be transformed to another equivalent problem. By creating a new set of due dates $\overline{d_j} = d_j + z$, we see that these due dates are now deadlines. The problem is to find a schedule which minimizes $\sum C_j$ subject to the constraint that every job is completed by its deadline. We denote this problem by $(1 \mid \text{hard } d_j \mid \sum C_j)$. The algorithm for finding the optimal schedule is based on the following result.

Lemma 5.5

There exists a schedule that minimizes $\sum C_j$ in which job k is scheduled last if and only if

(1) $\overline{d_k} \geq \sum_{j=1}^{n} p_j$, and

(2) $p_k \geq p_\ell$ for all jobs ℓ such that $\overline{d_\ell} \geq \sum_{j=1}^{n} p_j$.

Proof. Assume towards a contradiction that job k is not scheduled last. There is a set of jobs that is scheduled after job k, and some job ℓ that is scheduled last. Note that job ℓ must satisfy condition (1) for otherwise it would not meet its due date. This means that job ℓ does not satisfy condition (2) and so $p_\ell < p_k$. After swapping jobs k and ℓ, the sum of the completion times of jobs k and ℓ decreases, and the sum of the completion times of jobs scheduled between k and ℓ also decreases. Hence, the original schedule that positioned job ℓ last could not have minimized $\sum C_j$. □

In the following algorithm, J^c represents the set of jobs that have not been scheduled yet.

Algorithm 5.6: Minimizing Total Completion Time with Deadlines

Step 1. Set $k = n$, $\tau = \sum_{j=1}^{n} p_j$, and $J^c = \{1, \ldots, n\}$.

Step 2. Find a job $k^* \in J^c$ such that $\overline{d_{k^*}} \geq \tau$ and $p_{k^*} \geq p_\ell$ for all jobs $\ell \in J^c$ such that $\overline{d_\ell} \geq \tau$. Put job k^* in position k of the sequence.

Step 3. Decrease k by 1, decrease τ by p_{k^*}, and delete job k^* from J^c.

Step 4. If $k \geq 1$, then go back to Step 2. Otherwise, stop.

We give a simple example of the use of this backward algorithm.

Example 5.7

Consider the following instance of 5 jobs.

j	1	2	3	4	5
p_j	4	6	2	4	2
d_j	10	12	14	18	18

For ease of notation, we will let $\hat{J}(\tau) = \{\ell \in J^c : \overline{d_\ell} \geq \tau\}$ be the set of eligible jobs that have not been scheduled yet. We start with $\tau = \sum_{j=1}^{5} p_j = 18$, and applying the algorithm yields the following results.

Iteration	τ	$\hat{J}(\tau)$	k^*
1	18	$\{4, 5\}$	4
2	14	$\{3, 5\}$	3 or 5
3	12	$\{2, 5\}$ or $\{2, 3\}$	2
4	6	$\{1, 5\}$ or $\{1, 3\}$	1
5	2	$\{5\}$ or $\{3\}$	5 or 3

Thus, we obtain two optimal schedules, which are $5, 1, 2, 3, 4$ and $3, 1, 2, 5, 4$.

It can be shown that even when preemptions are allowed, the optimal schedules are non-preemptive for the $(1 \mid \text{hard } d_j \mid \sum C_j)$ problem. We can also assume feasibility of the problem in general as it can be checked easily using the EDD rule.

Finally, we note that a fairly large number of problems of the form $(1 \mid \beta \mid \gamma_1^{(1)}, \gamma_2^{(2)})$ have been studied in the literature. Very few of them can be solved in polynomial time. However, problems of the type $(1 \mid \beta \mid \sum w_j C_j^{(1)}, \gamma_2^{(2)})$ tend to be easy based on the analysis of the $(1 \mid\mid \sum C_j^{(1)}, L_{\max}^{(2)})$ problem we discussed at the beginning of this section. We will see one such example on Assignment 6.

5.3 Parametric Analysis of Multiple Objectives

Suppose that we have the two objectives γ_1 and γ_2. If the overall objective is $\theta_1\gamma_1 + \theta_2\gamma_2$ where θ_1 and θ_2 are the weights of the two objectives, then we can denote the scheduling problem by $(1 \mid \beta \mid \theta_1\gamma_1 + \theta_2\gamma_2)$. Multiplying both weights by the same constant does not change the problem, so we may assume without loss of generality that $\theta_1 + \theta_2 = 1$. For this section, we will focus on a specific class of schedules.

Definition 5.8

A schedule is called **Pareto optimal** if it is not possible to decrease the value of one objective without increasing the value of the other.

All Pareto optimal schedules can be represented by a set of points in the (γ_1, γ_2) plane. This set of points illustrates the tradeoffs between the two objectives. Consider the two objectives we analyzed in the previous section, namely $\sum C_j$ and $L_{\max}$. The two cases we covered are the two extreme points of the tradeoff curve. Note that if $\theta_1 \to 0$ and $\theta_2 \to 1$, then

$$(1 \mid \beta \mid \theta_1\gamma_1 + \theta_2\gamma_2) \to (1 \mid \beta \mid \gamma_2^{(1)}, \gamma_1^{(2)}),$$

while if $\theta_2 \to 0$ and $\theta_1 \to 1$, then

$$(1 \mid \beta \mid \theta_1\gamma_1 + \theta_2\gamma_2) \to (1 \mid \beta \mid \gamma_1^{(1)}, \gamma_2^{(2)}).$$

We will now focus on generating all Pareto optimal schedules for the problem $(1 \mid\mid \theta_1 \sum C_j + \theta_2 L_{\max})$.

On one end of the extremes is the $(1 \mid\mid \sum L_{\max}^{(1)}, \sum C_j^{(2)})$ problem, which could be solved using a more complicated backward algorithm where we have $L_{\max} = L_{\max}(\text{EDD})$.

On the other end, we have the $(1 \mid\mid \sum C_j^{(1)}, L_{\max}^{(2)})$ problem, where the total completion time was minimized by the SPT rule and ties were broken by EDD. We denoted this strategy by SPT/EDD, and we easily see that $L_{\max} = L_{\max}(\text{SPT/EDD})$ in this case. It is clear that

$$L_{\max}(\text{EDD}) \le L_{\max}(\text{SPT/EDD}).$$

The algorithm that generates all Pareto optimal solutions in the tradeoff curve contains two loops. One series of steps in the algorithm (the inner loop) is an adaptation of Algorithm 5.6. In addition to the optimal schedule with a maximum allowable $L_{\max}$, these steps determine the minimum increment δ in the $L_{\max}$ that would allow for a decrease in the minimum $\sum C_j$. The second (outer) loop of the algorithm contains the structure that generates all the Pareto optimal points. The outer loop calls the inner loop at each Pareto optimal point to generate a schedule at that point and also to determine how to move to the next efficient point. The algorithm starts out with the EDD schedule that generates the first Pareto optimal point in the upper left part of the trade-off curve. It determines the minimum increment in the $L_{\max}$ needed to achieve a reduction in $\sum C_j$. Given this new value of $L_{\max}$, it uses Algorithm 5.6 to determine the schedule which minimizes $\sum C_j$, and proceeds to determine the next increment. This goes on until the algorithm reaches $L_{\max}(\text{SPT/EDD})$.

Algorithm 5.9: Tradeoffs between Total Completion Time and Maximum Lateness

Step 1. Set $r = 1$. Set $L_{\max} = L_{\max}(\text{EDD})$ and $\overline{d_j} = d_j + L_{\max}$.

Step 2. Set $k = n$ and $J^c = \{1, \ldots, n\}$. Set $\tau = \sum_{j=1}^{n} p_j$ and $\delta = \tau$.

Step 3. Find a job $j^* \in J^c$ such that $\overline{d_{j^*}} \geq \tau$ and $p_{j^*} \geq p_\ell$ for all jobs $\ell \in J^c$ such that $\overline{d_\ell} \geq \tau$. Put job j^* in position k of the sequence.

Step 4. If there is no job ℓ such that $\overline{d_\ell} < \tau$ and $p_\ell > p_{j^*}$, then go to Step 5. Otherwise, let

$$\delta^{**} = \min\{\tau - \overline{d_\ell} : \text{jobs } \ell \text{ such that } \overline{d_\ell} < \tau \text{ and } p_\ell > p_{j^*}\}$$

and set $\delta = \min\{\delta, \delta^{**}\}$.

Step 5. Decrease k by 1, decrease τ by p_{j^*}, and delete job j^* from J^c. If $k \geq 1$, then go to Step 3. If $k = 0$, then go to Step 6.

Step 6. Set $L_{\max} = L_{\max} + \delta$. If $L_{\max} > L_{\max}(\text{SPT/EDD})$, then stop. Otherwise, set $r = r + 1$, $\overline{d_j} = \overline{d_j} + \delta$, and go to Step 2.

The outer loop consists of Steps 1 and 6, while the inner loop consists of Steps 2, 3, 4, and 5. Steps 2, 3, and 5 represent an adaptation of Algorithm 5.6, while Step 4 computes the minimum increment in the $L_{\max}$ needed to achieve a subsequent reduction in $\sum C_j$.

It can be shown that the number of Pareto optimal solutions is bounded above by $\binom{n}{2} + O(1)$, which is $O(n^2)$. Generating one Pareto optimal solution can be done in $O(n \log n)$ time. Therefore, the total computation time of Algorithm 5.9 is $O(n^3 \log n)$.

We now give an example of the use of the above algorithm.

Example 5.10

Consider the following set of 5 jobs.

j	1	2	3	4	5
p_j	1	3	6	7	9
d_j	30	27	20	15	12

The EDD sequence is $5, 4, 3, 2, 1$ with $L_{\max}(\text{EDD}) = 2$ and total completion time 98. The SPT/EDD sequence is $1, 2, 3, 4, 5$ with $L_{\max}(\text{SPT/EDD}) = 14$ and total completion time 58.

We give an example of one full iteration of the inner loop.

- Initially, we have $k = 5$, $\tau = \sum_{j=1}^{5} p_j = 26$, and so the set of eligible jobs to put at the end of the sequence is $\{1, 2\}$. Since job 2 has the largest processing time, we place it last. Then we have $\delta^{**} = \tau - \overline{d_3} = 26 - 22 = 4$ and so $\delta = 4$.
- Next, we have $k = 4$, $\tau = 23$, and job 1 is the only eligible job. Then $\delta^{**} = 23 - 22 = 1$ and $\delta = 1$.
- Then, we have $k = 3$, $\tau = 22$, and the only eligible job is job 3. We then have $\delta^{**} = \tau - \overline{d_4} = 22 - 17 = 5$.
- For the next iteration of the inner loop, we have $k = 2$, $\tau = 16$, and job 4 is the only eligible job. Then $\delta^{**} = \tau - \overline{d_5} = 16 - 12 = 4$.
- Finally, we pick job 5 at the end, and obtain the Pareto optimal schedule $5, 4, 3, 1, 2$ with $\delta = 1$.

The full results are given in the following table.

Iteration	$\sum C_j$	$L_{\max}$	Pareto optimal schedule	current $\tau + \delta$	δ
1	96	2	$5, 4, 3, 1, 2$	32 29 22 17 14	1
2	77	3	$1, 5, 4, 3, 2$	33 30 23 18 15	2
3	75	5	$1, 4, 5, 3, 2$	35 32 25 20 17	1
4	64	6	$1, 2, 5, 4, 3$	36 33 26 21 18	2
5	62	8	$1, 2, 4, 5, 3$	38 35 28 23 20	3
6	60	11	$1, 2, 3, 5, 4$	41 38 31 26 23	3
7	58	14	$1, 2, 3, 4, 5$	44 41 34 29 26	STOP

However, when one would consider the objective $\theta_1 \sum C_j + \theta_2 L_{\max}$, certain Pareto optimal schedules may never be optimal, no matter what the weights are.

