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.

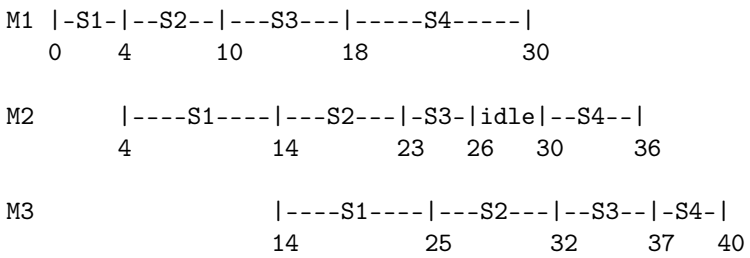
We denote the time required for machine M_i to process student S_j by p_{ij}.

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.



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 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.

- **Greed.** 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 in equation (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 call, so `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 $O(n \log n)$.