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 that 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 $C_{\max}$ among all feasible schedules.

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.

```
M1 |-S1-|--S2--|---S3---|-----S4------|----idle---|
   0    4      10       18            30          40

M2 |idle|----S1----|---S2---|-S3-|idle|--S4--|idle|
   0    4          14       23   26   30     36   40

M3 |-----idle------|----S1----|---S2---|--S3--|S4-|
   0               14         25       32     37  40
```

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 $(F3 \parallel C_{\max})$, 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 $\alpha = P2$.
- **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** ($\beta = prec$) 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** ($\beta = prmp$) 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 $C_j^S \leq C_j^{S'}$ 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 $C_j^S \leq C_j^{S'}$ 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 $\sum U_j^S \leq \sum U_j^{S'}$, 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 with 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. □

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 $k+1, \ldots, n, 1, \ldots, k$, 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

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

The ratio on the left-hand side is called the **ρ-factor** of the chain $1, \ldots, k$ and is denoted by $\rho(1, \ldots, k)$. 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 $1, \ldots, \ell^*$ 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 $1, \ldots, \ell^*$ is interrupted by a job, say v, from another chain. That is, the optimal sequence contains the subsequence $1, \ldots, u, v, u+1, \ldots, \ell^*$; call this subsequence S. It suffices to show that either with subsequence $v, u+1, \ldots, \ell^*$ which we denote S', or subsequence $1, \ldots, u, v$, 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 $1, \ldots, k$, 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 $1, \ldots, \ell^*$ 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 $x_1, \ldots, x_n \in \{0, 1\}$. 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

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

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

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

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 $L_{\max} = \max_{j \in \{1,\dots,n\}} L_j$, 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 $(1 \mid prec \mid h_{\max})$. 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 $h_{\max} = \max_{j \in \{1,\dots,n\}} h_j(C_j)$. 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 $\hat{J} \subseteq S^c = \{1, \dots, n\} \setminus S$. 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 $(1 \mid prec \mid h_{\max})$.

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 $(1 \mid prec \mid h_{\max})$.

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. □

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.

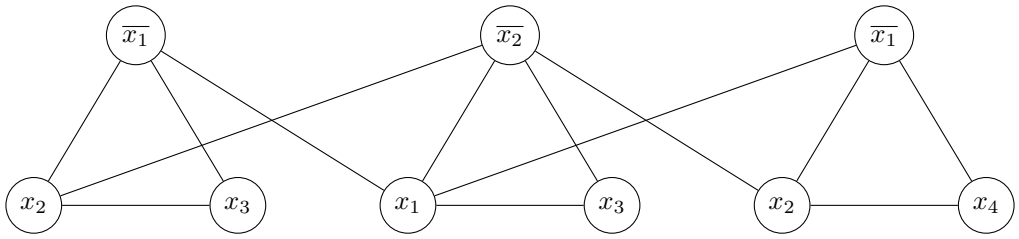
Theorem 3.2

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. □

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

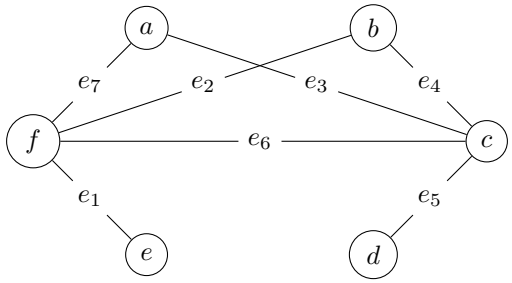
We have Vertex-Cover $\leq_P$ Set-Cover.

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

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.

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 PTAS for the Knapsack Problem

A **polynomial time approximation scheme (PTAS)** is a $(1+\varepsilon)$-approximation algorithm for any constant $\varepsilon > 0$. While a PTAS produces arbitrarily high quality solutions, the consequence is that it trades off accuracy for time. Here, we will give a PTAS for the KNAPSACK problem via rounding and scaling.

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.

First, we formulate the KNAPSACK problem as a decision problem and give a brief sketch 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 $\sum_{i \in S} v_i \geq V$?

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 one can show that

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