Consider the generalization $(1 \mid\mid \theta_1 \sum w_j C_j + \theta_2 L_{\max})$. It is clear that the two extreme points of the tradeoff curve can be determined in polynomial time using WSPT/EDD and EDD. However, even though the two endpoints of the tradeoff curve can be analyzed in polynomial time, the problem with arbitrary weights θ_1 and θ_2 is **NP**-hard.

The tradeoff curve that corresponds to the example in this section has the shape of a staircase. This shape is fairly common in a single machine environment with multiple objectives, especially when preemptions are not allowed. However, in other machine environments such as parallel machines, smoother curves may occur, especially when preemptions are allowed. We will not cover more parametric analysis in this course though.

6 Parallel Machine Models

6.1 Makespan without Preemptions

6.1.1 List Scheduling and Longest Processing Time

The makespan is not a very interesting objective for a single machine. However, for parallel machine models, minimizing the makespan has the effect of balancing the load over the various machines, which is an important objective in practice. Recall that we denote this by $(P_m \parallel C_{\max})$.

One can see that $(P_2 \parallel C_{\max})$ is **NP**-hard via a reduction from PARTITION. Indeed, consider an instance of PARTITION, where we are given $a_1, \ldots, a_t \in \mathbb{Z}^+$ and want to find a subset $S \subseteq \{1, \ldots, t\}$ such that

$$\sum_{j \in S} a_j = \frac{1}{2} \sum_{j=1}^{t} a_j = b.$$

Let $n = t$ be the number of jobs each with processing time $p_j = a_j$. Then it is easy to check that there is a schedule with optimal value at most $\frac{1}{2} \sum_{j=1}^{n} p_j$ if and only if there is a solution to the PARTITION instance.

Over the years, there have been many heuristics developed for $(P_m \parallel C_{\max})$. We focus on one such heuristic, and present it by discussing some history on how this algorithm was developed. First, we take a look at the list scheduling rule, which runs in $O(n \log m)$ time using a priority queue for the loads.

ALGORITHM 6.1: LIST SCHEDULING

Given n jobs in some arbitrary order, assign job j to machine i whose load is smallest so far for each $j \in [n]$.

There are two obvious lower bounds on OPT, namely

(1) $\text{OPT} \geq \max_{j \in [n]} p_j$ since the jobs are not preemptive, and

(2) $\text{OPT} \geq \frac{1}{m} \sum_{j=1}^{n} p_j$ because at least one machine must have load which is at least the average.

We shall denote the load of a machine i by $L(i) = p(S(i)) = \sum_{j \in S(i)} p_j$, where $S(i)$ is the set of jobs assigned to machine i.

THEOREM 6.2

Algorithm 6.1 is a 2-approximation for $(P_m \parallel C_{\max})$.

PROOF. Let i denote the machine which the highest load determining the objective value. Let j be the last job scheduled on machine i. When job j was assigned to machine i, it had the smallest load. Its load before the assignment was $L(i) - p_j$, and thus $L(i) - p_j \leq L(k)$ for all $1 \leq k \leq m$. Summing up this inequality over all k and dividing by m, we obtain

$$L(i) - p_j \leq \frac{1}{m} \sum_{k=1}^{m} L(k) = \frac{1}{m} \sum_{j=1}^{n} p_j \leq \text{OPT}.$$

We deduce that $L(i) = (L(i) - p_j) + p_j \leq 2 \cdot \text{OPT}$. □

Is this analysis tight? That is, are there any examples where the factor is as bad as 2? The answer is essentially yes. Suppose there are m machines. Take $m(m-1)$ jobs each with processing time 1, and let the last job in the order have processing time m. Then the list scheduling algorithm will try to balance the $m(m-1)$ jobs among the m machines first. Then we are forced to run the last job with processing time m last on one of the machines, and that machine has load $2m-1$. An optimal schedule instead places the job with processing time m on its own machine, and balances the load of the $m(m-1)$ jobs between the remaining $m-1$ machines. This gives a makespan of m instead.

It can be shown that Algorithm 6.1 is in fact a $(2-\frac{1}{m})$-approximation, and the above example shows that this is exactly a tight bound.

We now consider the **Longest Processing Time first (LPT)** rule.

Algorithm 6.3: LPT

Sort the n jobs in decreasing order of p_j, then run Algorithm 6.1.

This is equivalent to the following procedure: at time $t=0$, assign the m longest jobs to the m machines. Afterwards, whenever a machine is freed, the longest job among those not yet processed is put on the machine. This heuristic tries to place the shorter jobs more towards the end of the schedule, where they can be used for balancing the loads.

Exercise 6.4

Show that if an optimal schedule results in at most two jobs on any machine, then LPT is optimal.

Under the LPT rule in the case that $n \geq m+1$, we also have an additional lower bound on OPT. This is given by $\text{OPT} \geq 2p_{m+1}$ because at least one machine must run two jobs by the time it gets to the $(m+1)$-th job, and that machine has load at least $2p_{m+1}$ as the jobs are in decreasing order of the processing times.

Theorem 6.5

Algorithm 6.3 is a $\frac{3}{2}$-approximation for $(P_m \parallel C_{\max})$.

Proof. This is the same proof as Theorem 6.2, except that we now have $p_j \leq p_{m+1} \leq \text{OPT}/2$. □

This is actually a very crude bound. We give a more sophisticated analysis showing that the LPT rule is a $(\frac{4}{3}-\frac{1}{3m})$-approximation, and that this bound is tight.

Theorem 6.6

For $(P_m \parallel C_{\max})$, we have

$$\frac{C_{\max}(\text{LPT})}{C_{\max}(\text{OPT})} \leq \frac{4}{3} - \frac{1}{3m}.$$

Proof. We proceed by contradiction. Suppose there are counterexamples with ratio strictly larger than $\frac{4}{3}-\frac{1}{3m}$. Among these counterexamples, there must be a counterexample with the least number of jobs.

Suppose this smallest counterexample has n jobs. This counterexample has a useful property: under LPT, the shortest job is the last job to start its processing and also the last job to finish its processing. To see why this is true, note that by the definition of LPT, the shortest job is the last to finish its processing. On the other hand, if this job was not last to finish processing, then the deletion of this smallest job would result in a counterexample with fewer jobs since $C_{\max}(\text{LPT})$ remains the same while $C_{\max}(\text{OPT})$ either stays the same or decreases, which is a contradiction to our assumption that our counterexample was minimal.

Therefore, for our smallest counterexample, the starting time of the shortest job is $C_{\max}(\text{LPT}) - p_n$. At this point, all other machines are busy, so we have

$$C_{\max}(\text{LPT}) - p_n \leq \frac{1}{m}\sum_{j=1}^{n-1} p_j.$$

The right-hand side is an upper bound on the starting time of the shortest job, and is achieved when scheduling the first $n-1$ jobs according to LPT results in each machine having exactly the same amount of processing to do. Then we have

$$C_{\max}(\text{LPT}) \leq p_n + \frac{1}{m}\sum_{j=1}^{n-1} p_j = p_n\left(1 - \frac{1}{m}\right) + \frac{1}{m}\sum_{j=1}^{n} p_j.$$

We also know that $C_{\max}(\text{OPT}) \geq \frac{1}{m}\sum_{j=1}^{n} p_j$, so for the counterexample, we obtain

$$\frac{4}{3} - \frac{1}{3m} < \frac{C_{\max}(\text{LPT})}{C_{\max}(\text{OPT})} \leq \frac{p_n(1-\frac{1}{m}) + \frac{1}{m}\sum_{j=1}^{n} p_j}{C_{\max}(\text{OPT})} \leq \frac{p_n(1-\frac{1}{m})}{C_{\max}(\text{OPT})} + 1.$$

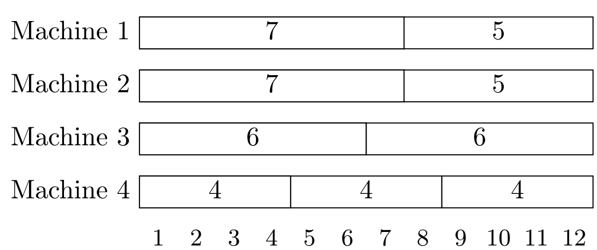
Rearranging gives us $C_{\max}(\text{OPT}) < 3p_n$. Note that this is a strict inequality. Since p_n is the shortest job, this implies that for the smallest counterexample, the optimal schedule results in at most two jobs on each machine. But by Exercise 6.4, LPT is optimal under this condition, which is a contradiction. $\square$

Example 6.7

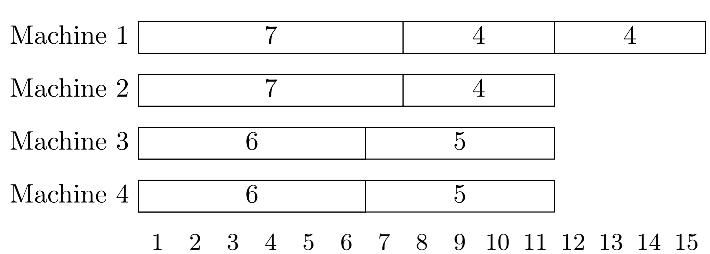
We consider an example where LPT performs very badly and attains the worst case bound. Suppose there are $m = 4$ parallel machines with the following $n = 9$ jobs.

j	1	2	3	4	5	6	7	8	9
p_j	7	7	6	6	5	5	4	4	4

Then the following schedule is optimal with $C_{\max} = 12$.



On the other hand, LPT gives the following schedule which has $C_{\max} = 15$, which is the worst case we could possibly have from Theorem 6.6.



6.1.2 Makespan with Precedence Constraints

Now, consider the same problem with the jobs subject to precedence constraints, namely $(P_m \mid \text{prec} \mid C_{\max})$. From a complexity point of view, this problem has to be at least as hard as the problem without precedence constraints. In particular, it can be shown that $(P_m \mid \text{prec} \mid C_{\max})$ is strongly **NP**-hard for $2 \leq m < n$, even in the case where the precedence constraints are chains. To obtain some insight into the effects of precedence constraints, we consider a number of special cases.

Suppose there are an unlimited number of machines in parallel, or that $m \geq n$ so the number of machines is at least as large as the number of jobs. We can denote this problem by $(P_\infty \mid \text{prec} \mid C_{\max})$. This is a classical problem in the field of project planning, and its study has led to the development of the well-known **Critical Path Method (CPM)** and **Project Evaluation Review Technique (PERT)**. In this case, the optimal schedule and the minimum makespan are determined by a very simple algorithm.

Algorithm 6.8: Minimizing the Makespan of a Project

Schedule the jobs one at a time starting at time zero. Whenever a job has been completed, start all jobs of in which all predecessors have been completed (that is, the set of all schedulable jobs).

So $(1 \mid \text{prec} \mid C_{\max})$ and $(P_\infty \mid \text{prec} \mid C_{\max})$, are easy. But we stated earlier that $(P_m \mid \text{prec} \mid C_{\max})$ is strongly **NP**-hard for $2 \leq m < n$. Even $(P_m \mid p_j = 1, \text{prec} \mid C_{\max})$, where the processing times are all unit, is not easy. However, by constraining the problem further and assuming that the precedence graph takes the form of a tree (either an intree or an outtree), the problem $(P_m \mid p_j = 1, \text{tree} \mid C_{\max})$ is easily solvable. This leads to the well-known scheduling rule known as the **Critical Path (CP)** rule.

Algorithm 6.9: Critical Path (CP)

The job at the head of the longest string of jobs in the precedence constraints graph has the highest priority. Ties can be broken arbitrarily.

We also consider a different priority rule that considers the largest total number of successors (not just the immediate successors) in the precedence graph. This is called the **Largest Number of Successors first (LNS)** rule.

Algorithm 6.10: Largest Number of Successors first (LNS)

The job with the largest total number of successors in the precedence graph has the highest priority. Ties can be broken arbitrarily.

Note that the LNS rule is equivalent to the CP rule when the precedence graph takes the form of chains or an intree. It can also be shown that LNS is optimal for outtrees, and thus LNS is also optimal for $(P_m \mid p_j = 1, \text{tree} \mid C_{\max})$.

We will not prove the optimality of the CP and LNS rules for $(P_m \mid p_j = 1, \text{tree} \mid C_{\max})$. But how do these rules perform for arbitrary precedence constraints when all processing times are equal to 1?

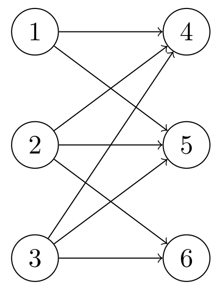
In the case where there are two parallel machines, it can be shown that

$$\frac{C_{\max}(\text{CP})}{C_{\max}(\text{OPT})} \leq \frac{4}{3}.$$

When there are more than two parallel machines, the worst case ratio is larger. We show that the worst case bound can be reached for two machines in the following example.

Example 6.11

Consider the instance of $(P_2 \mid p_j = 1, \text{prec} \mid C_{\max})$ with precedence constraints given by the following directed acyclic graph.



This is almost a complete bipartite graph; it is only missing the edge from job 1 to job 6. An optimal schedule is given below with $C_{\max} = 3$.

Machine 1	2	1	4
Machine 2	3	6	5

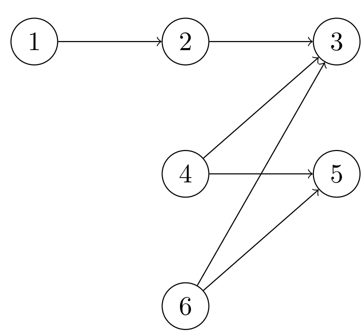
On the other hand, the CP rule could arbitrarily pick job 1 to be processed at time 0, which would prevent job 6 from running at time 1 as its predecessors are jobs 2 and 3. One such schedule obtained from the CP rule is as follows, and it has $C_{\max} = 4$, which matches the worst case bound above.

Machine 1	1	3	4	6
Machine 2	2		5	

Next, we give an example where LNS does not yield an optimal schedule for arbitrary precedence constraints.

Example 6.12

Consider the instance of $(P_2 \mid p_j = 1, \text{prec} \mid C_{\max})$ with precedence constraints given by the following directed acyclic graph.



An optimal schedule is given below with $C_{\max} = 3$.

Machine 1	1	2	3
Machine 2	4	6	5

The LNS rule would pick jobs 4 and 6 first as they each have two successors. This causes job 2 to finish at time 3 at the earliest, preventing job 3 from being run afterwards. So the LNS rule yields a schedule with $C_{\max} = 4$.

Machine 1	4	1	2	3
Machine 2	6	5		

Both the CP rule and the LNS rule have more generalized versions that can be applied to problems with arbitrary job processing times. Instead of counting the number of jobs (as in the case with unit processing times), these more generalized versions prioritize based on the total amount of processing remaining to be done on the jobs in question. The CP rule then gives the highest priority to the job that is heading the string of jobs with the largest total amount of processing (with the processing time of the job itself also being included in this total). The generalization of the LNS rule gives the highest priority to that job that precedes the largest total amount of processing; again the processing time of the job itself is also included in the total. The LNS name is clearly not appropriate for this generalization with arbitrary processing times, as it refers to a number of jobs rather than to a total amount of processing.

6.1.3 Makespan with Machine Dependent Constraints

We consider another generalization of the $(P_m \mid\mid C_{\max})$ problem where each job j is only allowed to be run on a subset M_j of the m parallel machines. We will again consider the case with unit processing times, denoted by $(P_m \mid p_j = 1, M_j \mid C_{\max})$.

We note that this problem can be represented using a bipartite graph with n edges $1, \ldots, n$ corresponding to the jobs and m edges $\mathrm{mc}_1, \ldots, \mathrm{mc}_m$ corresponding to the machines. We place an edge (j, mc_i) in the graph if and only if $i \in M_j$.

We say that the sets M_j are **nested** if exactly one of the following conditions hold for any two jobs j and k:

(i) $M_j = M_k$;

(ii) $M_j \subsetneq M_k$;

(iii) $M_k \subsetneq M_j$; or

(iv) $M_j \cap M_k = \varnothing$.

When the sets M_j are nested, the **Least Flexible Job (LFJ)** rule plays an important role.

Algorithm 6.13: Least Flexible Job (LFJ)

Every time a machine is freed, select among the available jobs the job that can be processed on the *smallest* number of machines. Ties can be broken arbitrarily.

This rule is rather crude as it does not specify, for example, which machine should be considered first when several machines become available at the same time. It can be shown that the LFJ rule is optimal for $(P_m \mid p_j = 1, M_j \mid C_{\max})$ when the sets M_j are nested by using a standard interchange argument. In particular, when there are two machines, the sets M_j are always nested, so the LFJ rule is optimal for $(P_2 \mid p_j = 1, M_j \mid C_{\max})$. However, in the case that $m \geq 3$ and the sets M_j are arbitrary, the LFJ rule will not always yield an optimal schedule. We give an example of this below.

Example 6.14

Consider the following instance of $(P_4 \mid p_j = 1, M_j \mid C_{\max})$ with $n = 8$ jobs.

j	1	2	3	4	5	6	7	8
M_j	$\{1,2\}$	$\{1,3,4\}$	$\{1,3,4\}$	$\{2\}$	$\{3,4\}$	$\{3,4\}$	$\{3,4\}$	$\{3,4\}$

It is clear that the sets M_j are not nested by considering M_1 and M_2. We now run the LFJ rule. Consider machine 1 first. The least flexible job that can be processed on machine 1 is job 1 since it can be processed on only two machines, while jobs 2 and 3 can be processed on three machines. The least flexible job to be processed on machine 2 is clearly job 4. At time 0, the least flexible jobs to be processed on machines 3 and 4 could be jobs 5 and 6. At time 1, after jobs 1, 4, 5, and 6 have completed their processing on the four machines, the least flexible job to be processed on machine 1 is job 2. But none of the remaining jobs can be processed on machine 2, so it is forced to idle. The least flexible jobs to go on machines 3 and 4 are jobs 7 and 8. Finally, job 3 can be run on machine 1 at time 3, so $C_{\max} = 3$. This schedule is illustrated below.

Machine 1	1	2	3
Machine 2	4		
Machine 3	5	7	
Machine 4	6	8	

An optimal schedule with $C_{\max} = 2$ is given below, so LFJ was not optimal.

Machine 1	2	3
Machine 2	1	4
Machine 3	5	6
Machine 4	7	8

From Example 6.14, one may expect that if a number of machines are free at the same point in time, it is advantageous to consider first the least flexible machine. The flexibility of a machine could be defined as the number of remaining jobs that can be processed (or the total amount of processing that can be done) on that machine. However, assigning any job to the **Least Flexible Machine (LFM)** at each point in time does not guarantee an optimal schedule in the case of Example 6.14.

Heuristics can be designed that combine the LFJ rule with the LFM rule, giving priority to the least flexible jobs on the least flexible machines. That is, consider at each point in time first the Least Flexible Machine (LFM) (that is, the machine that can process the smallest number of jobs) and assign to this machine the least flexible job that can be processed on it. Any ties may be broken arbitrarily. This heuristic may be referred to as the LFM-LFJ heuristic. However, we find again that the LFM-LFJ does not yield an optimal schedule for Example 6.14.

6.2 Makespan with Preemptions

We now consider the same problem with preemptions allowed, namely the $(P_m \mid \text{prmp} \mid C_{\max})$ problem. Usually, but not always, allowing preemptions simplifies the analysis of a problem. This is indeed the case for this problem, where it actually turns out that many schedules are optimal. First, consider the following linear programming formulation of the problem, where x_{ij} is the total time spent by machine i on job j.

$$\begin{aligned}
\min \quad & C_{\max} \\
\text{s.t.} \quad & \sum_{i=1}^{m} x_{ij} = p_j, && j \in [n] \\
& \sum_{i=1}^{m} x_{ij} \leq C_{\max}, && j \in [n] \\
& \sum_{j=1}^{n} x_{ij} \leq C_{\max}, && i \in [m] \\
& x_{ij} \geq 0, && i \in [m],\, j \in [n].
\end{aligned}$$

The first set of constraints ensures that each job receives the required amount of processing. The second set of constraints ensures that the total amount of processing of each job is less than the makespan. The third set of constraints ensures that the total amount of processing on each machine is less than the makespan. Finally, the last constraint ensures that the execution fragments are non-negative.

Since $C_{\max}$ is basically a decision variable and not an element of the resource vector of the linear program, we can rewrite the second and third set of constraints as

$$\begin{aligned}
C_{\max} - \sum_{i=1}^{m} x_{ij} &\geq 0, && j \in [n] \\
C_{\max} - \sum_{j=1}^{n} x_{ij} &\geq 0, && i \in [m].
\end{aligned}$$

This LP can be solved in polynomial time, but the solution of the LP does not prescribe an actual schedule; it merely specifies the amount of time job j should spend on machine i. However, with this information, a schedule can easily be constructed.

We consider another algorithm for $(P_m \mid \text{prmp} \mid C_{\max})$. This algorithm is based on the fact that it is easy to obtain an expression for the makespan under the optimal schedule. In the next lemma, we establish a lower bound. We leave its proof as an exercise.

Lemma 6.15

Under the optimal schedule for $(P_m \mid \text{prmp} \mid C_{\max})$, we have

$$C_{\max} \geq \max\left(p_1, \frac{1}{m}\sum_{j=1}^{n} p_j\right) =: C^*_{\max}.$$

Having a lower bound allows for the construction of a very simple algorithm that minimizes the makespan. The fact that this algorithm actually produces a schedule with a makespan that is equal to the lower bound shows that the algorithm yields an optimal schedule. This algorithm is also known as **McNaughton's wrap-around rule**.

Algorithm 6.16: Minimizing Makespan with Preemptions

1. Take the n jobs and process them one after another on a single machine in any sequence (that is, without preemption). The makespan is then equal to the sum of the n processing times and is at most $m \cdot C^*_{\max}$.
2. Take this single machine schedule and divide it into m equal parts.
3. Execute each of the m parts in Step 2 on a separate machine.

It is clear that the resulting schedule is feasible. Part of a job may appear at the end of the schedule for machine i while the remaining part may appear at the beginning of the schedule for machine $i+1$. As preemptions are allowed and the processing time of each job is less than $C^*_{\max}$, such a schedule is feasible. Moreover, this schedule has $C_{\max} = C^*_{\max}$, so it is also optimal.

Next, we consider the **Longest Remaining Processing Time first (LRPT)** rule. This schedule is the preemptive version of LPT (Algorithm 6.3). This schedule is structurally appealing, but it is mainly of academic interest. From a practical point, it has a serious drawback. In the deterministic case, the number of preemptions needed is usually infinite.

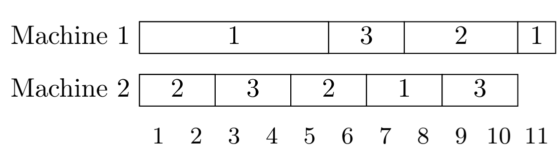
Example 6.17

Consider two jobs with unit processing time on a single machine. Under LRPT, the two jobs continuously have to rotate and wait for their next turn on the machine. That is, a job stays on the machine for a time period ε and after every time period ε, the job waiting preempts the machine. The makespan is equal to 2, and is independent of the schedule. However, the sum of completion times is $4-\varepsilon$, while under the non-preemptive schedule, it is 3. This suggests that LRPT is a bad rule for $(P_m \mid \text{prmp} \mid \sum C_j)$, but we are not interested in this problem at the moment.

We will now assume that we are working under a discrete time framework. All processing times are assumed to be integer, and the decision-maker is only allowed to preempt a machine at integer times. The following is an example of the LRPT rule under discrete time.

Example 6.18

Suppose there are two machines, and three jobs $1, 2, 3$ with processing times $8, 7, 6$ respectively. Then the schedule under LRPT is depicted below and has makespan 11.



To prove that LRPT is optimal for $(P_m \mid \text{prmp} \mid C_{\max})$ under discrete time, we require some special notation.

Suppose that at integer time t, the remaining processing times of the n jobs are $p_1(t), \dots, p_n(t)$. We let $\overline{p}(t) = (p_1(t), \dots, p_n(t))$ be the vector of processing times. Let $p_{(j)}$ denote the j-th largest element of $\overline{p}(t)$. We say that $\overline{p}(t)$ **majorizes** $\overline{q}(t)$, written $\overline{p}(t) \geq_m \overline{q}(t)$, if for all $1 \leq k \leq n$, we have

$$\sum_{j=1}^{k} p_{(j)}(t) \geq \sum_{j=1}^{k} q_{(j)}(t).$$

Example 6.19

Consider the vectors $\overline{p}(t) = (4, 8, 2, 4)$ and $\overline{q}(t) = (3, 0, 6, 6)$. Placing the elements of the vectors in decreasing order yields $(8, 4, 4, 2)$ and $(6, 6, 3, 0)$, and it is easily seen that $\overline{p}(t) \geq_m \overline{q}(t)$.

Lemma 6.20

If $\overline{p}(t) \geq_m \overline{q}(t)$, then LRPT applied to $\overline{p}(t)$ results in a makespan that is greater or equal to the makespan obtained from applying LRPT to $\overline{q}(t)$.

Proof. We proceed by induction on the *total* amount of remaining processing. To show that the lemma holds for $\overline{p}(t)$ and $\overline{q}(t)$ with total processing time $\sum_{j=1}^{n} p_j(t)$ and $\sum_{j=1}^{n} q_j(t)$ respectively, we assume that the result holds for all pairs of vectors with total remaining processing time at most $\sum_{j=1}^{n} p_j(t) - 1$ and $\sum_{j=1}^{n} q_j(t) - 1$, respectively. The base case can be easily checked by considering the two vectors $(1, 0, \ldots, 0)$ and $(1, 0, \ldots, 0)$.

If LRPT is applied for one time unit on $\overline{p}(t)$ and $\overline{q}(t)$ respectively, then the vectors of remaining processing times at time $t + 1$ are $\overline{p}(t + 1)$ and $\overline{q}(t + 1)$ respectively. It is clear that

$$\sum_{j=1}^{n} p_{(j)}(t + 1) \leq \sum_{j=1}^{n} p_{(j)}(t) - 1,$$
$$\sum_{j=1}^{n} q_{(j)}(t + 1) \leq \sum_{j=1}^{n} q_{(j)}(t) - 1.$$

It can be shown that if $\overline{p}(t) \geq_m \overline{q}(t)$, then $\overline{p}(t+1) \geq_m \overline{q}(t+1)$. Notice that LRPT results in a larger makespan at time $t + 1$ due to the induction hypothesis, and so it also results in a larger makespan at time t.

If there are fewer than m jobs remaining to be processed, then it is clear that the lemma holds. □

Theorem 6.21

LRPT is optimal for $(P_m \mid \text{prmp} \mid C_{\max})$ in discrete time.

Proof. The first step of the induction is shown as follows. Suppose no more than m jobs have processing times remaining and that these jobs all have only one unit of processing time left. Then clearly LRPT is optimal in this case.

Assume that LRPT is optimal for any vector $\overline{p}(t)$ such that

$$\sum_{j=1}^{n} p_{(j)}(t) \leq N - 1.$$

Now, consider a vector $\overline{p}(t)$ with

$$\sum_{j=1}^{n} p_{(j)}(t) = N.$$

We again proceed by induction on the total amount of remaining processing. Here, we assume towards a contradiction that LRPT is not optimal for $\overline{p}(t)$. Since LRPT is not optimal, then there must be some other rule, say R, that is optimal. Note that R does not act according to LRPT at time t, but from time $t + 1$ onwards, it must act according to LRPT due to the induction hypothesis. We compare applying LRPT at time t to $\overline{p}(t)$ as opposed to applying R at time t to the same vector $\overline{p}(t)$. Let $\overline{p}(t+1)$ and $\overline{p}'(t+1)$ denote the vectors of remaining processing times at time $t+1$ after applying LRPT and R. It is clear that $\overline{p}'(t+1) \geq_m \overline{p}(t+1)$. From Lemma 6.20, it follows that the makespan under R is larger than the makespan under LRPT. This is a contradiction, which completes the proof. □

Notice that when we multiply all processing times by a large integer K, the problem intrinsically does not change, as the relative lengths of the processing times remains the same. The optimal policy is again LRPT. But now, there are many more preemptions (at integral time units). Multiplying all processing times by K has the effect that the time slots become smaller relative to the processing times and the decision maker is allowed to preempt after shorter intervals. Letting $K \to \infty$ shows that LRPT is optimal in continuous time as well.

6.3 Dynamic Programming and PTAS for Makespan

We now return to the **NP**-hard problem $(P \mid\mid C_{\max})$, where the number of machines is *not* fixed. We first focus on a special case where the processing times p_j take on k distinct values. We use dynamic programming algorithm to solve this special case in $n^{O(k)}$ time. Assume that we are given a target schedule length T.

Suppose that the k distinct values are $b_1, \ldots, b_k$. Let $n_i = |\{j \in [n] : p_j = b_i\}|$ be the total number of jobs with processing time b_i. The key observation is that a subset of jobs can be described with a k-dimensional vector $\mathbf{x} = (x_1, \ldots, x_k)$ where x_i is the number of jobs of processing time b_i.

We define $M(\mathbf{x})$ to be the minimum number of machines needed to schedule all jobs in $\mathbf{x}$ with makespan $\leq T$. Next, we define Q to be the set of vectors $\mathbf{a} = (a_1, \ldots, a_k)$ that can be scheduled on **one** machine with makespan $\leq T$. That is, we have

$$Q = \left\{ \mathbf{a} = (a_1, \ldots, a_k) : \sum_{i=1}^{k} a_i b_i \leq T \right\}.$$

We will exclude the zero vector $\mathbf{0}$ from Q, as it is not useful to us. We now state the dynamic programming algorithm for our special case.

Algorithm 6.22: Dynamic programming for special case of $(P \mid\mid C_{\max})$

Initially, set $M(\mathbf{a}) = 1$ if $\mathbf{a} \in Q$, and set $M(\mathbf{0}) = 0$. We wish to compute $M(n_1, \ldots, n_k)$. We can do this via a k-dimensional table with $n_1 \times \cdots \times n_k$ entries via the recursive relation

$$M(x_1, \ldots, x_k) = 1 + \min\{M(x_1 - a_1, \ldots, x_k - a_k) : \mathbf{a} \in Q\}.$$

We see that there are at most n^k entries in the table, and that the computation of each entry relies on $O(n^k)$ other entries. Therefore, the total computation time of Algorithm 6.22 is $n^{O(k)}$.

It remains to handle the assumption that we know the target schedule length T. The easiest way to do this is to perform a binary search on all possible values of T. We will go into more detail on this later when we discuss the PTAS for the more general case of this problem.

Example 6.23

Suppose there are two distinct values of jobs, namely $b_1 = 1$ and $b_2 = 10$. Moreover, suppose that $n_1 = n_2 = 5$, and our target schedule length is $T = 12$. First, observe that

$$Q = \{(1,0), (2,0), (3,0), (4,0), (5,0), (0,1), (1,1), (2,1)\}$$

is the set of schedules that have makespan ≤ 12 on one machine. Then $M(0,0) = 0$ and $M(\mathbf{a}) = 1$ for all $\mathbf{a} \in Q$. We then use the relation $M(\mathbf{x}) = 1 + \min\{M(\mathbf{x} - \mathbf{a}) : \mathbf{a} \in Q\}$ to compute the rest of the table, as follows.

$M(x_1, x_2)$	$x_2 = 0$	$x_2 = 1$	$x_2 = 2$	$x_2 = 3$	$x_2 = 4$	$x_2 = 5$
$x_1 = 0$	0	1	2	3	4	5
$x_1 = 1$	1	1	2	3	4	5
$x_1 = 2$	1	1	2	3	4	5
$x_1 = 3$	1	2	2	3	4	5
$x_1 = 4$	1	2	2	3	4	5
$x_1 = 5$	1	2	3	3	4	5

We now discuss the general case $(P \mid\mid C_{\max})$ without the assumption where the processing times p_j take on k distinct values. In this case, we will focus mainly on the larger jobs. We will round and scale these jobs so that there is at most a constant number of sizes of large jobs, and apply the dynamic programming algorithm (Algorithm 6.22) to these rounded jobs. We then finish up by scheduling the small jobs greedily via the list scheduling algorithm (Algorithm 6.1).

We consider a **relaxed decision procedure (RDP)** for the problem, which takes as inputs an error parameter $\varepsilon > 0$ and a target makespan T. It returns "no" if the optimal makespan is $> T$, and returns "yes" if the optimal makespan is $\leq (1+\varepsilon)T$ along with such a schedule. Using this RDP, we obtain a $(1+\varepsilon)$-approximation algorithm for $(P \mid\mid C_{\max})$, which also takes $\varepsilon > 0$ and T as input.

ALGORITHM 6.24: PTAS FOR $(P \mid\mid C_{\max})$

(1) Partition the jobs into small jobs $\{j : p_j \leq \varepsilon T\}$ and large jobs $\{j : p_j > \varepsilon T\}$.

(2) Call the RDP on the large jobs. Note that this can be done by running the dynamic programming algorithm (Algorithm 6.22) after we perform rounding and scaling to the large jobs.

(3) If the RDP returns "no", then output "no schedule of length $\leq T$ exists". If the RDP returns "yes" and a schedule $\hat{S}$, apply list scheduling (Algorithm 6.1) to place the small jobs into $\hat{S}$. Call this schedule S^{new}.

(4) If the makespan of S^{new} is $> (1+\varepsilon)T$, then output "no schedule of length $\leq T$ exists". Otherwise, return "yes" and the schedule S^{new}.

Note that this algorithm can terminate at three points: Step 3 when the RDP returns "no", and Step 4 depending on the makespan of S^{new}. To prove correctness, it suffices to consider when the makespan of S^{new} is $> (1+\varepsilon)T$; it is clear that the other outputs are correct. Note that the last job to complete in S^{new} must be a small job, because if the last job were a large job, then the algorithm would have terminated at Step 3. Call this last job ℓ, and note that $C_\ell > (1+\varepsilon)T$ by assumption. Then the starting time of job ℓ is $S_\ell \geq C_\ell - \varepsilon T > T$, meaning that all machines are busy at time S_ℓ. Then the optimal makespan is $\geq S_\ell > T$, so the output is correct.

We now say a few words on how we can obtain the target makespan T. Let $\alpha = \max(p_{\max}, \frac{1}{m}\sum_{j=1}^{n} p_j)$. One can argue similar to Lemma 6.15 that α is a lower bound on the optimal makespan OPT for $(P \mid\mid C_{\max})$. This tells us that

$$\alpha - 1 < \text{OPT} < 2\alpha,$$

where the upper bound is obtained by our list scheduling result, namely Theorem 6.2. This means that we can start with the search space $(\alpha - 1, 2\alpha)$, and fix $T = \frac{3}{2}\alpha$. Given $\varepsilon > 0$, set $\hat{\varepsilon} = \varepsilon/3$. Then, call RDP with inputs $\hat{\varepsilon}$ and T. If the output is "no", then we know that $\text{OPT} \in (\frac{3}{2}\alpha, 2\alpha)$. On the other hand, if the output is "yes", we obtain a schedule with makespan $\leq (1+\hat{\varepsilon})\frac{3}{2}\alpha$. We can continue this binary search until the length of the search interval is $\leq \hat{\varepsilon}\alpha$.

6.4 Total Completion Time without Preemptions

Consider m machines in parallel and n jobs. Assume that the jobs are ordered such that $p_1 \geq \cdots \geq p_n$. Recall from Theorem 2.2 that the SPT rule is optimal for $(1 \mid\mid \sum C_j)$. This result can be proved in a different way fairly easily.

Let $p_{(j)}$ denote the processing time of the job in the j-th position in the sequence. Then the total completion time is can be expressed as

$$\sum C_j = np_{(1)} + (n-1)p_{(2)} + \cdots + 2p_{(n-1)} + p_{(n)}.$$

There are n coefficients $n, n-1, \ldots, 1$ to be assigned to n different processing times. In order to minimize the above sum, it follows that the smallest processing time p_n should be assigned to the largest coefficient n, the second smallest processing time p_{n-1} should be assigned to the second largest coefficient $n-1$, and so on. This implies that SPT is optimal. We can extend this type of argument to the parallel machine setting.

Theorem 6.25

The SPT rule is optimal for $(P_m \mid\mid \sum C_j)$.

Proof. In the case of parallel machines, there are nm coefficients that processing times can be assigned to. In particular, there are m copies of each of $n, n-1, \ldots, 1$. The processing times have to be assigned to a subset of these coefficients in order to minimize the sum of the products.

We may assume without loss of generality that n/m is an integer because if not, then we can add a number of dummy jobs of processing time 0. This does not change the problem because these jobs can be instantaneously processed at time 0 and would not contribute to the objective function.

In a similar manner to above, we see that the set of m longest processing times have to be assigned to the m copies of 1, and so on until the m shortest processing times are assigned to the m copies of n. We claim that this class of schedules includes SPT. According to the SPT rule, the smallest job has to go on machine 1 at time 0, the second smallest job goes on machine 2 at time 0, and so on. Next, the $(m+1)$-th smallest job follows the smallest job on machine 1, the $(m+2)$-th smallest job follows the second smallest job on machine 2, and so on. We see that this assignment satisfies the properties outlined above, so SPT is optimal. ☐

From the proof of Theorem 6.25, we see that the SPT is not the only schedule that is optimal. Many more schedules also minimize the total completion time. It turns out that it is fairly easy to characterize the class of schedules that minimizes the total completion time; we will show this on Assignment 8.

Unfortunately, we cannot generalize Theorem 2.3, which states that the WSPT rule is optimal for $(1 \mid\mid \sum w_j C_j)$, to the parallel machine setting. This is shown in the following example.

Example 6.26

Consider two machines and three jobs as below.

j	1	2	3
p_j	1	1	3
w_j	1	1	3

Scheduling jobs 1 and 2 at time 0 and job 3 at time 1 results in a total weighted completion time of 14, while scheduling job 3 at time 0 and jobs 1 and 2 on the other machine yields a total weighted completion time of 12. Note that with this set of data, any schedule is WSPT. However, changing the weights of jobs 1 and 2 to be $1 - \varepsilon$ shows that WSPT does not necessarily yield an optimal schedule.

In the literature, it has been shown that the WSPT heuristic is nonetheless a good one for the total weighted completion time on parallel machines. A worst case analysis of this heuristic yields the bound

$$\frac{\sum w_j C_j(\text{WSPT})}{\sum w_j C_j(\text{OPT})} < \frac{1}{2}(1+\sqrt{2}).$$

We now consider the model $(R_m \mid\mid \sum C_j)$. Recall from Section 1.2 that the machines in the R_m environment are entirely unrelated. For example, machine 1 may be able to process job 1 in a short time and may need a long time for job 2, while machine 2 may be able to process job 2 in a short time while taking a long time for job 1. Note that identical parallel machines where jobs j are restricted to machine sets M_j is also a special case of this. Indeed, the processing time of a job j on a machine that is not part of M_j can be considered very long, making it impossible to process the job on such a machine.

We can formulate the $(R_m \mid\mid \sum C_j)$ problem as an integer program with a special structure that makes it possible to solve the problem in polynomial time. Recall that if job j is processed on machine i and there are $k-1$ jobs following job j on this machine i, then job j contributes kp_{ij} to the value of the objective function. Let $x_{ikj} = 1$ if job j is scheduled as the k-th to last job on machine i, and $x_{ikj} = 0$ otherwise. Then the integer program is as follows:

$$\begin{aligned}
\min \quad & \sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{n} kp_{ij}x_{ikj} & \\
\text{s.t.} \quad & \sum_{i=1}^{m}\sum_{k=1}^{n} x_{ikj} = 1, & j \in [n] \\
& \sum_{j=1}^{n} x_{ikj} \leq 1, & i \in [m],\ k \in [n] \\
& x_{ikj} \in \{0,1\}, & i \in [m],\ k \in [n],\ j \in [n].
\end{aligned}$$

The constraints make sure that each job is scheduled exactly once, and each position on each machine is taken by at most one job. Note that the processing times only appear in the objective function.

This is a so-called weighted bipartite matching problem, which has n jobs on one side and nm positions on the other side (each machine can process at most n jobs). If job j is matched with (assigned to) position ik, then there is a cost kp_{ij}. The objective is to determine the matching in this bipartite graph with a minimum cost. It is known from the theory of network flows that the integrality constraints on the x_{ikj} may be replaced by non-negativity constraints without changing the feasible set. This weighted bipartite matching problem then reduces to a regular linear program for which there exist polynomial time algorithms. Note that the optimal schedule does not have to be a non-delay schedule.

7 Flow Shops

In many cases, each job has to undergo a series of operations. Often, these operations have to be done on all jobs in the same order, implying that the jobs have to follow the same route. The machines are then assumed to be set up in series and the environment is referred to as a flow shop. We saw an example of this in Example 1.3; other real life examples are manufacturing and assembly facilities.

We will mainly consider the makespan objective. The makespan objective is of considerable practical interest as its minimization is to a certain extent equivalent to the maximization of the utilization of the machines. The models, however, tend to be of such complexity that makespan results are already relatively hard to obtain. Total completion time and due date related objectives tend to be even harder.

7.1 Permutation Flow Shops

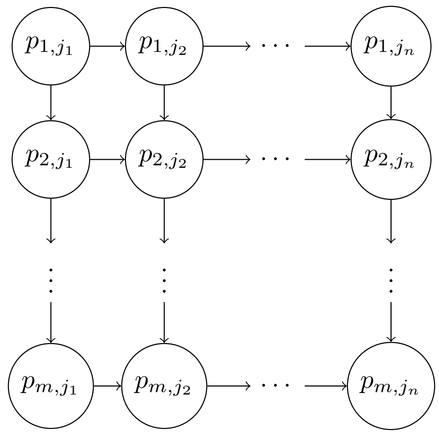
When searching for an optimal schedule for $(F_m \mid\mid C_{\max})$, a natural question to ask is whether it suffices merely to determine a permutation in which the jobs traverse the entire system. It may be possible for one job to "pass" another while they are waiting in queue for a machine that is busy. This means that the machines might not be operating according to the "first come first served" principle, and that the sequence in which the jobs go through the machines may change from one machine to another. Changing the sequence of the jobs waiting in a queue between two machines may at times result in a smaller makespan. However, it can be shown that there always exists an optimal schedule without job sequence changes between the first two machines and the last two machines. This implies that there are optimal schedules for $(F_2 \mid\mid C_{\max})$ and $(F_3 \mid\mid C_{\max})$ that do not require sequence changes between machines. There are examples of flow shops with four machines in which the optimal schedule does require a job sequence change between the second and third machine.

Finding an optimal schedule when sequence changes are allowed is significantly harder than finding an optimal schedule when sequence changes are not allowed. Flow shops that do not allow sequence changes between machines are called **permutation flow shops**. In these flow shops, the same sequence, or permutation, of jobs is maintained throughout. We denote this problem by $(F_m \mid \text{prmu} \mid C_{\max})$. Many results we will look at concern permutation flow shops.

Given a permutation schedule $j_1, \ldots, j_n$ for a flow shop with m machines, the completion time of a job j_k at machine i can be computed through a set of recursive equations. In particular, we have

$$
\begin{aligned}
C_{i,j_1} &= \sum_{\ell=1}^{i} p_{\ell,j_1}, && i = 1, \ldots, m,\\
C_{1,j_k} &= \sum_{\ell=1}^{k} p_{1,j_\ell}, && k = 1, \ldots, n,\\
C_{i,j_k} &= \max\{C_{i-1,j_k}, C_{i,j_{k-1}}\} + p_{i,j_k}, && i = 2, \ldots, m,\ k = 2, \ldots, n.
\end{aligned}
$$

Under the permutation schedule $j_1, \ldots, j_n$, we can also compute the makespan by determining a **critical path** in a directed graph corresponding to the schedule. For each operation, say the processing of job j_k on machine i, there is a node (i, j_k) with a weight equal to the processing time of job j_k on machine i. Each node (i, j_k) for $i = 1, \ldots, m-1$ and $k = 1, \ldots, n-1$ has edges going out to nodes $(i+1, j_k)$ and (i, j_{k+1}). Nodes corresponding to machine m only have one outgoing edge, as do nodes corresponding to job j_n. The node (m, j_n) has no outgoing edges. The total weight of the maximum weight path from node $(1, j_1)$ to node (m, j_n) is the makespan under the permutation schedule $j_1, \ldots, j_n$. We illustrate this directed graph below.

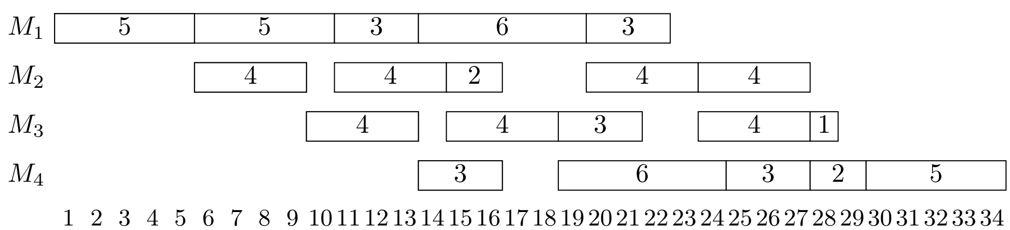


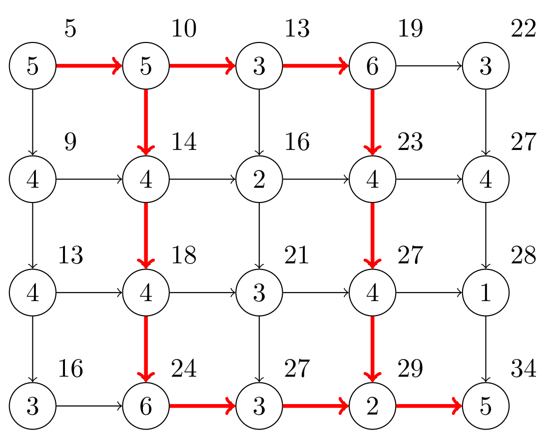
Example 7.1

Consider $n = 5$ jobs on $m = 4$ machines with the processing times below.

Jobs	j_1	j_2	j_3	j_4	j_5
p_{1,j_k}	5	5	3	6	3
p_{2,j_k}	4	4	2	4	4
p_{3,j_k}	4	4	3	4	1
p_{4,j_k}	3	6	3	2	5

Under the sequence j_1, j_2, j_3, j_4, j_5, the corresponding graph and Gantt chart are given below. In the top right corner of each node in the directed graph, we put the completion time of that node. It follows that the makespan is $C_{\max} = 34$, and it is determined by the two critical paths given in red.





7.2 Johnson's Rule on Two Machines

We now consider the $(F_2 \mid\mid C_{\max})$ problem. Note that there is no permutation schedule restriction here, as we noted that $(F_2 \mid\mid C_{\max})$ has an optimal schedule that is a permutation schedule. There are two machines and n jobs, with processing time $a_j = p_{1j}$ on machine 1 and processing time $b_j = p_{2j}$ on machine 2. This was one of the first problems to be analyzed in the early days of operations research and led to a classical paper in scheduling theory by S. M. Johnson. The rule that minimizes the makespan is commonly referred to as Johnson's rule.

Algorithm 7.2: Johnson's rule

(1) Set $i = 1$, $\ell = n$, and $\hat{J} = [n]$.

(2) If $\hat{J} = \varnothing$, then stop. Otherwise, set $\mu = \min\{a_j, b_j : j \in \hat{J}\}$.

(3) (a) If $\mu = a_j$, then place job j in position i of the sequence. Increase i by 1 and delete j from $\hat{J}$. Go back to Step 2.

(b) If $\mu = b_j$, then place job j in position ℓ of the sequence. Decrease ℓ by 1 and delete j from $\hat{J}$. Go back to Step 2.

Example 7.3

Consider the following $n = 5$ jobs on two machines.

j	1	2	3	4	5
a_j	5	2	3	6	3
b_j	1	4	3	4	4

Then at iteration 1, we have $\mu = b_1 = 1$, so we put job 1 in position 5. At iteration 2, we have $\mu = a_2 = 2$, so we put job 2 in position 1. At iteration 3, we arbitrarily pick $\mu = a_3 = 3$, so job 3 goes in position 2. At iteration 4, we get $\mu = a_5 = 3$, so job 5 goes in position 3. Finally, job 4 goes in position 4, so we obtain the final sequence $2, 3, 5, 4, 1$.

Note that Johnson's rule is equivalent to the following algorithm. Partition the jobs into two sets: S_1 contains all jobs j with $a_j \le b_j$, and S_2 contains all jobs j with $a_j > b_j$. Place the jobs in S_1 first in SPT order, and the jobs in S_2 last in LPT order. In this sense, this is also called the SPT(1)-LPT(2) rule.

Let us now show that Johnson's rule is indeed optimal for $(F_2 \mid\mid C_{\max})$. We first prove a key lemma.

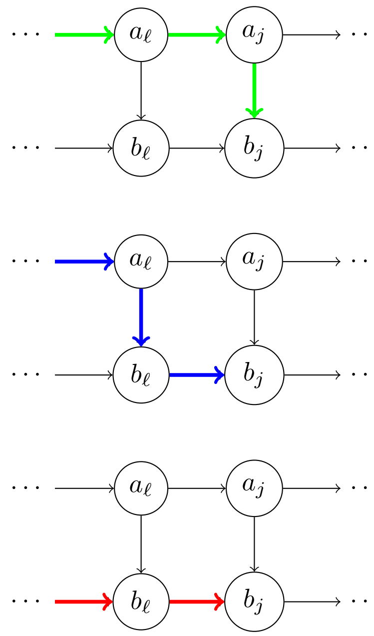
Lemma 7.4

(1) If $\mu = a_j = \min\{a_\ell, b_\ell : \ell \in [n]\}$ in the first iteration of Johnson's rule, then there is an optimal schedule that places job j first.

(2) If $\mu = b_j = \min\{a_\ell, b_\ell : \ell \in [n]\}$ in the first iteration of Johnson's rule, then there is an optimal schedule that places job j last.

Proof. We will only prove (1), as the proof of (2) is analogous. Suppose that we have an optimal schedule S^* that does not start with job j. Then there must be a job ℓ immediately preceding j in S. Let t^1_ℓ denote the starting time of ℓ on machine 1, and let t^2_ℓ denote the starting time of ℓ on machine 2. We interchange jobs j and ℓ to get a new schedule S'.

We claim that $C'_\ell \leq C_j$, where C'_ℓ is the completion time of job ℓ in S' and C_j is the completion time of job j in S. Observe that in the directed graph representation under S, there are three possible paths we could take, illustrated below.



Note that the green path is dominated by the blue path because $a_j \leq b_\ell$ by assumption. This shows that

$$C_j = \max\{t_\ell^2 + b_\ell + b_j, t_\ell^1 + a_\ell + b_\ell + b_j\}.$$

Constructing a similar diagram for S' gives us

$$C'_\ell = \max\{t_\ell^1 + a_j + a_\ell + b_\ell, t_\ell^1 + a_j + b_j + b_\ell, t_\ell^2 + b_j + b_\ell\}.$$

Noting that $a_j = \min\{a_j, b_j : j \in [n]\}$, it is easy to check that $C_j \geq C'_\ell$ by comparing the terms above. This means that S' is also an optimal schedule which has job j before job ℓ. We can continue to interchange jobs and repeat our argument until job j is first. $\square$

Theorem 7.5

Johnson's rule gives an optimal schedule for $(F_2 \mid\mid C_{\max})$.

Proof. We give a sketch of the proof. We proceed by induction on the number of iterations. The earlier iterations would have constructed a partial schedule. Apply Lemma 7.4 to the set of jobs that remain to be scheduled after $i - 1$ iterations. Note that Lemma 7.4 only sees the two jobs j minimizing the value of μ and the job ℓ immediately preceding job j; it "ignores" all other jobs. $\square$

As one might imagine, a schedule obtained from Johnson's rule is by no means the only schedule that is optimal for $(F_2 \mid\mid C_{\max})$. The class of optimal schedules appears to be hard to characterize and data dependent.

7.3 Mixed Integer Program Formulation for Permutation Schedules

Unfortunately, the schedule structure from Johnson's rule cannot be generalized to characterize optimal schedules for flow shops with more than two machines. Indeed, it turns out that even the three machine case is strongly **NP**-hard.

Theorem 7.6

The problem $(F_3 \mid\mid C_{\max})$ is strongly **NP**-hard.

Proof. We prove this via a reduction from 3-Partition. Given $a_1, \ldots, a_{3t}, b \in \mathbb{Z}^+$ under the usual assumptions, let the number of jobs be $n = 4t+1$, and let the processing times be

$$\begin{aligned}
p_{10} &= 0, & p_{20} &= b, & p_{30} &= 2b, & & \\
p_{1j} &= 2b, & p_{2j} &= b, & p_{3j} &= 2b, & & j = 1, \ldots, t-1, \\
p_{1t} &= 2b, & p_{2t} &= b, & p_{3t} &= 3b, & & \\
p_{1,t+j} &= 0, & p_{2,t+j} &= a_j, & p_{3,t+j} &= 0, & & j = 1, \ldots, 3t.
\end{aligned}$$

Let $z = (2t+1)b$. A makespan of value z can be obtained if the first $t+1$ jobs are scheduled according to the sequence $0, 1, \ldots, t$. These $t+1$ jobs then form a framework, leaving t gaps on machine 2. Then jobs $t+1, \ldots, 4t$ have to be partitioned into t sets of three jobs each, and these t sets have to be scheduled between the first $t+1$ jobs. Thus, a makespan of z can be obtained if and only if the 3-Partition instance has a solution. □

The problem $(F_m \mid \text{prmu} \mid C_{\max})$ can be formulated as a mixed integer program (MIP). Note that the fact that the problem can be formulated in this way does not imply that the problem is **NP**-hard. It could be that the MIP has a special structure that allows for a polynomial time algorithm, such as the one we saw for $(R_m \mid\mid \sum C_j)$ in Section 6.4. However, Theorem 7.6 shows that this is not the case.

We first define some variables that we need.

- Let x_{jk} be a binary variable equal to 1 if job j is the k-th job in the sequence, and 0 otherwise.
- The auxiliary variable I_{ik} denotes the idle time on machine i between the processing of the jobs in the k-th and $(k+1)$-th positions.
- The auxiliary variable W_{ik} denotes the waiting time of the job in the k-th position between machines i and $i+1$.

Naturally, there is a strong relationship between the variables W_{ik} and the variables I_{ik}. For example, if $I_{ik} > 0$, then $W_{i-1,k+1} = 0$. Formally, this relationship can be established by considering the difference between the time the job in the $(k+1)$-th position starts on machine $i+1$ and the time the job in the k-th position completes its processing on machine i. If Δ_{ik} denotes this difference and $p_{i(k)}$ is the processing time of the job in the k-th position on machine i, then

$$\Delta_{ik} = I_{ik} + p_{i(k+1)} + W_{i,k+1} = W_{ik} + p_{i+1(k)} + I_{i+1,k}.$$

Note that minimizing the makespan is equivalent to minimizing the total idle time on the last machine, namely machine m. This idle time is equal to

$$\sum_{i=1}^{m-1} p_{i(1)} + \sum_{j=1}^{n-1} I_{mj},$$

which is the idle time that must occur before the job in the first position reaches the last machine and the sum of the idle times between the jobs on the last machine. Moreover, we have the identity

$$p_{i(k)} = \sum_{j=1}^{n} x_{jk} p_{ij}.$$

Putting everything together, we can now formulate the MIP as follows:

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{m-1}\sum_{j=1}^{n} x_{j1}p_{ij} + \sum_{j=1}^{n-1} I_{mj} & \\
\text{s.t.} \quad & \sum_{j=1}^{n} x_{jk} = 1, & k \in [n] \\
& \sum_{k=1}^{n} x_{jk} = 1, & j \in [n] \\
& I_{ik} + \sum_{j=1}^{n} x_{j,k+1}p_{ij} + W_{i,k+1} - W_{ik} - \sum_{j=1}^{n} x_{jk}p_{i+1,j} - I_{i+1,k} = 0, & k \in [n-1],\ i \in [m-1] \\
& W_{i1} = 0, & i \in [m-1] \\
& I_{1k} = 0, & k \in [n-1].
\end{aligned}
$$

The first set of constraints specifies that exactly one job has to be assigned to position k for any k. The second set of constraints specifies that job j has to be assigned to exactly one position. The third set of constraints relate the decision variables x_{jk} to the physical constraints. These physical constraints enforce the necessary relationships between the idle time variables and the waiting time variables. Thus, the problem of minimizing the makespan in an m machine permutation flow shop is formulated as an MIP. The only integer variables are the binary decision variables x_{jk}. The idle time and waiting time variables are non-negative continuous variables.

8 Open Shops

Recall that in a flow shop, each job must follow the same route. In the next section, we will also discuss job shops, where each job has its own predetermined fixed route. However, in practice, it is often the case that the route of the job is immaterial and up to the schedule to decide. The routes of the job are *open* in this sense, and this model is called an open shop. But we still have the usual restrictions that each machine can only handle one job at a time, and two operations of the same job cannot be run at the same time.

8.1 Makespan without Preemptions

We first consider the $(O_2 \mid\mid C_{\max})$ problem where there are two machines. A job j can be processed first on machine 1 or vice versa; the decision maker can determine the routes. It is clear that

$$C_{\max} \geq \max\left(\sum_{j=1}^{n} p_{1j}, \sum_{j=1}^{n} p_{2j}\right),$$

since the makespan cannot be less than the workload on either machine.

One might expect that this is an equality, and this is true in most situations. However, to get equality, we must also consider $\max_{j\in[n]} p_{1j} + p_{2j}$ because of the restriction that two operations of the same job cannot be run at the same time. Thus, the optimal makespan is

$$C_{\max} = \max\left(\max_{j\in[n]}(p_{1j} + p_{2j}), \sum_{j=1}^{n} p_{1j}, \sum_{j=1}^{n} p_{2j}\right).$$

For now, we only consider non-delay schedules. That is, if there is a job waiting for processing when a machine is free, then that machine is not allowed to remain idle. It follows that an idle period can occur on a machine if and only if one job remains to be processed on that machine and when that machine is available, this last job is still being processed on the other machine. Such an idle period *can* cause an unnecessary increase in the makespan. If this last job turns out to be the very last job to complete all its processing, then the idle period does cause an increase in the makespan. But if this last job, after having completed its processing on the machine that was idle, is *not* the very last job to leave the system, then the makespan is still equal to the maximum of the two workloads.

We now give an algorithm that solves $(O_2 \mid\mid C_{\max})$ instances in polynomial time. We may assume without loss of generality that the longest processing time among the $2n$ processing times belongs to operation $(1, k)$, because we can swap the two machines if needed. That is, we have $p_{ij} \leq p_{1k}$ for all $i = 1, 2$ and $j = 1, \ldots, n$.

Algorithm 8.1

If operation $(1, k)$ is the longest operation, then job k must be started at time 0 on machine 2. After job k has completed its processing on machine 2, its operation $(1, k)$ has the lowest possible priority with regard to processing on machine 1. Then the processing of operation $(1, k)$ will be postponed as much as possible; it can only be processed on machine 1 if no other job is available for processing on machine 1. (This can only happen either if it is the last operation to be done on machine 1, or if it is the second last operation and the last operation is not available because it is still being processed on machine 2.) The $2(n-1)$ operations of the remaining $n-1$ jobs can be processed on the two machines in any order. However, unforced idleness is not allowed.

Algorithm 8.1 is a more general form of the **Longest Alternate Processing Time first (LAPT)** rule, which says that whenever a machine is freed, start processing the jobs that have not yet received processing on either machine the one with the longest processing time on the *other* machine.

Theorem 8.2

Algorithm 8.1 results in an optimal schedule for $(O_2 \mid\mid C_{\max})$ with makespan

$$C_{\max} = \max\left(\max_{j\in[n]}(p_{1j}+p_{2j}), \sum_{j=1}^{n} p_{1j}, \sum_{j=1}^{n} p_{2j}\right).$$

Proof. If the resulting schedule has no idle period on either machine, then it is of course optimal. However, an idle period may occur either on machine 1 or on machine 2. We consider two cases.

Case 1. Suppose an idle period occurs on machine 2. If this is the case, then only one more operation needs processing on machine 2 but this operation still has to complete its processing on machine 1. Assume that this operation belongs to job ℓ. When job ℓ starts on machine 2, job k starts on machine 1 and $p_{1k} > p_{2\ell}$. Hence, the makespan is determined by the completion of job k on machine 1 and no idle period has occurred on machine 1. This means the schedule is optimal.

Case 2. Suppose an idle period occurs on machine 1. An idle period on machine 1 can occur only when machine 1 is freed after completing all its operations with the exception of operation $(1, k)$, and operation $(2, k)$ of job k is still being processed on machine 2 at that point. In this case, the makespan is equal to $p_{2k} + p_{1k}$ and the schedule is optimal. $\square$

The LAPT rule we described above can be considered to be a special case of a more general rule that can be applied to open shops with more than two machines. This rule is called the **Longest Total Remaining Processing on Other Machines first (LTRPOM)** rule. According to this rule, the processing required on the machine currently available does not affect the priority level of a job. However, this rule does not always result in an optimal schedule because the $(O_m \mid\mid C_{\max})$ problem is **NP**-hard when $m \geq 3$.

Theorem 8.3

The $(O_3 \mid\mid C_{\max})$ problem is weakly **NP**-hard.

Proof. We prove this using a reduction from Partition to $(O_3 \mid\mid C_{\max})$. Given $a_1, \ldots, a_t, b \in \mathbb{Z}^+$ such that $\sum_{j=1}^{t} a_j = 2b$, we construct $n = 3t + 1$ jobs. There are three jobs corresponding to each value a_j, given by

$$\begin{array}{llll} p_{1j} = a_j, & p_{2j} = 0, & p_{3j} = 0, & j = 1, \ldots, t, \\ p_{1j} = 0, & p_{2j} = a_j, & p_{3j} = 0, & j = t+1, \ldots, 2t, \\ p_{1j} = 0, & p_{2j} = 0, & p_{3j} = a_j, & j = 2t+1, \ldots, 3t. \end{array}$$

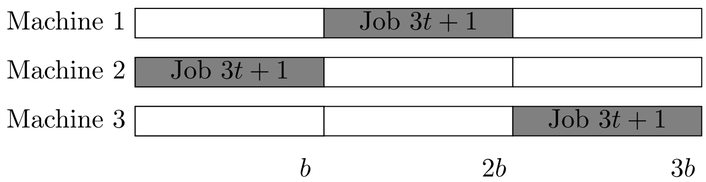
There is one more marker job with processing times

$$p_{1,3t+1} = p_{2,3t+1} = p_{3,3t+1} = b.$$

Let $z^* = 3b$. We claim that there is a schedule with makespan z^* if and the Partition instance has a solution. Suppose there is a schedule with makespan $z^* = 3b$. Assume without loss of generality that the marker job $3t + 1$ is scheduled on time interval $[b, 2b]$ on machine 1, because we could swap machines otherwise. Then the jobs processed at time interval $[0, b]$ on machine 1 correspond to a subset $S_1 \subseteq \{1, \ldots, t\}$ such that $\sum_{j\in S_1} a_j = b$, as do the jobs processed at time interval $[2b, 3b]$ on machine 2. Thus, the Partition instance has a solution.

Suppose that the Partition instance has a solution, namely a partition (S_1, S_2) of $\{1, \ldots, t\}$ such that $\sum_{j\in S_1} a_j = \sum_{j\in S_2} a_j = b$. Process job $3t + 1$ on machine 2 at time interval $[0, b]$, on machine 1 at time interval $[b, 2b]$, and on machine 3 at time interval $[2b, 3b]$. Then we could place all jobs in S_1 on machine 1 at time interval $[0, b]$ and all jobs in S_2 on machine 1 at time interval $[2b, 3b]$. This will extend naturally to a schedule with makespan $z^* = 3b$. $\square$

The schedule from the proof of Theorem 8.3 is described in the following Gantt chart.



Algorithm 8.1 is one of the few polynomial time algorithms for non-preemptive open shop problems. Most of the more general open shop models are **NP**-hard, such as $(O_2 \mid r_j \mid C_{\max})$. However, it turns out that the problem $(O_m \mid r_j, p_{ij} = 1 \mid C_{\max})$ can be solved in polynomial time. We discuss this further in Section 8.3.

8.2 Makespan with Preemptions

Preemptive open shop problems tend to be somewhat easier. In contrast to the $(O_m \mid\mid C_{\max})$ problem, we will see that the $(O_m \mid \text{prmp} \mid C_{\max})$ problem is solvable in polynomial time.

Note that the value of the makespan under Algorithm 8.1 (or LAPT) is a lower bound for the makespan with two machines, even when preemptions are allowed. It follows that these non-preemptive rules are also optimal for $(O_2 \mid \text{prmp} \mid C_{\max})$.

An easy lower bound for the makespan with $m \geq 3$ machines when preemptions are allowed is

$$C_{\max} \geq \max\left(\max_{j \in [n]} \sum_{i=1}^{m} p_{ij},\ \max_{i \in [m]} \sum_{j=1}^{n} p_{ij} \right).$$

That is, the makespan must be at least as large as the maximum workload on each of the m machines, and at least as large as the total amount of processing to be done on each of the n jobs. It turns out that it is not difficult to obtain a schedule whose makespan is equal to this lower bound. To see how this is done, we define

$$\mathrm{ML}_i = \sum_{j=1}^{n} p_{ij}$$

for each $i = 1, \ldots, m$ to be the load of machine i, and

$$\mathrm{JL}_j = \sum_{i=1}^{m} p_{ij}$$

for each $j = 1, \ldots, n$ to be the load of job j. We can view these as the sums of the rows and columns of the the $m \times n$ matrix $\mathbf{P}$ corresponding to the processing times p_{ij}. Let LB be the lower bound from above.

We call machine i **tight** if we have $\mathrm{ML}_i = \mathrm{LB}$. Similarly, we call job j **tight** if $\mathrm{JL}_j = \mathrm{LB}$. If machine i or job j are not tight, then they have **slack**. A set D of operations is called a **decrementing set** if it contains one entry for each tight job, one entry for each tight machine, and at most one entry for each slack row and slack column. Similar to the matrix $\mathbf{P}$ of processing times, this set D can be viewed as an $m \times n$ matrix consisting of entries from $\{0, 1\}$.

Theorem 8.4

A decrementing set always exists and can be computed in polynomial time.

We take this result for granted, but note that the proof is based on maximal cardinality matchings. We can use a decrementing set to construct a partial schedule of length Δ for some appropriately chosen Δ. We are now ready to state the algorithm for $(O_m \mid \text{prmp} \mid C_{\max})$, which, finds a schedule with makespan $C_{\max} = \mathrm{LB}$.

Algorithm 8.5

Repeat the following steps until all operations have been successfully scheduled.

(1) Find a decrementing set D.

(2) Find the maximum value of Δ such that

  (i) $\Delta \leq \min_{(i,j)\in D} p_{ij}$;

  (ii) $\Delta \leq \mathrm{LB} - \mathrm{ML}_i$ if machine i has slack and no operation in D;

  (iii) $\Delta \leq \mathrm{LB} - \mathrm{JL}_j$ if job j has slack and no operation in D.

(3) Schedule the operations in D for Δ time units in parallel.

(4) Update the values of p_{ij}, LB, ML_i, and JL_j.

Note that after each iteration, we are decreasing LB by Δ, and a slice of Δ time units is scheduled. This suggests that the algorithm terminates after $\leq$ LB iterations. But we can actually do a slightly better analysis than this: it terminates after $\leq nm + n + m$ iterations. To see why, observe that at each iteration, either

- an operation gets completely scheduled, or
- a slack machine or job becomes tight (and stays tight throughout the later iterations).

In particular, the algorithm is polynomial time and is correct.

Example 8.6

Consider 3 machines and 4 jobs with the processing times being the entries in the matrix

$$\mathbf{P} = \begin{bmatrix} 3 & 4 & 0 & 4 \\ 4 & 0 & 6 & 0 \\ 4 & 0 & 0 & 6 \end{bmatrix}.$$

We can easily see that LB $= 11$ by computing the sums of the rows and columns. In particular, observe that machine 1 and job 1 are tight, while all other machines and jobs are slack. One possible decrementing set comprises of the processing times $p_{12} = 4$, $p_{21} = 4$, and $p_{34} = 6$. We can choose $\Delta = 4$ in this case, and a partial schedule is constructed by scheduling job 2 on machine 1 for 4 time units, job 1 on machine 2 for 4 time units, and job 4 on machine 3 for 4 time units. The matrix is now

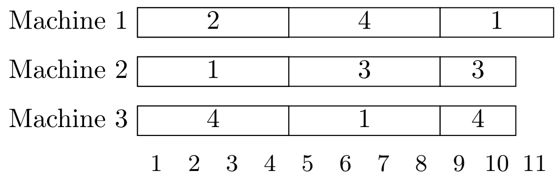
$$\mathbf{P} = \begin{bmatrix} 3 & 0 & 0 & 4 \\ 0 & 0 & 6 & 0 \\ 4 & 0 & 0 & 2 \end{bmatrix},$$

and LB $= 7$. This means that machine 1 and job 1 are again tight, with all other machines and jobs being slack. A decrementing set is obtained with the processing times $p_{14} = 4$, $p_{23} = 6$, and $p_{41} = 4$. Then for $\Delta = 4$, we can augment the partial schedule by assigning job 4 to machine 1 for 4 time units, job 3 to machine 2 for 4 time units, and job 1 to machine 3 for 4 time units. The matrix is updated to

$$\mathbf{P} = \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}.$$

At this point, there is an obvious way to assign the remaining processing times. We can put job 1 on machine 1 for 3 time units, job 3 on machine 2 for 2 time units, and job 4 on machine 3 for 2 time units.

The schedule from Example 8.6 is depicted in the following Gantt chart.



8.3 Maximum Lateness without Preemptions

The $(O_m \mid\mid L_{\max})$ problem is a generalization of the $(O_m \mid\mid C_{\max})$ problem, and is therefore at least as hard.

Theorem 8.7

The problem $(O_2 \mid\mid L_{\max})$ is strongly **NP**-hard.

Proof. We reduce 3-Partition to $(O_2 \mid\mid L_{\max})$. Given $a_1, \ldots, a_{3t}, b \in \mathbb{Z}^+$ under the usual assumptions, we construct $n = 4t$ jobs with processing times

$$\begin{array}{llll} p_{1j} = 0, & p_{2j} = a_j, & d_j = 3tb, & j = 1, \ldots, 3t, \\ p_{1j} = 0, & p_{2j} = 2b, & d_j = 2b, & j = 3t+1, \\ p_{1j} = 3b, & p_{2j} = 2b, & d_j = (3(j-3t)-1)b, & j = 3t+2, \ldots, 4t. \end{array}$$

In particular, jobs $3t+1, \ldots, 4t$ are marker jobs, with the last job having due date $(3t-1)b$. Then there exists a schedule with $L_{\max} \leq 0$ if and only if jobs $1, \ldots, 3t$ can be divided into t groups, each containing three jobs and requiring b units of processing time on machine 2. This is equivalent to the 3-Partition instance having a solution. $\square$

We can show that $(O_2 \mid\mid L_{\max})$ is equivalent to the $(O_2 \mid r_j \mid C_{\max})$ problem. Indeed, consider the $(O_2 \mid\mid L_{\max})$ problem with deadlines $\overline{d_j}$ rather than due dates d_j. Let

$$\overline{d_{\max}} = \max_{j \in [n]} \overline{d_j}.$$

Applying a time reversal to $(O_2 \mid\mid L_{\max})$, we see that finding a feasible schedule with $L_{\max} = 0$ is now equivalent to finding a schedule for $(O_2 \mid r_j \mid C_{\max})$ with release dates

$$r_j = \overline{d_{\max}} - \overline{d_j}$$

and a makespan that is less than $\overline{d_{\max}}$. Thus, the $(O_2 \mid r_j \mid C_{\max})$ problem is also strongly **NP**-hard.

Now, we consider the special case $(O_m \mid r_j, p_{ij} = 1 \mid L_{\max})$ that we mentioned at the end of Section 8.1. The fact that all the processing times are unit makes the problem considerably easier. The polynomial time solution procedure consists of three phases.

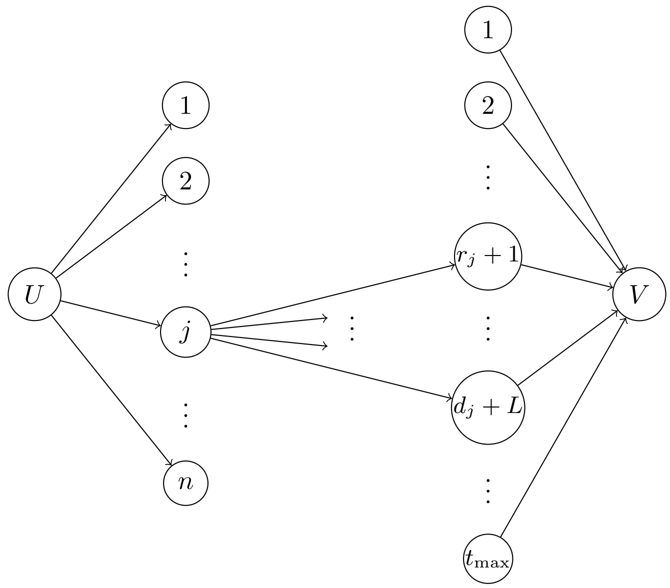
- **Phase 1:** Parametrizing and a binary search.
- **Phase 2:** Solving a network flow problem.
- **Phase 3:** Colouring a bipartite graph.

At Phase 1, we let L be a free parameter and assume that each job has a deadline $d_j + L$. The objective is to find a schedule in which each job is completed by its deadline, ensuring that $L_{\max} \leq L$. We set

$$t_{\max} = \max(d_1, \ldots, d_n) + L.$$

That is, no job should receive any processing after time $t_{\max}$.

Phase 2 focuses on the following network flow problem. There is a source node U that has n arcs going to nodes $1, \dots, n$, where node j corresponds to job j. The arc from the source node U to node j has capacity m, which is equal to the number of machines and the number of operations on each job. There is a second set of $t_{\max}$ nodes, with each node corresponding to one time unit. A node t with $t = 1, \dots, t_{\max}$ represents the time slot $[t-1, t]$. A node j has arcs going to nodes $r_j + 1, \dots, d_j + L$, and these arcs have unit capacity. Finally, each node in the set of $t_{\max}$ nodes has an arc of capacity m going to the sink node V. This capacity limit ensures that no more than m operations are processed in any given time period. The solution of this network flow problem indicates which time slots the m operations of job j need to be processed in.



One can also consider this network flow problem as a transportation model with

$$
\begin{aligned}
\min \quad & 0 \\
\text{s.t.} \quad & \sum_{t=r_j+1}^{d_j+L} x_{jt} = m, && j = 1, \dots, n, \\
& \sum_{j=1}^{n} x_{jt} \le m, && t = 1, \dots, t_{\max}, \\
& x_{jt} \le 1, && j = 1, \dots, n,\ t = 1, \dots, t_{\max}, \\
& x_{jt} \ge 0, && j = 1, \dots, n,\ t = 1, \dots, t_{\max}.
\end{aligned}
$$

The first set of constraints ensures that for each job, all operations are run and are timely. The second set of constraints ensures that each time unit can only process at most m jobs, corresponding to the number of machines. Note that the objective value is arbitrary here because all we need is a feasible solution. We recall that the unimodularity property ensures that an optimal solution is integer, and that such an LP can be solved in polynomial time.

However, the solution to the network flow problem cannot be immediately translated into a feasible schedule for the open shop, because in the network flow formulation, no distinction is made between the different machines. But we can transform the assignment of operations to time slots prescribed by the network flow solution into a feasible schedule in such a way that every job j is processed on a different machine at each time unit.

Phase 3 of the algorithm generates a feasible schedule. From the solution of the network flow problem, we can construct a bipartite graph with two sets N_1 and N_2 of nodes. The set N_1 has n nodes corresponding to the jobs, and N_2 has $t_{\max}$ nodes corresponding to the time slots. A node in N_1 is connected to the m nodes in N_2 corresponding to the time slots in which its operations are supposed to be processed, prescribed by the solution from Phase 2. So each node in N_1 is connected to exactly m nodes in N_2, while each node in N_2 is connected to at most m nodes in N_1. A result in graph theory tells us that if each node in a bipartite graph has at most m arcs, then the arcs can be coloured with m different colours in such a way that no node has two arcs of the same colour. Each colour then corresponds to a machine.

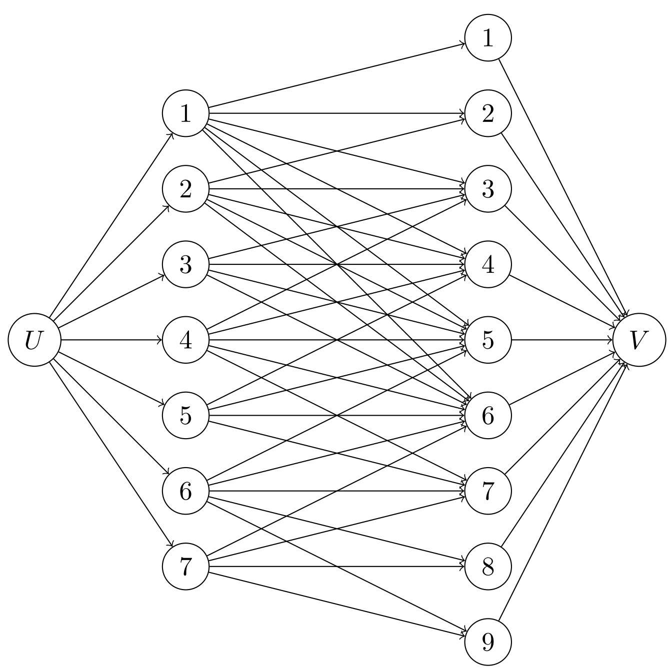
The colouring algorithm is fairly straightforward and was covered in MATH 239. Find a matching in the bipartite graph and colour all the edges in that matching with the same colour. Remove that matching from the graph and repeat this procedure until all edges have been coloured.

Example 8.8

Consider the following instance of $(O_3 \mid r_j, p_{ij} = 1 \mid L_{\max})$ with 3 machines and 7 jobs.

Jobs	1	2	3	4	5	6	7
r_j	0	1	2	2	3	4	5
d_j	5	5	5	6	6	8	8

Assume that $L = 1$. Each job has a deadline $\overline{d_j} = d_j + 1$, and we have $t_{\max} = 9$. In Phase 2, we have the network flow problem described below.

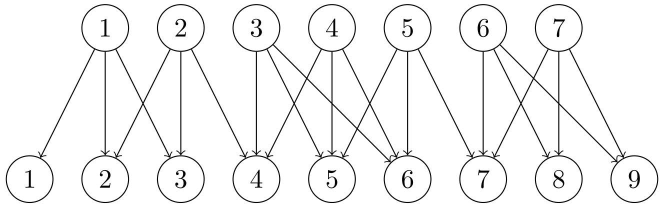


Solving this network flow problem, we see that the jobs can be processed during the time units in the following table.

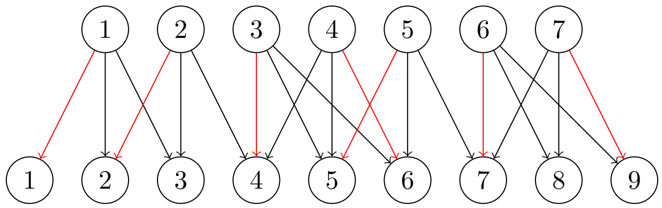
Jobs	1	2	3	4	5	6	7
Time Units	1, 2, 3	2, 3, 4	4, 5, 6	4, 5, 6	5, 6, 7	7, 8, 9	7, 8, 9

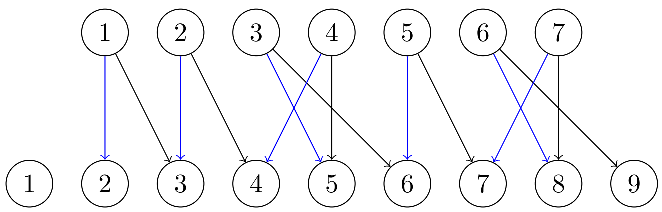
It is not hard to verify that at most three jobs are processed simultaneously for any given point in time.

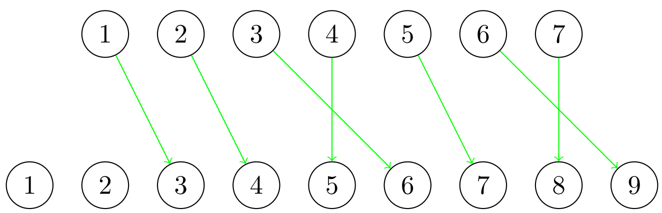
Phase 3 leads to the graph colouring problem for the following bipartite graph.



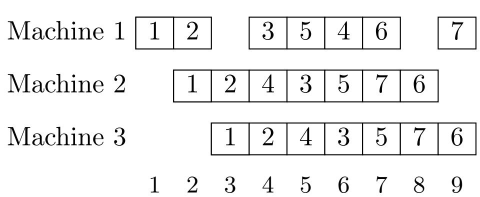
We find matchings and remove them until all edges are coloured.







Letting red denote machine 1, blue denote machine 2, and green denote machine 3, we are led to the following schedule which has $L_{\max} = 1$.



Note that we have not verified at this point whether or not there is a feasible schedule for $L = 0$. But by running this algorithm again with this parameter, we can find a certificate of infeasibility for the network flow problem. For example, one could determine the dual of the transportation model we discussed above and show that the dual is unbounded, which implies that the transportation model is infeasible. Thus, there does not exist a schedule in which every job is completed on time.